Université Sorbonne Paris Cité
École Doctorale 386
Sciences Mathématiques de Paris Centre

PEDECIBA — Facultad de Ingienería
Universidad de la República
Uruguay

Réalisabilité classique et effets de bord

THÈSE
en vue d'obtenir les grades de

DOCTEUR DE L'UNIVERSITÉ SORBONNE PARIS CITÉ
en Informatique Fondamentale

DOCTOR DE LA UNIVERSIDAD DE LA REPÚBLICA
en Matemática

Présentée et soutenue par

Étienne Miquey

le 17 Novembre 2017

devant le jury composé de:

Hugo Herbelin	Directeur de thèse	Directeur de Recherche, INRIA
Alexandre Miquel	Directeur de thèse	Professeur, Universidad de la República
Laurent Regnier	Rapporteur	Professeur, Université Aix-Marseille
Thomas Streicher	Rapporteur	Professeur, Technische Universität Darmstadt
Thomas Ehrhard	Président du Jury	Directeur de Recherche, CNRS
Walter Ferrer Santos	Examinateur	Professeur, Universidad de la República
Assia Mahboubi	Examinatrice	Chargée de recherche, INRIA
Colin Riba	Examinateur	Maître de conférence, ÉNS de Lyon

Résumé

Titre : Réalisabilité classique et effets de bords

Mots-clés : *contenu calculatoire de preuves classiques, effets de bord, axiomes du choix dépendant, évaluation paresseuse, types dépendants, réalisabilité classique, algèbres implicatives*

Cette thèse s'intéresse au contenu calculatoire des preuves classiques, et plus spécifiquement aux preuves avec effets de bord et à la réalisabilité classique de Krivine. Le manuscrit est divisé en trois parties, donc la première consiste en une introduction détaillée aux concepts utilisés par la suite.

La deuxième partie porte sur l'interprétation calculatoire de l'axiome du choix dépendant en logique classique, et en particulier au système dPA$^\omega$ d'Hugo Herbelin. Ce calcul fournit en effet, dans un cadre compatible avec la logique classique, un terme de preuve pour l'axiome du choix dépendant, qui peut être vu comme une adaptation de la preuve constructive de l'axiome du choix en théorie des types de Martin-Löf ou un internalisation dans un système de preuve de l'approche en réalisabilité de Berardi, Bezem et Coquand. L'objectif principal de cette partie est de démontrer la propriété de normalisation pour dPA$^\omega$, sur laquelle repose la cohérence du système. La difficulté d'une telle preuve est liée à la présence simultanée de types dépendants (pour la partie constructive du choix), d'opérateurs de contrôle (pour la logique classique), d'objets co-inductifs (pour ”encoder” les fonctions de type $\mathbb{N} \to A$ par des streams $(a_0, a_1, \dots)$) et l'évaluation paresseuse avec partage (pour ces objets co-inductifs). On montre dans un premier temps la normalisation du call-by-need classique (présenté comme une extension du $\lambda\mu\tilde{\mu}$-calcul avec des environnements partagé), en utilisant notamment des techniques de réalisabilité à la Krivine. On développe ensuite un calcul des séquents classique avec types dépendants, dont la correction est prouvée à l'aide d'une traduction CPS tenant compte des dépendances. En combinant les deux points précédents, on définit enfin une variante en calcul des séquents du système dont on peut finalement prouver la normalisation.

La dernière partie porte sur la structure algébrique des modèles induits par la réalisabilité classique. Ce travail se base sur une notion d'algèbres implicatives développée par Alexandre Miquel, une structure algébrique très simple généralisant à la fois les algèbres de Boole complètes et les algèbres de réalisabilité de Krivine, de manière à exprimer dans un même cadre la théorie du forcing (au sens de Cohen) et la théorie de la réalisabilité classique (au sens de Krivine). Le principal défaut de cette structure est qu'elle est très orientée vers le λ-calcul, et ne permet d'interpréter fidèlement que les langages en appel par nom. Pour remédier à cette situation, on introduit deux variantes des algèbres implicatives les algèbres disjonctives, centrées sur le “par” ⅋ de la logique linéaire (mais dans un cadre non linéaire) et naturellement adaptées aux langages en appel par nom, et les algèbres conjonctives, centrées sur le tenseur $\otimes$ de la logique linéaire et adaptées aux langages en appel par valeur. On prouve en particulier que les algèbres disjonctives ne sont que des cas particuliers d'algèbres implicatives et que l'on peut obtenir une algèbre conjonctive à partir d'une algèbre disjonctive (par renversement de l'ordre sous-jacent). De plus, on montre comment interpréter dans ces cadres les fragments du système L de Guillaume Munch-Maccagnoni en appel par valeur (dans les algèbres conjonctives) et en appel par nom (dans les algèbres disjonctives).

Abstract

Title: Classical realizability and side-effects

Keywords: *computational content of classical proof, side effects, dependent types, lazy evaluation, axiom of dependent choice, classical realizability, implicative algebras*

This thesis focuses on the computational content of classical proofs, and specifically on proofs with side-effects and Krivine classical realizability. The manuscript is divided in three parts, the first of which consists of a detailed introduction to the concepts used in the sequel.

The second part deals with the computational content of the axiom of dependent choice in classical logic. This works is in the continuity of dPA$^{\omega}$ system of Hugo Herbelin, which allows to adapt the constructive proof of the axiom of choice in Martin-Löf's type theory in order to turn it into a constructive proof of the axiom of dependent choice in a setting compatible with classical logic. The principal goal of this part is to prove the property of normalization for dPA$^{\omega}$, on which relies the consistency of the system. Such a proof is hard to obtain, due to the simultaneous presence of dependent types (for the constructive part of the choice), of control operators (for classical logic), of co-inductive objects (in order to "encode" functions of type $\mathbb{N} \to A$ as streams $(a_0, a_1, \dots)$) and of lazy evaluation with sharing (for this co-inductive objects). This difficulties are first studied separately. In particular, we show the normalization of classical call-by-need (presented as an extension of the $\lambda\mu\tilde{\mu}$-calculus with shared environments) by means of realizability techniques. Next, we develop a classical sequent calculus with dependent types, defined again as an adaptation of the $\lambda\mu\tilde{\mu}$-calcul whose soundness is proved thanks to a CPS-translation which takes the dependencies into account. Last, a sequent-calculus variant of dPA$^{\omega}$ is introduced, combining the two previous systems. Its normalization is finally proved using realizability techniques.

The last part dwells on the algebraic structure of the models induces by classical realizability. This work relies on the notion of implicative algebras developed by Alexandre Miquel, a very simple algebraic structure generalizing at the same time complete Boolean algebras and Krivine realizability algebras, in such a way that it allows to express in a same setting the theory of forcing (in the sense of Cohen) and the theory of classical realizability (in the sense of Krivine). The main default of these structures is that they are deeply oriented towards the λ-calculus, and that they only allows to faithfully interpret languages in call-by-name. To remediate the situation, we introduce two variants of implicative algebras: disjunctive algebras, centered on the "par" ($⅋$) connective of linear logic (but in a non-linear framework) and naturally adapted to languages in call-by-name; and conjunctives algebras, centered on the "tensor" ($\otimes$) connective of linear logic and adapted to languages in call-by-value. Amongst other things, we show that disjunctive algebras are particular cases of implicative algebras and that conjunctive algebras can be obtained from disjunctive algebras (by reversing the underlying order). Moreover, we show how to interpret in these frameworks the fragments of Guillaume Munch-Maccagnoni's system L for call-by-value (within conjunctive algebras) and for call-by-name (within disjunctive algebras).

Resumen

Título: Realizabilidad clásica y efectos de borde

Palabras claves: *contenido computacional de pruebas clásicas, efectos de bordes, tipos dependientes, evaluación perezosa, axioma de la elección dependiente, realizabilidad clásica, álgebras implicativas*

Esta tesis se enfoca en el contenido calculatorio de las pruebas clásicas, particularmente en las pruebas con efectos de borde y en la realizabilidad clásica de Krivine. El manuscrito está divido en tres partes, la primera constituyendo una introducción detallada a los conceptos y herramientas involucrados.

La secunda parte se concentra en el contenido calculatorio del axioma de elección dependiente en lógica clásica. Este trabajo se inscribe en la continuidad del sistema dPA$^\omega$ de Hugo Herbelin, que permite adaptar la prueba constructiva del axioma de elección en la teoría de tipos de Martin-Löf en una prueba constructiva del axioma de elección dependiente en un marco compatible con la lógica clásica. El objetivo principal de esta parte es la demostración de la propiedad de normalización para dPA$^\omega$, de la cual depende la coherencia del sistema. Recherche Décrire les activités de recherche en détaillant complètement au moins les trois dernières années effectives.

Sujet, lieu et nature de chaque activité Résultats obtenus résultats théoriques ou méthodologiques, synthèses, résultats d'expériences, mesures, évaluations, propositions de langages, d'architectures logicielles ou matérielles, développement de logiciels ou de matériels, dépôts de logiciels à l'APP. Présentation du projet de recherche personnel du candidat. Semejante prueba es difícil de conseguir, debido a la presencia simultanea de tipos dependientes (para la parte constructiva de la elección), de operadores de control (para la lógica clásica), de objetos coinductivos (para "codificar" una función del tipo $\mathbb{N} \to A$ mediante el flujo de sus valores $(a_0, a_1, \dots)$) y de evaluación perezosa (para esos objetos coinductivos). En una primera etapa, las dificultades están estudiada separadamente. En particular, demostramos la normalización del call-by-need clásico (presentado como una extensión del $\lambda\mu\tilde{\mu}$-cálculo con memoria compartida) usando técnicas de realizabilidad. Desarrollamos después un cálculo de los secuentes clásico con tipos dependientes, definido otra vez como una extensión del $\lambda\mu\tilde{\mu}$-cálculo, cuya corrección está demostrada por gracias a una traducción CPS que toma las dependencias en cuenta. Por último, introducimos una variante de dPA$^\omega$ en cálculo de los secuentes que combina los dos sistemas anteriores. Su normalización está finalmente demostrada usando técnicas de realizabilidad.

La última parte esta centrada en el estudio de las estructuras algébricas de los modelos inducidos por la realizabilidad clásica. Este trabajo está basado en la noción de álgebras implicativas de Alexandre Miquel, una estructura algébrica muy sencilla generalizando al mismo tiempo las álgebras completas de Boole y las álgebras de realizabilidad de Krivine, de tal forma que se puede expresar en un mismo marco la teoría del forcing (de Cohen) y la teoría de la realizabilidad clásica (de Krivine). El defecto principal de esas estructuras es que son profundamente orientadas hacia el λ-cálculo, y que solamente permiten una interpretación fiel de lenguajes en call-by-name. Para remediar a ese problema, introducimos dos variantes de las álgebras implicativas: las álgebras disjunctivas, centradas en el "par" ⅋ de la lógica linear (pero en un marco non-linear) y naturalmente adaptadas para lenguajes en call-by-name; y las álgebras conjunctivas, centradas en el tensor $\otimes$ de la lógica linear y adaptadas para lenguajes en call-by-value. Entre otras cosas, demostramos que las álgebras disjunctivas son casos particulares de las álgebras implicativas y que las álgebras conjunctivas pueden ser obtenidas por dualidad desde álgebras disjunctivas (invirtiendo el orden subyacente). Además, mostramos cómo interpretar en esos marcos los fragmentos del sistema L de Guillaume Munch-Maccagnoni's correspondiendo al call-by-value (en las álgebras conjunctivas) y al call-by-name (en las álgebras disjunctivas).

Remerciements & agradecimientos

Commençant par le commencement, je me dois de remercier avant tout et surtout Hugo et Alexandre que j'ai eu le plaisir d'avoir comme directeurs de thèse. Merci à l'un et à l'autre pour leurs encadrements pour les questions de recherche passionnantes qu'ils m'ont confiées et plus particulièrement pour la grande liberté qu'ils m'ont laissée pour les traiter. Merci à Alexandre pour cette fabuleuse idée que d'avoir émigré à Montevideo, micro-flux migratoire sans lequel il est fort probable que ces dernières années eurent été bien différentes. Un grand merci aussi pour les relectures en timing serré[1]. Merci à Hugo pour son enthousiasme sans cesse renouvelé pour mon travail, tant pour les multiples tentatives infructueuses que devant le résultat accompli. Cette thèse et moi-même devons beaucoup à de nombreuses heures passées ensemble devant des tableaux à en dégoûter plus d'un de toute forme de syntaxe. Merci à tous les deux pour toutes les remarques qui, en diverses occasions, m'ont permis de prendre du recul et voir un peu plus loin que le λ-terme au bout de mon nez.

Merci à Laurent et Thomas d'avoir accepté de rapporter ma thèse. Pour mon premier stage en L3, j'ai passé un certain temps sur un article du premier, et ai consacré un long moment à comprendre les travaux du second ces dernières années, c'est donc un véritable honneur de voir les rôles ainsi inversés. Merci aussi à Assia, Colin, Thomas et Walter, que je suis très heureux de compter dans mon jury.

Faire une thèse à PPS aura été un réel plaisir, tant pour la bienveillance générale qui y règne que pour le contexte scientifique privilégié : c'est un confort sans pareil de savoir que derrière les portes des bureaux avoisinants se trouvent la réponse à bien plus de questions que l'on ne pourrait avoir. Et puis, force est de constater que la foule de thésards qui peuple le troisième étage, étrange échantillon de population encore méconnu du grand public, contribue à sa façon à faire de ce lieu un cadre de travail fort agréable.

Tout d'abord, un immense merci à Jovana d'avoir été la meilleure cobureau imaginable[2]. Sa présence sereine, son addiction aux noix, sa force de propositions houblonnées et son intarissable patience pour stabiliser mon anglais vacillant ont sans nul doute été pour beaucoup dans mon entrain quotidien à venir au labo. Bienheureux seront ceux et celles qui auront cette chance dans le futur.

Sans changer de bureau, un grand merci à Ioana, Kuba et Shahin, qui malgré une criante absence de connaissance de Winnie l'ourson, m'ont accueilli parmi eux. Merci à Ioana de m'avoir assigné ce bureau là, à Kuba de m'avoir aidé à mieux comprendre mon emploi du temps et à Shahin de tous ses efforts pour ne pas se conformer au politiquement correct. Merci, plus récemment, à Tommaso, Nicolas[4], Adrien et Victor, mystérieux cobureaux apparus sans crier gare alors que j'étais en Uruguay.

Merci à Cyrille, qui compense honorablement son terrible manque de culture par un certain discernement footballistique[5]. Merci à Amina qui, en plus d'avoir été un modèle d'efficacité, nourrit une passion pour les points fixes à laquelle un petit bout de cette thèse est bien redevable. Merci à Pierre, compagnon de la dernière ligne droite, et grande source d'inspiration dans la composition artistique de

[1] Et mes plus plates excuses pour les nombreuses embûches à base de *s* et autres *two last* au fil des pages.

[2] Et la baby-sittrice attentionnée[3] d'une petite plante devenue grande à ses côtés.

[3] Même si traditionnellement à la fin d'un baby-sitting, on a plutôt tendance à rendre le bébé ;-)

[4] Un merci particulier de partager ce goût pour la résolution de problèmes informatiques pas franchement utiles, Bash, LaTeX et les palindromes en tête.

[5] Et possède en outre une grande capacité à reconnaître une chèvre quand il en voit une.

la figure page 48. Merci à tous ceux et celles à qui j'ai pu poser un certain nombre de questions, en particulier merci à Théo et aux différents membres du bureau Coq-compétent, à Léonard le catégoricien et à Hadrien, Atlas de la réalisabilité classique qui en porta le séminaire des semaines durant sur ses seules épaules. Merci à Charlotte, Colin et Mattieu, dernier.e.s survivant.e.s au plus profond du mois d'Août. Merci à Marie, la plus exilée des thésardes. Merci à PIM, entre autre, pour ses `frametitle`, à Thibaut pour ses tentatives répétées de me convertir au terrorisme. Un incommensurable abrazo à Lourdes[6], dont la gentillesse et l'attention quasi-maternelle pour nos autres thésards resteront à jamais sans égal. Merci enfin à tous les contributeurs de feu-le gâteau thésards, ce qui devrait inclure les oublié.e.s des lignes précédentes.

Merci aussi à Bruno et Florent, plus connus sous les noms de MC Guillon les Papillons et MC Herbert, vieux camarades d'amphis de jadis que j'ai eu la joie de retrouver comme voisins du dessus en début de thèse. Nul doute que la carrière musicale du SG Boyz band finira par décoller.

Parmi les plus du tout thésards, merci aux différents collègues de l'IRIF avec qui enseigner aura été un grand plaisir et un riche apprentissage : je pense notamment à Dominique, Michele, Ralf et Yann. Merci à Alexis, pour sa bienveillance permanente et tout le soin accordé aux thésards. Merci à Pierre-Louis, formidable directeur d'équipe s'il en est, et à Jean, convaincant exemple que l'on peut vivre avec bonheur la contradiction entre la noble passion pour le ballon rond et de bien plus inavouables convictions idéologiques. Merci aussi à Odile et Lindsay, dont le travail et la disponibilité m'auront précieusement simplifié la vie.

Repoussant les limites géographiques de ces remerciements, merci aux amateurs de Chocola qu'il est toujours un plaisir de retrouver en amphi B. Merci à Lionel pour le concept de séminaire-calanques. Merci à Daniel *'les-fonctions-sont-des-valeurs'* Hirschkoff, sans le cours duquel l'extradiégèse serait tristement restée étrangère à ma conception de l'informatique[7].

Sorti du monde académique, j'aimerais remercier en premier lieu le F. et le B., co-disciples d'une philosophie de vie reposant essentiellement sur un don de soi inconditionnel et des véhicules à plus ou moins de roues[8]. C'est un honneur que de vivre au quotidien la ride par mails avec de si belles personnes. Mention spéciale au B. pour de mémorables aventures pré-estivales, hautes en couleurs et en liquides bienfaisant.

Merci aux footeux/ses des samedi matin, Helmy, Bruno P., Ahmed, Lulú, Kévin, Guillaume, Jamel, Mustapha, Camille et tout ceux que j'oublie ici. Un merci tout particulier à Bruno G., qui non content d'avoir considérablement fait grossir les troupes, fit survivre cette belle liste mail quelques mois de plus. Merci aussi à la JUMP et à tous ceux et celles qui contribuent à en faire un club aussi chaleureux, et tout particulièrement à Ali, Julien et Vincent pour toute l'énergie investie en ce sens.

Un grand merci à la team *"¡Uruguay nomá!"* d'être venue me rendre visite pour un Noël fait de viande et de soleil. Moustache gracias à PAF (et par ailleurs pour tout ce que mon vélo lui doit), à Théo, Noémie, Nadège, Marjo, et à leurs précurseurs Lucie et Matthieu. Merci à Marjolaine *'jmm'* Lacombe pour ses nombreuses visites allée Darius Milhaud, pour toutes les randos, le kéfir, les aventures de John et Cindy, l'anglais et tout le reste.

Une grande dédicace aux émérites brondillants que sont Sami et Wissem. De Miribel à Lorient, de l'AS Bron à Oujda en passant par l'aiguille du Cédéra, je mesure pleinement la chance d'avoir des amis prêts à m'emmener (ou pire, me suivre) dans une foultitudes d'aventures[9]. Par proximité géographique, merci à Aymen de m'avoir offert un acolyte aux Essarts[10] à l'agenda post-bac incroyablement synchronisé.

[6] Y muchas gracías por la ayuda con el castellano de mis agradecimientos.

[7] Et, plus accessoirement, mon engouement pour le λ-calcul à la sauce Curry-Howard peut-être mort dans l'œuf.

[8] Pour une notice plus complète : `https://www.dailymotion.com/video/x2coryk`.

[9] Dont je tiens à préciser ici qu'elles ne finissent pas toujours toutes en plan galère.

[10] Sisi la famille !

Merci à Yannis et Coco de m'avoir permis de venir à la rencontre des sources du comté. D'avance merci à Margaux pour la formidable personne qu'elle va devenir. J'en profite pour saluer Jimou, et graver dans le marbre que ce n'était pas moi dans le jacuzzi. Merci à Anaëlle, Gaël, à Thibault, à Thanina, dont la présence parisienne à joyeusement agrémenté mon quotidien à de multiples et ludiques reprises. Merci à Clément et Tiziana, à la duchesse de Charenton et à son Franky de mari, et aux divers.es lyonnais.es fidèles au poste. Merci à la châtelaine de Viella pour le vin et la découverte de Niš.

Merci infiniment à Manue et Xavier ainsi qu'à mes grands-parents, dont les patrimoines immobiliers respectifs ont grandement contribué à rendre ces années plus douces ! Merci aussi à Gaëlle l'apprenti rideuse et Anna la moustachious d'avoir été des colocatrices de luxe. J'attends avec impatience la future maison en terre de l'une et la prochaine fête de mariage de l'autre.

Merci à ma famille, en particulier merci à mes parents de s'être prêté au jeu de la relecture de mon premier chapitre, merci à Marie et Nadège d'avoir pris tour à tour grand soin de mon alimentation le temps d'un été, et à Baptiste de toujours avoir une place de match sous le coude.

Merci aux différents auteur.e.s anonymes et démasqué.e.s d'un fantastique document secret. Soyez assuré.e.s que tous les moyens seront mis en œuvre pour produire une réponse à la hauteur de la production originale. Il y sera question, en néo-zélandais ou en bermudois, de Michelangelo et de ses problèmes de choix d'une cravate à canard, ainsi que de la dualité entre les crocodiles Haribo et les petits pruneaux. Spoiler alert, le druide de Maurice Thorez interviendra au moyen d'une galipette taco-récursive afin de forcer l'alignement des planètes et le pluralisme moral de l'appartement en restauration. La valeur vraie du coaching de B.G. sera alors déterminée grâce au manifeste oulipien, sorte d'α et d'Ω y compris pour les profanes.

Merci, enfin, à mon vélo, fidèle compagnon comme on n'en fait plus.

Yo no soy de por aquí, no es este pago mi pago...[11] Si bien no nací oriental, tuve el inmenso placer de pasar dos años fabulosos en Uruguay, y soy muy orgulloso de poder decir que de alguna forma hice, más allá de aquellas cosas matemáticas con las cuales cuenta este manuscrito, un verdadero *doctora-bó*[12]. Ese aprendizaje no tiene precio y se lo debo a mucha gente, que me gustaría agradecer también en esas páginas. De alguna manera, todos y todas acompañaron esta tesis, notablemente brindándome muchas perspectivas agradables por fuera del mundo académico, momentos muy valiosos a la hora de volver a enfrentarse con el pizarrón.

Primero que nada, tengo que agradecerle a Mauricio, sin quien nunca hubiese llegado a este maravilloso país[13]. Más allá de ser el director de mi pasantía de M1, me pasó los primeros piques que tuve acerca de Montevideo. Es más, me enseño a cebar maté, con eso digo todo. Por eso, por haber estado siempre disponible para facilitarme la vida con los trámites, por haber sido un inagotable proveedor en frutas en verano y por todo el resto, muchas gracias.

Agradezco también a toda la gente del IMERL y de la UDELAR con la cual pude compartir lindos momentos. En particular, muchas gracias a Walter y Octavio por hacer que el equipo de Lógica no se resuma a un núcleo de migrantes de PPS. Gracias a Matilde, Javier y Gabriel por acogerme en su oficina. Un abrazo especial a Maryorí, que tuve el placer de cruzar varias veces en circunstancias sumamente recomendables (y agradablemente más festivas que en el pasillo del IMERL).

Es un honor para mi poder agradecer en estas humildes líneas a Luis Suárez. Me refiero al Luis Suárez posta, el auténtico[14], quién además de haber sido un gran compañero milonguero, candombero y de cuantas vueltas en chiva, fue el chef el más eficaz que nunca haya tenido en casa (cómo se movió aquella cocina, ¡che!). Podría parecer anecdótico, pero también me inició a radio babel, que me acompañó con gusto esos tres últimos años. Cómo si esto fuera poco, encima me abrió la puerta de su casa y de

[11] No tengo ninguna duda que los lectores de esa sección sabrán reconocer el añorado autor de esa bellísima canción.

[12] Agradecemos a James Bó quien fue sin lugar a dudas la fuente de inspiración de ese chiste lamentable.

[13] *¿Mejor que Francia y mejor que París?*

[14] Es fácil distinguirlo del trucho, ya que el verdadero no muerde.

Treinta y Tres[15]. Eso fue y quedará ¡impagable!

Muchísimas gracias a toda la familia por adoptarme como uno de los suyos. Gracias a Mirta la viajera, al incansable Negro, a Gota mi doble vestido de rojo, a Marcos la mano verde, a Fabián el ciclista, a Ele y al peludo Octavio[16]. Un tremendo abrazo a Marcia, amorosa compañera duraznera, y a Matías, quien supo trascender la noción de desafío[17], de los dos recibí más cariño[18] de lo que nunca podré agradecerles. Sé que realmente cuento con una familia extraordinaria en muchos aspectos, y no daría ni un capítulo de esta tesis para listar todo por lo que les estoy agradecido, así que ahí está lo esencial: los quiero un montón.

Ya que estamos, muchas gracias a toda la barra olimareña y a su extensión valizera. Un abrazo especial a Mariana y su gentileza sin límites, a Bruno y Diego, altas fichas del paisaje local, y al Master, flor de oponente a la rigidez idiomática[19]. Un saludo cortés a su alteza la Ministra, estoy muy honrado de conocerla.

Siguiendo con los grandes, los que son irreemplazables, estoy re-contra-agradecido con el señor Antonio. Además de haber sido un vecino fantástico, un compañerazo callejero y un asador de novela, dedicó tiempo y sudor a que yo supiera hacer algo de mis diez dedos[20]. Algún día se comprometió con mi jefa que me iba a curtir, no hay nada más que decir que lo hizo con mucha nobleza. Muchísimas gracias por todo, usté sabe cuánto me gustaría que el océano se achique cada tanto... ¡Ojalá!

Como ya lo habrán entendido, vivir en Montevideo habrá sido un gran placer, por muchas razones. En particular, dentro de la categoría *"pilares de la cultura yorugua que ritmaron mi vida"* están nominados: tango, fútbol y candombe[21]. Empezando con el candombe, mando un abrazo grupal a todos los gurises de la Kasita, con los cuales gocé domingo tras domingo, *chikalakun kalakunchikalakuncha...* Son demasiados los que tendría que nombrar aquí, pero agradezco en particular a Vani, Ale, la Lulú, Marcel y el Pollo, barra brava de la banda y fieles compañer@s de la escala posterior en la Vidalita[22]. Abrazo también al Pablo, iniciador de cantarolas cómo pocos. Muchas gracias también a Chupete, a quien debo mucho, y a Tres Pelos por darle a luz a una hermosa cuerda.

Hablando de fútbol, no hay mucho más para decir que tuve la suerte de contar con una barra espectacular, tanto en su aplicación en mantener el partido de los miércoles, que en prolongar el tercer tiempo correspondiente. Muchas gracias, Leo, Leito, Edu *'supermatch'* Aguirre, Manuel, Luisma, Mintxo, Rodolfo, Martín, Pablo, Agustín, Pablo, Andrés y todos los demás postulantes al Gordón de ayer y al Morgan de hoy. Fue un placer compartir tremendos partidos y Chulo's nights con gente de caché literario.

Cerrando esta trinidad con el tango, tengo que darle un agradecimiento especial a Gabriela, quién no solamente fue la más mejor de todas las profes, sino que se convirtió en una presencia imprescindible en mis semanas. Muchísimas gracias por todo lo que me diste, por tanta paciencia y tanta dulzura. Gracias también a tod@s con l@s que compartí chelas o abrazos en alguna de esas noches montevideanas. En particular, gracias a Cecilia "milonga" Laffite, a Victoria la doctora y Andrea la expatriada.

Muchas gracias a toda la familia Vincent-Erro, gracias a quienes pude descubrir un buen trozo del trayecto del 468. Gracias pues a Jérôme por todas las invitaciones, a Tania, Blanca, Laura y Sara-Melina[23]. Muchísimas gracias a Marité, vecina fenomenal, por su increíble energía, por aguantar todas mis bromas sobre su acento belga y por su gentileza sin igual. Gracias a Gabriel y Désirée, por el alojamiento, los domingos con sorrentinos y tucos, los partidos de ajedrez y por hacer que yo me sintiera

[15] Y sobre todo, me hizo beber agua del Olimar para que a el volviera. Aparentemente funcionó.

[16] El mejor intérprete que *Tumba la casa* nunca tendrá.

[17] E inventar nuevas formas de perder al ping-pong frente a una amplia selección de paletas artesanales :-)

[18] Y una exigencia constante hacia mi pronunciación, por la cual también les agradezco mucho.

[19] *¿Alguna vez vieron a la leche de higo florecer?*

[20] Aprovecho para agradecer sus secuaces del taller, quienes siempre fueron vigilantes a que mis dedos sigan siendo diez.

[21] Suena tan uruguayo que hasta podría ser el título de un disco de Jaime Roos.

[22] Que, dicho sea al paso, es de lejos la mejor pizzería que se ubique en Pablo de María y Guaná.

[23] Creo que tenemos el montaje de un camión pendiente, ¡ya lo resolveremos!

cómo en casa en Vissi d'Arte. Un abrazón a María, por su constante cariño, por su socarronería de alto nivel, por la pistola de agua que me salvó la vida frente a un felino feroz y más que nada por *pomme de reinette et pomme d'api*. Un abrazo de gol a la Vero, amigaza multi-ámbitos y rostruda sin vergüenza al truco. Gracias por todas aquellas vueltas y expediciones que compartimos, al tablado, las llamadas, Isla patrulla, aquella noche llena de noctilucas, ¡y me quedo corto! Gracias por siempre estar.

Para concluir esos agradecimientos, quiero dedicar ese último parágrafo a Joaquín, una persona increíble, famoso integrante de la no menos increíble banda[24] Los Muleki, y sobre todo un gran compañero de primera hora. Pasaron unos años y cuántas cosas desde nuestro encuentro aquella noche en el balcón de la Casa Camaleón, pero queda la certeza que tuvo mucho que ver con mis ganas de volver al paisito después de mi primera pasantía. ¿Quién sabe si, sin ese encuentro, yo hubiera vuelto y hecho esta tesis? Lo más seguro que el interesado me contestaría sonriendo: *c'est la vie !*

Merci enfin, à Mathilde qui, en grande stratège, consomma elle-même les blagues les plus fines[25] que je ne pourrais donc point faire ici. Devant donc me résigner à des lignes plus solennelles, merci d'être venue à Montevideo, merci d'y être revenue[26]. Merci surtout, pour tout un tas de choses, en filigrane au long de ces remerciements et durant ces années dans le tourbillon de la vie[27], que je te dois aussi, un peu, beaucoup, à la folie.

[24] Probablemente la única en el mundo que toque un medley incluyendo Francky Vincent y Chico Trujillo, con eso digo todo.

[25] *"$\lambda\mu\tilde{\mu}$, et pourquoi pas $\lambda - \mathtt{ma} - \widetilde{\mathtt{ma}}$?"*, *"C'est toi le type dépendant !"*, etc.

[26] Comme annoncé par le grand Jacques.

[27] Quand on y pense, *aïe, aïe, aïe, ça fait déjà un fameux bail...*

Contents

Introduction

"The truth, the whole truth, and nothing but the truth." This famous oath could have constituted, back in the 17th century, Leibniz's profession of faith in seek of his *calculus ratiocinator.* Indeed he envisioned that every philosophical dispute may be settled by a calculation [107][28]:

> *"The only way to rectify our reasonings is to make them as tangible as those of the Mathematicians, so that we can find our error at a glance, and when there are disputes among persons, we can simply say: Let us calculate [calculemus], without further ado, to see who is right. [...] if controversies were to arise, there would be no more need of disputation between two philosophers than between two calculators. For it would suffice for them to take their pencils in their hands and to sit down at the abacus, and say to each other [...]: Let us calculate"*

While there are reasonable doubts about whether Leibniz intended for the so-called *calculus ratiocinator*, the system or device used to perform these logical deductions, to be an actual machine[29] or simply an abstract calculus, it is certain that he hoped to reduce all human reasonings to computation.

Alas, an obstacle—and not the least— was standing on his way: at the time, reasoning was taking the form of informal text, even in mathematics. Leibniz was then about to initiate a long path towards the formalization of mathematics. As a first step, he proposed the concept of *characteristica universalis* which was meant to embody every human concept. Leibniz indeed had a combinatorial view of human ideas, thinking that they *"can be resolved into a few as their primitives"* [106, p. 205]. This idealistic language should thereby assign a character to each primitive concept, from which we could form characters for derivative concepts by means of combinations of the symbols: *"it would be possible to find correct definitions and values and, hence, also the properties which are demonstrably implied in the definitions"* [106, p. 205]. Leibniz thus intended for the *characteristica universalis* to be a universal language, which was to be employed in the computation of the calculus ratiocinator. If, at the end of the story, this dream turned out to be a chimera, we should acknowledge that his set idea of relating logic to computation was brightly visionary. Due do this connection, we can trumpet that this thesis is part of a tradition of logic initiated by Leibniz himself. To find our way back from the present dissertation to the *calculus ratiocinator*, let us identify a few milestones[30] along the path.

It actually took two centuries until a major step was made in direction of a formalization of mathematics. In the meantime, the scientific community had to handle an episode which shook the very foundations of mathematics: the discovery of non-Euclidean geometries. Two millenia earlier, Euclid gave in his *Elements* the first axiomatic presentation of geometry. He placed at the head of his treatise a collection of definitions (*e.g.* *"a line is a length without breadth"*), common notions (*e.g.* *"things equal to*

[28] For the reader looking for a good old exercise of Latin, here comes the original quote [106, p. 200]: *Quo facto, quando orientur controversiae, non magis disputatione opus erit inter duos philosophus, quam inter duos computistas. Sufficiet enim calamos in manus sumere sedereque ad abacos, et sibi mutuo (accito si placet amico) dicere: calculemus.*

[29] Leibniz was one of the pioneers of mechanical calculator with his Stepped Reckoner, the first machine with all four arithmetic abilities.

[30] Amusingly, *calculus* precisely means *stone* in Latin. Despite serious *scrupula*, we could not refrain ourselves from annoying the reader with this insignificant observation.

the same thing are also equal to one another") and five postulates (*e.g. "to draw a straight-line from any point to any point"*). Amongst these postulates, the fifth, also called the *parallel postulate*, has literally retained mathematicians' attention for a thousand years:

> *If a straight line crossing two straight lines makes the interior angles on the same side less than two right angles, the two straight lines, if extended indefinitely, meet on that side on which are the angles less than the two right angles.*

Because of its surprising prolixity with respect to the first four postulates, numerous attempts were made with the aim of deducing the parallel postulate from the first four, none of them showed to be successful. In the 1820s, Nikolai Lobachevsky and János Bolyai independently tackled the problem in a radically new way. Instead of trying to obtain a proof of the parallel postulate, Bolyai considered a theory relying only on the first four postulates, which he called "absolute geometry" [19], leaving the door open to a further specification of the parallel postulate or its negation. In turn, Lobatchevsky built on the negation of the parallel postulate a different geometry that he called "imaginary" [110]. Interestingly, to justify the consistency of his system, Lobachevsky argued that any contradiction arising in his geometry would inevitably be matched by a contradiction in Euclidean geometry. This appears to be the earliest attempt of a proof of relative consistency[31]. A few years later, Bernhard Riemman published a dissertation in which he also constructed a geometry without the parallel postulate [145]. For the first time, some mathematical theories were neither relying on synthetic *a priori* judgments nor on empirical observations, and yet, they were consistent in appearance. Theses new geometries, by denying traditional geometry its best claim to certainty, posed to the community of mathematicians a novel challenge: *How can it be determined for sure that a theory is not contradictory?* If Leibniz was our first milestone on the way, we would like the second one to mark this question.

For years, non-Euclideans geometries have been the target of virulent criticism, the colorful language of which the decency forbids us from transcribing here. One of the strongest opponent to these geometries was Gottlob Frege, who notably wrote: *"No man can serve two masters. One cannot serve the truth and the untruth. If Euclidean geometry is true, then non-Euclidean geometry is false."* [49]. Frege was thus in line with the ground postulate of Leibniz's calculus ratiocinator that the truth of any statement can be decided. In this perspective, Frege accomplished a huge step for the formalization of mathematics. In 1879, he introduced his *Begriffsschift* [48], a formal language to express formulas and proofs. Frege aimed at expressing abstract logic by written signs in a more precise and clear manner than it would be possible by words (which is not without recalling Leibniz's intentions with the characteristica universalis). Especially, Frege was responsible for the introduction of the quantifiers ∀—"*for all*"—and ∃—"*there exists*"—and most importantly of a proof system based on axioms and inference rules. Thereby, he paved the way for a syntactic study of proofs, emphasizing the *provability* of formulas.

On the other hand, the earlier work of Boole [20] did not lead to a language peculiar to logical considerations, but rather to the application of the laws (and symbols) of algebra[32] to the realm of logic. In particular, Boole's approach consists in assigning a truth value to each proposition, pointing out the semantic notion of *validity* of formulas.

Despite Boole and Frege advances, when the 20th century began, the existence of *calculus ratiocinator* was still a plausible expectation in light of the state of the art in logic. Even without matching

[31] Actually, there is an earlier trace of such a proof in Thomas Reid's work [142]. He defined a non-Euclidean geometry, his so-called *"geometry of visibles"*, that he described as being the one perceived by the *Idomenians*, some imaginary beings deprived of the notion of thickness. Reid claims that the *"visible"* space can be represented by an arbitrary sphere encompassing the space. This can also be considered as a relative consistency proof, asserting that the geometry of visibles is consistent if spherical geometry is. A detailed discussion on Reid's geometry can be found in [36].

[32] According to Boole, *"the operations of Language, as an instrument of reasoning, may be conducted by a system of signs composed of [...] literal symbols $x, y, \ldots$ [...] signs of operation, as $=$, $-$, $\times$ [...] the sign of identity $=$. And these symbols of Logic are in their use subject to definite laws, partly agreeing with and partly differing from the laws of the corresponding symbols in the science of Algebra"* [20, Chapter II].

Leibniz's ambition of deciding the validity of any philosophical statement, the problem of deciding the truth merely within mathematics was still an open question. In 1900, Hilbert drew up a list of twenty-three problems—another milestone along our travel time—the second of which was to prove the compatibility of the arithmetical axioms, *"that is, that a finite number of logical steps based upon them can never lead to contradictory results"* [73]. Rooted in this question, Hilbert established in the 1920s a program aiming at a formalization of all mathematics in axiomatic form, together with a proof that this axiomatization is consistent. Hilbert's manifesto for a quest of foundations climaxed with the slogan *"No ignorabimus"* during a radio broadcast in 1930[33] [74]:

> *"For us mathematicians, there is no 'ignorabimus', and, in my opinion, there is none whatsoever for the natural sciences. In place of this foolish 'ignorabimus' let our watchword on the contrary be: We must know — we shall know!"*

In continuation of his program, Hilbert raised with Ackermann another fundamental question in 1928, which is known as the *Entscheidungsproblem* [75]: to decide if a formula of first-order logic is a tautology. By "to decide" is meant via an algorithm, by means of a procedure. The signification of "algorithm" should be taken in context: the very concept of computer was yet unknown, an algorithm was thus to be understood as a methodical way of solving a problem, as a computational recipe. By putting the computation at the heart of the problem, the *Entscheidungsproblem* enters directly into the heritage of Leibniz quest for a calculus ratiocinator.

Unfortunately, Hilbert's fine aspirations were quickly shattered. First by Gödel [61], who proved in 1931 that any consistent logical system, provided that it is expressive enough, featured a formula which is not provable in this system, nor is its negation. Worst, he showed in particular that the consistency of arithmetic could not be proved within arithmetic, giving then a definitive and negative answer to Hilbert's second problem. As for the *Entscheidungsproblem*, Church [25, 26] and Turing [154, 155] independently proved that no algorithm could ever decide the validity of first-order formulas. Both answers relied on a specific definition of the notion of computability, captured in one case by *Turing machines*, by the *λ-calculus* in the case of Church. Church and Turing proved that both formalisms were equivalent, laying the ground of a unified definition of what are the *"computable"* functions. In other words, the concept of computer was born.

Leaving aside a few decades and some noteworthy discoveries, the second to last milestone on our journey, arguably the most important one concerning this thesis, is due to Curry [33, 34] and Howard [77], in 1934 and 1969 respectively. Independently, they both observed that the proofs of a constructive subset of mathematics, called intuitionistic logic, coincide exactly with a typed subset of the λ-calculus. This observation had a particularly significant consequence: by asserting that (intuitionistic) *proofs* were nothing less than *programs*, it put the computation at the center of modern proof theory. Furthermore, it brought kind of a small revolution by giving the possibility of designing altogether a proof system and a programming language, bug-free by essence.

While the proofs-as-programs correspondence seemed for a time to be bounded to intuitionistic logic and purely functional programming language, Griffin discovered in 1990 that Scheme's control operator `call/cc` could be typed by a non-constructive principle named the law of Peirce [62]. Several calculi were born from this somewhat accidental breakthrough, allowing for a direct computational interpretation of classical logic. Especially, Krivine developed the theory of *classical realizability* based on an extension of the λ-calculus with `call/cc`, in which he tried to obtain programs for well-known axioms. In so doing, he adopted a conquerent state-of-mind, proposing to push further the limits of Curry-Howard correspondence by programming new proofs.

[33] In case some readers would not have found satisfaction with the former Latin exercise, here his the original German declaration: *"Für uns gibt es kein Ignorabimus, und, meiner Meinung nach, auch für die Naturwissenschaft überhaupt nicht. Statt des törichten Ignorabimus, heiße im Gegenteil unsere Losung: Wir müssen wissen — wir werden wissen!"*.

Yet, it would be unfair to reduce classical realizability, our last milestone, to its sole contribution to proof theory. To highlight its particular significance, allow us a slight digression back to the early 1900s. Indeed, we eluded in our presentation the fact that mathematics were affected by the so-called *foundational crisis*. To cut a long story short, Frege axiomatized in his *Begriffsschift* [48] a *set theory* built on Cantor's earlier ideas. This theory was intended to lay a foundational ground to the definition of all mathematics, but a few years later a paradox was discovered by Russell, proving the theory to be inconsistent. If the axiomatization of set theory was finally corrected by Zermelo and Fraenkel, further to this episode, the question of proving the consistency of a given axiomatization has been a central issue for logicians of the 20th century. Two axioms were particularly controversial, namely the *axiom of choice* and the *continuum hypothesis*. Relying on Boole's notion of validity, Gödel first proved in 1938 that both were consistent with Zermelo-Fraenkel set theory [66]. Cohen finally proved that these axioms were independent from set theory, by showing that their negations were also consistent with set theory. To this end, he developed the technique of *forcing* to construct specific models in which these axioms are not valid.

At the edge of the last decade, Krivine showed in an impressive series of papers [98, 99, 100, 101] that classical realizability also furnishes a surprising technique of model construction for classical theories. In particular, he proved that classical realizability subsumes forcing models, and even more, gives raise to unexpected models of set theories. Insofar as it opens the way for new perspectives in proof theory and in model theory, we can safely state that classical realizability plays an important role in the (modern) proofs-as-programs correspondence.

This thesis is in line with both facets of classical realizability. On the one hand, from the point of view of syntax and *provability*, we continue here a work started by Herbelin in 2012 [70] which provides a proof-as-program interpretation of classical arithmetic with dependent choice. Half of this thesis is devoted to proving the correctness of Herbelin's calculus, called dPA^{ω}, which takes advantage of several extensions of the proofs-as-programs correspondence to interpret the axiom of dependent choice. We rephrase here Herbelin's approach in a slightly different calculus, dLPA^{ω}, of which we analyze the different computational features separately. We finally prove the soundness of dLPA^{ω}, which allows us to affirm:

> *Constructive proofs of the axioms of countable and dependent choices can be obtained in classical logic by reifying the choice functions into the stream of their values.*

On the other hand, from the viewpoint of semantics and *validity*, we pursue the algebraic analysis of the models induced by classical realizability, which was first undertaken by Streicher [151], Ferrer, Guillermo, Malherbe [44, 45, 43], and Frey [50, 51]. More recently Miquel [121] proposed to lay the algebraic foundation of classical realizability within new structures which he called *implicative algebras*. These structures are a generalization of Boolean algebras (the common ground of model theory) based, as the name suggests, on an internal law representing the implication. Notably, implicative algebras allow for the interpretation of both programs (*i.e.* proofs) and their types (*i.e.* formulas) in the same structure. In this thesis, we deal with two similar notions: *disjunctive algebras*, which rely on internal laws for the negation and the disjunction, and *conjunctive algebras*, centered on the negation and the conjunction. We show how these structures underly specific models induced by classical realizability, and how they relate to Miquel's implicative algebras. In particular, if this part of the thesis were to be reduced to a take-away message, we would like this message to be:

> *The algebraic analysis of the models that classical realizability induces can be done within simple structures, amongst which implicative algebras define the more general framework.*

The main contributions of this thesis can be stated as follows.

1. A realizability interpretation *à la* Krivine of the $\overline{\lambda}_{[lv\tau\star]}$-calculus [4], which is a call-by-need calculus with control and explicit stores. This interpretation provides us with a proof of normalization for this calculus. In addition, it leads us toward a typed continuation-and-store-passing style translation, which relies on the untyped translation given in [4]. We relate the store-passing style translation with Kripke forcing translations.
2. A classical sequent calculus with dependent types, which we call dL. While dependent types are known to misbehave in presence of classical logic, we soundly combine both by means of a syntactic restriction for dependent types. We show how the sequent calculus presentation brings additional difficulties, which we solve by making use of delimited continuations. In particular, we define a typed continuation-passing style translation carrying the dependencies.
3. A proofs-as-programs interpretation of classical arithmetic with dependent choice, which we call dLPA$^{\omega}$. Our calculus is an adaptation of Herbelin's dPA$^{\omega}$ system, given in a sequent calculus presentation. Drawing on the techniques previously developed for the $\overline{\lambda}_{[lv\tau\star]}$-calculus and dL, we defined a realizability interpretation of dLPA$^{\omega}$. This implies in particular the soundness and the normalization of dLPA$^{\omega}$, properties which were not proved yet for dPA$^{\omega}$.
4. A Coq formalization of Miquel's implicative algebras [121]. Since implicative algebras aim, on a long-term perspective, at providing a foundational ground for the algebraic analysis of realizability models, I believe that having a Coq development supporting the theory is indeed an appreciable feature.
5. The definition and the study of disjunctive algebras. We show how these structures, which are similar to implicative structures, naturally arise from realizability models based on the decomposition of the implication $A \rightarrow B$ as $\neg A \vee B$. We study the intrinsic properties of disjunctive algebras, and we prove that they are particular cases of implicative algebras.
6. The notion of conjunctive algebra, which relies on the decomposition of the implication $A \rightarrow B$ as $\neg(A \wedge \neg B)$. We explain how these structures naturally underly the realizability interpretations of some specific call-by-value calculus. We then prove that any disjunctive algebra induces a conjunctive algebra by duality. The converse implication and the properties of conjunctive algebras are yet to be studied.

The thesis itself is broadly organized according to the contributions listed above. We give here a description of the different chapters which compose this manuscript.

The first part of this thesis consists of a preliminary introduction to the scientific topics involved in the thesis. We attempt to be as self-contained as possible, and in particular these chapters are there to introduce well-known definitions and illustrate techniques which are relevant to the later contributions. As such, experts in the field should feel free to skip this part, all the more as back references are made to these chapters when necessary.

In Chapter 1, we give a self-contained introduction to formal logic, and present the concepts of theory, proof, and model. We come back in details to the notions of provability and validity evoked in the introduction, which we illustrate with several examples. Hopefully, this chapter should be accessible to anyone with a scientific background.

In Chapter 2, we introduce the λ-calculus, which is the fundamental model of computation for the study of functional programming languages. We first present the untyped λ-calculus, and we focus on the key properties that are in play in the study of such a calculus. We then present the simply-typed λ-calculus and the proofs-as-programs correspondence. Once again, this chapter is meant to be accessible to curious non-specialists, which may understand here the second half of this thesis title.

In Chapter 3, we give a survey of Krivine's classical realizability. In particular, we introduce the λ_c-calculus with its abstract machine, and we give in details the definition of classical realizability. We then present some of its standard applications, both as a tool to analyze the computational behavior of programs and as a technique of model construction.

In Chapter 4, we present Gentzen's sequent calculus. together with its computational counterpart, Curien and Herbelin's $\lambda\mu\tilde{\mu}$-calculus. We take advantage of this section to illustrate (on the call-by-name and call-by-name $\lambda\mu\tilde{\mu}$-calculi) the benefits of continuation-passing style translations and their relations with realizability interpretations *à la* Krivine. In particular, the expert reader might be interested in our observation that Danvy's methodology of semantic artifacts can be used to derive realizability interpretations.

The second part of this thesis is devoted to the study of a proof system allowing for the definition of a proof term for the axiom of dependent choice.

In Chapter 5, we give a comprehensive introduction to Herbelin's approach to the problem with dPA$^\omega$ [70]. We explain how the different computational features of dPA$^\omega$—namely dependent types, control operators and a co-inductive fixpoint which is lazily evaluated—are used to prove the axioms of countable and dependent choices. We then focus on the difficulties in proving the soundness of dPA$^\omega$, which are precisely related to the simultaneous presence of all these features. Finally, we present our approach to the problem, and the organization of the subsequent chapters.

In Chapter 6, we present a call-by-need calculus with control, the $\overline{\lambda}_{[lv\tau\star]}$-calculus. This calculus features explicit environments in which terms are lazily stored, which we use afterwards in dLPA$^\omega$. To prepare the later proof of normalization for dLPA$^\omega$, we prove the normalization of the $\overline{\lambda}_{[lv\tau\star]}$-calculus by means of a realizability interpretation. We also give a typed continuation-and-store passing style, whose computational content highlights the already known connection between global memory and forcing translations.

In Chapter 7, we introduce dL, a sequent calculus with control and dependent types. Here again, the underlying motivation is to pave the way for the further introduction of dLPA$^\omega$. Nonetheless, such a calculus is an interesting object in itself, which motivates our thorough presentation of the topic. We thus explain how control and dependent types can be soundly combined by means of a syntactic restriction of dependencies. We show how the challenge posed by the sequent calculus presentation can be solved thanks to the unexpected use of delimited continuations. The latter has the significant benefits of making the calculus suitable for a typed continuation-passing style carrying the dependencies.

Finally, in Chapter 8, we present dLPA$^\omega$, a calculus which soundly combines all the computational features of dPA$^\omega$ in a sequent calculus fashion. We give a realizability interpretation for dLPA$^\omega$, whose definition relies on the interpretations previously defined for the $\overline{\lambda}_{[lv\tau\star]}$-calculus and dL. We deduce from this interpretation the soundness and normalization of dLPA$^\omega$, the primary objectives of this part of the thesis.

The third part of the thesis is dedicated to the study of algebraic structures arising from the models that Krivine's classical realizability induces.

In Chapter 9, we give a detailed introduction to the topic, starting from Kleene intuitionistic realizability to eventually reach the notion of realizability triposes. In particular, we recall some standard definitions of the categorical analysis of logic. Then we present the algebraic approach to classical realizability and the structures that are involved.

In Chapter 10, we present Miquel's implicative algebras [121], which aim at providing a general algebraic framework for the study of classical realizability models. We first give a self-contained presentation of the underlying implicative structures. We then explain how these structures can be turned into models by means of separators. Finally, we show the construction of the associated triposes together with some criteria to determine whether the induced model amounts to a forcing construction.

In Chapter 11, we follow the rationale guiding the definition of implicative algebras to introduce the notion of disjunctive algebra. Our main goal in this chapter is to draw the comparison with the implicative case, and especially to justify that the latter provides a more general framework than disjunctive algebras. After studying the properties peculiar to disjunctive algebras, we eventually prove that they indeed are particular cases of implicative algebras.

Last, in Chapter 12, we attempt to follow the same process in order to define the notion of conjunctive algebra. If we succeed in proving that any disjunctive algebra give raise to a conjunctive algebra by duality (which is to be related with the well-known duality between call-by-name and call-by-value), we do not prove the converse implication. We conclude by saying a word on the perspectives and questions related to the algebraization of classical realizability.

Part I

Prelude

1- Logic

1.1 Theory

A famous character of a well-known book[1] once said to a young student of his:

> *The truth. [...] It is a beautiful and terrible thing, and should therefore be treated with great caution.*

While inattentive readers of this best-seller might have missed the significance of this declaration, it makes no doubt that this wise character intended to point out the fact that *truth* is a concept that is not as well-defined as one believes. This thesis being somewhat centered on the notions of *truth* and *proofs*, our starting point will be the definition of these key notions. In spite of a long faith in a total and absolute truth that mathematics ought to contain, belief of which Leibniz's quest for a *calculus ratiocinator* and Hilbert's second problem[2] only were the top of the iceberg, one of the major lesson from the 20th century in logic is that the notion of mathematical truth is deeply relative to its context and not uniquely defined.

In the next sections, we shall present two very different notions of truth. Considering again the example of geometry[2], two concepts are to be opposed. On the one hand, the *theory* of Euclidean geometry is an axiomatization intended to give a faithful representation of the world, expressed by means of Euclide's postulates. On the other hand, a *model* of this theory is a particular structure in which all the axioms of the theory hold. As explained in the introduction, a given axiomatization might be satisfied by several models. From these concepts are derived two different notions of truth:

- *provability*, a syntactic notion, expresses the existence of a proof in a theory,
- *validity*, a semantic notion, expresses the validation of a formula by a particular model of the theory.

Let us contemplate the case of Euclid's parallel postulate to illustrate the distinction between these notions. The parallel postulate is independent from other Euclid's postulates, that is to say that in the theory where only the first two postulates (cf. introduction) are assumed, the parallel postulate is neither provable nor disprovable. Notwithstanding, there exists at the same time a model in which it is *valid* (euclidian geometry) and different models in which it is not (non-euclidian geometries).

We shall start this section by introducing different concepts that are necessary to the definition of the concept of theory in Section 1.1.1, and pursue with the definition of a model in Section 1.2.

1.1.1 Language

Roughly, we can say that a theory is given by a *language*, which defines formulas and thus the expressiveness of the theory; and by the set of *theorems*, the formulas that are considered as true. Presented

[1] We deliberately choose to leave the precise reference apart from our bibliography, such an item would indubitably put the scientific rigor of this manuscript in question.

[2] See the introduction.

this way, truth corresponds to true formulas, which seems—and is—terribly tautological. The interesting point resides in defining which are the true formulas, and especially in how we define them. But before refining our notion of theory, let us first examine some examples of languages.

Example 1.1 (Propositional logic)**.** The language of propositional logic consists in propositions that are formed themselves by other propositions and the use of logical connectives. Specifically, we assume given a denumerable set $\mathcal{A}$ of *atomic formulas* and we define the *propositions* (or formulas), that are denoted by capital letters A, B, by:

$$A, B ::= X \mid \neg A \mid A \Rightarrow B \mid A \wedge B \mid A \vee B \qquad (X \in \mathcal{A})$$

where $\neg A$ reads "*not A*", $A \Rightarrow B$ reads "*A implies B*", $A \wedge B$ reads "*A and B*", and $A \vee B$ reads "*A or B*". We often consider that we have two particular atomic formulas in $\mathcal{A}$: *true*, that we write $\top$, and *false* that we write $\bot$, and if so, $\neg A$ is defined as $A \Rightarrow \bot$. It may be observed that our choice of connectives is arbitrary in the sense that we could have defined formulas from less or more connectives, or more generally from a signature of logical connectives. ⌟

While propositional logic can tracked to the 3$^{\text{rd}}$ century B.C.[3], the development of predicate logic, that can be considered as the next major advancement in logic, is much more recent and due to Frege in the 1870s. Intuitively, propositional logic only allows for declarative sentences such as *"I am a cat"* or *"Plato is a cat"* (or logical composition of declarative sentences, as in *"I am a cat" implies "I like fish"*), but it does not allow to identify the common structure "*be a cat*". Neither does it relate the *"I"* which is a cat and the *"I"* which likes fish. Less does it permit to express something like *"If x is a cat then x likes fish"*. The statement *"x is a cat"* or "$\text{Cat}(x)$" is what is called a predicate, depending on a variable x, and more generally denoted by $P(x)$. The main achievement of Frege was to introduce this notion, together with the concept of *quantification*, allowing to specify the quantity of individuals for which a statement holds. The *universal quantification*, written $\forall$, denotes the fact that a statement holds *for all* individuals: $\forall x. \text{Cat}(x)$ is *"for all x, x is a cat"*. The *existential quantification*, written $\exists$, denotes the *existence of (at least) one* individual for which the statement holds: $\exists x. \text{Cat}(x)$ is *"there exists x such that x is a cat"*. The resulting language is called the language of predicate logic or language of first-order logic.

Example 1.2 (First-order logic)**.** The language of *first-order logic* is defined from two different syntactic categories:

- *terms* or *first-order expressions*, that are built from a fixed set $\mathcal{V}$ of variables and a fixed signature Σ_1 of functions symbols with their arities[4]:

$$e_1, e_2 \ ::= \ x \mid f(e_1, ..., e_k) \qquad (x \in \mathcal{V}, f \in \Sigma_1)$$

- *formulas*, that are defined from a fixed signature Σ_2 of predicate symbols with their arities:

$$A, B \ ::= \ P(e_1, \ldots, e_k) \mid \forall x.A \mid \exists x.A \mid A \Rightarrow B \mid A \wedge B \mid A \vee B \qquad (P \in \Sigma_2)$$

⌟

It is worth noting that this language strictly subsumes the language of propositional logic, where atomic formulas are nothing more than predicates of arity 0.

[3] More precisely, to the stoic Chrysippus, according to the Stanford Encyclopedia of Philosophy: `https://plato.stanford.edu/archives/spr2016/entries/logic-ancient/`.

[4] Such a signature can formally be defined as a pair $\Sigma_1 = (\mathcal{F}, \text{ar})$ where $\mathcal{F}$ is a denumerable set of functions symbols and ar is a function $\mathcal{F} \to \mathbb{N}$ which assigns to each function its arity, *i.e.* the number of arguments it takes.

Example 1.3 (First-order arithmetic)**.** The language of first-order arithmetic is a special case of a first-order language, where the signature for first-order expressions contains a constant 0 (function of arity 0), a symbol S (of arity 1) to denote the successor, as well as two function symbols $+$ and $\times$ denoting respectively the addition and the multiplication of natural numbers. As for the formulas, they are defined with the two quantifiers of first-order logic and one unique predicate symbol $=$ to denote the equality of terms. The resulting syntax, where $\mathcal{V}$ is the set of variables, is given by:

$$\begin{array}{llcll} \textbf{Terms} & e_1, e_2 & ::= & x \mid 0 \mid s(e) \mid e_1 + e_2 \mid e_1 \times e_2 & (x \in \mathcal{V}) \\ \textbf{Formulas} & A, B & ::= & e_1 = e_2 \mid \top \mid \bot \mid \forall x.A \mid \exists x.A \mid A \Rightarrow B \mid A \land B \mid A \lor B & \end{array}$$

⌟

These languages are called *first-order* because quantification is only authorized over first-order terms (natural numbers in the case of arithmetic). As we shall use further in this manuscript second-order or higher-order logic, let us give some more insight on this point.

Remark 1.4 (Order of a language)**.** Let us informally define $\texttt{Prop}$ as the "set" of propositions. Intuitively, we could think of $\texttt{Prop}$ as being the set that only contains *true* and *false*: $\texttt{Prop} = \{\top, \bot\}$. In the case of arithmetic, first-order individuals corresponds to natural numbers in $\mathbb{N}$. A predicate $P(x_1, \dots, x_k)$ is thus a function from $\mathbb{N}^k$ to $\texttt{Prop}$. Alternatively, one can think of a predicate $P(x)$ as a set P of naturals number, with $P(x) \equiv x \in P$. This way, second-order individuals are sets in $\mathcal{P}(\mathbb{N})$, third-order individuals are sets of sets in $\mathcal{P}(\mathcal{P}(\mathbb{N}))$, fourth-order sets of sets of sets, etc... : n^{th}-order individuals are elements of $\underbrace{\mathcal{P}(\cdots\mathcal{P}(}_{n-1}\mathbb{N})\cdots)$. With this intuition in mind, we say that a n^{th}-order language is a language that allows for quantifications ranging over n^{th}-order individuals. For instance:

- zero-order logic is just propositional logic, since it does not allow any quantification,
- first-order logic is indeed predicate logic, which allows for quantifications over terms and expresses properties about natural numbers,
- second-order logic corresponds to a language with quantifications ranging over predicates and expresses properties about sets of natural numbers,
- etc... ⌟

Up to now, in each example we only defined a language, whose symbols were not given any particular logical signification. Specifically, we said for instance that "=" denoted the equality, that "+" denoted the addition or that $s(0)$ was the successor of 0, so that any reader should be inclined to think of $s(0)$ as 1 and to $1 + 1$ as 2. But there is no formal reason to do so!

In other words, we do not have any relation yet between $s(0) + s(0)$ and $s(s(0))$. We can write $s(0) + s(0) = s(s(0))$ just like we can write $s(0) = 0$ or $\top \Rightarrow \bot$, because in both cases the language is expressive enough. But we still need to give some kind of meaning to these symbols, and a least to define what we consider as true statements. To put it differently, we need to define what is the logical content of a theory.

We can now refine our notion of theory. A theory consists in three elements, namely:

- a *language*, which delimits the expressiveness of the theory;
- *axioms*, a minimal set[5] of closed formulas taken as true;
- a *deductive system*, which allows to deduce theorems from the axioms.

By minimal, we mean that none of the axioms should be proved from the other one using the deductive system, which we shall now define. By closed, we mean that a formula can only contain variables that are bound by some quantifier. For instance, $\forall x.\exists y.y = x + x$ is a closed formula but $\exists y.y = x + x$ is not since x is *free*. Formally, we define by induction the set of free variables $FV(A)$ of a formula A and say that a formula A is *closed* if $FV(A) = \emptyset$.

[5] These sets will mostly be finite in this manuscript.

Definition 1.5 (Free variables)**.** The sets of free variables of first-order terms and formulas are inductively defined by :

$$\begin{array}{rcl@{\qquad\qquad}rcl}
FV(x) & \triangleq & \{x\} & FV(f(e_1,...,e_k)) & \triangleq & FV(e_1) \cup \ldots \cup FV(e_k) \\
FV(A \Rightarrow B) & \triangleq & FV(A) \cup FV(B) & FV(P(e_1,\ldots,e_k)) & \triangleq & FV(e_1) \cup \ldots \cup FV(e_k) \\
FV(A \wedge B) & \triangleq & FV(A) \cup FV(B) & FV(\forall x.A) & \triangleq & FV(A)\backslash\{x\} \\
FV(A \vee B) & \triangleq & FV(A) \cup FV(B) & FV(\exists x.A) & \triangleq & FV(A)\backslash\{x\}
\end{array}$$

⌟

Similarly, we define $A[e/x]$, which reads *"the formula A in which x is substituted by e"*, that we will use in the next section.

Definition 1.6 (Substitution)**.** The substitution of a variable x by an expression e is defined by induction over terms:

$$\begin{array}{rcll}
y[e/x] & \triangleq & e & (\text{if } x = y) \\
y[e/x] & \triangleq & y & (\text{if } x \neq y) \\
(f(e_1,...,e_k))[e/x] & \triangleq & f(e_1[e/x],\ldots,e_k[e/x]) &
\end{array}$$

and formulas:

$$\begin{array}{rcll}
(P(e_1,\ldots,e_k))[e/x] & \triangleq & P(e_1[e/x],\ldots,e_k[e/x]) & \\
(A \Rightarrow B)[e/x] & \triangleq & A[e/x] \Rightarrow B[e/x] & \\
(A \wedge B)[e/x] & \triangleq & A[e/x] \wedge B[e/x] & \\
(A \vee B)[e/x] & \triangleq & A[e/x] \vee B[e/x] & \\
(\forall y.A)[e/x] & \triangleq & \forall y.(A[e/x]) & (\text{if } x \neq y, y \notin FV(e)) \\
(\forall y.A)[e/x] & \triangleq & \forall y.A & (\text{otherwise}) \\
(\exists y.A)[e/x] & \triangleq & \exists y.(A[e/x]) & (\text{if } x \neq y, y \notin FV(e)) \\
(\exists y.A)[e/x] & \triangleq & \exists y.A & (\text{otherwise})
\end{array}$$

Observe that in the case where the variable x corresponds to the variable bound by a quantifier (*e.g.* $\forall x.A$), the substitution is erased. ⌟

1.1.2 Deductive system

The aim of a deductive system is to capture the notion of logical consequence in a theory. There exist numerous deductive systems doing so, of which the most known are Hilbert's deduction system, natural deduction and Gentzen's sequent calculus. We will implicitly present Hilbert's system in Chapter 10, and we will introduce sequent calculus in Chapter 4. Let us focus now on the system of natural deduction, that we present with explicit contexts. Assume that we have a fixed language, for instance the language of first-order logic. We call *context* any list (possibly empty) of formulas written $\Gamma \equiv A_1,\ldots,A_n$. Formally, this corresponds to the simple following grammar:

$$\Gamma ::= \varepsilon \mid \Gamma, A$$

and we define $FV(\Gamma)$ as the union of free variables in each formula:

$$FV(\varepsilon) \triangleq \varepsilon \qquad\qquad FV(\Gamma, A) \triangleq FV(\Gamma) \cup FV(A)$$

A *judgment* is a pair (Γ, A) written $\Gamma \vdash A$, where Γ is a context and A is a formula. Intuitively, the sequent $\Gamma \vdash A$ expresses that the formula A is a logical consequence of the hypotheses Γ. Sequents are deduced from each other by means of a *deductive system.* A deductive system is given by a set of *inference rules* , which are of the form:

$$\frac{J_1 \quad \ldots \quad J_n}{J}\ (\mathit{bli})$$

Propositional logic

(Introduction rules)

$$\frac{A \in \Gamma}{\Gamma \vdash A}\ (\mathrm{Ax}) \qquad \frac{}{\Gamma \vdash \top}\ (\top)$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ (\Rightarrow_I)$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\wedge_I)$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ (\vee_I^1) \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ (\vee_I^2)$$

(Elimination rules)

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ (\bot)$$

$$\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ (\Rightarrow_E)$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ (\wedge_E^1) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ (\wedge_E^2)$$

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ (\vee_E)$$

First-order logic

$$\frac{\Gamma \vdash A \quad x \notin FV(\Gamma)}{\Gamma \vdash \forall x.A}\ (\forall_I) \qquad \frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[t/x]}\ (\forall_E)$$

$$\frac{\Gamma \vdash A[t/x]}{\Gamma \vdash \exists x.A}\ (\exists_I) \qquad \frac{\Gamma \vdash \exists x.A \quad \Gamma, A \vdash B \quad x \notin FV(\Gamma, B)}{\Gamma \vdash B}\ (\exists_E)$$

Figure 1.1: Natural deduction

where *bli* is the name of the rule, where the judgment J is the conclusion of the rule and where $J_1, \ldots, J_n$ are its premises. The rules of natural deduction, given in Figure 1.1, are divided in two sorts of rules:

- *introduction rules*, that give the necessary premises to introduce a connective,
- *elimination rules*, that give a conclusion that is derivable from a connective.

For instance, the elimination for the connective $\Rightarrow$ is none other than the Aristotelian principle of *modus ponens*:

$$\frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ (\Rightarrow_E)$$

expressing that knowing $A \Rightarrow B$ and A, one can deduce B. Some rules (the axiom rule, the introduction of $\forall$ and the elimination of $\exists$) also have a side-condition to restrict their scope. For example, the rule (Ax) only applies if the formula A appears in the list Γ of hypotheses, while the introduction rule for $\forall$ applies only if the variable x does not occur freely in Γ (intuitively, x refers to any arbitrary term).

Succession of inferences are then arranged in the form of a *derivation tree*, whose root is traditionally located at the bottom. A sequent $\Gamma \vdash A$ is said to be *derivable* if there exists a derivation tree whose root is this sequent. This derivation tree is also called *proof tree* or simply *proof*.

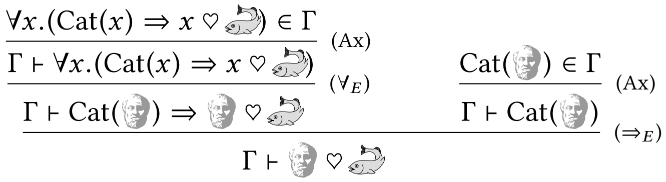
Example 1.7 (Plato likes fish)**.** Let us illustrate how natural deduction works by constructing the derivation tree corresponding to the syllogism: "*Plato is a cat, all cats likes fish thus Plato likes fish*". We define two predicates $\mathrm{Cat}(x)$ and $x \heartsuit y$ by :

$$\mathrm{Cat}(x) \triangleq \text{“ } x \text{ is a cat”} \qquad\qquad x \heartsuit y \triangleq \text{“ } x \text{ likes } y\text{”}$$

and denote Plato by and "fish" by . Our hypothesis, which will constitute the context Γ, are then defined by:

$\Gamma = \mathrm{Cat}($ $), \forall x.(\mathrm{Cat}(x) \Rightarrow x \heartsuit$ $)$

All this being set, we are now ready to give the expected derivation:



This proof tree reflects the structure of the expected proof. From bottom to top (and right to left), this proof can be read[6]:

- Plato likes fish by application of the modus ponens ($\Rightarrow_E$), since "if Plato is a cat then Plato likes fish" and "Plato is a cat",
- the latter holds because it is an hypothesis (Ax),
- the former holds because it is in fact true for any individual ($\forall_E$): "for all x, if x is a cat then x likes fish",
- this last statement is an hypothesis (Ax).

We enjoin the reader desirous of getting more familiar with the manipulation of proof trees to do the following exercises:

1. Introduce a predicate $\text{Fish}(x) \triangleq$ *"x is a fish"*. Then generalize the hypothesis as *"any cat like any fish"* and consider some fish to prove that Plato likes it.
2. Give a different derivation of the same judgment.
3. Change the hypothesis *"Plato is a cat"* by *"Plato does not like fish"* and prove that *"Plato is not a cat"*. ⌟

1.1.2.1 Intuitionistic and classical logic

Alternatively, one can think of an inference rule as a logical axiom. Indeed, the choice of inference rules is not inconsequential and all deductive systems are not equivalent. Natural deduction, as we presented it, is said to be intuitionistic or constructive, because it only entails constructive principle. For instance, to construct a proof of a disjunction $A \vee B$, we need to actually choose between its left-hand side A or its right-hand side B. As a consequence, the De Morgan law:

$$\neg(A \wedge B) \Rightarrow (\neg A) \vee (\neg B)$$

is not provable[7] in natural deduction with an empty context. Intuitively, this is due to the fact that the knowledge of $\neg(A \wedge B)$ only provides us with the information that *"A and B"* is not true, it does not tell us whether A or B (or both) is false. Hence we have no way to prove $(\neg A) \vee (\neg B)$, which requires to give either a proof of $\neg A$ or a proof of $\neg B$. Similarly, the principle of *excluded-middle*:

$$A \vee (\neg A)$$

[6] This corresponds to the way the proof tree is build. The natural way of constructing a "hand-written" proof would be just the opposite, from top to bottom: *We know that for any individual x, if x is a cat, then x likes fish. In particular, if Plato is a cat, then he likes fish. But we also know that Plato is a cat, hence he likes fish.*

[7] The De Morgan law is not "false" in the sense that its negation is provable (which is not), but it is indeed not provable (we will prove this in Section 1.2). Such an affirmation might seem puzzling at first sight (how can we prove the unprovability of a formula?), but it is one of the biggest motivation to the introduction of a semantical truth through models.

is not provable[8] for all formulas, since it requires to effectively know whether A is true or not. If we can prove one of A or $\neg A$, we can obviously prove $A \lor (\neg A)$, if not we are stuck.

On the opposite, *classical logic* allows for instance to deduce a proof of $A \lor B$ from a *reductio ad absurdum*: supposing that neither A nor B, one might obtain a proof of false ($\bot$) which is absurd, and conclude that the hypothesis was false, hence A or B is true. This formally corresponds to the addition of an extra logical axiom, which is usually chosen amongst these three principles:

$$\begin{array}{ccc} A \lor (\neg A) & (\neg\neg A) \Rightarrow A & ((A \Rightarrow B) \Rightarrow A) \Rightarrow A \\ \textit{(Excluded-middle)} & \textit{(Double-negation elimination)} & \textit{(Peirce's law)} \end{array}$$

None of these axioms is provable in intuitionistic natural deduction, and they are logically equivalent in the sense that any one of them is deducible from any other one[9]. It is worth saying that in spite of our presentation—which is mostly intuitionistic in this chapter—, classical logic is the logic the wo.man in the street is accustomed to. In particular, most of mathematicians consider the double-negation elimination or the excluded-middle as valid principles for reasoning and proving theorems.

Remark 1.8. The Curry-Howard correspondence, that will be presented in Section 2.3, makes this idea of *constructivism* even stronger: it associates to each proof a program whose computation corresponds to the proof. Originally formulated in an intuitionistic setting, it was then extended to a classical framework thanks to a clever interpretation of Peirce's law. All this manuscript is dedicated to the study of classical proofs through this interpretation. $\lrcorner$

1.1.3 Theory

Given by a language together with a deductive system and a set of axioms, a theory $\mathscr{T}$ allows to deduce theorems by means of logical consequences. Formally, a *demonstration* or *proof* of a formula A in the theory $\mathscr{T}$ is a derivation whose conclusion is of the form $\Gamma \vdash A$, where Γ is a (finite) set of axioms of $\mathscr{T}$. When such a demonstration exists, A is called a *theorem* of $\mathscr{T}$. The theory $\mathscr{T}$ is said to be incoherent or inconsistent whenever the formula $\bot$ is a theorem of $\mathscr{T}$ (or, equivalently, when any formula is a theorem of $\mathscr{T}$). Otherwise, the theory is said to be *coherent* or *consistent*. Furthermore, a theory $\mathscr{T}$ is said to be *complete* if for each formula A, either A is a theorem of $\mathscr{T}$ either its negation $\neg A$ is.

Example 1.9 (Intuitionistic logic)**.** The theory of intuitionistic propositional logic NJ is the theory obtain from the propositional rules of natural deduction (see Figure 1.1) with no further axioms. $\lrcorner$

Example 1.10 (Relations)**.** A *relation* corresponds to a predicate $\mathcal{R}(x,y)$ of arity 2, that we rather write $x \,\mathcal{R}\, y$. Numerous generic properties about relations can be defined in first-order logic, amongst which:

(R1)	*Reflexivity* :	$\forall x. x \,\mathcal{R}\, x$
(R2)	*Transitivity* :	$\forall x.\forall y.\forall z. x \,\mathcal{R}\, y \Rightarrow y \,\mathcal{R}\, z \Rightarrow x \,\mathcal{R}\, z$
(R3)	*Anti-symmetry* :	$\forall x.\forall y. x \,\mathcal{R}\, y \Rightarrow y \,\mathcal{R}\, x \Rightarrow x = y$
(R4)	*Symmetry* :	$\forall x.\forall y. x \,\mathcal{R}\, y \Rightarrow y \,\mathcal{R}\, x$
(R5)	*Totality* :	$\forall x.\forall y. x \,\mathcal{R}\, y \lor y \,\mathcal{R}\, x$

A relation is called a *pre-order*, and often written $\leq$, if it is reflexive and transitive *i.e.* if (R1),(R2) are theorems of the ambient theory. If (R3) is also a theorem (the pre-order is anti-symmetric), it is called an *order*. An order is total if it satisfies the condition (R5). An *equivalence* is a relation for which (R1),(R2) and (R4) holds. $\lrcorner$

[8] We will give a formal argument of this statement in Section 1.2.2. In fact, we will even prove that the excluded-middle is independent from intuitionistic logic, that is to say that neither the excluded-middle nor its negation are provable.

[9] Proving the equivalence is a nice and classical exercise.

Example 1.11 (Theory of equality)**.** The theory of equality, in the language of first-order arithmetic, corresponds to the following axioms:

(E1)	$\forall x.(x = x)$
(E2)	$\forall x.\forall y.\forall z.(x = y \wedge x = z \Rightarrow y = z)$
(E3)	$\forall x.\forall y.(x = y \Rightarrow s(x) = s(y))$
(E4)	$\forall x.\forall y.\forall z.(x = y \Rightarrow x + z = y + z)$
(E5)	$\forall x.\forall y.\forall z.(x = y \Rightarrow z + x = z + y)$
(E6)	$\forall x.\forall y.\forall z.(x = y \Rightarrow x \times z = y \times z)$
(E7)	$\forall x.\forall y.\forall z.(x = y \Rightarrow z \times x = z \times y)$

Observe that the first two axioms (E1) and (E2) imply that the relation of equality is reflexive, transitive, symmetric and anti-symmetric. ⌟

If equalities as $1 = 1$ or $1 + 2 = 1 + 2$ are simple consequences of the axioms (E1-E7), the equality $1 + 1 = 2$ (*i.e.* $s(0) + s(0) = s(s(0))$) is still not provable. Indeed, such an equality relies on properties of the addition and not of the equality. Similarly, $1 \times 1 = 1$ relies on properties of the multiplication. These properties are expressed by Peano axioms, which define the theory of first-order arithmetic.

Example 1.12 (Peano arithmetic)**.** The theory of *Peano arithmetic*, that we write (PA), is obtained by adding to the theory of equality the six axioms below:

(PA1)	$\forall x.(0 + x = x)$
(PA2)	$\forall x.\forall y.(s(x) + y = s(x + y))$
(PA3)	$\forall x.(0 \times x = 0)$
(PA4)	$\forall x.\forall y.(s(x) \times y = (x \times y) + y)$
(PA5)	$\forall x.\forall y.(s(x) = s(y) \Rightarrow x = y)$
(PA6)	$\forall x.(s(x) \neq 0)$

as well as the *axioms of induction*:

(PA7) $$\forall z_1 \ldots z_n (A[x/0] \wedge \forall x.(A \Rightarrow A[s(x)/x]) \Rightarrow \forall x.A)$$

for each formula A whose free variables are $x, z_1, \ldots, z_n$. ⌟

Finally, we have now at our disposal a theory in which we can indeed assert that $1 + 1 = 2$

Theorem 1.13 (1+1=2)**.** $\mathrm{PA} \vdash s(0) + s(0) = s(s(0))$

Proof. We only sketch the proof in english, and let any circumspect reader derive the formal proof tree. The axiom PA2 implies that $s(0) + s(0) = s(0 + s(0))$ and PA1 implies that $0 + s(0) = s(0)$. Using the axiom (E3) of equality, we deduced that $s(0 + s(0)) = s(s(0))$, and we conclude by transitivity of the equality (E2). □

It is easy to check that expected properties of arithmetic are provable with these axioms, for instance that the successor corresponds indeed to the addition of 1 (*i.e.* $s(0)$):

$$\mathrm{PA} \vdash \forall x.x + s(0) = s(x)$$

or that the principle of strong induction holds:

$$\mathrm{PA} \vdash \forall x.(\forall y.(y < x \Rightarrow A(y)) \Rightarrow A(x)) \Rightarrow \forall x A(x)$$

1.1.3.1 Gödel's incompleteness

Unfortunately for Leibniz's and Hilbert's dream of an absolute truth, the notion of provability does not meet this expectancy. Indeed, this syntactic concept of truth does not allow to decide of the truth of all statements: some statements are neither provable nor provable. More precisely, as soon as a theory $\mathscr{T}$ is expressive enough, either there is a closed formula G such that $\mathscr{T} \nvdash G$ and $\mathscr{T} \nvdash \neg G$ or the theory is incoherent. This is known as Gödel first incompleteness theorem [61], who managed to adapt the old liar's paradox:

"I am a liar"

to the theory of arithmetic. Roughly, Gödel defined an encoding $\ulcorner\cdot\urcorner$ of the formulas and demonstrations of first-order arithmetic to natural numbers[10] This encodings allows to convert the statement *"A is a theorem of $\mathscr{T}$"* into the statement *"x is the code of a theorem of $\mathscr{T}$"*, which can be expressed as an arithmetic formula. This permits the definition of the following formula G:

$$G \triangleq \neg Th(\ulcorner G \urcorner) \qquad \text{(“}\ulcorner G\urcorner \textit{ is not the code of a theorem of } \mathscr{T}\text{”).}$$

If $\mathscr{T}$ is coherent, $\mathscr{T}$ can not prove G, otherwise G would be a theorem and $\mathscr{T}$ would prove $\ulcorner G \urcorner$ is not the code of a theorem of $\mathscr{T}$. Neither can $\mathscr{T}$ prove $\neg G$, *i.e.* $\ulcorner G \urcorner$ is the code of a theorem, since G would not be a theorem and $\mathscr{T}$ would be inconsistent.

To Hilbert's claim *"For us mathematicians there is no 'Ignorabimus'[...] we shall know!"*, Gödel's theorem somehow answers: *"No, my dear, we won't !"*.

Theorem 1.14 (First incompleteness theorem). *If $\mathscr{T}$ is coherent and contains PA, then $\mathscr{T}$ is incomplete.*

1.2 Models

We shall now contemplate a semantic notion of truth, namely the satisfiability by a *model.* As explained in the introduction, while a theory specifies the axioms and rules that are to be satisfied, giving an axiomatic representation of the world, a model $\mathcal{M}$ of a theory $\mathscr{T}$ is the given of one possible world in which all the theorems of $\mathscr{T}$ are satisfied. If the distinction between the syntax and the semantics of a sentence can be traced back to older works[1], model theory as the study of the interpretation of a language by means of set-theoretic structures is mostly based on Alfred Tarski's truth definition [153].

Given a theory $\mathscr{T}$, that is to say a language $\mathscr{L}$ together with a set of axioms and deduction rules, a model is the given of a universe in which the language $\mathscr{L}$ is interpreted and of a relation of satisfiability such that the interpretation of each theorem of $\mathscr{T}$ is satisfied. Let us examine a simple example before giving a formal definition.

Example 1.15. Consider the language of first-order arithmetic (Example 1.3), in a theory without axioms (*i.e.* theorems are logical tautologies), and consider the statement:

$$\forall x.(0 + x = x)$$

which is the first axiom (PA1) of Peano arithmetic. In this context, it is not an theorem, hence it can be either true or false in a model. The first natural interpretation we might come with is to choose as universe the set $\mathbb{N}$ of natural numbers, to interpret '0' by the natural 0, '+' by the addition of natural

[10] You can think of this as an enumeration of every possible formulas and demonstrations. It corresponds to something like 0 is the code for $\top$,1 is the code for $\bot$,..., 42 is the code for the proof of the conjunction of formulas of code 5 and 7, etc... and $\ulcorner\forall x.\forall y.x + y = 27\urcorner = 137668$. The key point is that every formula and demonstration have a code.

[11] Besides the aforementioned works on non-Euclidean geometries, Frege's works can be pointed out: he formally introduced the distinction between the character x and the quoted 'x' to distinguish between the signified and the signifier.

numbers and '$=$' by the equality on natural numbers. We write $\mathbb{N} \vDash A$ to denote that $\mathbb{N}$ satisfies the formula A, and we define the satisfiability of the universal quantifier by:

$$\mathbb{N} \vDash \forall x.A(x) \qquad \text{if and only if} \qquad \text{for all } n \in \mathbb{N}, \mathbb{N} \vDash A(n)$$

Then (PA1) is true with respect to this interpretation, since for any natural number n, $\mathbb{N} \vDash 0 + n = n$.

Now, we could also give a different interpretation. Consider the set $\mathcal{W}$ of (finite) words defined on the usual alphanumeric alphabet '$0-9, a-z$'. We interpret 0 by the character 0, + by the concatenation of words and = by the equality. We define the satisfiability of the universal quantifier in a similar way:

$$\mathcal{W} \vDash \forall x.A(x) \qquad \text{if and only if} \qquad \text{for all } w \in \mathcal{W}, \mathcal{W} \vDash A(w)$$

Then (PA1) is false with respect to this interpretation: indeed, if we consider for instance the word '*abc*', we have $0 + abc = 0abc \neq abc$, *i.e.* $\mathcal{W} \nvDash 0 + abc = abc$. Thus $\mathcal{W}$ does not satisfies (PA1): $\mathcal{W} \nvDash \forall x.(0 + x = x)$. ⌟

Formally, given a language $\mathscr{L}$, a pair $(\mathcal{M},\mathcal{I})$ is said to be an $\mathscr{L}$-structure if $\mathcal{I}$ maps the symbols of $\mathscr{L}$ to appropriate elements of $\mathcal{M}$: function symbols are mapped to functions (of the corresponding arity) and predicates are mapped to functional relations. $\mathcal{M}$ is called the universe of the structure, and $\mathcal{I}$ its interpretation function.

Definition 1.16 (Model)**.** Given a $\mathscr{L}$-structure, a *formula* $A(m_1,\ldots,m_n)$ *with parameters in* $\mathcal{M}$ is defined as a formula $A(x_1,\ldots,x_n)$ whose free variables $x_1,\ldots,x_n$ have been substituted by elements $m_1,\ldots,m_n$ of $\mathcal{M}$. Finally, a $\mathscr{L}$-structure $(\mathcal{M},\mathcal{I})$ is said to be a *model* of a theory $\mathscr{T}$ if there is a relation of *satisfiability* over formulas with parameters in $\mathcal{M}$, such that every theorem of $\mathscr{T}$ are satisfied by $\mathcal{M}$. This relation is often denoted by $\mathcal{M} \vDash A$ and reads A is *valid* (or true) in $\mathcal{M}$ or $\mathcal{M}$ satisfies A. ⌟

In practice, the relation of satisfiability is defined primitively on atomic formulas and then by induction on the structure of a formula. If the definition is adequate with the deductive system, then the resulting relation defines indeed a model.

Definition 1.17 (Adequacy)**.** Let $\mathscr{L}$ be a language, $\mathscr{T}$ be a theory based on this language and $\mathcal{M}$ be an $\mathscr{L}$-structure.

- A judgment $\Gamma \vdash A$ in $\mathscr{T}$ is *adequate* (w.r.t. to the model $\mathcal{M}$) if the validity of the premises ($\mathcal{M} \vDash \Gamma$) entails the validity of the conclusion ($\mathcal{M} \vDash A$).
- More generally, we say that an inference rule

$$\frac{J_1 \quad \cdots \quad J_n}{J_0}$$

is adequate (w.r.t. to the model $\mathcal{M}$) if the adequacy of all judgments $J_1,\ldots,J_n$ implies the adequacy of the typing judgment J_0. ⌟

Proposition 1.18. *If all the axioms of a theory $\mathscr{T}$ are valid in a structure $\mathcal{M}$, and if all its rules of inference are adequate, then $\mathcal{M}$ is a model of $\mathscr{T}$.*

Proof. Indeed, if there is a proof of a formula A in $\mathscr{T}$, this proof is build out of axioms and inferences rules. Since axioms are valid in $\mathcal{M}$ and inference rules are adequate w.r.t. $\mathcal{M}$, by induction we get that adequate judgments at every floors of the tree. In particular, the root of the proof tree ($\mathscr{T} \vdash A$) is adequate, that is to say that $\mathcal{M} \vdash A$ is valid. This is true for every theorem of $\mathscr{T}$, hence $\mathcal{M}$ is a model of $\mathscr{T}$. □

In particular, if $\mathscr{T}$ is not coherent (*i.e.* $\mathscr{T} \vdash \bot$), then $\bot$ is valid in any model $\mathcal{M}$. By contraposition, this gives us a semantic criterion of coherency.

Corollary 1.19 (Coherence)**.** *If a theory $\mathscr{T}$ has a model $\mathcal{M}$ such that $\bot$ is not valid in $\mathcal{M}$, then $\mathscr{T}$ is coherent.*

Unlike for provability, in a model any statement is necessarily[2] either satisfied or not. Nevertheless, the same theory can admit very different models, and a statement can be true in some of them, false in others. This justifies the introduction of the notion of completeness, which corresponds to the implication dual to soundness (which is the very definition of a model):

$$\begin{array}{lrcl}
\text{(Soundness)} & \mathscr{T} \vdash A & \Rightarrow & \mathcal{M} \vDash A \\
\text{(Completeness)} & \mathcal{M} \vDash A & \Rightarrow & \mathscr{T} \vdash A
\end{array}$$

Definition 1.20 (Completeness)**.** A theory $\mathscr{T}$ is said to be *complete* with respect to a class of models $\mathcal{M}$ if for all formula A, the satisfiability of A in $\mathcal{M}$ ($\mathcal{M} \vDash A$) for any such model $\mathcal{M}$ implies the provability of A in $\mathscr{T}$ ($\mathscr{T} \vdash A$). ⌟

We shall examine now some examples of models.

1.2.1 Truth tables

The easiest model of all for propositional logic is known since the antiquity, and consists in a truth table with only two elements[3] $\top$ and $\bot$. The interpretation of the different connectives is defined as internal laws, whose values are given by the following truth tables:

$p \wedge q$		
q p	$\top$	$\bot$
$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$

$p \vee q$		
q p	$\top$	$\bot$
$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$

$p \Rightarrow q$		
q p	$\top$	$\bot$
$\top$	$\top$	$\bot$
$\bot$	$\top$	$\top$

p	$\neg p$
$\top$	$\bot$
$\bot$	$\top$

Formally, given a propositional theory $\mathscr{T}$ this corresponds to a model $\mathcal{M} = \{\top, \bot, \}$ such that the interpretation function maps every axioms (atomic propositions) to $\top$ and to the following definition of the satisfiability relation :

$$\begin{array}{lll}
\mathcal{M} \vDash \top & & \\
\mathcal{M} \vDash A \wedge B & \text{if and only if} & \mathcal{M} \vDash A \ \text{ and } \ \mathcal{M} \vDash B \\
\mathcal{M} \vDash A \vee B & \text{if and only if} & \mathcal{M} \vDash A \ \text{ or } \ \mathcal{M} \vDash B \\
\mathcal{M} \vDash A \Rightarrow B & \text{if and only if} & \mathcal{M} \vDash A \ \text{ implies } \ \mathcal{M} \vDash B \\
\mathcal{M} \vDash \neg A & \text{if and only if} & \mathcal{M} \nvDash A
\end{array}$$

This definition can be extended to judgments by defining:

$$\begin{array}{lll}
\mathcal{M} \vDash A_1, \ldots, A_n & \text{if and only if} & \mathcal{M} \vDash A_1 \wedge \cdots \wedge A_n \\
\mathcal{M} \vDash \Gamma \vdash A & \text{if and only if} & \mathcal{M} \vDash \Gamma \ \text{ implies } \ \mathcal{M} \vDash A
\end{array}$$

and it is easy to check that all the inference rules for propositional logic in Figure 1.1 are adequate. Besides, it is worth noting that such a model always validates the excluded middle since:

$$\mathcal{M} \vDash A \vee (\neg A) \ \Leftrightarrow \ \mathcal{M} \vDash A \text{ or } \mathcal{M} \vDash (\neg A) \ \Leftrightarrow \ \mathcal{M} \vDash A \text{ or } \mathcal{M} \nvDash A$$

[12] This actually means that we consider our meta-theory to be classical, but for the sake of simplicity, we do not want to dwell on considerations about meta-theory here.

[13] Formally, we should call them *True* and *False* (or with any other names), which are elements of the model, so as to distinguish them from $\top$ and $\bot$, which are elements of the syntax and of whom they are the interpretations. We abuse the notations in the same way for the logical connectives.

1.2.2 Heyting algebra

Heyting algebras, named after the mathematician Arend Heyting, are a generalization of truth tables for intuitionistic logic. They allow to interpret propositions in a partially ordered set that has more than just two points, where the structure of ordering reflects the logical behavior of connectives. The main intuition can be resumed by the motto:

"*the higher an element is, the truer it is*"

In particular, if $x \leq y$ and x is "true", then so is "y". Reading this order the other way around, $x \leq y$ means than x is more precise (or contains more information, is more constrained) than y. Implicative algebras, that we will present in Chapter 10, are a generalization of Heyting algebras (and of this intuition).

Definition 1.21 (Lattice)**.** A *lattice* is a partially ordered set $(\mathcal{L}, \leq)$ such that every pair of elements $(a,b) \in \mathcal{L}^2$ has a lower bound $a \wedge b$ and an upper bound $a \vee b$. ⌟

This defines two internal laws $\wedge, \vee : \mathcal{L}^2 \to \mathcal{L}$, of which we can show[4] that they fulfill the following properties:

- for all $a,b \in \mathcal{L}$, $a \wedge b = b \wedge a$ and $a \vee b = b \vee a$ (*Commutativity*)
- for all $a,b,c \in \mathcal{L}$, $a \wedge (b \wedge c) = (a \wedge b) \wedge c$ and $a \wedge (b \wedge c) = (a \wedge b) \wedge c$ (*Associativity*)
- for all $a,b \in \mathcal{L}$, $\forall a,b, a \wedge (a \vee b) = a = a \vee (a \wedge b)$ (*Absorption*)
- for all $a,b \in \mathcal{L}$, $a \leq b \Leftrightarrow a \vee b = b \Leftrightarrow a \wedge b = a$ (*Consistency (w.r.t.* $\leq$*)*)

Definition 1.22 (Heyting algebra)**.** A *Heyting algebra* $\mathcal{H}$ is defined as a bounded lattice $(\mathcal{H}, \leq)$ such that for all a and b in $\mathcal{H}$ there is a greatest element x of $\mathcal{H}$ such that

$$a \wedge x \leq b$$

This element is denoted by $a \to b$, while the upper and lower bound of $\mathcal{H}$ are respectively written $\top$ and $\bot$. ⌟

It is worth noting that by definition we have:

$$a \wedge (a \to b) \leq b$$

that is, following our intuition, that b is "truer" than $a \wedge (a \to b)$. Indeed, if a and $a \to b$ are true, so should be b according to the rule of *modus ponens*. Besides, $a \wedge (a \to b)$ is indeed more precise than just b, in that it contains information that b has not.

Given a Heyting algebra, it suffices to define the interpretation of atomic formulas to get a model of propositional intuitionistic logic. Assume that every atomic formula A is mapped to a *truth value* $|A|$ that is an element of $\mathcal{H}$, so that every axiom is mapped to $\top$. In the case of the theory NJ, this requirement simply corresponds to the equation $|\top| = \top$. Then we can naturally extend the definition of $|\cdot|$ to meet all the formulas:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
|A \wedge B| & \triangleq & |A| \wedge |B| & |A \Rightarrow B| & \triangleq & |A| \to |B| \\
|A \vee B| & \triangleq & |A| \vee |B| & |\neg A| & \triangleq & |A| \to \bot
\end{array}$$

and extend once again the definition to judgments by:

$$|A_1, \ldots, A_n| \triangleq |A_1| \wedge \ldots \wedge |A_n| \qquad |\Gamma \vdash A| \triangleq |\Gamma| \to |A|$$

[14]The lower bound $a \wedge b$ (resp. upper bound $a \vee b$) is define as the biggest (resp. lowest) element being lower (resp. bigger) than a and b: $a \wedge b \triangleq \min\{c \in \mathscr{L} : c \leq a \wedge c \leq b\}$. From this definition, it is an easy exercise to prove the expected properties.

Finally, satisfied formulas are defined as formulas whose truth value is $\top$:

$$\mathcal{H} \vDash A \qquad \text{if and only if} \qquad |A| = \top$$

It is easy to check that the rules of propositional logic are all adequate with this interpretation and thus that this indeed defines a model.

Proposition 1.23 (Soundness). *If $\mathcal{H}$ is a Heyting algebra and A a formula, then the provability of A implies its validity in $\mathcal{H}$:*

$$(\vdash A) \quad \Rightarrow \quad (\mathcal{H} \vDash A).$$

But more interestingly, intuitionistic logic has the property of being complete with respect to Heyting algebras. This means that a formula that is satisfied by any Heyting algebra is provable in natural deduction.

Proposition 1.24 (Completeness). *Let A be a formula. If for any Heyting algebra $\mathcal{H}$, A is valid ($\mathcal{H} \vDash A$) then A is provable:*

$$(\forall \mathcal{H}.\mathcal{H} \vDash A) \quad \Rightarrow \quad (\vdash A).$$

As a consequence, to know that a formula A is not provable in intuitionistic logic, it is enough to find one Heyting algebra in which it is not valid. Besides, if there is also one model in which it is valid, then the formula is independent: neither A nor its negation $\neg A$ are provable, and both theories obtained by defining A or its negation are coherent, since they admit a model.

This is for instance the case of the excluded-middle. Indeed, a truth table is a particular case of Heyting algebra reduced to two values $\bot$ and $\top$, so that we already know a model in which $A \vee (\neg A)$ is valid. We can easily construct a Heyting algebra in which it is not valid. Consider the lattice $\{0, 1/2, 1\}$, by definition of $\wedge, \vee, \Rightarrow, \neg$, we get:

$p \wedge q$			
q p	0	$1/2$	1
0	0	0	0
$1/2$	0	$1/2$	$1/2$
1	0	$1/2$	1

$p \vee q$			
q p	0	$1/2$	1
0	0	$1/2$	1
$1/2$	$1/2$	$1/2$	1
1	1	1	1

$p \to q$			
q p	0	$1/2$	1
0	1	1	1
$1/2$	0	1	1
1	0	$1/2$	1

p	$\neg p$
0	1
$1/2$	0
1	0

This defines a Heyting algebra $\mathcal{H}_{1/2}$, where we can observe that $1/2 \vee (\neg 1/2) = 1/2 \vee (1/2 \to 0) = 1/2 \vee 0 = 1/2$, which invalidates the excluded-middle. So that for any formula A mapped to $1/2$, the excluded-middle is not satisfied:

$$\mathcal{H}_{1/2} \nvDash A \vee (\neg A).$$

This concludes the proof of the independence of the excluded-middle from intuitionistic logic.

Last but not least, Heyting algebras also provide a model for first-order (intuitionistic) logic, provided that they are complete as a lattice.

Definition 1.25 (Complete lattice). A lattice $\mathcal{L}$ is said *complete* when every subset A of $\mathcal{L}$ admits a supremum, written $\bigwedge A$, and an infimum, written $\bigvee A$. A Heyting algebra $\mathcal{H}$ is complete if it is complete as a lattice. ⌟

Given a complete Heyting algebra $\mathcal{H}$, it is possible to construct a model for first-order logic. The interpretation of predicates and quantifiers is defined as follows:

- any k-ary predicate $P(x_1, \ldots, x_k)$ is interpreted as a k-ary function $\dot{P} : \mathcal{H}^k \to \mathcal{H}$, so that the formulas with parameters $P(m_1, \ldots, m_k)$ is interpreted by:

$$|P(m_1, \ldots, m_k)| = \dot{P}(m_1, \ldots, m_k)$$

- the universal quantifier $\forall$ is interpreted as the infimum over all possible instantiation of its variable by an element of $\mathcal{H}$:
$$|\forall x.A(x)| = \bigwedge_{m\in\mathcal{H}} |A(m)|$$

- the existential quantifier $\exists$ is interpreted as the supremum over all possible instantiation of its variable by an element of $\mathcal{H}$:
$$|\exists x.A(x)| = \bigvee_{m\in\mathcal{H}} |A(m)|$$

Observe that once again, this definition matches our intuition: $\forall x.A(x)$ is interpreted as an element that is lower (and contains indeed more information) than every possible $A(m)$; when $\exists x.A(x)$ is interpreted as an element higher (and contains indeed less information) than every possible $A(m)$.

1.2.3 Kripke forcing

Kripke models, introduced by Saul Kripke [89, 90], give another semantics for intuitionistic logic. They are quite different of Heyting algebras in that they are not based on a lattice and, most importantly, because the relation of satisfiability is defined in a very different way. Besides, we will use an intuition based on Kripke forcing in Chapter 6 (to define the environment-passing style translation of a classical call-by-need calculus), which also motivates their presentation in this section.

Intuitively, a Kripke model is a universe containing different worlds. Every world contains a specific information, and this information can only be refined in the future of this world. Each world is thus connected to the possible worlds accessible from it, which all contain at least the same information. We shall present another metaphor due to Van Dalen [158] after giving the formal definition of Kripke models.

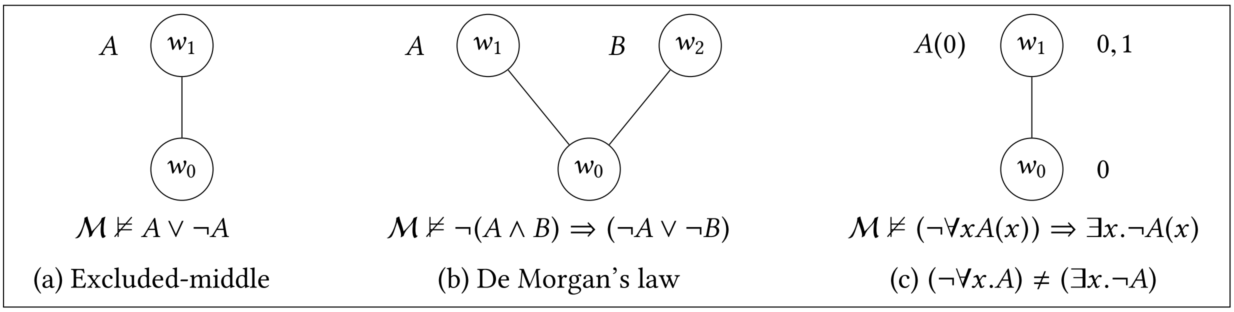
Definition 1.26 (Kripke model). A *Kripke model* is a quadruple $\mathcal{M} = (\mathcal{W}, \leq, D, V)$ where:

- $\mathcal{W}$ is a set of possible worlds,
- $\leq$ is a pre-order and denotes the relation of accessibility between worlds,
- D is a function that maps every world w to the set $D(w)$ of terms defined in it,
- V is a function that maps a k-ary predicate $P(x_1,...,x_k)$ and a world w to the set of tuple $(t_1,...,t_k) \in D(w)^k$ such that $P(t_1,...,t_k)$ is true in w.

The set $\mathcal{W}$ is supposed to be given with a distinguished world $w_0 \in \mathcal{W}$ such that every other world are accessible from it:
$$\forall w' \in \mathcal{W}, w_0 \leq w'$$

Furthermore, D and V are required to be monotonic in the sense that if an element is defined (resp. an atomic formula holds) in a given world w, then it has to be defined in every world w' accessible from w. Formally, for all $w, w' \in W$ and any predicate P:
$$w \leq w' \Rightarrow D(w) \subseteq D(w') \qquad\qquad w \leq w' \Rightarrow V(P,w) \subseteq V(P,w')$$
⌟

Given a Kripke model $\mathcal{M} = (\mathcal{W}, \leq, D, V)$, we define a relation $w \Vdash A$ that denotes the validity of the formula A in the world w. We say that the world w *forces* A and we call $\Vdash$ the *forcing relation*. This



$\mathcal{M} \nvDash A \vee \neg A$

(a) Excluded-middle

$\mathcal{M} \nvDash \neg(A \wedge B) \Rightarrow (\neg A \vee \neg B)$

(b) De Morgan's law

$\mathcal{M} \nvDash (\neg \forall x A(x)) \Rightarrow \exists x.\neg A(x)$

(c) $(\neg \forall x.A) \neq (\exists x.\neg A)$

Figure 1.2: Examples of Kripke counter-models

relation is defined by induction on the structure of formulas:

$$
\begin{array}{lcl}
w \Vdash P(t_1, \ldots t_k) & \triangleq & (t_1, \ldots, t_k) \in V(P, w) \\
w \Vdash A \wedge B & \triangleq & (w \Vdash A) \wedge (w \Vdash B) \\
w \Vdash A \vee B & \triangleq & (w \Vdash A) \vee (w \Vdash B) \\
w \Vdash A \Rightarrow B & \triangleq & \forall w' \geq w. w' \Vdash A \Rightarrow w' \Vdash B \\
w \Vdash \neg A & \triangleq & \forall w' \geq w. w' \nVdash A \\
w \Vdash \forall x. A(x) & \triangleq & \forall w' \geq w. \forall d \in D(w'), w' \Vdash A(d). \\
w \Vdash \exists x. A(x) & \triangleq & \exists d \in D(w), w \Vdash A(d) \\
w \Vdash \Gamma \vdash A & \triangleq & (\forall C \in \Gamma, w \Vdash C) \Rightarrow w \Vdash A
\end{array}
$$

Finally, we say that a model $\mathcal{M}$ satisfies a formula A (resp. a judgment $\Gamma \vdash A$) and write $\mathcal{M} \vDash A$ if and only if $w_0 \Vdash A$.

Remark 1.27. Van Dalen describes Kripke models using a different intuition. Rather than poorly reformulating his point of view, we quote his metaphor as such (see [158, pp.12-13]):

> *The basic idea is to mimic the mental activity of Brouwer's individual, who creates all of mathematics by himself. This idealized mathematician, also called creating subject by Brouwer, is involved in the construction of mathematical objects, and in the construction of proofs of statements. This process takes place in time. So at each moment he may create new elements, and at the same time he observes the basic facts that hold for his universe so far. In passing from one moment in time to the next, he is free how to continue his activity, so the picture of his possible activity looks like a partially ordered set (even like a tree). At each moment there is a number of possible next stages. These stages have become known as possible worlds. Observe that the 'truth' at a node w essentially depends on the future. This is an important feature in intuitionism (and in constructive mathematics, in general). The dynamic character of the universe demands that the future is taken into account. This is particularly clear for ∀. If we claim that "all dogs are friendly", then one unfriendly dog in the future may destroy the claim.*

⌟

This semantics is also sound and complete with respect to intuitionistic logic, and allows to define very simple models that do not satisfy classical principles. We give as examples in Figure 1.2 counter-models for the excluded-middle, the De Morgan's law and the equivalence between $\neg\forall x.A$ and $\exists x.\neg A$. Once again, thanks to the completeness of Kripke models, this is enough to prove that these principles (which all hold in the two-points Heyting algebra) are independent from intuitionistic logic.

1.2.4 The standard model of arithmetic

Lastly, we shall introduce briefly the standard model of arithmetic. This model is defined as the $\mathscr{L}$-structure (where $\mathscr{L}$ refers to the language of arithmetic) whose domain is the set $\mathbb{N}$ of natural numbers and in which each symbol of $\mathscr{L}$ is interpreted canonically (the symbol '0' is interpreted by 0, the symbol 's' by the function $n \mapsto n+1$, and so on). Abusing the notation, this $\mathscr{L}$-structure build on the set $\mathbb{N}$ is itself written $\mathbb{N}$. Formally, to each closed term t of the language $\mathscr{L}$ is associated a natural number $\mathrm{Val}(t)$, called the *value of* t. This value is defined inductively on the structure of t by:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathrm{Val}(0) & \triangleq & 0 & \mathrm{Val}(t+u) & \triangleq & \mathrm{Val}(t)+\mathrm{Val}(u) \\ \mathrm{Val}(s(t)) & \triangleq & \mathrm{Val}(t)+1 & \mathrm{Val}(t\times u) & \triangleq & \mathrm{Val}(t)\,\mathrm{Val}(u) \end{array}$$

and satisfies that for all $n \in \mathbb{N}$, $\mathrm{Val}(\overline{n}) = n$, where $\overline{n} = s^n(0)$. The satisfiability relation $\mathbb{N} \vDash A$ is defined again by induction on the structure of A by:

$$\begin{array}{lcl} \mathbb{N} \vDash t = u & \triangleq & Val(t) = Val(u) \\ \mathbb{N} \nvDash \bot & & \\ \mathbb{N} \vDash A \Rightarrow B & \triangleq & \mathbb{N} \nvDash A \vee \mathbb{N} \vDash B \\ \mathbb{N} \vDash A \wedge B & \triangleq & \mathbb{N} \vDash A \wedge \mathbb{N} \vDash B \\ \mathbb{N} \vDash A \vee B & \triangleq & \mathbb{N} \vDash A \vee \mathbb{N} \vDash B \\ \mathbb{N} \vDash \forall x.A & \triangleq & \text{for all } n \in \mathbb{N},\ \mathbb{N} \vDash A[\overline{n}/x] \end{array}$$

It is easy to show that this indeed defines a model of Peano arithmetic, and in particular that it entails its consistency. Yet, it should be observed that this definition is infinitary, since the interpretation of $\forall x.A$ requires to know the interpretation of $A[n/x]$ for all $n \in \mathbb{N}$. This implies that the meta-theory in which we reason needs to account for mechanisms allowing to construct infinitary objects and to reason on them. For instance, this is not possible within Peano arithmetic, where all the objects are finite natural numbers. Hence Peano arithmetic *a priori* cannot prove its own consistency, at least by this way. Gödel actually closed the problem with his second incompleteness theorem, which states that a consistent theory $\mathscr{T}$ containing (PA) cannot prove its own consistency unless it is inconsistent.

2- The λ-calculus

2.1 The λ-calculus

In the previous section, we introduced the concepts of *logic* and *proofs*. We shall now present the notion of *programs* and *computations* through the so-called λ-calculus. The λ-calculus is indeed to be understood as a minimalistic programming language: on the one hand, it is as powerful as any other programming language, and on the other hand, it is defined by a very simple syntax which makes it very practical to reason with.

The λ-calculus was originally introduced in 1932 by Church [24] with the aim of providing a foundation for logic which would be more natural than Russell's type theory or Zermelo's set theory, and would rather be based on functions[1]. While his formal system turned out to be inconsistent, fundamental discoveries were made at this time on the underlying pure λ-calculus. In particular, it gave a negative answer to Hilbert's long-standing *Entscheidungsproblem* for first-order logic: Church first proved in [26] that the convertibility problem for pure λ-calculus was recursively undecidable, then he deduced that no recursive decision procedure existed for validity in first-order predicate logic [25].

2.1.1 Syntax

The syntax of the λ-calculus is given by the following grammar:

$$t,u ::= x \mid \lambda x.t \mid t\,u$$

Rather than programs, we speak of *λ-terms* or simply of *terms*, and denote by Λ the set of all terms. The three syntactic categories of terms can be understood as follows:

- the term x designates a variable (and is formally taken among an alphabet of variables $\mathcal{V}$), just as the x is a variable in the mathematical expression x^2;
- $\lambda x.t$ is a function waiting for an argument bound by the variable x, where t, the body of the function, is a term depending on x. The working mathematician can think of $\lambda x.t$ as a notation for the function $x \mapsto t(x)$.
- $t\,u$ is the application of the term t to the term u.

While the notations might seem a bit puzzling at first sight, they have the huge benefits of unveiling the idea of *free* and *bound variables*. Consider for instance the term $\lambda x.yx$. The variable x occurs twice, and each occurrence plays a different role: in '$\lambda x.$', x declares the expected parameter x (we speak of *binding occurrence*); in 'yx', x refers to the previously defined parameter (we speak of *bound occurrence*. As it is used to bind variables, the constructor λ is also called a *binder*. Back to our example, unlike the variable x, the variable y occurs freely in the term $\lambda x.yx$. This is formally expressed by the fact that y belongs to the set of free variables of this term, whose definition is given hereafter.

[1] This has the advantage of avoiding the use of free variables, for reasons Church explained in [24, pp. 346–347].

Definition 2.1 (Free variables)**.** The set $FV(t)$ of *free variables* of the λ-term t is defined by induction over the syntax of terms:

$$FV(x) = \{x\} \qquad FV(\lambda x.t) = FV(t)\backslash\{x\} \qquad FV(tu) = FV(t) \cup FV(u)$$

A variable x is said to be *free in t* if $x \in FV(t)$. ⌟

Remark 2.2. We consider application to be left-associative, that is that $t\,u\,r$ abbreviates $(t\,u)\,r$. We also consider that application has precedence over abstraction: $\lambda x.t\,u$ is equivalent to $\lambda x.(\,tu)$. We might sometimes mark application by parentheses $(t)u$ to ease the reading of terms. Finally, we will often use the notation $\lambda xy.t$ as a shorthand for $\lambda x.\lambda y.t$ (and $\lambda xyz.t$ for $\lambda x.\lambda y.\lambda z.t$, *etc*). ⌟

2.1.2 Substitutions and α-conversion

Before going any further, we need to say a word about α-equivalence. Consider for instance the terms $\lambda x.x$ and $\lambda y.y$. As explained before, they correspond respectively to the functions $x \mapsto x$ and $y \mapsto y$, of which any mathematician would say that they are the same. In practice, they are the same up to the renaming of the bound variable x by y. Whenever two terms are the same up to the renaming of bound variables, we say that they are α-equivalent. For instance, the terms $(\lambda x.x)\lambda y.y$ and $(\lambda x.x)\lambda x.x$ are α-equivalent while $\lambda xy.y\,x$ and $\lambda xy.x\,y$ are not. This observation might seem meaningless from a mathematical point of view, since α-equivalent functions represent the same function. But from the point of view of programming language, this is much more subtle since two α-equivalent programs are different syntactic objects. When it is possible we will always reason up to α-equivalence, but we will see in Chapter 6 that it is not always possible to avoid considering this question.

Remark 2.3 (Integrals and α-conversion)**.** The reader inclined towards mathematical analogies can think of integrals as a good example for α-conversion (and binding of variables). For instance, the integrals $\int_0^t f(x)\mathrm{d}x$ and $\int_0^t f(y)\mathrm{d}y$ are the same (α-equivalent) since one can pass from one to the other by renaming the bound variable x in y (or the other way round). ⌟

This being said, we can now speak of *substitution.* Just as we defined it for first-order variables in formulas (Definition 1.6), we need to define the substitution of variables by λ-terms. Once more, substitution is a notion that is often taken for granted in mathematics. For instance, considering the polynomial $P(x) = x^2+3x+1$, $P(2)$ is $P(2) = 2^2+3\times2+1$, that is to say $P(x)$ in which 2 substitutes x, but substitution of a variable by an expression is never properly defined. This is fine as long as substitution is to be performed by human beings, since it is highly intuitive. However, when it comes to computers, this has to be precisely defined.

Definition 2.4 (Substitution)**.** The substitution of a variable x in a term t by u, written $t[u/x]$, is defined inductively on the structure of t by:

$$\begin{array}{rcll} x[u/x] & \triangleq & u & \\ y[u/x] & \triangleq & y & \\ (\lambda y.t)[u/x] & \triangleq & \lambda y.(t[u/x]) & \qquad\qquad (\text{if } x \neq y, y \notin FV(u)) \\ (\lambda x.t)[u/x] & \triangleq & \lambda x.t & \\ (t\,t')[u/x] & \triangleq & (t[u/x])\,(t'[u/x]) & \end{array}$$

⌟

It is worth noting that substitutions of the shape $(\lambda y.t)[u/x]$ are blocked when $y \neq x$ and $y \in FV(u)$. For that matter, since we reason up to α-equivalence and it is clear that $\lambda x.x$ and $\lambda y.y$ are α-equivalent, we can perform $(\lambda y.y)[u/x]$ which is equal to $\lambda y.y$ (*i.e.* $\lambda x.x$).

2.1.3 β-reduction

We said that λ-terms were our model for *programs*, we shall now see how they *compute*. As a matter of fact, computation is quite simple to understand since that it is defined by one unique rule. This rule is called the β-reduction and corresponds to mathematical application of a function to its argument. Consider for instance a polynomial $P(x)$, if you apply a function $x \mapsto P(x)$ to the integer 2, you want to "compute" to $P(2)$, that is $P(x)$ in which x has been substituted by 2. More generally, if you apply $x \mapsto P(x)$ to some term n (think for instance of $n = f(2)$ for some function f), you expect to get $P(n)$ (or $P(f(2))$), that is $P(x)$ in which x has been substituted by n. The β-reduction is defined accordingly: when a function $\lambda x.t$ is applied to a term u, it reduces to $t[u/x]$. This reduction rule is formally written:

$$(\lambda x.t)\,u \xrightarrow{1}_\beta t[u/x]$$

where the 1 denotes the fact that this reduction is performed in one step. The term $(\lambda x.t)\,u$ is called a *redex* since it gives rise to a step of reduction. The full β-reduction, written $\longrightarrow_\beta$, is defined as the contextual and reflexive-transitive closure of this rule:

- first we extend to contextual reduction (*i.e.* reduction within terms):

$$\begin{array}{rclcl} t\,u & \xrightarrow{1}_\beta & t'\,u & \qquad\qquad & (\text{if } \; t \xrightarrow{1}_\beta t' \,) \\ t\,u & \xrightarrow{1}_\beta & t\,u' & & (\text{if } \; u \xrightarrow{1}_\beta u' \,) \\ \lambda x.t & \xrightarrow{1}_\beta & \lambda x.t' & & (\text{if } \; t \xrightarrow{1}_\beta t' \,) \end{array}$$

- second we take the reflexive-transitive closure (*i.e.* consider an arbitrary number of step of reductions):

$$\begin{array}{lclcl} t & \xrightarrow{0}_\beta & u & \triangleq & t = u \\ t & \xrightarrow{n+1}_\beta & u & \triangleq & \exists t' \in \Lambda, t \xrightarrow{1}_\beta t' \wedge t' \xrightarrow{n}_\beta u \\ t & \xrightarrow{*}_\beta & u & \triangleq & \exists n \in \mathbb{N}, t \xrightarrow{n}_\beta u \\ t & \longrightarrow_\beta & u & \triangleq & t \xrightarrow{*}_\beta u \end{array}$$

Remark 2.5 (Contexts)**.** The contextual closure of β-reduction can also be done by defining *evaluation contexts* $C[]$ and by adding the rule:

$$C[t] \xrightarrow{1}_\beta C[t'] \qquad\qquad (\text{ if } t \xrightarrow{1}_\beta t')$$

The contexts corresponding to the full β-reduction are given by the following grammar:

$$C ::= [\,] \mid C\,u \mid t\,C \mid \lambda x.C$$

The use of contexts is a common and useful tool to specify reduction rules. ⌟

Remark 2.6 (Reduction vs. equality)**.** A major difference with mathematics is to be mentioned: if t reduces to u, we do not consider that t is equal to u. To carry on the comparison with mathematics, here we are somehow considering that $2 + 3 \longrightarrow 5$ and not that $2 + 3 = 5$. In other words, computation matters.

Nevertheless, we could still define an equality $=_\beta$ as the symmetric-transitive closure of the full β-reduction $\longrightarrow_\beta$ (or equivalently as the smallest equivalence relation containing $\longrightarrow_\beta$). This equality is usually called β-equivalence. ⌟

Now, let us consider the following λ-terms:

$$\begin{array}{rcl} S & = & \lambda xyz.x\,z\,(y\,z) \\ C & = & \lambda xy.x\,y \\ I & = & \lambda x.x \end{array}$$

and define $t = S\,(I\,C)\,(I\,I)\,(C\,I)$. It is an easy exercise to check that this term reduces to I, and it is interesting to observe that there are different ways to reduce t to obtain the result. This simple observation carries in fact two fundamental concepts: *determinism* and *confluence*.

Definition 2.7 (Determinism)**.** A reduction $\longrightarrow$ is said to be *non-deterministic* if there exists a term t and two terms u, u' such that $u \neq u'$ and $t \xrightarrow{1} u$ and $t \xrightarrow{1} u'$. This situation can be visually represented by:

$$\begin{tikzcd} & t \arrow[dl] \arrow[dr] & \\ u & & u' \end{tikzcd}$$

A reduction is said *deterministic* if it does not admit any such situation. ⌟

Definition 2.8 (Confluence)**.** A reduction $\longrightarrow$ is said to be confluent if whenever for any terms t, u, u' such that $t \longrightarrow u$ and $t \longrightarrow u'$, there exists a term r such that $u \longrightarrow r$ and $u' \longrightarrow r$. Visually, this can be expressed by:

$$\begin{tikzcd} & t \arrow[dl] \arrow[dr] & \\ u \arrow[dr, dashed] & & u' \arrow[dl, dashed] \\ & r & \end{tikzcd}$$

⌟

The full β-reduction is clearly non-deterministic, but it is also confluent. This property is fundamental in order to consider the λ-calculus as a suitable model of computation: it ensures that if an expression may be evaluated in two different ways, both will lead to the same result.

Example 2.9 (Arithmetical operations)**.** Confluence is an obvious property of arithmetical operations. For instance, we could turn the computational axioms (PA1-PA4) of first-order arithmetic into reduction rules:

$$\begin{array}{rcll} 0 + x & \xrightarrow{1} & x & \text{(for all } x \in \mathbb{N}) \\ s(x) + y & \xrightarrow{1} & s(x + y) & \text{(for all } x, y \in \mathbb{N}) \\ 0 \times x & \xrightarrow{1} & 0 & \text{(for all } x \in \mathbb{N}) \\ s(x) \times y & \xrightarrow{1} & (x \times y) + y & \text{(for all } x, y \in \mathbb{N}) \end{array}$$

Then, taking the contextual and transitive closures of this reduction, we can prove that it is confluent. This is nothing more than the well-known fact that to compute the value of an arithmetic expression, one can compute any of its subexpression in any order to get the final result. ⌟

Theorem 2.10 (Confluence)**.** *The β-reduction is confluent.*

Proof. The proof of this result can be found for instance in [10]. □

Finally, the λ-calculus is a model of computation (just like Turing machines) since any computation can be done using its formalism. Of course, this raises the question of defining what is a computation. We will not answer to this question here (there is plenty of literature on the subject), but we should mention that the definition of Turing-completeness is in fact simultaneous to the proof of Turing-completeness of the λ-calculus [154].

Theorem 2.11 (Turing-completeness)**.** *The λ-calculus and Turing machines are equivalent, that is, they can compute the same partial functions from $\mathbb{N}$ to $\mathbb{N}$.*

2.1.4 Evaluation strategies

One way to understand the property of confluence is that whatever the way we choose to perform a computation, it will lead to the same result. Thus we can actually choose any strategy of reduction. Indeed, when it comes to implementation, one has to decide what to do in the case of a critical pair and has roughly three choices: go to the left, go to the right or flip a coin. An *evaluation strategy* is a restriction of the full β-reduction to a deterministic reduction. We will mainly speak of three evaluation strategies in this manuscript, which are respectively called *call-by-name*, *call-by-value* and *call-by-need*. In a nutshell, when applying a function $\lambda x.t$ to a term u (which might itself contain redexes and be reducible):

- the call-by-name strategy will directly substitute x by u to give $t[u/x]$;
- the call-by-value strategy will first reduce u, try to reach a value[2] V and if so, substitute x by V to give $t[V/x]$;
- the call-by-need strategy will substitute x by a shared copy of u, and in the case where u has to be reduced at some point (is "*needed*"), it will reduce it and share the result of the computation.

If you think of a multivariate polynomial $P(x,y)$ where y does not occur, for instance $P(x,y) = 2x^2+x+1$, and you want to compute the result of the application of the function $x \mapsto P(x,y)$ to $2+3$. The call-by-name strategy will perform the substitution and give $2\times(2+3)^2+(2+3)+1$ (and then reduce $2+3$ to 5 twice), while the call-by-value strategy will reduce $2+3$ to 5 and then perform the substitution to give $2\times(5)^2+5+1$. The call-by-need strategy is slightly more subtle and will somehow reduce to a state $2x^2+x+1$ with the information that $x=2+3$. Then, since x is "needed", it will reduce $x=2+3$ to $x=5$, and then finish the computation. When applying the function $y \mapsto P(x,y)$ to $2+3$, since y does not appear in $P(x,y)$, neither the call-by-name nor the call-by-need strategies will compute $2+3$. On the contrary, the call-by-value strategy will compute $2+3$ it anyway before performing the substitution of y by 5 to reduce to the same expression $2x^2+x+1$.

These three evaluation strategies will be discussed more formally in the sequel, so that we delay their formal introduction to Chapter 4 for call-by-name and call-by-value, and to Chapter 5 and 6 for call-by-need.

2.1.5 Normalization

When we evoked the call-by-value evaluation strategy in the previous section, we said that to reduce a redex $(\lambda x.t)\,u$ it would *try* to reduce u to a value. Indeed, it is not always the case that a term reduces to a value, or more generally that a reduction ends. Indeed, consider for instance the following λ-terms:

$$\delta \triangleq \lambda x.x\,x \qquad\qquad \Omega \triangleq \delta\,\delta$$

and try to reduce Ω. You will observe that $\Omega \longrightarrow_\beta \Omega \longrightarrow_\beta \ldots$, so that the reduction never stops and never reaches a value. This terms is said to be *non-terminating*, *non-normalizing* or *diverging*. More surprisingly, if we consider the λ-term $t \triangleq (\lambda xy.x)\,I\,\Omega$, we can observe that if we reduce the rightmost redex in Ω, we will obtain $t \longrightarrow_\beta t \longrightarrow_\beta \ldots$. If we start by reducing the leftmost redex, we will get $(\lambda y.I)\,\Omega$, then we can still reduce it to itself by reducing the redex in Ω, or get I. To sum up, we are in front of the following situation:

[2] The notion of *value* depends on the choice of reduction rules and will be more formally defined in the future. Most of the time, the set of values V is defined by: $V ::= x \mid \lambda x.t$. For the moment, you can think of it as a term that is reduced enough to know how to drive the computation forward: a variable blocks the computation, while a function is demanding an argument.

$$\begin{tikzcd}
(\lambda xy.x)\,I\,\Omega \arrow[rr, "1"] \arrow[dr, "1"] & & (\lambda xy.x)\,I\,\Omega \arrow[rr, "1"] \arrow[dr, "1"] & & (\lambda xy.x)\,I\,\Omega \arrow[rr, dashed] \arrow[dr, "1"] & & \cdots & \\
& (\lambda y.I)\,\Omega \arrow[rr, "1"] \arrow[dr, "1"] & & (\lambda y.I)\,\Omega \arrow[rr, "1"] \arrow[d, "1"] & & (\lambda y.I)\,\Omega \arrow[rr, dashed] \arrow[dll, "1"] & & \cdots \\
& & & I & & & &
\end{tikzcd}$$

which can be compacted into:

$$\begin{tikzcd}
(\lambda xy.x)\,I\,\Omega \arrow[loop above, "n"] \arrow[r, "1"] & (\lambda y.I)\,\Omega \arrow[loop above, "n"] \arrow[r, "1"] & I
\end{tikzcd}$$

In this example, we see that the reduction term t can either loop forever on t or $(\lambda y.I)\,\Omega$, or reduce to I that is not reducible. This term is said to be *weakly normalizing*, because there exists a reduction path which is normalizing, and others which do not terminate.

Definition 2.12 (Normalization)**.** A term t is said to be in *normal form* if it can not be reduced, that is if it does not contain any redex. A reduction path *normalizes* if it ends on a term in normal form. A term is said to be *strongly normalizing* if all its reduction paths normalize. It is called *weakly normalizing* if there is one reduction path which normalizes. ⌟

Example 2.13. The terms I and $I\,I$ are strongly normalizing, the term $(\lambda x.I)\,\Omega$ is weakly normalizing and Ω is not normalizing. ⌟

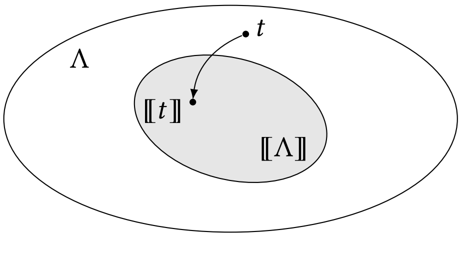
2.1.6 On pureness and side-effects

The λ-calculus is said to be a *purely functional* language. This designation refers to the fact that it behaves like mathematical functions: when computing the application of a function to its arguments, the result of the computation only depends on the arguments. In particular, it does not depend of an exterior state (like a memory cell, the hour or the temperature of the room, etc...). Neither does it modify any such state nor does it write in a file or print things on a screen. As opposed to pure computations, a computation with *side-effects* refers to a computation which modifies something else than its return value. For instance, if we define the following programs in pseudo-code:

```
                        | program bli(a):  | program blo(a):
program bla(a):         |     print(42)    |     b:=a
    return a+2          |     return a+2   |     return a+2
```

then `bla` is a purely functional program, whereas `bli` and `blo` are not. Indeed, `bli` prints 42 and `blo` assigns a value in a global state `b`, and both operations are side-effects.

Even though we explained that any computation could be performed in the formalism of the pure λ-calculus, side-effects are not computable as such. Yet, they can be simulated by means of computational translations. In a few words, for a given effect, there is a corresponding translation $[\![\cdot]\!]$ which embeds the whole λ-calculus Λ into a fragment $[\![\Lambda]\!] \subset \Lambda$ in which everything works like if this side-effect was computable.



$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ (\mathrm{Ax}) \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x\,.\,t : A \to B}\ (\lambda) \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B}\ (@)$$

Figure 2.1: Simply-typed λ-calculus

For instance, for the `print` operation, you can think of a translation such that every term t is translated into a function $[\![t]\!]$ taking a printing function in argument and computing more or less like t. Then, within $[\![\Lambda]\!]$, it becomes possible to use a printing operation since every term has one at hand. Besides, every computation that was possible in Λ is reproducible through the translation in $[\![\Lambda]\!]$, so that the Turing-completeness is not affected.

In Section 8.3, we will present formally the case of continuation-passing style translation which enables us to simulate backtracking operations.

2.2 The simply-typed λ-calculus

If we look closer at the diverging term Ω and try to draw a analogy with a mathematical function, we remark that there is no simple function equivalent to its constituent δ. Indeed, such a function would be $x \mapsto x(x)$ and would require to be given an argument that is both a function and an argument for this function. A way of analyzing more precisely the impossibility is to reason in terms of *types*. The type of a mathematical element is the generic set to which it belongs, for instance $\mathbb{N}$ for a natural number, $\mathbb{R}$ for a real or $\mathbb{N} \to \mathbb{N}$ for a function from integers to integers. Assume for instance that the argument x is of type T. As x is applied to itself, it means that x is also a function of type $T \to U$ for some type U, hence we would have $T = T \to U$. If you think of this in terms of integers and functions, this would require for instance an equality as $\mathbb{N} = \mathbb{N} \to \mathbb{N}$, which does not hold.

The formal idea underlying this intuition is the notion of *simple type*. The grammar of simple types is given by:

$$T, U ::= X \mid T \to U \qquad\qquad (X \in \mathcal{A})$$

where $\mathcal{A}$ is a set of atomic types. An atomic type intuitively represents a base type (as $\mathbb{N}$), while $T \to U$ is the type of functions from T to U.

Definition 2.14 (Type system)**.** A *typing judgment* is triple (Γ, t, T) written $\Gamma \vdash t : T$ which reads *"t has type T in the context Γ"* and where the *typing context* Γ is a list of pairs of the forms $x : T$ (with x a variable and T a type). This hypothesis means that the variable x is assumed of type T. Formally, typing contexts are defined by:

$$\Gamma, \Gamma' ::= \varepsilon \mid \Gamma, x : T$$

where we assume that a variable x occurs at most once in a context Γ. A *type system* allows to assign a type to term by means of *typing rules*, which are simply defined as inference rules whose premises and conclusion are typing judgments, and a typing derivation is a derivation using typing rules. ⌟

Given a type system, we say that a term t is *typable* if there exists a type T such that the typing judgment $\vdash t : T$ is derivable. The *simply-typed λ-calculus* is the restriction of λ-calculus to the set of terms that are typable using the type system described in Figure 2.1.

Remark 2.15 (Untypability of Ω)**.** The typing rules are in one-to-one correspondence with the syntactic categories of the λ-calculus. This implies that the only possible way to type $\delta = \lambda x.xx$ would be along a derivation of this shape:

$$\dfrac{\dfrac{\dfrac{?}{x :?A \vdash x :?C \to ?B}\ (\mathrm{Ax}) \qquad \dfrac{?}{x :?A \vdash x :?C}\ (\mathrm{Ax})}{x :?A \vdash xx :?B}\ (@)}{\vdash \lambda x.xx :?A \to ?B}\ (\lambda)$$

where we mark all the hypothetical types with a question mark. In details, we first would have to introduce an arrow of type $?A \to ?B$ for some types $?A$ and $?B$, resulting in an hypothesis $x :?A$. Then we would necessarily have to type the application $xx :?B$, which requires to type x (the argument) with a type $?C$ and x (the function) with the type $?C \to ?B$. Since the only available hypothesis on x is $x :?A$, this implies that $?C = ?A$ and that $?C = ?A \to ?B$. Since the syntactic equality $?A = ?A \to ?B$ do not hold, this is impossible. Thus δ and Ω are not typable. ⌟

We can check that the type system follows our intuition, since a term $\lambda x.t$ is indeed typed by $T \to U$ provided that the term t is of type U if x is of typed U. Similarly, if t is of type $T \to U$ and u is of type T, then the application $t\,u$ is of type U, just as the application of a function of type $\mathbb{N} \to \mathbb{N}$ to an integer has the type $\mathbb{N}$. However, the fact that a term t has a type $T \to U$ does not mean that it is of the form $\lambda x.t'$. It is rather to be understood as the fact that t can be reduced to a term of this shape. This is stressed by the following fundamental results, that express that typing is preserved through reduction and that typable terms are normalizing.

Proposition 2.16 (Subject reduction). *If t is a term such that $\Gamma \vdash t : T$ for some context Γ and some type T, and if besides $t \longrightarrow_\beta u$ for some term u, then $\Gamma \vdash u : T$.*

Proof. By induction on the reduction rules, it mostly amounts to showing that substitution preserves typing, that if $\Gamma, x : T \vdash t : U$ and $\Gamma \vdash u : T$, then $\Gamma \vdash t[u/x] : U$. The latter is proved by induction on typing rules. □

Theorem 2.17 (Normalization). *If t is a term such that $\Gamma \vdash t : T$ for some context Γ and some type T, then t strongly normalizes.*

Proof. A proof of this result can be found in [12]. We will use very similar ideas in the next chapters to prove normalization properties. □

These two results are crucial when defining a calculus. Subject reduction is sometimes called *type safety*, since it ensures that typability is not affected by reduction. The normalization is also a property of security for a language: it guarantees that any (typed) computation will eventually terminate. This is why these properties will be milestones (or grails, depending on the difficulty of proving them) for the various calculi we study in Chapter 5 to 8.

2.3 The Curry-Howard correspondence

If, hypothetically, one day a reader starts this manuscript without any knowledge of the Curry-Howard correspondence and arrives at this point, she is about to be rewarded by learning something wonderful. The Curry-Howard correspondence is based on a very simple observation [77]. If you compare the following propositional logical rules:

$$\frac{A \in \Gamma}{\Gamma \vdash A}\ (\text{Ax}) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ (\Rightarrow_I) \qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ (\Rightarrow_E)$$

with the typing rules we just defined:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ (\text{Ax}) \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x\,.\,t : A \to B}\ (\lambda) \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B}\ (@)$$

you will observe a striking similarity. The structure of these rules is indeed exactly the same, up to the presence of λ-terms in typing rules. In addition to seeing λ-terms as terms representing mathematical functions, we can thus also consider them as *proof terms*. Take for instance the rule (λ), it can be read: if t is a proof of B under the assumption of a proof x of A, then $\lambda x.t$ is a proof of $A \Rightarrow B$, that is a term waiting for a proof of A to give a proof of B. Similarly, the rule (@) tells us that if t is a proof of $A \Rightarrow B$ and u is a proof of A, then $t\,u$ is a proof of B, which is exactly the principle of *modus ponens*.

Based on this observation, for now on we will identify the two arrow connectives ⇒ and →, and we consider that types are propositional formulas and vice versa. This is schematically represented by the following informal diagram:

$$\begin{tikzcd} \text{Types} \arrow[r, equal] \arrow[d, dash, dashed] & \text{Formulas} \arrow[d, dash, dashed] \\ \lambda\text{-terms} \arrow[r, equal] & \text{Proofs} \end{tikzcd}$$

This correspondence is sometimes also called the *Curry-Howard isomorphism* (since typing rules and logical rules are in one-to-one correspondence) or the *proof-as-program interpretation*. This observation, which is somewhat obvious once we saw it, is actually the cornerstone of modern proof theory. The benefits of this interpretation are two-ways. From proofs to programs, many logical principles can be revisited computationally. A famous example of this is Gödel negative translation which computationally corresponds to continuation-passing style translation (see Section 4.3.2). But the other way round is the more interesting[3]: enrich our comprehension of logic from programming principles. This is one of the motivation to extend this correspondence.

2.4 Extending the correspondence

2.4.1 $\lambda^{\times+}$-calculus

As we saw, the simply-typed λ-calculus is in correspondence with a fragment of propositional logic that is called *minimal logic*. To recover a full interpretation of propositional logic, we need to give a computational content to the connective $\wedge$ and $\vee$. The natural way[4] of doing this is to enrich the calculus with new syntactic constructions which have the expected typing rules. If we consider for example the rules for the conjunction:

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,(\wedge_I) \qquad\qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,(\wedge_E^1) \qquad\qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\,(\wedge_E^2)$$

we see on the introduction rule that the corresponding should be able to compose a proof of A and a proof of B to get a proof of $A \wedge B$. This naturally corresponds to a pair (t,u) of proofs (and to the type $A \times B$), while the elimination rules, allowing to extract a proof of A (or B) from a proof of $A \wedge B$, naturally lead us to the first and second projection π_1 and π_2. We can then extend the syntax to define the $\lambda^{\times}$-calculus (or λ-calculus with pairs):

$$t,u ::= x \mid \lambda x.t \mid t\,u \mid (t,u) \mid \pi_1(t) \mid \pi_2(t)$$

This also induces two new reductions rules when projections (the *destructor*) are applied to a pair (the *constructor*):

$$\pi_1(t,u) \xrightarrow{1}_\beta t \qquad\qquad \pi_2(t,u) \xrightarrow{1}_\beta u$$

[3] For this reason, we prefer the name of *proof-as-program* correspondence.

[4] We will see in the next chapter (Section 3.3.1.1) that another solution consists in encoding the connective in the logic and transporting this encoding to λ-terms. In the case of conjunction, this corresponds to the usual encodings of pairs and projections: $(t,u) \triangleq \lambda x.xtu$, $\pi_1(t) \triangleq \lambda xy.x$ and $\pi_2(t) \triangleq \lambda xy.y$.

and the following typing rules:

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash (t,u) : A \times B}\ (\wedge_I) \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_1(t) : A}\ (\wedge_E^1) \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_2(t) : B}\ (\wedge_E^2)$$

Similarly, we can add *pattern-matching* to meet the disjunction rules. This consists again in three steps. First, we extend the syntax with left and right injections $\iota_1(t)$ and $\iota_2(t)$ and pattern matching $\texttt{match}\ t\ \texttt{with}\ [x \mapsto u_1 \mid y \mid u_2]$:

$$t,u ::= \cdots \mid \iota_1(t) \mid \iota_2(t) \mid \texttt{match}\ t\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_2]$$

Second, we define the reduction rules for the case where we apply the disjunctive destructor to one of the constructor:

$$\begin{array}{l} \texttt{match}\ \iota_1(t)\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_2] \xrightarrow{1}_\beta u_1[t/x] \\ \texttt{match}\ \iota_2(t)\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_2] \xrightarrow{1}_\beta u_2[t/x] \end{array}$$

Last, we add the expected typing rules:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \iota_1(t) : A + B}\ (\iota_1) \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \iota_2(t) : A + B}\ (\iota_2)$$

$$\frac{\Gamma \vdash t : A + B \quad \Gamma, x_1 : A \vdash u_1 : C \quad \Gamma, x_2 : B \vdash u_2 : C}{\Gamma \vdash \texttt{match}\ t\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_2] : C}\ (\texttt{match})$$

The resulting calculus, called the $\lambda^{\times,+}$-calculus, still satisfies the property of subject reduction and typable terms are also normalizing. We have thus extended the matching of types and formulas to conjunction and disjunction, to obtain the following correspondence:

Types	Formulas
$\rightarrow$	$\Rightarrow$
$\times$	$\wedge$
$+$	$\vee$

2.4.2 Entering the cube

Up to now, we stressed the link between the simply-typed λ-calculus and minimal logic, and between the $\lambda^{\times,+}$-calculus and propositional logic. We can actually add some entries to our table of correspondence for other logical systems:

Calculus	Logical system
simply-typed λ-calculus	minimal logic
$\lambda^{\times+}$ -calculus	propositional logic
$\lambda\Pi$ -calculus	first-order logic
System F	second-order logic

We will not introduce formally the $\lambda\Pi$-calculus or System F (which is also referred to as $\lambda 2$) at this stage. We mention them, amongst others, because we will use variants of these calculi in the next chapters, and more importantly to give an overview of the possible flavors of extensions for the simply-typed λ-calculus.

The λ-cube, introduced by Barendregt [11], presents a broader set of calculi extending the simply-typed calculus:

$$\begin{tikzcd}
& \lambda\omega \arrow[rr] & & \lambda\Pi\omega \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda\Pi 2 \arrow[ur] & \\
& \lambda\underline{\omega} \arrow[uu] \arrow[rr] & & \lambda\Pi\underline{\omega} \arrow[uu] \\
\lambda{\to} \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda\Pi \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

On each axis of the cube is added a new form of abstraction:

- the vertical axis adds the dependency of terms in types,
- the horizontal axis adds the dependency of types in terms,
- the last axis adds the dependency of types in types.

For instance, terms of the $\lambda 2$-calculus can take a type in argument, making the calculus polymorphic. Roughly, this means that we can generalize the simply-typed identity $\lambda x.x$ of type $\mathbb{N} \to \mathbb{N}$ or $A \to A$ into a term of type $\forall X.X \to X$ (where X is an abstraction over types). On the opposite, types of the $\lambda\Pi$-calculus can depend on a term, allowing intuitively the definition of a type $\mathrm{Vect}(n)$ of "tuple of integers of size n" and of a term of type $\forall n.\mathrm{Vect}(n)$.

2.4.3 Classical logic

A notable example of extension in the proof-as-program direction is due to Griffin in the early 90s [62]. He discovered that the control operator `call/cc` (for *call with current continuation*) of the Scheme programming language could be typed with Peirce's law:

$$\frac{}{\vdash \texttt{call/cc} : ((A \to B) \to A) \to A}\ (\mathrm{cc})$$

In particular, this typing rule is sound with respect to the computational behavior of `call/cc`, which allows terms for backtracking. We leave detailed explanations about this fact for the next chapter (Section 3.2), but this discovery was essential to mention already in this chapter.

Indeed, as Peirce's law implies, in an intuitionistic framework, all the other forms of classical reasoning (see Section 1.1.2.1), this discovery opened the way for a direct computational interpretation of classical proofs. But most importantly, this lead to a paradigmatic shift from the point of view of logic. Instead of trying to get an axiom by means of logical translations (*e.g.* negative translation for classical reasoning), and then transfer this translation to program along the Curry-Howard correspondence (*e.g.* continuation-passing style for negative translation), one could rather try to add an operator whose computational behavior is adequate with the expected axiom. This is one of the underlying motto of Krivine classical realizability that we will present in the next chapter.

In the spirit of the Curry-Howard correspondence, if an extension of the λ-calculus is to bring more logical power, it should come thanks to more computational power. This is for instance the case of side-effects (such that backtracking, addition of a store, exceptions, etc...), that the pure λ-calculus does not handle directly. So that we can add the following entry in the proof-as-program Rosetta Stone[5]:

Computation	Logic
side-effects	new reasoning principles

[5] We do plead guilty to stealing the Rosetta Stone from Pédrot's PhD thesis [133].

This thesis is in line with this perspective. Half of if (Part II) is precisely dedicated to the study of a calculus which, by the use of side-effects and extension of the λ-calculus, allows to derive a proof of the axiom of dependent choice. The other half (Part III) is devoted to the study of algebraic models which arise from the interpretation of logic through classical realizability.

3- Krivine's classical realizability

This chapter aims at being a survey on Krivine's classical realizability. Our intention is twofold. On the one hand, we recall in broad lines the key definitions of Krivine's classical realizability, and we take advantage of this to introduce some techniques that we use in the sequel of this thesis. On the other hand, we present standard applications of Krivine realizability to the study of the computational content of classical proofs and to models theory. These applications are again loosely introduced, with references pointing to articles where they are presented more in details. Nonetheless, we hope that this overview justifies our interest in the topic and in particular the third part of this manuscript, which is dedicated to the study of algebraic structures for Krivine realizability.

3.1 Realizability in a nutshell

3.1.1 Intuitionistic realizability

The very first ideas of realizability are to be found in the Brouwer-Heyting-Kolmogoroff (BHK) interpretation, which was in fact anterior to its actual formulation, done independently by Heyting for propositional logic [72] and Kolmogoroff for predicate logic [88]. The BHK-interpretation gives the meaning of a statement A by explaining what constitutes an evidence[1] while 'evidence of A' for logically compound A is explained by giving evidences of its constituents. For propositional logic:

1. a evidence of $A \wedge B$ is given by presenting a evidence of A and a evidence of B;
2. a evidence of $A \vee B$ is given by presenting either a evidence of A or a evidence of B (plus the stipulation that this evidence is presented as evidence for $A \vee B$);
3. a evidence of $A \to B$ is a construction which transforms any evidence of A into a evidence of B;
4. absurdity $\bot$ (contradiction) has no evidence; a evidence of $A \to \bot$ is a construction which transforms any evidence of A into a evidence of $\bot$.

In this definition, notions such as *"construction"*, *"transformation"* or the very notion of *"evidence"* can be understood in different ways, and indeed they have been. Intuitionistic realizability can precisely be viewed as the replacement of the notion of *"evidence"* by the formal notion of *"realizer"*, which, again, can be defined in different ways. The original presentation of realizability, due to Kleene [87], define realizers as computable functions. Each function φ is in fact identified to its Gödel's number[2] n, and *"transformation"* is defined by means of function application. Kleene's definition can be reformulated[3] as follows:

[1] We voluntarily use the terminology of "evidence" instead of "proof", to which we already gave a syntactic meaning. Besides, if we regard the BHK-interpretation of propositions with the λ-calculus in mind, we observe that evidences of A correspond to "values" of type A rather than "proofs".

[2] In practice, any other enumeration of computable functions do the job just as well, that is to say that encoding is irrelevant to the principle of Kleene's realizability.

[3] In the original presentation, a pair (n, m) is encoded by its Gödel's number $2^n 3^m$, $\texttt{left}(n)$ is the pair $(0, n)$ and $\texttt{right}(m)$ is the pair $(1, m)$.

1. 0 realizes $\top$;
2. if n realizes A and m realizes B, then the pair (n,m) realizes $A \wedge B$;
3. if n realizes A, then $\mathtt{left}(n)$ realizes $A \vee B$, and similarly, $\mathtt{right}(m)$ realizes $A \vee B$ if m realizes B;
4. the function φ_n realizes $A \to B$ if for any m realizing A, $\varphi_n(m)$ realizes B;
5. a realizer of $\neg A$ is a function realizing $A \to \bot$.

This definition can be revisited using the $\lambda^{\times +}$-calculus extended with natural numbers as the language for computable functions. We do not describe formally this calculus here[4], but only assume that the calculus contains a term $\bar{n}$ for each natural number n. We give the interpretation for first-order arithmetic formulas (see Example 1.3).

1. $t \Vdash \top$ if $t \xrightarrow{*} 0$;
2. $t \Vdash e_1 = e_2$ if $e_1^{\mathbb{N}} = e_2^{\mathbb{N}}$ and $t \xrightarrow{*} 0$;
3. $t \Vdash A \wedge B$ if $t \xrightarrow{*} (t_1, t_2)$ such that $t_1 \Vdash A$ and $t_2 \Vdash B$;
4. $t \Vdash A \vee B$ if $t \xrightarrow{*} \iota_1(u)$ and $u \Vdash A$, or if $t \xrightarrow{*} \iota_2(u)$ and $u \Vdash B$;
5. $t \Vdash A \to B$ if for any $u \Vdash A$, $tu \Vdash B$;
6. $t \Vdash \neg A$ if $t \Vdash A \to \bot$;
7. $t \Vdash \forall x.A$ if for any n, $t\,\bar{n} \Vdash A(n)$;
8. $t \Vdash \exists x.A$ if $t \xrightarrow{*} (\bar{n}, u)$ and $u \Vdash A(n)$.

where $e^{\mathbb{N}}$ is the valuation of the first-order expression e in the standard model $\mathbb{N}$ (see Section 1.2.4).

The main observation is that this definition is purely computational, as opposed to the syntactic definition of typing. In fact, it is a strict generalization of typing in the sense that it can be shown that a term of type A is a realizer of A: this is the property of *adequacy*. One of the consequence of the computational definition is that the relation $t \Vdash A$ is undecidable: given a term t and a formula A, there is no algorithm deciding whether t is a realizer of A. This is again to be opposed with the typing relation.

If this interpretation has shown to be fruitful over the years[5], it is intrinsically bound to intuitionistic logic and incompatible with an extension to classical logic. Indeed, Kleene's realizability takes position against the excluded-middle, as shown by the following proposition:

Proposition 3.1. *There exists a formula H such that the negation of $\forall x(H(x) \vee \neg H(x))$ is realized.*

Proof. Consider the primitive recursive function $h : \mathbb{N}^2 \to N$ defined by:

$$h(n,k) = \begin{cases} 1 & \text{if the } n^{\text{th}} \text{ Turing machine stops after } k \text{ steps} \\ 0 & \text{otherwise} \end{cases}$$

and define the formula $H(x) \triangleq \exists y.(h(x,y) = 1)$, also called *halting predicate*. Assume now that there is a term t realizing the formula $\forall x.(H(x) \vee \neg H(x))$ and define $u \triangleq \lambda n.\mathtt{match}\ t\, n\ \mathtt{with}\ [x \mapsto 1 \mid y \mapsto 0]$. Then, for any $n \in \mathbb{N}$:

[4] You can think of the syntax and reduction rules of the (untyped) $\lambda^{\times +}$-calculus (Section 2.4.1) extended with terms $0, S, \mathtt{rec}$ standing for zero, the successor and a recursion operator. The $\mathtt{rec}$ operator can be defined in various way, the point being that it allows to perform recursion over natural numbers. For instance, it could be given the following reduction rules :

$$\begin{array}{rcl} \mathtt{rec}\ 0\, t_0\, t_S & \to & t_0 \\ \mathtt{rec}\,(S\,u)\, t_0\, t_S & \to & t_S\, u\,(\mathtt{rec}\ u\, t_0\, t_S) \end{array}$$

Formally, this can also be seen as a fragment of PCF [137].

[5] See for instance Van Oosten's historical essay [159] on this topic.

1. either $t\bar{n} \xrightarrow{*} \iota_1(t')$ in which case $u\,\bar{n} \xrightarrow{*} 1$ and $H(n)$ is realized (by t'), *i.e.* the n^{th} Turing machine halts,
2. either $t\bar{n} \xrightarrow{*} \iota_2(t')$ in which case $u\,\bar{n} \xrightarrow{*} 0$ and $\neg H(n)$ is realized (by t'), *i.e.* the n^{th} Turing machine does not halt.

Thus u decides the halting problem, which is absurd. As a consequence, there is no such t, and in particular, any term realizes the formula $\neg(\forall x(H(x) \vee \neg H(x)))$. □

3.1.2 Classical realizability

To address the incompatibility of Kleene's realizability with classical reasoning, Krivine introduced in the middle of the 90s the theory of *classical realizability* [97], which is a complete reformulation[6] of the very principles of realizability to make them compatible with classical reasoning. Although it was initially introduced to interpret the proofs of classical second-order arithmetic, the theory of classical realizability can be scaled to more expressive theories such as Zermelo-Fraenkel set theory [93] or the calculus of constructions with universes [117].

This theory has shown in the past twenty years to be a very powerful framework, both as a tool to analyze programs and as a way to build new models of set theory. We shall now present briefly these aspects before introducing formally Krivine classical realizability.

3.1.2.1 A powerful tool to reason on programs

Krivine realizability, in what concerns the analysis of programs, can be understood as a relaxation of the Curry-Howard isomorphism. As a proof-as-program correspondence, it is indeed more flexible in that it includes programs that are correct with respect to the execution, but that are not typable. In other words, given a formula A and a problem t, when the Curry-Howard isomorphism somewhat said that t is a *proof* of A if its *syntax* matches the structure of A; Krivine realizability rather has for slogan:

if t *computes* correctly, then it is a *realizer.*

For instance, the following dummy program:

```
program dummy(n):
  if n=n+1 then { return 'Hello'} else { return 27 }
```

can not be simply typed with Nat → Nat while this program has the computational behavior that is expected from this type: when applied to a natural number, it always returns the natural number 27.

If this example is easy to understand, it is quite arbitrary and does not bring any interesting perspective. Yet they are more interesting cases, for instance the term of Maurey $M_{a,b}$. This term, defined by:

$$M_{a,b} \triangleq \lambda nm.n\,F\,(\lambda x.a)\,(m\,F\,(\lambda x.b))$$

where $F \triangleq \lambda fg.gf$ and a,b are free variables, decides which of two natural numbers is the smaller. Indeed, when applied to the Church numerals $\overline{n}$ and $\overline{m}$, $M_{a,b}\,\overline{n}\,\overline{m}$ reduces[7] to a if $n \leq m$ and to b if $\overline{m} < \overline{n}$. In particular, if $\mathtt{tt}$ and $\mathtt{ff}$ are the Boolean term for *true* and *false*, $M_{\mathtt{tt},\mathtt{ff}}$ reduces to $\mathtt{tt}$ if $n \leq m$ and to $\mathtt{ff}$ otherwise. Following our realizability motto, since the term $M_{\mathtt{tt},\mathtt{ff}}$ computes the formula *"n is lower than m"*, *a fortiori* it should realize it [8]. However, as shown by Krivine [91], it can not be typed

[6] As observed in several articles [129, 118], classical realizability can in fact be seen as a reformulation of Kleene's realizability through Friedman's A-translation [53].

[7] We recall that the Church numeral $\overline{n}$ is defined by $\lambda fx.f^n x$: $\overline{0} = \lambda fx.x$, $\overline{1} = \lambda fx.fx$, $\overline{2} = \lambda fx.f(fx)$, etc... The verification of the statement is a pleasant exercise of λ-calculus.

[8] This claim can be formalized with a clever definition of the realized formula, and is a nice (but tricky) exercise of realizability.

in Peano second-order arithmetic (or System F), which is the language of Krivine realizability. This illustrates perfectly the fact that realizability includes strictly more programs (and not only dummy ones) than just typed programs.

As we will see in the next sections, the definition of Krivine realizability interpretation of formulas is again purely computational, and thus the relation of $t \Vdash A$ is also undecidable. Worse, the computational analysis of programs is harder than in the intuitionistic case because of the `call/cc` operator which enables programs to backtrack. Even though, Krivine realizability has shown to be a powerful tool to prove properties on the computational behavior of programs. In particular, the adequacy of the interpretation (with respect to typing rules) gives for free the normalization of typed terms. Besides, the computational content of a realizer can be specified by means of a game-theoretic interpretation, but we will come back to this in Section 3.5.2.

3.1.2.2 Terms as semantics

As in intuitionistic realizability, every formula A is interpreted in classical realizability as a set $|A|$ of programs called the *realizers* of A, that share a common computational behavior determined by the structure of the formula A. This point of view is related to the point of view of deduction (and of typing) via the property of *adequacy*, that expresses that any program of type A realizes the formula A, and thus has the computational behavior expected from the formula A.

However the difference between intuitionistic and classical realizability is that in the latter, the set of realizers of A is defined indirectly, that is from a set $\|A\|$ of execution contexts (represented as argument stacks) that are intended to challenge the truth of A. Intuitively, the set $\|A\|$ (which we shall call the *falsity value of* A) can be understood as the set of all possible counter-arguments to the formula A. In this framework, a program realizes the formula A—i.e. belongs to the *truth value* $|A|$—if and only if it is able to defeat all the attempts to refute A by a stack in $\|A\|$. Another difference with the intuitionistic setting resides in the classical notion of a realizer whose definition is parameterized by a *pole*, which represents a particular sets of challenges and that we shall define and discuss in Section 3.4.1.1.

We shall discuss the underlying game-theoretic intuition more in depth at the end of this chapter (Section 3.5.2.2), and say a word about some surprisingly new model-theoretic perspectives brought by this semantics (Section 3.5.3).

3.1.2.3 Modular implementation of logic

As we advocated in the previous chapter (Section 2.4.3), the proofs-as-programs interpretation of logic suggests that any logical extension should be made through an extension of the programming language. Krivine classical realizability precisely follows this slogan, since classical logic is obtained through the λ_c-calculus which is an extension of the λ-calculus with the `call/cc` operator. Much more than that, as we shall explain in Section 3.2.3, the λ_c-calculus is modular in essence and really turns the motto:

"With side-effects come new reasoning principles."

into a general recipe: to extend the logic with an axiom A, one should add an extra instruction with the adequate reduction rules, and give it the type A. If the computational behavior is indeed correct with respect to A, then the typing rules will automatically be adequate with respect to the realizability interpretation. This is for instance the methodology followed by Krivine to obtain a realizer of the axiom of dependent choice with the `quote` instruction, [94].

3.2 The λ_c-calculus

3.2.1 Terms and stacks

The λ_c-calculus distinguishes two kinds of syntactic expressions: *terms*, which represent programs, and *stacks*, which represent evaluation contexts. Formally, terms and stacks of the λ_c-calculus are defined from three auxiliary sets of symbols, that are pairwise disjoint:

1. A denumerable set $\mathcal{V}_\lambda$ of λ-variables (notation: x, y, z, etc.)
2. A countable set C of instructions, which contains at least an instruction cc (denoting 'call/cc', for: *call with current continuation*).
3. A nonempty countable set $\mathcal{B}$ of stack constants, also called stack bottoms (notation: α, β, γ, etc.)

The syntax of terms, stacks and processes is given by the following grammar:

$$\begin{array}{llcll}
\textbf{Terms} & t,u & ::= & x \mid \lambda x.t \mid tu \mid \mathbf{k}_\pi \mid \kappa & x, \in \mathcal{V}_\lambda, \kappa \in C \\
\textbf{Stacks} & \pi & ::= & \alpha \ \mid\ t \cdot \pi & (\alpha \in \mathcal{B}, t \text{ closed}) \\
\textbf{Processes} & p,q & ::= & t \star \pi & (t \text{ closed})
\end{array}$$

As usual, terms and stacks are considered up to α-conversion and we denote by $t[u/x]$ the term obtained by replacing every free occurrence of the variable x by the term u in the term t, possibly renaming the bound variables of t to prevent name clashes. The sets of all closed terms and of all (closed) stacks are respectively denoted by Λ and Π.

Definition 3.2 (Proof-like terms). – We say that a λ_c-term t is *proof-like* if t contains no continuation constant $\mathbf{k}_\pi$. We denote by PL the set of all proof-like terms. ⌟

Finally, every natural number $n \in \mathbb{N}$ is represented in the λ_c-calculus as the closed proof-like term $\overline{n}$ defined by

$$\overline{n} \ \equiv\ \overline{s}^n\overline{0} \ \equiv\ \underbrace{\overline{s}(\cdots(\overline{s}}_{n}\,\overline{0})\cdots)\,,$$

where $\overline{0} \equiv \lambda xf\,.\,x$ and $\overline{s} \equiv \lambda nxf\,.\,f(nxf)$ are Church's encodings of zero and the successor function in the pure λ-calculus. Note that this encoding slightly differs from the traditional encoding of numerals in the λ-calculus, although the term $\overline{n} \equiv \overline{s}^n\overline{0}$ is clearly β-convertible to Church's encoding $\lambda xf\,.\,f^n x$—and thus computationally equivalent. The reason for preferring this modified encoding is that it is better suited to the call-by-name discipline of Krivine's Abstract Machine (KAM) we will now present.

3.2.2 Krivine's Abstract Machine

In the λ_c-calculus, computation occurs through the interaction between a closed term and a stack within Krivine's Abstract Machine (KAM). Before turning into a central piece of classical realizability, this abstract machine was a very standard tool to implement (call-by-name) λ-calculus [96]. Formally, we call a *process* any pair $t \star \pi$ formed by a closed term t and a stack π. The set of all processes is written $\Lambda \star \Pi$ (which is just another notation for the Cartesian product of Λ by Π).

Definition 3.3 (Relation of evaluation). We call a relation of *one step evaluation* any binary relation $\succ_1$ over the set $\Lambda \star \Pi$ of processes that fulfills the following four axioms:

$$\begin{array}{lrcl}
(\textsc{Push}) & tu \star \pi & \succ_1 & t \star u \cdot \pi \\
(\textsc{Grab}) & (\lambda x\,.\,t) \star u \cdot \pi & \succ_1 & t[u/x] \star \pi \\
(\textsc{Save}) & \mathrm{cc} \star t \cdot \pi & \succ_1 & t \star \mathbf{k}_\pi \cdot \pi \\
(\textsc{Restore}) & \mathbf{k}_\pi \star t \cdot \pi' & \succ_1 & t \star \pi
\end{array}$$

The reflexive-transitive closure of $\succ_1$ is written $\succ$. ⌟

One of the specificities of the λ_c-calculus is that it comes with a binary relation of (one step) evaluation $\succ_1$ that is not *defined*, but *axiomatized* via the rules (Push), (Grab), (Save) and (Restore). In practice, the binary relation $\succ_1$ is simply another parameter of the definition of the calculus, just like the sets C and $\mathcal{B}$. Strictly speaking, the λ_c-calculus is not a particular extension of the λ-calculus, but a family of extensions of the λ-calculus parameterized by the sets $\mathcal{B}$, C and the relation of one step evaluation $\succ_1$. (The set $\mathcal{V}_\lambda$ of λ-variables—that is interchangeable with any other denumerable set of symbols—does not really constitute a parameter of the calculus.)

3.2.3 Adding new instructions

The main interest of keeping open the definition of the sets $\mathcal{B}$, C and of the relation evaluation $\succ_1$ (by axiomatizing rather than defining them) is that it makes possible to enrich the calculus with extra instructions and evaluation rules, simply by putting additional axioms about C, $\mathcal{B}$ and $\succ_1$. On the other hand, the definitions of classical realizability [97] as well as its main properties do not depend on the particular choice of $\mathcal{B}$, C and $\succ_1$, although the fine structure of the corresponding realizability models is of course affected by the presence of additional instructions and evaluation rules. Standard examples of extra instructions in the set C are:

1. The instruction quote, which comes with the evaluation rule

   (Quote) $$\mathtt{quote} \star t \cdot \pi \ \succ_1 \ t \star \overline{n}_\pi \cdot \pi\,,$$

   where $\pi \mapsto n_\pi$ is a recursive injection from Π to $\mathbb{N}$. Intuitively, the instruction quote computes the 'code' n_π of the stack π, and passes it (using the encoding $n \mapsto \overline{n}$ described in Section 3.2.1) to the term t. This instruction was originally introduced to realize the axiom of dependent choices [94].

2. The instruction eq, which comes with the evaluation rule

   (Eq) $$\mathsf{eq} \star t_1 \cdot t_2 \cdot u \cdot v \cdot \pi \ \succ_1 \ \begin{cases} u \star \pi & \text{if } t_1 \equiv t_2 \\ v \star \pi & \text{if } t_1 \not\equiv t_2 \end{cases}$$

   Intuitively, the instruction eq tests the syntactic equality of its first two arguments t_1 and t_2 (up to α-conversion), giving the control to the next argument u if the test succeeds, and to the second next argument v otherwise. In presence of the quote instruction, it is possible to implement a closed λ_c-term eq′ that has the very same computational behavior as eq, by letting

   $$\mathsf{eq}' \ \equiv \ \lambda x_1 x_2 \,.\, \mathtt{quote}\,(\lambda n_1 y_1 \,.\, \mathtt{quote}\,(\lambda n_2 y_2 \,.\, \mathsf{eq\_nat}\, n_1\, n_2)\, x_2)\, x_1\,,$$

   where eq_nat is any closed λ-term that tests the equality between two numerals (using the encoding $n \mapsto \overline{n}$).

3. The instruction stop, which comes with no evaluation rule. The only purpose of this instruction is to stop evaluation; the contents of the facing stack is implicitly the result of the computation. This instruction turns out to be very useful for witness extraction procedures [118].

4. The instruction ⫛ ('fork'), which comes with the two evaluation rules

   (Fork) $$⫛ \star t_0 \cdot t_1 \cdot \pi \succ_1 t_0 \star \pi \qquad \text{and} \qquad ⫛ \star t_0 \cdot t_1 \cdot \pi \succ_1 t_1 \star \pi\,.$$

   Intuitively, the instruction ⫛ behaves as a non deterministic choice operator, that indifferently selects its first or its second argument. The main interest of this instruction is that it makes evaluation non deterministic, in the following sense:

Definition 3.4 (Deterministic evaluation)**.** We say that the relation of evaluation $\succ_1$ is *deterministic* when the two conditions $p \succ_1 p'$ and $p \succ_1 p''$ imply $p' \equiv p''$ (syntactic identity) for all processes p, p' and p''. Otherwise, $\succ_1$ is said to be *non deterministic.* ⌟

The smallest relation of evaluation, that is defined as the union of the four rules (Push), (Grab), (Save) and (Restore), is clearly deterministic. The property of determinism still holds if we enrich the calculus with an instruction eq together with the aforementioned evaluation rules or with the instruction quote.

On the other hand, the presence of an instruction ⫛ with the corresponding evaluation rules definitely makes the relation of evaluation non deterministic.

3.2.4 The thread of a process and its anatomy

Given a process p, we call the *thread* of p and write $\mathbf{th}(p)$ the set of all processes p' such that $p \succ p'$:

$$\mathbf{th}(p) \;=\; \{p' \in \Lambda \star \Pi \;:\; p \succ p'\}\,.$$

This set has the structure of a finite or infinite (di)graph whose edges are given by the relation $\succ_1$ of one step evaluation. In the case where the relation of evaluation is deterministic, the graph $\mathbf{th}(p)$ can be either:

1. *Finite and cyclic from a certain point*, because the evaluation of p loops at some point. A typical example is the process $\mathbf{I} \star \delta\delta \cdot \alpha$ (where $\mathbf{I} \equiv \lambda x\,.\,x$ and $\delta \equiv \lambda x\,.\,xx$), that enters into a 2-cycle after one evaluation step:

$$\mathbf{I} \star \delta\delta \cdot \alpha \;\succ_1\; \delta\delta \star \alpha \;\succ_1\; \delta \star \delta \cdot \alpha \;\succ_1\; \delta\delta \star \alpha \;\succ_1\; \cdots$$

2. *Finite and linear*, because the evaluation of p reaches a state where no more rule applies. For example:

$$\mathbf{II} \star \alpha \;\succ_1\; \mathbf{I} \star \mathbf{I} \cdot \alpha \;\succ_1\; \mathbf{I} \star \alpha\,.$$

3. *Infinite and linear*, because p has an infinite execution that never reaches twice the same state. A typical example is given by the process $\delta'\delta' \star \alpha$, where $\delta' \equiv \lambda x\,.\,x\,x\,\mathbf{I}$:

$$\delta'\delta' \star \alpha \;\succ_3\; \delta'\delta' \star \mathbf{I} \cdot \alpha \;\succ_3\; \delta'\delta' \star \mathbf{I} \cdot \mathbf{I} \cdot \alpha \;\succ_3\; \delta'\delta' \star \mathbf{I} \cdot \mathbf{I} \cdot \mathbf{I} \cdot \alpha \;\succ_3\; \cdots$$

3.3 Classical second-order arithmetic

In Section 3.2 we focused on the *computing facet* of the theory of classical realizability. In this section, we will now present its *logical facet* by introducing the language of classical second-order logic with the corresponding type system. In Section 3.3.3, we will deal with the particular case of *second-order arithmetic* and present its axioms.

3.3.1 The language of second-order logic

The language of second-order logic distinguishes two kinds of expressions: *first-order expressions* representing individuals, and *formulas*, representing propositions about individuals and sets of individuals (represented using second-order variables as we shall see below).

3.3.1.1 First-order expressions and formulas

First-order expressions are formally defined as in first-order arithmetic (see Example 1.3) from

1. a *first-order signature* Σ which we assume to contain a constant symbol 0 ('zero'), a unary function symbol s ('successor') as well as a function symbol f for every primitive recursive function (including symbols $+$, $\times$, etc.), each of them being given its standard interpretation in $\mathbb{N}$ (see Section 3.3.3).
2. A denumerable set $\mathcal{V}_1$ of *first-order variables*. For convenience, we shall still use the lowercase letters x, y, z, etc. to denote first-order variables, but these variables should not be confused with the λ-variables introduced in Section 3.2.

This results in the following formal definition:

$$\textbf{First-order terms} \qquad e_1, e_2 \ ::= \ x \mid f(e_1, \dots, e_k) \qquad\qquad (x \in \mathcal{V}_1, f \in \Sigma)$$

The set $FV(e)$ of all (free) variables of a first-order expression e is defined as expected, as well as the corresponding operation of substitution (see Definitions 1.5 and 1.6).

Formulas of second-order logic are defined from an additional set of symbols $\mathcal{V}_2$ of *second-order variables* (or *predicate variables*), using the uppercase letters X, Y, Z, etc. to represent such variables:

$$\textbf{Formulas} \qquad A, B \ ::= \ X(e_1, \dots, e_k) \mid A \to B \mid \forall x.A \mid \forall X.A \qquad\qquad (X \in \mathcal{V}_2)$$

We assume that each second-order variable X comes with an arity $k \geq 0$ (that we shall often leave implicit since it can be easily inferred from the context), and that for each arity $k \geq 0$, the subset of $\mathcal{V}_2$ formed by all second-order variables of arity k is denumerable.

Intuitively, second-order variables of arity 0 represent (unknown) propositions, unary predicate variables represent predicates over individuals (or *sets* of individuals) whereas binary predicate variables represent binary relations (or sets of pairs), etc.

The set of free variables of a formula A is written $FV(A)$. (This set may contain both first-order and second-order variables.) As usual, formulas are identified up to α-conversion, neglecting differences in bound variable names. Given a formula A, a first-order variable x and a closed first-order expression e, we denote by $A[e/x]$ the formula obtained by replacing every free occurrence of x by the first-order expression e in the formula A, possibly renaming some bound variables of A to avoid name clashes.

Lastly, although the formulas of the language of second-order logic are constructed from atomic formulas only using implication and first- and second-order universal quantifications, we can define other logical constructions (negation, conjunction disjunction, first- and second-order existential quantification as well as Leibniz equality) using the so-called second-order encodings:

$$\begin{array}{rcl@{\qquad}rcl}
\bot & \triangleq & \forall Z.Z & A \Leftrightarrow B & \triangleq & (A \to B) \land (B \to A) \\
\neg A & \triangleq & A \to \bot & \exists x.A(x) & \triangleq & \forall Z.(\forall x.(A(x) \to Z) \to Z) \\
A \land B & \triangleq & \forall Z.((A \to B \to Z) \to Z) & \exists X.A(X) & \triangleq & \forall Z.(\forall X.(A(X) \to Z) \to Z) \\
A \lor B & \triangleq & \forall Z.((A \to Z) \to (B \to Z) \to Z) & e_1 = e_2 & \triangleq & \forall W.(W(e_1) \to W(e_2))
\end{array}$$

3.3.1.2 Predicates and second-order substitution

We call a *predicate of arity k* any expression which associates to the variable $x_1, \dots, x_k$ a formula C depending on these variables. More formally, we could (ab)use the λ-notation to define them as expressions of the form $P \equiv \lambda x_1 \cdots x_k \,.\, C$ where C is then an arbitrary formula. The set of free variables of a k-ary predicate $P \equiv \lambda x_1 \cdots x_k \,.\, C$ is defined by $FV(P) \equiv FV(C) \setminus \{x_1, \dots, x_k\}$, and the application of the predicate $P \equiv \lambda x_1 \cdots x_k \,.\, C$ to a k-tuple of first-order expressions $e_1, \dots, e_k$ is defined by letting

$$P(e_1, \dots, e_k) \ \equiv \ (\lambda x_1 \cdots x_k \,.\, C)(e_1, \dots, e_k) \ \equiv \ C[e_1/x_1, \dots, e_k/x_k]$$

$$\frac{(x:A)\in\Gamma}{\Gamma\vdash x:A}\,(\text{Ax})\qquad\frac{\Gamma,x:A\vdash t:B}{\Gamma\vdash\lambda x\,.\,t:A\to B}\,(\to_I)\qquad\frac{\Gamma\vdash t:A\to B\qquad\Gamma\vdash t:A}{\Gamma\vdash tu:B}\,(\to_E)$$

$$\frac{\Gamma\vdash t:A\quad x\notin FV(\Gamma)}{\Gamma\vdash t:\forall x.A}\,(\forall^1_I)\qquad\frac{\Gamma\vdash t:\forall x.A}{\Gamma\vdash t:A\{x:=e\}}\,(\forall^1_E)\qquad\frac{\Gamma\vdash t:A\quad X\notin FV(\Gamma)}{\Gamma\vdash t:\forall X.A}\,(\forall^2_I)$$

$$\frac{\Gamma\vdash t:\forall X.A}{\Gamma\vdash t:A\{X:=P\}}\,(\forall^2_E)\qquad\frac{}{\Gamma\vdash\text{cc}:((A\to B)\to A)\to A}\,(\text{cc})$$

Figure 3.1: Typing rules of second-order logic

(by analogy with β-reduction). Given a formula A, a k-ary predicate variable X and an actual k-ary predicate P, we finally define the operation of *second-order substitution* $A[P/X]$ as follows:

$$\begin{array}{rcll}
X(e_1,\dots,e_k)[P/X] & \triangleq & P(e_1,\dots,e_k) & \\
Y(e_1,\dots,e_m)[P/X] & \triangleq & Y(e_1,\dots,e_m) & (Y\neq X)\\
(A\to B)[P/X] & \triangleq & A[P/X]\to B[P/X] & \\
(\forall x.A)[P/X] & \triangleq & \forall x.A[P/X] & (x\notin FV(P))\\
(\forall X.A)[P/X] & \triangleq & \forall X.A & \\
(\forall Y.A)[P/X] & \triangleq & \forall Y.A[P/X] & (Y\neq X,\ Y\notin FV(P))
\end{array}$$

3.3.2 A type system for classical second-order logic

We shall now present the deduction system of classical second-order logic as a type system based on typing judgments of the form $\Gamma\vdash t:A$, where t is a proof-like term, i.e. a λ_c-term containing no continuation constant $\mathbf{k}_\pi$; and A is a formula of second-order logic.

The type system of classical second-order logic is defined from the typing rules of Figure 3.1. These typing rules are the usual typing rules of AF2 [92], plus a specific typing rule for the instruction cc which permits to recover the full strength of classical logic.

Using the encodings of second-order logic, we can derive from the typing rules of Figure 3.1 the usual introduction and elimination rules of absurdity, conjunction, disjunction, (first- and second-order) existential quantification and Leibniz equality [92]. As explained in Section 1.1.2.1, the typing rule for `call/cc` (law of Peirce) allows us to construct proof-terms for classical reasoning principles such as the excluded middle, *reductio ad absurdum*, de Morgan laws, etc.

3.3.3 Classical second-order arithmetic (PA2)

From now on, we consider the particular case of *second-order arithmetic* (PA2), where first-order expressions are intended to represent natural numbers. For that, we assume that every k-ary function symbol $f\in\Sigma$ comes with an interpretation in the standard model of first-order arithmetic (Section 1.2.4) as a function $[\![f]\!]:\mathbb{N}^k\to\mathbb{N}$, so that we can give a denotation $[\![e]\!]\in\mathbb{N}$ to every closed first-order expression e. Moreover, we assume that each function symbol associated to a primitive recursive definition (cf Section 3.3.1.1) is given its standard interpretation in $\mathbb{N}$. In this way, every numeral $n\in\mathbb{N}$ is represented in the world of first-order expressions as the closed expression $s^n(0)$ that we still write n, since $[\![s^n(0)]\!]=n$.

3.3.3.1 Induction

Following Dedekind's construction of natural numbers, we consider the predicate $\mathsf{Nat}(x)$ [60, 92] defined by

$$\mathsf{Nat}(x)\ \triangleq\ \forall Z.(Z(0)\to\forall y.(Z(y)\to Z(s(y)))\to Z(x))\,,$$

that defines the smallest class of individuals containing zero and closed under the successor function. One of the main properties of the logical system presented above is that the axiom of induction, that we can write $\forall x.\mathsf{Nat}(x)$, is not derivable from the rules of Figure 3.1. As proved by Krivine [97, Theorem 12], this axiom is not even (universally) realizable in general. To recover the strength of arithmetic reasoning, we need to relativize all first-order quantifications to the class $\mathsf{Nat}(x)$ of Dedekind numerals using the shorthands for *numeric quantifications*

$$\begin{array}{rcl} \forall^{\mathsf{nat}}x.A(x) & \triangleq & \forall x.(\mathsf{Nat}(x) \to A(x)) \\ \exists^{\mathsf{nat}}x.A(x) & \triangleq & \forall Z.(\forall x.(\mathsf{Nat}(x) \to A(x) \to Z) \to Z) \end{array}$$

so that the *relativized induction axiom* becomes provable in second-order logic [92]:

$$\forall Z.(Z(0) \to \forall^{\mathsf{nat}}x.(Z(x) \to Z(s(x))) \to \forall^{\mathsf{nat}}x.Z(x))\,.$$

3.3.3.2 The axioms of PA2

Formally, a formula A is a *theorem* of second-order arithmetic (PA2) if it can be derived from Peano axioms (see Example 1.12), expressing that the successor function is injective and not surjective:

$$\text{(PA5)} \quad \forall x.\forall y.(s(x) = s(y) \to x = y) \qquad\qquad \text{(PA6)} \quad \forall x.(s(x) \neq 0)$$

and from the definitional equalities attached to the (primitive recursive) function symbols of the signature:

$$\begin{array}{llll} \text{(PA1)} & \forall x.(0 + x = x) & \text{(PA2)} & \forall x.\forall y.(s(x) + y = s(x + y)) \\ \text{(PA3)} & \forall x.(0 \times x = 0) & \text{(PA4)} & \forall x.\forall y.(s(x) \times y = (x \times y) + y) \end{array}$$

etc... Unlike the non relativized induction axiom—that requires a special treatment in PA2—we shall see in Section 3.4.6 that all these axioms are realized by simple proof-like terms.

Observe that we consider here an unusual definition of (PA2), since the usual one includes the induction rule as an axiom. Nonetheless, the two theories are related through the relativization of first-order quantifications. Namely, if A is a theorem of (PA2) with induction, then the relativized formula A^{Nat} is a theorem of $(PA2)$ without induction.

3.4 Classical realizability semantics

3.4.1 Generalities

Given a particular instance of the λ_c-calculus (defined from particular sets $\mathcal{B}$, C and from a particular relation of evaluation $\succ_1$ as described in Section 3.2), we shall now build a classical realizability model in which every closed formula A of the language of PA2 will be interpreted as a set of closed terms $|A| \subseteq \Lambda$, called the *truth value* of A, and whose elements will be called the *realizers* of A.

3.4.1.1 Poles, truth values and falsity values

Formally, the construction of the realizability model is parameterized by a *pole* $\perp\!\!\!\perp$ in the sense of the following definition:

Definition 3.5 (Poles). A *pole* is any set of processes $\perp\!\!\!\perp \subseteq \Lambda \star \Pi$ which is closed under anti-evaluation, in the sense that both conditions $p \succ p'$ and $p' \in \perp\!\!\!\perp$ together imply that $p \in \perp\!\!\!\perp$ for all processes $p, p' \in \Lambda \star \Pi$. ⌟

Given a fixed set of processes, the following two examples are standard methods to define a pole. The first one is straightforward in that it simply consists in taking the closure by anti-evaluation. The second one might be more disconcerting, and consists in taking the set of processes which are unreachable by reduction.

Example 3.6 (Goal-oriented pole)**.** Given a set of processes P, the set of all processes that reach an element of P after zero, one or several evaluation steps, that is:

$$\perp\!\!\!\perp \triangleq \{p \in \Lambda \star \Pi \ : \ \exists p' \in P\ (p \succ p')\}$$

is a valid pole. Indeed, if p,p' are processes such that $p \succ p'$ and $p' \in \perp\!\!\!\perp$, by definition there is a process $p_0 \in P$ such that $p' \succ p_0$. Thus $p \succ p' \succ p_0$ and $p \in \perp\!\!\!\perp$, which concludes the proof that $\perp\!\!\!\perp$ is closed by anti-reduction. By definition, the set $\perp\!\!\!\perp$ is the smallest pole that contains the set of processes P as a subset. ⌟

Example 3.7 (Thread-oriented pole)**.** Given a set of processes P, the complement set of the union of all threads starting from an element of P, that is:

$$\perp\!\!\!\perp \triangleq \Big(\bigcup_{p\in P} \mathbf{th}(p)\Big)^c \equiv \bigcap_{p\in P}\big(\mathbf{th}(p)\big)^c$$

is a valid pole. It is indeed quite easy to check that $\perp\!\!\!\perp$ is closed by anti-reduction. Consider two processes p,p' such that $p \succ p'$ and $p' \in P$, and assume that there is a process $p_0 \in P$ such that $p_0 \succ p$. Then $p_0 \succ p'$ which contradicts the fact that $p' \in \perp\!\!\!\perp$. Thus there is no such process p_0 and $p \in \perp\!\!\!\perp$. This pole is also the largest one that does not intersect P. ⌟

Let us now consider a fixed pole $\perp\!\!\!\perp$. We call a *falsity value* any set of stacks $S \subseteq \Pi$. Every falsity value $S \subseteq \Pi$ induces a *truth value* $S^{\perp\!\!\!\perp} \subseteq \Lambda$ that is defined by

$$S^{\perp\!\!\!\perp} = \{t \in \Lambda \ : \ \forall \pi \in S\ (t \star \pi) \in \perp\!\!\!\perp\}\,.$$

Intuitively, every falsity value $S \subseteq \Pi$ represents a particular set of *tests*, while the corresponding truth value $S^{\perp\!\!\!\perp}$ represent the set of all *programs* that passes all tests in S (w.r.t. the pole $\perp\!\!\!\perp$, that can be seen as the *challenge* or the *referee*). From the definition of $S^{\perp\!\!\!\perp}$, it is clear that the larger the falsity value S, the smaller the corresponding truth value $S^{\perp\!\!\!\perp}$, and vice-versa.

3.4.1.2 Formulas with parameters

In order to interpret second-order variables that occur in a given formula A, it is convenient to enrich the language of PA2 with a new predicate symbol $\dot{F}$ of arity k for every *falsity value function* F of arity k, that is, for every function $F : \mathbb{N}^k \to \mathcal{P}(\Pi)$ that associates a falsity value $F(n_1,\ldots,n_k) \subseteq \Pi$ to every k-tuple $(n_1,\ldots,n_k) \in \mathbb{N}^k$. A formula of the language enriched with the predicate symbols $\dot{F}$ is then called a *formula with parameters.* Formally, this corresponds to the formulas defined by:

$$A,B \ ::= \ X(e_1,\ldots,e_k) \mid A \to B \mid \forall x.A \mid \forall X.A \mid \dot{F}(e_1,\ldots,e_k) \qquad\qquad X \in \mathcal{V}_2, F \in \mathcal{P}(\Pi)^{\mathbb{N}^k}$$

The notions of a *predicate with parameters* and of a *typing context with parameters* are defined similarly. The notations $FV(A)$, $FV(P)$, $FV(\Gamma)$, $\mathrm{dom}(\Gamma)$, $A[e/x]$, $A[P/X]$, etc. are extended to all formulas A with parameters, to all predicates P with parameters and to all typing contexts Γ with parameters in the obvious way.

3.4.2 Definition of the interpretation function

The interpretation of the closed formulas with parameters follows the intuition that the falsity value $\|A\|$ of a formula A contains tests that terms have to challenge to be in the corresponding truth value $|A|$. In particular, a test for $A \to B$ consists in a defender of A together with a test for B, while a test for a quantified formula $\forall x.A$ (resp. $\forall X.A$) is simply a test for one of the possible instantiations for the variable x (resp. X).

Definition 3.8 (Interpretation of closed formulas with parameters). The falsity value $\|A\| \subseteq \Pi$ of a closed formula A with parameters is defined by induction on the number of connectives/quantifiers in A from the equations

$$\begin{array}{rcll}
\|\dot{F}(e_1,\ldots,e_k)\| & \triangleq & F([\![e_1]\!],\ldots,[\![e_k]\!]) & \\
\|A \to B\| & \triangleq & |A| \cdot \|B\| \quad = \quad \big\{t \cdot \pi \ : \ t \in |A|,\ \pi \in \|B\|\big\} & \\
\|\forall x.A\| & \triangleq & \bigcup_{n \in \mathbb{N}} \|A[n/x]\| & \\
\|\forall X.A\| & \triangleq & \bigcup_{F:\mathbb{N}^k \to \mathcal{P}(\Pi)} \|A[\dot{F}/X]\| & \text{(if } X \text{ has arity } k\text{)}
\end{array}$$

whereas its truth value $|A| \subseteq \Lambda$ is defined by $|A| = \|A\|^{\perp\!\!\!\perp}$. Finally, defining $\top \equiv \dot{\emptyset}$ (recall that we have $\bot \equiv \forall X.X$), one can check that we have :

$$\|\top\| = \emptyset \qquad\qquad |\top| = \Lambda \qquad\qquad \|\bot\| = \Pi$$

⌟

Since the falsity value $\|A\|$ (resp. the truth value $|A|$) of A actually depends on the pole $\perp\!\!\!\perp$, we shall write it sometimes $\|A\|_{\perp\!\!\!\perp}$ (resp. $|A|_{\perp\!\!\!\perp}$) to recall the dependency.

Definition 3.9 (Realizers). Given a closed formula A with parameters and a closed term $t \in \Lambda$, we say that:

1. t *realizes* A and write $t \Vdash A$ when $t \in |A|_{\perp\!\!\!\perp}$. (This notion is relative to a particular pole $\perp\!\!\!\perp$.)
2. t *universally realizes* A and write $t \Vvdash A$ when $t \in |A|_{\perp\!\!\!\perp}$ for all poles $\perp\!\!\!\perp$.

⌟

From these definitions, we clearly have

$$|\forall x\, A| \ = \ \bigcap_{n \in \mathbb{N}} |A\{x := n\}| \qquad \text{and} \qquad |\forall X\, A| \ = \ \bigcap_{F:\mathbb{N}^k \to \mathcal{P}(\Pi)} |A\{X := \dot{F}\}|\,.$$

On the other hand, the truth value $|A \to B|$ of an implication $A \to B$ slightly differs from its traditional interpretation in Kleene's realizability (Section Section 3.1.1). Writing

$$|A| \to |B| \ = \ \{t \in \Lambda \ : \ \text{for all } u \in \Lambda\,,\ u \in |A| \text{ implies } tu \in |B|\}\,,$$

we can check that:

Lemma 3.10. *For all closed formulas A and B with parameters:*

1. $|A \to B| \subseteq |A| \to |B|$ *(adequacy of modus ponens).*
2. *The converse inclusion does not hold in general, unless the pole $\perp\!\!\!\perp$ is insensitive to the rule* (Push)*, that is:* $tu \star \pi \in \perp\!\!\!\perp$ *iff* $t \star u \cdot \pi \in \perp\!\!\!\perp$ *(for all $t, u \in \Lambda$, $\pi \in \Pi$).*
3. *In all cases, $t \in |A| \to |B|$ implies $\lambda x\,.\,tx \in |A \to B|$ (for all $t \in \Lambda$).*

Proof. These simple statements are a nice pretext to a first manipulation of the definitions.

1. Let $t \in |A \to B|$ and $u \in |A|$. To prove that $tu \in |B|$, we consider an arbitrary stack $\pi \in \|B\|$. By applying the rule (Push) we get $tu \star \pi \succ_1 t \star u \cdot \pi$. Since $t \in |A \to B|$ and $u \cdot \pi \in \|A \to B\|$, the process $t \star u \cdot \pi$ belongs to $\perp\!\!\!\perp$. Hence $tu \star \pi \in \perp\!\!\!\perp$ by anti-evaluation.

2. Let $t \in |A| \to |B|$. To prove that $t \in |A \to B|$, we consider an arbitrary element of the falsity value $\|A \to B\|$, that is, a stack $u \cdot \pi$ where $u \in |A|$ and $\pi \in \|B\|$. We clearly have $tu \star \pi \in \perp\!\!\!\perp$, since $tu \in |B|$ from our assumption on t. But since $\perp\!\!\!\perp$ is insensitive to the rule (Push), we also have $t \star u \cdot \pi \in \perp\!\!\!\perp$.
3. Let $t \in |A| \to |B|$. To prove that $\lambda x \,.\, tx \in |A \to B|$, we consider an arbitrary element of the falsity value $\|A \to B\|$, that is, a stack $u{\cdot}\pi$ where $u \in |A|$ and $\pi \in \|B\|$. We have $\lambda x \,.\, tx{\star}u{\cdot}\pi \succ_1 tu{\star}\pi \in \perp\!\!\!\perp$ (since $tu \in |B|$), hence $\lambda x \,.\, tx \star u \cdot \pi \in \perp\!\!\!\perp$ by anti-evaluation.

□

Besides, it is easy to prove that cc is indeed a universal realizer of Peirce's law:

Lemma 3.11 (Law of Peirce). *Let A and B be two closed formulas with parameters:*

1. *If $\pi \in \|A\|$, then $\mathbf{k}_\pi \Vdash A \to B$.*
2. *$\mathrm{cc} \mid\!\Vdash ((A \to B) \to A) \to A$.*

Proof. 1. Let $\pi \in \|A\|$. To prove that $\mathbf{k}_\pi \in |A \to B|$, we need to check that $\mathbf{k}_\pi \star t \cdot \pi' \in \perp\!\!\!\perp$ for all $t \in |A|$ and $\pi' \in \|B\|$. By applying the rule (Restore) we get $\mathbf{k}_\pi \star t \cdot \pi' \succ_1 t \star \pi \in \perp\!\!\!\perp$ (since $t \in |A|$ and $\pi \in \|A\|$), hence $\mathbf{k}_\pi \star t \cdot \pi' \in \perp\!\!\!\perp$ by anti-evaluation.

2. To prove that $\mathrm{cc} \Vdash ((A \to B) \to A) \to A$ (for any pole $\perp\!\!\!\perp$), we need to check that $\mathrm{cc} \star t \cdot \pi \in \perp\!\!\!\perp$ for all $t \in |(A \to B) \to A|$ and $\pi \in \|A\|$. By applying the rule (Save) we get $\mathrm{cc} \star t \cdot \pi \succ_1 t \star \mathbf{k}_\pi \cdot \pi$. But since $\mathbf{k}_\pi \in |A \to B|$ (from (1)) and $\pi \in \|A\|$, we have $\mathbf{k}_\pi \cdot \pi \in \|(A \to B) \to A\|$, so that $t \star \mathbf{k}_\pi \cdot \pi \in \perp\!\!\!\perp$. Hence $\mathrm{cc} \star t \cdot \pi \in \perp\!\!\!\perp$ by anti-evaluation.

□

3.4.3 Valuations and substitutions

In order to express the soundness invariants relating the type system of Section 3.3.3 with the classical realizability semantics defined above, we need to introduce some more terminology.

Definition 3.12 (Valuations). A *valuation* is a function ρ that associates a natural number $\rho(x) \in \mathbb{N}$ to every first-order variable x and a falsity value function $\rho(X) : \mathbb{N}^k \to \mathcal{P}(\Pi)$ to every second-order variable X of arity k.

1. Given a valuation ρ, a first-order variable x and a natural number $n \in \mathbb{N}$, we denote by $\rho, x \leftarrow n$ the valuation defined by:
$$(\rho, x \leftarrow n) \triangleq \rho_{|\,\mathrm{dom}(\rho)\setminus\{x\}} \cup \{x \leftarrow n\}\,.$$
2. Given a valuation ρ, a second-order variable X of arity k and a falsity value function $F : \mathbb{N}^k \to \mathcal{P}(\Pi)$, we denote by $\rho, X \leftarrow F$ the valuation defined by:
$$(\rho, X \leftarrow F) \triangleq \rho_{|\,\mathrm{dom}(\rho)\setminus\{X\}} \cup \{X \leftarrow F\}\,.$$

⌟

To every pair (A, ρ) formed by a (possibly open) formula A of PA2 and a valuation ρ, we associate a *closed* formula with parameters $A[\rho]$ that is defined by
$$A[\rho] \triangleq A[\rho(x_1)/x_1, \ldots, \rho(x_n)/x_n, \dot{\rho}(X_1)/X_1, \ldots, \dot{\rho}(X_m)/X_m]$$
where $x_1, \ldots, x_n, X_1, \ldots, X_m$ are the free variables of A, and writing $\dot{\rho}(X_i)$ the predicate symbol associated to the falsity value function $\rho(X_i)$. This operation naturally extends to typing contexts by letting
$$(x_1 : A_1, \ldots, x_n : A_n)[\rho] \triangleq x_1 : A_1[\rho], \ldots, x_n : A_n[\rho]\,.$$

Definition 3.13 (Substitutions)**.** A *substitution* is a finite function σ from λ-variables to closed λ_c-terms. Given a substitution σ, a λ-variable x and a closed λ_c-term u, we denote by $\sigma, x := u$ the substitution defined by $(\sigma, x := u) \equiv \sigma_{|\,\mathrm{dom}(\sigma)\setminus\{x\}} \cup \{x := u\}$. ⌟

Given an open λ_c-term t and a substitution σ, we denote by $t[\sigma]$ the term defined by

$$t[\sigma] \;\triangleq\; t[\sigma(x_1)/x_1, \ldots, \sigma(x_n)/x_n]$$

where $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_n\}$. Notice that $t[\sigma]$ is closed as soon as $FV(t) \subseteq \mathrm{dom}(\sigma)$. We say that a substitution σ *realizes* a closed context Γ with parameters and write $\sigma \Vdash \Gamma$ if:

1. $\mathrm{dom}(\sigma) = \mathrm{dom}(\Gamma)$;
2. $\sigma(x) \Vdash A$ for every declaration $(x : A) \in \Gamma$.

3.4.4 Adequacy

The adequacy of typing judgments and typing rules with respect to a pole is defined exactly like the adequacy with respect to a model (Definition 1.17). Given a fixed pole $⫫$, we say that:

1. A typing judgment $\Gamma \vdash t : A$ is *adequate* (w.r.t. the pole $⫫$) if for all valuations ρ and for all substitutions $\sigma \Vdash \Gamma[\rho]$ we have $t[\sigma] \Vdash A[\rho]$.
2. More generally, we say that an inference rule

$$\frac{J_1 \quad \cdots \quad J_n}{J_0}$$

is adequate (w.r.t. the pole $⫫$) if the adequacy of all typing judgments $J_1, \ldots, J_n$ implies the adequacy of the typing judgment J_0.

Proposition 3.14 (Adequacy)**.** *The typing rules of Figure 3.1 are adequate w.r.t. any pole $⫫$, as well as all the judgments $\Gamma \vdash t : A$ that are derivable from these rules.*

Proof. The rule for cc directly stems from Lemma 3.11, while introduction and elimination rules for universal quantifiers results from the definition of the corresponding falsity values. We will only sketch the proof for the introduction and elimination rules of implication.

- **Case** ($\to_I$). Assume that $\Gamma \vdash t : A \to B$ and $\Gamma \vdash u : B$ are adequate w.r.t. $⫫$, and pick a valuation ρ and a substitution σ such that $\sigma \Vdash \Gamma[\rho]$. We want to show that $(tu)[\sigma] \Vdash B[\rho]$. It suffices to show that if $\pi \in \|B[\rho]\|$, then $(tu)[\sigma] \star \pi \in ⫫$. Applying the (Push) rule, we get :

$$(tu)[\sigma] \star \pi \;\succ\; t[\sigma] \star u[\sigma] \cdot \pi$$

By hypothesis, we have $u[\sigma] \Vdash A[\rho]$ (and then $u[\sigma] \cdot \pi \in \|(A \to B)[\rho]\|)$), and $t[\sigma] \Vdash (A \to B)[\rho]$, so that $t[\sigma] \star u[\sigma] \cdot \pi$ belongs to $⫫$. We conclude by anti-reduction.

- **Case** ($\to_E$). Assume that $\Gamma, x : A \vdash t : B$ is adequate w.r.t $⫫$. This means that for any valuation ρ, any $u \Vdash A[\rho]$ and any $\sigma \Vdash \Gamma[\rho]$, denoting by σ' the substitution $\sigma, x := u$, we have $t[\sigma'] \Vdash B[\rho]$. Let us pick a valuation ρ and a substitution σ such that $\sigma \Vdash \Gamma[\rho]$. We want to show that $(\lambda x.t)[\sigma] \Vdash (A \to B)[\rho]$. Let $u \cdot \pi$ be a stack in $\|(A \to B)[\rho]\|$. Applying the (Grab) rule, we have :

$$(\lambda x.t)[\sigma] \star u \cdot \pi \;\succ\; t[\sigma, x := u] \star \pi$$

By hypothesis, we have $u \Vdash A[\rho]$, and so $t[\sigma, x := u] \Vdash B[\rho]$. Thus $t[\sigma, x := u] \star \pi$ belongs to $⫫$. and we conclude by anti-reduction. □

Since the typing rules of Figure 3.1 involve no continuation constant, every realizer that comes from a proof of second order logic by Proposition 3.14 is thus a proof-like term.

3.4.5 The induced model

It is not innocent if the sets $|A|$ introduced in the previous sections were called *truth values*. Indeed, this construction defined a model for second-order logic where truth values are made of λ_c-terms. In a nutshell, starting from the standard model $\mathbb{N}$ for first-order expressions and an instance of the λ_c-calculus (that is with `call/cc` only or other extras instructions), the choice of a particular pole $\perp\!\!\!\perp$ defines a truth value for all formulas of the language. Naively, we could be tempted to define the valid formulas as the one whose truth value is not empty. Yet, this raises a problem of consistency:

Proposition 3.15. *If $\perp\!\!\!\perp \neq \emptyset$, then there is a term t such that for all formula A, $t \in |A|$.*

Proof. Assume that the $\perp\!\!\!\perp$ is not empty, and let $\langle t \| \pi \rangle$ be a process in $\perp\!\!\!\perp$. Then for any formula A, $\mathbf{k}_\pi t \Vdash A$. Indeed, for any stack ρ (and in particular any stack in $\|A\|$), we have:

$$\mathbf{k}_\pi t \star \rho \;\succ\; \mathbf{k}_\pi \star t \cdot \rho \;\succ\; t \star \pi$$

The last process being in the pole, they all are by anti-evaluation, and thus $\mathbf{k}_\pi t \star \rho \in \perp\!\!\!\perp$. □

If we examine $k_\pi\, t$, the guilty term in the previous proof, there is two observations to do. First, it is worth noting that independently of t and π, this term can not be typed since there is no typing rule for continuations k_π. Second, sticking with the intuition that a realizer is a term that can challenge successfully any tests in the falsity value, this term is morally a cheater: in front of a test ρ, it actually refuses to challenge it, drops it and goes directly to the test π for which it already knows a winning defender t. Therefore, the problem comes from the presence of a continuation constant, and we should restrict truth values to terms without continuation constants, *i.e.* to proof-like terms.

To ease the next definition[9], we restrict ourselves to the *full standard model* of PA2. In this model, first-order individuals are interpreted by the elements of $\mathbb{N}$, while second-order objects of arity k are interpreted in the sets of k-ary relations on the set $\mathbb{N}$. We denote this model by $\mathcal{M}$.

Definition 3.16 (Realizability model)**.** Given the full standard model $\mathcal{M}$ of PA2 and a pole $\perp\!\!\!\perp$, we call realizability model and denote by $\mathcal{M}_{\perp\!\!\!\perp}$ the model in which the validity of formulas is defined by:

$$\mathcal{M}_{\perp\!\!\!\perp} \Vdash A \qquad \text{if and only if} \qquad |A| \cap PL \neq \emptyset$$

┘

The previous definition gives a simple criterion of consistency for realizability models:

Proposition 3.17 (Consistency)**.** *The model $\mathcal{M}_{\perp\!\!\!\perp}$ induce by the pole $\perp\!\!\!\perp$ is consistent if and only if for each proof-like term t, there exists one stack π such that $t \star \perp\!\!\!\perp \notin \perp\!\!\!\perp$.*

Proof. Recall that $\|\bot\| = \Pi$. Hence $\mathcal{M}_{\perp\!\!\!\perp} \Vdash \bot$ if and only if there exists a proof-like term t such that $t \Vdash \bot$, *i.e.* for any stack π, $t \star \pi \in \perp\!\!\!\perp$. Thus $\mathcal{M}_{\perp\!\!\!\perp} \nVdash \bot$ if and only if for each proof-like term t there is at least one stack π such that $t \star \pi \notin \perp\!\!\!\perp$. □

3.4.6 Realizing the axioms of PA2

Let us recall that in PA2, Leibniz equality $e_1 = e_2$ is defined by $e_1 = e_2 \equiv \forall Z\,(Z(e_1) \to Z(e_2))$.

Proposition 3.18 (Realizing Peano axioms)**.** *:*

1. $\lambda z\,.\,z \;\Vdash\; \forall x\,\forall y\,(s(x) = s(y) \to x = y)$

[9] The definition of realizability models could be reformulated to consider a ground model of PA2 as parameter, but this would require a formal definition of the models of PA2. This would have been unnecessarily complex for the sole purpose of perceiving the spirit of realizability models.

2. $\lambda z\,.\,zu \ \Vdash \ \forall x\,(s(x) = 0 \to \bot)$ *(where u is any term such that* $FV(u) \subseteq \{z\}$*).*

3. $\lambda z\,.\,z \ \Vdash \ \forall x_1 \cdots \forall x_k\,(e_1(x_1,\ldots,x_n) = e_2(x_1,\ldots,x_k))$
for all arithmetic expressions $e_1(x_1,\ldots,x_n)$ *and* $e_2(x_1,\ldots,x_k)$ *such that*
$\mathbb{N} \models \forall x_1 \cdots \forall x_k\,(e_1(x_1,\ldots,x_n) = e_2(x_1,\ldots,x_k))$.

Proof. The proof is an easy verification, and can be found in [97]. □

From this we deduce the main theorem, proving that any realizability model is a model of PA2:

Theorem 3.19 (Realizing the theorems of PA2). *If A is a theorem of PA2 (in the sense defined in Section 3.3.3.2), then there is a closed proof-like term t such that* $t \Vdash A$.

Proof. Immediately follows from Prop. 3.14 and 3.18. □

3.4.7 The full standard model of PA2 as a degenerate case

It is easy to see that when the pole $\perp\!\!\!\perp$ is empty, the classical realizability model defined above collapses to the full standard model $\mathcal{M}$ of PA2. For that, we first notice that when $\perp\!\!\!\perp = \emptyset$, the truth value $S^{\perp\!\!\!\perp}$ associated to an arbitrary falsity value $S \subseteq \Pi$ can only take two different values: $S^{\perp\!\!\!\perp} = \Lambda_c$ when $S = \emptyset$, and $S^{\perp\!\!\!\perp} = \emptyset$ when $S \neq \emptyset$. Moreover, we easily check that the realizability interpretation of implication and universal quantification mimics the standard truth value interpretation of the corresponding logical construction in the case where $\perp\!\!\!\perp = \emptyset$. It is easy to check that:

Proposition 3.20. *If* $\perp\!\!\!\perp = \emptyset$*, then for every closed formula A of PA2 we have*

$$|A| = \begin{cases} \Lambda & \text{if } \mathcal{M} \models A \\ \emptyset & \text{if } \mathcal{M} \not\models A \end{cases}$$

An interesting consequence of the above proposition is the following:

Corollary 3.21. *If a closed formula A has a universal realizer* $t \Vdash A$*, then A is true in the full standard model* $\mathcal{M}$ *of PA2.*

Proof. If $t \Vdash A$, then $t \in |A|_\emptyset$. Therefore $|A|_\emptyset = \Lambda$ and $\mathcal{M} \models A$. □

However, the converse implication is false in general, since the formula $\forall x\,\mathrm{Nat}(x)$ (cf Section 3.3.3.1) that expresses the induction principle over individuals is obviously true in $\mathcal{M}$, but it has no universal realizer when evaluation is deterministic [97, Theorem 12].

3.5 Applications

We present in this section some applications of Krivine realizability, both on its logical and computational facets. While we introduce theses applications in the framework of the λ_c-calculus, keep in mind that they are not peculiar to this calculus. As we will see in the next sections, other calculi are suitable for a realizability interpretation *à la* Krivine, and can thus benefit from the results expressed thereafter.

3.5.1 Soundness and normalization

Once the realizability interpretation is defined and the adequacy proved, the soundness of the language is a direct consequence of the adequacy. Indeed, if there was a proof t of $\bot$, then by adequacy t would be a uniform realizer of $\bot$. Thus the existence of one consistent model is enough to contradict this possibility, ensuring the correction of the type system. Similarly, the normalization of the language is also a direct consequence of the adequacy and the following observation:

Proposition 3.22 (Normalizing processes)**.** *The set* $\bot\!\!\!\bot_{\Downarrow} \triangleq \{p \in \Lambda \times \Pi : p \textit{ normalizes}\}$ *defines a valid pole.*

Proof. We need to check that $\bot\!\!\!\bot_{\Downarrow}$ is closed by anti-reduction, so let p,p' be two processes such that $p \succ p'$ and $p' \in \bot\!\!\!\bot_{\Downarrow}$. The latter means by definition that p' normalizes. Since $p \succ p'$, necessarily p normalizes too and thus belongs to the pole $\bot\!\!\!\bot_{\Downarrow}$. □

Note that we only consider the normalization with respect to the evaluation strategy of the processes, which corresponds to the weak-head reduction in the sense of the λ-calculus. In particular, this is weaker than the strong and weak normalizations of the λ-calculus (see Section 2.1.5). We will use this observation in Chapters 4 and 6 to prove normalization properties of different calculi.

3.5.2 Specification problem

The specification problem for a formula A can be expressed through the following question:

Which are the terms t *such that* $t \Vvdash A$ *?*

In other words, it poses the question of exhibiting a (computational) characterization for the realizers of A. Thanks to the adequacy of the interpretation with respect to typing, such a characterization would also apply to terms of type A.

3.5.2.1 Toy example: $\forall X.X \to X$

In the language of second-order logic, the type of the identity function $I = \lambda x.x$ is described by the formula $\forall X(X \to X)$. A closed term $t \in \Lambda$ is said to be *identity-like* if $t \star u \cdot \pi \succ u \star \pi$ for all $u \in \Lambda$ and $\pi \in \Pi$. Examples of identity-like terms are of course the identity function I but also terms such as II, δI (where $\delta \equiv \lambda x.xx$), $\lambda x.\mathrm{cc}(\lambda k.x)$, $\mathrm{cc}(\lambda k.kI\delta k)$, etc. It is easy to verify that any identity-like term is a universal realizer of the formula $\forall X.X \to X$. But the converse also holds, and thus provides an answer to the specification problem for the formula $\forall X.(X \to X)$.

Proposition 3.23. *For all terms* $t \in \Lambda$*, we have:*

$$t \Vvdash \forall X.(X \to X) \qquad\qquad \Leftrightarrow \qquad\qquad t \textit{ is identity-like}$$

Proof. The interesting direction of the proof is from left to right. We prove it with the so-called *methods of threads* [63]. Assume $t \Vvdash \forall X(X \to X)$, and consider $u \in \Lambda, \pi \in \Pi$. We want to prove that $t \star u \cdot \pi \succ u \star \pi$. We define the pole

$$\bot\!\!\!\bot \equiv (\mathbf{th}(t \star u \cdot \pi))^c \equiv \{p \in \Lambda \star \Pi : (t \star u \cdot \pi \not\succ p)\}$$

as well as the falsity value $S = \{\pi\}$. From the definition of $\bot\!\!\!\bot$, we know that $t \star u \cdot \pi \notin \bot\!\!\!\bot$. As $t \Vdash \dot{S} \to \dot{S}$ and $\pi \in \|\dot{S}\|$, necessarily $u \nVdash S$. This means that $u \star \pi \notin \bot\!\!\!\bot$, that is $t \star u \cdot \pi \succ u \star \pi$. □

3.5.2.2 Game-theoretic interpretation

In the previous section we gave a toy example of specification that was proved using the method of threads. If this method is very useful, it has the drawbacks of becoming very painful when the formula to specify get more complex. A more scalable way to obtain specifications (which uses the threads method as a technical tool) is to strengthen the intuition of an opposition between two players underlying Krivine realizability. In addition to being a useful specification method, this idea that realizers of a formula are its defenders, turns out to be a helpful intuition when defining the realizability interpretation of a language.

As we only want to give an oversight of the corresponding game-theoretic intuitions, we will illustrate this methodology with an example. Precise definitions, proofs etc... can be found in [63, 64, 65]. We choose as a running example the formula $\Phi_f \triangleq \exists x.\forall y.f(x) \leq f(y)$, where f is any computable function from $\mathbb{N}$ to $\mathbb{N}$, expressing the fact that f admits a minimum. We could have chosen any arithmetical formula (see [65]), or second-order formulas, as Peirce's law (see [63, 64]). We believe this example to be representative enough of the general situation and easier to understand that an example in a second-order setting.

Eloise and Abelard Still writing $\mathcal{M}$ for the full standard model of PA2, the formula Φ_f naturally induces a game between two players $\exists$ and $\forall$, that we name[10] Eloise and Abelard. Both players instantiate the corresponding quantifiers in turns, Eloise for defending the formula and Abelard for attacking it. The game, whose depth is bounded by the number of quantifications, proceeds as follows:

- Eloise has to give an integer $m \in \mathbb{N}$ to instantiate the existential quantifier, and the game goes on over the closed formula $\forall y.f(m) \leq f(y)$.
- Abelard has to give an integer $n \in \mathbb{N}$, and the game goes on the closed formula $f(m) \leq f(n)$.
- Eloise has then two choices: either she backtracks to the first step to give another instantiation m' for x, and the game goes on; or she chooses to interrupt the game. If so, Eloise wins if $\mathcal{M} \vDash f(m) \leq f(n)$, otherwise Abelard wins. If the game goes on forever, Abelard wins.

Observe that the fact Eloise wins the game on a position (m,n) does not mean that m is a minimum for the function f: it only means that Abelard failed in finding an integer n such that $f(n) < f(m)$. Nonetheless, if Eloise actually knows that some integer m is a minimum for f, she will obviously win the game regardless of what Abelard plays.

We say that a player has a *winning strategy* if (s)he has a way of playing that ensures him/her the victory independently of the opponent moves, which corresponds to the definition of Coquand's game [27]. It is obvious from Tarski's definition of truth (see Section 1.2) that the closed formula Φ_f is valid in the ground model if and only if Eloise has a winning strategy.

Intuitively, Eloise is playing as a realizer should, and Abelard is an opponent choosing amongst falsity values. This intuition can be formalized by implementing the previous game within the λ_c-calculus. A realizer will then corresponds to a winning strategy for Eloise, and reciprocally.

Relativization to canonical integers The implementation of the previous game in the λ_c-calculus actually requires a preliminary step. Indeed, as such first-order quantifications are not given any computational content: integers are instantiated in formulas which are only evaluated in the end within the ground model. To make these integers appear in the computations, we need to relativize first-order quantifications to the class $\mathsf{Nat}(x)$ (just like in Section 3.3.3.1). However, if we have as expected $\bar{n} \Vdash \mathsf{Nat}(n)$ for any $n \in \mathbb{N}$, there are realizers of $\mathsf{Nat}(n)$ different from $\bar{n}$. Intuitively, a term $t \Vdash \mathsf{Nat}(n)$ represents the integer n, but n might be present only as a computation, and not directly as a computed value.

The usual technique to retrieve $\bar{n}$ from such a term consist in the use of a *storage operator T*, which simulates a call-by-value reduction (for integers) on the first argument on the stack. While such a term is easy to define, it make the the definition of the game harder, and we do not want to bother the reader with such technical details[11]. Rather than that, we define a new asymmetrical implication where the left member must be an integer value (somehow forcing call-by-value reduction on all integers), and

[10] The names Eloise and Abelard are due to Thierry Coquand, who also defined the game in question [27].

[11] For further details about the relativization and storage operator, please refer to Section 2.9 and 2.10.1 of Rieg's Ph.D. thesis [144].

the interpretation of this new implication.

Formulas	$A,B ::= \dots \mid \{e\} \to A$
Falsity value	$\|\{e\} \to A\| \triangleq \{\bar{n} \cdot \pi : [\![e]\!] = n \wedge \pi \in \|A\|\}$

We finally define the corresponding shorthands for relativized quantifications:

$$\begin{array}{rcl} \forall^{\mathbb{N}} x\, A(x) & \triangleq & \forall x\, (\{x\} \to A(x)) \\ \exists^{\mathbb{N}} x\, A(x) & \triangleq & \forall Z\, (\forall x(\{x\} \to A(x) \to Z) \to Z) \end{array}$$

which is easy to check to be equivalent (in terms of realizability) to the one defined in Section 3.3.3.1 [65].

Realizability game In order to play using realizers, we will slightly change the setting of the previous game, adding processes. One should notice that we only add more information, so that this new game is somewhat a "decorated" version of the previous one.

To describe the match, we use processes which evolve throughout the match according to the following rules:

1. Eloise proposes a term $t_0 \in$ PL supposed to defend Φ_f and Abelard proposes a stack $u_0 \cdot \pi_0$ supposed to attack the formula Φ. We say that at time 0, the process $p_0 := t_0 \star u_0 \cdot \pi_0$ is the current process.
2. Assume that p_i is the current process. Eloise evaluates p_i in order to reach one of the following situations:

   - $p_i \succ u_0 \star \overline{m} \cdot t \cdot \pi$. If so, Eloise *can* decide to play by communicating her answer (t,m) to Abelard and standing for his answer, and Abelard *must* answer a new integer n together with a new stack $u' \cdot \pi'$. The current process then becomes $p_{i+1} := t \star \overline{n} \cdot u' \cdot \pi'$.
   - $p_i \succ u \star \pi$ for some u,π that were previously played by Abelard in a position in which x,y were instantiated by (m,n). In this case, Eloise wins if $\mathcal{M} \models f(m) \leq f(n)$.

   If none of the above moves is possible, then Abelard wins.

Starting with a term t is a "good move" for Eloise if and only if, proposed as a defender of the formula, t defines an initial winning state (for Eloise), independently from the initial stack proposed by Abelard. In this case, adopting the point of view of Eloise, we just say that t is a *winning strategy* for the formula Φ_f.

This furnishes us an answer to the specification problem for the formula Φ_f: winning strategies of this game exactly characterized the realizer of the formula Φ_f.

Theorem 3.24. *If a closed λ_c-term t is a winning strategy for Eloise if and only if $t \Vdash \Phi_f$.*

Proof. This is a particular case of the more general case of arithmetical formulas proved in [65]. □

3.5.3 Model theory

Up to this point, we only presented applications of Krivine realizability on its computational side. Yet, we explained that realizability offered a way to build models for second-order logic, (this can actually be extended, for instance for set theory [93]). More interestingly, classical realizability appears to be a generalization of Cohen's technique of forcing, introduced to construct a model of set theory in which the continuum hypothesis[12] is not valid. As shown by Krivine [98] and Miquel [120], the forcing

[12] The continuum hypothesis expresses the fact that there is no set whose cardinality would be strictly more than the cardinal of $\mathbb{N}$ and strictly less than the cardinal of $\mathcal{R}$.

construction can be computationally analyzed as a program transformation in the framework of the λ_c-calculus. In particular, classical realizability can simulate any forcing construction[13].

Even more surprising is the fact that the realizability semantics lead to the construction of new models, studied by Krivine in a series of papers [98, 99, 100, 101]. Briefly, the fact that $\forall x.\mathsf{Nat}(x)$ is not realized witnesses that a model has more individuals than the natural numbers. In a well-chosen model[14] $\mathcal{M}_{⫫}$, one can show that $\mathcal{M}_{⫫} \models \mathsf{Nat}(n)$ for any $n \in \mathbb{N}$ while $\mathcal{M}_{⫫} \models \exists x.\neg\mathsf{Nat}(x)$. Otherwise said, the model attests the presence of unnamed elements. It turns out that this allows to define "pathological" infinite sets[15] $\nabla_n \triangleq \{x : x < n\}$ such that the following statements are valid for any $n, m \in \mathbb{N}$:

1. ∇_2 is not well-ordered
2. there is an injection from ∇_n to ∇_{n+1}
3. there is no surjection from ∇_n to ∇_{n+1}
4. $\nabla_m \times \nabla_n \simeq \nabla_{mn}$

These sets being subsets of $\mathcal{P}(\mathbb{N})$, observe that the first property implies that the axiom of choice (AC) is not valid, while items 2 and 3 prove that the continuum hypothesis (CH) is not valid either [99].

As far as we know, usual techniques to construct model of set theory do not allow to define directly a model in which both (AC) and (CH) are not valid. Besides, a construction by means of forcing can not break the axiom of choice, hence classical realizability is a strict generalization of forcing in this sense. For these reasons amongst others, classical realizability tends to be a promising framework to build new models. In particular, it justifies our quest (Part III) for an algebraic structure as general as possible in which the λ_c-calculus and these constructions can be embedded.

[13] An example of this is the extraction of Herbrand tree by forcing in [143].

[14] In Krivine's papers, it is the model of threads, in which each proof-like term t_n is associated with a stack constant α_n and the pole is defined as $⫫ \triangleq \bigcap_{n\in\mathbb{N}}(\mathbf{th}(t_n \star \alpha_n))^c$. This set is indeed a valid pole (see Example 3.7) and is consistent according to Proposition 3.17.

[15] In the ground model or any standard model, ∇_n is just $\{0, 1, ..., n-1\}$ *i.e.* n from a set-theoretic point of view.

4- The $\lambda\mu\tilde{\mu}$-calculus

4.1 Sequent calculus

4.1.1 Gentzen's LK calculus

The sequent calculus was originally introduced by Gentzen [56, 57] who was trying to reformulate the system of natural deduction in a more symmetric presentation. He was looking at the time for a proof of normalization for the natural deduction system in order to prove the coherence of first-order arithmetic. The principal novelty of this system is that it gives an equal importance to left and right parts (hypotheses and conclusions) of sequents. In particular, sequents are of the form $\Gamma \vdash \Delta$, where both Γ and Δ are sequences of formulas. Besides, the deductive system does no longer make the distinction between introduction and elimination rules but is only compound of (left and right) introduction rules. Intuitively, a sequent is provable if the conjunction of hypotheses on the left entails the disjunction of (possible) conclusions on the right. More precisely, we can define the formula associated to the sequent $A_1,\ldots,A_n \vdash B_1,\ldots,B_p$ as the formula $A_1 \wedge \ldots \wedge A_n \to B_1 \vee \ldots \vee B_p$, and prove the previous statement, namely that a sequent is valid if and only if its associated formula is valid (Proposition 4.3). To put it differently, a sequent $\Gamma \vdash \Delta$ is intuitively derivable if there is a formula in Δ that is provable using the hypotheses in Γ.

4.1.1.1 Language

In the original presentation of Gentzen [56, 57], who was interested in first-order arithmetic, first-order expressions and binary predicates where defined by the following grammar:

$$\begin{array}{lrcl} \textbf{Terms} & t,u & ::= & x \mid n \in \mathbb{N} \mid t+u \mid t-u \mid t\times u \\ \textbf{Predicates} & P & ::= & t = u \mid t < u \end{array}$$

As explained in Section 1.1.1, this corresponds to the axiomatic part of a theory. Here we rather want to deal with the deductive part of the proof system, that is the set of inferences rules that encompasses the logical part of the theory. Hence we shall consider the generic case of first-order logic formulas (see Example 1.2), which are built from a fixed set $\mathcal{V}$ of variables and a fixed signature Σ_1 for first-order terms, and from a signature Σ_2 for predicates:

$$\begin{array}{lrcll} \textbf{Terms} & e_1,e_2 & ::= & x \mid f(e_1,\ldots,e_k) & (x \in \mathcal{V}, f \in \Sigma_1) \\ \textbf{Predicates} & A,B & ::= & P(e_1,\ldots,e_k) \mid \forall x.A \mid \exists x.A \mid A \to B \mid A \wedge B \mid A \vee B & (P \in \Sigma_2) \end{array}$$

A sequent, written $\Gamma \vdash \Delta$, is a pair of two (possibly empty) lists of formulas Γ and Δ, defined by:

$$\Gamma,\Delta ::= \varepsilon \quad\mid\quad \Gamma,A$$

Identity rules

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta}\,(\textsc{Cut}) \qquad\qquad \frac{}{A \vdash A}\,(\textsc{Ax})$$

Structural rules

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\,(w_r) \qquad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\,(c_r) \qquad \frac{\Gamma \vdash \sigma(\Delta)}{\Gamma \vdash \Delta}\,(\sigma_r)$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\,(w_l) \qquad \frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\,(c_l) \qquad \frac{\sigma(\Gamma) \vdash \Delta}{\Gamma \vdash \Delta}\,(\sigma_l)$$

Logical rules

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\,(\neg_r) \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta}\,(\to_r) \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}\,(\wedge_r) \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}\,(\vee_r)$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\,(\neg_l) \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \to B \vdash \Delta}\,(\to_l) \qquad \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\,(\wedge_l) \qquad \frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}\,(\vee_l)$$

$$\frac{\Gamma \vdash A, \Delta \quad x \notin FV(\Gamma, \Delta)}{\Gamma \vdash \forall x.A, \Delta}\,(\forall_r) \qquad \frac{\Gamma, A[t/x] \vdash \Delta}{\Gamma, \forall x.A \vdash \Delta}\,(\forall_l) \qquad \frac{\Gamma \vdash A[t/x], \Delta}{\Gamma \vdash \exists x.A, \Delta}\,(\exists_r) \qquad \frac{\Gamma, A \vdash \Delta \quad x \notin FV(\Gamma, \Delta)}{\Gamma, \exists x.A \vdash \Delta}\,(\exists_l)$$

Figure 4.1: Gentzen LK calculus

4.1.1.2 Deductive system

The rules of Gentzen deductive system, given in Figure 4.1 and named LK, are splitted in three groups:

- *identity rules*, which specify the two pure manners of proving a sequent, namely reducing to an hypothesis or by introducing a *cut* over a formula;
- *structural rules*, which correspond to contexts management: they allows us to weaken, rearrange (σ is a permutation) or duplicate formulas within left and right contexts;
- *logical rules*, which are the left and right introduction rules for logical connectives.

Intuitively, a sequent $\Gamma \vdash \Delta$ is derivable if there is a formula in Δ that is provable using the hypotheses in Γ. This intuition is actually valid up to the subtlety that we do not necessarily know which formula of the right-handside is proven. In fact, there is not necessarily one specific formula that is proven, but rather a superposition of formulas. For instance, as we shall see a derivation of the sequent $\vdash A(x) \vee \neg A(x)$ proves neither $A(x)$ nor $\neg A(x)$, it only proves that for any x, one of both is true. If $A(x)$ is the formula *"the cat is alive at the instant x"*, we are in presence of a Schrödinger's cat[1].

This presentation is indeed more symmetric than natural deduction, in that it highlights the dual behaviors of hypothesis and conclusions. This observation will be reflected through the proofs-as-programs interpretation of sequent calculus in the next section. Lastly, this deduction system encompasses classical logic. In particular, it is easy to derive proofs for the excluded-middle, the double-negation elimination or the law of Peirce (see Figure 4.2). Actually, the case of intuitionistic logic, named LJ, corresponds to the same calculus where only one formula is allowed in the right-hand side of sequents.

As an example to illustrate the construction of proof derivations in LK, we shall now prove the claim that a sequent is provable if and only if its associate formula is.

[1] We are very grateful to Alexandre Miquel for this very nice metaphor.

$$\dfrac{\dfrac{\dfrac{}{A \vdash A}\,(\mathrm{Ax})}{\vdash A, \neg A}\,(\neg_r)}{\vdash A \lor \neg A}\,(\lor_r)$$

(a) Excluded-middle

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,(\mathrm{Ax})}{\vdash A, \neg A}\,(\neg_r)}{\neg(\neg A) \vdash A}\,(\neg_l)}{\vdash \neg(\neg A) \to A}\,(\to_r)$$

(b) Double-negation elimination

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,(\mathrm{Ax})}{A \vdash B, A}\,(w_r)}{\vdash A \to B, A}\,(\to_r) \qquad \dfrac{}{A \vdash A}\,(\mathrm{Ax})}{(A \to B) \to A \vdash A}\,(\to_l)}{\vdash ((A \to B) \to A) \to A}\,(\to_r)$$

(c) Peirce's law

Figure 4.2: Proof of classical principles in LK

Definition 4.1 (Admissible rule)**.** A rule is said to be *admissible* in a proof system if there exists a derivation of its conclusion using its hypotheses as axioms. ⌟

Lemma 4.2. *The following rules are admissible in LK:*

$$\dfrac{A \in \Gamma}{\Gamma \vdash A}\,(Ax_r) \qquad\qquad \dfrac{A \in \Delta}{A \vdash \Delta}\,(Ax_l) \qquad\qquad \dfrac{A \in \Gamma \quad A \in \Delta}{\Gamma \vdash \Delta}\,(Ax)$$

Proof. We only give the proof for the first rule. Knowing that $A \in \Gamma$ we can assume that Γ is of the general form $B_1, \ldots, B_n, A, C_1, \ldots, C_p$ and prove the first rule as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,(\mathrm{Ax}_l)}{\vdots\ (w_l)}}{\dfrac{A, B_1, \ldots, B_n, C_1, \ldots, C_{p-1} \vdash A}{A, B_1, \ldots, B_n, C_1, \ldots, C_{p-1}, C_p \vdash A}\,(w_l)}}{B_1, \ldots, B_n, A, C_1, \ldots, C_{p-1}, C_p \vdash A}\,(\sigma_l)$$

Proofs for the other two cases are very similar. □

Proposition 4.3 (Associated formula)**.** *A sequent* $\Gamma \vdash \Delta$ *is valid if and only if its associated formula is valid.*

Proof. The proof on the left-to-right part is left as an exercise for the willful reader. We only give the right-to-left proof in the case where Γ and Δ both contains two formulas:

$$\dfrac{\vdash A_1 \land A_2 \to B_1 \lor B_2 \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A_1, A_2 \vdash A_1}\,(\mathrm{Ax}_r) \quad \dfrac{}{A_1, A_2 \vdash A_2}\,(\mathrm{Ax}_r)}{A_1, A_2 \vdash A_1 \land A_2}\,(\land_r)}{A_1, A_2 \vdash A_1 \land A_2, B_1, B_2}\,(w_r) \qquad \dfrac{\dfrac{\dfrac{}{B_2 \vdash B_1, B_2}\,(\mathrm{Ax}_l)}{B_1 \lor B_2 \vdash B_1, B_2}\,(\lor_l)}{A_1, A_2, B_1 \lor B_2 \vdash B_1, B_2}\,(w_r)}{A_1, A_2, A_1 \land A_2 \to B_1 \lor B_2 \vdash B_1, B_2}\,(\to_l)}{A_1, A_2 \vdash B_1, B_2}\,(\textsc{Cut})'$$

We implicitly use the fact that the following rule is admissible (which also is an easy exercise):

$$\dfrac{\vdash A \quad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta}\,(\textsc{Cut})'$$

□

4.1.2 Alternative presentation

In order to give a computational content to sequent calculus, we will use a slightly different presentation. While this presentation does not bring any logical benefits (it actually has the drawback of making the size of proofs grow), it forces the derivation to be somewhat more structured by preventing arbitrary changes of side (left or right) when applying inference rules. Quite the opposite, at any time is explicitly identified which formula is being worked on. In a nutshell, instead of considering one unique kind of sequent $\Gamma \vdash \Delta$, this presentation now distinguishes between three kinds of sequents:

Identity rules:

$$\frac{A \in \Delta}{\Gamma \mid A \vdash \Delta}\;(\mathrm{Ax}_l) \qquad \frac{A \in \Gamma}{\Gamma \vdash A \mid \Delta}\;(\mathrm{Ax}_r) \qquad \frac{\Gamma \vdash A \mid \Delta \qquad \Gamma \mid A \vdash \Delta}{\Gamma \vdash \Delta}\;(\textsc{Cut})$$

Structural rules:

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \mid A \vdash \Delta}\;(\mathtt{foc}_l) \qquad \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash A \mid \Delta}\;(\mathtt{foc}_r)$$

Logical rules:

$$\frac{\Gamma, A \vdash B \mid \Delta}{\Gamma \vdash A \to B \mid \Delta}\;(\to_r) \qquad \frac{\Gamma \vdash A \mid \Delta \quad \Gamma \vdash B \mid \Delta}{\Gamma \vdash A \wedge B \mid \Delta}\;(\wedge_r) \qquad \frac{\Gamma \vdash A \mid \Delta}{\Gamma \vdash A \vee B \mid \Delta}\;(\vee_r^1) \qquad \frac{\Gamma \vdash B \mid \Delta}{\Gamma \vdash A \vee B \mid \Delta}\;(\vee_r^2)$$

$$\frac{\Gamma \vdash A \mid \Delta \qquad \Gamma \mid B \vdash \Delta}{\Gamma \mid A \to B \vdash \Delta}\;(\to_l) \qquad \frac{\Gamma, A, B \vdash \Delta}{\Gamma \mid A \wedge B \vdash \Delta}\;(\wedge_l) \qquad \frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma \mid A \vee B \vdash \Delta}\;(\vee_l)$$

Figure 4.3: Sequent calculus with focus

$$\dfrac{\dfrac{}{(A \to B) \to A \vdash (A \to B) \to A \mid A}\;(\mathrm{Ax}_r) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(A \to B) \to A,\ A \vdash A \mid B}\;(\mathrm{Ax}_r) \quad \dfrac{}{(A \to B) \to A \mid A \vdash A, B}\;(\mathrm{Ax}_l)}{(A \to B) \to A,\ A \vdash A, B}\;(\textsc{Cut})}{(A \to B) \to A,\ A \vdash B \mid A}\;(\mu)}{(A \to B) \to A \vdash A \to B \mid A}\;(\to_r) \qquad \dfrac{}{(A \to B) \to A \mid A \vdash A}\;(\mathrm{Ax}_l)}{(A \to B) \to A \mid (A \to B) \to A \vdash A}\;(\to_l)}{\dfrac{\dfrac{(A \to B) \to A \vdash A}{(A \to B) \to A \vdash A \mid}\;(\mathtt{foc}_r)}{\vdash ((A \to B) \to A) \to A \mid}\;\to_r}\;(\textsc{Cut})$$

Figure 4.4: Peirce's law

1. sequents of the form $\Gamma \vdash A \mid \Delta$, where the focus is put on the (right) formula A;
2. sequents of the form $\Gamma \mid A \vdash \Delta$, where the focus is put on the (left) formula A;
3. sequents of the form $\Gamma \vdash \Delta$, where no focus is set.

In a right (resp. left) sequent $\Gamma \vdash A \mid \Delta$, the singled out formula[2] A reads as the conclusion *"where the proof shall continue"* (resp. hypothesis *"where it happened before"*). The rules of this sequent calculus with focus are given in Figure 4.3 for the propositional fragment. It is easy to check that any of the structural and identity rules of LK are admissible within this framework, and that any derivation in one system is derivable in the other. We could also have given the rules for first-order quantifications in the same way, but it is not the point here. Actually, neither did we include the negation rule, which we could have done directly. Another solution to retrieve the negation would be to add constant symbols $\top$ and $\bot$ with the following axioms:

$$\frac{}{\Gamma \mid \bot \vdash \Delta}\;(\bot) \qquad \frac{}{\Gamma \vdash \top \mid \Delta}\;(\top)$$

Then defining the negation by $\neg A \triangleq A \to \bot$, it is easy to check that the rules $(\neg_r)$ and $(\neg_l)$ are admissible.

To be fair, we should confess two things. First, that in itself, this presentation is mainly motivated here to make a transition to the type system of the $\lambda\mu\tilde{\mu}$-calculus, that we shall introduce in the next

[2] This formula is often referred to as the formula in the *stoup*, a terminology due to Girard [59].

section. That is, as a deductive system for mathematicians, this is LK buried under administrative duties. As an example to illustrate the difference between LK and this presentation, we give in Figure 4.4 the derivation tree for the law of Peirce, which is indeed bigger than its twin in LK. Second, we should mention that LK can be directly use as a type system for a calculus, namely Munch-Maccagnoni's system L [126]. If the second part of this thesis is presented in the framework of $\lambda\mu\tilde{\mu}$-calculus, it could as well have been rephrased entirely using system L, of which we use fragments in the third part. In other words, the current section is motivated by the sole purpose of making obvious the equivalence between both presentations.

4.2 The $\lambda\mu\tilde{\mu}$-calculus

We shall now present the $\lambda\mu\tilde{\mu}$-calculus, originally introduced by Curien and Herbelin [32] to emphasize implicit symmetries of computation such as the duality between programs and contexts or the duality between call-by-name and call-by-value evaluation strategies. One of the huge advantages that this calculus has over the usual λ-calculus is that its reduction system comes directly in the form of an abstract machine. As we will discuss in the next sections, this is particularly convenient when it comes to the definition of a realizability interpretation or of a continuation-passing style translation. Actually, this also was one of the starting observation that led to the very definition of the $\lambda\mu\tilde{\mu}$-calculus[3]: when it comes to abstract machines, the evolution of types has much more to do with sequent calculus than with natural deduction. Consider for instance the rules (Push) and (Grab) of Krivine abstract machine:

$$\begin{array}{llcl}
(\textsc{Push}) & tu \star \pi & \succ & t \star u \cdot \pi \\
(\textsc{Grab}) & (\lambda x\,.\,t) \star u \cdot \pi & \succ & t[u/x] \star \pi
\end{array}$$

In the first rule, if u has type A and π type B, then resulting stack $u \cdot \pi$ is of type $A \to B$: this is a left-introduction rule of implication. Then the second rule reads as a cut between two implications which have been introduced on each side:

$$\dfrac{\dfrac{\Gamma, x : A \vdash t : B \mid \Delta}{\Gamma \vdash \lambda x.t : A \to B \mid \Delta}{\scriptstyle(\to_r)} \qquad \dfrac{\Gamma \vdash u : A \mid \Delta \quad \Gamma \mid \pi : B \vdash \Delta}{\Gamma \mid u \cdot \pi : A \to B \vdash \Delta}{\scriptstyle(\to_l)}}{(\lambda x.t \star u \cdot \pi) : (\Gamma \vdash \Delta)}{\scriptstyle(\textsc{Cut})}$$

where we make use of the three kinds of sequents from last section.

4.2.1 Syntax

The syntax of the $\lambda\mu\tilde{\mu}$-calculus, just like the one of the λ_c-calculus, is divided in three categories: *terms* (or proofs), which represent programs; *evaluation contexts*[4] (or co-proofs), which represent environments of execution; *commands*, which are pairs consisting of a term and a context and represent a closed system containing both the program and its environment. Formally, terms, contexts and commands are defined by the following grammar:

$$\begin{array}{llcl}
\textbf{Terms} & p & ::= & a \mid \lambda a.p \mid \mu\alpha.c \\
\textbf{Contexts} & e & ::= & \alpha \mid p \cdot e \mid \tilde{\mu}a.c \\
\textbf{Commands} & c & ::= & \langle p \| e \rangle
\end{array}$$

where variables $a, b, \dots$ and co-variables $\alpha, \beta, \dots$ range over two fixed alphabets. To draw the parallel with the λ_c-calculus and the Curry-Howard correspondence, a command is a process or a state of an

[3] See the introduction of [32].

[4] We draw the reader's attention to the fact that the terminology of *contexts* is already overloaded, and we insist on the fact that here they refer to co-terms. Nonetheless, the usual notion of evaluation contexts (see Remark 2.5) and this one are not disconnected, since both refer to the environment in which a term is evaluated.

abstract machine, representing the evaluation of a proof (the program) against a co-proof (the context). The notion of evaluation context is a generalization of the notion of stacks where $\tilde{\mu}a.c$ can be read as a context `let` $a = [\,]$ `in` c. As for terms, the μ operator comes from Parigot's $\lambda\mu$-calculus [131], $\mu\alpha$ binds a context to a context variable α in the same way $\tilde{\mu}a$ binds a proof to some proof variable a. In particular, as we shall see now, it allows to capture evaluation contexts and as such is a control operator which plays a role similar to `call/cc`.

4.2.2 Reduction rules and evaluation strategies

The reduction rules of the $\lambda\mu\tilde{\mu}$-calculus are parameterized by a particular set of proofs, written $\mathcal{V}$, and a particular set of contexts, written $\mathcal{E}$:

$$\begin{array}{lcll}
\langle p \| \tilde{\mu}a.c\rangle & \to & c[p/a] & (p \in \mathcal{V})\\
\langle \mu\alpha.c \| e\rangle & \to & c[e/\alpha] & (e \in \mathcal{E})\\
\langle \lambda a.p \| u \cdot e\rangle & \to & \langle u \| \tilde{\mu}a.\langle p \| e\rangle\rangle &
\end{array}$$

If $\mathcal{V}$ and $\mathcal{E}$ are not restricted enough, these rules admit a critical pair:

$$\begin{tikzcd}
& \langle \mu\alpha.c \| \tilde{\mu}a.c'\rangle \arrow[dl] \arrow[dr] & \\
c[\tilde{\mu}a.c'/\alpha] & & c'[\mu\alpha.c/a]
\end{tikzcd}$$

Unlike the λ-calculus, the $\lambda\mu\tilde{\mu}$-calculus is clearly not confluent: in the above critical pair, if $c = \langle b \| \beta\rangle$ and $c' = \langle d \| \gamma\rangle$ for distinct variables, then the reduction is blocked after one step for each command and $c \neq c'$. Moreover, the critical pair can be interpreted in terms of non-determinism. Indeed, we can define a fork instruction by $\pitchfork \triangleq \lambda ab.\mu\alpha.\langle \mu\_.\langle a \| \alpha\rangle \| \tilde{\mu}\_.\langle b \| \alpha\rangle\rangle$, which verifies indeed that:

$$(\textsc{Fork}) \qquad \langle \pitchfork \| p_0 \cdot p_1 \cdot e\rangle \to \langle p_0 \| e\rangle \qquad \text{and} \qquad \langle \pitchfork \| p_0 \cdot p_1 \cdot e\rangle \to \langle p_1 \| e\rangle.$$

The difference between call-by-name and call-by-value can be characterized by how this critical pair is solved, by defining $\mathcal{V}$ and $\mathcal{E}$ in such a way that the two rules do not overlap. This justifies the definition of a subcategory V of proofs, that we call *values*, and of the dual subset E of contexts that we call *co-values*:

$$\text{(Values)} \quad V ::= a \mid \lambda a.p \qquad\qquad \text{(Co-values)} \quad E ::= \alpha \mid q \cdot e$$

The call-by-name evaluation strategy amounts to the case where $\mathcal{V} \triangleq$ *Proofs* and $\mathcal{E} \triangleq$ *Co-values.* This is reflected in the reduction of the command where a function is applied to a stack:

$$\langle \lambda a.p \| u \cdot e\rangle \;\to\; \langle u \| \tilde{\mu}a.\langle p \| e\rangle\rangle \;\to\; \langle p[u/a] \| e\rangle$$

We observe that the variable is substituted no matter what by the proof u (unreduced). Dually, the call-by-value corresponds to $\mathcal{V} \triangleq$ *Values* and $\mathcal{E} \triangleq$ *Contexts.* In this case, assuming that the proof u reduces[5] to a value V_u, the previous command will reduce as follows:

$$\langle \lambda a.p \| u \cdot e\rangle \;\to\; \langle u \| \tilde{\mu}a.\langle p \| e\rangle\rangle \;\overset{*}{\to}\; \langle V_u \| \tilde{\mu}a.\langle p \| e\rangle\rangle \;\to\; \langle p[V_u/a] \| e\rangle$$

where the substitution in p is done only after u has reduced. If u does not reduce to a value in front of $\tilde{\mu}a.\langle p \| e\rangle$ (which is the case if u drops its evaluation context), this substitution never happens.

Finally, it is worth noting that the μ binder is a *control operator*, since it allows for catching evaluation contexts and backtracking further in the execution. This is then the key ingredient that makes the $\lambda\mu\tilde{\mu}$-calculus a proof system for classical logic, as the continuation-passing style translation or the embedding of `call/cc` will emphasize in the next sections.

[5] That is to say that for any command e, the command $\langle u \| e\rangle$ reduces to $\langle V_u \| e\rangle$.

$$\frac{\Gamma \vdash p : A \mid \Delta \qquad \Gamma \mid e : A \vdash \Delta}{\langle p \| e \rangle : (\Gamma \vdash \Delta)}\ (\textsc{Cut})$$

$$\frac{(a : A) \in \Gamma}{\Gamma \vdash a : A \mid \Delta}\ (\mathrm{Ax}_r) \qquad \frac{\Gamma, a : A \vdash p : B \mid \Delta}{\Gamma \vdash \lambda a.p : A \to B \mid \Delta}\ (\to_r) \qquad \frac{c : (\Gamma \vdash \Delta, \alpha : A)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\ (\mu)$$

$$\frac{(\alpha : A) \in \Delta}{\Gamma \mid \alpha : A \vdash \Delta}\ (\mathrm{Ax}_l) \qquad \frac{\Gamma \vdash p : A \mid \Delta \qquad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid p \cdot e : A \to B \vdash \Delta}\ (\to_l) \qquad \frac{c : (\Gamma, a : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}a.c : A \vdash \Delta}\ (\tilde{\mu})$$

Figure 4.5: The simply-typed $\lambda\mu\tilde{\mu}$-calculus

4.2.3 Type system

4.2.3.1 Two-sided sequents

The type system for the simply-typed $\lambda\mu\tilde{\mu}$-calculus, given in Figure 4.5, corresponds exactly to the deductive system of sequent calculus with focus in Figure 4.3. It is therefore the programming counterpart of a proof-as-program correspondence between sequent calculus and abstract machines. Commands are typed by the (Cut) rule, right introduction rules correspond to typing rules for proofs, while left introduction rules are typing rules for evaluation contexts. The duality between hypotheses and conclusion in the sequent calculus is thus directly reflected into the duality between proofs and contexts.

4.2.3.2 One-sided sequents

The very same type system can be expressed through one-sided sequents, where hypotheses in Γ and Δ are regrouped in a same context, written $\Gamma \cup \Delta$, where hypotheses $\alpha : A$ formerly in Δ are distinguished with an annotation on the type: $\alpha : A^{\perp\!\!\!\perp}$. The typing rules are the same, except that the three kinds of sequents are now denoted by:

$$\Gamma \vdash p : A \qquad\qquad \Gamma \vdash e : A^{\perp\!\!\!\perp} \qquad\qquad \Gamma \vdash c$$

In the case of simple types, the ordering of hypotheses is irrelevant, in the sense that any sequent derivable with a context Γ would also be derivable with $\sigma(\Gamma)$ for any permutation σ. However, if necessary (for instance with dependent types), it is always possible to consider that hypotheses are introduced with an index so that $\Gamma \cup \Delta$ is defined to match the order of introduction of the hypotheses. Technically, it suffices to redefine inferences rules to include these indices, for instance:

$$\frac{c : (\Gamma \vdash \Delta, \alpha :_n A) \quad |\Gamma| + |\Delta| = n}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}$$

This allows us to define a function `join` by:

$$\begin{array}{rcl} \mathtt{join}((a :_n A, \Gamma), \Delta, n) & = & (a : A), \mathtt{join}(\Gamma, \Delta, n+1) \\ \mathtt{join}(\Gamma, (\alpha :_n A, \Delta), n) & = & (\alpha : A^{\perp\!\!\!\perp}), \mathtt{join}(\Gamma, \Delta, n+1 \\ \mathtt{join}(\varepsilon, \varepsilon, n) & = & \varepsilon \end{array}$$

and we let $\Gamma \cup \Delta \triangleq \mathtt{join}(\Gamma, \Delta, 0)$. One-sided or two-sided sequents are then essentially a matter of taste. In the next chapters we will mostly use two-sided sequents, because they are closer to the original presentations of LK or the $\lambda\mu\tilde{\mu}$-calculus. Yet, we always consider that contexts are implicitly numbered so that we can make use of $\Gamma \cup \Delta$ in the right order if needed.

$$
\dfrac{
\dfrac{}{a : (A \to B) \to A \vdash a : (A \to B) \to A \mid \bullet}\,(\textsc{Ax}_r)
\qquad
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\bullet, a' : A \vdash a' : A \mid \bullet}\,(\textsc{Ax}_r)
\quad
\dfrac{}{\bullet \mid \alpha : A \vdash \alpha : A, \bullet}\,(\textsc{Ax}_l)
}{\langle a' \| \alpha \rangle : (\bullet, a' : A \vdash \alpha : A, \beta : B)}\,(\textsc{Cut})
}{\bullet, a' : A \vdash \mu\beta.\langle a' \| \alpha \rangle : B \mid \alpha : A)}\,\mu
}{\bullet \vdash \lambda a'.\mu\beta.\langle a' \| \alpha \rangle \mid \alpha : A}\,\to_r
\quad
\dfrac{}{\mid \alpha : A \vdash \alpha : A}\,(\textsc{Ax}_l)
}{\bullet \mid \lambda a'.\mu\beta.\langle a' \| \alpha \rangle \cdot \alpha : (A \to B) \to A \vdash \alpha : A)}\,\to_l
}{
\dfrac{
\dfrac{\langle a \| \lambda a'.\mu\beta.\langle a' \| \alpha \rangle \cdot \alpha \rangle : (a : (A \to B) \to A \vdash \alpha : A)}{a : (A \to B) \to A \vdash \mu\alpha.\langle a \| \lambda a'.\mu\beta.\langle a' \| \alpha \rangle \cdot \alpha \rangle : A \mid}\,\mu
}{\vdash \lambda a.\mu\alpha.\langle a \| \lambda a'.\mu\beta.\langle a' \| \alpha \rangle \cdot \alpha \rangle : ((A \to B) \to A) \to A \mid}\,\to_r
}\,(\textsc{Cut})
$$

Figure 4.6: Proof term for Peirce's law

4.2.4 Embedding of the λ_c-calculus

In order to get more familiar with the syntax and computation of the $\lambda\mu\tilde{\mu}$-calculus, let us draw the analogy with the λ_c-calculus. Let us begin by embedding the syntax of the call-by-name Krivine abstract machine for λ-terms (that is without `call/cc`). The embedding $[\![\cdot]\!]$ is straightforward:

$$
\begin{array}{rcl|rcl|rcl}
[\![t \star \pi]\!] & \triangleq & \langle [\![t]\!] \| [\![\pi]\!] \rangle & [\![\lambda x.t]\!] & \triangleq & \lambda x.[\![t]\!] & [\![\alpha]\!] & \triangleq & \alpha \\
[\![x]\!] & \triangleq & x & [\![t\,u]\!] & \triangleq & \mu\alpha.\langle [\![t]\!] \| [\![u]\!] \cdot \alpha \rangle & [\![t \cdot \pi]\!] & \triangleq & [\![t]\!] \cdot [\![e]\!]
\end{array}
$$

It is then an easy exercise to check that typing judgments are preserved through the embedding[6], and it also easily verified that in the call-by-name setting, reductions are also preserved:

$$
\begin{array}{lrclcl}
(\textsc{Push}) & [\![tu \star \pi]\!] & = & \langle \mu\alpha.\langle [\![t]\!] \| [\![u]\!] \cdot \alpha \rangle \| [\![\pi]\!] \rangle & \to & \langle [\![t]\!] \| [\![u]\!] \cdot [\![\pi]\!] \rangle \; = \; [\![t \star u \cdot \pi]\!] \\
(\textsc{Grab}) & [\![\lambda x\,.\,t \star u \cdot \pi]\!] & = & \langle \lambda x.[\![t]\!] \;\|\; [\![u]\!] \cdot [\![\pi]\!] \rangle & \xrightarrow{2} & \langle [\![t]\!][[\![u]\!]/x] \| [\![\pi]\!] \rangle \; = \; [\![t[u/x] \star \pi]\!]
\end{array}
$$

Actually, the full λ_c calculus can be retrieved since the `call/cc` operator and continuation constants $\mathbf{k}_\pi$ can also be soundly embedded. Interestingly, by being more atomic the syntax of the $\lambda\mu\tilde{\mu}$-calculus forces us to define both terms in a way that the corresponding reductions rules:

$$
\begin{array}{llcl}
(\textsc{Save}) & \texttt{call/cc} \star t \cdot \pi & \succ & t \star \boldsymbol{k}_\pi \cdot \pi \\
(\textsc{Restore}) & \boldsymbol{k}_\pi \star t \cdot \pi' & \succ & t \star \pi
\end{array}
$$

are decomposed into elementary steps. Indeed, let us define the following proof terms:

$$
\texttt{call/cc} \triangleq \lambda a.\mu\alpha.\langle a \| \boldsymbol{k}_\alpha \cdot \alpha \rangle \qquad\qquad \boldsymbol{k}_e \triangleq \lambda a'.\mu_{\_}.\langle a' \| e \rangle
$$

and set $[\![\mathtt{cc}]\!] \triangleq$ `call/cc` and $[\![\mathbf{k}_\pi]\!] \triangleq \mathbf{k}_{[\![\pi]\!]}$. As expected, `call/cc` can be typed with Peirce's law (see Figure 4.6), as a matter of fact its very definition is obtained from the proof of Peirce's law in Figure 4.4 through Curry-Howard isomorphism. Let us observe the computational behavior of `call/cc`: in front of a context of the right shape (that is a stack $q \cdot e$ with e of type A), it catches the context e thanks to the $\mu\alpha$ binder and reduces as follows:

$$
\langle \texttt{call/cc} \| q \cdot e \rangle \;=\; \langle \lambda a.\mu\alpha.\langle a \| \boldsymbol{k}_\alpha \cdot \alpha \rangle \| q \cdot e \rangle \quad\to\quad \langle \mu\alpha.\langle q \| \boldsymbol{k}_\alpha \cdot \alpha \rangle \| e \rangle \quad\to\quad \langle q \| \boldsymbol{k}_e \cdot e \rangle
$$

In particular, if $q \cdot e \;=\; [\![t \cdot \pi]\!]$, we recognize the (Save) rule. Notice also that the proof term now on top of the stack $\boldsymbol{k}_e \;=\; \lambda a'.\mu_{\_}.\langle a' \| e \rangle$ (which, if e was of type A, is of type $A \to B$, see Figure 4.6) contains

[6] That is to say that if a typing judgment $\Gamma \vdash t : A$ is derivable then $\Gamma \vdash [\![t]\!] : A \mid \varepsilon$ is derivable within the $\lambda\mu\tilde{\mu}$-calculus. To be precise, this would require to restrict to simple types for t or to extend the $\lambda\mu\tilde{\mu}$-calculus type system to second-order, but in fact both lead to the desired result.

a second binder $\mu_\_$. In front of a stack $q' \cdot e'$, this binder will now catch the context e' and replace it by the former context e:

$$\langle \boldsymbol{k}_e \| q' \cdot e' \rangle \;=\; \langle \lambda a'.\mu_\_.\langle a' \| e \rangle \| q' \cdot e' \rangle \quad \to \quad \langle \mu_\_.\langle q' \| e \rangle \| e' \rangle \quad \to \quad \langle q' \| e \rangle$$

Here again, we recognize exactly the (Restore) rule of the λ_c-calculus. For both cc and $\mathbf{k}_\pi$ (and both reduction rules), their definitions in the $\lambda\mu\tilde{\mu}$-calculus is more atomic and highlights that these terms computes in two elementary steps: they first grab (by means of a λ abstraction) a term t on the stack, then they capture the evaluation context e (by means of a μ abstraction) and reduce accordingly to their specification (`call/cc` furnishes to t the continuation $\mathbf{k}_e$ while $\mathbf{k}_{e'}$ drops the continuation context and let t be evaluated in the (restored) context e').

4.2.5 Soundness

When defining a proof system by means of a calculus, one should necessarily proceed to a sanity check. It is standard to consider a calculus safe if it enjoys properties such as type safety (like subject reduction), soundness and normalization, which correspond respectively to the following questions: Is the reduction system correct with respect to the type system? Is there a proof of false? Does the typing ensure normalization of terms?

There are actually many ways to answer each of these questions. Let us briefly present three of them. The first option is to prove everything directly, from scratch. The property of subject reduction is usually proved by a cautious induction over the reduction rules, with a bunch of auxiliary lemmas about substitution. Assuming that the normalization holds, it can be combined with subject reduction to prove the soundnes: if there was a proof of false then this proof can be reduced to a term in normal form (normalization) which is also a proof of false (subject reduction). Then if suffices to show that there is no such term. Finally, the normalization is proved by any possible means (most of the time it is the hardest part), for instance by a combinatorial argument, like identifying a decreasing quantity on the typing derivation, or by adapting one the following techniques.

A second technique consists in the definition of a realizability interpretation for the calculus. While the interpretation can be tricky in itself to define and prove adequate, in the end the adequacy generally gives normalization and soundness for free.

A third solution relies on the definition of an embedding into another proof system for which these properties holds. Then, if the translation is adequate in the sense that it preserves types and reduction, the normalization of the target calculus ensures the one of the source, and the non existence of a proof of false (or the corresponding translated type) in the target language should also ensure the soundness of the source language. Aside from proving these properties, an interest of this technique is that it might decompose or reduce difficulties of the source calculus (for instance the presence of control operators) into well-known pieces of the target calculus (for instance the simply-typed λ-calculus). A standard class of such embeddings are the continuation-passing style translations that we shall now present.

We will then take the call-by-name and call-by-value $\lambda\mu\tilde{\mu}$-calculi as examples, and use both a continuation-passing style translation and a realizability interpretation in each case to prove that these calculi enjoy the properties of soundness and normalization.

4.3 Continuation-passing style translation

4.3.1 Principles

In the realm of the proofs-as-programs correspondence, continuation-passing style (CPS) translations are twofold: they bring both a program translation and a logical translation. We shall first focus on the computational aspect, and emphasize the logical side in the next section. As a program translation, continuation-passing style translations are a well-known class of computational reductions from a calculus to another one. In particular, they have a lot of application in terms of compilation. The terminology was first introduced in 1975 by Sussman and Steele in a technical report about the Scheme programming language [152]. They illustrate this technique with the example of the factorial. Using a mixed notation between pseudo-code and λ-calculus[7], a standard recursive definition of the factorial is given by:

$$\texttt{fact\_aux} \;:=\; \lambda n.\texttt{if}\ n\ =\ 0\ \texttt{then}\ 1\ \texttt{else}\ n \times \texttt{fact}\,(n-1)$$

It is easy to check that `fact` computes correctly the factorial, for instance when applied to 3 it reduces as follows:

$$\texttt{fact}\ 3 \;\to\; 3 \times \texttt{fact}\ 2 \;\to\; 3 \times 2 \times \texttt{fact}\,1 \;\to\; 3 \times 2 \times 1 \times \texttt{fact}\ 0 \;\to\; 3 \times 2 \times 1 \;\to\; 6$$

However, there is another way to drive the same computation forward, which Sussman and Steele [152] describe by:

> *"It is always possible, if we are willing to specify explicitly what to do with the answer, to perform any calculation in this way: rather than reducing to its value, it reduces to an application of a continuation to its value. That is, in this continuation-passing programming style, a function always "returns" its result by "sending" it to another function. This is the key idea."*

This corresponds to this alternative definition of the factorial:

$$\texttt{fact} \;:=\; \lambda nk.\texttt{if}\ n\ =\ 0\ \texttt{then}\ k\,1\ \texttt{else}\ \texttt{fact}\,(n-1)\,(\lambda r.k\,(n \times r))$$

where the abstracted variable k is expecting the *continuation* as an argument. A continuation is a function waiting for the return value to drive the computation forward. In other words, from the point of view of the program, a continuation is a term that reifies the future of the computation. For instance, when applied to 3 and a function `answer` as continuation, the execution thread of `fact` is now:

$$\begin{array}{rcl}
\texttt{fact}\ 3\,\texttt{answer} & \to & \texttt{fact}\ 2\,(\lambda r.\,\texttt{answer}\,(3 \times r)) \\
& \to & \texttt{fact}\ 1\,(\lambda r.(\lambda r.\,\texttt{answer}\,(3 \times r))\,(2 \times r)) \\
& \to & \texttt{fact}\ 0\,(\lambda r.(\lambda r.(\lambda r.\,\texttt{answer}\,(3 \times r))\,(2 \times r)\,(1 \times r)) \\
& \to & (\lambda r.(\lambda r.(\lambda r.\,\texttt{answer}\,(3 \times r))\,(2 \times r))\,(1 \times r))\,1 \\
& \to & (\lambda r.(\lambda r.\,\texttt{answer}\,(3 \times r))\,(2 \times r))\,1 \\
& \to & (\lambda r.\,\texttt{answer}\,(3 \times r))\,2 \\
& \to & \texttt{answer}\,6
\end{array}$$

We notice that if the first argument n is different from 0, `fact` makes a recursive call to itself with $n-1$ and a new continuation that is waiting for the answer r to compute the product $n \times r$ and return it to the former continuation[8]. This idea could of course be generalized to translate as well the arithmetic primitives: any integer n could be transformed into the function $\overline{n} := \lambda k.k\,n$ that expects a continuation and apply this continuation to n. Similarly, the multiplication operator could be transformed into an operator $\overline{\times}$ waiting for the translations $\overline{n},\overline{m}$ of two integers and a continuation k, furnishing to $\overline{n}$ and $\overline{m}$ the adequate continuations to extract their values and finally return the multiplication to k: $\overline{\times} :=$

[7] This could be formally embedded in the $\lambda^{\times+}$-calculus with integers, but there is no interest in being so formal here.

[8] In fact, we could optimize the continuation in the continuation-passing style translated form of the factorial to obtain an alternative definition of the factorial function, which has the same computational behavior of without continuation:

$$\begin{array}{rcl}
\texttt{fact} & := & \lambda n.\texttt{fact\_aux}\,n\,1 \\
\texttt{fact\_aux} & := & \lambda mr.\texttt{if}\ m\ =\ 0\ \texttt{then}\ r\ \texttt{else}\ \texttt{fact\_aux}\,(n-1)\,(n \times r)
\end{array}$$

In that case, the function `fact` is said to be tail-recursive, and reduces as follows:

$$\texttt{fact}\ 3 \;\to\; \texttt{fact\_aux}\ 3\,1 \;\to\; \texttt{fact\_aux}\ 2\,3 \;\to\; \texttt{fact\_aux}\ 1\,6 \;\to\; \texttt{fact\_aux}\ 0\,6 \;\to\; 6$$

where we skipped the arithmetic reductions.

$\lambda tuk.t\,(\lambda n.u\,(\lambda m.k\,(n\times m)))$. Again, when applied to a continuation answer and the translation of 3 and 2, this term will compute the expected result by passing of continuations along the execution:

$$\begin{array}{rcl}
\overline{\times}\,\overline{3}\,\overline{2}\,\texttt{answer} & \to & \overline{3}\,(\lambda n.\overline{2}\,(\lambda m.\texttt{answer}\,(n\times m))) \\
& \to & (\lambda n.\overline{2}\,(\lambda m.\texttt{answer}\,(n\times m)))\,3 \\
& \to & \overline{2}\,(\lambda m.\texttt{answer}\,(3\times m)) \\
& \to & (\lambda m.\texttt{answer}\,(3\times m))\,2 \\
& \to & \texttt{answer}\,6
\end{array}$$

It is worth noting that the continuation-passing style translation also proposes an operational semantics in that it makes explicit the order in which the reduction steps are computed. In particular, different evaluation strategies correspond to different continuation-passing style translations[9]. This was studied by Plotkin for the call-by-name and call-by-value strategies within the λ-calculus [139], and we shall recall in the sequel the corresponding translations for the $\lambda\mu\tilde{\mu}$-calculus [32].

In addition to the operational semantics, continuation-passing style translations allow to benefit from properties already proved for the target calculus. Besides, the passing of continuations provides a way to handle the flow of control, and in particular to embed control operators (like `call/cc` or the μ operator). For instance, we will see how to define translations $p \mapsto [\![p]\!]$ from the simply-typed $\lambda\mu\tilde{\mu}$-calculus (the *source* language) to the simply-typed λ-calculus (the *target* language) along which the properties of normalization and soundness can be transfered. In details, these translations will preserve reduction, in that a reduction step in the source language gives rise to a step (or more) in the target language:

$$c \xrightarrow{1} c' \qquad \Rightarrow \qquad [\![c]\!] \xrightarrow{+}_{\beta} [\![c']\!] \tag{4.1}$$

We will say that a translation is *typed* when it comes with a translation $A \mapsto [\![A]\!]$ from types of the source language to types of the target language, such that a typed proof in the source language is translated into a typed proof of the target language:

$$\Gamma \vdash p : A \mid \Delta \qquad \Rightarrow \qquad [\![\Gamma]\!], [\![\Delta]\!] \vdash [\![p]\!] : [\![A]\!] \tag{4.2}$$

Lastly, these translations will map the type $\bot$ into a type $[\![\bot]\!]$ which is not inhabited:

$$\nvdash p : [\![\bot]\!] \tag{4.3}$$

Assuming that the previous properties hold, one automatically gets:

Theorem 4.4 (Benefits of the translation)**.** *If the target language of the translation is sound and normalizing, and if besides the equations (4.1), (4.2) and (4.3) hold, then:*

1. *If $[\![p]\!]$ normalizes, then p normalizes*
2. *If p is typed, then p normalizes*
3. *The source language is sound,* i.e. *there is no proof $\vdash p : \bot$*

Proof.
1. By contrapositive, if p does not normalizes, then according to equation (4.1) neither does $[\![p]\!]$.
2. If p is typed, then $[\![p]\!]$ is also typed by(4.2), and thus normalizes. Using the first item, p normalizes.
3. By *reductio ad absurdum*, direct consequence of (4.3). □

[9] For instance, in our example the translation of the operator × corresponds to the call-by-name translation, because it is waiting for the unevaluated translations of 3 and 2 and takes the responsibility of evaluating them when needed. On the opposite, the call-by-value translation $\overline{\times} := \lambda nmk.k\,(n\times m)$ would have been waiting directly for integers (values) and the application of a function to its argument (that is the translation of $t\,u$) should then have been in charge of performing the evaluation of the argument: $\overline{t\,u} := \lambda k.\overline{u}\,\lambda v.\overline{t}\,v\,k$.

4.3.2 The underlying negative translation

As mentioned in the last paragraphs, continuation-passing style translations have their logical counterpart, since they induce a translation on formulas. If we observe for instance the translation of 2, defined as $\lambda k.k\,2$, we see that it now expects a continuation waiting for an integer (atomic type `nat`) whose return type is unknown, say R. That is, the atomic type `nat` is translated into:

$$\overline{\mathsf{nat}} \triangleq (\mathsf{nat} \to R) \to R$$

As for the multiplication operator, its translation $\overline{\times}$, which is waiting for two translated integers and a continuation is now of type:

$$\overline{(\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat})} \triangleq \overline{\mathsf{nat}} \to \overline{\mathsf{nat}} \to (\mathsf{nat} \to R) \to R \;\; = \;\; \overline{\mathsf{nat}} \to \overline{\mathsf{nat}} \to \overline{\mathsf{nat}}$$

In the case where R is taken to be $\bot$, this corresponds exactly to Gödel-Gentzen negative translation ϕ^N of formula:

$$\begin{array}{rcll|rcl}
\phi^N & \triangleq & \neg\neg\phi & (\phi \text{ atomic}) & & & \\
(\phi \to \psi)^N & \triangleq & \phi^N \to \psi^N & & (\neg\phi)^N & \triangleq & \neg\phi^N \\
(\phi \vee \psi)^N & \triangleq & \neg(\neg\phi^N \wedge \neg\psi^N) & & (\forall x.\phi)^N & \triangleq & \forall x.\neg\phi^N \\
(\phi \wedge \psi)^N & \triangleq & (\phi^N \wedge \psi^N) & & (\exists x.\phi)^N & \triangleq & \neg(\forall x.\neg\phi^N)
\end{array}$$

This translation actually defines an embedding of classical (first-order) logic into intuitionistic (first-order) logic, in the sense that if $\mathscr{T}$ is a set of axioms, then the sequent $\mathscr{T} \vdash \Phi$ is provable in LK if and only if the translated sequent $\mathscr{T}^N \vdash \Phi^N$ is provable in LJ (intuitionistic sequent calculus). This is to be related with the fact that it allows to embed control operators in the λ-calculus. Since classical logic is computationally obtained from intuitionistic logic (λ-calculus) by addition of a control operator, it is quite natural that a sound embedding of the calculus with control operator back to the λ-calculus defines an embedding of classical logic within intuitionistic logic.

4.3.3 The benefits of semantic artifacts

Continuation-passing style translations are thus a powerful tool both on the computational and the logical facets of the proofs-as-programs correspondence, which we use in the forthcoming sections to prove normalization and soundness of the $\lambda\mu\tilde{\mu}$-calculus. Rather than giving directly the appropriate definitions, we would like to insist on a convenient methodology to obtain CPS translations as well as realizability interpretations (which are deeply connected). This methodology is directly inspired from Danvy *et al* method to derive hygienic semantics artifacts for a call-by-need calculus [37]. Reframed in out setting, it essentially consists in the successive definitions of:

1. an operational semantics,
2. a small-step calculus or abstract machine,
3. a continuation-passing style translation,
4. a realizability model.

The first step is nothing more than the usual definition of a reduction system. The second step consists in refining the reduction system to obtain small-step reduction rules (as opposed to big-step ones), that are finer-grained reduction steps. These steps should be as atomic as possible, and in particular, they should correspond to an abstract machine in which the sole analysis of the term (or the context) should determine the reduction to perform. Such a machine is called in *context-free form* [37]. If so, the definition of a CPS translation is almost straightforward, as well as the realizability interpretation. Let us now illustrate this methodology on the call-by-name and call-by-value $\lambda\mu\tilde{\mu}$-calculi.

4.4 The call-by-name $\lambda\mu\tilde{\mu}$-calculus

4.4.1 Reduction rules

We recall here the (big-step) reduction rules of the call-by-name $\lambda\mu\tilde{\mu}$-calculus (Section 4.2.2), where the $\tilde{\mu}$ operator gets the priority over the μ operator:

$$\begin{array}{ccc}
\langle p \| \tilde{\mu}a.c\rangle & \to & c[p/a] \\
\langle \mu\alpha.c \| E\rangle & \to & c[E/\alpha] \\
\langle \lambda a.p \| q \cdot e\rangle & \to & \langle q \| \tilde{\mu}a.\langle p \| e\rangle\rangle
\end{array}$$

As such, these rules define an abstract machine which is not in context-free from since to reduce a command one need to analyze simultaneously what is the term and what is the context.

4.4.2 Small-step abstract machine

To alleviate this ambiguity, we will refine the reduction system into small-step rules in which it is always specified which part of the command is being analyzed. If we examine the big-step rules, the only case where the knowledge of only one side suffices: when the context is of the form $\tilde{\mu}a.c$, which has the absolute priority. So that we can start our analysis of a command by looking at its left-hand side. If it is a $\tilde{\mu}a.c$, we reduce it, otherwise, we can look at the right-hand side. Now, if the term is of the shape $\mu\alpha.c$, it should be reduced, otherwise, we can analyze the left-hand side again. The only case left is when the context is a stack $q \cdot e$ and the term is a function $\lambda a.p$, in which case the command reduces.

The former case suggests two things: first, that the reduction should proceed by alternating examination of the left-hand and the right-hand side of commands. Second, that there is a descent in the syntax from the most general level (context e) to the most specific one (values[10] V), passing by p and E in the middle:

Terms	p	$::=$	$\mu\alpha.c \mid a \mid V$	**Contexts**	e	$::=$	$\tilde{\mu}a.c \mid E$
Values	V	$::=$	$\lambda a.p$	**Co-values**	E	$::=$	$\alpha \mid p \cdot e$

So as to stick to this intuition, we denote commands with the level of syntax we are examining (c_e, c_t, c_E, c_V), and define a new set of reduction rules which are of two kinds: computational steps, which reflect the former reduction steps, and administrative steps, which organize the descent in the syntax. For each level in the syntax, we define one rule for each possible construction. For instance, at level e, there is one rule if the context is of the shape $\tilde{\mu}a.c$, and one rule if it is of shape E. This results in the following set of small-step reduction rules:

$$\begin{array}{ccc}
\langle p \| \tilde{\mu}a.c\rangle_e & \rightsquigarrow & c_e[p/a] \\
\langle p \| E\rangle_e & \rightsquigarrow & \langle p \| E\rangle_p \\
\langle \mu\alpha.c \| E\rangle_p & \rightsquigarrow & c_e[E/\alpha] \\
\langle V \| E\rangle_p & \rightsquigarrow & \langle V \| E\rangle_E \\
\langle V \| q \cdot e\rangle_E & \rightsquigarrow & \langle V \| q \cdot e\rangle_V \\
\langle \lambda a.p \| q \cdot e\rangle_V & \rightsquigarrow & \langle q \| \tilde{\mu}a.\langle p \| e\rangle\rangle_e
\end{array}$$

where the last two rules could be compressed in one rule:

$$\langle \lambda a.p \| q \cdot e\rangle_E \rightsquigarrow \langle q \| \tilde{\mu}a.\langle p \| e\rangle\rangle_e$$

Note that there is no rule for variables and co-variables, since they block the reduction. It is obvious that theses rules are indeed a decomposition of the previous ones, in the sense that if c, c' are two commands such that $c \xrightarrow{1} c'$, then there exists $n > 1$ such that $c \overset{n}{\rightsquigarrow} c'$.

[10] Observe that values usually include variables, but here we rather consider them in the category p. This is due to the fact that the operator $\tilde{\mu}$ catches proofs at level p and variables are hence intended to be substituted by proofs at this level. Through the CPS, we will see that we actually need values to be considered at level p as they are indeed substituted by proofs translated at this level.

4.4.3 Call-by-name type system

The previous subdivision of the syntax and reductions also suggests a fine-grained type system, where sequents are annotated with the adequate syntactic categories:

$$\frac{\Gamma \vdash_V V : A \mid \Delta}{\Gamma \vdash_p V : A \mid \Delta}(V) \qquad \frac{(a : A) \in \Gamma}{\Gamma \vdash_p a : A \mid \Delta}(\mathrm{Ax}_r) \qquad \frac{c : (\Gamma \vdash_c \Delta, \alpha : A)}{\Gamma \vdash_p \mu\alpha.c : A \mid \Delta}(\mu) \qquad \frac{\Gamma, a : A \vdash_p p : B \mid \Delta}{\Gamma \vdash_V \lambda a.p : A \to B \mid \Delta}(\to_r)$$

$$\frac{\Gamma \mid E : A \vdash_E \Delta}{\Gamma \mid E : A \vdash_e \Delta}(E) \qquad \frac{c : (\Gamma, a : A \vdash_c \Delta)}{\Gamma \mid \tilde{\mu}a.c : A \vdash_e \Delta}(\tilde{\mu}) \qquad \frac{(\alpha : A) \in \Delta}{\Gamma \mid \alpha : A \vdash_E \Delta}(\mathrm{Ax}_l) \qquad \frac{\Gamma \vdash_p p : A \mid \Delta \quad \Gamma \mid e : B \vdash_e \Delta}{\Gamma \mid p \cdot e : A \to B \vdash_E \Delta}(\to_l)$$

While this does not bring any benefit when building typing derivations (when collapsed at level e and p, this type system is exactly the original one), it has the advantage of splitting the rules in more atomic ones which are closer from the reduction system. Hence it will be easier to prove that the CPS translation is typed using these rules as induction bricks.

4.4.4 Continuation-passing style translation

4.4.4.1 Translation of terms

Once we have an abstract-machine in context-free form at hands, the corresponding continuation-passing style translation is straightforward. It suffices to start from the higher level in the descent (here e) and to define a translation for each level which, for each element of the syntax, simply describe the corresponding small-step rule. In the current case, this leads to the following definition:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\llbracket \tilde{\mu}a.c \rrbracket_e\, p & \triangleq & (\lambda a.\llbracket c \rrbracket_c)\, p & \llbracket V \rrbracket_p\, E & \triangleq & E\, \llbracket V \rrbracket_V \\
\llbracket E \rrbracket_e\, p & \triangleq & p\, \llbracket E \rrbracket_E & \llbracket q \cdot e \rrbracket_E\, V & \triangleq & V\, \llbracket q \rrbracket_p\, \llbracket e \rrbracket_e \\
\llbracket \mu\alpha.c \rrbracket_p\, E & \triangleq & (\lambda\alpha.\llbracket c \rrbracket_c)\, E & \llbracket \alpha \rrbracket_E & \triangleq & \alpha \\
\llbracket a \rrbracket_p & \triangleq & a & \llbracket \lambda a.p \rrbracket_V\, q\, e & \triangleq & (\lambda a.e\, \llbracket p \rrbracket_p)\, q
\end{array}$$

where administrative reductions peculiar to the translation (like continuation-passing) are compressed, and where $\llbracket \langle p \| e \rangle \rrbracket_c \triangleq \llbracket e \rrbracket_e\, \llbracket p \rrbracket_p$. The expanded version is simply:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\llbracket \tilde{\mu}a.c \rrbracket_e & \triangleq & \lambda a.\llbracket c \rrbracket_c & \llbracket V \rrbracket_p & \triangleq & \lambda E.E\, \llbracket V \rrbracket_V \\
\llbracket E \rrbracket_e & \triangleq & \lambda p.p\, \llbracket E \rrbracket_E & \llbracket q \cdot e \rrbracket_E & \triangleq & \lambda V.V\, \llbracket q \rrbracket_p\, \llbracket e \rrbracket_e \\
\llbracket \mu\alpha.c \rrbracket_p & \triangleq & \lambda\alpha.\llbracket c \rrbracket_c & \llbracket \alpha \rrbracket_E & \triangleq & \alpha \\
\llbracket a \rrbracket_p & \triangleq & a & \llbracket \lambda a.p \rrbracket_V & \triangleq & \lambda qe.(\lambda a.e\, \llbracket p \rrbracket_p)\, q
\end{array}$$

This induces a translation of commands at each level of the translation:

$$\begin{array}{r@{\qquad\qquad}r}
\llbracket \langle p \| e \rangle \rrbracket_c^e \triangleq \llbracket e \rrbracket_e\, \llbracket p \rrbracket_p & \llbracket \langle V \| E \rangle \rrbracket_c^E \triangleq \llbracket E \rrbracket_E\, \llbracket V \rrbracket_V \\
\llbracket \langle p \| E \rangle \rrbracket_c^p \triangleq \llbracket p \rrbracket_p\, \llbracket E \rrbracket_E & \llbracket \langle V \| q \cdot e \rangle \rrbracket_c^V \triangleq \llbracket V \rrbracket_V\, \llbracket q \rrbracket_p\, \llbracket e \rrbracket_e
\end{array}$$

which is easy to prove correct with respect to computation, since the translation is defined from the reduction rules. We first prove that substitution is sound through the translation, and then prove that the whole translation preserves the reduction.

Lemma 4.5. *For any variable a (co-variable α) and any proof q (co-value E), the following holds for any command c:*

$$\llbracket c[q/a] \rrbracket_c \;=\; \llbracket c \rrbracket_c[\llbracket q \rrbracket_p/a] \qquad\qquad \llbracket c[E/\alpha] \rrbracket_c \;=\; \llbracket c \rrbracket_c[\llbracket E \rrbracket_E/\alpha]$$

The same holds for substitution within proofs and contexts.

Proof. Easy induction on the syntax of commands, proofs and contexts, the key cases corresponding to (co-)variables:

$$[\![\alpha]\!]_e[[\![E]\!]_E/\alpha] \;=\; (\lambda p.p\,\alpha)[[\![E]\!]_E/\alpha] \;=\; \lambda p.p\,[\![E]\!]_E \;=\; [\![E]\!]_e \;=\; [\![\alpha[E/\alpha]]\!]_e$$

□

Proposition 4.6. *For all levels ι, o of e, p, E, and any commands c, c', if $c_\iota \overset{1}{\rightsquigarrow} c'_o$, then $[\![c]\!]_c^\iota \xrightarrow{+}_\beta [\![c']\!]_c^o$.*

Proof. The proof is an easy induction on the reduction $\rightsquigarrow$. Administrative reductions are trivial, the cases for μ and $\tilde{\mu}$ correspond to the previous lemma, which leaves us with the case for λ:

$$[\![\langle\lambda a.p \| q\cdot e\rangle]\!]_c^V = (\lambda qe.(\lambda a.e\,[\![p]\!]_p)\,q)\,[\![q]\!]_p\,[\![e]\!]_e \xrightarrow{2}_\beta (\lambda a.[\![e]\!]_e\,[\![p]\!]_p)\,[\![q]\!]_p = [\![\langle q \| \tilde{\mu}a.\langle p \| e\rangle\rangle]\!]_c^e$$

□

4.4.4.2 Translation of types

The computational translation induces the following translation on types:

$$\begin{array}{lll}
[\![A]\!]_e & \triangleq [\![A]\!]_p \to \bot & \\
[\![A]\!]_p & \triangleq [\![A]\!]_E \to \bot & \\
[\![A]\!]_E & \triangleq [\![A]\!]_V \to \bot & \\
[\![A \to B]\!]_V & \triangleq [\![A]\!]_p \to [\![B]\!]_e \to \bot & \\
[\![X]\!]_V & \triangleq X & (X \text{ variable})
\end{array}$$

where we take $\bot$ as return type for continuations. This extends naturally to typing contexts, where the translation of Γ is defined at level p while Δ is translated at level E:

$$[\![\Gamma, a : A]\!]_p \triangleq [\![\Gamma]\!]_p, a : [\![A]\!]_p \qquad\qquad [\![\Delta, \alpha : A]\!]_E \triangleq [\![\Delta]\!]_E, \alpha : [\![A]\!]_E$$

As we did not include any constant of atomic types, the choice for the translation of atomic types is somehow arbitrary, and corresponds to the idea that a constant $\boldsymbol{c}$ would be translated into $\lambda k.k\,\boldsymbol{c}$. We could also have translated atomic types at level p, with constants translated as themselves. In any case, the translation of proofs, contexts and commands is well-typed:

Proposition 4.7. *For any contexts Γ and Δ, we have*

1. *if* $\Gamma \vdash p : A \mid \Delta$ *then* $[\![\Gamma]\!]_p, [\![\Delta]\!]_E \vdash [\![p]\!]_p : [\![A]\!]_p$
2. *if* $\Gamma \mid e : A \vdash \Delta$ *then* $[\![\Gamma]\!]_p, [\![\Delta]\!]_E \vdash [\![e]\!]_e : [\![A]\!]_e$
3. *if* $c : \Gamma \vdash \Delta$ *then* $[\![\Gamma]\!]_p, [\![\Delta]\!]_E \vdash [\![c]\!]_c : \bot$

Proof. The proof is done by induction over the typing derivation. We can refine the statement by using the type system presented in Section 4.4.3, and proving two additional statements: if $\Gamma \vdash_V V : A \mid \Delta$ then $[\![\Gamma]\!]_p, [\![\Delta]\!]_E \vdash [\![V]\!]_V : [\![A]\!]_p$ (and similarly for E). We only give two cases, other cases are easier or very similar.

- **Case** c. If $c = \langle p \| e\rangle$ is a command typed under the hypotheses Γ, Δ:

$$\frac{\Gamma \vdash_p p : A \mid \Delta \qquad \Gamma \mid e : A \vdash_e \Delta}{\langle p \| e\rangle : \Gamma \vdash_c \Delta}\ \text{(Cut)}$$

then by induction hypotheses for e and p, we have that $[\![\Gamma]\!]_p, [\![\Delta]\!]_E \vdash [\![e]\!]_e : [\![A]\!]_p \to \bot$ and that $[\![\Gamma]\!]_p, [\![\Delta]\!]_E \vdash [\![p]\!]_p : [\![A]\!]_p$, thus we deduce that $[\![\Gamma]\!]_p, [\![\Delta]\!]_E \vdash [\![e]\!]_e\,[\![p]\!]_p : \bot$.

- **Case** V. If $\lambda a.p$ has type $A \to B$:

$$\dfrac{\Gamma, a : A \vdash_p p : B \mid \Delta}{\Gamma \vdash_V \lambda a.p : A \to B \mid \Delta}\ {\scriptstyle (\to_r)}$$

then by induction hypothesis, we get that $\llbracket \Gamma \rrbracket_p, \llbracket \Delta \rrbracket_E, a : \llbracket A \rrbracket_p \vdash \llbracket p \rrbracket_p : \llbracket B \rrbracket_p$. By definition, we have $\llbracket \lambda a.p \rrbracket_V = \lambda qe.(\lambda a.e\, \llbracket p \rrbracket_p)\, q$, which we can type:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{e : \llbracket B \rrbracket_e \vdash e : \llbracket B \rrbracket_p \to \bot}\ {\scriptstyle (\mathrm{Ax})} \qquad \llbracket \Gamma \rrbracket_p, \llbracket \Delta \rrbracket_E, a : \llbracket A \rrbracket_p \vdash \llbracket p \rrbracket_p : \llbracket B \rrbracket_p}{\llbracket \Gamma \rrbracket_p, \llbracket \Delta \rrbracket_E, e : \llbracket B \rrbracket_e, a : \llbracket A \rrbracket_p \vdash e\, \llbracket p \rrbracket_p : \bot}\ {\scriptstyle (\to_E)}}{\llbracket \Gamma \rrbracket_p, \llbracket \Delta \rrbracket_E, e : \llbracket B \rrbracket_e \vdash \lambda a.e\, \llbracket p \rrbracket_p : \llbracket A \rrbracket_p \to \bot}\ {\scriptstyle (\to_I)} \qquad \dfrac{}{q : \llbracket A \rrbracket_p \vdash q : \llbracket A \rrbracket_p}\ {\scriptstyle (\mathrm{Ax})}}{\dfrac{\llbracket \Gamma \rrbracket_p, \llbracket \Delta \rrbracket_E, q : \llbracket A \rrbracket_p, e : \llbracket B \rrbracket_e \vdash (\lambda a.e\, \llbracket p \rrbracket_p)\, q : \bot}{\llbracket \Gamma \rrbracket_p, \llbracket \Delta \rrbracket_E \vdash \lambda qe.(\lambda a.e\, \llbracket p \rrbracket_p)\, q : \llbracket A \rrbracket_p \to \llbracket B \rrbracket_e \to \bot}\ {\scriptstyle (\to_I)}}\ {\scriptstyle (\to_E)}$$

□

Up to this point, we already proved enough to obtain the normalization of the $\lambda\mu\tilde{\mu}$-calculus for the operational semantics considered:

Theorem 4.8 (Normalization). *Typed commands of the simply typed call-by-name $\lambda\mu\tilde{\mu}$-calculus are normalizing.*

Proof. By applying the generic result for translations (Theorem 4.4) since the required conditions are satisfied: the simply-typed λ-calculus is normalizing (Theorem 2.17), and Propositions 4.18 and 4.19 correspond exactly to equations (5.1) and (5.2). □

It only remains to prove that there is no term of the type $\llbracket \bot \rrbracket_p$ to ensure the soundness of the $\lambda\mu\tilde{\mu}$-calculus.

Proposition 4.9. *There is no term t in the simply typed λ-calculus such that $\vdash t : \llbracket \bot \rrbracket_p$.*

Proof. By definition, $\llbracket \bot \rrbracket_p = (\bot \to \bot) \to \bot$. Since $\lambda x.x$ is of type $\bot \to \bot$, if there was such a term t, then we would obtain $\vdash t\, \lambda x.x : \bot$, which is absurd. □

Theorem 4.10. *There is no proof p (in the simply typed call-by-name $\lambda\mu\tilde{\mu}$-calculus) such that $\vdash p : \bot \mid$.*

Proof. Simple application of Theorem 4.4. □

4.4.5 Realizability interpretation

We shall present in this section a realizability interpretation *à la* Krivine for the call-by-name $\lambda\mu\tilde{\mu}$-calculus. As Krivine classical realizability is naturally suited for a second-order setting, we shall first extend the type system to second-order logic. As we will see, the adequacy of the typing rules for universal quantification almost comes for free. However, we could also have sticked to the simple-typed setting, whose interpretation would have required to explicitly interpret each atomic type by a falsity value.

4.4.5.1 Extension to second-order

We first give the usual typing rules *à la* Curry for first- and second-order universal quantifications in the framework of the $\lambda\mu\tilde{\mu}$-calculus. Note that in the call-by-name setting, these rules are not restricted and defined at the highest levels of the hierarchy (e for context, p for proofs).

$$\dfrac{\Gamma \mid e : A[n/x] \vdash \Delta}{\Gamma \mid e : \forall x.A \vdash \Delta}\ {\scriptstyle (\forall_l^1)} \qquad\qquad \dfrac{\Gamma \vdash p : A \mid \Delta \quad x \notin FV(\Gamma, \Delta)}{\Gamma \vdash p : \forall x.A \mid \Delta}\ {\scriptstyle (\forall_r^1)}$$

$$\dfrac{\Gamma \mid e : A[B/X] \vdash \Delta}{\Gamma \mid e : \forall X.A \vdash \Delta}\ {\scriptstyle (\forall_l^2)} \qquad\qquad \dfrac{\Gamma \vdash p : A \mid \Delta \quad X \notin FV(\Gamma, \Delta)}{\Gamma \vdash p : \forall X.A \mid \Delta}\ {\scriptstyle (\forall_r^2)}$$

4.4.5.2 Realizability interpretation

We shall now present the realizability interpretation. As shown in Section 4.2.4, the call-by-name evaluation strategy allows to fully embed the λ_c-calculus. It is no surprise that the respective realizability interpretations for these calculi are very close. The major difference lies in the presence of the $\tilde{\mu}$ operator which has no equivalent in the λ_c-calculus, and forces to add a level in the interpretation. While we could directly state the definition and prove its adequacy, we rather wish to attract the reader attention to the fact that this definition is a consequence of the small-steps operational semantics. Indeed, going back to the intuition of a game underlying the definition of Krivine realizability, we are looking for sets of proofs (truth values) and set of contexts (falsity values) which are "well-behaved" against their respective opponents. That is, given a formula A, we are looking for players for A which compute "correctly" in front of any contexts opposed to A. If we take a closer look at the definition of the context-free abstract machine (Section 4.4.2), we see that the four levels e,p,E,V are precisely defined as sets of objects computing "correctly" in front of any object in the previous category: for instance, proofs in p are defined together with their reductions in front of any context in E. This was already reflected in the continuation-passing style translation. This suggests a four-level definition of the realizability interpretation, which we compact in three levels as the lowest level V can easily be inlined at level p (this was already the case in the small-step operational semantics and we could have done it also for the CPS).

The interpretation uses again the standard model $\mathbb{N}$ for the interpretation of first-order expressions and is parameterized by a pole $\perp\!\!\!\perp$, whose definition exactly matches the one for the λ_c-calculus:

Definition 4.11 (Pole). A *pole* is any subset $\perp\!\!\!\perp$ of commands which is closed by anti-reduction, that is for all commands c, c', if $c \in \perp\!\!\!\perp$ and $c \to c'$, then $c \in \perp\!\!\!\perp$. ⌟

We try to stick as much as possible to the notations and definitions of Krivine realizability. In particular, we define Π (the base set for falsity values) as the set of all co-values: $\Pi \triangleq E$. Falsity value functions, which are again defined as functions $F : \mathbb{N}^k \to \mathcal{P}(\Pi)$, are once more associated with predicate symbols $\dot{F}$, so that we use the very same definition of formulas with parameters. The interpretation of formulas with parameters is defined by induction on the structure of formulas:

$$\begin{array}{rcl}
\|\dot{F}(e_1,\ldots,e_k)\|_E & \triangleq & F(\llbracket e_1 \rrbracket,\ldots,\llbracket e_k \rrbracket) \\
\|A \to B\|_E & \triangleq & \{p \cdot e : \ p \in |A|_p \wedge e \in \|B\|_e\} \\
\|\forall x.A\|_E & \triangleq & \bigcup\limits_{n \in \mathbb{N}} \|A[n/x]\|_E \\
\|\forall X.A\|_E & \triangleq & \bigcup\limits_{F:\mathbb{N}^k \to \mathcal{P}(\Pi)} \|A[\dot{F}/X]\|_E \\
|A|_p & \triangleq & \|A\|_E^{\perp\!\!\perp} \ = \ \{p : \ \forall e \in \|A\|_E, \langle p\|e\rangle \in \perp\!\!\!\perp\} \\
\|A\|_e & \triangleq & |A|_p^{\perp\!\!\perp} \ = \ \{e : \ \forall e \in \|A\|_E, \langle p\|e\rangle \in \perp\!\!\!\perp\}
\end{array}$$

This definition exactly matches the one for the λ_c-calculus, considering that the "extra" level of interpretation $\|A\|_e$ is hidden in the latter, since all stacks are co-values. The expected monotonicity properties are satisfied:

Proposition 4.12 (Monotonicity). *For any formula A, the following hold:*

1. $\|A\|_E \subseteq \|A\|_e$
2. $|A|_p^{\perp\!\!\perp\perp\!\!\perp} = |A|_p$
3. $|\forall x.A|_p = \bigcap_{n \in \mathbb{N}} |A[n/x]|_p$
4. $|\forall X.A|_p = \bigcap_{F:\mathbb{N}^k \to \mathcal{P}(\Pi)} |A[\dot{F}/X]|_p$
5. $\|\forall x.A\|_e \supseteq \bigcup_{n \in \mathbb{N}} \|A[n/x]\|_e$
6. $\|\forall X.A\|_e \supseteq \bigcup_{F:\mathbb{N}^k \to \mathcal{P}(\Pi)} \|A[\dot{F}/X]\|_e$

Proof. These properties actually hold for arbitrary sets A and orthogonality relation $\perp$. Facts 1 and 2 are simply the usual properties of bi-orthogonal sets: $A \subseteq A^{\perp\perp}$ and $A^{\perp\perp\perp} = A^{\perp}$. Facts 3 and 4 are the usual equality $(\bigcup_{A\in\mathcal{A}} A)^{\perp} = \bigcap_{A\in\mathcal{A}} A^{\perp}$. Facts 5 and 6 are the inclusion $(\bigcap_{A\in\mathcal{A}} A)^{\perp} \supseteq \bigcup_{A\in\mathcal{A}} A^{\perp}$. □

A valuation is defined again as a function ρ which associates a natural number $\rho(x) \in \mathbb{N}$ to every first-order variable x and a falsity value function $\rho(X) : \mathbb{N}^k \to \mathcal{P}(\Pi)$ to every second-order variable X of arity k. As for substitutions, written σ, they now map variables to closed proofs (written $\sigma, a := p$) and co-variables to co-values (written $\sigma, \alpha := E$). We denote again by $A[\rho]$ (resp. $p[\sigma], e[\sigma, \ldots$) the closed formula (resp. proofs, context,...) where all variables are substituted by their values through ρ.

Given a closed (one-sided) context Γ, we say that a substitution σ realizes Γ, which we write $\sigma \Vdash \Gamma$, if for any $(a : A) \in \Gamma$, $\sigma(a) \in |A|_p$ and if for any $(\alpha : A^{\perp\!\!\perp}) \in \Gamma$, $\sigma(\alpha) \in \|A\|_E$. We are now equipped to prove the adequacy of the typing rules for the (call-by-name) $\lambda\mu\tilde{\mu}$-calculus with respect to the realizability interpretation we defined.

Proposition 4.13 (Adequacy)**.** *Let Γ, Δ be typing contexts, ρ be any valuation and σ be a substitution such that $\sigma \Vdash (\Gamma \cup \Delta)[\rho]$, then*

1. *if* $\Gamma \vdash p : A \mid \Delta$, *then* $p[\sigma] \in |A[\rho]|_p$
2. *if* $\Gamma \mid e : A \vdash \Delta$, *then* $e[\sigma] \in \|A[\rho]\|_e$
3. *if* $c : \Gamma \vdash \Delta$, *then* $c[\sigma] \in \perp\!\!\perp$

Proof. By mutual induction over the typing derivation.

- **Case** (Cut). We are in the following situation:

$$\frac{\Gamma \vdash p : A \mid \Delta \quad \Gamma \mid e : A \vdash \Delta}{\langle p \| e \rangle : \Gamma \vdash \Delta}\ (\textsc{Cut})$$

By induction, we have $p[\sigma] \in |A[\rho]|_p$ and $e[\sigma] \in \|A[\rho]\|_e$, thus $\langle p[\sigma] \| e[\sigma] \rangle \in \perp\!\!\perp$.

- **Case** (Ax_r). We are in the following situation:

$$\frac{(a : A) \in \Gamma}{\Gamma \vdash a : A \mid \Delta}\ (\mathrm{Ax}_r)$$

Since $\sigma \Vdash \Gamma[\rho]$, we deduce that $\sigma(a) \in |A|_p \subset |A[\rho]|$.

- **Case** (Ax_l). We are in the following situation:

$$\frac{(\alpha : A) \in \Delta}{\Gamma \mid \alpha : A \vdash \Delta}\ (\mathrm{Ax}_l)$$

Since $\sigma \Vdash \Delta[\rho]$, we deduce that $\sigma(\alpha) \in \|A[\rho]\|$.

- **Case** (μ). We are in the following situation:

$$\frac{c : (\Gamma \vdash \Delta, \alpha : A)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\ (\mu)$$

Let E be any context in $\|A[\rho]\|_E$, then $(\sigma, \alpha := E) \Vdash (\Gamma \cup \Delta)[\rho], \alpha : A^{\perp\!\!\perp}[\rho]$. By induction, we can deduce that $c[\sigma, \alpha := E] = (c[\sigma])[E/\alpha] \in \perp\!\!\perp$. By definition,

$$\langle (\mu\alpha.c)[\sigma] \| E \rangle = \langle \mu\alpha.c[\sigma] \| E \rangle \to c[\sigma][E/\alpha] \in \perp\!\!\perp$$

thus we can conclude by anti-reduction.

- **Case** $(\tilde\mu)$. We are in the following situation:

$$\frac{c : (\Gamma, a : A \Vdash \Delta)}{\Gamma \mid \tilde\mu a.c : A \vdash \Delta}\ (\tilde\mu)$$

Let p be a proof in $|A[\rho]|_p$, by assumption we have $(\sigma, a := p) \Vdash (\Gamma, a : A \cup \Delta)[\rho]$. As a consequence, we deduce from the induction hypothesis that $c[\sigma, a := p] = (c[\sigma])[p/a] \in \perp\!\!\!\perp$. By definition, we have:

$$\langle p \| (\tilde\mu a.c)[\sigma]\rangle \ = \ \langle p \| \tilde\mu a.c[\sigma]\rangle \to (c[\sigma])[p/a] \in \perp\!\!\!\perp$$

so that we can conclude by anti-reduction.

- **Case** $(\to_r)$. We are in the following situation:

$$\frac{\Gamma, a : A \vdash p : B \mid \Delta}{\Gamma \vdash \lambda a.p : A \to B \mid \Delta}\ (\to_r)$$

Let $q \cdot e$ be a stack in $\|(A \to B)[\rho]\|_E$, that is to say that $q \in |A[\rho]|_p$ and $e \in \|B[\rho]\|_e$. By definition, since $q \in |A[\rho]|_p$, we have $(\sigma, a := q) \Vdash (\Gamma, a : A \cup \Delta)[\rho]$. By induction hypothesis, this implies in particular that $p[\sigma, a := q] \in |B[\rho]|_p$ and thus $\langle p[\sigma, a := q] \| e\rangle \in \perp\!\!\!\perp$. We can now use the closure by anti-reduction to get the expected result:

$$\langle \lambda a.p[\sigma] \| q \cdot e\rangle \to \langle q \| \tilde\mu a.\langle p[\sigma] \| e\rangle\rangle \to \langle p[\sigma, a := q] \| e\rangle \in \perp\!\!\!\perp$$

- **Case** $(\to_l)$. We are in the following situation:

$$\frac{\Gamma \vdash q : A \mid \Delta \quad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid q \cdot e : A \to B \vdash \Delta}\ \to_E$$

By induction hypothesis, we obtain that $q[\sigma] \in |A[\rho]|_p$ and $e[\sigma] \in \|B[\rho]\|_e$. By definition, we thus have that $(q \cdot e)[\sigma] \in \|A \to B\|_E \subseteq \|A \to B\|_e$.

- **Case** $(\forall_r^1)$. We are in the following situation:

$$\frac{\Gamma \vdash p : A \mid \Delta \quad x \notin FV(\Gamma, \Delta)}{\Gamma \vdash p : \forall x.A \mid \Delta}\ (\forall_r^1)$$

By induction hypothesis, since $x \notin FV(\Gamma, \Delta)$, for any $n \in \mathbb{N}$ we have $(\Gamma \cup \Delta)[\rho, x \leftarrow n] = (\Gamma \cup \Delta)[\rho]$ and thus $\sigma \vdash (\Gamma \cup \Delta)[\rho, x \leftarrow n]$. We obtain by induction hypothesis that $p[\sigma] \in |A[\rho, x \leftarrow n]|_p$ for any $n \in \mathbb{N}$, *i.e.* that $p[\sigma] \in \bigcap_{n \in \mathbb{N}} |A[\rho, x \leftarrow n]|_p = |\forall x.A[\rho]|_p$. The case $(\forall_r^2)$ is identical to this one.

- **Case** $(\forall_l^1)$. We have that

$$\frac{\Gamma \mid e : A[n/x] \vdash \Delta}{\Gamma \mid e : \forall x.A \vdash \Delta}\ (\forall_l^1)$$

thus by induction hypothesis we get that $e[\sigma] \in \|(A[n/x])[\rho]\|_e$. Therefore we have in particular that $e[\sigma] \in \bigcup_{n \in \mathbb{N}} \|(A[n/x])[\rho]\|_e \subseteq \|\forall x.A[\rho]\|_e$ (Proposition 4.22). The case $(\forall_r^2)$ is identical to this one. □

Once the adequacy is proved, normalization and soundness almost come for free. The normalization is a direct corollary of the following observation, whose proof is the same as for Proposition 6.9:

Proposition 4.14. *The set* $\perp\!\!\!\perp_{\Downarrow} \triangleq \{c : c \text{ normalizes}\}$ *of normalizing commands defines a valid pole.*

Theorem 4.15 (Normalization)**.** *For any contexts* Γ, Δ *and any command* c*, if* $c : \Gamma \vdash \Delta$*, then* c *normalizes.*

Proof. By adequacy, any typed command c belongs to the pole $\perp\!\!\!\perp_{\Downarrow}$ modulo the closure under a substitution σ realizing the typing contexts. It suffices to observe that to obtain a closed term, any free variable a of type A in c can be substituted by an inert constant $\boldsymbol{a}$ which will realize its type (since it forms a normalizing command in front of any E in $\|A\|_E$). Thus $c[\boldsymbol{a}/a, \boldsymbol{b}/b, \ldots]$ normalizes and so does c. □

Similarly, the soundness is an easy consequence of adequacy, since the existence of a proof p of type $\perp = \forall X.X$ would imply that $p \in |\perp|_p$ for any pole $\perp\!\!\!\perp$. For any consistent pole (say the empty pole), this is absurd.

Theorem 4.16 (Soundness). *There is no proof p (in the second-order call-by-name $\lambda\mu\tilde{\mu}$-calculus) such that $\vdash p : \perp \mid$.*

For what concerns the induced model, it is worth noting that the notion of proof-like terms for the λ_c-calculus corresponds to closed proofs in the $\lambda\mu\tilde{\mu}$-calculus. Indeed, recall that continuation constants are translated by $\boldsymbol{k}_e \triangleq \lambda a'.\mu\_.\langle a' \| e\rangle$, where e necessarily contains a free co-variable (or a stack bottom if we had included co-constants in our syntax). The restriction to closed realizers is thus enough to obtain a sound model.

4.5 The call-by-value $\lambda\mu\tilde{\mu}$-calculus

We shall now reproduce this approach for the call-by-value $\lambda\mu\tilde{\mu}$-calculus. Since most of the steps are very similar, we will try to be briefer in this section.

4.5.1 Reduction rules

We recall the reductions rules for the call-by-value evaluation strategy, in which μ gets the priority over $\tilde{\mu}$:

$$\begin{array}{ccc}
\langle \mu\alpha.c \| e\rangle & \rightarrow & c[e/\alpha] \\
\langle V \| \tilde{\mu}a.c\rangle & \rightarrow & c[V/a] \\
\langle \lambda a.p \| q \cdot e\rangle & \rightarrow & \langle q \| \tilde{\mu}a.\langle p \| e\rangle\rangle
\end{array}$$

4.5.2 Small-step abstract machine

We can split again the previous operational semantics into small-step reduction rules. The underlying syntactical subcategories for proofs, contexts and command are almost the same as in the call-by-name setting, except that variables are now substituted by (and thus at the level of) values, while co-variables are no longer co-values. Besides, the absolute priority is given to proofs at level p, so that the hierarchy is reordered in p, e, V, E. The corresponding syntax is given by:

$$\begin{array}{llll@{\qquad}llll}
\textbf{Terms} & p & ::= & \mu\alpha.c \mid V & \textbf{Contexts} & e & ::= & \tilde{\mu}a.c \mid E \mid \alpha \\
\textbf{Values} & V & ::= & a \mid \lambda a.p & \textbf{Co-values} & E & ::= & p \cdot e
\end{array}$$

and the small-step reduction system is given by:

$$\begin{array}{ccc}
\langle \mu\alpha.c \| e\rangle_p & \rightsquigarrow & c_p[e/\alpha] \\
\langle V \| e\rangle_p & \rightsquigarrow & \langle V \| e\rangle_e \\
\langle V \| \tilde{\mu}a.c\rangle_e & \rightsquigarrow & c_p[V/a] \\
\langle V \| E\rangle_e & \rightsquigarrow & \langle V \| E\rangle_V \\
\langle \lambda a.p \| E\rangle_V & \rightsquigarrow & \langle \lambda a.p \| E\rangle_E \\
\langle \lambda a.p \| q \cdot e\rangle_E & \rightsquigarrow & \langle q \| \tilde{\mu}a.\langle p \| e\rangle\rangle_p
\end{array}$$

This defines an abstract-machine in context-free form, and the last two rules can again be compacted in one. We could also give a type system subdivided according to the syntactic hierarchy, which is exactly as expected. At this stage, we hope that any reader would be bored if we were to introduce it formally, therefore we shall omit it.

4.5.3 Continuation-passing style translation

4.5.3.1 Translation of terms

Having the abstract-machine in context-free form at our disposal, we can give the continuation-passing style corresponding to this operational semantics. The direct translation of small-step rules gives:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
[\![\langle p \| e\rangle]\!]_c & \triangleq & [\![p]\!]_p\,[\![e]\!]_e & [\![q\cdot e]\!]_e\,V & \triangleq & V\,[\![q]\!]_p\,[\![e]\!]_e \\
[\![\mu\alpha.c]\!]_p\,e & \triangleq & (\lambda\alpha.[\![c]\!]_c)\,e & [\![\alpha]\!]_e & \triangleq & \alpha \\
[\![V]\!]_p\,e & \triangleq & e\,[\![V]\!]_V & [\![a]\!]_V & \triangleq & a \\
[\![\tilde{\mu}a.c]\!]_e\,V & \triangleq & (\lambda a.[\![c]\!]_c)\,V & [\![\lambda a.p]\!]_V\,q\,e & \triangleq & q\,(\lambda a.[\![p]\!]_p\,e)
\end{array}$$

where administrative reductions particular to the translation are compressed. The expanded version is then:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
[\![\langle p \| e\rangle]\!]_c & \triangleq & [\![p]\!]_p\,[\![e]\!]_e & [\![q\cdot e]\!]_E & \triangleq & \lambda V.V\,[\![q]\!]_p\,[\![e]\!]_e \\
[\![\mu\alpha.c]\!]_p & \triangleq & \lambda\alpha.[\![c]\!]_c & [\![\alpha]\!]_E & \triangleq & \alpha \\
[\![V]\!]_p & \triangleq & \lambda e.e\,[\![V]\!]_V & [\![a]\!]_V & \triangleq & a \\
[\![\tilde{\mu}a.c]\!]_e & \triangleq & \lambda a.[\![c]\!]_c & [\![\lambda a.p]\!]_V & \triangleq & \lambda qe.q\,(\lambda a.[\![p]\!]_p\,e)
\end{array}$$

This induces a translation of commands at each level of the translation:

$$[\![\langle p \| e\rangle]\!]_c^p \triangleq [\![p]\!]_p\,[\![e]\!]_e \qquad\qquad [\![\langle V \| p\rangle]\!]_c^e \triangleq [\![e]\!]_e\,[\![V]\!]_V \qquad\qquad [\![\langle V \| q\cdot e\rangle]\!]_c^V \triangleq [\![V]\!]_V\,[\![q]\!]_p\,[\![e]\!]_e$$

which is again easy to prove correct with respect to computation, since the translation is defined from the reduction rules. This requires again a lemma on the soundness of substitution through the CPS.

Lemma 4.17. *For any variable a (co-variable α) and any value V (context e), the following holds for any command c:*

$$[\![c[V/a]]\!]_c \;=\; [\![c]\!]_c[[\![V]\!]_V/a] \qquad\qquad [\![c[e/\alpha]]\!]_c \;=\; [\![c]\!]_c[[\![e]\!]_e/\alpha]$$

The same holds for substitution within proofs and contexts.

Proof. By induction on the syntax of commands, proofs and contexts, the key cases corresponding to (co-)variables:

$$[\![a]\!]_p[[\![V]\!]_V/a] \;=\; (\lambda e.e\,a)[[\![V]\!]_V/a] \;=\; \lambda e.e\,[\![V]\!]_V \;=\; [\![V]\!]_p \;=\; [\![a[V/a]]\!]_p$$

□

Proposition 4.18. *For all levels ι,o of e,p,E, and any commands c,c', if $c_\iota \overset{1}{\rightsquigarrow} c'_o$, then $[\![c]\!]_c^\iota \xrightarrow{+}_\beta [\![c']\!]_c^o$.*

Proof. The proof is again an easy induction on the reduction $\rightsquigarrow$. Administrative reductions are trivial, the cases for μ and $\tilde{\mu}$ correspond to the previous lemma, which leaves us again with the more interesting cases of λ:

$$[\![\langle \lambda a.p \| q\cdot e\rangle]\!]_c^V = (\lambda qe.q\,(\lambda a.[\![p]\!]_p\,e))\,[\![q]\!]_p\,[\![e]\!]_e \xrightarrow{2}_\beta [\![q]\!]_p\,(\lambda a.[\![p]\!]_p\,[\![e]\!]_e) = [\![\langle q \| \tilde{\mu}a.\langle p \| e\rangle\rangle]\!]_c^p$$

□

4.5.3.2 Translation of types

The computational translation induces the following translation on types:

$$\begin{array}{lll}
\llbracket A\rrbracket_p & \triangleq \llbracket A\rrbracket_e \to \bot & \\
\llbracket A\rrbracket_e & \triangleq \llbracket A\rrbracket_V \to \bot & \\
\llbracket A\to B\rrbracket_V & \triangleq \llbracket A\rrbracket_p \to \llbracket B\rrbracket_e \to \bot & \\
\llbracket X\rrbracket_V & \triangleq X & \qquad (X \text{ variable})
\end{array}$$

where we take $\bot$ as return type for continuations. This translation extends naturally to contexts, where the translation of Γ is defined at level V while Δ is translated at level e:

$$\llbracket\Gamma, a : A\rrbracket_V \triangleq \llbracket\Gamma\rrbracket_V, a : \llbracket A\rrbracket_V \qquad\qquad \llbracket\Delta, \alpha : A\rrbracket_e \triangleq \llbracket\Delta\rrbracket_e, \alpha : \llbracket A\rrbracket_e$$

The translation of proofs, contexts and commands is well-typed:

Proposition 4.19. *For any contexts Γ and Δ, we have*

1. *if* $\Gamma \vdash p : A \mid \Delta$ *then* $\llbracket\Gamma\rrbracket_V, \llbracket\Delta\rrbracket_e \vdash \llbracket p\rrbracket_p : \llbracket A\rrbracket_p$
2. *if* $\Gamma \mid e : A \vdash \Delta$ *then* $\llbracket\Gamma\rrbracket_V, \llbracket\Delta\rrbracket_e \vdash \llbracket e\rrbracket_e : \llbracket A\rrbracket_e$
3. *if* $c : \Gamma \vdash \Delta$ *then* $\llbracket\Gamma\rrbracket_V, \llbracket\Delta\rrbracket_e \vdash \llbracket c\rrbracket_c : \bot$

Proof. The proof is done by induction over the typing derivation. The proof is essentially the same than in the call-by-name case, the main difference being in the case of $(\to_r)$, which is the only one we give here. If $\lambda a.p$ has type $A \to B$:

$$\frac{\Gamma, a : A \vdash_p p : B \mid \Delta}{\Gamma \vdash_V \lambda a.p : A \to B \mid \Delta}\,(\to_r)$$

then by induction hypothesis, we get that $\llbracket\Gamma\rrbracket_V, \llbracket\Delta\rrbracket_e, a : \llbracket A\rrbracket_V \vdash \llbracket p\rrbracket_p : \llbracket B\rrbracket_p$. By definition, we have $\llbracket\lambda a.p\rrbracket_V = \lambda qe.q\,(\lambda a.\llbracket p\rrbracket_p\, e)$, which we can type:

$$\dfrac{\dfrac{}{q : \llbracket A\rrbracket_p \vdash q : \llbracket A\rrbracket_e \to \bot}\,(\text{Ax}) \qquad \dfrac{\dfrac{\llbracket\Gamma\rrbracket_p, \llbracket\Delta\rrbracket_E, a : \llbracket A\rrbracket_p \vdash \llbracket p\rrbracket_p : \llbracket B\rrbracket_e \to \bot \qquad \dfrac{}{e : \llbracket B\rrbracket_e \vdash e : \llbracket B\rrbracket_e}\,(\text{Ax})}{\llbracket\Gamma\rrbracket_V, \llbracket\Delta\rrbracket_e, e : \llbracket B\rrbracket_e, a : \llbracket A\rrbracket_V \vdash \llbracket p\rrbracket_p\, e : \bot}\,(\to_E)}{\llbracket\Gamma\rrbracket_V, \llbracket\Delta\rrbracket_e, e : \llbracket B\rrbracket_e \vdash \lambda a.\llbracket p\rrbracket_p\, e : \llbracket A\rrbracket_e}\,(\to_I)}{\dfrac{\llbracket\Gamma\rrbracket_V, \llbracket\Delta\rrbracket_e, q : \llbracket A\rrbracket_p, e : \llbracket B\rrbracket_e \vdash q\,(\lambda a.\llbracket p\rrbracket_p\, e) : \bot}{\llbracket\Gamma\rrbracket_V, \llbracket\Delta\rrbracket_e \vdash \lambda qe.q\,(\lambda a.\llbracket p\rrbracket_p\, e) : \llbracket A\rrbracket_p \to \llbracket B\rrbracket_e \to \bot}\,(\to_I)}\,(\to_E)$$

□

The continuation-passing style translation preserves both reduction and typing, thus it is sufficient to deduce the normalization and the soundness (observe that we have again $\llbracket\bot\rrbracket_p = (\bot \to \bot) \to \bot$) for the call-by-value $\lambda\mu\tilde{\mu}$-calculus. The proofs are exactly the same as in the call-by-name case.

Theorem 4.20 (Normalization). *Typed commands of the simply-typed call-by-value $\lambda\mu\tilde{\mu}$-calculus are normalizing.*

Theorem 4.21 (Soundness). *There is no proof p (in the simply-typed call-by-value $\lambda\mu\tilde{\mu}$-calculus) such that* $\vdash p : \bot \mid$.

4.5.4 Realizability interpretation

The realizability interpretation follows the same guidelines than in the call-by-name setting. The major change comes with the syntactic hierarchy: given a formula A, its interpretation $|A|_p$ (the truth value $|A|$) will be defined by orthogonality to $\|A\|_e$ (falsity value $\|A\|$), which will be itself defined by orthogonality to $|A|_V$. The latter is sometimes called *truth value of values* of the formula A, and is reminiscent of call-by-value interpretations in Krivine realizability (see for instance [126, 108]). The main consequence of these bi-orthogonal definitions of truth values is that it requires a value restriction for universal quantifications:

$$\frac{\Gamma \mid e : A[n/x] \vdash \Delta}{\Gamma \mid e : \forall x.A \vdash \Delta}\ (\forall_l^1) \qquad\qquad \frac{\Gamma \vdash V : A \mid \Delta \quad x \notin FV(\Gamma,\Delta)}{\Gamma \vdash V : \forall x.A \mid \Delta}\ (\forall_r^1)$$

$$\frac{\Gamma \mid e : A[B/X] \vdash \Delta}{\Gamma \mid e : \forall X.A \vdash \Delta}\ (\forall_l^2) \qquad\qquad \frac{\Gamma \vdash V : A \mid \Delta \quad X \notin FV(\Gamma,\Delta)}{\Gamma \vdash V : \forall X.A \mid \Delta}\ (\forall_r^2)$$

As we will study value restriction more in depth in Chapter 7 (with different motivations), we do not want to give too much details at this stage. We only mention that this restriction is necessary to obtain the adequacy of typing rules, and can be understood as a consequence of the strict inclusion between the orthogonal of an intersection and the union of orthogonal sets: $\bigcup_{A\in\mathcal{A}} A^{\perp} \subsetneq (\bigcap_{A\in\mathcal{A}} A)^{\perp}$. For further explanations on the topic, we refer the reader to the appendices of [126].

Apart from this, the interpretation is straightforward. Poles are defined as usual as sets of commands closed under anti-reduction, and predicates are now interpreted as function $F : \mathbb{N}^k \to \mathcal{P}(\mathcal{V}^0)$ where $\mathcal{V}^0$ is the set of closed values. The interpretation of formulas with parameters is then defined by induction on the structure of formulas:

$$\begin{array}{rcl}
|\dot{F}(e_1,\dots,e_k)|_V & \triangleq & F([\![e_1]\!],\dots,[\![e_k]\!]) \\
|A \to B|_V & \triangleq & \{\lambda a.p : \forall u \in |A|_V, p[u/a] \in |B|_p\} \\
|\forall x.A|_V & \triangleq & \bigcap_{n\in\mathbb{N}} |A[n/x]|_V \\
|\forall X.A|_V & \triangleq & \bigcap_{F:\mathbb{N}^k\to\mathcal{P}(\mathcal{V}^0)} |A[\dot{F}/X]|_V \\
\|A\|_e & \triangleq & |A|_V^{\perp\!\!\!\perp} \ = \ \{e \mid \forall V \in |A|_V, \langle V\|e\rangle \in \perp\!\!\!\perp\} \\
|A|_p & \triangleq & \|A\|_e^{\perp\!\!\!\perp} \ = \ \{t \mid \forall e \in \|A\|_e, \langle p\|e\rangle \in \perp\!\!\!\perp\}
\end{array}$$

The intuition underlying this definition is the very same: a proof in the truth value (of values) $|\forall x.A|_V$ of a universally quantified formula has to be in the corresponding truth value $|A[n/x]|_V$ for every possible instantiation $n \in \mathbb{N}$ of the variable x. As for values in $|A \to B|_V$, they are functions of the form $\lambda a.p$ where, according to the operational semantics, the abstracted a variable is intended to be substituted by a value (*i.e.* a realizer in $|A|_V$), giving raise to a proof at level p (*i.e.* a realizer in $|B|_p$).

This interpretation satisfies the following monotonicity relations:

Proposition 4.22 (Monotonicity). *For any formula A, the following hold:*

1. $|A|_V \subseteq |A|_p$
2. $\|A\|_e^{\perp\!\!\!\perp\perp\!\!\!\perp} = \|A\|_e$
3. $\|\forall x.A\|_e \supseteq \bigcup_{n\in\mathbb{N}} \|A[n/x]\|_e$
4. $\|\forall X.A\|_e \supseteq \bigcup_{F:\mathbb{N}^k\to\mathcal{P}(\Pi)} \|A[\dot{F}/X]\|_e$
5. $|\forall x.A|_p \subseteq \bigcap_{n\in\mathbb{N}} |A[n/x]|_p$
6. $|\forall X.A|_p \subseteq \bigcap_{F:\mathbb{N}^k\to\mathcal{P}(\Pi)} |A[\dot{F}/X]|_p$

Proof. Usual properties of orthogonality with respect to unions and intersections. □

A valuation is defined again as a function ρ which associates a natural number $\rho(x) \in \mathbb{N}$ to every first-order variable x and a function $\rho(X) : \mathbb{N}^k \to \mathcal{P}(\mathcal{V}^0)$ to every second-order variable X of arity k.

As for substitutions, written σ, they now map variables to closed values (written $\sigma, a := V$) and co-variables to contexts (written $\sigma, \alpha := e$).

Given a closed (one-sided) context Γ, we say that a substitution σ realizes Γ, which we write $\sigma \Vdash \Gamma$, if for any $(a : A) \in \Gamma, \sigma(a) \in |A|_V$ and if for any $(\alpha : A^{\perp\!\!\perp}) \in \Gamma, \sigma(\alpha) \in \|A\|_e$. We are now equipped to prove the adequacy of the typing rules for the (call-by-value) $\lambda\mu\tilde{\mu}$-calculus with respect to the realizability interpretation we defined.

Proposition 4.23 (Adequacy)**.** *Let* Γ, Δ *be typing context, and* $\rho \Vdash \Gamma$ *and* $\rho \Vdash \Delta$*, then*

1. *if* $\Gamma \vdash p : A \mid \Delta,$ *then* $p[\sigma] \in |A[\rho]|_p$
2. *if* $\Gamma \mid e : A \vdash \Delta,$ *then* $e[\sigma] \in \|A[\rho]\|_e$
3. *if* $c : \Gamma \vdash \Delta,$ *then* $c[\sigma] \in \perp\!\!\perp$

Proof. The proof is again a mutual induction over the typing derivation. Cases (Cut),(Ax$_r$),(Ax$_l$),(μ),($\tilde{\mu}$),($\forall_l^1$) and ($\forall_l^2$) are essentially the same as in the call-by-name setting. Cases ($\forall_r^1$),($\forall_r^2$) are the same, except that they require to refine the induction hypotheses to also prove that if $\Gamma \vdash V : A \mid \Delta,$ then $V[\sigma] \in |A[\rho]|_V$. We only prove the two cases left, which are the cases for the implication.

- **Case** ($\rightarrow_r$). We are in the following situation:

$$\frac{\Gamma, a : A \vdash p : B \mid \Delta}{\Gamma \vdash \lambda a.p : A \rightarrow B \mid \Delta}\ (\rightarrow_r)$$

By induction hypothesis, if $V \in |A[\rho]|_V$, then $(\sigma, a := V) \Vdash (\Gamma, a : A \cup \Delta)$ and thus $p[\sigma, a := V] \in |B[\rho]|_p$. By definition of truth values of values, $\lambda a.p[\sigma] = (\lambda a.p)[\sigma]$ is thus in $|(A \rightarrow B)[\rho]|_V$.

- **Case** ($\rightarrow_l$). We are in the following situation:

$$\frac{\Gamma \vdash q : A \mid \Delta \quad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid q \cdot e : A \rightarrow B \vdash \Delta}\ (\rightarrow_l)$$

Let $\lambda a.p \in |(A \rightarrow B)[\rho]|_V$, that is $p[V/a] \in |B[\rho]|_p$ for any $V \in |A[\rho]|_V$. By induction, we have that $q[\sigma] \in |A[\rho]|_p$. Besides,

$$\langle \lambda a.p \| q[\sigma] \cdot e[\sigma] \rangle \rightarrow \langle q[\sigma] \| \tilde{\mu} a. \langle p \| e[\sigma] \rangle \rangle$$

thus by anti-reduction, it suffices to show that $\tilde{\mu} a. \langle p \| e \rangle \in \|A[\rho]\|_e$. Once more, considering $V \in |A[\rho]|_V$, since

$$\langle V \| \tilde{\mu} a. \langle p \| e[\sigma] \rangle \rangle \rightarrow \langle p[V/a] \| e[\sigma] \rangle$$

we can conclude by anti-reduction: using the hypothesis for $p[V/a]$ and the induction hypothesis to get $e[\sigma] \in \|B[\rho]\|_e$, we deduce that the latter command is in the pole. □

Normalization and soundness are again direct consequences of adequacy, the proofs being similar we do not recall them.

Theorem 4.24 (Normalization)**.** *Typed commands of the second-order call-by-value* $\lambda\mu\tilde{\mu}$*-calculus are normalizing.*

Theorem 4.25 (Soundness)**.** *There is no proof p (in the second-order call-by-value* $\lambda\mu\tilde{\mu}$*-calculus) such that* $\vdash p : \bot \mid$ *.*

4.6 From adequacy to operational semantics

We should say a word about the dogmatism of our presentation. As we were interested in proving properties of a language with its operational semantics, we started from the reduction system, then defined the adequate realizability interpretation. However, as highlighted by Dagand and Scherer [35], it is possible to work the other way round. While studying the computational content of the adequacy lemma[11] (in the case of simply-typed lambda-calculus), they showed in passing that one could first define the desired interpretation (*i.e.* truth and falsity values at each levels), then deduce the reduction rules from the proof of adequacy. Their paper was supported by a Coq development which we adapted to match the framework of the $\lambda\mu\tilde{\mu}$-calculus[12]. To better illustrate this observation, our development also includes a positive product type $A\times B$ (inhabited by pairs and contexts of the shape $\tilde{\mu}(a,b).c$ to destruct pairs). We give several cases depending on whether product type and arrow type are interpreted in a call-by-value or call-by-name fashion.

To come full circle, we would like to attract the reader's attention to the fact that when the adequacy lemma is defined as a program, it almost gives the definition of the corresponding CPS translation. This is particularly reflected on the call-by-value cases for pairs and stacks. In the latter, using informal notations, the function **rea** which proves the adequacy is defined by:

$$\mathbf{rea}\ \ (u\cdot e : \Gamma \mid A \to B \vdash \Delta)\ (\rho \Vdash \Gamma)\ (\sigma \Vdash \Delta)\ :=\ \lambda f.(\mathbf{rea}\ u\,\rho\,\sigma)\ (\lambda V.f\ V\ (\mathbf{rea}\ e\,\rho\,\sigma))$$

which is to compare with the following (call-by-value) CPS translation:

$$[\![q\cdot e]\!]_e\ \triangleq\ \lambda f.[\![q]\!]_p(\lambda V.f\ V\ [\![e]\!]_e)$$

This corresponds intuitively to the following reduction rules:

$$\begin{array}{ccc}
\langle p \| q\cdot e\rangle & \to & \langle q \| \tilde{\mu}a.\langle p \| a\cdot e\rangle\rangle \\
\langle V \| \tilde{\mu}a.c\rangle & \to & c[V/a] \\
\langle \lambda a.p \| V\cdot e\rangle & \to & \langle p[V/a] \| e\rangle
\end{array}$$

All in all, if the reader was to remember only one idea of this chapter, we would like this idea to be the claim that given a calculus, the given of a fine-grain operational semantics naturally induces a continuation-passing style translation and a realizability interpretation *à la* Krivine (and even vice-versa). This should not come as a surprise as all these artifacts relies on a common notion of computation, which they share. As we saw with the call-by-name and call-by-value $\lambda\mu\tilde{\mu}$-calculi, these artifacts can be derived methodically and provides us with powerful proof tools.

[11] The main claim of their paper is that proofs of normalization by realizability and by evaluation are almost the same, in that the proof of the adequacy lemma, as a program (that is, roughly, a function taking a typing derivation for a term and constructing the proof that this term is a realizer of the corresponding type), is a normalization machine: it takes a term and evaluates it again a well-chosen stack to use induction hypotheses. If we observe carefully the proofs of adequacy for the λ_c-calculus or the ones of the $\lambda\mu\tilde{\mu}$-calculi we presented, this is indeed their computational contents: almost all cases are proved by reducing a process, then using induction hypotheses and the closure of the pole under anti-reduction.

[12] The source can be browsed here or downloaded here.

Part II

A constructive proof of dependent choice compatible with classical logic

5- The starting point: dPA$^{\omega}$

Axiom of choice

The axiomatization of a theory, as we explained in Chapter 1, is to be understood as an intent to give a formal and truthful representation of a given world or structure. As long as this structure deals with finite objects that have a concrete representation in the physical world, it is easy to agree on what it "is" or "should be" (and thus on whether the axiomatization is truthful). However, as soon as the theory involves infinite objects, this question quickly turns out to be more the matter of one's personal "religion" than the empirical observation of a physical object. In particular, some undeniable properties of finite objects become much more questionable in the case of infinite sets. Consider for instance the following problem, as presented[1] by Russell [146, pp.125-127]:

> *[Imagine a] millionaire who bought a pair of socks whenever he bought a pair of shoes, and never at any other time, and who had such a passion for buying both that at last he had $\aleph_0$ pairs of shoes and $\aleph_0$ pairs of socks. The problem is: How many shoes had he, and how many socks?*

The cardinal $\aleph_0$ defines exactly the infinite quantity of natural numbers: an infinite set is of cardinality $\aleph_0$ if it can be enumerated by the natural numbers. In particular, since there is a bijection from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$, $\aleph_0$ is not increased by doubling.

> *One would naturally suppose that he had twice as many shoes and twice as many socks as he had pairs of each, and that therefore he had $\aleph_0$ of each [...].*

To prove this claim, it is thus necessary and sufficient to give an enumeration of the millionaire's shoes and socks. Yet, this is not possible *a priori*:

> *In our case it can be done with the shoes, but not with the socks, except by some very artificial device. The reason for the difference is this: Among shoes we can distinguish right and left, and therefore we can make a selection of one out of each pair, [...] but with socks no such principle of selection suggests itself [...].*
>
> *We may put the matter in another way. To prove that a class has $\aleph_0$ terms, it is necessary and sufficient to find some way of arranging its terms in a progression. There is no difficulty in doing this with the shoes. The pairs are given as forming an $\aleph_0$, and therefore as the field of a progression. Within each pair, take the left shoe first and the right second, keeping the order of the pair unchanged; in this way we obtain a progression of all the shoes. But with the socks we shall have to choose arbitrarily, with each pair, which to put first; and an infinite number of arbitrary choices is an impossibility. Unless we can find a rule for selecting, i.e. a relation which is a selector, we do not know that a selection is even theoretically possible. [...]*
> *The case of the socks, with a little goodwill on the part of the reader, may serve to show how a selection might be impossible.*

[1] Russell actually presented the story with boots. We replaced it with shoes in the quote, which we found to be more asymmetric. Russell might never had one of these ugly (and symmetric) plastic rain boots.

More generally, it is unclear if one should be able to pick an element of an infinite set, and from a theoretical point of view, this indeed requires an extra axiom, called the axiom of choice. This axiom, which was first introduced by Zermelo in the realm of set theory [163], is functionally expressed by:

$$AC \triangleq (\forall x \in A.\exists y \in B.P(x,y)) \to (\exists f \in B^A.\forall x \in A.P(x,f(x)))$$

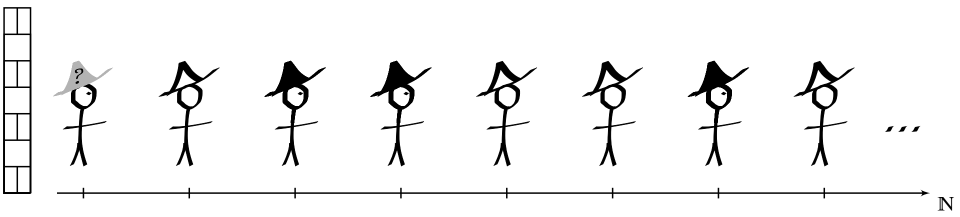
which stipulates the existence of a choice function[2]. This axiom was shown to be independent of Zermelo-Fraenkel set theory (ZF)[3]. Even if it is very tempting to consider natural the possibility of selecting one element within an infinite set (since it is for finite sets), such an axiom leads to very surprising consequences. The most striking example one is certainly the Banach-Tarski paradox [9], which shows that the unit ball

$$\mathcal{B} := \{(x,y,z) \in \mathcal{R}^3 : x^2 + y^2 + z^2 = 1\}$$

in three dimensions can be disassembled into a finite number of pieces, which can then be reassembled (after translating and rotating each of the pieces) to form two disjoint copies of the ball $\mathcal{B}$.

Another dazzling paradox is a variant of the famous riddle where a column of prisoners is facing a wall, each of them having a black or white hat on his head of which he ignores the color. Each prisoner (from the end of the line) has to guess in turns his hat color. They are eventually released if at most one prisoner is wrong. They are allowed to talk through a strategy in the beginning, and they indeed have a way to end up free in this situation. Now, let us turn the prisoners around and consider the following infinite version:

> *A countable infinite number of prisoners are placed on the natural numbers, facing in the positive direction (*i.e. *everyone can see an infinite number of prisoners). Hats will be placed and each prisoner will be asked what his hat color is.*



> *However, to complicate things, prisoners cannot hear previous guesses or whether they were correct. In this situation, what is the best strategy?*

Admitting the axiom choice[4], the answer is quite counter-intuitive: the prisoners have a (common) strategy to guess the color of their own hat, in such a way that only a finite number of them will make wrong guesses. Even more shocking, the strategy is so robust that we could consider any number of colors (even an uncountable one), the prisoners will still only make a finite number of wrong guesses... The solution is left to the sagacity of the reader[5] but the “problem” here is very similar to the Banach-Tarski paradox, where the pieces used in this decomposition are highly pathological in nature and cannot be constructed without the axiom of choice.

In short, the question of knowing whether the axiom of choice is wrong or not can not be given any mathematical answer. Indeed, the axiom of choice is independent from the axioms of set theory.

[2] If we define the predicate $P(x,y)$ as $y \in x$, it exactly says that if all the sets $x \in A$ are non-empty, there exists a choice function: $(\forall x \in A.x \neq \emptyset) \to \exists f \in \cup A^A.\forall x \in A.f(x) \in x$.

[3] Gödel proved that the theory ZF + AC is consistent, and Cohen proved the same for the theory ZF + ¬AC. Details on these proofs and much more about the axiom of choice can be found for instance in Jech's book on the topic [83].

[4] We also assume that each prisoner can see the ω prisoners in front of him, have infinite memory and so forth.

[5] Clue: the definition of clever equivalence classes and the use of AC to pick representatives can be helpful. The full answer is available here [125].

Intuitively, the axiom of choice does not reflect anything concrete in our living world. Adding it or not to a theory is thus a matter of one's belief, with its logical strength as benefits and its paradoxical consequences as withdraws.

Dependent and countable choices

In fact, a huge part of mathematics does not require the axiom of choice in full strength. For instance, most of analysis[6] can be done in a system of axioms containing a weaker form of choice, namely the axiom of dependent choice. This axiom expresses the possibility of constructing a sequence where each element has to be chosen in function of the anterior. Formally, it is defined by:

$$DC \triangleq (\forall x \in A.\exists y \in A.P(x,y)) \rightarrow \forall x_0 \in A.\exists f \in A^{\mathbb{N}}.(f(0) = x_0 \wedge \forall n \in \mathbb{N}.P(f(n), f(S(n))))$$

This axiom does not lead to the paradoxical consequences of the full axiom of choice, and is in practice expressive enough for most of the mathematics[7].

Another weaker form of choice, which is actually the one involved in Russell shoes-and-socks metaphor, is the axiom of countable choice. It is simply defined as the axiom of choice where universal variables are bound to the set of natural numbers $\mathbb{N}$:

$$AC_{\mathbb{N}} \triangleq (\forall x \in \mathbb{N}.\exists y \in B.P(x,y)) \rightarrow \exists f \in B^{\mathbb{N}}.\forall x \in \mathbb{N}.P(x, f(x))$$

It is quite easy to check that the full axiom of choice (AC) implies the axiom of dependent choice (DC), which itself implies the axiom of countable choice ($AC_{\mathbb{N}}$) (converse implications are false). Dependent and countable choices are the axiom that will be at the heart of this part of the monograph.

5.1 Computational content of the axiom of choice

5.1.1 Martin-Löf Type Theory

In the line of Curry-Howard isomorphism, it is natural to wonder what is the computational content of the axiom choice, that is, what would be a program whose type is (AC). In fact, through the Brouwer-Heyting-Kolmogoroff interpretation of intuitionistic logic (see Section 3.1.1), a proof of $\forall x.\exists y.P(x,y)$ is precisely a function which associates to any m a proof of $\exists y.P(m,y)$, which is itself a pair made of a certificate n and a proof of $P(m,n)$. Thus, there exists *de facto* a function f such that for any m, $P(m, f(m))$ holds. Otherwise said, through this interpretation, the axiom of choice should then be a trivial theorem.

This idea is the key of Martin-Löf's proof for the axiom choice in his constructive type theory [115]. One of the crucial differences with the different theories we presented until here, is that types (*i.e.* formulas) are now dependent on terms (*i.e.* on proofs). Just like first-order arithmetic includes a quantification $\forall x.A$ ranging over natural numbers and leading to formulas $A[n/x]$ for each possible instantiation $n \in \mathbb{N}$ of x, Martin-Löf type theory includes a dependent product type written $\Pi(x : A).B$ where the variable x ranges over the terms of type A. In particular, if t is a term of type $\Pi(x : A).B$ and u is a term of type A, the term tu is then of type $B[u/x]$:

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : \Pi(x : A).B}\ (\Pi_I) \qquad\qquad \frac{\Gamma \vdash t : \Pi(x : A).B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B[u/x]}\ (\Pi_E)$$

[6] Notably, Baire category theorem has been proved equivalent to the axiom of dependent choice. More generally, a large class of theorems whose proof are done by constructing a sequence by induction requires this axiom.

[7] More details on this (and more generally on the axiomatic strength required by theorems of mathematics) can be found in the introduction of Simpson's book [149].

It is worth noting that in the case where B does not refer to x, these rules exactly correspond to the usual rules $(\to_I)$ and $(\to_E)$.

The fact that formulas can now refer to terms allows us to strengthen the rules for existential quantification. They now reflect the BHK interpretation for existential proofs, which inhabits a dependent sum type written $\Sigma(x:A).B$: a proof term of type $\Sigma(x:A).B$ is a pair (t,u) such that t—the *witness*—is of type A, while u—the *proof*—is of type $B[t/x]$. Dually to this construction, there are now two elimination rules[8]: one with a destructor `wit` to extract the witness, the second one with a destructor `prf` to extract the proof:

$$\frac{\Gamma\vdash t:A \quad \Gamma\vdash u:B[t/x]}{\Gamma\vdash (t,u):\Sigma(x:A).B}\,(\Sigma_I) \qquad \frac{\Gamma\vdash t:\Sigma(x:A).B}{\Gamma\vdash \mathtt{wit}\ t:A}\,(\mathtt{wit}) \qquad \frac{\Gamma\vdash t:\Sigma(x:A).B}{\Gamma\vdash \mathtt{prf}\ t:B[\mathtt{wit}\ t/x]}\,(\mathtt{prf})$$

Note that this extension of types with dependencies corresponds to the horizontal axis of the λ-cube (Section 2.4.2). In the sequel, we will present in more details a full dependent system with its type system and reduction rules. As for now, let us just mention that these terms reduce as follows:

$$(\lambda x.t)\,u \to t[u/x] \qquad \mathtt{wit}\,(t,u) \to t \qquad \mathtt{prf}\,(t,u) \to u$$

These reductions naturally induce a relation on types: we write $A \triangleright B$ if reducing some term occurring in A yields B. The reflexive-symmetric-transitive closure of this relation is written $A\equiv B$ and the type system includes a conversion rules according to this relation:

$$\frac{\Gamma\vdash t:A \quad A\equiv B}{\Gamma\vdash t:B}\,(\text{CONV})$$

Having said this, we dispose of enough structure to give a proof term for the axiom of choice, which is nothing more than an implementation of the intuition above: given a proof H of $\Pi(x:A).\Sigma(y:B).P(x,y)$, the choice function simply maps any x to the witness of Hx, while the proof that this function is sound w.r.t. P returns the corresponding witness. This term can indeed be given the type of the axiom of choice:

$$\vdash \lambda H.(\lambda x.\mathtt{wit}\,(Hx),\lambda x.\mathtt{prf}\,(Hx)) : AC$$

where AC is defined in terms of dependent product and sum:

$$AC \triangleq \Pi(x:A).\Sigma(y:B).P(x,y) \to \Sigma(f:\Pi(x:A).B).\Pi(x:A).P(x,f(x))$$

5.1.2 Incompatibility with classical logic

Unsurprisingly, this proof does not scale to classical logic (otherwise the axiom of choice would be a theorem of Zermelo-Fraenkel set theory, which is a classical theory). We give two explanations for this, first a metaphysical argument for this natural limitation in terms of computability, second a technical description of the incompatibility of classical logic and dependent types.

[8] Actually, the original presentation [115] only has one rule, called dependent elimination rule, given by:

$$\frac{\Gamma\vdash c:\Sigma(x:A).B \quad \Gamma,x:A,y:B[x]\vdash d:C[(x,y)]}{\Gamma\vdash E(c,\lambda xy.d):C[c]}\,(\Sigma_E)$$

As for the reduction rule, it was defined by:

$$E((t,u),\lambda xy.d[(x,y)]) \to d[(t,u)]$$

It is easy to check that defining the primitives `wit` c and `prf` c respectively by $tE(c,\lambda xy.x)$ and $E(c,\lambda xy.y)$ allow to recover the corresponding typing and reduction rules, and vice-versa.

5.1.2.1 Computing the uncomputable

Imagine that we could dispose, in a type theoretic (or BHK interpretation, realizability) fashion, of a classical framework including a proof term t for the axiom of choice:

$$\vdash t : \forall x \in A.\exists y \in B.P(x,y) \to \exists f \in B^A.\forall x \in A.P(x,f(x))$$

Consider now any undecidable[9] predicate $U(x)$ over a domain X. Since we are in a classical framework, using the middle-excluded, the formula $U(x) \lor \neg U(x)$ is true for any $x \in X$. This can be strengthened into the formula:

$$\forall x \in X.\exists y \in \{0,1\}.(U(x) \land y = 1) \lor (\neg U(x) \land y = 0)$$

which is provable as well and thus should have a proof term u. Now, this has the shape of the hypothesis of the axiom of choice, so that by application of t to u, we should obtain a term:

$$\vdash t\,u : \exists f \in \{0,1\}^X.\forall x \in X.(U(x) \land f(x) = 1) \lor (\neg U(x) \land f(x) = 0)$$

In particular, the term $\mathtt{wit}\ (t\,u)$ would be a function which, for any $x \in X$, outputs 1 if $U(x)$ is true, and 0 otherwise. This is absurd, since U is undecidable.

This handwavy explanation gives us a metamathematical argument on the impossibility of having a proof system which is classical as a logic, entails the axiom of choice and where proofs fully compute. Since the existence of consistent classical theories with the axiom of choice (like set theory) has been proven, the incompatibility is to be found with the constructive character of Martin-Löf type theory. Actually, the compatibility of AC with constructive theories is very sensitive to the definition of "constructive" and is already discussed[10]. In the next sections, we will present an intent to give a proof of the axiom of dependent choice that is constructive and yet compatible with classical logic.

5.1.2.2 Inconsistency

Technically, another reason why Martin-Löf type theory cannot scale to classical logic is that the simultaneous presence of control operators and dependent types leads to inconsistencies. This was observed by Herbelin [69] in a weaker setting, which we recap hereafter.

Let us adopt here a stratified presentation of dependent types, by syntactically distinguishing *terms*—that represent mathematical objects—from *proof terms*—that represent mathematical proofs. In other words, we syntactically separate the categories corresponding to witnesses and proofs in dependent sum types. Consider a minimal logic of Σ-types and equality, whose formulas, terms (only representing natural number) and proofs are defined as follows:

Formulas	A,B	$::=$	$t = u \mid \exists x^{\mathbb{N}}.A$
Terms	t,u	$::=$	$n \in \mathbb{N} \mid \mathtt{wit}\ p$
Proofs	p,q	$::=$	$\mathtt{refl} \mid \mathtt{subst}\ p\,q \mid (t,p) \mid \mathtt{prf}\ p$

Let us explain the different proof terms by presenting their typing rules. First of all, the pair (t,p) is a proof for an existential formula $\exists x^{\mathbb{N}}.A$ (or $\Sigma(x : \mathbb{N}).A$) where t is a witness for x and p is a certificate for $A[t/x]$. This implies that both formulas and proofs are dependent on terms, which is usual in mathematics. What is less usual in mathematics is that, as in Martin-Löf type theory, dependent types also allow for terms (and thus for formulas) to be dependent on proofs, by means of the constructors $\mathtt{wit}\ p$

[9] That is to say that $U(x)$ is a predicate such that there exists no program p which, given any input $x \in X$, computes whether $U(x)$ is true or not.

[10] There is plenty of literature on constructive choiceless mathematics. The reader can for instance read this very interesting argument of Andrej Bauer rejecting AC in a constructive (in the sense of computable) setting: `https://mathoverflow.net/a/23043`.

and $\texttt{prf}\ p$. The typing rules are the same as in the previous section for Σ-types, except that there are separated typing judgments for terms, which can only be of type $\mathbb{N}$.:

$$\frac{\Gamma \vdash p : A(t) \quad \Gamma \vdash t : \mathbb{N}}{\Gamma \vdash (t,p) : \exists x^{\mathbb{N}} A}\,(\exists_I) \qquad \frac{\Gamma \vdash (t,p) : \exists x^{\mathbb{N}}.A}{\Gamma \vdash \texttt{prf}\ p : A[\texttt{wit}\ p/x]}\,(\texttt{prf}) \qquad \frac{\Gamma \vdash t : \exists x^{\mathbb{N}}.A}{\Gamma \vdash \texttt{wit}\ t : \mathbb{N}}\,(\texttt{wit}) \qquad \frac{n \in \mathbb{N}}{\Gamma \vdash n : \mathbb{N}}$$

Then, $\texttt{refl}$ is a proof term for equality, and $\texttt{subst}\ p\,q$ allows to use a proof of an equality $t = u$ to convert a formula $A(t)$ into $A(u)$:

$$\frac{t \to u}{\Gamma \vdash \texttt{refl} : t = u}\,(\texttt{refl}) \qquad \frac{\Gamma \vdash p : t = u \quad \Gamma \vdash q : B[t]}{\Gamma \vdash \texttt{subst}\ p\,q : B[u]}\,(\texttt{subst})$$

The reduction rules for this language, which are safe with respect to typing, are then:

$$\texttt{wit}\,(t,p) \to t \qquad \texttt{prf}\,(t,p) \to p \qquad \texttt{subst refl}\ p \to p$$

Starting from this (sound) minimal language, Herbelin showed that its classical extension with the control operators $\texttt{call/cc}_k$ and $\texttt{throw}\ k$ permits to derive a proof of $0 = 1$ [69]. The $\texttt{call/cc}_k$ operator, which is a binder for the variable k, is intended to catch its surrounding evaluation context. On the contrary, $\texttt{throw}\ k$ (in which k is bound) discards the current context and restores the context captured by $\texttt{call/cc}_k$. The addition to the type system of the typing rules for these operators (that are similar to the different control operators presented in the prelude):

$$\frac{\Gamma, k : \neg A \vdash p : A}{\Gamma \vdash \texttt{call/cc}_k\, p : A} \qquad \frac{\Gamma, k : \neg A \vdash p : A}{\Gamma, k : \neg A \vdash \texttt{throw}\ k\, p : B}$$

allows the definition of the following proof:

$$p_0 \triangleq \texttt{call/cc}_k\,(0, \texttt{throw}\ k\,(1, \texttt{refl})) : \exists x^{\mathbb{N}}.x = 1$$

Intuitively such a proof catches the context, give 0 as witness (which is incorrect), and a certificate that will backtrack and give 1 as witness (which is correct) with a proof of the equality.

If besides, the following reduction rules[11] are added:

$$\begin{array}{rcll} \texttt{wit}\,(\texttt{call/cc}_k\, p) & \to & \texttt{call/cc}_k(\texttt{wit}\,(p[k(\texttt{wit}\,\{\ \})/k])) & \\ \texttt{call/cc}_k\, t & \to & t & (k \notin FV(t)) \end{array}$$

then we can formally derive obtain a proof of $1 = 0$. Indeed, the seek of a witness by the term $\texttt{wit}\ p_0$ will reduce to $\texttt{call/cc}_k\ 0$, which itself reduces to 0. The proof term $\texttt{refl}$ is thus a proof of $\texttt{wit}\ p_0 = 0$, and we obtain indeed a proof of $1 = 0$:

$$\frac{\dfrac{\vdash p_0 : \exists x^{\mathbb{N}}.x = 1}{\vdash \texttt{prf}\ p_0 : \texttt{wit}\ p_0 = 1}\,(\texttt{prf}) \qquad \dfrac{\texttt{wit}\ p_0 \to 0}{\vdash \texttt{refl} : \texttt{wit}\ p_0 = 0}\,(\texttt{refl})}{\vdash \texttt{subst}\,(\texttt{prf}\ p_0)\ \texttt{refl} : 1 = 0}\,(\texttt{subst})$$

The bottom line of this example is that the same proof p_0 is behaving differently in different contexts thanks to control operators, causing inconsistencies between the witness and its certificate. The easiest and usual approach to prevent this is to impose a restriction to values (which are already reduced) for proofs appearing inside dependent types and within the operators $\texttt{wit}$ and $\texttt{prf}$, together with a call-by-value discipline. In particular, in the present example this would prevent us from writing $\texttt{wit}\ p_0$ and $\texttt{prf}\ p_0$.

[11] Technically this requires to extend the language to authorize the construction of terms $\texttt{call/cc}_k\, t$ and of proofs $\texttt{throw}\ t$. The first rule expresses that $\texttt{call/cc}_k$ captures the context $\texttt{wit}\,\{\,\}$ and replaces every occurrence of $\texttt{throw}\ k\,t$ with $\texttt{throw}\ k\,(\texttt{wit}\ t)$. The second one just expresses the fact that $\texttt{call/cc}_k$ can be dropped when applied to a term t which does not contain the variable k.

5.2 A constructive proof of dependent choice compatible with classical logic

We shall now present dPA$^\omega$, a proof system that was introduced by Herbelin [70] as a mean to give a computational content to the axiom of choice in a classical setting. The calculus is a fine adaptation of Martin-Löf proof which circumvents the different difficulties caused by classical logic. Rather than restating dPA$^\omega$ in full details, for which we refer the reader to [70], let us describe informally the rationale guiding its definition and the properties that it verifies. We shall then present the missing bit of his calculus which led us to this work, namely the normalization, and our approach to prove it.

5.2.1 Realizing countable and dependent choices in presence of classical logic

As we saw in Section 5.1.1, the dependent sum type of Martin-Löf's type theory provides a strong existential elimination, which allows us to prove the full axiom of choice. The proof is simple and constructive:

$$\begin{array}{rcl} AC_A & := & \lambda H.(\lambda x.\mathtt{wit}\,(Hx),\lambda x.\mathtt{prf}\,(Hx)) \\ & : & \forall x^A.\exists y^B.P(x,y) \to \exists f^{A\to B}.\forall x^A.P(x,f(x)) \end{array}$$

To scale up this proof to classical logic, the first idea in Herbelin's work [70] is to restrict the dependent sum type to a fragment of his system which is called *negative-elimination-free* (NEF). This fragment contains slightly more proofs than just values, but is still computationally compatible with classical logic.

The second idea is to represent a countable universal quantification as an infinite conjunction. This allows us to internalize into a formal system the realizability approach of [15, 40] as a direct proofs-as-programs interpretation. Informally, let us imagine that given a proof $H : \forall x^A.\exists y^B.P(x,y)$, we could create the sequence $H_\infty = (H0, H1, \ldots, Hn, \ldots)$ and select its n^{th}-element with some function $\mathtt{nth}$. Then one might wish that

$$\lambda H.(\lambda n.\mathtt{wit}\,(\mathtt{nth}\;n\;H_\infty),\lambda n.\mathtt{prf}\,(\mathtt{nth}\;n\;H_\infty))$$

could stand for a proof for $AC_{\mathbb{N}}$. However, even if we were effectively able to build such a term, H_∞ might still contain some classical proof. Therefore two copies of $H\,n$ might end up being different according to the contexts in which they are executed, and then return two different witnesses. This problem could be fixed by using a shared version of H_∞, say

$$\lambda H.\,\mathtt{let}\,a = H_\infty\,\mathtt{in}\;(\lambda n.\mathtt{wit}\,(\mathtt{nth}\;n\;a),\lambda n.\mathtt{prf}\,(\mathtt{nth}\;n\;a))\,.$$

It only remains to formalize the intuition of H_∞. This is done by means of a coinductive fixpoint operator. We write $\mathtt{cofix}^t_{bx}[p]$ for the co fixpoint operator binding the variables b and x, where p is a proof and t a term. Intuitively, such an operator is intended to reduce according to the rule:

$$\mathtt{cofix}^t_{bx}[p] \;\to\; p[t/x][\lambda y.\mathtt{cofix}^t_{bx}[p]/b]$$

This is to be compared with the usual inductive fixpoint operator which we write $\mathtt{ind}^t_{bx}[p_0\,|\,p_S]$ (which binds the variables b and x) and which reduces as follows:

$$\mathtt{ind}^0_{bx}[p_0\,|\,p_S] \;\to\; p_0 \qquad\qquad \mathtt{ind}^{S(t)}_{bx}[p_0\,|\,p_S] \;\to\; p_S[t/x][\mathtt{ind}^t_{bx}[p_0\,|\,p_S]/b]$$

The presence of coinductive fixpoints allows us to consider the proof term $\mathtt{cofix}^0_{bn}[(Hn,b(S(n)))]$, which implements a stream eventually producing the (informal) infinite sequence H_∞. Indeed, this proof term reduces as follows:

$$\mathtt{cofix}^0_{bn}[(Hn,b(S(n)))] \to (H\,0,\mathtt{cofix}^1_{bn}[(Hn,b(S(n)))]) \to (H\,0,(H\,1,\mathtt{cofix}^2_{bn}[(Hn,b(S(n)))])) \to \ldots$$

This allows for the following definition of a proof term for the axiom of countable choice:

$$AC_{\mathbb{N}} := \lambda H.\,\mathtt{let}\,a = \mathtt{cofix}^0_{bn}[(Hn, b(S(n)))]\,\mathtt{in}\,(\lambda n.\,\mathtt{wit}\,(\mathtt{nth}\;n\;a), \lambda n.\,\mathtt{prf}\,(\mathtt{nth}\;n\;a))\,.$$

Whereas $\mathtt{let}\,a = \ldots\,\mathtt{in}\ldots$ suggests a call-by-value discipline, we cannot afford to pre-evaluate each component of the stream. In turn, this imposes a *lazy* call-by-value evaluation discipline for coinductive objects. However, this still might be responsible for some non-terminating reductions, all the more as classical proofs may contain backtrack.

If we analyze what this construction does at the level of types[12], at first approximation it turns a proof (H) of the formula $\forall x^{\mathbb{N}}.A(x)$ (with $A(x) = \exists y.P(x,y)$ in that case) into a proof (the stream H_∞) of the (informal) infinite conjunction $A(0) \wedge A(1) \wedge A(2) \wedge \ldots$. Formally, a proof $\mathtt{cofix}^t_{bx}[p]$ is an inhabitant of a coinductive formula, written $\nu^t_{Xx}A$ (where t is a terms and which binds the variables X and n). The typing rule is given by:

$$\frac{\Gamma \vdash t : T \quad \Gamma, x : T, b : \forall y^T.Xy \vdash p : A}{\Gamma \vdash \mathtt{cofix}^t_{bx}[p] : \nu^t_{Xx}A}\;(\mathtt{cofix})$$

with the side condition that X can only occurs in positive position in A. Coinductive formulas are defined with a reduction rules which is very similar to the rule for the co-fixpoint:

$$\nu^t_{Xx}A \quad \triangleright \quad A[t/x][\nu^t_{Xy}A/Xy]$$

In particular, the term $\mathtt{cofix}^0_{bn}[(Hn, b(S(n))]$ is thus an inhabitant of a coinductively defined (infinite) conjunction, written $\nu^0_{Xn}(A(n) \wedge X(S(n)))$. This formula indeed reduces accordingly to the reduction of the stream:

$$\nu^0_{Xn}(A(n) \wedge X(S(n))) \;\triangleright\; A(0) \wedge [\nu^1_{Xn}(A(n) \wedge X(S(n)))] \;\triangleright\; A(0) \wedge A(1) \wedge [\nu^0_{Xn}(A(n) \wedge X(S(n)))] \;\triangleright\; \ldots$$

More generally, at the level of formulas, the key was to identify the formula $A(x)$ and a suitable law $g : \mathbb{N} \to T$ to turn a proof of $\forall x^T.A(x)$ into the conjunction $A(g(0)) \wedge A(g(1)) \wedge A(g(2)) \wedge \ldots$. In the case of the axiom of countable choice, this law was simply this identity. In the case of the axiom of dependent choice, the law g we are looking for is precisely the choice function. We can thus use the same trick to define a proof term for DC. The stream we actually construct corresponds to the coinductive formula $\nu^{x_0}_{Xn}[\exists y^{\mathbb{N}}.(P(x,y) \wedge X(y))]$, which ultimately unfolds into:

$$\nu^{x_0}_{Xn}[\exists y.(P(x,y) \wedge X(y))] \quad \triangleright \quad \cdots \quad \triangleright \quad \exists x_1^{\mathbb{N}}.(P(x_0,x_1) \wedge \exists x_2^{\mathbb{N}}.(P(x_1,x_2) \wedge \exists x_3^{\mathbb{N}}.(P(x_2,x_3) \wedge \ldots)))$$

Given a proof $H : \forall x.\exists y.P(x,y)$ and a term x_0 , we can define a stream corresponding to this coinductive formula by $\boldsymbol{str}\,x_0 := \mathtt{cofix}^{x_0}_{bn}[(\mathtt{dest}\;H\,n\;\mathtt{as}\,((y,c))\;\mathtt{in}\,(y,(c,(b\,y)))]$. This term reduces as expected:

$$(x_0, \boldsymbol{str}\,x_0) \;\to\; (x_0,(x_1,(p_1, \boldsymbol{str}\,x_1))) \;\to\; (x_0,(x_1,(p_1,(x_2,(p_2, \boldsymbol{str}\,x_2))))) \;\to\; \ldots$$

where $p_i : P(x_{i-1},x_i)$. From there, it is almost direct to extract the choice function f (which maps any $n \in \mathbb{N}$ to x_n) and the corresponding certificate that $(f(0) = x_0 \wedge \forall n \in \mathbb{N}.P(f(n),f(S(n))))$. In practice, it essentially amounts to define the adequate $\mathtt{nth}$ function. We will give a complete definition of the proof term for the axiom of dependent choice in Chapter 8.

[12] We delay the formal introduction of a type system and the given of the typing derivation for $AC_{\mathbb{N}}$ to Chapter 8.

5.2.2 An overview of dPA$^\omega$

Formally, the calculus dPA$^\omega$ is a proof system for the language of classical arithmetic in finites types (abbreviated PA$^\omega$), where the 'd' stands for "dependent". Its stratified presentation allows us to separate terms (the arithmetical objects) from proofs. Finite types and formulas are thus separated as well, corresponding to the following syntax:

$$\begin{array}{lrcl} \textbf{Types} & T,U & ::= & \mathbb{N} \mid T \to U \\ \textbf{Formulas} & A,B & ::= & \top \mid \bot \mid t = u \mid A \wedge B \mid A \vee B \mid \Pi a : A.B \mid \forall x^T.A \mid \exists x^T.A \mid \nu^t_{x,f} A \end{array}$$

Terms, denoted by $t,u,\ldots$ are meant to represent arithmetical objects, their syntax thus includes:

- a term 0 and a successor S;
- an operator $\mathtt{rec}^t_{xy}[t_0 \mid t_S]$ for recursion, which binds the variables x and y: where t is the term on which the recursion is performed, t_0 is the term for the case $t = 0$ ans t_S is the term for case $t = S(t')$;
- λ-abstraction $\lambda x.t$ to define functions;
- terms application $t\,u$;
- a `wit` constructor to extract the witness of a dependent sum.

As for proofs, denoted by $p,q,\ldots$, they contain:

- pairs (p,q) to prove logical conjunctions;
- destructors of pairs $\mathtt{split}\ p\ \mathtt{as}\ (a_1,a_2)\ \mathtt{in}\ q$ which binds the variables a_1 and a_2 in q;
- injections $\iota_i(p)$ for the logical disjunction;
- pattern-matching $\mathtt{case}\ p\ \mathtt{of}\ [a_1.p_1 \mid a_2.p_2]$ which binds the variables a_1 in p_1 and a_2 in p_2;
- a proof term `refl` which is the proof of atomic equalities $t = t$;
- $\mathtt{subst}\ p\,q$ which eliminates an equality proof $p : t = u$ to get a proof of $B[u]$ from a proof $q : B[t]$;
- pairs (t,p) where t is a term and p a proof for the dependent sum type;
- $\mathtt{prf}\ p$ which allows us to extract the certificate of a dependent pair;
- non-dependent destructors $\mathtt{dest}\ p\ \mathtt{as}\ (x,a)\ \mathtt{in}\ q$ which binds the variables x and a in q;
- abstractions over terms $\lambda x.p$ and applications $p\,t$;
- (possibly) dependent abstractions over proofs $\lambda a.p$ and applications $p\,q$;
- a construction $\mathtt{let}\ a = p\ \mathtt{in}\ q$, which binds the variable a in q and which allows for sharing;
- operators $\mathtt{ind}^t_{ax}[p_0 \mid p_S]$ and $\mathtt{cofix}^t_{bx}[p]$ that we already described for inductive and coinductive reasoning;
- control operators $\mathtt{catch}_\alpha\, p$ (which binds the variable α in p) and $\mathtt{throw}\ \alpha\, p$ (where α is a variable and p a proof)
- $\mathtt{exfalso}\ p$ where p is intended to be a proof of false.

This results in the following syntax:

$$\begin{array}{lrcl} \textbf{Terms} & t,u & ::= & x \mid 0 \mid S(t) \mid \mathtt{rec}^t_{xy}[t_0 \mid t_S] \mid \lambda x.t \mid t\ u \mid \mathtt{wit}\ p \\ \textbf{Proofs} & p,q & ::= & a \mid \iota_i(p) \mid \mathtt{case}\ p\ \mathtt{of}\ [a_1.p_1 \mid a_2.p_2] \mid (p,q) \mid \mathtt{split}\ p\ \mathtt{as}\ (a_1,a_2)\ \mathtt{in}\ q \\ & & \mid & (t,p) \mid \mathtt{prf}\ p \mid \mathtt{dest}\ p\ \mathtt{as}\ (x,a)\ \mathtt{in}\ q \mid \lambda x.p \mid p\,t \\ & & \mid & \lambda a.p \mid p\,q \mid \mathtt{let}\ a = p\ \mathtt{in}\ q \mid \mathtt{refl} \mid \mathtt{subst}\ p\,q \\ & & \mid & \mathtt{ind}^t_{ax}[p_0 \mid p_S] \mid \mathtt{cofix}^t_{bx}[p] \mid \mathtt{exfalso}\ p \mid \mathtt{catch}_\alpha\, p \mid \mathtt{throw}\ \alpha\, p \end{array}$$

The problem of degeneracy caused by the conjoint presence of classical proofs and dependent types is solved by enforcing a compartmentalization between them. Dependent types are restricted to the set of *negative-elimination-free* proofs (NEF), which are a generalization of values preventing from backtracking evaluations by excluding expressions of the form $p\,q$, $p\,t$, $\texttt{exfalso}\ p$, $\texttt{catch}_\alpha\,p$ or $\texttt{throw}\ \alpha p$ which are outside the body of a λx or λa. Syntactically, they are defined by:

$$\begin{array}{lrl}
\textbf{Values} & V_1,V_2 ::= & a \mid \iota_i(V) \mid (V_1,V_2) \mid (t,V) \mid \lambda x.p \mid \lambda a.p \mid \texttt{refl} \\
\textbf{NEF} & N_1,N_2 ::= & a \mid \iota_i(N) \mid \texttt{case}\ p\ \texttt{of}\,[a_1.N_1 \mid a_2.N_2] \mid (N_1,N_2) \mid \texttt{split}\ N_1\ \texttt{as}\,(a_1,a_2)\ \texttt{in}\ N_2 \\
 & \mid & (t,N) \mid \texttt{prf}\ N \mid \texttt{dest}\ N_1\ \texttt{as}\,(x,a)\ \texttt{in}\ N_2 \mid \lambda x.p \\
 & \mid & \lambda a.p \mid \texttt{let}\,a = N_1\,\texttt{in}\,N_2 \mid \texttt{refl} \mid \texttt{subst}\ N_1\,N_2 \\
 & \mid & \texttt{ind}_{ax}^{t}[N_0 \mid N_S] \mid \texttt{cofix}_{bx}^{t}[N]
\end{array}$$

This allows to restrict typing rules involving dependencies, notably the rules for prf or let = in:

$$\frac{\Gamma \vdash p : \exists x^T.A(x) \quad p \in \textsc{nef}}{\Gamma \vdash \texttt{prf}\ p : A(\texttt{wit}\ p)}\ (\texttt{prf}) \qquad \frac{\Gamma \vdash p : A \quad \Gamma, a : A \vdash q : B \quad a \notin FV(B)\ \text{if}\ p \notin \textsc{nef}}{\Gamma \vdash \texttt{let}\,a = p\,\texttt{in}\,q : B[p/a]}\ (\textsc{Cut})$$

About reductions, let us simply highlight the fact that they globally follow a call-by-value discipline, for instance in this sample:

$$\begin{array}{lcl}
(\lambda a.p)\,q & \rightarrow & \texttt{let}\,a = q\,\texttt{in}\,p \\
\texttt{let}\,a = (p_1,p_2)\,\texttt{in}\,p & \rightarrow & \texttt{let}\,a_1 = p_1\,\texttt{in}\,\texttt{let}\,a_2 = p_2\,\texttt{in}\,p[(a_1,a_2)/a] \\
\texttt{let}\,a = V\,\texttt{in}\,p & \rightarrow & p[V/a]
\end{array}$$

except for co-fixpoints which are lazily evaluated:

$$\begin{array}{lcl}
F[\texttt{let}\,a = \texttt{cofix}_{bx}^{t}[q]\,\texttt{in}\,p] & \rightarrow & \texttt{let}\,a = \texttt{cofix}_{bx}^{t}[q]\,\texttt{in}\,F[p] \\
\texttt{let}\,a = \texttt{cofix}_{bx}^{t}[q]\,\texttt{in}\,D[a] & \rightarrow & \texttt{let}\,a = q[\lambda y.\texttt{cofix}_{bx}^{y}[q]/b][t/x]\,\texttt{in}\,D[a]
\end{array}$$

In the previous rules, the first one expresses the fact that evaluation of co-fixpoint under contexts $F[\,]$ are momentarily delayed. The second rules precisely corresponds to a context where the co-fixpoint is linked to a variable a whose value is needed, a step of unfolding is then performed.

The full type system, as well as the complete set of reduction rules, are given in [70], and will be restated with a different presentation in Chapter 8. In the same paper, some important properties of the calculus are given. In particular, dPA$^\omega$ verifies the property of subject reduction, and provided it is normalizing, there is no proof of false.

Theorem 5.1 (Subject reduction)**.** *If* $\Gamma \vdash p : A$ *and* $p \rightarrow q$*, then* $\Gamma \vdash q : A$*.*

Proof (sketch). By induction on the derivation of $p \rightarrow q$, see [70]. □

Theorem 5.2 (Conservativity)**.** *Provided dPA$^\omega$ is normalizing, if A is $\rightarrow$-$\vee$-*wit*-$\forall$-free, and $\vdash_{dPA^\omega} p : A$, there is a value V such that $\vdash_{HA^\omega} V : A$.*

Proof (sketch). Considering a closed proof p of A, p can be reduced. By analysis of the different possible cases, it can be found a closed value of type A. Then using the fact that A is a $\rightarrow$-$\vee$-wit-$\forall$-free formula, V does not contain any subexpression of the form $\lambda x.p$ or $\lambda a.p$, by extension it does not contain either any occurrence of $\texttt{exfalso}\ p$, $\texttt{catch}_\alpha\,p$ or $\texttt{throw}\ \alpha p$ and is thus a proof of A already in HA^ω. □

Theorem 5.3 (Consistency)**.** *Provided dPA$^\omega$ is normalizing, it is consistent, that is:* $\nvdash_{dPA^\omega} p : \bot$*.*

Proof. The formula $\bot$ is a particular case of $\rightarrow$-$\vee$-wit-$\forall$-free formula, thus the existence of a proof of false in dPA$^\omega$ would imply the existence of a contradiction already in HA^ω, which is absurd. □

The last two results rely on the property of normalization. Unfortunately, the proof sketch that is given in [70] to support the claim that dPA$^\omega$ normalizes turns out to be hard to formalize properly. Since, moreover, dPA$^\omega$ contains both control operators (allowing for backtrack) and co-fixpoints (allowing infinite objects, like streams), which can be combined and interleaved, we should be very suspicious *a priori* about this property. Anyhow, the proof sketch from [70] to use metamathematical arguments, which are more distant from a computational analysis through a proof by realizability or by means of a continuation-passing style translation. Such proofs are of interest in themselves already for what they taught us about the fine behavior of a calculus.

5.3 Toward a proof of normalization for dPA$^\omega$

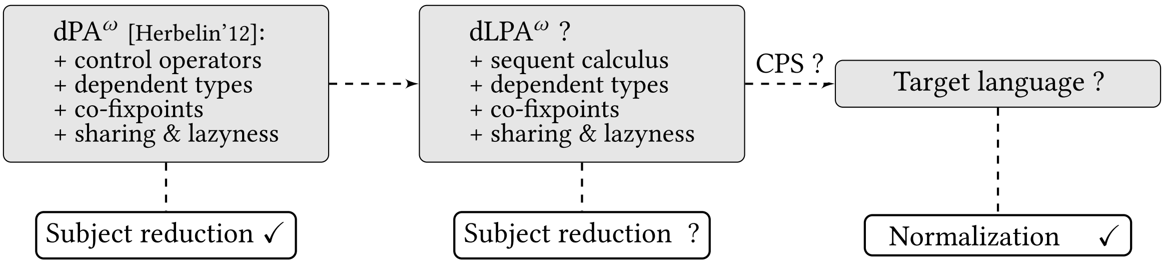
5.3.1 The big picture

An important part of this thesis has been devoted to the search for a proof of normalization for dPA$^\omega$ by means of a realizability interpretation or by a continuation-passing-style translation. Aside from the very result of normalization, this approach is of interest for different reasons which are deeply related to the difficulties of obtaining such a proof. Indeed, a direct continuation-passing style is very harsh to obtain for dPA$^\omega$ as such. In addition to the difficulties caused by control operators and co-fixpoints, the reduction system is defined in a natural-deduction style with contextual rules (as in the rule to reduce proofs of the shape $\mathtt{let}\, a = \mathtt{cofix}^t_{bx}[p]\ \mathtt{in}\, D[a]$) where the contexts involved can be of arbitrary depth. This kind of rules are, in general and especially in this case, very difficult to translate faithfully through a continuation-passing style translation.

All in all, there are several difficulties in getting a direct proof by CPS or realizability. Hence, we shall study them separately, hopefully solving them independently will lead us to a solution to the main problem. Roughly, our strategy consists of two steps:

1. reduce dPA$^\omega$ to an equivalent presentation in a sequent calculus fashion,
2. use the methodology of semantic artifacts to define a CPS or a realizability interpretation.

Indeed, a sequent calculus presentation of a calculus is usually a good intermediate step for compilation or for CPS translations [39]. This presentation should of course verify at least the property of subject reduction and its reduction system should mimic the one of dPA$^\omega$. Schematically, this corresponds to the following roadmap where question marks indicate what is to be done:



To be fair, this approach is idealistic. In particular, we will not formally define an embedding for the first arrow, since we are not interested in dPA$^\omega$ for itself, but rather in the computational content of the proofs for countable and dependent choice. Hence, we will content ourself with a sequent calculus presentation of dPA$^\omega$ which allows for similar proof terms, which we call dLPA$^\omega$, without bothering to prove that the reduction systems are equivalent. As for the second arrow, as advocated in the previous section, the search for a continuation-passing style translation or a realizability interpretation can coincide for a large part. We shall thus apply the methodology of semantic artifacts and in the end, choose the easiest possibility.

From this roadmap actually arises two different subproblems that are already of interest in themselves. Forgetting about the general context of dPA$^\omega$, we shall first wonder whether these easier questions have an answer:

1. Is it possible to define a (classical) sequent calculus with a form of dependent types? If so, would it be compatible with a typed continuation-passing style translation?
2. Can we prove the normalization of a call-by-need calculus with control operators? Can we define a Krivine realizability interpretation of such a calculus?

5.3.2 Realizability interpretation and CPS translation of classical call-by-need

Fortunately, there were already some work in the direction proposed by the second item. In two consecutive articles, Ariola *et al.* studied the question of defining sequent-calculus style versions of call-by-need, leading to a natural extension of call-by-need with control operators [6, 4]. Such a calculus can be expressed in the framework of the $\lambda\mu\tilde{\mu}$-calculus (Chapter 4), and by applying the same methodology of semantic artifacts, the authors showed how to derive (an untyped) CPS translation to the pure λ-calculus. This translation is in fact an environment-and-continuation-passing style translation, so that there is no direct way of inferring a type translation from the computational one. The question thus becomes: can we type this translation to prove the normalization of a call-by-need calculus with control operators? Does this translation lead to a realizability interpretation as it usually does with the call-by-name and call-by-value $\lambda\mu\tilde{\mu}$-calculi?

We shall see in Chapter 6 that the methodology of semantics artifacts can be pushed one step further to obtain a realizability interpretation for the $\overline{\lambda}_{[lv\tau\star]}$-calculus, a call-by-need calculus with control operators and explicit stores. Aside to prove the normalization of the calculus, this also open the door to the interpretation of stores, memory cells in Krivine realizability. Besides, this interpretation a type system, which is an extension of system F and that we call F_Υ. This allows us to type the CPS translation from [4]. Interestingly, we will see that through the translation, the preservation of typing for the store (which is extensible) is obtained by means of a Kripke-style forcing. As far as we know, all these results constitute new contributions.

5.3.3 A sequent calculus with dependent types

The first question, that is to develop a (classical) sequent calculus with dependent types and to ensure the compatibility with a CPS translation, is harder. Indeed, while sequent calculi smoothly supports abstract machine and continuation-passing style interpretations, there is no such presentation of a language with dependent types. Besides, viewed the other way round—can we add control operators to a language with dependent types?—, the question has to do with the more general problem of including side-effects in (dependent) type theory. This issue is one of the hot topic from the past few years in theoretical computer science, in that it aims at filling the gap between type theories and mainstream languages. If there have been proposals for different classes of side-effects, mainly through monads, control operators and classical logic usually do not fit in the picture.

In Chapter 7, we shall start from the call-by-value $\lambda\mu\tilde{\mu}$-calculus and see how to design a minimal language with a value restriction and a type system that includes a list of explicit dependencies to maintain type safety. We will then show how to relax the value restriction and introduce delimited continuations to directly prove the consistency by means of a continuation-passing-style translation. The translation will faithfully embody the dependencies and preserve the normalization. Finally, we will relate our calculus to a similar system by Lepigre [108], whose consistency is proved by means of a realizability interpretation. We present a methodology to transfer properties from his system to our calculus, in particular we can infer proofs of normalization and soundness for our calculus.

6- Normalization of classical call-by-need

The call-by-need evaluation strategy

> *A famous functional programmer once was asked to give an overview talk. He began with : "This talk is about lazy functional programming and call by need." and paused. Then, quizzically looking at the audience, he quipped: "Are there any questions?" There were some, and so he continued: "Now listen very carefully, I shall say this only once."*

This story, borrowed from [37], illustrates demand-driven computation and memoization of intermediate results, two key features of the call-by-need evaluation strategy that distinguish it from the call-by-name and call-by-value evaluation strategies (see Section 2.1.4).

The *call-by-name* evaluation strategy passes arguments to functions without evaluating them, postponing their evaluation to each place where the argument is needed, re-evaluating the argument several times if needed. For instance, the following reduction paths correspond to call-by-name evaluations in the λ-calculus extended with natural numbers:

$$\begin{array}{rlll}
(\lambda xy.yx)\,(2+3)\,(\lambda x.1) & \longrightarrow_\beta (\lambda y.y(2+3))\,\lambda x.1 \longrightarrow_\beta & (\lambda x.1)\,(2+3) & \longrightarrow_\beta 1 \\
(\lambda xy.yx)\,(2+3)\,(\lambda x.x) & \longrightarrow_\beta (\lambda y.y(2+3))\,\lambda x.x \longrightarrow_\beta & (\lambda x.x)\,(2+3) & \longrightarrow_\beta 2+3 \longrightarrow_\beta 5 \\
(\lambda xy.yx)\,(2+3)\,(\lambda x.x\times x) & \xrightarrow{2}_\beta (\lambda x.x\times x)\,(2+3) \longrightarrow_\beta & (2+3)\times(2+3) & \xrightarrow{2}_\beta 5\times 5 \longrightarrow_\beta 25
\end{array}$$

We observe for instance that (2 + 3) is never evaluated in the first example, while it is computed twice for the third one.

Conversely, the *call-by-value* evaluation strategy evaluates the arguments of a function into so-called "values" prior to passing them to the function. The evaluation is then shared between the different places where the argument is needed. Yet, if the argument is not needed, it is evaluated uselessly. The evaluation of the same examples in call-by-value gives:

$$\begin{array}{rlllll}
(\lambda xy.yx)\,(2+3)\,(\lambda x.1) & \longrightarrow_\beta & (\lambda xy.yx)\,5\,(\lambda x.1) & \longrightarrow_\beta & (\lambda y.y5)\,(\lambda x.1) & \longrightarrow_\beta (\lambda x.1)\,5 \longrightarrow_\beta 1 \\
(\lambda xy.yx)\,(2+3)\,(\lambda x.x) & \longrightarrow_\beta & (\lambda xy.yx)\,5\,(\lambda x.x) & \longrightarrow_\beta & (\lambda y.y5)\,(\lambda x.x) & \longrightarrow_\beta (\lambda x.x)\,5 \longrightarrow_\beta 5 \\
(\lambda xy.yx)\,(2+3)\,(\lambda x.x\times x) & \longrightarrow_\beta & (\lambda xy.yx)\,5\,(\lambda x.x\times x) & \longrightarrow_\beta & (\lambda y.y5)\,(\lambda x.x\times x) & \xrightarrow{2}_\beta \quad 5\times 5 \quad \longrightarrow_\beta 25
\end{array}$$

We notice that in the first case, (2 + 3) is always evaluated once, which is better in the third case but useless in the first one. Also, remark that at the time where it is evaluated (the first step), it is impossible to predict how many times the argument will be used because it depends on the function that will be bind later to y (compare the second and third examples).

The *call-by-need* evaluation strategy is an evaluation strategy which evaluates arguments of functions only when needed, and, when needed, shares the computed results across all places where the argument is needed. In the first presentations of call-by-need λ-calculi [7, 112], this was done thanks to an additional $\mathtt{let}\,x = \ldots\,\mathtt{in}\ldots$ constructor. The first example, in call-by-need, reduces as follows:

$$\begin{array}{rl}
(\lambda xy.yx)\,(2+3)\,(\lambda x.1) & \longrightarrow_\beta \mathtt{let}\,x = 2+3\,\mathtt{in}\,(\lambda y.yx)\,(\lambda x.1) \\
& \longrightarrow_\beta \mathtt{let}\,x = 2+3\,\mathtt{in}\ \mathtt{let}\,y = \lambda x.1\,\mathtt{in}\ y\,x \\
& \longrightarrow_\beta \mathtt{let}\,x = 2+3\,\mathtt{in}\ \mathtt{let}\,y = \lambda x.1\,\mathtt{in}\,(\lambda x.1)x \\
& \longrightarrow_\beta \mathtt{let}\,x = 2+3\,\mathtt{in}\ \mathtt{let}\,y = \lambda x.1\,\mathtt{in}\ \mathtt{let}\,z = x\,\mathtt{in}\,1
\end{array}$$

In particular, we observe that since it is never needed, $(2+3)$ is not evaluated. As for the third example, the reduction path is as follows[1]:

$$\begin{array}{rl}
(\lambda xy.yx)\,(2+3)\,(\lambda x.x\times x) & \longrightarrow_\beta \mathtt{let}\,x = 2+3\,\mathtt{in}(\lambda y.yx)\,(\lambda x.x\times x)\\
& \longrightarrow_\beta \mathtt{let}\,x = 2+3\,\mathtt{in}\,\mathtt{let}\,y = (\lambda x.x\times x)\,\mathtt{in}\,yx\\
& \longrightarrow_\beta \mathtt{let}\,x = 2+3\,\mathtt{in}\,\mathtt{let}\,y = (\lambda x.x\times x)\,\mathtt{in}(\lambda x.x\times x)x\\
& \longrightarrow_\beta \mathtt{let}\,x = 2+3\,\mathtt{in}\,\mathtt{let}\,y = (\lambda x.x\times x)\,\mathtt{in}\,\mathtt{let}\,z = x\,\mathtt{in}\,z\times z\\
& \longrightarrow_\beta \mathtt{let}\,x = 2+3\,\mathtt{in}\,\mathtt{let}\,y = (\lambda x.x\times x)\,\mathtt{in}\,\mathtt{let}\,z = x\,\mathtt{in}\,x\times z\\
& \longrightarrow_\beta \mathtt{let}\,x = 5\,\mathtt{in}\,\mathtt{let}\,y = (\lambda x.x\times x)\,\mathtt{in}\,\mathtt{let}\,z = x\,\mathtt{in}\,x\times z\\
& \longrightarrow_\beta \mathtt{let}\,x = 5\,\mathtt{in}\,\mathtt{let}\,y = (\lambda x.x\times x)\,\mathtt{in}\,\mathtt{let}\,z = x\,\mathtt{in}\,5\times z\\
& \longrightarrow_\beta \mathtt{let}\,x = 5\,\mathtt{in}\,\mathtt{let}\,y = (\lambda x.x\times x)\,\mathtt{in}\,\mathtt{let}\,z = x\,\mathtt{in}\,5\times x\\
& \longrightarrow_\beta \mathtt{let}\,x = 5\,\mathtt{in}\,\mathtt{let}\,y = (\lambda x.x\times x)\,\mathtt{in}\,\mathtt{let}\,z = x\,\mathtt{in}\,5\times 5\\
& \longrightarrow_\beta \mathtt{let}\,x = 5\,\mathtt{in}\,\mathtt{let}\,y = (\lambda x.x\times x)\,\mathtt{in}\,\mathtt{let}\,z = x\,\mathtt{in}\,25
\end{array}$$

We see that each time that function is applied to an argument, the latter is lazily stored. When, further in the execution, $(2+3)$ is demanded by the left-member of the multiplication, its value is computed. Thanks to the $\mathtt{let}\,x = \dots\mathtt{in}\dots$ binder, this value is shared and when it is required a second time by the right-member of the multiplication, it is already available.

The call-by-need evaluation is at the heart of a functional programming language such as Haskell. It has in common with the call-by-value evaluation strategy that all places where a same argument is used share the same value. Nevertheless, it observationally behaves like the call-by-name evaluation strategy, in the sense that a given computation eventually evaluates to a value if and only if it evaluates to the same value (up to inner reduction) along the call-by-name evaluation. In particular, in a setting with non-terminating computations, it is not observationally equivalent to the call-by-value evaluation. Indeed, if the evaluation of a useless argument loops in the call-by-value evaluation, the whole computation loops (*e.g.* in $(\lambda\_.I)\,\Omega)$), which is not the case of call-by-name and call-by-need evaluations.

Continuation-passing style semantics

The call-by-name, call-by-value and call-by-need evaluation strategies can be turned into equational theories. For call-by-name and call-by-value, this was done by Plotkin [139] through continuation-passing style semantics characterizing these theories. For call-by-name, the corresponding induced equational theory[2] is Church's original theory of the λ-calculus based on the operational rule β.

For call-by-value, Plotkin showed that the induced equational theory includes the key operational rule β_V. The induced equational theory was further completed implicitly by Moggi [124] with the convenient introduction of a native `let` operator. Moggi's theory was then explicitly shown complete for CPS semantics by Sabry and Felleisen [148].

For the call-by-need evaluation strategy, a specific equational theory reflecting the intensional behavior of the strategy into a semantics was proposed independently by Ariola and Felleisen [3] and by Maraist, Odersky and Wadler [113]. A continuation-passing style semantics was proposed in the 90s by Okasaki, Lee and Tarditi [128]. However, this semantics does not ensure normalization of simply-typed call-by-need evaluation, as shown in [4], thus failing to ensure a property which holds in the simply-typed call-by-name and call-by-value cases (see Chapter 4).

[1] Observe that, as in the first example, we need to perform α-conversion on the fly, due to the $\mathtt{let}\cdots = \dots\mathtt{in}\dots$ bindings which behave like an explicit substitution. We will come back to this point in Section 6.4.1.

[2] Later on, Lafont, Reus and Streicher [103] gave a more refined continuation-passing style semantics which also validates the extensional rule η.

The $\overline{\lambda}_{lv}$-calculus: call-by-need with control

Continuation-passing style semantics *de facto* gives a semantics to the extension of λ-calculus with control operators, i.e. with operators such as Scheme's `call/cc`, Felleisen's $\mathcal{C}$, $\mathcal{K}$, or $\mathcal{A}$ operators [41], Parigot's μ and [ ] operators [130], Crolard's `catch` and `throw` operators [31]. In particular, even though call-by-name and call-by-need are observationally equivalent in the pure λ-calculus, their different intentional behaviors induce different continuation-passing style semantics, leading to different observational behaviors when control operators are considered.

Nonetheless, the semantics of calculi with control can also be reconstructed from an analysis of the duality between programs and their evaluation contexts, and the duality between the `let` construct (which binds programs) and a control operator such as Parigot's μ (which binds evaluation contexts). As explained in Chapter 4, such an analysis can be done in the context of the $\lambda\mu\tilde{\mu}$-calculus [32, 68].

Such an analysis is done in [4] in a variant of the $\lambda\mu\tilde{\mu}$-calculus which includes co-constants ranged over by $\boldsymbol{\kappa}$. Recall from Section 4.2 that the syntax of the $\lambda\mu\tilde{\mu}$-calculus can be refined into the following subcategories of terms and contexts:

Terms	t	$::=$	$\mu\alpha.c \mid V$	**Contexts**	e	$::=$	$\tilde{\mu}x.c \mid E$
Values	V	$::=$	$a \mid \lambda x.t \mid \boldsymbol{k}$	**Co-values**	E	$::=$	$\alpha \mid t \cdot e \mid \boldsymbol{\kappa}$

to which we add constants $\boldsymbol{k}$ and co-constants $\boldsymbol{\kappa}$. Then, by presenting reduction rules parameterized over a set of terms $\mathcal{V}$ and a set of evaluation contexts $\mathcal{E}$:

$$\begin{array}{lcll}\langle t \| \tilde{\mu}x.c\rangle & \to & c[t/x] & t \in \mathcal{V} \\ \langle \mu\alpha.c \| e\rangle & \to & c[e/\alpha] & e \in \mathcal{E} \\ \langle \lambda x.t \| u \cdot e\rangle & \to & \langle u \| \tilde{\mu}x.\langle t \| e\rangle\rangle & \end{array}$$

the difference between call-by-name and call-by-value can be characterized by the definition of these sets: the call-by-name evaluation strategy amounts to the case where $\mathcal{V} \triangleq$ *Proofs* and $\mathcal{E} \triangleq$ *Co-values* while call-by-value dually corresponds to $\mathcal{V} \triangleq$ *Values* and $\mathcal{E} \triangleq$ *Contexts*.

As for the call-by-need case, intuitively, we would like to set $\mathcal{V} \triangleq$ *Values* (we only substitute evaluated terms of which we share the value) and $\mathcal{E} \triangleq$ *Co-values* (a term is only reduced if it is in front of a co-value). However, such a definition is clearly not enough since any command of the shape $\langle \mu\alpha.c \| \tilde{\mu}x.c'\rangle$ would be blocked. We thus need to understand how the computation is driven forward, that is to say when we need to reduce terms. We observed that contexts that are either a co-constant $\boldsymbol{\kappa}$ or an applicative context[3] $t \cdot E$ eagerly demand a value. Such contexts are called *forcing contexts*, and denoted by F. When a variable x is in front of a forcing context, that is in $\langle x \| F\rangle$, the variable x is said to be *needed* or *demanded.* This allows us to identify meta-contexts C which are nesting of commands of the form $\langle t \| e\rangle$ for which neither t is in $\mathcal{V}$ (meaning it is some $\mu\alpha.c$) nor e in $\mathcal{E}$ (meaning it is an instance of some $\tilde{\mu}x.c$ which is not a forcing context). These contexts, defined by the following grammar:

Meta-contexts $\quad C[\,] \ ::= \ [\,] \mid \langle \mu\alpha.c \| \tilde{\mu}x.C[\,]\rangle$

are such that in a $\tilde{\mu}$-binding of the form $\tilde{\mu}x.C[\langle x \| F\rangle]$, x is needed and a value is thus expected. These contexts, called *demanding contexts* are evaluation contexts whose evaluation is blocked on the evaluation of x, therefore requiring the evaluation of what is bound to x. In this case, we say that the bound variable x has been *forced.*

All this suggests another refinement of the syntax, introducing a division between *weak* co-values (resp. *weak* values), also called *catchable* contexts (since they are the one caught by a $\mu\alpha$ binder), and *strong* co-values (resp. *strong* values), which are precisely the forcing contexts. In comparison, with

[3] There is a restriction on the form of applicative contexts: the general form $t \cdot e$ is not necessarily a valid application, since for example in $\langle \mu\alpha.c \| t \cdot \tilde{\mu}x\langle y \| \alpha\rangle\rangle$, the context $t \cdot \tilde{\mu}x\langle y \| \alpha\rangle$ forces the execution of c even though its value is not needed. Applicative contexts are thus considered of the restricted shape $t \cdot E$.

$$\frac{(x:A)\in\Gamma}{\Gamma\vdash x:A\mid\Delta}\ (x) \qquad \frac{\Gamma,x:A\vdash t:B\mid\Delta}{\Gamma\vdash\lambda x.t:A\to B\mid\Delta}\ (\to_r) \qquad \frac{c:(\Gamma\vdash\Delta,\alpha:A)}{\Gamma\vdash\mu\alpha.c:A\mid\Delta}\ (\mu)$$

$$\frac{(\alpha:A)\in\Delta}{\Gamma\mid\alpha:A\vdash\Delta}\ (\alpha) \qquad \frac{\Gamma\vdash t:A\mid\Delta \qquad \Gamma\mid E:B\vdash\Delta}{\Gamma\mid t\cdot E:A\to B\vdash\Delta}\ (\to_l) \qquad \frac{c:(\Gamma,x:A\vdash\Delta)}{\Gamma\mid\tilde{\mu}x.c:A\vdash\Delta}\ (\tilde{\mu})$$

$$\frac{\Gamma\vdash t:A\mid\Delta \qquad \Gamma\mid e:A\vdash\Delta}{\langle t\|e\rangle:(\Gamma\vdash\Delta)}\ (\textsc{Cut}) \qquad \frac{(\boldsymbol{\kappa}:A)\in\mathcal{S}}{\Gamma\mid\boldsymbol{\kappa}:A\vdash\Delta}\ (\boldsymbol{\kappa}) \qquad \frac{(\boldsymbol{k}:X)\in\mathcal{S}}{\Gamma\vdash\boldsymbol{k}:X\mid\Delta}\ (\boldsymbol{k})$$

Figure 6.1: Typing rules for $\overline{\lambda}_{lv}$

our former division, note that *catchable contexts* correspond to the union of former *co-values* with *demanding contexts*. Formally, the syntax is defined by[4]:

Strong values	v	$::=$	$\lambda x.t \mid \boldsymbol{k}$	**Forcing contexts**	F	$::=$	$t\cdot E \mid \boldsymbol{\kappa}$
Weak values	V	$::=$	$v \mid x$	**Catchable contexts**	E	$::=$	$F \mid \alpha \mid \tilde{\mu}x.C[\langle x\|F\rangle]$
Terms	t	$::=$	$V \mid \mu\alpha.c$	**Contexts**	e	$::=$	$E \mid \tilde{\mu}x.c$

We can finally define $\mathcal{V} \triangleq$ *Weak values* and $\mathcal{E} \triangleq$ *Catchable contexts*. The so-defined call-by-need calculus is close to the calculus called $\overline{\lambda}_{lv}$ in Ariola *et al* [4][5].

The $\overline{\lambda}_{lv}$ reduction, written as $\to_{lv}$, denotes thus the compatible reflexive transitive closure of the rules:

$$\begin{array}{ccc}\langle V\|\tilde{\mu}x.c\rangle & \to_{lv} & c[V/x]\\ \langle \mu\alpha.c\|E\rangle & \to_{lv} & c[E/\alpha]\\ \langle \lambda x.t\|u\cdot E\rangle & \to_{lv} & \langle u\|\tilde{\mu}x.\langle t\|E\rangle\rangle\end{array}$$

Observe that the next reduction is not necessarily at the top of the command, but may be buried under several bound computations $\mu\alpha.c$. For instance, the command $\langle\mu\alpha.c\|\tilde{\mu}x_1.\langle x_1\|\tilde{\mu}x_2.\langle x_2\|F\rangle\rangle\rangle$, where x_1 is not needed, reduces to $\langle\mu\alpha.c\|\tilde{\mu}x_1.\langle x_1\|F\rangle\rangle$, which now demands x_1.

The $\overline{\lambda}_{lv}$-calculus can be equipped with a type system (see Figure 6.1) made of the usual rules of the classical sequent calculus [32], where we adopt the convention that constants $\boldsymbol{k}$ and co-constants $\boldsymbol{\kappa}$ come with a signature $\mathcal{S}$ which assigns them a type.

Realizability and CPS interpretations of classical call-by-need

In the cases of the call-by-name and call-by-value evaluation strategies, the approach based on the $\lambda\mu\tilde{\mu}$-calculus leads to continuation-passing style semantics (Sections 4.4.4 and 4.5.3) similar to the ones given by Plotkin or, in the call-by-name case, also to the one by Lafont, Reus and Streicher [103]. In the case of call-by-need calculus, a continuation-passing style semantics for $\overline{\lambda}_{lv}$ is defined in [4] via a calculus called $\overline{\lambda}_{[lv\tau\star]}$. This calculus is equivalent to $\overline{\lambda}_{lv}$ but is presented in such a way that the head redex of a command can be found by looking only at the surface of the command, from which a continuation-passing style semantics directly comes. This semantics, distinct from the one in [128], is the object of study in this chapter.

The contribution of this chapter is twofold. On the one hand, we give a proof of normalization for the $\overline{\lambda}_{[lv\tau\star]}$-calculus. The normalization is obtained by means of a realizability interpretation of the calculus,

[4] In syntactic category, we implicitly assume $\tilde{\mu}x.c$ to only cover the cases which are not of the form $\tilde{\mu}x.C[\langle x\|F\rangle]$.

[5] The difference is in the fact that we had constants to preserve the duality. Also, a similar calculus, which we shall call weak $\overline{\lambda}_{lv}$, was previously studied in [6] with $\mathcal{E}$ defined instead to be $\tilde{\mu}x.C[\langle x\|E\rangle]$ (with same definition of C) and a definition of $\mathcal{V}$ which was different whether $\tilde{\mu}x.c$ was a forcing context ($\mathcal{V}$ was then the strong values) or not ($\mathcal{V}$ was then the weak values). Another variant is discussed in Section 6 of [4] where $\mathcal{E}$ is similarly defined to be $\tilde{\mu}x.C[\langle x\|E\rangle]$ and $\mathcal{V}$ is defined to be (uniformly) the strong values. All three semantics seem to make sense to us.

which is inspired from Krivine classical realizability [95]. As advocated in Section 4.3.3, the realizability interpretation is obtained by pushing one step further the methodology of Danvy's semantics artifacts already used in [4] to derive the continuation-passing-style semantics. While we only use it here to prove the normalization of the $\overline{\lambda}_{[lv\tau\star]}$-calculus, our interpretation incidentally suggests a way to adapt Krivine's classical realizability to a call-by-need setting. This opens the door to the computational interpretation of classical proofs using lazy evaluation or shared memory cells.

On the other hand, we provide a type system for the continuation-passing-style transformation presented in [4] for the $\overline{\lambda}_{[lv\tau\star]}$-calculus such that the translation is well-typed. This presents various difficulties. First, since the evaluation of terms is shared, the continuation-passing-style translation is actually combined with a store-passing-style transformation. Second, as the store can grow along the execution, the translation also includes a Kripke-style forcing to address the extensibility of the store. This induces a target language which we call system F_Υ and which is an extension of Girard-Reynolds system F [60] and Cardelli system F$_{<:}$ [22]. Last but not least, the translation needs to take into account the problem of α-conversion. In a nutshell, this is due to the fact that terms can contain unbound variables that refer to elements of the store. So that a collision of names can result in auto-references and non-terminating terms. We deal with this in two-ways: we first elude the problem by using a fresh name generator and an explicit renaming of variables through the translation. Then we refine the translation to use De Bruijn levels to access elements of the store, which has the advantage of making it closer to an actual implementation. Surprisingly, the passage to De Bruijn levels also unveils some computational content related to the extension of stores.

6.1 The $\overline{\lambda}_{[lv\tau\star]}$-calculus

6.1.1 Syntax

While all the results that are presented in the sequel of this chapter could be directly expressed using the $\overline{\lambda}_{lv}$-calculus, the continuation-passing style translation we present naturally arises from the decomposition of this calculus into a different calculus with an explicit *environment*, the $\overline{\lambda}_{[lv\tau\star]}$-calculus [4]. Indeed, as we shall explain thereafter, the decomposition highlights different syntactic categories that are deeply involved in the definition and the typing of the continuation-passing style translation.

The $\overline{\lambda}_{[lv\tau\star]}$-calculus is a reformulation of the $\overline{\lambda}_{lv}$-calculus with explicit environments, which we call *stores*, that are denoted by τ. Stores consists of a list of bindings of the shape $[x := t]$, where x is a term variable and t a term, and of bindings of the shape $[\alpha := e]$ where α is a context variable and e a context. For instance, in the closure $c\tau[x := t]\tau'$, the variable x is bound to t in c and τ'. Besides, the term t might be an unevaluated term (*i.e.* lazily stored), so that if x is eagerly demanded at some point during the execution of this closure, t will be reduced in order to obtain a value. In the case where t indeed produces a value V, the store will be updated with the binding $[x := V]$. However, a binding of this shape (with a value) is fixed for the rest of the execution. As such, our so-called stores somewhat behave like lazy explicit substitutions or mutable environments [6].

The lazy evaluation of terms allows us to reduce a command $\langle \mu\alpha.c \| \tilde{\mu}x.c' \rangle$ to the command c' together with the binding $[x := \mu\alpha.c]$. In this case, the term $\mu\alpha.c$ is left unevaluated ("frozen") in the store, until possibly reaching a command in which the variable x is needed. When evaluation reaches a command of the form $\langle x \| F \rangle\tau[x := \mu\alpha.c]\tau'$, the binding is opened and the term is evaluated in front

[6] To draw the comparison between our structures and the usual notions of stores and environments, two things should be observed. First, the usual notion of store refers to a structure of list that is fully mutable, in the sense that the cells can be updated at any time and thus values might be replaced. Second, the usual notion of environment designates a structure in which variables are bounded to closures made of a term and an environment. In particular, terms and environments are duplicated, *i.e.* sharing is not allowed. Such a structure resemble to a tree whose nodes are decorated by terms, as opposed to a machinery allowing sharing (like ours) whose the underlying structure is broadly a directed acyclic graphs. See for instance [104] for a Krivine abstract machine with sharing.

$$
\begin{array}{ccc}
\langle t \| \tilde{\mu}x.c\rangle\tau & \to & c\tau[x := t] \\
\langle \mu\alpha.c \| E\rangle\tau & \to & c\tau[\alpha := E] \\
\langle V \| \alpha\rangle\tau[\alpha := E]\tau' & \to & \langle V \| E\rangle\tau[\alpha := E]\tau' \\
\langle x \| F\rangle\tau[x := t]\tau' & \to & \langle t \| \tilde{\mu}[x].\langle x \| F\rangle\tau'\rangle\tau \\
\langle V \| \tilde{\mu}[x].\langle x \| F\rangle\tau'\rangle\tau & \to & \langle V \| F\rangle\tau[x := V]\tau' \\
\langle \lambda x.t \| u \cdot E\rangle\tau & \to & \langle u \| \tilde{\mu}x.\langle t \| E\rangle\rangle\tau
\end{array}
$$

Figure 6.2: Reduction rules of the $\overline{\lambda}_{[lv\tau\star]}$-calculus

of the context $\tilde{\mu}[x].\langle x \| F\rangle\tau'$:

$$\langle x \| F\rangle\tau[x := \mu\alpha.c]\tau' \to \langle \mu\alpha.c \| \tilde{\mu}[x].\langle x \| F\rangle\tau'\rangle\tau$$

The reader can think of the previous rule as the "defrosting" operation of the frozen term $\mu\alpha.c$: this term is evaluated in the prefix of the store τ which predates it, in front of the context $\tilde{\mu}[x].\langle x \| F\rangle\tau'$ where the $\tilde{\mu}[x]$ binder is waiting for an (unfrozen) value. This context keeps trace of the suffix of the store τ' that was after the binding for x. This way, if a value V is indeed furnished for the binder $\tilde{\mu}[x]$, the original command $\langle x \| F\rangle$ is evaluated in the updated full store:

$$\langle V \| \tilde{\mu}[x].\langle x \| F\rangle\tau'\rangle\tau \to \langle V \| F\rangle\tau[x := V]\tau'$$

The brackets are used to express the fact that the variable x is forced at top-level (unlike contexts of the shape $\tilde{\mu}x.C[\langle x \| F\rangle]$ in the $\overline{\lambda}_{lv}$-calculus). The reduction system resembles the one of an abstract machine. Especially, it allows us to keep the standard redex at the top of a command and avoids searching through the meta-context for work to be done.

Note that our approach slightly differ from [4] in that we split values into two categories: strong values (v) and weak values (V). The strong values correspond to values strictly speaking. The weak values include the variables which force the evaluation of terms to which they refer into shared strong value. Their evaluation may require capturing a continuation. The syntax of the language is given by:

$$
\begin{array}{llll@{\qquad}llll}
\textbf{Strong values} & v & ::= & \lambda x.t \mid \boldsymbol{k} & \textbf{Forcing contexts} & F & ::= & \boldsymbol{\kappa} \mid t \cdot E \\
\textbf{Weak values} & V & ::= & v \mid x & \textbf{Catchable contexts} & E & ::= & F \mid \alpha \mid \tilde{\mu}[x].\langle x \| F\rangle\tau \\
\textbf{Terms} & t & ::= & V \mid \mu\alpha.c & \textbf{Evaluation contexts} & e & ::= & E \mid \tilde{\mu}x.c
\end{array}
$$

$$
\begin{array}{llll}
\textbf{Closures} & l & ::= & c\tau \\
\textbf{Commands} & c & ::= & \langle t \| e\rangle \\
\textbf{Stores} & \tau & ::= & \varepsilon \mid \tau[x := t] \mid \tau[\alpha := E]
\end{array}
$$

The reduction, written $\to$, is the compatible reflexive transitive closure of the rules [7] given in Figure 6.2.

The different syntactic categories can be understood as the different levels of alternation in a context-free abstract machine: the priority is first given to contexts at level e (lazy storage of terms), then to terms at level t (evaluation of $\mu\alpha$ into values), then back to contexts at level E and so on until level v. These different categories are directly reflected in the definition of the context-free abstract machine (that we will present in Section 6.1.3) and in the continuation-passing style translation (and thus involved when typing it). We choose to highlight this by distinguishing different types of sequents already in the typing rules that we shall now present.

[7] We chose to make the substitutions of α variables effective while they are kept in an environment in [4]. This explains that we have one less rule.

$$\frac{(\boldsymbol{k}:X)\in\mathcal{S}}{\Gamma\vdash_v \boldsymbol{k}:X}\,(\boldsymbol{k})\qquad \frac{\Gamma,x:A\vdash_t t:B}{\Gamma\vdash_v \lambda x.t:A\to B}\,(\to_r)\qquad \frac{(x:A)\in\Gamma}{\Gamma\vdash_V x:A}\,(x)\qquad \frac{\Gamma\vdash_v v:A}{\Gamma\vdash_V v:A}\,(\uparrow^V)$$

$$\frac{(\boldsymbol{\kappa}:A)\in\mathcal{S}}{\Gamma\vdash_F \boldsymbol{\kappa}:A^{\perp\!\!\!\perp}}\,(\boldsymbol{\kappa})\qquad \frac{\Gamma\vdash_t t:A\quad \Gamma\vdash_E E:B^{\perp\!\!\!\perp}}{\Gamma\vdash_F t\cdot E:(A\to B)^{\perp\!\!\!\perp}}\,(\to_l)\qquad \frac{(\alpha:A)\in\Gamma}{\Gamma\vdash_E \alpha:A^{\perp\!\!\!\perp}}\,(\alpha)\qquad \frac{\Gamma\vdash_F F:A^{\perp\!\!\!\perp}}{\Gamma\vdash_E F:A^{\perp\!\!\!\perp}}\,(\uparrow^E)$$

$$\frac{\Gamma\vdash_V V:A}{\Gamma\vdash_t V:A}\,(\uparrow^t)\qquad \frac{\Gamma,\alpha:A^{\perp\!\!\!\perp}\vdash_c c}{\Gamma\vdash_t \mu\alpha.c:A}\,(\mu)\qquad \frac{\Gamma\vdash_E E:A^{\perp\!\!\!\perp}}{\Gamma\vdash_e E:A^{\perp\!\!\!\perp}}\,(\uparrow^e)\qquad \frac{\Gamma,x:A\vdash_c c}{\Gamma\vdash_e \tilde\mu x.c:A^{\perp\!\!\!\perp}}\,(\tilde\mu)$$

$$\frac{\Gamma,x:A,\Gamma'\vdash_F F:A^{\perp\!\!\!\perp}\quad \Gamma\vdash_\tau \tau:\Gamma'}{\Gamma\vdash_E \tilde\mu[x].\langle x\|F\rangle\tau:A^{\perp\!\!\!\perp}}\,(\tilde\mu^{[]})\qquad \frac{\Gamma\vdash_t t:A\quad \Gamma\vdash_e e:A^{\perp\!\!\!\perp}}{\Gamma\vdash_c \langle t\|e\rangle}\,(c)\qquad \frac{\Gamma,\Gamma'\vdash_c c\quad \Gamma\vdash_\tau \tau:\Gamma'}{\Gamma\vdash_l c\tau}\,(l)$$

$$\frac{}{\Gamma\vdash_\tau \varepsilon:\varepsilon}\,(\varepsilon)\qquad \frac{\Gamma\vdash_\tau \tau:\Gamma'\quad \Gamma,\Gamma'\vdash_t t:A}{\Gamma\vdash_\tau \tau[x:=t]:\Gamma',x:A}\,(\tau_t)\qquad \frac{\Gamma\vdash_\tau \tau:\Gamma'\quad \Gamma,\Gamma'\vdash_E E:A^{\perp\!\!\!\perp}}{\Gamma\vdash_\tau \tau[\alpha:=E]:\Gamma',\alpha:A^{\perp\!\!\!\perp}}\,(\tau_E)$$

Figure 6.3: Typing rules of the $\overline{\lambda}_{[lv\tau\star]}$-calculus

6.1.2 Type system

Unlike in the usual type system for sequent calculus where a judgment contains two typing contexts (one on the left for proofs, denoted by Γ, one on the right for contexts denoted by Δ), we use one-sided sequents (see Section 4.2.3.2): we group both typing contexts into one single context, denoting the types for contexts (that used to be in Δ) with the exponent $\perp\!\!\!\perp$. This allows us to draw a strong connection in the sequel between the typing context Γ and the store τ, which contain both kind of terms.

We have nine kinds of sequents, one for typing each of the nine syntactic categories. We write them with an annotation on the $\vdash$ sign, using one of the letters v, V, t, F, E, e, l, c, τ. Sequents themselves are of four sorts: those typing values and terms are asserting a type, with the type written on the right; sequents typing contexts are expecting a type A with the type written $A^{\perp\!\!\!\perp}$; sequents typing commands and closures are black boxes neither asserting nor expecting a type; sequents typing substitutions are instantiating a typing context. In other words, we have the following nine kinds of sequents:

$$\begin{array}{lll} \Gamma\vdash_l l & \Gamma\vdash_t t:A & \Gamma\vdash_e e:A^{\perp\!\!\!\perp} \\ \Gamma\vdash_c c & \Gamma\vdash_V V:A & \Gamma\vdash_E E:A^{\perp\!\!\!\perp} \\ \Gamma\vdash_\tau \tau:\Gamma' & \Gamma\vdash_v v:A & \Gamma\vdash_F F:A^{\perp\!\!\!\perp} \end{array}$$

where types and typing contexts are defined by:

$$A,B ::= X \mid A\to B \qquad\qquad \Gamma ::= \varepsilon \mid \Gamma,x:A \mid \Gamma,\alpha:A^{\perp\!\!\!\perp}$$

The typing rules are given on Figure 6.3 where we assume that a variable x (resp. co-variable α) only occurs once in a context Γ (we implicitly assume the possibility of renaming variables by α-conversion). This type system enjoys the property of subject reduction, whose proof is done by reasoning by induction over the derivation of the reduction $c\tau\to c'\tau'$, and relies on the fact that the type system admits a weakening rule.

Lemma 6.1. *The following rule is admissible for any level o of the hierarchy e,t,E,V,F,v,c,l,τ:*

$$\frac{\Gamma\vdash_o o:A\quad \Gamma\subseteq\Gamma'}{\Gamma'\vdash_o o:A}\,(w)$$

Proof. Easy induction on the structure of typing derivations obtained through the type system in Figure 6.3. □

Theorem 6.2 (Subject reduction). *If* $\Gamma \vdash_l c\tau$ *and* $c\tau \to c'\tau'$ *then* $\Gamma \vdash_l c'\tau'$.

Proof. By induction over the induction over the derivation of the reduction $c\tau \to c'\tau'$ (see Figure 6.2).

- **Case** $\langle t \| \tilde{\mu}x.c\rangle\tau \to c\tau[x := t]$. A typing derivation of the closure on the left-hand side has the form:

$$\dfrac{\dfrac{\Pi_t}{\Gamma,\Gamma' \vdash_t t : A} \quad \dfrac{\dfrac{\Pi_c}{\Gamma,\Gamma',x : A \vdash_c c}\,(c)}{\Gamma,\Gamma' \vdash_e \tilde{\mu}x.c : A}\,(\tilde{\mu}) \quad \dfrac{\Pi_\tau}{\Gamma \vdash_\tau \tau : \Gamma'}}{\dfrac{\Gamma,\Gamma' \vdash_c \langle t \| \tilde{\mu}x.c\rangle}{\Gamma \vdash_l \langle t \| \tilde{\mu}x.c\rangle\tau}\,(l)}\,(c)$$

hence we can derive:

$$\dfrac{\dfrac{\Pi_c}{\Gamma,\Gamma',x : A \vdash_c c}\,(c) \quad \dfrac{\dfrac{\Pi_\tau}{\Gamma \vdash_\tau \tau : \Gamma'} \quad \dfrac{\Pi_t}{\Gamma,\Gamma' \vdash_t t : A}}{\Gamma \vdash_\tau \tau[x := t] : (\Gamma',x : A)}\,(\tau_t)}{\Gamma \vdash_l c\tau[x := t]}\,(l)$$

- **Case** $\langle \mu\alpha.c \| E\rangle\tau \to c\tau[\alpha := E]$. A typing derivation of the closure on the left-hand side has the form:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi_c}{\Gamma,\Gamma',\alpha : A^{\perp\!\!\!\perp} \vdash_c c}\,(c)}{\Gamma,\Gamma' \vdash_t \mu\alpha.c : A}\,(\mu) \quad \dfrac{\dfrac{\Pi_E}{\Gamma,\Gamma' \vdash_E E : A^{\perp\!\!\!\perp}}}{\Gamma,\Gamma' \vdash_e E : A^{\perp\!\!\!\perp}}\,(\uparrow^e)}{\Gamma,\Gamma' \vdash_c \langle \mu\alpha.c \| E\rangle}\,(c) \quad \dfrac{\Pi_\tau}{\Gamma \vdash_\tau \tau : \Gamma'}}{\Gamma \vdash_l \langle \mu\alpha.c \| E\rangle\tau}\,(l)$$

hence we can derive:

$$\dfrac{\dfrac{\Pi_c}{\Gamma,\Gamma',\alpha : A^{\perp\!\!\!\perp} \vdash_c c}\,(c) \quad \dfrac{\dfrac{\Pi_\tau}{\Gamma \vdash_\tau \tau : \Gamma'} \quad \dfrac{\Pi_E}{\Gamma,\Gamma' \vdash_E E : A}}{\Gamma \vdash_\tau \tau[\alpha := E] : (\Gamma',\alpha : A^{\perp\!\!\!\perp})}\,(\tau_E)}{\Gamma \vdash_l c\tau[\alpha := E]}\,(l)$$

- **Case** $\langle V \| \alpha\rangle\tau[\alpha := E]\tau' \to \langle V \| E\rangle\tau[\alpha := E]\tau'$. A typing derivation of the closure on the left-hand side has the form:

$$\dfrac{\dfrac{\dfrac{\Pi_V}{\Gamma,\Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1 \vdash_t V : A} \quad \dfrac{\dfrac{\dfrac{}{\Gamma,\Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1 \vdash_F \alpha : A^{\perp\!\!\!\perp}}}{\Gamma,\Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1 \vdash_E \alpha : A^{\perp\!\!\!\perp}}\,(\alpha)}{\Gamma,\Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1 \vdash_e \alpha : A^{\perp\!\!\!\perp}}\,(\uparrow^e)}{\Gamma,\Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1 \vdash_c \langle V \| \alpha\rangle}\,(c) \quad \dfrac{\dfrac{\dfrac{\Pi_\tau}{\Gamma \vdash \tau : \Gamma_0} \quad \dfrac{\Pi_E}{\Gamma,\Gamma_0 \vdash_E E : A^{\perp\!\!\!\perp}}}{\Gamma \vdash_\tau \tau[\alpha := E] : \Gamma_0,\alpha : A^{\perp\!\!\!\perp}}\,(\tau_E) \quad \Pi_{\tau'}}{\Gamma \vdash_\tau \tau[\alpha := E]\tau' : \Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1}\,(\tau\tau')}{\Gamma \vdash_l \langle V \| \alpha\rangle\tau[\alpha := E]\tau'}\,(l)$$

where we cheated to compact each typing judgment for τ' (corresponding to types in Γ_1) in $\Pi_{\tau'}$. Therefore, we can derive:

$$\dfrac{\dfrac{\dfrac{\Pi_V}{\Gamma,\Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1 \vdash_t V : A} \quad \dfrac{\dfrac{\Pi_E}{\Gamma,\Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1 \vdash_E E : A^{\perp\!\!\!\perp}}}{\Gamma,\Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1 \vdash_e E : A^{\perp\!\!\!\perp}}\,(\uparrow^e)}{\Gamma,\Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1 \vdash_c \langle V \| E\rangle}\,(c) \quad \dfrac{\dfrac{\dfrac{\Pi_\tau}{\Gamma \vdash \tau : \Gamma_0} \quad \dfrac{\Pi_E}{\Gamma,\Gamma_0 \vdash_E E : A^{\perp\!\!\!\perp}}}{\Gamma \vdash_\tau \tau[\alpha := E] : \Gamma_0,\alpha : A^{\perp\!\!\!\perp}}\,(\tau_E) \quad \Pi_{\tau'}}{\Gamma \vdash_\tau \tau[\alpha := E]\tau' : \Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1}\,(\tau\tau')}{\Gamma \vdash_l \langle V \| \alpha\rangle\tau[\alpha := E]\tau'}\,(l)$$

- **Case** $\langle x\|F\rangle\tau[x:=t]\tau' \to \langle t\|\tilde{\mu}[x].\langle x\|F\rangle\tau'\rangle\tau$. A typing derivation of the closure on the left-hand side has the form:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma,\Gamma_0,x:A,\Gamma_1 \vdash_V x:A}\,(x)}{\Gamma,\Gamma_0,x:A,\Gamma_1 \vdash_t x:A}\,(\uparrow^t) \qquad \dfrac{\Pi_F}{\Gamma,\Gamma_0,x:A,\Gamma_1 \vdash_e F:A^{\perp\!\!\!\perp}}}{\Gamma,\Gamma_0,x:A,\Gamma_1 \vdash_c \langle x\|F\rangle}\,(c) \qquad \dfrac{\dfrac{\dfrac{\Pi_\tau}{\Gamma\vdash\tau:\Gamma_0} \quad \dfrac{\Pi_t}{\Gamma,\Gamma_0\vdash_t t:A}}{\Gamma\vdash_\tau \tau[x:=t]:\Gamma_0,x:A}\,(\tau_t) \qquad \Pi_{\tau'}}{\Gamma\vdash_\tau \tau[x:=t]\tau':\Gamma_0,x:A,\Gamma_1}\,(\tau\tau')}{\Gamma\vdash_l \langle V\|F\rangle\tau[x:=t]\tau'}\,(l)$$

hence we can derive:

$$\dfrac{\dfrac{\dfrac{\Pi_t}{\Gamma,\Gamma_0,\Gamma_1\vdash_t t:A} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma,\Gamma_0,x:A,\Gamma_1 \vdash_V x:A}\,(x)}{\Gamma,\Gamma_0,x:A,\Gamma_1 \vdash_t x:A}\,(\uparrow^t) \qquad \dfrac{\Pi_F}{\Gamma,\Gamma_0,x:A,\Gamma_1 \vdash_e F:A^{\perp\!\!\!\perp}}}{\Gamma,\Gamma_0,x:A,\Gamma_1 \vdash_c \langle x\|F\rangle}\,(c) \qquad \dfrac{\Pi_{\tau'}}{\Gamma,\Gamma_0,x:A\vdash_\tau \tau':\Gamma_1}}{\Gamma,\Gamma_0,x:A\vdash_l \langle x\|F\rangle\tau'}\,(l)}{\Gamma,\Gamma_0\vdash_E \tilde{\mu}[x].\langle x\|F\rangle\tau':A^{\perp\!\!\!\perp}}\,(\tilde{\mu}^{[]})}{\Gamma,\Gamma_0\vdash_e \tilde{\mu}[x].\langle x\|F\rangle\tau':A^{\perp\!\!\!\perp}}\,(\uparrow^e)}{\Gamma,\Gamma_0\vdash_c \langle t\|\tilde{\mu}[x].\langle x\|F\rangle\tau'\rangle}\,(c) \qquad \dfrac{\Pi_\tau}{\Gamma\vdash\tau:\Gamma_0}}{\Gamma\vdash_l \langle t\|\tilde{\mu}[x].\langle x\|F\rangle\tau'\rangle\tau}\,(l)$$

- **Case** $\langle V\|\tilde{\mu}[x].\langle x\|F\rangle\tau'\rangle\tau \to \langle V\|F\rangle\tau[x:=V]\tau'$. A typing derivation of the closure on the left-hand side has the form:

$$\dfrac{\dfrac{\dfrac{\Pi_V}{\Gamma,\Gamma_0,\Gamma_1\vdash_t V:A} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma,\Gamma_0,x:A,\Gamma_1 \vdash_V x:A}\,(x)}{\Gamma,\Gamma_0,x:A,\Gamma_1 \vdash_t x:A}\,(\uparrow^t) \qquad \dfrac{\Pi_F}{\Gamma,\Gamma_0,x:A,\Gamma_1 \vdash_e F:A^{\perp\!\!\!\perp}}}{\Gamma,\Gamma_0,x:A,\Gamma_1 \vdash_c \langle x\|F\rangle}\,(c) \qquad \dfrac{\Pi_{\tau'}}{\Gamma,\Gamma_0,x:A\vdash_\tau \tau':\Gamma_1}}{\Gamma,\Gamma_0,x:A\vdash_l \langle x\|F\rangle\tau'}\,(l)}{\Gamma,\Gamma_0\vdash_E \tilde{\mu}[x].\langle x\|F\rangle\tau':A^{\perp\!\!\!\perp}}\,(\tilde{\mu}^{[]})}{\Gamma,\Gamma_0\vdash_e \tilde{\mu}[x].\langle x\|F\rangle\tau':A^{\perp\!\!\!\perp}}\,(\uparrow^e)}{\Gamma,\Gamma_0\vdash_c \langle V\|\tilde{\mu}[x].\langle x\|F\rangle\tau'\rangle}\,(c) \qquad \dfrac{\Pi_\tau}{\Gamma\vdash\tau:\Gamma_0}}{\Gamma\vdash_l \langle V\|\tilde{\mu}[x].\langle x\|F\rangle\tau'\rangle\tau}\,(l)$$

Therefore we can derive:

$$\dfrac{\dfrac{\dfrac{\Pi_V}{\Gamma,\Gamma_0,x:A,\Gamma_1 \vdash_t V:A} \qquad \dfrac{\Pi_F}{\Gamma,\Gamma_0,x:A,\Gamma_1 \vdash_e F:A^{\perp\!\!\!\perp}}}{\Gamma,\Gamma_0,x:A,\Gamma_1 \vdash_c \langle V\|F\rangle}\,(c) \qquad \dfrac{\dfrac{\dfrac{\Pi_\tau}{\Gamma\vdash\tau:\Gamma_0} \quad \dfrac{\Pi_V}{\Gamma,\Gamma_0\vdash_t V:A}}{\Gamma\vdash_\tau \tau[x:=V]:\Gamma_0,x:A}\,(\tau_t) \qquad \Pi_{\tau'}}{\Gamma\vdash_\tau \tau[x:=V]\tau':\Gamma_0,x:A,\Gamma_1}\,(\tau\tau')}{\Gamma\vdash_l \langle V\|F\rangle\tau[x:=V]\tau'}\,(l)$$

where we implicitly used Lemma 6.1 to weaken Π_V:

$$\dfrac{\dfrac{\Pi_V}{\Gamma,\Gamma_0\vdash_t V:A} \quad \Gamma,\Gamma_0\subseteq\Gamma,\Gamma_0,x:A,\Gamma_1}{\Gamma,\Gamma_0,x:A,\Gamma_1\vdash_t V:A}\,(w)$$

- **Case** $\langle \lambda x.t \| u \cdot E\rangle\tau \rightarrow \langle u \| \tilde\mu x.\langle t \| E\rangle\rangle\tau$. A typing proof for the closure on the left-hand side is of the form:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_t}{\Gamma,\Gamma',x:A\vdash_t t:B}}{\Gamma,\Gamma'\vdash_v \lambda x.t:A\to B}{\scriptstyle(\to_r)}}{\Gamma,\Gamma'\vdash_V \lambda x.t:A\to B}{\scriptstyle(\uparrow^V)}}{\Gamma,\Gamma'\vdash_t \lambda x.t:A\to B}{\scriptstyle(\uparrow^t)} \quad \dfrac{\dfrac{\dfrac{\dfrac{\Pi_u}{\Gamma,\Gamma'\vdash_t u:A} \quad \dfrac{\Pi_E}{\Gamma,\Gamma'\vdash_E E:B^{\bot\!\!\!\bot}}}{\Gamma,\Gamma'\vdash_F u\cdot E:(A\to B)^{\bot\!\!\!\bot}}{\scriptstyle(\to_l)}}{\Gamma,\Gamma'\vdash_E u\cdot E:(A\to B)^{\bot\!\!\!\bot}}{\scriptstyle(\uparrow^E)}}{\Gamma,\Gamma'\vdash_e u\cdot E:(A\to B)^{\bot\!\!\!\bot}}{\scriptstyle(\uparrow^e)}}{\dfrac{\Gamma,\Gamma'\vdash_c \langle\lambda x.t\|u\cdot E\rangle}{}}{\scriptstyle(c)} \qquad$$

$$\dfrac{\Gamma,\Gamma'\vdash_c \langle\lambda x.t\|u\cdot E\rangle \qquad \dfrac{\Pi_\tau}{\Gamma\vdash_\tau \tau:\Gamma'}}{\Gamma\vdash_l \langle\lambda x.t\|u\cdot E\rangle\tau}{\scriptstyle(l)}$$

We can thus build the following derivation:

$$\dfrac{\dfrac{\dfrac{\Pi_u}{\Gamma,\Gamma'\vdash_t u:A} \quad \dfrac{\dfrac{\dfrac{\Pi_t}{\Gamma,\Gamma',x:A\vdash_t t:B} \quad \dfrac{\dfrac{\Pi_E}{\Gamma,\Gamma',x:A\vdash_E E:B^{\bot\!\!\!\bot}}}{\Gamma,\Gamma',x:A\vdash_e E:B^{\bot\!\!\!\bot}}{\scriptstyle(\uparrow^e)}}{\Gamma,\Gamma',x:A\vdash_c \langle t\|E\rangle}{\scriptstyle(c)}}{\Gamma,\Gamma'\vdash_e \tilde\mu x.\langle t\|E\rangle:A^{\bot\!\!\!\bot}}{\scriptstyle(\tilde\mu)}}{\Gamma,\Gamma'\vdash_c \langle u\|\tilde\mu x.\langle t\|E\rangle\rangle}{\scriptstyle(c)} \qquad \dfrac{\Pi_\tau}{\Gamma\vdash_\tau \tau:\Gamma'}}{\Gamma\vdash_l \langle u\|\tilde\mu x.\langle t\|E\rangle\rangle\tau}{\scriptstyle(l)}$$

where we implicitly used Lemma 6.1 to weaken Π_E:

$$\dfrac{\dfrac{\Pi_E}{\Gamma,\Gamma\vdash_E E:B^{\bot\!\!\!\bot}} \quad \Gamma,\Gamma'\subseteq\Gamma,\Gamma',x:A}{\Gamma,\Gamma',x:A\vdash_E E:B^{\bot\!\!\!\bot}}{\scriptstyle(w)}$$

□

6.1.3 Small-step reductions rules

As in the cases of the call-by-name and call-by-value $\lambda\mu\tilde\mu$-calculi (see Sections 4.4 and 4.5), the reduction system can be decomposed into small-step reduction rules. We annotate again commands with the level of syntax we are examining ($c_e, c_t, \dots$), and define a new set of reduction rules which separate computational steps (corresponding to big-step reductions), and administrative steps, which organize the descent in the syntax. In order, a command first put the focus on the context at level e, then on the term at level t, and so on following the hierarchy e, t, E, V, F, v. This results again in an abstract machine in context-free form, since each step only analyzes one component of the command, the "active" term or context, and is parametric in the other "passive" component. In essence, for each phase of the machine, either the term or the context is fully in control and independent, regardless of what the other half happens to be.

We recall the resulting abstract machine from [4] in Figure 6.4. Except for a subtlety of α-conversion that we will explain in Section 6.4.1, these rules directly lead to the definition of the CPS in [4] that we shall type in the next sections. Furthermore, the realizability interpretation *à la* Krivine (that we are about to present in the coming section) is deeply based upon this set of rules. Indeed, remember that a realizer is precisely a term which is going to behave well in front of any opponent in the opposed falsity value. We shall thus take advantage of the context-free rules where at each level, the reduction step is defined independently of the passive component.

$$
\begin{array}{rcl}
\langle t \| \tilde{\mu}x.c\rangle_e \tau & \rightarrow & c_e \tau[x := t] \\
\langle t \| E\rangle_e \tau & \rightarrow & \langle t \| E\rangle_t \tau \\
\\
\langle \mu\alpha.c \| E\rangle_t \tau & \rightarrow & c_e \tau[\alpha := E] \\
\langle V \| E\rangle_t \tau & \rightarrow & \langle V \| E\rangle_E \tau \\
\\
\langle V \| \alpha\rangle_E \tau[\alpha := E]\tau' & \rightarrow & \langle V \| E\rangle_E \tau[\alpha := E]\tau' \\
\langle V \| \tilde{\mu}[x].\langle x \| F\rangle\tau'\rangle_E \tau & \rightarrow & \langle V \| F\rangle_V \tau[x := V]\tau' \\
\langle V \| F\rangle_E \tau & \rightarrow & \langle V \| F\rangle_V \tau \\
\\
\langle x \| F\rangle_V \tau[x := t]\tau' & \rightarrow & \langle t \| \tilde{\mu}[x].\langle x \| F\rangle\tau'\rangle\tau \\
\langle v \| E\rangle_V \tau & \rightarrow & \langle v \| F\rangle_V \tau \\
\\
\langle v \| u \cdot E\rangle_F \tau & \rightarrow & \langle v \| e \cdot E\rangle_v \tau \\
\\
\langle \lambda x.t \| u \cdot E\rangle_v \tau & \rightarrow & \langle u \| \tilde{\mu}x.\langle t \| E\rangle\rangle_e \tau
\end{array}
$$

Figure 6.4: Context-free abstract machine for the $\overline{\lambda}_{[lv\tau\star]}$-calculus

6.2 Realizability interpretation of the simply-typed $\overline{\lambda}_{[lv\tau\star]}$-calculus

6.2.1 Normalization by realizability

The proof of normalization for the $\overline{\lambda}_{[lv\tau\star]}$-calculus that we present in this section is inspired from techniques of Krivine's classical realizability [95], whose notations we borrow. Actually, it is also very close to a proof by reducibility[8]. In a nutshell, to each type A is associated a set $|A|_t$ of terms whose execution is guided by the structure of A. These terms are the ones usually called *realizers* in Krivine's classical realizability. Their definition is in fact indirect, and is done by orthogonality to a set of "correct" computations, called a *pole*. The choice of this set is central when studying models induced by classical realizability for second-order-logic, but in the present case we only pay attention to the particular pole of terminating computations. This is where lies the main difference with a proof by reducibility, where everything is done with respect to SN, while our definition are parametric in the pole (which is chosen to be the set of normalizing closures in the end). The adequacy lemma, which is the central piece, consists in proving that typed terms belong to the corresponding sets of realizers, and are thus normalizing.

More in details, our proof can be sketched as follows. First, we generalize the usual notion of closed term to the notion of closed *term-in-store*. Intuitively, this is due to the fact that we are no longer interested in closed terms and substitutions to close open terms, but rather in terms that are closed when considered in the current store. This is based on the simple observation that a store is nothing more than a shared substitution whose content might evolve along the execution. Second, we define the notion of *pole* $⫫$, which are sets of closures closed by anti-evaluation and store extension. In particular, the set of normalizing closures is a valid pole. This allows us to relate terms and contexts thanks to a notion of orthogonality with respect to the pole. We then define for each formula A and typing level o (of e, t, E, V, F, v) a set $|A|_o$ (resp. $\|A\|_o$) of terms (resp. contexts) in the corresponding syntactic category. These sets correspond to reducibility candidates, or to what is usually called truth values and falsity values in realizability.

Finally, the core of the proof consists in the adequacy lemma, which shows that any closed term of type A at level o is in the corresponding set $|A|_o$. This guarantees that any typed closure is in any pole, and in particular in the pole of normalizing closures. Technically, the proof of adequacy evaluates in each case a state of an abstract machine (in our case a closure), so that the proof also proceeds by

[8] See for instance the proof of normalization for system D presented in [92, 3.2])

evaluation. A more detailed explanation of this observation as well as a more introductory presentation of normalization proofs by classical realizability are given in an article by Dagand and Scherer [35].

6.2.2 Realizability interpretation for the $\overline{\lambda}_{[lv\tau\star]}$-calculus

We begin by defining some key notions for stores that we shall need further in the proof.

Definition 6.3 (Closed store). We extend the notion of free variable to stores:

$$\begin{array}{rcl} FV(\varepsilon) & \triangleq & \emptyset \\ FV(\tau[x := t]) & \triangleq & FV(\tau) \cup \{y \in FV(t) : y \notin \mathrm{dom}(\tau)\} \\ FV(\tau[\alpha := E]) & \triangleq & FV(\tau) \cup \{\beta \in FV(E) : \beta \notin \mathrm{dom}(\tau)\} \end{array}$$

so that we can define a *closed store* to be a store τ such that $FV(\tau) = \emptyset$. ⌟

Definition 6.4 (Compatible stores). We say that two stores τ and τ' are *independent* and note $\tau \# \tau'$ when $\mathrm{dom}(\tau) \cap \mathrm{dom}(\tau') = \emptyset$. We say that they are *compatible* and note $\tau \diamond \tau'$ whenever for all variables x (resp. co-variables α) present in both stores: $x \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau')$; the corresponding terms (resp. contexts) in τ and τ' coincide: formally $\tau = \tau_0[x := t]\tau_1$ and $\tau' = \tau_0'[x := t]\tau_1'$. Finally, we say that τ' is an *extension* of τ and note $\tau \lhd \tau'$ whenever $\mathrm{dom}(\tau) \subseteq \mathrm{dom}(\tau')$ and $\tau \diamond \tau'$. ⌟

Definition 6.5 (Compatible union). We denote by $\overline{\tau\tau'}$ the compatible union $\mathtt{join}(\tau,\tau')$ of closed stores τ and τ', defined by:

$$\begin{array}{rcll} \mathtt{join}(\tau_0[x := t]\tau_1, \tau_0'[x := t]\tau_1') & \triangleq & \tau_0\tau_0'[x := t]\mathtt{join}(\tau_1,\tau_1') & (\text{if } \tau_0\#\tau_0') \\ \mathtt{join}(\tau,\tau') & \triangleq & \tau\tau' & (\text{if } \tau\#\tau') \\ \mathtt{join}(\varepsilon,\tau) & \triangleq & \tau & \\ \mathtt{join}(\tau,\varepsilon) & \triangleq & \tau & \end{array}$$

⌟

The following lemma (which follows easily from the previous definition) states the main property we will use about union of compatible stores.

Lemma 6.6. *If τ and τ' are two compatible stores, then $\tau \lhd \overline{\tau\tau'}$ and $\tau' \lhd \overline{\tau\tau'}$. Besides, if τ is of the form $\tau_0[x := t]\tau_1$, then $\overline{\tau\tau'}$ is of the form $\overline{\tau_0}[x := t]\overline{\tau_1}$ with $\tau_0 \lhd \overline{\tau_0}$ and $\tau_1 \lhd \overline{\tau_1}$.*

As we explained in the introduction of this section, we will not consider closed terms in the usual sense. Indeed, while it is frequent in the proofs of normalization (*e.g.* by realizability or reducibility) of a calculus to consider only closed terms and to perform substitutions to maintain the closure of terms, this only makes sense if it corresponds to the computational behavior of the calculus. For instance, to prove the normalization of $\lambda x.t$ in typed call-by-name $\lambda\mu\tilde{\mu}$-calculus, one would consider a substitution ρ that is suitable for with respect to the typing context Γ, then a context $u \cdot e$ of type $A \to B$, and evaluates :

$$\langle \lambda x.t_\rho \| u \cdot e \rangle \quad \to \quad \langle t_\rho[u/x] \| e \rangle$$

Then we would observe that $t_\rho[u/x] = t_{\rho[x:=u]}$ and deduce that $\rho[x := u]$ is suitable for $\Gamma, x : A$, which would allow us to conclude by induction.

However, in the $\overline{\lambda}_{[lv\tau\star]}$-calculus we do not perform global substitution when reducing a command, but rather add a new binding $[x := u]$ in the store:

$$\langle \lambda x.t \| u \cdot E \rangle\tau \quad \to \quad \langle t \| E \rangle\tau[x := u]$$

Therefore, the natural notion of closed term invokes the closure under a store, which might evolve during the rest of the execution (this is to contrast with a substitution).

Definition 6.7 (Term-in-store)**.** We call *closed term-in-store* (resp. *closed context-in-store*, *closed closures*) the combination of a term t (resp. context e, command c) with a closed store τ such that $FV(t) \subseteq \mathrm{dom}(\tau)$. We use the notation $(t|\tau)$ to denote such a pair. ⌟

We should note that in particular, if t is a closed term, then $(t|\tau)$ is a term-in-store for any closed store τ. The notion of closed term-in-store is thus a generalization of the notion of closed terms, and we will (ab)use of this terminology in the sequel. We denote the sets of closed closures by C_0, and will identify $(c|\tau)$ and the closure $c\tau$ when c is closed in τ. Observe that if $c\tau$ is a closure in C_0 and τ' is a store extending τ, then $c\tau'$ is also in C_0. We are now equipped to define the notion of pole, and verify that the set of normalizing closures is indeed a valid pole.

Definition 6.8 (Pole)**.** A subset $\perp\!\!\!\perp \subseteq C_0$ is said to be *saturated* or *closed by anti-reduction* whenever for all $(c|\tau), (c'|\tau') \in C_0$, if $c'\tau' \in \perp\!\!\!\perp$ and $c\tau \to c'\tau'$ then $c\tau \in \perp\!\!\!\perp$. It is said to be *closed by store extension* if whenever $c\tau \in \perp\!\!\!\perp$, for any store τ' extending τ: $\tau \lhd \tau'$, $c\tau' \in \perp\!\!\!\perp$. A *pole* is defined as any subset of C_0 that is closed by anti-reduction and store extension. ⌟

The following proposition is the one supporting the claim that our realizability proof is almost a reducibility proof whose definitions have been generalized with respect to a pole instead of the fixed set SN.

Proposition 6.9. *The set* $\perp\!\!\!\perp_{\Downarrow} = \{c\tau \in C_0 : \ c\tau \textit{ normalizes } \}$ *is a pole.*

Proof. As we only considered closures in C_0, both conditions (closure by anti-reduction and store extension) are clearly satisfied:

- if $c\tau \to c'\tau'$ and $c'\tau'$ normalizes, then $c\tau$ normalizes too;
- if c is closed in τ and $c\tau$ normalizes, if $\tau \lhd \tau'$ then $c\tau'$ will reduce as $c\tau$ does (since c is closed under τ, it can only use terms in τ' that already were in τ) and thus will normalize. □

Definition 6.10 (Orthogonality)**.** Given a pole $\perp\!\!\!\perp$, we say that a term-in-store $(t|\tau)$ is *orthogonal* to a context-in-store $(e|\tau')$ and write $(t|\tau)\perp\!\!\!\perp(e|\tau')$ if τ and τ' are compatible and $\langle t \| e\rangle\overline{\tau\tau'} \in \perp\!\!\!\perp$. ⌟

Remark 6.11. The reader familiar with Krivine's forcing machine [98] might recognize his definition of orthogonality between terms of the shape (t,p) and stacks of the shape (π,q), where p and q are forcing conditions:

$$(t,p)\perp\!\!\!\perp(\pi,q) \Leftrightarrow (t \star \pi, p \wedge q) \in \perp\!\!\!\perp$$

(The meet of forcing conditions is indeed a refinement containing somewhat the "union" of information contained in each, just like the union of two compatible stores.) ⌟

We can now relate closed terms and contexts by orthogonality with respect to a given pole. This allows us to define for any formula A the sets $|A|_v, |A|_V, |A|_t$ (resp. $\|A\|_F, \|A\|_E$, $\|A\|_e$) of realizers (or reducibility candidates) at level v, V, t (resp F, E, e) for the formula A. It is to be observed that realizers are here closed terms-in-store.

Definition 6.12 (Realizers)**.** Given a fixed pole $\perp\!\!\!\perp$, we set:

$$\begin{array}{rcl}
|X|_v & = & \{(\boldsymbol{k}|\tau) : \quad \vdash \boldsymbol{k} : X\} \\
|A \to B|_v & = & \{(\lambda x.t|\tau) : \forall u\tau', \tau \diamond \tau' \wedge (u|\tau') \in |A|_t \Rightarrow (t|\overline{\tau\tau'}[x := u]) \in |B|_t\} \\
\|A\|_F & = & \{(F|\tau) : \forall v\tau', \tau \diamond \tau' \wedge (v|\tau') \in |A|_v \Rightarrow (v|\tau')\perp\!\!\!\perp(F|\tau)\} \\
|A|_V & = & \{(V|\tau) : \forall F\tau', \tau \diamond \tau' \wedge (F|\tau') \in \|A\|_F \Rightarrow (V|\tau)\perp\!\!\!\perp(F|\tau')\} \\
\|A\|_E & = & \{(E|\tau) : \forall V\tau', \tau \diamond \tau' \wedge (V|\tau') \in |A|_V \Rightarrow (V|\tau')\perp\!\!\!\perp(E|\tau)\} \\
|A|_t & = & \{(t|\tau) : \forall E\tau', \tau \diamond \tau' \wedge (E|\tau') \in \|A\|_E \Rightarrow (t|\tau)\perp\!\!\!\perp(E|\tau')\} \\
\|A\|_e & = & \{(e|\tau) : \forall t\tau', \tau \diamond \tau' \wedge (t|\tau') \in |A|_t \Rightarrow (t|\tau')\perp\!\!\!\perp(e|\tau)\}
\end{array}$$

⌟

Remark 6.13. We draw the reader attention to the fact that we should actually write $|A|_v^{\bot\!\!\!\bot}$, $\|A\|_F^{\bot\!\!\!\bot}$, etc... and $\tau \Vdash_{\bot\!\!\!\bot} \Gamma$, because the corresponding definitions are parameterized by a pole $\bot\!\!\!\bot$. As it is common in Krivine's classical realizability, we ease the notations by removing the annotation $\bot\!\!\!\bot$ whenever there is no ambiguity on the pole. ⌟

If the definition of the different sets might seem complex at first sight, we claim that they are quite natural with regard to the methodology of Danvy's semantics artifacts presented in [4]. Indeed, having an abstract machine in context-free form (the last step in this methodology before deriving the CPS) allows us to have both the term and the context (in a command) that behave independently of each other. Intuitively, a realizer at a given level is precisely a term which is going to behave well (be in the pole) in front of any opponent chosen in the previous level (in the hierarchy v, F, V,etc...). For instance, in a call-by-value setting, there are only three levels of definition (values, contexts and terms) in the interpretation, because the abstract machine in context-free form also has three. Here the ground level corresponds to strong values, and the other levels are somewhat defined as terms (or context) which are well-behaved in front of any opponent in the previous one. The definition of the different sets $|A|_v, \|A\|_F, |A|_V$, etc... directly stems from this intuition.

In comparison with the usual definition of Krivine's classical realizability, we only considered orthogonal sets restricted to some syntactical subcategories. However, the definition still satisfies the usual monotonicity properties of bi-orthogonal sets:

Proposition 6.14. *For any type A and any given pole $\bot\!\!\!\bot$, we have the following inclusions:*

1. $|A|_v \subseteq |A|_V \subseteq |A|_t$;
2. $\|A\|_F \subseteq \|A\|_E \subseteq \|A\|_e$.

Proof. All the inclusions are proved in a similar way. We only give the proof for $|A|_v \subseteq |A|_V$. Let $\bot\!\!\!\bot$ be a pole and $(v|\tau)$ be in $|A|_v$. We want to show that $(v|\tau)$ is in $|A|_V$, that is to say that v is in the syntactic category V (which is true), and that for any $(F|\tau') \in \|A\|_F$ such that $\tau \diamond \tau'$, $(v|\tau)\bot\!\!\!\bot(F|\tau')$. The latter holds by definition of $(F|\tau') \in \|A\|_F$, since $(v|\tau) \in |A|_v$. □

We now extend the notion of realizers to stores, by stating that a store τ realizes a context Γ if it binds all the variables x and α in Γ to a realizer of the corresponding formula.

Definition 6.15. Given a closed store τ and a fixed pole $\bot\!\!\!\bot$, we say that τ *realizes* Γ, which we write[9] $\tau \Vdash \Gamma$, if:

1. for any $(x : A) \in \Gamma$, $\tau \equiv \tau_0[x := t]\tau_1$ and $(t|\tau_0) \in |A|_t$
2. for any $(\alpha : A^{\bot\!\!\!\bot}) \in \Gamma$, $\tau \equiv \tau_0[\alpha := E]\tau_1$ and $(E|\tau_0) \in \|A\|_E$

⌟

In the same way as weakening rules (for the typing context) were admissible for each level of the typing system :

$$\frac{\Gamma \vdash_t t : A \quad \Gamma \subseteq \Gamma'}{\Gamma' \vdash_t t : A} \qquad \frac{\Gamma \vdash_e e : A^{\bot\!\!\!\bot} \quad \Gamma \subseteq \Gamma'}{\Gamma' \vdash_e e : A^{\bot\!\!\!\bot}} \qquad \cdots \qquad \frac{\Gamma \vdash_\tau \tau : \Gamma'' \quad \Gamma \subseteq \Gamma'}{\Gamma' \vdash_\tau \tau : \Gamma''}$$

the definition of realizers is compatible with a weakening of the store.

Lemma 6.16 (Store weakening). *Let τ and τ' be two stores such that $\tau \lhd \tau'$, let Γ be a typing context and let $\bot\!\!\!\bot$ be a pole. The following statements hold:*

1. $\overline{\tau\tau'} = \tau'$

[9] Once again, we should formally write $\tau \Vdash_{\bot\!\!\!\bot} \Gamma$ but we will omit the annotation by $\bot\!\!\!\bot$ as often as possible.

2. *If* $(t|\tau) \in |A|_t$ *for some closed term* $(t|\tau)$ *and type* A*, then* $(t|\tau') \in |A|_t$*. The same holds for each level* e, E, V, F, v *of the typing rules.*
3. *If* $\tau \Vdash \Gamma$ *then* $\tau' \Vdash \Gamma$*.*

Proof. 1. Straightforward from the definitions.

2. This essentially amounts to the following observations. First, one remarks that if $(t|\tau)$ is a closed term, so is $(t|\overline{\tau\tau'})$ for any store τ' compatible with τ. Second, we observe that if we consider for instance a closed context $(E|\tau'') \in \|A\|_E$, then $\overline{\tau\tau'}\diamond\tau''$ implies $\tau\diamond\tau''$, thus $(t|\tau) \perp\!\!\!\perp (E|\tau'')$ and finally $(t|\overline{\tau\tau'}) \perp\!\!\!\perp (E|\tau'')$ by closure of the pole under store extension. We conclude that $(t|\tau') \perp\!\!\!\perp (E|\tau'')$ using the first statement.
3. By definition, for all $(x : A) \in \Gamma$, τ is of the form $\tau_0[x := t]\tau_1$ such that $(t|\tau_0) \in |A|_t$. As τ and τ' are compatible, we know by Lemma 8.16 that $\overline{\tau\tau'}$ is of the form $\tau'_0[x := t]\tau'_1$ with τ'_0 an extension of τ_0, and using the first point we get that $(t|\tau'_0) \in |A|_t$. □

We are now equipped to prove the adequacy of the type system for the $\overline{\lambda}_{[lv\tau\star]}$-calculus with respect to the realizability interpretation.

Definition 6.17 (Adequacy)**.** Given a fixed pole $\perp\!\!\!\perp$, we say that:

- A typing judgment $\Gamma \vdash_t t : A$ is *adequate* (w.r.t. the pole $\perp\!\!\!\perp$) if for all stores $\tau \Vdash \Gamma$, we have $(t|\tau) \in |A|_t$.
- More generally, we say that an inference rule

$$\frac{J_1 \quad \cdots \quad J_n}{J_0}$$

is adequate (w.r.t. the pole $\perp\!\!\!\perp$) if the adequacy of all typing judgments $J_1, \ldots, J_n$ implies the adequacy of the typing judgment J_0.

⌟

Remark 6.18. 1. As usual, it is clear from the latter definition that a typing judgment that is derivable from a set of adequate inference rules is adequate too.

2. The interpretation we gave here relies on the fact that the calculus is simply-typed with constants inhabiting the atomic types. If we were interested in open formulas (or second-order logic), we should as usual (see Section 3.4.4) consider valuation to close formulas, which would map second-order variables to set of strong values. ⌟

Proposition 6.19 (Adequacy)**.** *The typing rules of Figure 6.3 for the* $\overline{\lambda}_{[lv\tau\star]}$*-calculus without co-constants are adequate with any pole. In other words, if* Γ *is a typing context,* $\perp\!\!\!\perp$ *a pole and* τ *a store such that* $\tau \Vdash \Gamma$*, then the following holds:*

1. *If* v *is a strong value such that* $\Gamma \vdash_v v : A$*, then* $(v|\tau) \in |A|_v$*.*
2. *If* F *is a forcing context such that* $\Gamma \vdash_F F : A^{\perp\!\!\!\perp}$*, then* $(F|\tau) \in \|A\|_F$*.*
3. *If* V *is a weak value such that* $\Gamma \vdash_V V : A$*, then* $(V|\tau) \in |A|_V$*.*
4. *If* E *is a catchable context such that* $\Gamma \vdash_E E : A^{\perp\!\!\!\perp}$*, then* $(E|\tau) \in \|A\|_F$*.*
5. *If* t *is a term such that* $\Gamma \vdash_t t : A$*, then* $(t|\tau) \in |A|_t$*.*
6. *If* e *is a context such that* $\Gamma \vdash_e e : A^{\perp\!\!\!\perp}$*, then* $(e|\tau) \in \|A\|_e$*.*
7. *If* c *is a command such that* $\Gamma \vdash_c c$*, then* $c\tau \in \perp\!\!\!\perp$*.*
8. *If* τ' *is a store such that* $\Gamma \vdash_\tau \tau' : \Gamma'$*, then* $\tau\tau' \Vdash \Gamma, \Gamma'$*.*
9. *If* $c\tau'$ *is a closure such that* $\Gamma \vdash_l c\tau'$*, then* $c\tau\tau' \in \perp\!\!\!\perp$*.*

Proof. We proceed by induction over the typing rules.

- **Case** Constants. This case stems directly from the definition of $|X|_v$ for X atomic.

- **Case** $(\to_r)$. This case exactly matches the definition of $|A \to B|_v$. Assume that

$$\frac{\Gamma, x : A \vdash_t t : B}{\Gamma \vdash_v \lambda x.t : A \to B}\ (\to_r)$$

and let $\perp\!\!\!\perp$ be a pole and τ a store such that $\tau \Vdash \Gamma$. If $(u|\tau')$ is a closed term in the set $|A|_t$, then, up to α-conversion for the variable x, $\overline{\tau\tau'} \Vdash \Gamma$ by Lemma 6.16 and $\overline{\tau\tau'}[x := u] \Vdash \Gamma, x : A$. Using the induction hypothesis, $(t|\overline{\tau\tau'}[x := u])$ is indeed in $|B|_t$.

- **Case** $(\to_l)$. Assume that

$$\frac{\Gamma \vdash_t u : A \quad \Gamma \vdash_E E : B^{\perp\!\!\!\perp}}{\Gamma \vdash_F u \cdot E : (A \to B)^{\perp\!\!\!\perp}}\ (\to_l)$$

and let $\perp\!\!\!\perp$ be a pole and τ a store such that $\tau \Vdash \Gamma$. Let $(\lambda x.t|\tau')$ be a closed term in the set $|A \to B|_v$ such that $\tau \diamond \tau'$, then we have:

$$\langle \lambda x.t \| u \cdot E \rangle \overline{\tau\tau'} \quad \to \quad \langle u \| \tilde{\mu} x. \langle t \| E \rangle \rangle \overline{\tau\tau'} \quad \to \quad \langle t \| E \rangle \overline{\tau\tau'}[x := u]$$

By definition of $|A \to B|_v$, this closure is in the pole, and we can conclude by anti-reduction.

- **Case** $(\uparrow^V)$. This case, as well as every other case where typing a term (resp. context) at a higher level of the hierarchy (rules $(\uparrow^E)$, $(\uparrow^t)$, $(\uparrow^e)$), is a simple consequence of Proposition 6.14. Indeed, assume for instance that

$$\frac{\Gamma \vdash_v v : A}{\Gamma \vdash_V v : A}\ (\uparrow^V)$$

and let $\perp\!\!\!\perp$ be a pole and τ a store such that $\tau \Vdash \Gamma$. By induction hypothesis, we get that $(v|\tau) \in |A|_v$. Thus, if $(F|\tau')$ is in $\|A\|_F$, by definition $(v|\tau) \perp\!\!\!\perp (F|\tau')$.

- **Case** (x). Assume that

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash_V x : A}\ (x)$$

and let $\perp\!\!\!\perp$ be a pole and τ a store such that $\tau \Vdash \Gamma$. As $(x : A) \in \Gamma$, we know that τ is of the form $\tau_0[x := t]\tau_1$ with $(t|\tau_0) \in |A|_t$. Let $(F|\tau')$ be in $\|A\|_F$, with $\tau \diamond \tau'$. By Lemma 8.16, we know that $\overline{\tau\tau'}$ is of the form $\overline{\tau_0}[x := t]\overline{\tau_1}$. Hence, we have:

$$\langle x \| F \rangle \overline{\tau_0}[x := t]\overline{\tau_1} \quad \to \quad \langle t \| \tilde{\mu}[x].\langle x \| F \rangle \overline{\tau_1} \rangle \overline{\tau_0}$$

and it suffices by anti-reduction to show that the last closure is in the pole $\perp\!\!\!\perp$. By induction hypothesis, we know that $(t|\tau_0) \in |A|_t$ thus we only need to show that it is in front of a catchable context in $\|A\|_E$. This corresponds exactly to the next case that we shall prove now.

- **Case** $(\tilde{\mu}^{[]})$. Assume that

$$\frac{\Gamma, x : A, \Gamma' \vdash_F F : A \quad \Gamma, x : A \vdash \tau' : \Gamma'}{\Gamma \vdash_E \tilde{\mu}[x].\langle x \| F \rangle \tau' : A}\ (\tilde{\mu}^{[]})$$

and let $\perp\!\!\!\perp$ be a pole and τ a store such that $\tau \Vdash \Gamma$. Let $(V|\tau_0)$ be a closed term in $|A|_V$ such that $\tau_0 \diamond \tau$. We have that :

$$\langle V \| \tilde{\mu}[x].\langle x \| F \rangle \overline{\tau'} \rangle \overline{\tau_0\tau} \quad \to \quad \langle V \| F \rangle \overline{\tau_0\tau}[x := V]\tau'$$

By induction hypothesis, we obtain $\tau[x := V]\tau' \Vdash \Gamma, x : A, \Gamma'$. Up to α-conversion in F and τ', so that the variables in τ' are disjoint from those in τ_0, we have that $\overline{\tau_0\tau} \Vdash \Gamma$ (by Lemma 6.16) and then $\tau'' \triangleq \overline{\tau_0\tau}[x := V]\tau' \Vdash \Gamma, x : A, \Gamma'$. By induction hypothesis again, we obtain that $(F|\tau'') \in \|A\|_F$ (this was an assumption in the previous case) and as $(V|\tau_0) \in |A|_V$, we finally get that $(V|\tau_0) \perp\!\!\!\perp (F|\tau'')$ and conclude again by anti-reduction.

- **Cases** (α). This case is obvious from the definition of $\tau \Vdash \Gamma$.

- **Case** (μ). Assume that

$$\frac{\Gamma, \alpha : A^{\perp\!\!\!\perp} \vdash_c c}{\Gamma \vdash_t \mu\alpha.c : A}\ (\mu)$$

and let $\perp\!\!\!\perp$ be a pole and τ a store such that $\tau \Vdash \Gamma$. Let $(E|\tau')$ be a closed context in $\|A\|_E$ such that $\tau \diamond \tau'$. We have that :

$$\langle \mu\alpha.c \| E \rangle \overline{\tau\tau'} \quad \rightarrow \quad c\overline{\tau\tau'}[\alpha := E]$$

Using the induction hypothesis, we only need to show that $\overline{\tau\tau'}[\alpha := E] \Vdash \Gamma, \alpha : A^{\perp\!\!\!\perp}, \Gamma'$ and conclude by anti-reduction. This obviously holds, since $(E|\tau') \in \|A\|_E$ and $\overline{\tau\tau'} \Vdash \Gamma$ by Lemma 8.16.

- **Case** ($\tilde{\mu}$). This case is identical to the previous one.

- **Case** (c). Assume that

$$\frac{\Gamma \vdash_t t : A \quad \Gamma \vdash_e e : A^{\perp\!\!\!\perp}}{\Gamma \vdash_c \langle t \| e \rangle}\ (c)$$

and let $\perp\!\!\!\perp$ be a pole and τ a store such that $\tau \Vdash \Gamma$. Then by induction hypothesis $(t|\tau) \in |A|_t$ and $(e|\tau) \in \|A\|_e$, so that $\langle t \| e \rangle \tau \in \perp\!\!\!\perp$.

- **Case** (τ_t). This case directly stems from the induction hypothesis which exactly matches the definition of $\tau\tau'[x := t] \Vdash \Gamma, \Gamma', x : A$. The case for the rule (τ_E) is identical, and the case for the rule (ε) is trivial.

- **Case** (l). This case is a direct consequence of induction hypotheses for τ and c. Assume indeed that:

$$\frac{\Gamma, \Gamma' \vdash_c c \quad \Gamma \vdash_\tau \tau' : \Gamma'}{\Gamma \vdash_l c\tau'}\ (l)$$

Then by induction hypotheses $\tau\tau' \Vdash \Gamma, \Gamma'$ and thus $c\tau\tau' \in \perp\!\!\!\perp$.

□

The previous result required to consider the $\overline{\lambda}_{[l\upsilon\tau\star]}$-calculus without co-constants. Indeed, we consider co-constants as coming with their typing rules, potentially giving them any type (whereas constants can only be given an atomic type). Thus, there is *a priori* no reason[10] why their types should be adequate with any pole.

However, as observed in the previous remark, given a fixed pole it suffices to check whether the typing rules for a given co-constant are adequate with this pole. If they are, any judgment that is derivable using these rules will be adequate.

Corollary 6.20. *If $c\tau$ is a closure such that $\vdash_l c\tau$ is derivable, then for any pole $\perp\!\!\!\perp$ such that the typing rules for co-constants used in the derivation are adequate with $\perp\!\!\!\perp$, $c\tau \in \perp\!\!\!\perp$.*

[10] Think for instance of a co-constant of type $(A \rightarrow B)^{\perp\!\!\!\perp}$, there is no reason why it should be orthogonal to any function in $|A \rightarrow B|_\upsilon$.

We can now put our focus back on the normalization of typed closures. As we already saw in Proposition 6.9, the set $⫫_{\Downarrow}$ of normalizing closure is a valid pole, so that it only remains to prove that any typing rule for co-constants is adequate with $⫫_{\Downarrow}$.

Lemma 6.21. *Any typing rule for co-constants is adequate with the pole* $⫫_{\Downarrow}$, i.e. *if* Γ *is a typing context, and* τ *is a store such that* $\tau \Vdash \Gamma$, *if* $\boldsymbol{\kappa}$ *is a co-constant such that* $\Gamma \vdash_F \boldsymbol{\kappa} : A^{⫫}$, *then* $(\boldsymbol{\kappa}|\tau) \in \|A\|_F$.

Proof. This lemma directly stems from the observation that for any store τ and any closed strong value $(v|\tau') \in |A|_v$, $\langle v\|\boldsymbol{\kappa}\rangle\overline{\tau\tau'}$ does not reduce and thus belongs to the pole $⫫_{\Downarrow}$. □

As a consequence, we obtain the normalization of typed closures of the full calculus.

Theorem 6.22. *If* $c\tau$ *is a closure of the* $\overline{\lambda}_{[lv\tau\star]}$*-calculus such that* $\vdash_l c\tau$ *is derivable, then* $c\tau$ *normalizes.*

Besides, the translations[11] from $\overline{\lambda}_{lv}$ to $\overline{\lambda}_{[lv\tau\star]}$ defined by Ariola *et al.* both preserve normalization of commands [4, Theorem 2,4]. As it is clear that they also preserve typing, the previous result also implies the normalization of the $\overline{\lambda}_{lv}$-calculus:

Corollary 6.23. *If* c *is a closure of the* $\overline{\lambda}_{lv}$*-calculus such that* $c : (\,\vdash\,)$ *is derivable, then* c *normalizes.*

This is to be contrasted with Okasaki, Lee and Tarditi's semantics for the call-by-need λ-calculus, which is not normalizing in the simply-typed case, as shown in Ariola *et al* [4].

6.3 A typed store-and-continuation-passing style translation

Guided by the normalization proof of the previous section, we shall now present a type system adapted to the continuation-passing style translation defined in [4]. The computational part is almost the same, except for the fact that we explicitly handle renaming through a substitution σ that replaces names of the source language by names of the target.

6.3.1 Guidelines of the translation

The transformation is actually not only a continuation-passing style translation. Because of the sharing of the evaluation of arguments, the store associating terms to variables has to be passed around. Passing the store amounts to combining the continuation-passing style translation with a store-passing style translation. Additionally, the store is extensible, so, to anticipate extension of the store, Kripke style forcing has to be used too, in a way comparable to what is done in step-indexing translations. Before presenting in detail the target system of the translation, let us explain step by step the rationale guiding the definition of the translation. To facilitate the comprehension of the different steps, we illustrate each of them with the translation of the sequent $a : A, \alpha : A^{⫫}, b : B \vdash_e e : C$.

Step 1 - Continuation-passing style. In a first approximation, let us look only at the continuation-passing style part of the translation of a $\overline{\lambda}_{[lv\tau\star]}$ sequent.

As shown in [4] and as emphasized by the definition of realizers (see Definition 6.12) reflecting the 6 nested syntactic categories used to define $\overline{\lambda}_{[lv\tau\star]}$, there are 6 different levels of control in call-by-need, leading to 6 mutually defined levels of interpretation. We define $[\![A \to B]\!]_v$ for strong values as $[\![A]\!]_t \to [\![B]\!]_E$, we define $[\![A]\!]_F$ for forcing contexts as $\neg\, [\![A]\!]_v$, $[\![A]\!]_V$ for weak values as $\neg\, [\![A]\!]_F = \overset{2}{\neg}\, [\![A]\!]_v$, and so on until $[\![A]\!]_e$ defined as $\overset{5}{\neg}\, [\![A]\!]_v$ (where $\overset{0}{\neg}\, A \triangleq A$ and $\overset{n+1}{\neg}\, A \triangleq \neg\, \overset{n}{\neg}\, A$).

As we already observed in the previous section (see Definition 8.18), hypotheses from a context Γ of the form $\alpha : A^{⫫}$ are to be translated as $[\![A]\!]_E = \overset{3}{\neg}\, [\![A]\!]_v$ while hypotheses of the form $x : A$ are to be

[11] There is actually an intermediate step to a calculus named $\bar{\lambda}_{[l\tau v]}$.

translated as $[\![A]\!]_t \overset{4}{=\neg} [\![A]\!]_v$. Up to this point, if we denote this translation of Γ by $[\![\Gamma]\!]$, in the particular case of $\Gamma \vdash_t A$ the translation is $[\![\Gamma]\!] \vdash [\![A]\!]_t$ and similarly for other levels, *e.g.* $\Gamma \vdash_e A$ translates to $[\![\Gamma]\!] \vdash [\![A]\!]_e$.

Example 6.24 (Translation, step 1)**.** Up to now, the translation taking into account the continuation-passing style of $a : A, \alpha : A^{\perp\!\!\!\perp}, b : B \vdash_e e : C$ is simply:

$$\begin{aligned}[\![a : A, \alpha : A^{\perp\!\!\!\perp}, b : B \vdash_e e : C]\!] &= a : [\![A]\!]_t \quad , \alpha : [\![A]\!]_E \quad , b : [\![B]\!]_t \quad \vdash [\![e]\!]_e : [\![C]\!]_e \\ &= a \overset{4}{:\neg} [\![A]\!]_v , \alpha \overset{3}{:\neg} [\![A]\!]_v , b \overset{4}{:\neg} [\![B]\!]_v \vdash [\![e]\!]_e \overset{5}{:\neg} [\![C]\!]_v\end{aligned}$$

⌟

Step 2 - Store-passing style. The continuation-passing style part being settled, the store-passing style part should be considered. In particular, the translation of $\Gamma \vdash_t A$ is not anymore a sequent $[\![\Gamma]\!] \vdash [\![A]\!]_t$ but instead a sequent roughly of the form $\vdash [\![\Gamma]\!] \to [\![A]\!]_t$, with actually $[\![\Gamma]\!]$ being passed around not only at the top-level of $[\![A]\!]_t$ but also every time a negation is used. We write this sequent $\vdash [\![\Gamma]\!] \triangleright_t A$ where $\triangleright_t A$ is defined by induction on t and A, with

$$\begin{aligned}[\![\Gamma]\!] \triangleright_t A &= [\![\Gamma]\!] \to ([\![\Gamma]\!] \triangleright_E A) \to \bot \\ &= [\![\Gamma]\!] \to ([\![\Gamma]\!] \to ([\![\Gamma]\!] \triangleright_V A) \to \bot) \to \bot = \ldots\end{aligned}$$

Moreover, the translation of each type in Γ should itself be abstracted over the store at each use of a negation.

Example 6.25 (Translation, step 2)**.** Up to now, the continuation-and-store passing style translation of $a : A, \alpha : A^{\perp\!\!\!\perp}, b : B \vdash_e e : C$ is:

$$\begin{aligned}[\![a : A, \alpha : A^{\perp\!\!\!\perp}, b : B \vdash_e e : C]\!] = \quad &\vdash [\![e]\!]_e : [\![a : A, \alpha : A^{\perp\!\!\!\perp}, b : B]\!] \triangleright_e C \\ = \quad &\vdash [\![e]\!]_e : [\![a : A, \alpha : A^{\perp\!\!\!\perp}, b : B]\!] \to ([\![a : A, \alpha : A^{\perp\!\!\!\perp}, b : B]\!] \triangleright_t C) \to \bot = \ldots\end{aligned}$$

where:

$$\begin{aligned}[\![a : A, \alpha : A^{\perp\!\!\!\perp}, b : B]\!] &= [\![a : A, \alpha : A^{\perp\!\!\!\perp}]\!],\ b : [\![a : A, \alpha : A^{\perp\!\!\!\perp}]\!] \triangleright_t B \\ &= [\![a : A, \alpha : A^{\perp\!\!\!\perp}]\!],\ b : [\![a : A, \alpha : A^{\perp\!\!\!\perp}]\!] \to ([\![a : A, \alpha : A^{\perp\!\!\!\perp}]\!] \triangleright_E B) \to \bot = \ldots \\ [\![a : A, \alpha : A^{\perp\!\!\!\perp}]\!] &= [\![a : A]\!],\ \alpha : [\![a : A]\!] \triangleright_E A \\ &= [\![a : A]\!],\ \alpha : [\![a : A]\!] \to ([\![a : A]\!] \to \triangleright_E A) \to \bot = \ldots \\ [\![a : A]\!] &= a : \varepsilon \triangleright_t A \ = \ a \overset{4}{:\neg} [\![A]\!]_v\end{aligned}$$

⌟

Step 3 - Extension of the store. The store-passing style part being settled, it remains to anticipate that the store is extensible. This is done by supporting arbitrary insertions of any term at any place in the store. The extensibility is obtained by quantifying over all possible extensions of the store at each level of the negation. This corresponds to the intuition that in the realizability interpretation, given a sequent $\Gamma \vdash_t t : A$ we showed that for any store τ such that $\tau \Vdash \Gamma$, we had $(t|\tau)$ in $|A|_t$. But the definition of $\tau \Vdash \Gamma$ is such that for any $\Gamma' \supseteq \Gamma$, if $\tau \Vdash \Gamma'$ then $\tau \Vdash \Gamma$, so that actually $(t|\tau')$ is also $|A|_t$. The term t was thus compatible with any extension of the store.

For this purpose, we use as a type system an adaptation of System $F_{<:}$ [22] extended with stores, defined as lists of assignations $[x := t]$. *Store types*, denoted by Υ, are defined as list of types of the form $(x : A)$ where x is a name and A is a type properly speaking and admit a subtyping notion $\Upsilon' <: \Upsilon$ to express that Υ' is an extension of Υ. This corresponds to the following refinement of the definition of $[\![\Gamma]\!] \triangleright_t A$:

$$\begin{aligned}[\![\Gamma]\!] \triangleright_t A &= \forall \Upsilon <: [\![\Gamma]\!].\Upsilon \to (\Upsilon \triangleright_E A) \to \bot \\ &= \forall \Upsilon <: [\![\Gamma]\!].\Upsilon \to (\forall \Upsilon' <: \Upsilon.\Upsilon' \to \Upsilon' \triangleright_V A \to \bot) \to \bot = \ldots\end{aligned}$$

The reader can think of subtyping as a sort of Kripke forcing [89], where *worlds* are store types Υ and *accessible worlds* from Υ are precisely all the possible $\Upsilon' <: \Upsilon$.

Example 6.26 (Translation, step 3)**.** The translation, now taking into account store extensions, of $a : A, \alpha : A^{\perp\!\!\!\perp}, b : B \vdash_e e : C$ becomes:

$$\begin{array}{rcl}
\llbracket a : A, \alpha : A^{\perp\!\!\!\perp}, b : B \vdash_e e : C \rrbracket & = & \vdash \llbracket e \rrbracket_e : \llbracket a : A, \alpha : A^{\perp\!\!\!\perp}, b : B \rrbracket \triangleright_e C \\
& = & \vdash \llbracket e \rrbracket_e : \forall \Upsilon <: \llbracket a : A, \alpha : A^{\perp\!\!\!\perp}, b : B \rrbracket . \Upsilon \to (\Upsilon \triangleright_t C) \to \bot = \ldots
\end{array}$$

where:

$$\begin{array}{rcl}
\llbracket a : A, \alpha : A^{\perp\!\!\!\perp}, b : B \rrbracket & = & \llbracket a : A, \alpha : A^{\perp\!\!\!\perp} \rrbracket,\ b : \llbracket a : A, \alpha : A^{\perp\!\!\!\perp} \rrbracket \triangleright_t B \\
& = & \llbracket a : A, \alpha : A^{\perp\!\!\!\perp} \rrbracket,\ b : \forall \Upsilon <: \llbracket a : A, \alpha : A^{\perp\!\!\!\perp} \rrbracket . \Upsilon \to (\Upsilon \triangleright_E B) \to \bot = \ldots \\
\llbracket a : A, \alpha : A^{\perp\!\!\!\perp} \rrbracket & = & \llbracket a : A \rrbracket,\ \alpha : \llbracket a : A \rrbracket \triangleright_E A \\
& = & \llbracket a : A \rrbracket,\ \alpha : \forall \Upsilon <: \llbracket a : A \rrbracket . \Upsilon \to (\Upsilon \to \triangleright_E A) \to \bot = \ldots \\
\llbracket a : A \rrbracket & = & a : \varepsilon \triangleright_t A \ = \ a : \forall \Upsilon . \Upsilon \to (\Upsilon \triangleright_E A) \to \bot
\end{array}$$

┘

Step 4 - Explicit renaming As we will explain in details in the next section (see Section 6.4.1), we need to handle the problem of renaming the variables during the translation. We assume that we dispose of a generator of fresh names (in the target language). In practice, this means that the implementation of the CPS requires for instance to have a list keeping tracks of the variables already used. In the case where variable names can be reduced to natural numbers, this can be easily done with a reference that is incremented each time a fresh variable is needed. The translation is thus annotated by a substitution σ which binds names from the source language with names in the target language. For instance, the translation of a typing context $a : A, \alpha : A^{\perp\!\!\!\perp}, b : B$ is now:

$$\llbracket a : A, \alpha : A^{\perp\!\!\!\perp}, b : B \rrbracket^\sigma \quad = \quad \sigma(a) : \varepsilon \triangleright_t A,\ \sigma(\alpha) : \llbracket a : A \rrbracket^\sigma \triangleright_E A,\ \sigma(b) : \llbracket a : A, \alpha : A^{\perp\!\!\!\perp} \rrbracket^\sigma \triangleright_t B$$

6.3.2 The target language: System $\mathbf{F}_\Upsilon$

The target language is thus the usual λ-calculus, which is extended with stores (defined lists of pairs of a name and a term) and second-order quantification over store types. We refer to this language as System F_Υ. We assume that types contain at least a constant for each atomic type X of the original system, and we still denote this constant by X. This allows us to define an embedding ι from the original type system to this one by:

$$\iota(X) = X \qquad\qquad \iota(A \to B) = \iota(A) \to \iota(B).$$

The syntax for terms and types is given by:

$$\begin{array}{rl|rl}
t, u ::= & \boldsymbol{k} \mid x \mid \lambda x.t \mid tu \mid \tau & A, B ::= & X \mid \bot \mid \Upsilon \triangleright_\tau \Upsilon' \mid A \to B \mid \forall Y \diamond \Upsilon . A \\
& \mid \mathtt{let}\, x_{\tau_0}, x, x_{\tau_1} = \mathtt{split}\ \tau''\, y\, \mathtt{in}\, t & \Upsilon, \Upsilon' ::= & \varepsilon \mid (x : A) \mid (x : A^{\perp\!\!\!\perp}) \mid Y \mid \Upsilon, \Upsilon' \\
\tau, \tau' ::= & \varepsilon \mid \tau[x := t] & \Gamma, \Gamma' ::= & \varepsilon \mid \Gamma, x : A \mid \Gamma, Y <: \Upsilon
\end{array}$$

We introduce a new symbol $\Upsilon \triangleright_\tau \Upsilon'$ to denote the fact that a store has a type conditioned by Υ (which should be the type of the head of the list). In order to ease the notations, we will denote Υ instead of $\varepsilon \triangleright_\tau \Upsilon$ in the sequel. On the contrary, $\Upsilon \triangleright_t A$ is a shorthand (defined in Figure 6.6). The type system is given in Figure 6.5 where we assume that a name can only occur once both in typing contexts Γ and stores types Υ.

Remark 6.27. We shall make a few remarks about our choice of rules for typing stores. First, observe that we force elements of the store to have types of the form $\Upsilon \triangleright_t A$, that is having the structure of types obtained through the CPS translation. Even though this could appear as a strong requirement, it appears naturally when giving a computational contents to the inclusion $\Upsilon <: '\Upsilon$ with De Bruijn levels (see Section 6.4.4). Indeed, a De Bruijn level (just as a name) can be understood as a pointer to a

$$\dfrac{(\boldsymbol{k} : X) \in \mathcal{S}}{\Gamma \vdash \boldsymbol{k} : X}\,(c) \qquad \dfrac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\,(\mathrm{Ax}) \qquad \dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\,(\lambda) \qquad \dfrac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B}\,(@)$$

$$\dfrac{\Gamma, Y <: \Upsilon \vdash t : A \quad Y \notin FV(\Gamma)}{\Gamma \vdash t : \forall Y <: \Upsilon.A}\,(\forall_I) \qquad \dfrac{\Gamma \vdash t : \forall Y <: \Upsilon.A \quad \Gamma \vdash \Upsilon' <: \Upsilon}{\Gamma \vdash t : A\{Y := \Upsilon'\}}\,(\forall_E)$$

$$\dfrac{\Gamma, x_{\tau_0} : \Upsilon_0, x : \Upsilon_0 \triangleright_t A, x_{\tau_1} : (\Upsilon_0, y : A) \triangleright_\tau \Upsilon_1 \vdash t : B \quad \Gamma \vdash \tau : \Upsilon_0, y : A, \Upsilon_1}{\Gamma; \Sigma \vdash \texttt{let}\, x_{\tau_0}, x, x_{\tau_1} = \texttt{split}\ \ \texttt{as}\,(\tau)\ \texttt{in}\ y\,\texttt{in}\, t : B}\,(\texttt{split})$$

$$\dfrac{}{\Gamma \vdash \varepsilon : \varepsilon \triangleright_\tau \varepsilon}\,(\varepsilon) \qquad \dfrac{\Gamma \vdash t : \Upsilon_0 \triangleright_t A}{\Gamma \vdash [x := t] : \Upsilon_0 \triangleright_\tau x : A}\,(\tau_t) \qquad \dfrac{\Gamma \vdash t : \Upsilon_0 \triangleright_E A}{\Gamma \vdash [x := t] : \Upsilon_0 \triangleright_\tau x : A^{\perp\!\!\perp}}\,(\tau_E)$$

$$\dfrac{\Gamma \vdash \tau : \Upsilon_0 \triangleright_\tau \Upsilon \qquad \Gamma \vdash \tau' : (\Upsilon_0, \Upsilon) \triangleright_\tau \Upsilon'}{\Gamma \vdash \tau\tau' : \Upsilon_0 \triangleright_\tau \Upsilon, \Upsilon'}\,(\tau\tau') \qquad \dfrac{(\Upsilon' <: \Upsilon) \in \Gamma}{\Gamma \vdash \Upsilon' <: \Upsilon}\,(<:_{\mathrm{ax}}) \qquad \dfrac{}{\Gamma \vdash Y <: Y}\,(<:_Y)$$

$$\dfrac{}{\Gamma \vdash \Upsilon <: \varepsilon}\,(<:_\varepsilon) \qquad \dfrac{\Gamma \vdash \Upsilon' <: \Upsilon}{\Gamma \vdash (\Upsilon', x : A) <: (\Upsilon, x : A)}\,(<:_1) \qquad \dfrac{\Gamma \vdash \Upsilon' <: \Upsilon}{\Gamma \vdash \Upsilon', \Upsilon'' <: \Upsilon}\,(<:_2)$$

$$\dfrac{\Gamma \vdash \Upsilon'' <: \Upsilon' \quad \Gamma \vdash \Upsilon' <: \Upsilon}{\Gamma \vdash \Upsilon'' <: \Upsilon}\,(<:_3) \qquad \dfrac{\Gamma \vdash \tau : \Upsilon_0' \triangleright_\tau \Upsilon' \quad \Gamma \vdash \Upsilon' <: \Upsilon \quad \Gamma \vdash \Upsilon_0 <: \Upsilon_0'}{\Gamma \vdash \tau : \Upsilon_0 \triangleright_\tau \Upsilon}\,(\tau_{<:})$$

$$\dfrac{\Gamma[(Y_0, x : A, Y_1)/Y] \vdash t : B[(Y_0, x : A, Y_1)/Y] \quad \Gamma \vdash Y <: (\Upsilon_0, x : A, \Upsilon_1)}{\Gamma \vdash t : B}\,(<:_{\texttt{split}})$$

Figure 6.5: Typing rules of System F_Υ

particular cell of the store. Therefore, we need to update pointers when inserting a new element (as in Proposition 6.32). Such an operation would not have any sense (and in particular would be ill-typed) for an element that is not of type $\Upsilon \triangleright_t A$. One could circumvent this by tagging each cell of the store with a flag (using a sum type) indicating whether the corresponding elements have a type of this form or not. Second, note that each element of the store has a type depending on the type of the head of the store. Once again, this is natural and only reflects what was already happening in the source language or within the realizability interpretation. ⌟

The translation of judgments and types is given in Figure 6.6, where we made explicit the renaming procedure from the $\overline{\lambda}_{[lv\tau\star]}$-calculus to the target language. We denote by $\sigma \fatsemi \Gamma$ the fact that σ is a substitution suitable to rename every names present in Γ.

As for the reduction rules of the language, there is only two of them, namely the usual β-reduction and the split of a store with respect to a name:

$$\begin{array}{rcll} \lambda x.t\,u & \to & t[u/x] & \\ \texttt{let}\,x_0, x, x_1 = \texttt{split}\ \tau\, y\,\texttt{in}\, t & \to & t[\tau_0/x_0, u/x, \tau_1/x_1] & \qquad\text{(where } \tau = \tau_0[y := u]\tau_1) \end{array}$$

6.3.3 The typed translation

We consider in this section that we dispose of a generator of fresh names (for instance a global counter) and use names explicitly both in the language (for stores) and in the type system (for their types). The

$$\begin{array}{llllll}
\llbracket \Gamma \vdash_e e : A^{\perp\!\!\!\perp} \rrbracket & \triangleq & \forall \sigma,\ \sigma \mathbin{\mathfrak{s}} \Gamma & \Rightarrow & (\vdash\ \llbracket e \rrbracket_e^\sigma : \llbracket \Gamma \rrbracket_\Gamma^\sigma \triangleright_e \iota(A)) & \\
\llbracket \Gamma \vdash_t t : A \rrbracket & \triangleq & \forall \sigma,\ \sigma \mathbin{\mathfrak{s}} \Gamma & \Rightarrow & (\vdash\ \llbracket t \rrbracket_t^\sigma : \llbracket \Gamma \rrbracket_\Gamma^\sigma \triangleright_t \iota(A)) & \\
\llbracket \Gamma \vdash_E E : A^{\perp\!\!\!\perp} \rrbracket & \triangleq & \forall \sigma,\ \sigma \mathbin{\mathfrak{s}} \Gamma & \Rightarrow & (\vdash\ \llbracket E \rrbracket_E^\sigma : \llbracket \Gamma \rrbracket_\Gamma^\sigma \triangleright_E \iota(A)) & \\
\llbracket \Gamma \vdash_V V : A \rrbracket & \triangleq & \forall \sigma,\ \sigma \mathbin{\mathfrak{s}} \Gamma & \Rightarrow & (\vdash\ \llbracket V \rrbracket_V^\sigma : \llbracket \Gamma \rrbracket_\Gamma^\sigma \triangleright_V \iota(A)) & \\
\llbracket \Gamma \vdash_F F : A^{\perp\!\!\!\perp} \rrbracket & \triangleq & \forall \sigma,\ \sigma \mathbin{\mathfrak{s}} \Gamma & \Rightarrow & (\vdash\ \llbracket F \rrbracket_F^\sigma : \llbracket \Gamma \rrbracket_\Gamma^\sigma \triangleright_F \iota(A)) & \\
\llbracket \Gamma \vdash_v v : A \rrbracket & \triangleq & \forall \sigma,\ \sigma \mathbin{\mathfrak{s}} \Gamma & \Rightarrow & (\vdash\ \llbracket v \rrbracket_v^\sigma : \llbracket \Gamma \rrbracket_\Gamma^\sigma \triangleright_v \iota(A)) & \\
\llbracket \Gamma \vdash_c c \rrbracket & \triangleq & \forall \sigma,\ \sigma \mathbin{\mathfrak{s}} \Gamma & \Rightarrow & (\vdash\ \llbracket c \rrbracket_c^\sigma : \llbracket \Gamma \rrbracket_\Gamma^\sigma \triangleright_c \bot) & \\
\llbracket \Gamma \vdash_l l \rrbracket & \triangleq & \forall \sigma,\ \sigma \mathbin{\mathfrak{s}} \Gamma & \Rightarrow & (\vdash\ \llbracket l \rrbracket_l^\sigma : \llbracket \Gamma \rrbracket_\Gamma^\sigma \triangleright_c \bot) & \\
\llbracket \Gamma \vdash_\tau \tau : \Gamma' \rrbracket & \triangleq & \forall \sigma,\ \sigma \mathbin{\mathfrak{s}} \Gamma & \Rightarrow & (\vdash\ \tau' : \llbracket \Gamma \rrbracket_\Gamma^{\sigma'} \triangleright_\tau \llbracket \Gamma' \rrbracket_\Gamma^{\sigma'}) & (\text{where } \tau',\sigma' = \llbracket \tau \rrbracket_\tau^\sigma)
\end{array}$$

$$\sigma \mathbin{\mathfrak{s}} \Gamma \triangleq \sigma \text{ injective } \wedge \mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\sigma)$$

$$\llbracket \Gamma, a : A \rrbracket_\Gamma^\sigma \triangleq \llbracket \Gamma \rrbracket_\Gamma^\sigma, \sigma(a) : \iota(A) \qquad \llbracket \Gamma, \alpha : A^{\perp\!\!\!\perp} \rrbracket_\Gamma^\sigma \triangleq \llbracket \Gamma \rrbracket_\Gamma^\sigma, \sigma(\alpha) : \iota(A)^{\perp\!\!\!\perp} \qquad \llbracket \varepsilon \rrbracket_\Gamma^\sigma \triangleq \varepsilon$$

$$\begin{array}{l|ll}
\Upsilon \triangleright_c A \triangleq \forall Y <: \Upsilon.\, Y \to \bot & \Upsilon \triangleright_V A & \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_F A) \to \bot \\
\Upsilon \triangleright_e A \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_t A) \to \bot & \Upsilon \triangleright_F A & \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_v A) \to \bot \\
\Upsilon \triangleright_t A \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_E A) \to \bot & \Upsilon \triangleright_v A \to B & \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_t A) \to (Y \triangleright_E B) \to \bot \\
\Upsilon \triangleright_E A \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_V A) \to \bot & \Upsilon \triangleright_v X & \triangleq X
\end{array}$$

Figure 6.6: Translation of judgments and types

next section will be devoted to the presentation of the translation using De Bruijn levels instead of names.

The translation of terms is given in Figure 6.7 where we assume that for each constant $\boldsymbol{k}$ of type X (resp. co-constant $\boldsymbol{\kappa}$ of type $A^{\perp\!\!\!\perp}$) of the source system, we have a constant of type X in the signature $\mathcal{S}$ of target language, constant that we also denote by $\boldsymbol{k}$ (resp. $\boldsymbol{\kappa}$ of type $A \to \bot$). Except for the explicit renaming, the translation is the very same as in Ariola *et al.*, hence their results are preserved with our translation. In particular, if two closures l, l' are such that $l \to l'$, then[12] $\llbracket l \rrbracket_l^\sigma =_{\beta,\eta} \llbracket l' \rrbracket_l^\sigma$ (see [4, Theorem 6]).

We first prove a few technical results that we will use afterwards in the proof of the main theorem.

Lemma 6.28 (Suitable substitution)**.** *For all σ and Γ such that σ is suitable for Γ, if τ is a store such that $\Gamma \vdash_\tau \tau : \Gamma'$ for some Γ', if $\tau', \sigma' = \llbracket \tau \rrbracket_\tau^\sigma$ then σ' is suitable for Γ, Γ' and $\llbracket \Gamma \rrbracket_\Gamma^\sigma = \llbracket \Gamma \rrbracket_\Gamma^{\sigma'}$.*

Proof. Obvious from the definition. □

Lemma 6.29 (Subtyping identity)**.** *The following rule is admissible:* $\dfrac{}{\Sigma \vdash \Upsilon <: \Upsilon}$

Proof. Straightforward induction on the structure of Υ, applying repeatedly the $(<:_1)$-rule (or the $(<:_Y)$-rule). □

[12] Such a statement could be refined to prove that that the translation preserves the reduction. As in the call-by-name and call-by-value cases (see Proposition 4.18), it would require to define a translation at each level ($e, t, \ldots$) for commands, to finally prove that if $c_\iota \tau \xrightarrow{1} c_o \tau'$, then $\llbracket c\tau \rrbracket_\iota^\sigma \xrightarrow{+} \llbracket c\tau' \rrbracket_o^{\sigma'}$. We claim that this would not present any specific difficulty, but that it is no longer worth bothering ourself with such a proof since we already proved the normalization.

$$
\begin{array}{lcll}
\llbracket \boldsymbol{k} \rrbracket_v^\sigma & \triangleq & \boldsymbol{k} & \\
\llbracket \lambda x.t \rrbracket_v^\sigma\ \tau\, u\, E & \triangleq & \llbracket t \rrbracket_t^{\sigma[x:=n]}\ \ \tau[n := u]\ E & (n \text{ fresh}) \\
\\
\llbracket \boldsymbol{\kappa} \rrbracket_F^\sigma & \triangleq & \boldsymbol{\kappa} & \\
\llbracket t \cdot E \rrbracket_F^\sigma\ \tau\ v & \triangleq & v\ \tau\ \llbracket t \rrbracket_t^\sigma\ \llbracket E \rrbracket_E^\sigma & \\
\\
\llbracket v \rrbracket_V^\sigma\ \tau\ F & \triangleq & F\ \tau\ \llbracket v \rrbracket_v^\sigma & \\
\llbracket x \rrbracket_V^\sigma\ \tau[\sigma(x) := t]\tau'\ F & \triangleq & t\ \tau\ (\lambda\tau\lambda V.V\ \tau[\sigma(x) :=\uparrow^t V]\tau'\ F) & (\text{with } \uparrow^t V = \lambda\tau E.E\,\tau\, V) \\
\\
\llbracket \alpha \rrbracket_E^\sigma\ \tau[\sigma(\alpha) := E]\tau'\ V & \triangleq & E\ \tau[\sigma(\alpha) := E]\tau'\ V & \\
\llbracket \tilde{\mu}[x].\langle x \| F\rangle\tau' \rrbracket_E^\sigma\ \tau\ V & \triangleq & V\ \tau[n :=\uparrow^t V]\tau''\ \llbracket F \rrbracket_F^{\sigma'} & (\text{where } n \text{ fresh}, \tau'',\sigma' = \llbracket \tau \rrbracket_\tau^{\sigma[x:=n]}) \\
\\
\llbracket V \rrbracket_t^\sigma\ \tau\ E & \triangleq & E\ \tau\ \llbracket V \rrbracket_V^\sigma & \\
\llbracket \mu\alpha.c \rrbracket_t^\sigma\ \tau\ E & \triangleq & \llbracket c \rrbracket_c^{\sigma[\alpha:=n]}\ \tau[n := E] & (n \text{ fresh}) \\
\\
\llbracket E \rrbracket_e^\sigma\ \tau\ t & \triangleq & t\ \tau\ \llbracket E \rrbracket_E^\sigma & \\
\llbracket \tilde{\mu}x.c \rrbracket_e^\sigma\ \tau\ t & \triangleq & \llbracket c \rrbracket_c^{\sigma[x:=n]}\ \tau[n := t] & (n \text{ fresh}) \\
\\
\llbracket \langle t \| e\rangle \rrbracket_c^\sigma\ \tau & \triangleq & \llbracket e \rrbracket_e^\sigma\ \tau\ \llbracket t \rrbracket_t^\sigma & \\
\llbracket c\ \tau \rrbracket_l^\sigma\ \tau_0 & \triangleq & \llbracket c \rrbracket_c^{\sigma'}\ \tau_0\tau' & (\text{where } \tau',\sigma' = \llbracket \tau \rrbracket_\tau^\sigma) \\
\\
\llbracket \varepsilon \rrbracket_\tau^\sigma & \triangleq & \varepsilon,\sigma & \\
\llbracket \tau'[x := t] \rrbracket_\tau^\sigma & \triangleq & \tau'[n := \llbracket t \rrbracket_t^{\sigma'}],\sigma[x := n] & (\text{where } \tau',\sigma' = \llbracket \tau \rrbracket_\tau^\sigma,\ n \text{ fresh}) \\
\llbracket \tau'[\alpha := E] \rrbracket_\tau^\sigma & \triangleq & \tau'[n := \llbracket E \rrbracket_E^{\sigma'}],\sigma[\alpha := n] & (\text{where } \tau',\sigma' = \llbracket \tau \rrbracket_\tau^\sigma,\ n \text{ fresh})
\end{array}
$$

Figure 6.7: Translation of terms

Lemma 6.30 (Weakening)**.** *The following rule is admissible:*

$$\frac{\Gamma \vdash t : A \quad \Gamma \subseteq \Gamma'}{\Gamma' \vdash t : A}\ (w)$$

Proof. Straightforward induction on typing derivations. □

Lemma 6.31 (Terms subtyping)**.** *The following rule is admissible:*

$$\frac{\Gamma \vdash t : \forall Y <: \Upsilon_0.A \quad \Gamma \vdash \Upsilon_1 <: \Upsilon_0}{\Gamma \vdash t : \forall Y <: \Upsilon_1.A} <:_\forall$$

Proof. We can derive:

$$\dfrac{\dfrac{\Gamma, X <: \Upsilon_1 \vdash t : \forall Y <: \Upsilon_0.A \qquad \dfrac{\dfrac{}{\Gamma, Y <: \Upsilon_1 \vdash Y <: \Upsilon_1}\,(<:_{\mathsf{ax}}) \qquad \Gamma \vdash \Upsilon_1 <: \Upsilon_0}{\Gamma, Y <: \Upsilon_1 \vdash Y <: \Upsilon_0}\,(<:_3)}{\Gamma, Y <: \Upsilon_1 \vdash t : A}\,(\forall_E) \qquad Y \notin FV(\Gamma)}{\Gamma \vdash t : \forall Y <: \Upsilon_1.A}\,(\forall_I)$$

where we use Lemma 6.30 to weaken $\Gamma, X <: \Upsilon_1$ to Γ.

□

Corollary 6.32. *For any level o of the hierarchy e,t,E,V,F,v, the following rule is admissible:*

$$\frac{\Gamma \vdash t : \Upsilon_0 \triangleright_o A \quad \Gamma \vdash \Upsilon_1 <: \Upsilon_0}{\Gamma \vdash t : \Upsilon_1 \triangleright_o A} <:_\triangleright$$

Theorem 6.33 (Preservation of typing)**.** *The translation is well-typed, i.e.*

1. *if* $\Gamma \vdash_v v : A$ *then* $\llbracket \Gamma \vdash_v v : A \rrbracket$
2. *if* $\Gamma \vdash_F F : A^{\perp\!\!\!\perp}$ *then* $\llbracket \Gamma \vdash_F F : A^{\perp\!\!\!\perp} \rrbracket$
3. *if* $\Gamma \vdash_V V : A$ *then* $\llbracket \Gamma \vdash_V V : A \rrbracket$
4. *if* $\Gamma \vdash_E E : A^{\perp\!\!\!\perp}$ *then* $\llbracket \Gamma \vdash_E E : A^{\perp\!\!\!\perp} \rrbracket$
5. *if* $\Gamma \vdash_t t : A$ *then* $\llbracket \Gamma \vdash_t t : A \rrbracket$
6. *if* $\Gamma \vdash_e e : A^{\perp\!\!\!\perp}$ *then* $\llbracket \Gamma \vdash_e e : A^{\perp\!\!\!\perp} \rrbracket$
7. *if* $\Gamma \vdash_c c$ *then* $\llbracket \Gamma \vdash_c c \rrbracket$
8. *if* $\Gamma \vdash_l l$ *then* $\llbracket \Gamma \vdash_l l \rrbracket$
9. *if* $\Gamma \vdash_\tau \tau : \Gamma'$ *then* $\llbracket \Gamma \vdash_\tau \tau : \Gamma' \rrbracket$

Proof. By induction over the typing rules. Let Γ be a typing context and σ be a suitable translation of names of Γ. We (ab)use of Lemma 6.30 to make the derivations more compact by systematically weakening contexts as soon as possible. We also compact the first $\forall$- and λ-introductions in one rule.

1. Strong values

- **Case** $\llbracket \boldsymbol{k} \rrbracket_v^\sigma$. $\llbracket \boldsymbol{k} \rrbracket_v^\sigma = \boldsymbol{k}$, which has the desired type by hypothesis.

- **Case** $\llbracket \lambda x_i.t \rrbracket_v^\sigma$. In the source language, we have:

$$\dfrac{\Gamma, x : A \vdash_t t : B}{\Gamma \vdash_v \lambda x : A \to B}$$

Hence, if n is fresh (w.r.t. σ), $\sigma[x := n]$ is suitable for $\Gamma, x : A$, and we get by induction a proof Π_t of $\llbracket t \rrbracket_t^{\sigma[x:=n]} : \llbracket \Gamma, x : A \rrbracket_\Gamma^{\sigma[x:=n]} \triangleright_t \iota(B)$. Observing that $\llbracket \Gamma, x : A \rrbracket_\Gamma^{\sigma[x:=n]} = \llbracket \Gamma \rrbracket_\Gamma^\sigma, n : \iota(A)$ we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi_t}{\vdash \llbracket t \rrbracket_t^{\sigma[x:=n]} : \llbracket \Gamma, x : A \rrbracket_\Gamma^{[\sigma,x:=n]} \triangleright_t \iota(B)} \quad \dfrac{\dfrac{}{Y <: \llbracket \Gamma \rrbracket_\Gamma^\sigma \vdash Y <: \llbracket \Gamma \rrbracket_\Gamma^\sigma}\;{\scriptstyle(<:_{\mathsf{ax}})}}{Y <: \llbracket \Gamma \rrbracket_\Gamma^\sigma \vdash Y, n : A <: \llbracket \Gamma \rrbracket_\Gamma^\sigma, n : A}\;{\scriptstyle(<:_2)}}{Y <: \llbracket \Gamma \rrbracket_\Gamma^\sigma \vdash \llbracket t \rrbracket_t^{\sigma[x:=n]} : Y, n : A \to Y, n : A \triangleright_E \iota(B) \to \bot}\;{\scriptstyle(\forall_E)} \quad \boldsymbol{\Pi_\tau}}{Y <: \llbracket \Gamma \rrbracket_\Gamma^\sigma, \tau : Y, u : Y \triangleright_t \iota(A); \vdash \llbracket t \rrbracket_t \; \tau[u] : Y, n : A \triangleright_E \iota(B) \to \bot}\;{\scriptstyle(@)} \quad \boldsymbol{\Pi_E}}{Y <: \llbracket \Gamma \rrbracket_\Gamma^\sigma, \tau : Y, u : Y \triangleright_t \iota(A), E : Y \triangleright_E \iota(B) \vdash \llbracket t \rrbracket_t^{\sigma[x:=n]} \; \tau[u] \; E : \bot}\;{\scriptstyle(@)}}{\vdash \lambda \tau u E. \llbracket t \rrbracket_t^{\sigma[x:=n]} \; \tau[u] \; E : \forall Y <: \llbracket \Gamma \rrbracket_\Gamma^\sigma . Y \to Y \triangleright_t \iota(A) \to Y \triangleright_E \iota(B) \to \bot}\;{\scriptstyle(\lambda)}$$

where:

- Π_τ is the following subproof:

$$\dfrac{\dfrac{}{\tau : Y \vdash \tau : Y}\;{\scriptstyle(\mathrm{Ax})} \quad \dfrac{\dfrac{}{u : Y \triangleright_t \iota(A) \vdash u : Y \triangleright_t \iota(A)}\;{\scriptstyle(\mathrm{Ax})}}{Y \triangleright_t \iota(A) \vdash [n := u] : Y \triangleright_\tau \iota(A)}\;{\scriptstyle(\tau_t)}}{\tau : Y, u : Y \triangleright_t \iota(A); \vdash \tau[n := u] : Y, n : A}\;{\scriptstyle(\tau\tau')}$$

- Π_E is the following proof (derivable using Corollary 6.32):

$$\dfrac{\dfrac{}{E : Y \triangleright_E \iota(B) \vdash E : (Y, n : \iota(A)) \triangleright_E \iota(B)}\;{\scriptstyle(\mathrm{Ax})} \quad \dfrac{\dfrac{}{\vdash Y <: Y}\;{\scriptstyle(<:_{\mathsf{ax}})}}{\vdash (Y, n : \iota(A)) <: Y}\;{\scriptstyle(<:_2)}}{E : Y \triangleright_E \iota(B) \vdash E : (Y, n : \iota(A)) \triangleright_E \iota(B)}\;{\scriptstyle <:_\triangleright}$$

2. Forcing contexts

- **Case** $\llbracket \boldsymbol{\kappa} \rrbracket_F^\sigma$. $\llbracket \boldsymbol{\kappa} \rrbracket_F^\sigma = \boldsymbol{\kappa}$, which has the desired type by hypothesis.

- **Case** $\llbracket t.E \rrbracket^{\sigma}_{F}$. In the source language, we have:

$$\frac{\Gamma \vdash_t t : A \quad \Gamma \vdash_E E : B^{\perp\!\!\perp}}{\Gamma \vdash_F t \cdot E : (A \to B)^{\perp\!\!\perp}}$$

Therefore, we obtain by induction a proof of $\vdash \llbracket t \rrbracket_t : \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma} \triangleright_t \iota(A)$ (and a proof of $\vdash \llbracket E \rrbracket_t : \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma} \triangleright_E \iota(B)$) that can be turned (using Corollary 6.32) into a proof Π_t of $Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma} \vdash \llbracket t \rrbracket_t : Y \triangleright_t \iota(A)$ for any Y (resp. Π_E of $Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma} \vdash \llbracket E \rrbracket_t : Y \triangleright_E \iota(B)$). Thus, we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{v : Y \triangleright_v (\iota(A) \to \iota(B)) \vdash v : Y \triangleright_v \iota(A) \to \iota(B)}\,(\mathrm{Ax}) \quad \dfrac{}{\vdash Y <: Y}\,(<:_{\mathsf{ax}})}{v : Y \triangleright_v (\iota(A) \to \iota(B)) \vdash v : Y \to Y \triangleright_t \iota(A) \to Y \triangleright_E B \to \bot}\,(\forall_E) \quad \dfrac{}{\tau : Y \vdash \tau : Y}\,(\mathrm{Ax})}{\tau : Y, v : Y \triangleright_v (\iota(A) \to \iota(B)) \vdash v\ \tau : Y \triangleright_t \iota(A) \to Y \triangleright_E \iota(B) \to \bot}\,(@) \quad \boldsymbol{\Pi_t}}{Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma}, \tau : Y, v : Y \triangleright_v (\iota(A) \to \iota(B)) \vdash v\ \tau\ \llbracket t \rrbracket_t : Y \triangleright_E \iota(B) \to \bot}\,(@) \quad \boldsymbol{\Pi_E}}{Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma}, \tau : Y, v : Y \triangleright_v (\iota(A) \to \iota(B))(\vdash v\ \tau\ \llbracket t \rrbracket_t\ \llbracket E \rrbracket^{\sigma}_{E} : \bot}\,(@)}{\vdash \lambda \tau v. v\ \tau\ \llbracket t \rrbracket_t\ \llbracket E \rrbracket^{\sigma}_{E} : \forall Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma}. Y \to Y \triangleright_v (\iota(A) \to \iota(B)) \to \bot}\,(\lambda)$$

3. Weak values

- **Case** $\llbracket v \rrbracket_V$. In the source language, we have:

$$\frac{\Gamma \vdash_v v : A}{\Gamma \vdash_V v : A}$$

Then we have by induction hypothesis a proof Π_v of $\vdash \llbracket v \rrbracket^{\sigma}_{v} : \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma} \triangleright_v \iota(A)$ and we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{F : Y \triangleright_F \iota(A) \vdash F : Y \triangleright_F \iota(A)}\,(\mathrm{Ax}) \quad \dfrac{}{\vdash Y <: Y}\,(<:_{\mathsf{ax}})}{F : Y \triangleright_F \iota(A) \vdash F : Y \to Y \triangleright_v \iota(A) \to \bot}\,(\forall_E) \quad \dfrac{}{\tau : Y \vdash \tau : Y}\,(\mathrm{Ax})}{Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma}, \tau : Y, F : Y \triangleright_F \iota(A) \vdash F\ \tau : Y \triangleright_v \iota(A) \to \bot}\,(@) \quad \dfrac{\boldsymbol{\Pi_v} \quad \dfrac{}{Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma} \vdash Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma}}\,(\mathrm{Ax})}{Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma} \vdash \llbracket v \rrbracket^{\sigma}_{v} : Y \triangleright_v \iota(A)}\,{<:_{\triangleright}}}{Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma}, \tau : Y, F : Y \triangleright_F \iota(A) \vdash F\ \tau\ \llbracket v \rrbracket^{\sigma}_{v} : \bot}\,(@)}{\vdash \lambda \tau F. F\ \tau\ \llbracket v \rrbracket^{\sigma}_{v} : \forall Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma}. Y \to Y \triangleright_F \iota(A) \to \bot}\,(\lambda)$$

where we used Corollary 6.32 on the right part of the proof. Observe that $\uparrow^t V$ is in fact independent of the level t and that we could as well have written $\llbracket v \rrbracket_V = \uparrow \llbracket v \rrbracket^{\sigma}_{v}$. We thus proved the admissibility of the following rule:

$$\frac{\Gamma \vdash V : \Upsilon \triangleright_V A}{\Gamma \vdash \uparrow^t V : \Upsilon \triangleright_t A}\,(\uparrow)$$

- **Case** $\llbracket x \rrbracket_V$. In the source language, we have:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash_V x : A}$$

so that Γ is of the form $\Gamma_0, x : A, \Gamma_1$. By definition, we have:

$$\llbracket x \rrbracket_V = \lambda \tau F.\, \mathtt{let}\, \tau_0, t, \tau_1 = \mathtt{split}\ \tau\ n\ \mathtt{in}\, t\ \tau_0\ (\lambda \tau_0' V. V\ \tau_0'[n := \uparrow^t V] \tau_1\ F) \qquad \text{where } n = \sigma(x)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{t : Y_0 \triangleright_t \iota(A) \vdash t : Y_0 \triangleright_t \iota(A)}\,(\mathrm{Ax}) \quad \dfrac{}{\vdash Y_0 <: Y_0}\,(<:_{\mathsf{ax}})}{t : Y_0 \triangleright_t \iota(A) \vdash t : Y_0 \to Y_0 \triangleright_E \iota(A) \to \bot}\,(\forall_E) \quad \dfrac{}{\tau_0 : Y_0 \vdash \tau_0 : Y_0}\,(\mathrm{Ax})}{\tau_0 : Y_0, t : Y_0 \triangleright_t A \vdash t\ \tau_0 : Y_0 \triangleright_E \iota(A) \to \bot}\,(@) \quad \boldsymbol{\Pi_E}}{\tau_0 : Y_0, t : Y_0 \triangleright_t \iota(A), \tau_1 : (Y_0, n : \iota(A)) \triangleright_\tau Y_1, F : (Y_0, n : \iota(A), Y_1) \triangleright_F \iota(A) \vdash t\ \tau_0\ E : \bot}\,(@)}{\tau : (Y_0, n : \iota(A), Y_1), F : (Y_0, n : \iota(A), Y_1) \triangleright_F \iota(A) \vdash \mathtt{let}\, \tau_0, t, \tau_1 = \mathtt{split}\ \tau\ n\ \mathtt{in}\, t\ \tau_0\ E : \bot}\,(\mathtt{split}) \quad \boldsymbol{\Pi_Y}}{Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma}, \tau : Y, F : Y \triangleright_F \iota(A) \vdash \mathtt{let}\, \tau_0, t, \tau_1 = \mathtt{split}\ \tau\ n\ \mathtt{in}\, t\ \tau_0\ E : \bot}\,(<:_{\mathtt{split}})}{\vdash \lambda \tau F.\, \mathtt{let}\, \tau_0, t, \tau_1 = \mathtt{split}\ \tau\ n\ \mathtt{in}\, t\ \tau_0\ E : \forall Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma}. Y \to Y \triangleright_F \iota(A) \to \bot}\,(\lambda)$$

where:

- Π_Y is simply the axiom rule:

$$\dfrac{}{Y <: (\llbracket\Gamma_0\rrbracket_\Gamma^\sigma, n : \iota(A), \llbracket\Gamma_1\rrbracket_\Gamma^\sigma) \vdash Y <: (\llbracket\Gamma_0\rrbracket_\Gamma^\sigma, n : \iota(A), \llbracket\Gamma_1\rrbracket_\Gamma^\sigma)}\,(<:_{\mathsf{ax}})$$

- $E = (\lambda\tau_0' V.V\ \tau_0'[n := V]\tau_1\ F)$ and Π_E is the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{V : Y_0' \triangleright_V \iota(A) \vdash\uparrow^t V : Y_0' \triangleright_t \iota(A)}\,(\text{Ax}) \qquad \dfrac{}{\vdash Y_0', n : A, Y_1 <: Y_0', n : A}\,(\text{Ax})}{V : Y_0' \triangleright_V \iota(A) \vdash V : (Y_0', n : A, Y_1) \to (Y_0', n : A, Y_1) \triangleright_E \iota(A) \to \bot}\,(\forall_E) \qquad \boldsymbol{\Pi_\tau}}{\tau_1 : (Y_0, n : A) \triangleright_\tau Y_1, Y_0' <: Y_0, \tau_0' : Y_0', V : Y_0' \triangleright_V \iota(A) \vdash V\ \tau_0'[n :=\uparrow^t V]\tau_1 : (Y_0', n : \iota(A), Y_1) \triangleright_F \iota(A) \to \bot}\,(@) \qquad \boldsymbol{\Pi_F}}{\dfrac{\tau_1 : (Y_0, n : A) \triangleright_\tau Y_1, F : (Y_0, n : \iota(A), Y_1) \triangleright_F \iota(A), Y_0' <: Y_0, \tau_0' : Y_0', V : Y_0' \triangleright_V \iota(A) \vdash V\ \tau_0'[n :=\uparrow^t V]\tau_1\ F : \bot}{\tau_1 : (Y_0, n : A) \triangleright_\tau Y_1, F : (Y_0, n : \iota(A), Y_1) \triangleright_F \iota(A) \vdash \lambda\tau_0' V.V\ \tau_0'[n :=\uparrow^t V]\tau_1\ F : Y_0 \triangleright_E \iota(A)}\,(\lambda)}\,(@)$$

- Π_F is the following proof, obtained by Corollary 6.32:

$$\dfrac{\dfrac{}{F : (Y_0, n : \iota(A), Y_1) \triangleright_F \iota(A) \vdash F : (Y_0, n : \iota(A), Y_1) \triangleright_F \iota(A)}\,(\text{Ax}) \qquad \dfrac{\dfrac{\dfrac{}{Y_0' <: Y_0 \vdash Y_0' <: Y_0}\,(<:_{\mathsf{ax}})}{\vdots}\,(<:_1)}{Y_0' <: Y_0 \vdash (Y_0', n : \iota(A), Y_1) <: (Y_0, n : \iota(A), Y_1)}\,(<:_1)}{Y_0' <: Y_0, F : (Y_0, n : \iota(A), Y_1) \triangleright_F \iota(A) \vdash F : (Y_0', n : \iota(A), Y_1) \triangleright_F \iota(A)}\,<:_\triangleright$$

- Π_τ is the following derivation

$$\dfrac{\dfrac{\dfrac{}{\tau_0' : Y_0' \vdash \tau_0' : Y_0'}\,(\text{Ax}) \qquad \dfrac{\dfrac{\dfrac{}{V : Y_0' \triangleright_V \iota(A) \vdash V : Y_0' \triangleright_V A}\,(\text{Ax})}{V : Y_0' \triangleright_V \iota(A) \vdash\uparrow^t V : Y_0' \triangleright_t A}\,(\uparrow)}{V : Y_0' \triangleright_V \iota(A) \vdash [n :=\uparrow^t V] : Y_0' \triangleright_\tau n : \iota(A)}\,(\tau_t)}{Y_0' <: Y_0, \tau_0' : Y_0', V : Y_0' \triangleright_V \iota(A) \vdash \tau_0'[n := V] : Y_0', n : \iota(A)}\,(\tau\tau') \qquad \boldsymbol{\Pi_{\tau_1}}}{\tau_1 : (Y_0, n : \iota(A)) \triangleright_\tau Y_1, Y_0' <: Y_0, \tau_0' : Y_0', V : Y_0' \triangleright_V \iota(A) \vdash \tau_0'[n := V]\tau_1 : Y_0', n : A, Y_1}\,(\tau_{<:})$$

- Π_{τ_1} is the following derivation:

$$\dfrac{\dfrac{}{\tau_1 : (Y_0, n : \iota(A)) \triangleright_\tau Y_1 \vdash \tau_1 : (Y_0, n : \iota(A)) \triangleright_\tau Y_1}\,(\text{Ax}) \qquad \dfrac{\dfrac{}{Y_0' <: Y_0 \vdash Y_0' <: Y_0}\,(<:_{\mathsf{ax}})}{Y_0' <: Y_0 \vdash Y_0', n : \iota(A) <: Y_0, n : \iota(A)}\,(<:_1)}{\tau_1 : (Y_0, n : \iota(A)) \triangleright_\tau Y_1, Y_0' <: Y_0 \vdash \tau_1 : (Y_0', n : \iota(A)) \triangleright_\tau Y_1}\,(\tau_{<:})$$

4. Catchable contexts

- **Case** $\llbracket F\rrbracket_E^\sigma$. This case is similar to the case $\llbracket v\rrbracket_V$.

- **Case** $\llbracket\tilde\mu[x].\langle x\|F\rangle\tau'\rrbracket_E^\sigma$. In the source language, we have:

$$\dfrac{\Gamma, x : A, \Gamma' \vdash_F F : A^{\perp\!\!\perp} \qquad \Gamma \vdash_\tau \tau : \Gamma'}{\Gamma \vdash_E \tilde\mu[x].\langle x\|F\rangle\tau : A^{\perp\!\!\perp}}$$

If n is fresh (w.r.t σ), $\sigma[x := n]$ is suitable for $\Gamma, x : A$, and we then have by induction hypothesis a proof of $\vdash \tau'' : \llbracket\Gamma, x : A\rrbracket^{\sigma'} \triangleright_\tau \llbracket\Gamma'\rrbracket^{\sigma'}$ and a proof Π_F of $\vdash \llbracket F\rrbracket_F^{\sigma'} : \llbracket\Gamma, x : A\rrbracket^{\sigma'} \triangleright_F \iota(A)$ where $\tau'', \sigma' = \llbracket\tau'\rrbracket^{\sigma,[x:=n]}$ for some fresh n. We can thus derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{V : Y \triangleright_V \iota(A) \vdash Y \triangleright_V \iota(A)}\,(\text{Ax}) \qquad \dfrac{\dfrac{}{\vdash Y <: Y}\,(<:_{\mathsf{ax}})}{\vdash Y, n : \iota(A), \llbracket\Gamma'\rrbracket_\Gamma^{\sigma'} <: Y}\,(<:_2)}{V : Y \triangleright_V \iota(A) \vdash V : (Y, n : \iota(A), \llbracket\Gamma'\rrbracket_\Gamma^{\sigma'}) \to (Y, n : \iota(A), \llbracket\Gamma'\rrbracket_\Gamma^{\sigma'}) \triangleright_F \iota(A) \to \bot}\,(\forall_E) \qquad \boldsymbol{\Pi_\tau}}{\tau : Y, V : Y \triangleright_V \iota(A) \vdash V\ \tau[n :=\uparrow^t V]\tau'' : (Y, n : \iota(A), \llbracket\Gamma'\rrbracket_\Gamma^{\sigma'} \iota(A) \triangleright_F \to \bot}\,(@) \qquad \boldsymbol{\Pi_F}}{\dfrac{Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma'}, \tau : Y, V : Y \triangleright_V \iota(A) \vdash V\ \tau[n :=\uparrow^t V]\tau''\ \llbracket F\rrbracket_F^{\sigma'} : \bot}{\vdash \lambda\tau V.V\ \tau[n :=\uparrow^t V]\tau''\ \llbracket F\rrbracket_F^{\sigma'} : \forall Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma'}.Y \to Y \triangleright_V \iota(A) \to \bot}\,(\lambda)}\,(@)$$

where:

- Π_F is the following proof, derived using Corollary 6.32 and Lemma 6.28:

$$\dfrac{\vdash \llbracket F\rrbracket_F^{\sigma'} : \llbracket\Gamma,n:\iota(A),\Gamma'\rrbracket_\Gamma^{\sigma'} \triangleright_F \iota(A) \qquad \dfrac{\dfrac{}{Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma'} \vdash Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma'}}\,(\text{Ax})}{Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma'} \vdash Y,n:\iota(A),\llbracket\Gamma'\rrbracket_\Gamma^{\sigma'} <: \llbracket\Gamma,n:\iota(A),\Gamma'\rrbracket_\Gamma^{\sigma'}}\,(<:_1)}{Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma'} \vdash \llbracket F\rrbracket_F^{\sigma'} : Y,n:\iota(A),\llbracket\Gamma'\rrbracket_\Gamma^{\sigma'} \triangleright_F \iota(A)}\,(\forall_E)$$

- Π_τ is the following proof:

$$\dfrac{\dfrac{\dfrac{}{\tau : Y \vdash \tau : Y}\,(\text{Ax}) \qquad \dfrac{\dfrac{\dfrac{}{V : Y \triangleright_V \iota(A) \vdash V : Y \triangleright_V \iota(A)}\,(\text{Ax})}{V : Y \triangleright_V \iota(A) \vdash\uparrow^t V : Y \triangleright_t \iota(A)}\,(\uparrow)}{V : Y \triangleright_V \iota(A) \vdash [n := V] : Y \triangleright_\tau n : \iota(A)}\,(\tau_t)}{\tau : Y, V : Y \triangleright_V \iota(A) \vdash \tau[n :=\uparrow^t V] : Y,n:\iota(A)}\,(\tau\tau') \qquad \boldsymbol{\Pi_{\tau'}}}{Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma'}, \tau : Y, V : Y' \triangleright_V \iota(A) \vdash \tau[n :=\uparrow^t V]\llbracket\tau'\rrbracket_\tau^{\sigma[x:=n]} : (Y,n:\iota(A),\llbracket\Gamma'\rrbracket^{\sigma[x:=n]})}\,(\tau\tau')$$

- $\Pi_{\tau'}$ is the following proof, obtained from the induction hypothesis for τ':

$$\dfrac{\vdash \llbracket\tau'\rrbracket_\tau^{\sigma[x:=n]} : \llbracket\Gamma\rrbracket_\Gamma^\sigma, n:\iota(A) \triangleright_\tau \llbracket\Gamma'\rrbracket^{\sigma[x:=n]} \qquad \dfrac{\overline{Y <: \llbracket\Gamma\rrbracket_\Gamma^\sigma \vdash Y <: \llbracket\Gamma\rrbracket_\Gamma^\sigma}}{Y <: \llbracket\Gamma\rrbracket_\Gamma^\sigma \vdash Y,n:\iota(A) <: \llbracket\Gamma\rrbracket_\Gamma^\sigma, n:\iota(A)}}{Y <: \llbracket\Gamma\rrbracket_\Gamma^\sigma \vdash \llbracket\tau'\rrbracket_\tau^{\sigma[x:=n]} : Y,n:\iota(A) \triangleright_\tau \llbracket\Gamma'\rrbracket^{\sigma[x:=n]}}$$

5. Terms

- **Case $\llbracket V\rrbracket_t$.** This case is similar to the case $\llbracket v\rrbracket_V$.

- **Case $\llbracket \mu\alpha.c\rrbracket_t$.** In the $\overline{\lambda}_{[lv\tau\star]}$-calculus, we have:

$$\dfrac{\Gamma,\alpha : A^{\perp\!\!\perp} \vdash_c c}{\Gamma \vdash_t \mu\alpha.c : A}$$

If n is fresh (w.r.t σ), $\sigma[\alpha := n]$ is suitable for $\Gamma,\alpha : A^{\perp\!\!\perp}$, and we then have by induction hypothesis a proof Π_c of $\vdash \llbracket c\rrbracket_c^{[\sigma,x:=n]} : \llbracket\Gamma,\alpha : A^{\perp\!\!\perp}\rrbracket_\Gamma^{\sigma[\alpha:=n]} \triangleright_c \perp$. We can thus derive, using Lemma 6.28 to identify $\llbracket\Gamma\rrbracket_\Gamma^\sigma$ and $\llbracket\Gamma\rrbracket_\Gamma^{\sigma,[\alpha:=n]}$:

$$\dfrac{\dfrac{\dfrac{\vdash \llbracket c\rrbracket_c^{\sigma[\alpha:=n]} : \llbracket\Gamma,\alpha : A^{\perp\!\!\perp}\rrbracket_\Gamma^{\sigma[\alpha:=n]} \triangleright_c \perp \qquad \dfrac{\dfrac{}{Y <: \llbracket\Gamma\rrbracket_\Gamma^\sigma \vdash Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma[\alpha:=n]}}\,(<:_{\text{ax}})}{Y <: \llbracket\Gamma\rrbracket_\Gamma^\sigma \vdash (Y,n:\iota(A)^{\perp\!\!\perp}) <: \llbracket\Gamma,\alpha : A^{\perp\!\!\perp}\rrbracket^{\sigma[\alpha:=n]}}\,(<:_1)}{Y <: \llbracket\Gamma\rrbracket_\Gamma^\sigma \vdash \llbracket c\rrbracket_c^{\sigma[\alpha:=n]} : (Y,n:\iota(A)^{\perp\!\!\perp}) \to \perp}\,(\forall_E) \qquad \boldsymbol{\Pi_\tau}}{Y <: \llbracket\Gamma\rrbracket_\Gamma^\sigma, \tau : Y, E : Y \triangleright_E \iota(A) \vdash \llbracket c\rrbracket_c^{\sigma[\alpha:=n]}\ \tau[n := E] : \perp}\,(@)}{\vdash \lambda\tau E.\llbracket c\rrbracket_c^{\sigma[\alpha:=n]}\ \tau[n := E] : \forall Y <: \llbracket\Gamma\rrbracket_\Gamma^\sigma.Y \to Y \triangleright_E \iota(A) \to \perp}\,(\lambda)$$

where Π_τ is the following derivation:

$$\dfrac{\dfrac{}{\tau : Y \vdash \tau : Y}\,(\text{Ax}) \qquad \dfrac{\dfrac{}{E : Y \triangleright_E \iota(A) \vdash E : Y \triangleright_E \iota(A)}\,(\text{Ax})}{E : Y \triangleright_E \iota(A) \vdash [n := E] : (Y \triangleright_\tau n : \iota(A)^{\perp\!\!\perp})}\,(\tau_t)}{\tau : Y, E : Y \triangleright_E \iota(A) \vdash \tau[n := E] : (Y,n:\iota(A)^{\perp\!\!\perp})}\,(\tau\tau')$$

6. Contexts

- **Case** $\llbracket E \rrbracket_e$. This case is similar to the case $\llbracket v \rrbracket_V$.

- **Case** $\llbracket \tilde{\mu}x.c \rrbracket_e$. This case is similar to the case $\llbracket \mu\alpha.c \rrbracket_t$.

7. Commands

- **Case** $\llbracket \langle t \| e \rangle \rrbracket_c$. In the $\overline{\lambda}_{[lv\tau\star]}$-calculus, we have:

$$\frac{\Gamma \vdash_t t : A \qquad \Gamma \vdash_e e : A^{\perp\!\!\perp}}{\Gamma \vdash_c \langle t \| e \rangle}$$

We thus get by induction two proofs $\vdash \llbracket e \rrbracket_e : \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma} \triangleright_e \iota(A)$ and $\vdash \llbracket t \rrbracket_t : \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma} \triangleright_t \iota(A)$ We can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash \llbracket e \rrbracket_e : \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma} \triangleright_e \iota(A) \qquad \mathbf{\Pi}_Y}{Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma} \vdash \llbracket e \rrbracket_e : Y \to Y \triangleright_t \iota(A) \to \bot}\,(\forall_E) \qquad \dfrac{}{\tau : Y \vdash \tau : Y}\,(\text{Ax})}{Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma}, \tau : Y \vdash \llbracket e \rrbracket_e\ \tau : Y \triangleright_t \iota(A) \to \bot}\,(@) \qquad \dfrac{\vdash \llbracket t \rrbracket_t : \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma} \triangleright_t \iota(A) \quad \mathbf{\Pi}_Y}{Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma} \vdash \llbracket t \rrbracket_t : Y \triangleright_t \iota(A)}\,(\forall_E)}{Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma}, \tau : Y \vdash \llbracket e \rrbracket_e\ \tau\ \llbracket t \rrbracket_t : \bot}\,(@)}{\vdash \lambda\tau.\llbracket e \rrbracket_t\ \tau\ \llbracket t \rrbracket_t : \forall Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma}.Y \to \bot}\,(\lambda)$$

where Π_Y is simply the axiom rule:

$$\dfrac{}{Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma} \vdash Y <: \llbracket \Gamma \rrbracket^{\sigma}_{\Gamma}}\,(<:_{\text{ax}})$$

8. Closures

- **Case** $\llbracket \langle t \| e \rangle \tau \rrbracket^{\sigma}_{l}$. In the $\overline{\lambda}_{[lv\tau\star]}$-calculus, we have:

$$\frac{\Gamma, \Gamma' \vdash_c c \qquad \Gamma \vdash_\tau \tau : \Gamma'}{\Gamma \vdash_l c\tau}$$

We thus get by induction two proofs $\vdash \tau' : \llbracket \Gamma \rrbracket^{\sigma'}_{\Gamma} \triangleright_\tau \llbracket \Gamma' \rrbracket^{\sigma'}_{\Gamma}$ and $\vdash \llbracket c \rrbracket^{\sigma'}_{c} : \llbracket \Gamma, \Gamma' \rrbracket^{\sigma'}_{\Gamma} \triangleright_c \bot$ where $\tau', \sigma' = \llbracket \tau \rrbracket^{\sigma}_{\tau}$. We can derive:

$$\dfrac{\dfrac{\dfrac{\vdash \llbracket c \rrbracket^{\sigma'}_{c} : \llbracket \Gamma, \Gamma' \rrbracket^{\sigma'}_{\Gamma} \triangleright_c \bot \qquad \dfrac{\dfrac{}{Y <: \llbracket \Gamma \rrbracket^{\sigma'}_{\Gamma} \vdash Y <: \llbracket \Gamma \rrbracket^{\sigma'}_{\Gamma}}\,(<:_{\text{ax}})}{Y <: \llbracket \Gamma \rrbracket^{\sigma'}_{\Gamma} \vdash Y, \llbracket \Gamma' \rrbracket^{\sigma'}_{\Gamma} <: \llbracket \Gamma, \Gamma' \rrbracket^{\sigma'}_{\Gamma}}\,(<:_1)}{Y <: \llbracket \Gamma \rrbracket^{\sigma'}_{\Gamma} \vdash \llbracket c \rrbracket^{\sigma'}_{c} : Y, \llbracket \Gamma' \rrbracket^{\sigma'}_{\Gamma} \to \bot}\,(\forall_E) \qquad \mathbf{\Pi}_{\boldsymbol{\tau}}}{Y <: \llbracket \Gamma \rrbracket^{\sigma'}_{\Gamma}, \tau_0 : Y \vdash \llbracket c \rrbracket^{\sigma'}_{c}\ \tau_0\tau' : \bot}\,(@)}{\vdash \lambda\tau_0.\llbracket c \rrbracket^{\sigma'}_{c}\ \tau_0\tau' : \forall Y <: \llbracket \Gamma \rrbracket^{\sigma'}_{\Gamma}.Y \to \bot}\,(\lambda)$$

where Π_τ is the following subderivation:

$$\dfrac{\dfrac{}{\tau_0 : Y \vdash \tau_0 : Y}\,(\text{Ax}) \qquad \dfrac{\vdash \tau' : \llbracket \Gamma \rrbracket^{\sigma'}_{\Gamma} \triangleright_\tau \llbracket \Gamma' \rrbracket^{\sigma'}_{\Gamma} \quad \dfrac{}{Y <: \llbracket \Gamma \rrbracket^{\sigma'}_{\Gamma} \vdash Y <: \llbracket \Gamma \rrbracket^{\sigma'}_{\Gamma}}\,(<:_{\text{ax}})}{Y <: \llbracket \Gamma \rrbracket^{\sigma'}_{\Gamma} \vdash \tau' : Y \triangleright_\tau \llbracket \Gamma' \rrbracket^{\sigma'}_{\Gamma}}\,(\tau_{<:})}{Y <: \llbracket \Gamma \rrbracket^{\sigma'}_{\Gamma}, \tau_0 : Y \vdash \tau_0\tau' : Y, \llbracket \Gamma' \rrbracket^{\sigma'}_{\Gamma}}\,(\tau\tau')$$

9. Stores

- **Case** $\tau[x := t]$. We only consider the case $\tau[x := t]$, the proof for the case $\tau[\alpha := E]$ is identical. This corresponds to the typing rule:

$$\frac{\Gamma \vdash_\tau \tau : \Gamma' \qquad \Gamma, \Gamma' \vdash_t t : A}{\Gamma \vdash_\tau \tau[x := t] : \Gamma', x : A}$$

By induction, we obtain two proofs of $\vdash \tau' : \llbracket\Gamma\rrbracket_{\Gamma}^{\sigma'} \triangleright_{\tau} \llbracket\Gamma'\rrbracket_{\Gamma}^{\sigma'}$ and $\vdash \llbracket t\rrbracket_{t}^{\sigma'} : \llbracket\Gamma,\Gamma'\rrbracket_{\Gamma}^{\sigma'} \triangleright_{t} \iota(A)$ where $\tau',\sigma' = \llbracket\tau\rrbracket_{\tau}^{\sigma}$ We can thus derive:

$$\dfrac{\vdash \tau' : \llbracket\Gamma\rrbracket_{\Gamma}^{\sigma'} \triangleright_{\tau} \llbracket\Gamma'\rrbracket_{\Gamma}^{\sigma'} \qquad \dfrac{\vdash \llbracket t\rrbracket_{t}^{\sigma'} : \llbracket\Gamma,\Gamma'\rrbracket_{\Gamma}^{\sigma'} \triangleright_{t} n : \iota(A)}{\vdash [n := \llbracket t\rrbracket_{t}^{\sigma'}] : \llbracket\Gamma,\Gamma'\rrbracket_{\Gamma}^{\sigma'} \triangleright_{\tau} n : \iota(A)}\,{\scriptstyle(\tau_t)}}{\vdash \tau'[n := \llbracket t\rrbracket_{t}^{\sigma'}] : \llbracket\Gamma\rrbracket_{\Gamma}^{\sigma'} \triangleright_{\tau} \llbracket\Gamma'\rrbracket_{\Gamma}^{\sigma'}, n : \iota(A)}\,{\scriptstyle(\tau\tau')}$$

□

Combining the preservation of reduction through the CPS and a proof of normalization of our target language (that one could obtain for instance using realizability techniques again), the former theorem would provide us with an alternative proof of normalization of the $\bar{\lambda}_{l\upsilon}$- and $\bar{\lambda}_{[l\upsilon\tau\star]}$-calculi.

6.4 Introducing De Bruijn levels

One standard way to handle issues related to α-conversion is to use De Bruijn indices [38]. In a nutshell, the De Bruijn notation is a nameless representation for λ-terms which replaces a bounded variable x by the number of λ that are crossed between the variable and its binder. For instance, the term $\lambda x.x$ is written $\lambda.0$, $\lambda xy.x$ is written $\lambda.\lambda.1$ and $\lambda x.x(\lambda y.xy)$ is written $\lambda.0(\lambda.1\,0)$. On the contrary, De Bruijn levels attributes a fixed number to λ binders (according to their "levels", that is how many former binders are crossed to reach them) and number variables in function of their binder's number. For instance, in the term $\lambda x.x(\lambda y.xy)$, the first binder λx is at top-level (level 0), while λy is at level 1. Using De Bruijn levels, this term is thus written $\lambda.1(\lambda.0\,1)$. These well-known techniques are very useful when it comes to implementation to prevent problem of α-conversion.

As we shall now see, the problem α-conversion needs to be handled carefully for the $\bar{\lambda}_{[l\upsilon\tau\star]}$-calculus and its continuation-passing-style translation, leading otherwise to non-terminating computations. This is why we needed to add explicit renaming to the translation of the previous section, since this problem was not tackled in the original translation. Another way of solving this difficulty consists in an adaptation of De Bruijn levels. Interestingly, it turns out that through the CPS, De Bruijn levels unveil some computational content related with store extensions.

6.4.1 The need for α-conversion

As for the proof of normalization, we observe in Figure 6.7 that the translation relies on names which implicitly suggests ability to perform α-conversion at run-time. Let us take a closer look at an example to better understand this phenomenon.

Example 6.34 (Lack of α-conversion). Let us consider a typed closure $\langle t\|e\rangle\tau$ such that:

$$\dfrac{\dfrac{\dfrac{\pi_t}{\Gamma \vdash_t t : A} \qquad \dfrac{\pi_e}{\Gamma \vdash_e e : A^{\perp\!\!\perp}}}{\Gamma \vdash_c \langle t\|e\rangle} \qquad \dfrac{\pi_\tau}{\vdash_\tau \tau : \Gamma}}{\vdash_l \langle t\|e\rangle\tau}$$

Assume that both t and e introduce a new variable x in their sub-derivations π_t and π_e, which will be the case for instance if $t = \mu\alpha.\langle u\|\tilde{\mu}x.\langle x\|\alpha\rangle\rangle$ and $e = \tilde{\mu}x.\langle x\|F\rangle$. This is compatible with previous typing derivation, however, this command would reduce (without α-conversion) as follows:

$$\begin{aligned}
\langle \mu\alpha.\langle u\|\tilde{\mu}x.\langle x\|\alpha\rangle\rangle\|\tilde{\mu}x.\langle x\|F\rangle\rangle &\to \langle x\|F\rangle[x := \mu\alpha.\langle u\|\tilde{\mu}x.\langle x\|\alpha\rangle\rangle] \\
&\to \langle \mu\alpha.\langle u\|\tilde{\mu}x.\langle x\|\alpha\rangle\rangle\|\tilde{\mu}[x].\langle x\|F\rangle\rangle \\
&\to \langle u\|\tilde{\mu}x.\langle x\|\alpha\rangle\rangle[\alpha := \tilde{\mu}[x].\langle x\|F\rangle] \\
&\to \langle x\|\alpha\rangle[\alpha := \tilde{\mu}[x].\langle x\|F\rangle, x := u] \\
&\to \langle x\|\tilde{\mu}[x].\langle x\|F\rangle\rangle[\alpha := \tilde{\mu}[x].\langle x\|F\rangle, x := u] \\
&\to \langle x\|F\rangle[\alpha := \tilde{\mu}[x].\langle x\|F\rangle, x := u, x := x] \\
&\to \langle x\|\tilde{\mu}[x].\langle x\|F\rangle\rangle[\alpha := \tilde{\mu}[x].\langle x\|F\rangle, x := u] \\
&\to \ldots
\end{aligned}$$

This command will then loop forever because of the auto-reference $[x := x]$ in the store. ⌟

This problem is reproduced through a naive CPS translation without renaming (as it was originally defined in [4]). In fact, the translation is somewhat even more problematic. Since "different" variables named x (that is variables which are bound by different binders) are translated independently (*e.g.* $\llbracket\langle t\|e\rangle\rrbracket$ is defined from $\llbracket e\rrbracket$ and $\llbracket t\rrbracket$), there is no hope to perform α-conversion on the fly during the translation. Moreover, our translation (as well as the original CPS in [4]) is defined modulo administrative translation (observe for instance that the translation of $\llbracket\lambda x.v\rrbracket_v^\sigma\ \tau\ V$ makes the λx binder vanish). Thus, the problem becomes unsolvable after the translation, as illustrated in the following example.

Example 6.35 (Lack of α-conversion in the CPS)**.** The naive translation (*i.e.* without renaming) of the same closure is again a program that will loop forever:

$$\begin{aligned}
\llbracket c\varepsilon\rrbracket &= \llbracket e\rrbracket_e\ \varepsilon\ \llbracket t\rrbracket_t = \llbracket\tilde{\mu}x.\langle x\|F\rangle\rrbracket_e\ \varepsilon\ \llbracket t\rrbracket_t \\
&= \llbracket\langle x\|F\rangle\rrbracket_c\ [x := \llbracket t\rrbracket_t] \\
&= \llbracket x\rrbracket_x\ [x := \llbracket t\rrbracket_t]\ \llbracket F\rrbracket_F \\
&= \llbracket\mu\alpha.\langle u\|\tilde{\mu}x.\langle x\|\alpha\rangle\rangle\rrbracket_t\ \varepsilon\ (\lambda\tau\lambda V.V\ \tau[x :=\uparrow^t V]\ \llbracket F\rrbracket_F) \\
&= \llbracket\langle u\|\tilde{\mu}x.\langle x\|\alpha\rangle\rangle\rrbracket_t\ [\alpha := \lambda\tau\lambda V.V\ \tau[x :=\uparrow^t V]\ \llbracket F\rrbracket_F] \\
&= \llbracket\tilde{\mu}x.\langle x\|\alpha\rangle\rrbracket_e\ [\alpha := \lambda\tau\lambda V.V\ \tau[x :=\uparrow^t V]\ \llbracket F\rrbracket_F]\ \llbracket u\rrbracket_t \\
&= \llbracket\langle x\|\alpha\rangle\rrbracket_c\ [\alpha := \lambda\tau\lambda V.V\ \tau[x :=\uparrow^t V]\ \llbracket F\rrbracket_F, x := \llbracket u\rrbracket_t] \\
&= \llbracket\alpha\rrbracket_E\ [\alpha := \lambda\tau\lambda V.V\ \tau[x :=\uparrow^t V]\ \llbracket F\rrbracket_F, x := \llbracket u\rrbracket_t]\ \llbracket x\rrbracket_V \\
&= (\lambda\tau\lambda V.V\ \tau[x :=\uparrow^t V])\ [\alpha := \lambda\tau\lambda V.V\ \tau[x :=\uparrow^t V]\ \llbracket F\rrbracket_F, x := \llbracket u\rrbracket_t]\ \llbracket x\rrbracket_V \\
&\to \llbracket x\rrbracket_V\ [\alpha := \lambda\tau\lambda V.V\ \tau[x :=\uparrow^t V]\ \llbracket F\rrbracket_F, x := \llbracket u\rrbracket_t, x := \llbracket x\rrbracket_t]
\end{aligned}$$

Observe that as the translation is defined modulo administrative reduction, the first equations indeed are equalities, and that when the reduction is performed, the two "different" x are not bound anymore. Thus, there is no way to achieve any kind of α-conversion to prevent the formation of the cyclic reference $[x := \llbracket x\rrbracket_V]$. ⌟

This is why we would need either to be able to perform α-conversion while executing the translation of a command, assuming that we can find a smooth way to do it, or to explicitly handle the renaming as we did in Section 8.3. As highlighted by the next example, this problem does not occur with the translation we defined, since two different fresh names are attributed to the "different" variables x.

Example 6.36 (Explicit renaming)**.** To compact the notations, we will write $[{}^x_m|{}^\alpha_\gamma|\ldots]$ for the renaming substitution $[x := m, \alpha := \gamma, \ldots]$, where we adopt the convention that the most recent binding is on written on the right. As a binding $[x := n]$ overwrites any former binding $[x := m]$, we write $[{}^{\alpha}_{\gamma}|{}^{x}_{n}]$ instead of $[{}^{x}_{m}|{}^{\alpha}_{\gamma}|{}^{x}_{n}]$.

$$\begin{aligned}
\llbracket c\varepsilon \rrbracket^{\varepsilon} &= \llbracket e \rrbracket_{e}^{\varepsilon}\ \varepsilon\ \llbracket t \rrbracket_{t}^{\varepsilon} = \llbracket \tilde{\mu}x.\langle x \| F\rangle \rrbracket_{e}^{\varepsilon}\ \varepsilon\ \llbracket t \rrbracket_{t}^{\varepsilon}\\
&= \llbracket \langle x \| F\rangle \rrbracket_{c}^{[{}^{x}_{m}]}\ [m := \llbracket t \rrbracket_{t}]\\
&= \llbracket x \rrbracket_{t}^{[{}^{x}_{m}]}\ [m := \llbracket t \rrbracket_{t}^{\varepsilon}]\ \llbracket F \rrbracket_{F}^{[{}^{x}_{m}]}\\
&= \llbracket \mu\alpha.\langle u \| \tilde{\mu}x.\langle x \| \alpha\rangle\rangle \rrbracket_{t}^{[{}^{x}_{m}]}\ \varepsilon\ (\lambda\tau\lambda V.V\ \tau[m :=\uparrow^{t} V]\ \llbracket F \rrbracket_{F}^{[{}^{x}_{m}]})\\
&= \llbracket \langle u \| \tilde{\mu}x.\langle x \| \alpha\rangle\rangle \rrbracket_{t}^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}]}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^{t} V]\ \llbracket F \rrbracket_{F}^{[{}^{x}_{m}]}]\\
&= \llbracket \tilde{\mu}x.\langle x \| \alpha\rangle \rrbracket_{e}^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}]}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^{t} V]\ \llbracket F \rrbracket_{F}^{[{}^{x}_{m}]}]\ \llbracket u \rrbracket_{t}^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}]}\\
&= \llbracket \langle x \| \alpha\rangle \rrbracket_{c}^{[x:=m,\alpha:=\gamma,x:=n]}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^{t} V]\ \llbracket F \rrbracket_{F}^{[{}^{x}_{m}]}, n := \llbracket u \rrbracket_{t}^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}]}]\\
&= \llbracket \alpha \rrbracket_{E}^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}|{}^{x}_{n}]}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^{t} V]\ \llbracket F \rrbracket_{F}^{[{}^{x}_{m}]}, n := \llbracket u \rrbracket_{t}^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}]}] \llbracket x \rrbracket_{V}^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}|{}^{x}_{n}]}\\
&= (\lambda\tau\lambda V.V\ \tau[m :=\uparrow^{t} V])\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^{t} V]\ \llbracket F \rrbracket_{F}^{[{}^{x}_{m}]}, n := \llbracket u \rrbracket_{t}^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}]}]\ \llbracket x \rrbracket_{V}^{[{}^{\alpha}_{\gamma}|{}^{x}_{n}]}\\
&\to \llbracket x \rrbracket_{V}^{[{}^{\alpha}_{\gamma}|{}^{x}_{n}]}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^{t} V]\ \llbracket F \rrbracket_{F}^{[{}^{x}_{m}]}, n := \llbracket u \rrbracket_{t}^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}]}, m := \llbracket x \rrbracket_{t}^{[{}^{\alpha}_{\gamma}|{}^{x}_{n}]}]\\
&= \llbracket x \rrbracket_{V}^{[{}^{\alpha}_{\gamma}|{}^{x}_{n}]}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^{t} V]\ \llbracket F \rrbracket_{F}^{[{}^{x}_{m}]}, n := \llbracket u \rrbracket_{t}^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}]}, m := \llbracket x \rrbracket_{t}^{[{}^{\alpha}_{\gamma}|{}^{x}_{n}]}]
\end{aligned}$$

We observe that in the end, the variable m is bound to the variable n, which is now correct. ⌟

Another way of ensuring the correctness of our translation is to correct the problem already in the $\overline{\lambda}_{[lv\tau\star]}$, using what we call De Bruijn levels. As we observed in the first example of this section, the issue arises when adding a binding $[x := ...]$ in a store that already contained a variable x. We thus need to ensure the uniqueness of names within the store. An easy way to do this consists in changing the names of variable bounded in the store by the position at which they occur in the store, which is obviously unique. Just as De Bruijn indices are pointers to the correct binder, De Bruijn levels are pointers to the correct cell of the environment. Before presenting formally the corresponding system and the adapted translation, let us take a look at the same example that we reduce using this idea. We use a mixed notation for names, writing x when a variable is bounded by a λ or a $\tilde{\mu}$, and x_i (where i is the relevant information) when it refers to a position in the store.

Example 6.37 (Reduction with De-Bruijn levels)**.** The same reduction is now safe if we replace stored variables by their De Bruijn level:

$$\begin{aligned}
\langle \mu\alpha.\langle u \| \tilde{\mu}x.\langle x \| \alpha\rangle\rangle \| \tilde{\mu}x.\langle x \| F\rangle\rangle &\to \langle x_0 \| F\rangle[{}^{0}\mu\alpha.\langle u \| \tilde{\mu}x.\langle x \| \alpha\rangle\rangle]\\
&\to \langle \mu\alpha.\langle u \| \tilde{\mu}x.\langle x \| \alpha\rangle\rangle \| \tilde{\mu}[x].\langle x \| F\rangle\rangle\\
&\to \langle u \| \tilde{\mu}x.\langle x \| \alpha_0\rangle\rangle[{}^{0}\tilde{\mu}[x].\langle x \| F\rangle]\\
&\to \langle x_1 \| \alpha_0\rangle[{}^{0}\tilde{\mu}[x].\langle x \| F\rangle, {}^{1}u]\\
&\to \langle x_1 \| \tilde{\mu}[x].\langle x \| F\rangle\rangle[\tilde{\mu}[x].\langle x \| F\rangle, {}^{1}u]\\
&\to \langle x_2 \| F\rangle[{}^{0}\tilde{\mu}[x].\langle x \| F\rangle, {}^{1}u, {}^{2}x_1]\\
&\to \langle x_1 \| \tilde{\mu}[x].\langle x \| F\rangle\rangle[{}^{0}\tilde{\mu}[x].\langle x \| F\rangle, {}^{1}u]\\
&\to \langle u \| F\rangle[{}^{0}\tilde{\mu}[x].\langle x \| F\rangle, {}^{1}u, {}^{2}u]
\end{aligned}$$

where x_i is a convenient notation to design the variable named with De Bruijn level i (*i.e.* pointers to the i^{th} cell). The exponents ${}^{0}, {}^{1}, \ldots$ to number the cells are only there to ease the readability. ⌟

6.4.2 The $\overline{\lambda}_{[lv\tau\star]}$-calculus with De Bruijn levels

We now use De Bruijn levels for variables (and co-variables) that are bounded in the store. We use the mixed notation[13] x_i where the relevant information is x when the variable is bounded within a proof (that is by a λ or $\tilde{\mu}$ binder), and where the relevant information is the number i once the variable has been bounded in the store (at position i). For binders of evaluation contexts, we similarly use De Bruijn levels, but with variables of the form α_i, where, again, α is a fixed name indicating that the variable is binding evaluation contexts, and the relevant information is the index i.

The corresponding syntax is now given by:

$$\begin{array}{llcl@{\qquad}llcl}
\textbf{Strong values} & v & ::= & \boldsymbol{k} \mid \lambda x_i.t & \textbf{Forcing contexts} & F & ::= & \boldsymbol{\kappa} \mid t \cdot E \\
\textbf{Weak values} & V & ::= & v \mid x_i & \textbf{Catchable contexts} & E & ::= & F \mid \alpha_i \mid \tilde{\mu}[x_i].\langle x_i \| F\rangle\tau \\
\textbf{Terms} & t,u & ::= & V \mid \mu\alpha_i.c & \textbf{Evaluation contexts} & e & ::= & E \mid \tilde{\mu}x_i.c
\end{array}$$

$$\begin{array}{llcl}
\textbf{Closures} & l & ::= & c\tau \\
\textbf{Commands} & c & ::= & \langle t \| e\rangle \\
\textbf{Stores} & \tau & ::= & \varepsilon \mid \tau[x_i := t] \mid \tau[\alpha_i := E]
\end{array}$$

The presence of names in the stores is absolutely useless[14] and only there for readability. As the store can be dynamically extended during the execution, the location of a term in the store and the corresponding pointer are likely to evolve (monotonically). Therefore, we need to be able to update De Bruijn levels within terms (contexts, etc...). To this end, we define the lifted term $\uparrow_n^{+i} t$ as the term t where all the free variables x_j with $j > n$ (resp. α_j) have been replaced by x_{j+i}. Formally, they are defined as follows:

$$\begin{array}{lcll}
\uparrow_n^{+i}(c\tau) & \triangleq & (\uparrow_n^{+i} c)(\uparrow_n^{+i} \tau) & \\
\uparrow_n^{+i}(\langle t \| e\rangle) & \triangleq & \langle \uparrow_n^{+i} t \| \uparrow_n^{+i} e\rangle & \\
 & & & \\
\uparrow_n^{+i} \varepsilon & \triangleq & \varepsilon & \\
\uparrow_n^{+i}(\tau[x_j := t]) & \triangleq & \uparrow_n^{+i}(\tau)([\uparrow_n^{+i} x_j :=\uparrow_n^{+i} t] & \\
\uparrow_n^{+i}(\tau[\alpha_j := E]) & \triangleq & \uparrow_n^{+i}(\tau[\uparrow_n^{+i} \alpha_j :=\uparrow_n^{+i} E] & \\
 & & & \\
\uparrow_n^{+i}(\boldsymbol{k}) & \triangleq & \boldsymbol{k} & \\
\uparrow_n^{+i}(\lambda x_j.t) & \triangleq & \lambda(\uparrow_n^{+i} x_j).(\uparrow_n^{+i} t) & \\
\uparrow_n^{+i}(x_j) & \triangleq & x_j & (\text{if } j < n) \\
\uparrow_n^{+i}(x_j) & \triangleq & x_{j+i} & (\text{if } j \geq n) \\
\uparrow_n^{+i}(\mu\alpha_j.c) & \triangleq & \mu(\uparrow_n^{+i} \alpha_i).(\uparrow_n^{+i} c) & \\
 & & & \\
\uparrow_n^{+i}(\boldsymbol{\kappa}) & \triangleq & \boldsymbol{\kappa} & \\
\uparrow_n^{+i}(t \cdot E) & \triangleq & (\uparrow_n^{+i} t) \cdot (\uparrow_n^{+i} E) & \\
\uparrow_n^{+i}(\alpha_j) & \triangleq & \alpha_j & (\text{if } j < n) \\
\uparrow_n^{+i}(\alpha_j) & \triangleq & \alpha_{j+i} & (\text{if } j \geq n) \\
\uparrow_n^{+i}(\tilde{\mu}[x_j].\langle x_j \| F\rangle\tau) & \triangleq & \tilde{\mu}[\uparrow_n^{+i} x_i].(\uparrow_n^{+i} \langle x_i \| F\rangle\tau) & \\
\uparrow_n^{+i}(\tilde{\mu}x_j.c) & \triangleq & \tilde{\mu}(\uparrow_n^{+i} x_i).(\uparrow_n^{+i} c) &
\end{array}$$

The corresponding reduction rules are given in Figure 6.8. Note that we choose to perform indices substitutions as soon as they come (maintaining the property that x_n is a variable referring to the $(n+1)^{\text{th}}$ element of the store), while it would also have been possible to store and compose them along

[13] Observe that we could also use usual De Bruijn indices for bounded variables within the terms

[14] In fact, it could even leads to inconsistencies if cell j was of the shape $[x_i := \ldots]$. The reduction rules will ensure that this never happens but if it was the case, the only relevant information would be the number of the cell (j).

$$\begin{array}{ccll}
\langle t \| \tilde{\mu} x_i . c \rangle \tau & \rightarrow & c[x_n/x_i]\tau[x_n := t] & \text{with } |\tau| = n \\
\langle \mu \alpha_i . c \| E \rangle \tau & \rightarrow & c[\alpha_n/\alpha_i]\tau[\alpha_n := E] & \text{with } |\tau| = n \\
\langle V \| \alpha_n \rangle \tau & \rightarrow & \langle V \| \tau(n) \rangle \tau & \\
\langle x_n \| F \rangle \tau[x_n := t]\tau' & \rightarrow & \langle t \| \tilde{\mu}[x_n].\langle x_n \| F \rangle \tau' \rangle \tau & \\
\langle V \| \tilde{\mu}[x_i].\langle x_i \| F \rangle \tau' \rangle \tau & \rightarrow & \langle V \| \uparrow_n^{+i} F \rangle \tau[x_n := V](\uparrow_n^{+i} \tau') & \text{with } |\tau| = n \\
\langle \lambda x_i . t \| u \cdot E \rangle \tau & \rightarrow & \langle u \| \tilde{\mu} x_n . \langle t[x_n/x_i] \| E \rangle \rangle \tau & \text{with } |\tau| = n
\end{array}$$

Figure 6.8: Reduction rules of the $\overline{\lambda}_{[lv\tau\star]}$-calculus with De Bruijn indices

the execution (so that x_n is a variable referring to the $(\sigma(n)+1)^{\text{th}}$ element of the store where σ is the current substitution). This could have seemed more natural for the reader familiar with compilation procedures that do not modify at run time but rather maintain the location of variables through this kind of substitution.

The typing rules are unchanged except for the one where indices should now match the length of the typing context. The resulting type system is given in Figure 6.9.

6.4.3 System F_Υ with De Bruijn levels

The translation for judgments and types, given in Figure 6.11, is almost the same than in the previous section, except that we avoid using names and rather use De Bruijn levels.

As for the target language, it is again an adaptation of System F with stores (lists), in which store subtyping is now witnessed by explicit coercions.

Definition 6.38 (Coercion)**.** We defined coercions to witness store subtyping $\Upsilon' <: \Upsilon$ as finite monotonic functions σ such that $\mathtt{dom}(\sigma) = [\![0, |\Upsilon| - 1]\!]$, $\mathtt{codom}(\sigma) \subseteq [\![0, |\Upsilon'| - 1]\!]$ and such that for all $i < |\Upsilon|$, $\Upsilon_i = \Upsilon'_{\sigma(i)}$. ⌟

In other words, σ indicates where to find each type of the list Υ in the list Υ'. We denote by $\sigma_{|n}$ the restriction of σ to $[0, n-1]$ and $\mathtt{id}_n$ the identity on $[0, n-1]$. We also define σ_p^+ the canonical extension of a function σ whose domain is $[\![0, n-1]\!]$ for some n and whose co-domain is included in $[\![0, p-1]\!]$ for some p by:

$$\sigma_p^+ : \left\{ \begin{array}{ccc} [0,n] & \rightarrow & [0,p] \\ i < n & \mapsto & \sigma(i) \\ n & \mapsto & p \end{array} \right.$$

Lemma 6.39. *If σ witnesses $\Upsilon' <: \Upsilon$ for some Υ, Υ', then $\sigma^+_{|\Upsilon'|}$ witnesses $\Upsilon', A <: \Upsilon, A$ for any type A.*

As we now got rid of names, we will now split stores with respect to an index. So that if we consider for instance a store of type $\Upsilon' <: (\Upsilon_0, A, \Upsilon_1)$, the knowledge of the position where to find the expected element of type A becomes crucial. In practice, it will be guided by the coercion witnessing $\Upsilon' <: (\Upsilon_0, A, \Upsilon_1)$. But to ensure the correctness of our typing rules, we now need to consider second-order variables (which are in fact vectors of second-order variables) with their arities. That is to say that we should denote by Y^p the vector of variables $Y_0, \ldots, Y_{p-1}$ and that $\forall Y <: \Upsilon . A$ is equivalent

$$\forall p_0 \forall Y^{p_0} \ldots \forall p_n \forall Y^{p_n} . (Y^{p_0} \Upsilon(0) Y^{p_1} \Upsilon(1) \ldots Y^{p_n}) <: \Upsilon \rightarrow A$$

where we have in fact $p_0 = \sigma(0)$, $p_1 = \sigma(1) - p_0 - 1$, etc... In particular, a careful manipulation of variables with their arities allows us to prove the following lemma:

Lemma 6.40. *The typing rules given for coercions in Figure 6.10 are equivalent to Definition 6.38,* i.e. *for all Υ, Υ', for all $i < |\Upsilon|$, $\Upsilon_i = \Upsilon'_{\sigma(i)}$.*

$$\frac{(\boldsymbol{k} : A) \in \mathcal{S}}{\Gamma \vdash_v \boldsymbol{k} : A} \qquad \frac{\Gamma, x_n : A \vdash_t t : B \quad |\Gamma| = n}{\Gamma \vdash_v \lambda x_n.t : A \to B} \qquad \frac{\Gamma(n) = (x_n : A)}{\Gamma \vdash_V x_n : A} \qquad \frac{\Gamma \vdash_v v : A}{\Gamma \vdash_V v : A}$$

$$\frac{(\boldsymbol{\kappa} : A) \in \mathcal{S}}{\Gamma \vdash_F \boldsymbol{\kappa} : A^{\perp\!\!\!\perp}} \qquad \frac{\Gamma \vdash_t t : A \qquad \Gamma \vdash_E E : B^{\perp\!\!\!\perp}}{\Gamma \vdash_F t \cdot E : (A \to B)^{\perp\!\!\!\perp}} \qquad \frac{\Gamma, \alpha_n : A^{\perp\!\!\!\perp} \vdash_c c \quad |\Gamma| = n}{\Gamma \vdash_t \mu\alpha_n.c : A} \qquad \frac{\Gamma \vdash_F F : A^{\perp\!\!\!\perp}}{\Gamma \vdash_E F : A^{\perp\!\!\!\perp}}$$

$$\frac{\Gamma \vdash_V V : A}{\Gamma \vdash_t V : A} \qquad \frac{\Gamma, \alpha_n : A^{\perp\!\!\!\perp} \vdash_c c \quad |\Gamma| = n}{\Gamma \vdash_t \mu\alpha_n.c : A} \qquad \frac{\Gamma \vdash_E E : A^{\perp\!\!\!\perp}}{\Gamma \vdash_e E : A^{\perp\!\!\!\perp}} \qquad \frac{\Gamma, x_n : A \vdash_c c \quad |\Gamma| = n}{\Gamma \vdash_e \tilde{\mu}x_n.c : A^{\perp\!\!\!\perp}}$$

$$\frac{\Gamma, x_i : A, \Gamma' \vdash_F F : A^{\perp\!\!\!\perp} \quad \Gamma, x_i : A \vdash_\tau \tau : \Gamma' \quad |\Gamma| = i}{\Gamma \vdash_E \tilde{\mu}[x_i].\langle x_i \| F \rangle \tau : A^{\perp\!\!\!\perp}} \qquad \frac{}{\Gamma \vdash_\tau \varepsilon : \varepsilon}$$

$$\frac{\Gamma \vdash_\tau \tau : \Gamma' \qquad \Gamma, \Gamma' \vdash_t t : A \quad |\Gamma, \Gamma'| = n}{\Gamma \vdash_\tau \tau[x_n := t] : \Gamma', x_n : A} \qquad \frac{\Gamma \vdash_\tau \tau : \Gamma' \qquad \Gamma, \Gamma' \vdash_E E : A^{\perp\!\!\!\perp} \quad |\Gamma, \Gamma'| = n}{\Gamma \vdash_\tau \tau[\alpha_n := E] : \Gamma', \alpha_n : A^{\perp\!\!\!\perp}}$$

$$\frac{\Gamma \vdash_t t : A \quad \Gamma \vdash_e e : A^{\perp\!\!\!\perp}}{\Gamma \vdash_c \langle t \| e \rangle} \qquad \frac{\Gamma, \Gamma' \vdash_c c \quad \Gamma \vdash_\tau \tau : \Gamma'}{\Gamma \vdash_l c\tau}$$

Figure 6.9: Typing rules for the $\overline{\lambda}_{[lv\tau\star]}$-calculus with De Bruijn

$$\frac{(x : A) \in \Gamma}{\Gamma; \Sigma \vdash x : A}\;(\mathrm{Ax}) \qquad \frac{\Gamma, x : A; \Sigma \vdash t : B \quad |\Gamma| = n}{\Gamma; \Sigma \vdash \lambda x.t : A \to B}\;(\lambda) \qquad \frac{\Gamma; \Sigma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma; \Sigma \vdash t\,u : B}\;(@)$$

$$\frac{\Gamma; \Sigma, \sigma : X <: \Upsilon \vdash t : A \quad X \notin FV(\Gamma, \Sigma)}{\Gamma; \Sigma \vdash \lambda\sigma.t : \forall X <: \Upsilon.A}\;(\forall_I) \qquad \frac{\Gamma; \Sigma \vdash t : \forall X <: \Upsilon.A \quad \Sigma \vdash \sigma : \Upsilon' <: \Upsilon}{\Gamma; \Sigma \vdash t\,\sigma : A\{X := \Upsilon'\}}\;(\forall_E)$$

$$\frac{(c : A) \in \mathcal{S}}{\Gamma; \Sigma \vdash \boldsymbol{k} : A}\;(c) \qquad \frac{\Gamma, x_{\tau_0} : \Upsilon_0, x : A, x_{\tau_1} : \Upsilon_1; \Sigma \vdash t : A \quad \Gamma \vdash \tau : \Upsilon_0, B, \Upsilon_1 \quad |\Upsilon_0| = n}{\Gamma; \Sigma \vdash \mathtt{let}\, x_{\tau_0}, x, x_{\tau_1} = \mathtt{split}\ \ \mathtt{as}\,(\tau)\ \mathtt{in}\ \ n\,\mathtt{in}\, t : A}\;(\mathtt{split})$$

$$\frac{}{\Gamma; \Sigma \vdash \varepsilon : \varepsilon \triangleright_\tau \varepsilon}\;(\varepsilon) \qquad \frac{\Gamma; \Sigma \vdash t : \Upsilon \triangleright_t A}{\Gamma; \Sigma \vdash [t] : \Upsilon \triangleright_\tau A}\;(\tau_t) \qquad \frac{\Gamma; \Sigma \vdash t : \Upsilon \triangleright_E A}{\Gamma; \Sigma \vdash [t] : \Upsilon \triangleright_\tau A^{\perp\!\!\!\perp}}\;(\tau_E)$$

$$\frac{\Gamma \vdash \tau : \Upsilon_0 \triangleright_\tau \Upsilon \qquad \Gamma \vdash \tau' : (\Upsilon_0, \Upsilon) \triangleright_\tau \Upsilon'}{\Gamma \vdash \tau\tau' : \Upsilon_0 \triangleright_\tau \Upsilon, \Upsilon'}\;(\tau\tau') \qquad \frac{}{\Sigma \vdash \sigma : \Upsilon' <: \varepsilon}\;(<:_\varepsilon)$$

$$\frac{(\sigma : \Upsilon' <: \Upsilon) \in \Sigma}{\Sigma \vdash \sigma : \Upsilon' <: \Upsilon}\;(<:_{\mathrm{ax}}) \qquad \frac{\Sigma \vdash \sigma : \Upsilon' <: \Upsilon \quad \sigma(|\Upsilon|) = |\Upsilon'|}{\Sigma \vdash \sigma : (\Upsilon', A) <: (\Upsilon, A)}\;(<:_1) \qquad \frac{\Sigma \vdash \sigma : \Upsilon' <: \Upsilon}{\Sigma \vdash \sigma : (\Upsilon', A) <: \Upsilon}\;(<:_2)$$

Figure 6.10: Typing rules of System F_Υ with De Bruijn levels

$$\begin{array}{lcl}
\llbracket \Gamma \vdash_e e : A^{\bot\!\!\bot} \rrbracket & \triangleq & \vdash \llbracket e \rrbracket_e : \llbracket \Gamma \rrbracket_\Gamma \triangleright_e \iota(A) \\
\llbracket \Gamma \vdash_t t : A \rrbracket & \triangleq & \vdash \llbracket t \rrbracket_t : \llbracket \Gamma \rrbracket_\Gamma \triangleright_t \iota(A) \\
\llbracket \Gamma \vdash_E E : A^{\bot\!\!\bot} \rrbracket & \triangleq & \vdash \llbracket E \rrbracket_E : \llbracket \Gamma \rrbracket_\Gamma \triangleright_E \iota(A) \\
\llbracket \Gamma \vdash_V V : A \rrbracket & \triangleq & \vdash \llbracket V \rrbracket_V : \llbracket \Gamma \rrbracket_\Gamma \triangleright_V \iota(A) \\
\llbracket \Gamma \vdash_F F : A^{\bot\!\!\bot} \rrbracket & \triangleq & \vdash \llbracket F \rrbracket_F : \llbracket \Gamma \rrbracket_\Gamma \triangleright_F \iota(A)
\end{array}
\qquad
\begin{array}{lcl}
\llbracket \Gamma \vdash_v v : A \rrbracket & \triangleq & \vdash \llbracket v \rrbracket_v : \llbracket \Gamma \rrbracket_\Gamma \triangleright_v \iota(A) \\
\llbracket \Gamma \vdash_c c \rrbracket & \triangleq & \vdash \llbracket c \rrbracket_c : \llbracket \Gamma \rrbracket_\Gamma \triangleright_c \bot \\
\llbracket \Gamma \vdash_l l \rrbracket & \triangleq & \vdash \llbracket l \rrbracket_l^{|\Gamma|} : \llbracket \Gamma \rrbracket_\Gamma \triangleright_c \bot \\
\llbracket \Gamma \vdash_\tau \tau : \Gamma' \rrbracket & \triangleq & \vdash \llbracket \tau \rrbracket_\tau : \llbracket \Gamma \rrbracket_\Gamma \triangleright_\tau \llbracket \Gamma' \rrbracket_\Gamma
\end{array}$$

$$\llbracket \varepsilon \rrbracket_\Gamma \triangleq \varepsilon \qquad \llbracket \Gamma, x_i : A \rrbracket_\Gamma \triangleq \llbracket \Gamma \rrbracket_\Gamma, \iota(A) \qquad \llbracket \Gamma, \alpha_i : A^{\bot\!\!\bot} \rrbracket_\Gamma \triangleq \llbracket \Gamma \rrbracket_\Gamma, \iota(A)^{\bot\!\!\bot}$$

$$\begin{array}{l}
\Upsilon \triangleright_c A \triangleq \forall Y <: \Upsilon.\, Y \to \bot \\
\Upsilon \triangleright_e A \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_t A) \to \bot \\
\Upsilon \triangleright_t A \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_E A) \to \bot \\
\Upsilon \triangleright_E A \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_V A) \to \bot
\end{array}
\qquad
\begin{array}{ll}
\Upsilon \triangleright_V A & \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_F A) \to \bot \\
\Upsilon \triangleright_F A & \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_v A) \to \bot \\
\Upsilon \triangleright_v A \to B & \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_t A) \to (Y \triangleright_E B) \to \bot \\
\Upsilon \triangleright_v X & \triangleq X
\end{array}$$

Figure 6.11: Translation of judgments and types

Even though arities are crucial to ensure the correctness of the definition in Figure 6.10 (in particular to define the relation $\sigma : \Upsilon' <: \Upsilon$ by means of inference rules), to ease the notation we will omit the arity most of the time. We will use the notation $\forall Y^n <: \Upsilon.A$ only when necessary.

The syntax of terms and types is given by:

$$\begin{array}{rcl}
t, u & ::= & x \mid \lambda x.t \mid t\,u \mid \tau \mid \lambda\sigma.t \mid t\,\sigma \\
 & \mid & \texttt{let}\,\tau, x, \tau' = \texttt{split}\,\tau''\,n\,\texttt{in}\,t \\
\tau, \tau' & ::= & \varepsilon \mid \tau[t]
\end{array}
\quad\Bigg|\quad
\begin{array}{rcl}
A, B & ::= & X \mid \bot \mid \Upsilon \triangleright_\tau \Upsilon' \mid A \to B \mid \forall Y <: \Upsilon.\, A \\
\Upsilon, \Upsilon' & ::= & \varepsilon \mid \Upsilon, A \mid \Upsilon, A^{\bot\!\!\bot} \mid Y
\end{array}$$

Once again, we will use Υ as a shorthand for typing stores of type $\varepsilon \triangleright_\tau A$. The typing rules are given in Figure 6.10 where the typing contexts are divided in two parts, Γ containing typing hypotheses and Σ the subtyping hypotheses, that are defined by:

$$\Gamma, \Gamma' ::= \varepsilon \mid \Gamma, x : A \qquad\qquad \Sigma, \Sigma' ::= \varepsilon \mid \Sigma, \sigma : (\Upsilon' <: \Upsilon)$$

Now that we gave a computational content to the subtyping relation, some properties that were defined axiomatically in Section 8.3 are now deducible from the characteristics of the coercions σ.

Proposition 6.41. *The subtyping relation $<:$ is an order relation on store types.*

1. *For any Υ, $\Sigma \vdash \mathtt{id}_{|\Upsilon|} : \Upsilon <: \Upsilon$*
2. *If $\Sigma \vdash \sigma : \Upsilon <: \Upsilon'$ and $\Sigma \vdash \sigma' : \Upsilon' <: \Upsilon''$, then $\Sigma \vdash \sigma' \circ \sigma : \Upsilon <: \Upsilon''$.*
3. *If $\Sigma \vdash \sigma : \Upsilon <: \Upsilon'$ and $\Sigma \vdash \sigma' : \Upsilon' <: \Upsilon$, then $\sigma' \circ \sigma = \sigma' \circ \sigma = \mathtt{id}_{|\Upsilon|}$ and $\Upsilon = \Upsilon'$.*

Proof. Straightforward from the definition of $\sigma : \Upsilon' <: \Upsilon$:

1. Obvious.
2. For all $i < |\Upsilon|$, we have $\Upsilon''_{\sigma'(\sigma(i)} = \Upsilon'_{\sigma(i))} = \Upsilon_i$.
3. Using the second item, we deduce that $\sigma' \circ \sigma$ witnesses $\Upsilon <: \Upsilon$. Both σ and σ' being monotonic functions, we deduce that $\sigma' = \sigma = \mathtt{id}_{|\Upsilon|}$ and that for all $i < |\Upsilon|$, $\Upsilon_i = \Upsilon'_i$. □

Proposition 6.42. *For any function σ and any types Υ,Υ', if $\vdash \sigma : \Upsilon' <: \Upsilon$ and Υ is of the form $\Upsilon = \Upsilon_0, A, \Upsilon_1$, then Υ' is of the form $\Upsilon' = \Upsilon_0', A, \Upsilon_1'$ such that $|\Upsilon_0'| = \sigma(|\Upsilon_0|)$ and $|\Upsilon_1'| = \sigma(|\Upsilon|) - |\Upsilon_0'| - 1$.*

Proof. Straightforward from the definitions. □

The former propositions shows that the following subtyping rules (where we use a compact version of the second-order variable) are admissible:

$$\frac{\Sigma \vdash \sigma : \Upsilon <: \Upsilon' \quad \Sigma \vdash \sigma' : \Upsilon' <: \Upsilon''}{\Sigma \vdash \sigma' \circ \sigma : \Upsilon <: \Upsilon''}\ (<:_3) \qquad \frac{\Gamma';\Sigma' \vdash t : B \quad \Sigma \vdash \sigma : Y <: \Upsilon_0, A, \Upsilon_1}{\Gamma;\Sigma \vdash t : B}\ (<:_{\mathtt{split}})$$

where $\Gamma' = \Gamma[(Y_0^{\sigma(n)}, A, Y_1)/Y]$, $\Sigma' = \Sigma[(Y_0^{\sigma(n)}, A, Y_1)/X]$, and $Y_0^{\sigma(n)}, Y_1$ are fresh variables. Observe that the second one is a tautology that we only used to avoid the heavy syntactical manipulation of vectors of variables within proof trees.

Lemma 6.43 (Weakening). *The following rules are admissible:*

$$\frac{\Gamma;\Sigma \vdash t : A \quad \Sigma \subseteq \Sigma'}{\Gamma;\Sigma' \vdash t : A}\ (\Gamma_w) \qquad \frac{\Gamma;\Sigma \vdash t : A \quad \Gamma \subseteq \Gamma'}{\Gamma';\Sigma \vdash t : A}\ (\Sigma_w)$$

Proof. Easy induction on typing derivations. In the case of second-order quantification, we might need to rename the second-order variable X if it occurs in Σ' (resp.Γ') and not in Σ (resp. Γ). □

6.4.4 A typed CPS translation with De Bruijn levels

We shall now present the translation of terms and prove its correctness with respect to types. The translation, which is given in Figure 6.12, is similar to the translation with names in Section 8.3 plus the manipulation of coercions. Once again, we assume that for each constant $\boldsymbol{k}$ of type A (resp. co-constant $\boldsymbol{\kappa}$ of type $A^{\perp\!\!\perp}$) of the source system, we have a constant of type A in the signature of the target language that we also denote by $\boldsymbol{k}$ (resp. $\boldsymbol{\kappa}$ of type $A \to \bot$). We will now prove a bunch of lemmas that will be useful in the proof of the main theorem.

First, we show that the type of the store expected through the translation can be weakened. This is a sanity-check reflecting the usual weakening in the source language.

Lemma 6.44. *The following rule is admissible for any level o of the hierarchy e,t,E,V,F,v:*

$$\frac{\Gamma;\Sigma \vdash t : \Upsilon \triangleright_o A}{\Gamma;\Sigma \vdash t : \Upsilon, B \triangleright_o A}$$

Proof. Directly follows from the observation that we can always derive:

$$\frac{\Sigma \vdash \sigma : \Upsilon' <: \Upsilon, B}{\Sigma \vdash \sigma : \Upsilon' <: \Upsilon}$$

□

Then we show that the bounded quantification can be composed with subtyping relation witnessed by a coercion, by means of a lifting on the term accordingly with the coercion.

Lemma 6.45. *The following rules is admissible:*

$$\frac{\Gamma;\Sigma \vdash t : \forall Y <: \Upsilon_0.A \quad \Sigma \vdash \sigma : \Upsilon_1 <: \Upsilon_0}{\Gamma;\Sigma \vdash (\uparrow^{\sigma} t) : \forall Y <: \Upsilon_1.A}$$

$$
\begin{aligned}
&(\uparrow^{\sigma} t)\ \sigma' && \triangleq\ t\ (\sigma' \circ \sigma)\\
&(\uparrow^{\sigma} \tau[t]) && \triangleq\ (\uparrow^{\sigma} \tau)[\uparrow^{\sigma} t]\\
\\
&\llbracket \boldsymbol{k} \rrbracket_v && \triangleq\ \boldsymbol{k}\\
&\llbracket \lambda x_i.t \rrbracket_v\ \sigma\ \tau\ u\ E && \triangleq\ \llbracket t \rrbracket_t\ \sigma^{+}_{|\tau|}\ \tau[u]\ E\\
\\
&\llbracket \boldsymbol{\kappa} \rrbracket_F && \triangleq\ \boldsymbol{\kappa}\\
&\llbracket t \cdot E \rrbracket_F\ \sigma\ \tau\ v && \triangleq\ v\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^{\sigma} \llbracket t \rrbracket_t)\ (\uparrow^{\sigma} \llbracket E \rrbracket_E)\\
\\
&\llbracket v \rrbracket_V\ \sigma\ \tau\ F && \triangleq\ F\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^{\sigma} \llbracket v \rrbracket_v)\\
&\llbracket x_i \rrbracket_V\ \sigma\ \tau[t]\tau'\ F && \triangleq\ t\ \mathtt{id}_{|\tau|}\ \tau\ (\lambda \sigma' \tau'' \lambda V.V\ \tau''[\uparrow^{t} V](\uparrow^{\sigma''} \tau')\ (\uparrow^{\sigma''} F))\\
& && \text{where } n = |\tau| = \sigma(i),\ k = |\tau''| - n,\ p = n + |\tau'|,\ \sigma'' = \sigma' \circ \delta^{+k}_{[n,p]}\\
& && \ \text{and } \uparrow^{t} V = \lambda \sigma \tau E.E\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^{\sigma} V)\\
\\
&\llbracket \alpha_i \rrbracket_E\ \sigma\ \tau\ V && \triangleq\ \mathtt{let}\ \tau', x, \tau'' = \mathtt{split}\ \ \mathtt{as}\,(\sigma)\ \mathtt{in}\,(i)\,\tau\ \mathtt{in}\,x\ \mathtt{id}_{|\tau|}\ \tau\ V\\
&\llbracket \tilde{\mu}[x_i].\langle x_i \| F \rangle \tau' \rrbracket_E\ \sigma\ \tau\ V && \triangleq\ V\ \mathtt{id}_{|\tau|}\ \tau[\uparrow^{t} V](\uparrow^{\sigma'} \llbracket \tau' \rrbracket_\tau)\ (\uparrow^{\sigma'} \llbracket F \rrbracket_F)\\
& && \text{where } n = |\tau|,\ k = n - i,\ p = n + |\tau'|,\ \sigma' = \sigma \circ \delta^{+k}_{[i,p]}\\
\\
&\llbracket V \rrbracket_t\ \sigma\ \tau\ E && \triangleq\ E\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^{\sigma} \llbracket V \rrbracket_V)\\
&\llbracket \mu \alpha_i.c \rrbracket_t\ \sigma\ \tau\ E && \triangleq\ \llbracket c \rrbracket_c\ \sigma^{+}_{|\tau|}\ \tau[E]\\
\\
&\llbracket E \rrbracket_e\ \sigma\ \tau\ t && \triangleq\ t\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^{\sigma} \llbracket E \rrbracket_E)\\
&\llbracket \tilde{\mu} x_i.c \rrbracket_e\ \sigma\ \tau\ t && \triangleq\ \llbracket c \rrbracket_c\ \sigma^{+}_{|\tau|}\ \tau[t]\\
\\
&\llbracket \langle t \| e \rangle \rrbracket_c\ \sigma\ \tau && \triangleq\ \llbracket e \rrbracket_e\ \sigma\ \tau\ (\uparrow^{\sigma} \llbracket t \rrbracket_t)\\
&\llbracket c\tau \rrbracket^{n}_{l}\ \sigma\ \tau' && \triangleq\ \llbracket c \rrbracket_c\ \sigma'\ \tau'(\uparrow^{\sigma'} \llbracket \tau \rrbracket_\tau)\\
& && \text{where } k = |\tau'| - n,\ p = n + |\tau|,\ \sigma' = \sigma \circ \delta^{+k}_{[n,p]}\\
\\
&\llbracket \varepsilon \rrbracket_\tau && \triangleq\ \varepsilon\\
&\llbracket \tau_0[x_i := t] \rrbracket_\tau && \triangleq\ \llbracket \tau_0 \rrbracket_\tau[\llbracket t \rrbracket_t]\\
&\llbracket \tau_0[\alpha_i := E] \rrbracket_\tau && \triangleq\ \llbracket \tau_0 \rrbracket_\tau[\llbracket E \rrbracket_E]\\
\\
&\delta^{+i}_{[n,p]} && \triangleq\ \begin{cases} j \mapsto j + i & \text{if } n \le j < p\\ j \mapsto j & \text{if } j < n \end{cases}
\end{aligned}
$$

Figure 6.12: Translation of terms

Proof. We assume that the variable X is not $FV(\Gamma,\Sigma)$, otherwise it suffices to rename it. Unfolding the definition of $\uparrow^{\sigma} t$, we can derive:

$$\dfrac{\dfrac{\dfrac{\Gamma;\Sigma \vdash t : \forall X <: \Upsilon_0.A}{\Gamma;\Sigma,\sigma' : X <: \Upsilon_1 \vdash t : \forall X <: \Upsilon_0.A} \quad \dfrac{\Sigma \vdash \sigma : \Upsilon' <: \Upsilon_1 \quad \Sigma,\sigma' : X <: \Upsilon_1 \vdash \sigma' : X <: \Upsilon_1}{\Sigma,\sigma' : X <: \Upsilon_1 \vdash \sigma' \circ \sigma : X <: \Upsilon_0}}{\Gamma;\Sigma,\sigma' : X <: \Upsilon_1 \vdash t\ (\sigma' \circ \sigma) : A} \qquad X \notin FV(\Gamma,\Sigma)}{\Gamma;\Sigma \vdash \lambda\sigma'.t\ (\sigma' \circ \sigma) : \forall X <: \Upsilon_1.A}$$

where we use Lemma 6.43 to weaken $\Sigma,\sigma : X <: \Upsilon_1$. □

We deduce from the former lemma the following corollary that will be crucial when typing the translation of terms.

Corollary 6.46. *For any level o of the hierarchy e,t,E,V,F,v, the following rule are admissible:*

$$\dfrac{\Gamma;\Sigma \vdash t : \Upsilon_0 \triangleright_o A \quad \Sigma \vdash \sigma : \Upsilon_1 <: \Upsilon_0}{\Gamma;\Sigma \vdash (\uparrow^{\sigma} t) : \Upsilon_1 \triangleright_o A} \qquad \dfrac{\Gamma;\Sigma \vdash \tau : \Upsilon_0 \triangleright_\tau \Upsilon \quad \Sigma \vdash \sigma : \Upsilon_1\Upsilon <: \Upsilon_0\Upsilon}{\Gamma;\Sigma \vdash (\uparrow^{\sigma} \tau) : \Upsilon_1 \triangleright_\tau \Upsilon}$$

The following lemma shows that the operation of lifting values to terms is sound with respect to the translation of types.

Lemma 6.47 (Lifting values)**.** *The following rule is admissible:*

$$\dfrac{\Gamma;\Sigma \vdash V : \Upsilon \triangleright_V A}{\Gamma;\Sigma \vdash \uparrow^{t} V : \Upsilon \triangleright_t A}\ (\uparrow)$$

Proof.

$$\dfrac{\dfrac{\Pi_E \qquad \dfrac{\Gamma;\Sigma \vdash V : \Upsilon \triangleright_V A \quad \dfrac{}{\sigma : Y <: \Upsilon \vdash \sigma : Y <: \Upsilon}\ (<:_{\mathsf{ax}})}{\Gamma;\Sigma,\sigma : Y <: \Upsilon \vdash \uparrow^{\sigma} V : Y \triangleright_V A}}{\Gamma,\tau : Y, E : \Upsilon \triangleright_E A;\Sigma;\sigma : Y <: \Upsilon \vdash E\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^{\sigma} V) : \bot}\ (@)}{\Gamma;\Sigma \vdash \lambda\sigma\tau E.E\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^{\sigma} V) : \Upsilon \triangleright_t A}\ (\lambda)$$

where we used Corollary 6.46 and Π_E is the following derivation:

$$\dfrac{\dfrac{\dfrac{}{E : \Upsilon \triangleright_E A; \vdash E : Y \triangleright_E A \to \bot}\ (\mathrm{Ax}) \quad \dfrac{}{\vdash \mathtt{id}_{|\tau|} : Y <: Y}\ (<:_{\mathsf{ax}})}{E : \Upsilon \triangleright_E A; \vdash E\ \mathtt{id}_{|\tau|} : Y \to Y \triangleright_V A \to \bot}\ (\forall_E) \quad \dfrac{}{\tau : Y; \vdash \tau : Y}\ (\mathrm{Ax})}{\tau : Y, E : \Upsilon \triangleright_E A; \vdash E\ \mathtt{id}_{|\tau|}\ \tau : Y \triangleright_V A \to \bot}\ (@)$$

□

We now prove the soundness of the rules for forming stores through the translation.

Lemma 6.48 (Store formation)**.** *The following rules are admissible:*

$$\dfrac{\Gamma;\Sigma \vdash \tau : \Upsilon \quad \Gamma;\Sigma \vdash t : \Upsilon \triangleright_t A}{\Gamma;\Sigma \vdash \tau[t] : \Upsilon, A} \qquad \dfrac{\Sigma \vdash \sigma : \Upsilon <: [\![\Gamma_0]\!]}{\Sigma \vdash \sigma^{+}_{|\Upsilon|} : (\Upsilon, A) <: [\![\Gamma_0, A]\!]}$$

The same holds for $\Gamma \vdash E : \Upsilon \triangleright_E \iota(A)$ *and* $\Gamma \vdash \tau[E] : \Upsilon, A^{\bot\!\!\bot}$.

Proof. The left rule is a straightforward application of $(\tau\tau')$- and (τ_t)-rules:

$$\dfrac{\Gamma;\Sigma \vdash \tau[t] : Y, A \quad \dfrac{\Gamma;\Sigma \vdash t : Y \triangleright_t \iota(A)}{\Gamma;\Sigma \vdash [t] : Y \triangleright_\tau \iota(A)^{\bot\!\!\bot}}\ (\tau_t)}{\Gamma;\Sigma \vdash \tau[t] : Y, A}\ (\tau\tau')$$

The right one is a reformulation of Lemma 6.39. □

Similarly, we can prove that the shifts accordingly to a coercion are sound with respect to types:

Lemma 6.49 (Shifts). *For any $\Upsilon_0, \Upsilon_0', \Upsilon_1$, if $\sigma : \Upsilon_0' <: \Upsilon_0$ and $n = |\Upsilon_0|, p = n + |\Upsilon_1|$, $k = [\Upsilon_0'| - |\Upsilon_0|$, if we define $\sigma' = \sigma \circ \delta^{+k}_{[n,p]}$ then $\sigma' : (\Upsilon_0'\Upsilon_1) <: (\Upsilon_0\Upsilon_1)$.*
In particular, the following rules are admissible for any level o:

$$\frac{\Gamma;\Sigma \vdash t : \Upsilon_0\Upsilon_1 \triangleright_o A \quad \Sigma \vdash \sigma : \Upsilon_0' <: \Upsilon_0}{\Gamma;\Sigma \vdash (\uparrow^{\sigma'} t) : \Upsilon_0'\Upsilon_1 \triangleright_o A} \qquad \frac{\Gamma;\Sigma \vdash \tau : \Upsilon_0 \triangleright_\tau \Upsilon_1 \quad \Sigma \vdash \sigma : \Upsilon_0' <: \Upsilon_0}{\Gamma;\Sigma \vdash (\uparrow^{\sigma'} \tau) : \Upsilon_0' \triangleright_\tau \Upsilon_1}$$

Proof. We denote by $\Upsilon(i)$ the i^{th}-element of the list Υ. By definition, we have:

$$\sigma'(i) = \begin{cases} i+k & \text{if } n \leq i < p \\ \sigma'(i) & \text{if } j < n \end{cases}$$

We have:

$$\begin{array}{rcll} (\Upsilon_0'\Upsilon_1)(\sigma'(i)) & = & \Upsilon_0'(\sigma'(i)) = \Upsilon_0'(\sigma(i)) = \Upsilon_0(i) & (\text{if } i < n) \\ (\Upsilon_0'\Upsilon_1)(\sigma'(i)) & = & (\Upsilon_0'\Upsilon_1)(i+k) = \Upsilon_1(i+k-|\Upsilon_0'|) = \Upsilon_1(i-|\Upsilon_0|) = (\Upsilon_0\Upsilon_1)(i) & (\text{otherwise}) \end{array}$$

Thus we can conlude $\sigma' : (\Upsilon_0'\Upsilon_1) <: (\Upsilon_0\Upsilon_1)$. □

We are finally equipped to prove the main theorem of this section, that is the correctness of the translation with respect to types.

Theorem 6.50. *The translation is well-typed, i.e.*

1. *if* $\Gamma \vdash_v v : A$ *then* $[\![\Gamma \vdash_v v : A]\!]$
2. *if* $\Gamma \vdash_F F : A^{\perp\!\!\perp}$ *then* $[\![\Gamma \vdash_F F : A^{\perp\!\!\perp}]\!]$
3. *if* $\Gamma \vdash_V V : A$ *then* $[\![\Gamma \vdash_V V : A]\!]$
4. *if* $\Gamma \vdash_E E : A^{\perp\!\!\perp}$ *then* $[\![\Gamma \vdash_E E : A^{\perp\!\!\perp}]\!]$
5. *if* $\Gamma \vdash_t t : A$ *then* $[\![\Gamma \vdash_t t : A]\!]$
6. *if* $\Gamma \vdash_e e : A^{\perp\!\!\perp}$ *then* $[\![\Gamma \vdash_e e : A^{\perp\!\!\perp}]\!]$
7. *if* $\Gamma \vdash_c c$ *then* $[\![\Gamma \vdash_c c]\!]$
8. *if* $\Gamma \vdash_l l$ *then* $[\![\Gamma \vdash_l l]\!]$
9. *if* $\Gamma \vdash_\tau \tau$ *then* $[\![\Gamma \vdash_\tau \tau : \Gamma']\!]$

Proof. The proof is almost the same as the proof of Theorem 6.33, using the previous lemmas. We reason by induction over the typing rules of Figure 6.9. We (ab)use of Lemma 6.43 to make the derivations more compact by systematically weakening contexts as soon as possible, and compact the first $(\forall_I)$ and (λ) rules in one rule.

1. Strong values

- **Case** $[\![\boldsymbol{k}]\!]_v$. $[\![\boldsymbol{k}]\!]_v = \boldsymbol{k}$, which has the desired type by hypothesis.

- **Case** $\lambda x_i.t$. In the source language, we have:

$$\frac{\Gamma, x_i : A \vdash_t t : B \quad |\Gamma| = i}{\Gamma \vdash_v \lambda x_i : A \to B}$$

Hence, we get by induction a proof Π_t of $[\![t]\!]_t : [\![\Gamma, x_i : A]\!] \triangleright_t \iota(B)$ and we can derive:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\Pi_t}{\vdash [\![t]\!]_t : \forall Y' <: [\![\Gamma, x_i : A]\!].Y' \to Y' \triangleright_E \iota(B) \to \bot} \quad \boldsymbol{\Pi_\sigma}}{;\sigma : Y <: [\![\Gamma]\!] \vdash [\![t]\!]_t\ \sigma^+_{|\tau|} : (Y,\iota(A)) \to (Y,\iota(A)) \triangleright_E \iota(B) \to \bot}\,(\forall_E) \quad \boldsymbol{\Pi_\tau}}{\tau : Y, u : Y \triangleright_t \iota(A); \sigma : Y <: [\![\Gamma]\!] \vdash [\![t]\!]_t\ \sigma^+_{|\tau|}\ \tau[u] : (Y,\iota(A)) \triangleright_E \iota(B) \to \bot}\,(@) \quad \boldsymbol{\Pi_E}}{\tau : Y, u : Y \triangleright_t \iota(A), E : Y \triangleright_E \iota(B); \sigma : Y <: [\![\Gamma]\!] \vdash [\![t]\!]_t\ \sigma^+_{|\tau|}\ \tau[u]\ E^+ : \bot}\,(@)}{\vdash \lambda\sigma\tau u E.[\![t]\!]_t\ \sigma^+_{|\tau|}\ \tau[u]\ E : \forall Y <: [\![\Gamma]\!].Y \to Y \triangleright_t \iota(A) \to Y \triangleright_E \iota(B) \to \bot}\,(\lambda)$$

where:

- Π_E is a proof of $E : Y \triangleright_E \iota(B) \vdash E : (Y, \iota(A)) \triangleright_E \iota(B)$ (derivable according to Lemma 6.44);
- Π_τ is a proof of $\tau : Y, u : Y \triangleright_t \iota(A); \vdash \tau[u] : Y, \iota(A)$ (derivable according to Lemma 6.48);
- Π_σ is obtained by Lemma 6.48:

$$\dfrac{\dfrac{}{\sigma : Y <: \llbracket \Gamma \rrbracket \vdash \sigma : Y <: \llbracket \Gamma \rrbracket}\,(<:_{\mathrm{ax}})}{\sigma : Y <: \llbracket \Gamma \rrbracket \vdash \sigma^+_{|\tau|} : (Y, \iota(A)) <: \llbracket \Gamma, x_i : A \rrbracket}$$

2. Forcing contexts

- **Case** $\llbracket \boldsymbol{\kappa} \rrbracket_F$. $\llbracket \boldsymbol{\kappa} \rrbracket_F = \boldsymbol{\kappa}$, which has the desired type by hypothesis.

- **Case** $\llbracket t.E \rrbracket_F$. In the source language, we have:

$$\dfrac{\Gamma \vdash_t t : A \quad \Gamma \vdash_E E : B^{\perp\!\!\perp}}{\Gamma \vdash_F t \cdot E : (A \to B)^{\perp\!\!\perp}}$$

Therefore we have by induction hypothesis that $\vdash \llbracket t \rrbracket_t : \llbracket \Gamma \rrbracket_\Gamma \triangleright_t \iota(A)$ and $\vdash \llbracket E \rrbracket_t : \llbracket \Gamma \rrbracket_\Gamma \triangleright_E \iota(B)$, so that we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{v : Y \triangleright_v \iota(A) \to \iota(B); \vdash v : \forall Y' <: Y : Y' \to Y' \triangleright_t \iota(A) \to Y' \triangleright_E \iota(B) \to \bot}\,(\mathrm{Ax}) \quad \boldsymbol{\Pi_\sigma}}{\tau : Y, v : Y \triangleright_v \iota(A) \to \iota(B); \vdash v\ \mathtt{id}_{|\tau|} : Y \to Y \triangleright_t \iota(A) \to Y \triangleright_E B \to \bot}\,(\forall_I) \quad \boldsymbol{\Pi_\tau}}{\tau : Y, v : Y \triangleright_v \iota(A) \to \iota(B); \vdash v\ \mathtt{id}_{|\tau|}\ \tau\ : Y \triangleright_t \iota(A) \to Y \triangleright_E \iota(B) \to \bot}\,(@) \quad \boldsymbol{\Pi_t}}{\tau : Y, v : Y \triangleright_v \iota(A) \to \iota(B); \sigma : Y <: \llbracket \Gamma \rrbracket \vdash v\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma \llbracket t \rrbracket_t) : Y \triangleright_E \iota(B) \to \bot}\,(@) \quad \boldsymbol{\Pi_E}}{\tau : Y, v : Y \triangleright_v \iota(A) \to \iota(B); \sigma : Y <: \llbracket \Gamma \rrbracket \vdash v\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma \llbracket t \rrbracket_t)\ (\uparrow^\sigma \llbracket E \rrbracket_E) : \bot}\,(@)}{\vdash \lambda \sigma \tau v. v\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma \llbracket t \rrbracket_t)\ (\uparrow^\sigma \llbracket E \rrbracket_E) : \forall Y <: \llbracket \Gamma \rrbracket_\Gamma . Y \to Y \triangleright_v \iota(A) \to \iota(B) \to \bot}\,(\lambda)$$

where:

- Π_E is a proof of $\varepsilon; \sigma : Y <: \llbracket \Gamma \rrbracket \vdash (\uparrow^\sigma \llbracket E \rrbracket_E) : Y \triangleright_E \iota(B)$, derived from the induction hypothesis for t and Corollary 6.46;
- Π_t is a proof of $\varepsilon; \sigma : Y <: \llbracket \Gamma \rrbracket \vdash (\uparrow^\sigma \llbracket t \rrbracket_t) : Y \triangleright_t \iota(A)$, derived from the induction hypothesis for E and Corollary 6.46;
- Π_τ is the axiom rule $\tau : Y; \vdash \tau : Y$;
- Π_σ is a proof of $\mathtt{id}_{|\tau|} : Y <: Y$ (Proposition 6.41).

3. Weak values

- **Case** $\llbracket v \rrbracket_V$. In the source language, we have:

$$\dfrac{\Gamma \vdash_v v : A}{\Gamma \vdash_V v : A}$$

Hence we have by induction hypothesis that $\vdash \llbracket v \rrbracket_v : \llbracket \Gamma \rrbracket_\Gamma \triangleright_v \iota(A)$ and we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{F : Y \triangleright_F \iota(A) \vdash F : \forall Y' <: Y.Y' \to Y' \triangleright_v \iota(A) \to \bot \quad \Pi_Y}{F : Y \triangleright_F \iota(A); \sigma : Y <: \llbracket \Gamma \rrbracket \vdash F\ \mathtt{id}_{|\tau|} : Y \to Y \triangleright_v \iota(A) \to \bot}\,(@) \quad \dfrac{}{\tau : Y; \vdash \tau : Y}}{\tau : Y, F : Y \triangleright_F \iota(A) \vdash F\ \mathtt{id}_{|\tau|}\ \tau : Y \triangleright_v \iota(A) \to \bot}\,(@) \quad \Pi_v}{\tau : Y, F : Y \triangleright_F \iota(A); \sigma : Y <: \llbracket \Gamma \rrbracket \vdash F\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^{\sigma} \llbracket v \rrbracket_v) : \bot}\,(@)}{\vdash \lambda \sigma \tau F.F\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^{\sigma} \llbracket v \rrbracket_v) : \forall Y <: \llbracket \Gamma \rrbracket.Y \to Y \triangleright_F \iota(A) \to \bot}\,(\lambda)$$

where:

  - Π_v is a proof of $\varepsilon; \sigma : Y <: \llbracket \Gamma \rrbracket \vdash (\uparrow^{\sigma} \llbracket v \rrbracket_v) : Y \triangleright_v \iota(A)$, derivable from the induction hypothesis and Corollary 6.46.
  - Π_τ is the axiom rule $\tau : Y; \vdash \tau : Y$
  - Π_Y is a proof of $\mathtt{id}_{|\tau|} : Y <: Y$ (Proposition 6.41)

- **Case** $\llbracket x_i \rrbracket_V$. In the source language, we have:

$$\dfrac{\Gamma(i) = (x_i : A)}{\Gamma \vdash_V x_i : A}$$

so that Γ is of the form $\Gamma', x_i : A, \Gamma''$. By definition, we have:

$$\llbracket x_i \rrbracket_V = \lambda \sigma \tau F.\,\mathtt{let}\,\tau_0, t, \tau_1 = \mathtt{split}\ n\,\tau\,\mathtt{in}\,t\ \mathtt{id}_n\ \tau_0\ (\lambda \sigma' \tau_0' \lambda V.V\ \tau''[\uparrow^t\ V](\uparrow^{\sigma''} \tau_1)\ (\uparrow^{\sigma''} F))$$

where $n = \sigma(i)$,$k = |\tau_0| - n$, $p = n + |\tau_1|$, $\sigma'' = \sigma' \circ \delta^{+k}_{[n,p]}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{t : Y_0^n \triangleright_t \iota(A) \vdash t : Y_0^n \triangleright_t \iota(A)}\,(\mathrm{Ax}) \quad \dfrac{}{\vdash \mathtt{id}_n : Y_0^n <: Y_0^n}\,(<:_{\mathrm{ax}})}{t : Y_0^n \triangleright_t \iota(A); \vdash t\ \mathtt{id}_n : Y_0^n \to Y_0^n \triangleright_E \iota(A) \to \bot}\,(\forall_E) \quad \dfrac{}{\tau_0 : Y_0^n \vdash \tau_0 : Y_0^n}\,(\mathrm{Ax})}{\tau_0 : Y_0^n, t : Y_0^n \triangleright_t A; \vdash t\ \mathtt{id}_n\ \tau_0 : Y_0^n \triangleright_E \iota(A) \to \bot}\,(@) \quad \mathbf{\Pi}_E}{\tau_0 : Y_0^n, t : Y_0^n \triangleright_t \iota(A), \tau_1 : (Y_0^n, n : \iota(A)) \triangleright_\tau Y_1, F : (Y_0^n, n : \iota(A), Y_1) \triangleright_F \iota(A); \vdash t\ \mathtt{id}_n\ \tau_0\ E : \bot}\,(@) \quad |Y_0^n| = n}{\tau : (Y_0^n, n : \iota(A), Y_1), F : (Y_0^n, n : \iota(A), Y_1) \triangleright_F \iota(A); \vdash \mathtt{let}\,\tau_0, t, \tau_1 = \mathtt{split}\ \tau\, n\,\mathtt{in}\,t\ \mathtt{id}_n\ \tau_0\ E : \bot}\,(\mathtt{split}) \quad \mathbf{\Pi}_\sigma}{\tau : Y, F : Y \triangleright_F \iota(A); \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma \vdash \mathtt{let}\,\tau_0, t, \tau_1 = \mathtt{split}\ \tau\, n\,\mathtt{in}\,t\ \mathtt{id}_n\ \tau_0\ E : \bot}\,(<:_{\mathtt{split}})}{\vdash \lambda \sigma \tau F.\,\mathtt{let}\,\tau_0, t, \tau_1 = \mathtt{split}\ \tau\, n\,\mathtt{in}\,t\ \mathtt{id}_n\ \tau_0\ E : \forall Y <: \llbracket \Gamma \rrbracket_\Gamma.Y \to Y \triangleright_F \iota(A) \to \bot}\,(\lambda)$$

where:

  - Π_σ is simply the axiom rule:

$$\dfrac{}{\sigma : Y <: (\llbracket \Gamma_0 \rrbracket_\Gamma, n : \iota(A), \llbracket \Gamma_1 \rrbracket_\Gamma) \vdash \sigma : Y <: (\llbracket \Gamma_0 \rrbracket_\Gamma, n : \iota(A), \llbracket \Gamma_1 \rrbracket_\Gamma)}\,(<:_{\mathrm{ax}})$$

  - $E = \lambda \sigma' \tau'' \lambda V.V\ \tau_0'[\uparrow^t\ V](\uparrow^{\sigma''} \tau_1)\ (\uparrow^{\sigma''} F))$ and Π_E is the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{V : Y_0' \triangleright_V \iota(A); \vdash \uparrow^t V : Y_0' \triangleright_t \iota(A)}\,(\mathrm{Ax}) \quad \dfrac{}{\vdash \mathtt{id}_p : Y_0', A, Y_1 <: Y_0', A, Y_1}}{V : Y_0' \triangleright_V \iota(A); \vdash V\ \mathtt{id}_p : (Y_0', \iota(A), Y_1) \to (Y_0', \iota(A), Y_1) \triangleright_F \iota(A) \to \bot}\,(\forall_E) \quad \mathbf{\Pi}_\tau}{\tau_1 : (Y_0^n, \iota(A)) \triangleright_\tau Y_1, \tau_0' : Y_0', V : Y_0' \triangleright_V \iota(A); \vdash V\ \mathtt{id}_p\ \tau_0'[\uparrow^t V](\uparrow^{\sigma'} \tau_1) : (Y_0', \iota(A), Y_1) \triangleright_F \iota(A) \to \bot}\,(@) \quad \mathbf{\Pi}_F}{\Gamma, \tau_0' : Y_0', V : Y_0' \triangleright_V \iota(A); \sigma' : Y_0' <: Y_0^n \vdash V\ \mathtt{id}_p\ \tau_0'[\uparrow^t V](\uparrow^{\sigma''} \tau')\ (\uparrow^{\sigma''} F) : \bot}\,(@)}{\Gamma \vdash \lambda \sigma' \tau_0' V.V\ \mathtt{id}_p\ \tau_0'[\uparrow^t V](\uparrow^{\sigma''} \tau_1)\ (\uparrow^{\sigma''} F) : Y_0^n \triangleright_F \iota(A)}\,(\lambda)$$

where $\Gamma = \tau_1 : (Y_0^n, \iota(A)) \triangleright_\tau Y_1, F : (Y_0^n, \iota(A), Y_1) \triangleright_F \iota(A)$.

- Π_F is the following proof, obtained by Lemma 6.49:

$$\dfrac{\dfrac{}{F : (Y_0^n,\iota(A),Y_1) \triangleright_F \iota(A); \vdash F : (Y_0^n,\iota(A),Y_1) \triangleright_F \iota(A)}\,(\text{Ax}) \qquad \dfrac{}{\sigma' : Y_0' <: Y_0^n \vdash \sigma' : Y_0' <: Y_0^n}\,(\text{Ax})}{F : (Y_0^n,\iota(A),Y_1) \triangleright_F \iota(A); \sigma_1 : Y_0' <: Y_0^n \vdash (\uparrow^{\sigma''} F) : (Y_0',\iota(A),Y_1) \triangleright_F \iota(A)}$$

- Π_τ is the following derivation

$$\dfrac{\dfrac{\dfrac{}{\tau_0' : Y_0' \vdash \tau_0' : Y_0'}\,(\text{Ax}) \qquad \dfrac{\dfrac{\dfrac{}{V : Y_0' \triangleright_V \iota(A) \vdash V : Y_0' \triangleright_V A}\,(\text{Ax})}{V : Y_0' \triangleright_V \iota(A) \vdash \uparrow^t V : Y_0' \triangleright_t A}\,(\uparrow)}{V : Y_0' \triangleright_V \iota(A) \vdash [\uparrow^t V] : Y_0' \triangleright_\tau \iota(A)}\,(\tau_t)}{Y_0' <: Y_0^n, \tau_0' : Y_0', V : Y_0' \triangleright_V \iota(A) \vdash \tau_0'[\uparrow^t V] : Y_0', n : \iota(A)}\,(\tau\tau') \qquad \mathbf{\Pi}_{\tau_1}}{\tau_1 : (Y_0^n,\iota(A)) \triangleright_\tau Y_1, Y_0' <: Y_0^n, \tau_0' : Y_0', V : Y_0' \triangleright_V \iota(A) \vdash \tau_0'[V](\uparrow^{\sigma''} \tau_1) : Y_0', \iota(A), Y_1}\,(\tau_{<:})$$

- Π_{τ_1} is obtained by Lemma 6.49:

$$\dfrac{\dfrac{}{\tau_1 : (Y_0, n : \iota(A)) \triangleright_\tau Y_1 \vdash \tau_1 : (Y_0,\iota(A)) \triangleright_\tau Y_1}\,(\text{Ax}) \qquad \dfrac{}{\sigma' : Y_0' <: Y_0^n \vdash \sigma' : Y_0' <: Y_0^n}\,(<:_{\mathsf{ax}})}{\tau_1 : (Y_0^n,\iota(A)) \triangleright_\tau Y_1; \sigma' : Y_0' <: Y_0 \vdash (\uparrow^{\sigma''} \tau_1) : Y_0', n : \iota(A) \triangleright_\tau Y_1}$$

4. Catchable contexts

- **Case** $\llbracket F \rrbracket_E$. This case is similar to the case $\llbracket v \rrbracket_V$.

- **Case** $\llbracket \tilde{\mu}[x_i].\langle x_i \| F \rangle \tau' \rrbracket_E$. In the source language, we have:

$$\dfrac{\Gamma, x_i : A, \Gamma' \vdash_F F : A^{\perp\!\!\!\perp} \quad \Gamma, x_i : A \vdash_\tau \tau' : \Gamma' \quad |\Gamma| = i}{\Gamma \vdash_E \tilde{\mu}[x_i].\langle x_i \| F \rangle \tau' : A^{\perp\!\!\!\perp}}$$

We have by induction hypothesis a proof of $\vdash \llbracket \tau' \rrbracket_\tau : \llbracket \Gamma, x_i : A \rrbracket_\Gamma \triangleright_\tau \llbracket \Gamma' \rrbracket_\Gamma$ and a proof Π_F of $\vdash \llbracket F \rrbracket_F : \llbracket \Gamma, x_i : A, \Gamma' \rrbracket_\Gamma \triangleright_F \iota(A)$. We can thus derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{V : Y \triangleright_V \iota(A); \vdash V : Y \triangleright_t \iota(A)}\,(\text{Ax}) \qquad \dfrac{}{\vdash \mathtt{id}_p : (Y, \iota(A), \llbracket \Gamma' \rrbracket_\Gamma) <: Y}}{V : Y \triangleright_V \iota(A); \vdash V\ \mathtt{id}_p : (Y, \iota(A), \llbracket \Gamma' \rrbracket_\Gamma) \to (Y, \iota(A), \llbracket \Gamma' \rrbracket_\Gamma) \triangleright_F \iota(A) \to \bot}\,(\forall_E) \qquad \mathbf{\Pi}_\tau}{\tau : Y, V : Y \triangleright_V \iota(A); \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma \vdash V\ \mathtt{id}_p\ \tau[\uparrow^t V](\uparrow^{\sigma'} \llbracket \tau' \rrbracket_\tau) : (Y, \iota(A), \llbracket \Gamma' \rrbracket_\Gamma) \triangleright_F \iota(A) \to \bot}\,(@) \qquad \mathbf{\Pi}_F}{\dfrac{\Gamma, \tau : Y, V : Y \triangleright_V \iota(A); \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma \vdash V\ \mathtt{id}_p\ \tau[\uparrow^t V](\uparrow^{\sigma'} \llbracket \tau' \rrbracket_\tau)\ (\uparrow^{\sigma'} \llbracket F \rrbracket_F) : \bot}{\Gamma \vdash \lambda \sigma \tau V. V\ \mathtt{id}_p\ \tau[\uparrow^t V](\uparrow^{\sigma'} \llbracket \tau' \rrbracket_\tau)\ (\uparrow^{\sigma'} \llbracket F \rrbracket_F) : \llbracket \Gamma \rrbracket_\Gamma \triangleright_F \iota(A)}\,(\lambda)}\,(@)$$

where:

- $n = |\tau|$, $k = n - i$, $p = n + |\tau'|$, $\sigma' = \sigma \circ \delta^{+k}_{[i,p]}$
- Π_F is the following proof, obtained by Lemma 6.49:

$$\dfrac{; \vdash F : (\llbracket \Gamma \rrbracket_\Gamma, \iota(A), \llbracket \Gamma' \rrbracket_\Gamma) \triangleright_F \iota(A) \qquad \dfrac{}{\sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma \vdash \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma}\,(\text{Ax})}{; \sigma : Y <: \llbracket \Gamma' \rrbracket_\Gamma \vdash (\uparrow^{\sigma'} F) : (Y, \iota(A), \llbracket \Gamma' \rrbracket_\Gamma) \triangleright_F \iota(A)}$$

- Π_τ is the following proof:

$$\dfrac{\dfrac{\dfrac{}{\tau : Y \vdash \tau : Y}\,(\text{Ax}) \qquad \dfrac{\dfrac{\dfrac{}{V : Y \triangleright_V \iota(A) \vdash V : Y \triangleright_V \iota(A)}\,(\text{Ax})}{V : Y \triangleright_V \iota(A) \vdash\uparrow^t V : Y \triangleright_t \iota(A)}\,(\uparrow)}{V : Y \triangleright_V \iota(A) \vdash [V] : Y \triangleright_\tau \iota(A)}\,(\tau_t)}{\tau : Y, V : Y \triangleright_V \iota(A) \vdash \tau[\uparrow^t V] : Y, \iota(A)}\,(\tau\tau') \qquad \mathbf{\Pi}_{\tau'}}{\tau : Y, V : Y' \triangleright_V \iota(A); \sigma : Y <: \llbracket\Gamma\rrbracket_\Gamma \vdash \tau[\uparrow^t V]\llbracket\tau'\rrbracket_\tau : (Y, \iota(A), \llbracket\Gamma'\rrbracket^{\sigma[x:=n]})}\,(\tau\tau')$$

- $\Pi_{\tau'}$ is the following proof, obtained from the induction hypothesis for τ' and Lemma 6.49:

$$\dfrac{\vdash \llbracket\tau'\rrbracket_\tau : \llbracket\Gamma\rrbracket_\Gamma, \iota(A) \triangleright_\tau \llbracket\Gamma'\rrbracket_\Gamma \qquad \dfrac{}{\sigma : Y <: \llbracket\Gamma\rrbracket_\Gamma \vdash \sigma : Y <: \llbracket\Gamma\rrbracket_\Gamma}\,(<:_{\mathsf{ax}})}{; \sigma : Y <: \llbracket\Gamma\rrbracket_\Gamma \vdash\uparrow^{\sigma'} \llbracket\tau'\rrbracket_\tau : Y, \iota(A) \triangleright_\tau \llbracket\Gamma'\rrbracket_\Gamma}$$

5. Terms

- **Case** $\llbracket V\rrbracket_t$. This case is similar to the case $\llbracket v\rrbracket_V$.

- **Case** $\llbracket \mu\alpha_i.c\rrbracket_t$. In the $\overline{\lambda}_{[lv\tau\star]}$-calculus, we have:

$$\dfrac{\Gamma, \alpha_i : A^{\perp\!\!\!\perp} \vdash_c c \quad |\Gamma| = i}{\Gamma \vdash_t \mu\alpha_i.c : A}$$

Hence we have by induction a proof of $; \vdash \llbracket c\rrbracket_c : \llbracket\Gamma, x_i : A^{\perp\!\!\!\perp}\rrbracket_\Gamma \triangleright_c \bot$ and we can derive:

$$\dfrac{\dfrac{\dfrac{; \vdash \llbracket c\rrbracket_c : \llbracket\Gamma, x_i : A^{\perp\!\!\!\perp}\rrbracket_\Gamma \triangleright_c \bot \qquad \mathbf{\Pi}_\sigma}{\tau : Y; \sigma : Y <: \llbracket\Gamma\rrbracket_\Gamma \vdash \llbracket c\rrbracket_c\ \sigma^+_{|\tau|} : (Y, \iota(A)^{\perp\!\!\!\perp}) \to \bot}\,(\forall_E) \qquad \mathbf{\Pi}_\tau}{\tau : Y, E : Y \triangleright_E \iota(A); \sigma : Y <: \llbracket\Gamma\rrbracket_\Gamma \vdash \llbracket c\rrbracket_c\ \sigma^+_{|\tau|}\ \tau[E] : \bot}\,(@)}{; \vdash \lambda\sigma\tau E.\llbracket c\rrbracket_c\ \sigma^+_{|\tau|}\ \tau[E] : \llbracket\Gamma\rrbracket_\Gamma \triangleright_t \iota(A)}\,(\lambda)$$

where

- Π_σ is the following derivation, obtained by Lemma 6.48 (since $|\tau|$ matches $|Y|$):

$$\dfrac{\dfrac{}{\sigma : Y <: \llbracket\Gamma\rrbracket_\Gamma \vdash \sigma : Y <: \llbracket\Gamma\rrbracket_\Gamma}\,(<:_{\mathsf{ax}})}{\sigma : Y <: \llbracket\Gamma\rrbracket_\Gamma \vdash \sigma^+_{|\tau|} : (Y, \iota(A)^{\perp\!\!\!\perp}) <: \llbracket\Gamma, x_i : \iota(A)^{\perp\!\!\!\perp}\rrbracket_\Gamma}$$

- Π_E is also obtained by Lemma 6.48:

$$\dfrac{\dfrac{}{\tau : Y, E : Y \triangleright_E \iota(A); \vdash \tau[E] : Y, \iota(A)^{\perp\!\!\!\perp}}\,(\text{Ax}) \qquad \dfrac{}{E : Y \triangleright_E \iota(A); \vdash E : Y \triangleright_E \iota(A)}\,(\text{Ax})}{\tau : Y, E : Y \triangleright_E \iota(A); \vdash \tau[E] : Y, \iota(A)^{\perp\!\!\!\perp}}$$

6. Contexts

- **Case** $\llbracket E\rrbracket_e$. This case is similar to the case $\llbracket v\rrbracket_V$.

- **Case** $\llbracket \tilde{\mu}x_i.c\rrbracket_e$. This case is similar to the case $\llbracket \mu\alpha_i.c\rrbracket_t$.

7. Commands

- **Case** $\llbracket\langle t\|e\rangle\rrbracket_c$. In the $\overline{\lambda}_{[lv\tau\star]}$-calculus we have:

$$\frac{\Gamma\vdash_t t:A\quad \Gamma\vdash_e e:A^{\perp\!\!\perp}}{\Gamma\vdash_c\langle t\|e\rangle}$$

thus we get by induction two proofs of $;\vdash\llbracket t\rrbracket_t:\llbracket\Gamma\rrbracket_\Gamma\triangleright_t\iota(A)$ and $;\vdash\llbracket e\rrbracket_c:\llbracket\Gamma\rrbracket_\Gamma\triangleright_e\iota(A)$. We can then derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{;\vdash\llbracket e\rrbracket_e:\llbracket\Gamma\rrbracket_\Gamma\triangleright_e\iota(A)}{\tau:Y;\sigma:Y<:\llbracket\Gamma\rrbracket_\Gamma\vdash\llbracket e\rrbracket_e\;\sigma:Y\to Y\triangleright_t\iota(A)\to\bot}\,(\forall_E)\quad \Pi_\sigma}{\tau:Y;\sigma:Y<:\llbracket\Gamma\rrbracket_\Gamma\vdash\llbracket e\rrbracket_e\;\sigma\;\tau:Y\triangleright_t\iota(A)\to\bot}\,(@)\quad \dfrac{}{\tau:Y;\vdash\tau:Y}\,(\mathrm{Ax})}{\tau:Y;\sigma:Y<:\llbracket\Gamma\rrbracket_\Gamma\vdash\llbracket e\rrbracket_e\;\sigma\;\tau\;(\uparrow^\sigma\llbracket t\rrbracket_t):\bot}\,(@)\quad \Pi_t}{;\vdash\lambda\sigma\tau.\llbracket e\rrbracket_e\;\sigma\;\tau\;(\uparrow^\sigma\llbracket t\rrbracket_t):\llbracket\Gamma\rrbracket_\Gamma\triangleright_c\bot}\,(\lambda)$$

where:

- Π_σ is the axiom rule: $\dfrac{}{\sigma:Y<:\llbracket\Gamma\rrbracket_\Gamma\vdash\sigma:Y<:\llbracket\Gamma\rrbracket_\Gamma}\,(<:_{\mathsf{ax}})$
- Π_t is obtained using Lemma 6.45:

$$\dfrac{;\vdash\llbracket t\rrbracket_t:\llbracket\Gamma\rrbracket_\Gamma\triangleright_t\iota(A)\quad \dfrac{}{;\sigma:Y<:\llbracket\Gamma\rrbracket_\Gamma\vdash\sigma:Y<:\llbracket\Gamma\rrbracket_\Gamma}\,(<:_{\mathsf{ax}})}{;\sigma:Y<:\llbracket\Gamma\rrbracket_\Gamma\vdash\uparrow^\sigma\llbracket t\rrbracket_t:Y\triangleright_t\iota(A)}$$

8. Closures

- **Case** $\llbracket c\tau'\rrbracket_l^n$. In the $\overline{\lambda}_{[lv\tau\star]}$-calculus, we have:

$$\frac{\Gamma,\Gamma'\vdash_c c\quad \Gamma\vdash_\tau\tau':\Gamma'}{\Gamma\vdash_l c\tau'}$$

where n matches $|\Gamma|$. We thus get by induction two proofs $;\vdash\llbracket\tau'\rrbracket_\tau:\llbracket\Gamma\rrbracket_\Gamma\triangleright_\tau\llbracket\Gamma'\rrbracket_\Gamma$ and $\vdash\llbracket c\rrbracket_c:\llbracket\Gamma,\Gamma'\rrbracket_\Gamma\triangleright_c\bot$. We can derive:

$$\dfrac{\dfrac{\dfrac{\vdash\llbracket c\rrbracket_c:\llbracket\Gamma,\Gamma'\rrbracket_\Gamma\triangleright_c\bot\qquad\Pi_\sigma}{;\sigma:Y<:\llbracket\Gamma\rrbracket_\Gamma\vdash\llbracket c\rrbracket_c\;\sigma':(Y,\llbracket\Gamma'\rrbracket_\Gamma)\to\bot}\,(\forall_E)\quad \dfrac{\dfrac{}{\tau:Y;\vdash\tau:Y}\,(\mathrm{Ax})\qquad\Pi_\tau}{\tau:Y;\sigma:Y<:\llbracket\Gamma\rrbracket_\Gamma\vdash\tau(\uparrow^{\sigma'}\llbracket\tau'\rrbracket_\tau):Y\llbracket\Gamma'\rrbracket_\Gamma}\,(\tau\tau')}{\tau:Y;\sigma:Y<:\llbracket\Gamma\rrbracket_\Gamma\vdash\llbracket c\rrbracket_c\;\sigma'\;\tau'(\uparrow^{\sigma'}\llbracket\tau'\rrbracket_\tau:\bot}\,(@)}{;\vdash\lambda\sigma\tau.\llbracket c\rrbracket_c\;\sigma'\;\tau'(\uparrow^{\sigma'}\llbracket\tau'\rrbracket_\tau)}\,(\lambda)$$

where $k=|\tau'|-n$, $p=n+|\tau|$, $\sigma'=\sigma\circ\delta_{[n,p]}^{+k}$ and:

- Π_σ is a proof of $\sigma:Y<:\llbracket\Gamma\rrbracket_\Gamma\vdash\sigma':(Y,\llbracket\Gamma'\rrbracket_\Gamma)<:\llbracket\Gamma,\Gamma'\rrbracket_\Gamma$ obtained by Lemma 6.49;
- $\Pi_{\tau'}$ is the following proof also obtained by Lemma 6.49:

$$\dfrac{;\vdash\llbracket\tau'\rrbracket_\tau:\llbracket\Gamma\rrbracket_\Gamma\triangleright_\tau\llbracket\Gamma'\rrbracket_\Gamma\quad \dfrac{}{\vdash\sigma:Y<:\llbracket\Gamma\rrbracket_\Gamma}\,(<:_{\mathsf{ax}})}{\vdash(\uparrow^{\sigma'}\llbracket\tau'\rrbracket_\tau):Y\triangleright_\tau\llbracket\Gamma'\rrbracket_\Gamma}$$

9. Stores

- **Case** $\llbracket\tau[x_i:=t]\rrbracket_\tau$. We only consider the case $\tau[x_i:=t]$, the proof for the case $\tau[\alpha_i:=E]$ is identical. This corresponds to the typing rules:

$$\frac{\Gamma\vdash_\tau\tau:\Gamma'\qquad\Gamma,\Gamma'\vdash_t t:A\quad|\Gamma,\Gamma'|=i}{\Gamma\vdash_\tau\tau[x_i:=t]:\Gamma',x_i:A}$$

By induction, we obtain two proofs of $\vdash\llbracket\tau\rrbracket_\tau:\llbracket\Gamma\rrbracket_\Gamma\triangleright_\tau\llbracket\Gamma'\rrbracket_\Gamma$ and $\vdash\llbracket t\rrbracket_t:\llbracket\Gamma,\Gamma'\rrbracket_\Gamma\triangleright_t\iota(A)$. We can thus derive:

$$\dfrac{\vdash\llbracket\tau\rrbracket_\tau:\llbracket\Gamma\rrbracket_\Gamma\triangleright_\tau\llbracket\Gamma'\rrbracket_\Gamma\quad \dfrac{\vdash\llbracket t\rrbracket_t:\llbracket\Gamma,\Gamma'\rrbracket_\Gamma\triangleright_t\iota(A)}{\vdash[\llbracket t\rrbracket_t]:\llbracket\Gamma,\Gamma'\rrbracket_\Gamma\triangleright_\tau\iota(A)}\,(\tau_t)}{\vdash\llbracket\tau\rrbracket_\tau[\llbracket t\rrbracket_t]:\llbracket\Gamma\rrbracket_\Gamma\triangleright_\tau\llbracket\Gamma'\rrbracket_\Gamma,\iota(A)}\,(\tau\tau')$$

□

6.5 Conclusion and perspectives

6.5.1 Conclusion

In this chapter, we presented a system of simple types for a call-by-need calculus with control. We proved that this type system is safe, in the sense that it satisfies the subject reduction property (Theorem 6.2) and the (weak) normalization property (Theorem 6.22). We proved the normalization by means of realizability-inspired interpretation of the $\overline{\lambda}_{[lv\tau\star]}$-calculus. Incidentally, this opens the doors to the computational analysis (in the spirit of Krivine classical realizability) of classical proofs using control, laziness and shared memory.

Besides, we introduced system F_Υ as a type system for the target of a continuation-and-store-passing style translation for the $\overline{\lambda}_{[lv\tau\star]}$-calculus, and we proved that the translation was well-typed (Theorem 6.33). Furthermore, we also refined our presentation to define both source and target languages with explicit De Bruijn levels, making them both more compatible with an implementation.

Last, we believe that the principles guiding the typing of the translation emphasized its computational content, whose three main ingredients are the following:

1. a continuation-passing style translation,
2. a store-passing style translation,
3. a Kripke forcing-like manner of typing the extensibility of the store.

The latter is particularly highlighted in the translation with De Bruijn levels, where levels need to be shifted when extending the store and coercions give a computational content to the subtyping relation (*i.e.* to store extension).

6.5.2 About stores and forcing

Actually, the connection between (Kripke) forcing and the store-passing style translation does not come as a surprise. Indeed, the translation on types logically accounts for the compilation of the calculus with stores to a calculus without store. In the realm of functional programming, memory states are given a meaning through the state monad. For instance, the monadic translation of an arrow enriches it with a state S:

$$[\![A \rightarrow B]\!] \triangleq S \times A \rightarrow S \times B$$

In particular, the result of a function may depend on the current state. If one observes precisely our realizability interpretation, it is very similar to our definition of truth and falsity values: for a type A, its interpretation is roughly of the shape $A \times \tau$. It is folklore that the state monad can be categorically interpreted by means of presheaves construction [138, 116]. Interestingly, Kripke models are a particular case of presheaves semantics [123]. Cohen forcing construction is also interpreted in terms of presheaves [111], and this interpretation scales to type theory [82, 81]. Therefore, the state monad and the forcing translation were already known to be connected. Last but not least, the analysis of Cohen forcing in the framework of Krivine classical realizability [98, 120] relies on an extension of Krivine abstract machine with a cell (which contains the forcing condition). In short, our typed store-passing style translation is just another observation of the connection between forcing translations and explicit stores as a side-effect.

6.5.3 Extension to 2^{nd}-order type systems

We focused in this chapter on simply-typed versions of the $\overline{\lambda}_{lv}$ and $\overline{\lambda}_{[lv\tau\star]}$ calculi. But as it is common in Krivine classical realizability, first and second-order quantifications (in Curry style) come for free through the interpretation. This means that we can for instance extend the language of types to second-order arithmetic:

$$\begin{array}{rcl} e_1, e_2 & ::= & x \mid f(e_1,\dots,e_k) \\ A,B & ::= & X(e_1,\dots,e_k) \mid A \to B \mid \forall x.A \mid \forall X.A \end{array}$$

We can then define the following rules to introduce the universal quantification:

$$\frac{\Gamma \vdash_v v : A \quad x \notin FV(\Gamma)}{\Gamma \vdash_v v : \forall x.A}\ (\forall_r^1) \qquad\qquad \frac{\Gamma \vdash_v v : A \quad X \notin FV(\Gamma)}{\Gamma \vdash_v v : \forall X.A}\ (\forall_r^2)$$

Observe that these rules need to be restricted at the level of strong values, just as they are restricted to values in the case of call-by-value (see Section 4.5.4). As for the left rules, they can be defined at any levels, let say the more general e:

$$\frac{\Gamma \vdash_e e : (A[n/x])^{\perp\!\!\!\perp}}{\Gamma \vdash_e e : (\forall x.A)^{\perp\!\!\!\perp}}\ (\forall_l^1) \qquad\qquad \frac{\Gamma \vdash_e e : (A[B/X])^{\perp\!\!\!\perp}}{\Gamma \vdash_e e : (\forall X.A)^{\perp\!\!\!\perp}}\ (\forall_l^2)$$

where n is any natural number and B any formula. The usual (call-by-value) interpretation of the quantification is defined as an intersection over all the possible instantiations of the variables within the model. First-order variables are to be instantiated by integers, while second-order variables are to be instantiated by sets of terms at the lowest level, *i.e.* closed strong-values in store (which we write $\mathcal{V}_0$):

$$|\forall x.A|_v = \bigcap_{n\in\mathbb{N}} |A[n/x]|_v \qquad\qquad |\forall X.A|_v = \bigcap_{S\in\mathcal{P}(\mathcal{V}_0)} |A[S/X]|_v$$

It is then routine to check that the typing rules are adequate with the realizability interpretation.

6.5.4 Related work & further work

In a recent paper, Kesner uses an intersection type system to characterize normalizing by-need terms [86]. Even though her calculus is not classical, it might be interesting to adapt her approach to our framework. Specifically, we have the intuition that intersection types could be an alternative to our subtyping relation in the target language of the CPS.

As for call-by need with control, recent work by Pédrot and Saurin [134] relates (classical) call-by-need with linear head-reduction from a computational point of view. If they do not provide any type system or normalization results, they connect their framework with a variant of the $\overline{\lambda}_{lv}$-calculus (in natural deduction style). Our techniques should then be adaptable to their framework in order to equip their calculi with type systems and prove similar results.

This chapter naturally raises the question of studying the system F_Υ that we used as target language of our translation. In particular, it might be interesting to understand the logical strength of such a system. It seems to be stronger than systems F or F$_{<:}$ in that is allows a restricted form of dependent types: the second-order quantification range over vectors of arbitrary size. It is probably weaker than a higher order calculus with unrestricted dependencies in types, like the calculus of constructions (which is logically as strong as F$_\omega$). Yet, it might also be the case that a clever analysis of the translation could lead to a bound on the size of the store extension at each step. This would offer a way to remove this dependency and to embed the target language into system F.

7- A classical sequent calculus with dependent types

Side-effects and dependent types

In Chapter 5, we introduced dependent types from the point of view of logic, in the realm of Martin-Löf type theory, but actually, as a programming features, restricted form of dependent types were anterior to this. For instance, in the 60s the programming language FORTRAN IV already allowed programmers to define arrays of a given dimension, and in this sense, (restricted form of) dependent types are as old as high-level programming languages.

From the point of view of programming, dependent types allow us to assign more precise types—and thus more precise specifications—to existing programs. Dependent types are provided by Coq or Agda, two of the most actively developed proof assistants, which both rely on a constructive type theory: Coquand and Paulin-Mohring's calculus of inductive constructions for Coq [29], and Martin-Löf's type theory [114] for Agda. Yet, both systems lack of classical logic and more generally of side-effects, which make them impractical as programming languages.

In practice, effectful languages give to the programmer a more explicit access to low-level control (that is: to the way the program is executed on the available hardware), and make some algorithms easier to implement. Common effects, such as the explicit manipulation of the memory, the generation of random numbers and input/output facilities are available in all practical programming languages (*e.g.* OCaml, C++, Python, Java,...).

As we saw in Section 5.1.2.2,dependent types misbehave in the presence of control operators, and lead to logical inconsistencies. Since the same problem arises with a wider class of effects, it seems that we are facing the following dilemma: either we choose an effectful language (allowing us to write more programs) while accepting the lack of dependent types, or we choose a dependently typed language (allowing us to write finer specifications) and give up effects.

Many works have tried to fill the gap between real programming languages and logic, by accomodating weaker forms of dependent types with computational effects (*e.g.* divergence, I/O, local references, exceptions). Amongst other works, we can cite the recent works by Ahman et al [1], by Vákár [156] or by Pédrot and Tabareau who proposed a systematical way to add effects to type theory [141]. Side-effects—that are impure computations in functional programming—are interpreted by means of monads. Interestingly, control operators can be interpreted in a similar way through the continuation monad, but the continuation monads generally lacks the properties necessary to fit the picture.

Although dependent types and classical logic have been deeply studied separately, the problem of accomodating both features in one and the same system has not found a completely satisfying answer yet. Recent works from Herbelin [70] and Lepigre [108] proposed some restrictions on dependent types to make them compatible with a classical proof system, while Blot [17] designed a hybrid realizability model where dependent types are restricted to an intuitionistic fragment.

Call-by-value and value restriction

In languages enjoying the Church-Rosser property (like the λ-calculus or Coq), the order of evaluation is irrelevant, and any reduction path will ultimately lead to the same value. In particular, the call-by-name and call-by-value evaluation strategies will always give the same result. However, this is no longer the case in presence of side-effects. Indeed, consider the simple case of a function applied to a term producing some side-effects (for instance increasing a reference). In call-by-name, the computation of the argument is delayed to the time of its effective use, while in call-by-value the argument is reduced to a value before performing the application. If, for instance, the function never uses its argument, the call-by-name evaluation will not generate any side-effect, and if it uses it twice, the side-effect will occur twice (and the reference will have its value increased by two). On the contrary, in both cases the call-by-value evaluation generates the side-effect exactly once (and the reference has its value increased by one).

In this chapter, we present a language following the call-by-value reduction strategy. While this design choice is strongly related with our long term perspective of giving a sequent calculus presentation of dPA$^\omega$(following the call-by-value strategy but for the lazy parts), this also constitutes a goal in itself. Indeed, when considering a language with control operators (or other kinds of side-effects), soundness often turns out to be subtle to preserve in call-by-value. The first issues in call-by-value in the presence of side-effects were related to references [162] and polymorphism [67]. In both cases, a simple and elegant solution (but unnecessarily restrictive in practice [55, 108]) to solve the inconsistencies consists to introduce a value restriction for the problematic cases, restoring then a sound type system. Recently, Lepigre presented a proof system providing dependent types and a control operator [108], whose consistency is preserved by means of a semantical value restriction defined for terms that behave as values up to observational equivalence. In the present work, we will rather use a syntactic restriction to a fragment of proofs that allows slightly more than values. As will see, the restriction that arises naturally coincides with the negative-elimination-free fragment of Herbelin's dPAω system [70].

A sequent calculus presentation

The main achievement of this chapter is to give a sequent calculus presentation of a call-by-value language with classical control and dependent types, and to justify its soundness through a continuation-passing style translation. Our calculus is an extension of the $\lambda\mu\tilde{\mu}$-calculus [32] with dependent types. Amongst other motivations, such a calculus is close to an abstract machine, which makes it particularly suitable to define CPS translations or to be an intermediate language for compilation [39].

Additionally, while we consider in this chapter the specific case of a calculus with classical logic, the sequent calculus presentation itself is responsible for another difficulty. As we will see, the usual call-by-value strategy of the $\lambda\mu\tilde{\mu}$-calculus causes subject reduction to fail, which would happen already in an intuitionistic type theory. We claim that the solutions we give in this chapter also provide us with solutions in the intuitionistic case. In particular, the system we develop might be a first step to allow the adaption of the well-understood continuation-passing style translations for ML in order to design a (dependently) typed compilation of a system with dependent types such as Coq.

Delimited continuations and CPS translation

The main challenge in designing a sequent calculus with dependent types resides in the fact that the natural relation of reduction one would expect in such a framework is not safe with respect to types. As we will discuss in Section 7.1.4, the problem can be understood as a desynchronization of the type system with respect to the reduction. A simple solution to resolve this, presented in Section 7.1, consists to add an explicit list of dependencies in the typing derivations. This has the advantage of giving a calculus that is very close to the original. However, it is not suitable for obtaining a continuation-passing style translation.

We thus present a second way to solve this issue by introducing delimited continuations [5], which are used to force the purity needed for dependent types in an otherwise non purely functional language. It also justifies the relaxation of the value restriction and leads to the definition of the negative-elimination-free fragment (Section 7.2). Additionally, it allows for the design, in Section 8.3, of a continuation-passing style translation that preserves dependent types and allows for proving the soundness of our system. Finally, it also provides us with a way to embed our calculus into Lepigre's calculus [108], as we shall see in Section 7.4, and in particular it furnishes us a realizability interpretation.

7.1 A minimal classical language with dependent types

The easiest and usual approach to prevent inconsistencies to arise from the simultaneous presence of classical logic is to impose a restriction to values for proofs appearing inside dependent types and operators. In particular, this would prevent us from writing $\texttt{wit}\ p_0$ and $\texttt{prf}\ p_0$ in Herbelin's example.

In this section we will focus on value restriction in the framework of the $\lambda\mu\tilde{\mu}$-calculus, and show how it allows us to keep the proof system is consistent. We shall then see, in Section 7.2, how to relax this constraint.

7.1.1 A minimal language with value restriction

We follow here the stratified presentation[1] of dependent types from the previous section. We place ourselves in the framework of the $\lambda\mu\tilde{\mu}$-calculus to which we add:

- a language of *terms* which contain an encoding[2] of the natural numbers,
- proof terms (t,p) to inhabit the strong existential $\exists x^{\mathbb{N}}.A$ together with the first and second projections, called respectively `wit` (for terms) and `prf` (for proofs),
- a proof term `refl` for the equality of terms and a proof term `subst` for the convertibility of types over equal terms.

For simplicity reasons, we will only consider terms of type $\mathbb{N}$ throughout this chapter. We address the question of extending the domain of terms in Section 7.5.2. The syntax of the corresponding system, that we call dL, is given by:

$$\begin{array}{llcll}
\textbf{Terms} & t & ::= & x \mid \overline{n} \mid \texttt{wit}\ V & (n \in \mathbb{N})\\
\textbf{Proof terms} & p & ::= & V \mid \mu\alpha.c \mid (t,p) \mid \texttt{prf}\ V \mid \texttt{subst}\ p\,q & \\
\textbf{Proof values} & V & ::= & a \mid \lambda a.p \mid \lambda x.p \mid (t,V) \mid \texttt{refl} & \\
\textbf{Contexts} & e & ::= & \alpha \mid p \cdot e \mid t \cdot e \mid \tilde{\mu}a.c & \\
\textbf{Commands} & c & ::= & \langle p \| e \rangle &
\end{array}$$

The formulas are defined by:

$$\textbf{Formulas} \qquad A,B \quad ::= \quad \top \mid \bot \mid t = u \mid \forall x^{\mathbb{N}}.A \mid \exists x^{\mathbb{N}}.A \mid \Pi a : A.B\,.$$

Note that as in dPA^{ω} we included a dependent product $\Pi a : A.B$ at the level of proof terms, but that in the case where $a \notin FV(B)$ this amounts to the usual implication $A \to B$.

[1] This design choice is usually a matter of taste and convenient for us in the perspective of adapting dPA^{ω}. However, it also has the advantage of clearly enlightening the different treatments for term and proofs through the CPS in the next sections.

[2] The nature of the representation is irrelevant here as we will not compute over it. We can for instance add one constant for each natural number.

$$\begin{array}{rcl|rcll}
\langle \mu\alpha.c \| e\rangle & \rightsquigarrow & c[e/\alpha] & \langle (t,p) \| e\rangle & \rightsquigarrow & \langle p \| \tilde\mu a.\langle (t,a) \| e\rangle\rangle & (p \notin V) \\
\langle V \| \tilde\mu a.c\rangle & \rightsquigarrow & c[V/a] & \langle \texttt{prf}\,(t,V) \| e\rangle & \rightsquigarrow & \langle V \| e\rangle & \\
\langle \lambda a.p \| q\cdot e\rangle & \rightsquigarrow & \langle q \| \tilde\mu a.\langle p \| e\rangle\rangle & \langle \texttt{subst}\; p\, q \| e\rangle & \rightsquigarrow & \langle p \| \tilde\mu a.\langle \texttt{subst}\; a\, q \| e\rangle\rangle & (p \notin V) \\
\langle \lambda x.p \| t\cdot e\rangle & \rightsquigarrow & \langle p[t/x] \| e\rangle & \langle \texttt{subst refl}\; q \| e\rangle & \rightsquigarrow & \langle q \| e\rangle &
\end{array}$$

$$\texttt{wit}\;(t,V) \to t \qquad\qquad t \to t' \Rightarrow c[t] \;\rightsquigarrow\; c[t']$$

Figure 7.1: Reduction rules of dL

7.1.2 Reduction rules

As explained in Section 5.1.2.2, a backtracking proof might give place to different witnesses and proofs according to the context of reduction, leading to inconsistencies [69]. The substitution at different places of a proof which can backtrack, as the call-by-name evaluation strategy does, is thus an unsafe operation. On the contrary, the call-by-value evaluation strategy forces a proof to reduce first to a value (thus furnishing a witness) and to share this value amongst all the commands. In particular, this maintains the value restriction along reduction, since only values are substituted.

The reduction rules, defined in Figure 7.1 (where $t \to t'$ denotes the reduction of terms and $c \rightsquigarrow c'$ the reduction of commands), follow the call-by-value evaluation principle. In particular one can see that whenever the command is of the shape $\langle C[p] \| e\rangle$ where $C[p]$ is a proof built on top of p which is not a value, it reduces to $\langle p \| \tilde\mu a.\langle C[a] \| e\rangle\rangle$, opening the construction to evaluate p[3].

Additionally, we denote by $A \equiv B$ the transitive-symmetric closure of the relation $A \triangleright B$, defined as a congruence over term reduction (*i.e.* if $t \to t'$ then $A[t] \;\triangleright\; A[t']$) and by the rules:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
0 = 0 & \triangleright & \top & 0 = S(u) & \triangleright & \bot \\
S(t) = 0 & \triangleright & \bot & S(t) = S(u) & \triangleright & t = u
\end{array}$$

7.1.3 Typing rules

As we explained before, in this section we limit ourselves to the simple case where dependent types are restricted to values, to make them compatible with classical logic. But even with this restriction, defining the type system in the most naive way leads to a system in which subject reduction will fail. Having a look at the β-reduction rule gives us an insight of what happens. Let us imagine that the type system of the $\lambda\mu\tilde\mu$-calculus has been extended to allow dependent products instead of implications. and consider a proof $\lambda a.p : \Pi a : A.B$ and a context $q \cdot e : \Pi a : A.B$. A typing derivation of the corresponding command would be of the form:

$$\dfrac{\dfrac{\dfrac{\Pi_p}{\Gamma, a : A \vdash p : B \mid \Delta}}{\Gamma \vdash \lambda a.p : \Pi a : A.B \mid \Delta}\,(\to_r) \qquad \dfrac{\dfrac{\Pi_q}{\Gamma \vdash q : A \mid \Delta} \quad \dfrac{\Pi_e}{\Gamma \mid e : B[q/a] \vdash \Delta}}{\Gamma \mid q \cdot e : \Pi a : A.B \vdash \Delta}\,(\to_l)}{\langle \lambda a.p \| q \cdot e\rangle : \Gamma \vdash \Delta}\,(\textsc{Cut})$$

while this command would reduce as follows:

$$\langle \lambda a.p \| q \cdot e\rangle \rightsquigarrow \langle q \| \tilde\mu a.\langle p \| e\rangle\rangle.$$

[3] The reader might recognize the rule (ς) of Wadler's sequent calculus [161].

On the right-hand side, we see that p, whose type is $B[a]$, is now cut with e whose type is $B[q]$. Consequently, we are not able to derive a typing judgment[4] for this command anymore:

$$\dfrac{\dfrac{\Pi_q}{\Gamma \vdash q : A \mid \Delta} \qquad \dfrac{\dfrac{\Gamma, a : A \vdash p : \cancel{B[a]} \mid \Delta \quad \Gamma, a : A \mid e : \cancel{B[q]} \vdash \Delta}{\langle p \| e \rangle : \Gamma, a : A \vdash \Delta}\;\textit{Mismatch}}{\Gamma \mid \tilde{\mu} a.\langle p \| e \rangle : A \vdash \Delta}\,(\tilde{\mu})}{\langle q \| \tilde{\mu} a.\langle p \| e \rangle \rangle : \Gamma \vdash \Delta}\,(\textsc{Cut})$$

The intuition is that in the full command, a has been linked to q at a previous level of the typing judgment. However, the command is still safe, since the head-reduction imposes that the command $\langle p \| e \rangle$ will not be executed before the substitution of a by q[5] is performed and by then the problem would have been solved. Roughly speaking, this phenomenon can be seen as a desynchronization of the typing process with respect to computation. The synchronization can be re-established by making explicit a *dependencies list* in the typing rules, which links $\tilde{\mu}$ variables (here a) to the associate proof term on the left-hand side of the command (here q). We can now obtain the following typing derivation:

$$\dfrac{\dfrac{\Pi_q}{\Gamma \vdash q : A \mid \Delta} \qquad \dfrac{\dfrac{\dfrac{\Pi_p}{\Gamma, a : A \vdash p : B[a] \mid \Delta} \quad \dfrac{\Pi_e}{\Gamma, a : A \mid e : B[q] \vdash \Delta; \{\cdot|p\}\{a|q\}}}{\langle p \| e \rangle : \Gamma, a : A \vdash \Delta; \{a|q\}}\,(\textsc{Cut})}{\Gamma \mid \tilde{\mu} a.\langle p \| e \rangle : A \vdash \Delta; \{.|q\}}\,(\tilde{\mu})}{\langle q \| \tilde{\mu} a.\langle p \| e \rangle \rangle : \Gamma \vdash \Delta; \varepsilon}\,(\textsc{Cut})$$

Formally, we denote by $\mathcal{D}$ the set of proofs we authorize in dependent types, and define it for the moment as the set of values:

$$\mathcal{D} \triangleq V.$$

We define a list of dependencies σ as a list binding pairs of proof terms[6]:

$$\sigma ::= \varepsilon \mid \sigma\{p|q\},$$

and we define A_σ as the set of types that can be obtained from A by replacing none or all occurrences of p by q for each binding $\{p|q\}$ in σ such that $q \in \mathcal{D}$:

$$A_\varepsilon \triangleq \{A\} \qquad\qquad A_{\sigma\{p|q\}} \triangleq \begin{cases} A_\sigma \cup (A[q/p])_\sigma & \text{if } q \in \mathcal{D} \\ A_\sigma & \text{otherwise.} \end{cases}$$

The list of dependencies is filled while going up in the typing tree, and it can be used when typing a command $\langle p \| e \rangle$ to resolve a potential inconsistency between their types:

$$\dfrac{\Gamma \vdash p : A \mid \Delta; \sigma \quad \Gamma \mid e : B \vdash \Delta; \sigma\{\cdot|p\} \quad B \in A_\sigma}{\langle p \| e \rangle : \Gamma \vdash \Delta; \sigma}\,(\textsc{Cut})$$

Remark 7.1. The reader familiar with explicit substitutions [52] can think of the list of dependencies as a fragment of the substitution that is available when a command c is reduced. Another remark is

[4] Observe that the problem here arises independently of the value restriction or not (that is whether we consider that q is a value or not), and is peculiar to the sequent calculus presentation).

[5] Note that even if we were not restricting ourselves to values, this would still hold: if at some point the command $\langle p \| e \rangle$ is executed, it is necessarily after that q has produced a value to substitute for a.

[6] In practice we will only bind a variable with a proof term, but it is convenient for proofs to consider this slightly more general definition.

$$\frac{\Gamma \vdash p : A \mid \Delta;\sigma \quad \Gamma \mid e : A' \vdash \Delta;\sigma\{\cdot|p\} \quad A' \in A_\sigma}{\langle p\|e\rangle : \Gamma \vdash \Delta;\sigma}\,(\textsc{Cut})$$

$$\frac{(a : A) \in \Gamma}{\Gamma \vdash a : A \mid \Delta;\sigma}\,(\mathrm{Ax}_r) \qquad \frac{(\alpha : A) \in \Delta}{\Gamma \mid \alpha : A \vdash \Delta;\sigma\{\cdot|p\}}\,(\mathrm{Ax}_l) \qquad \frac{c : (\Gamma \vdash \Delta, \alpha : A;\sigma)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta;\sigma}\,(\mu)$$

$$\frac{c : (\Gamma, a : A \vdash \Delta;\sigma\{a|p\})}{\Gamma \mid \tilde\mu a.c : A \vdash \Delta;\sigma\{\cdot|p\}}\,(\tilde\mu) \qquad \frac{\Gamma, a : A \vdash p : B \mid \Delta;\sigma}{\Gamma \vdash \lambda a.p : \Pi a : A.B \mid \Delta;\sigma}\,(\rightarrow_r)$$

$$\frac{\Gamma \vdash q : A \mid \Delta;\sigma \quad \Gamma \mid e : B[q/a] \vdash \Delta;\sigma\{\cdot|\dagger\} \quad q \notin \mathcal{D} \rightarrow a \notin FV(B)}{\Gamma \mid q \cdot e : \Pi a : A.B \vdash \Delta;\sigma\{\cdot|p\}}\,(\rightarrow_l)$$

$$\frac{\Gamma, x : \mathbb{N} \vdash p : A \mid \Delta;\sigma}{\Gamma \vdash \lambda x.p : \forall x^{\mathbb{N}}.A \mid \Delta;\sigma}\,(\forall_l) \qquad \frac{\Gamma \vdash t : \mathbb{N} \vdash \Delta;\sigma \quad \Gamma \mid e : A[t/x] \vdash \Delta;\sigma\{\cdot|\dagger\}}{\Gamma \mid t \cdot e : \forall x^{\mathbb{N}}.A \vdash \Delta;\sigma\{\cdot|p\}}\,(\forall_r)$$

$$\frac{\Gamma \vdash t : \mathbb{N} \mid \Delta;\sigma \quad \Gamma \vdash p : A(t) \mid \Delta;\sigma}{\Gamma \vdash (t,p) : \exists x^{\mathbb{N}}.A(x) \mid \Delta;\sigma}\,(\exists_r) \qquad \frac{\Gamma \vdash p : \exists x^{\mathbb{N}}.A(x) \mid \Delta;\sigma \quad p \in \mathcal{D}}{\Gamma \vdash \texttt{prf}\ p : A(\texttt{wit}\ p) \mid \Delta;\sigma}\,\texttt{prf}$$

$$\frac{\Gamma \vdash p : A \mid \Delta;\sigma \quad A \equiv B}{\Gamma \vdash p : B \mid \Delta;\sigma}\,(\equiv_r) \qquad \frac{\Gamma \mid e : A \vdash \Delta;\sigma \quad A \equiv B}{\Gamma \mid e : B \vdash \Delta;\sigma}\,(\equiv_l)$$

$$\frac{\Gamma \vdash p : t = u \mid \Delta;\sigma \quad \Gamma \vdash q : B[t/x] \mid \Delta;\sigma}{\Gamma \vdash \texttt{subst}\ p\,q : B[u/x] \mid \Delta;\sigma}\,(\texttt{subst}) \qquad \frac{\Gamma \vdash t : \mathbb{N} \mid \Delta;\sigma}{\Gamma \vdash \texttt{refl} : t = t \mid \Delta;\sigma}\,(\texttt{refl})$$

$$\frac{}{\Gamma, x : \mathbb{N} \vdash x : \mathbb{N} \mid \Delta;\sigma}\,(\mathrm{Ax}_t) \qquad \frac{n \in \mathbb{N}}{\Gamma \vdash \bar{n} : \mathbb{N} \mid \Delta;\sigma}\,(\mathrm{Ax}_n) \qquad \frac{\Gamma \vdash p : \exists x A(x) \mid \Delta;\sigma \quad p \in \mathcal{D}}{\Gamma \vdash \texttt{wit}\ p : \mathbb{N} \mid \Delta;\sigma}\,(\texttt{wit})$$

Figure 7.2: Typing rules of dL

that the design choice for the (Cut) rule is arbitrary, in the sense that we chose to check whether B is in A_σ. We could equivalently have checked whether the condition $\sigma(A) = \sigma(B)$ holds, where $\sigma(A)$ refers to the type A where for each binding $\{p|q\} \in \sigma$ with $q \in \mathcal{D}$, all the occurences of p have been replaced by q. ⌟

Furthermore, when typing a stack with the $(\rightarrow_l)$ rule, we need to drop the open binding in the list of dependencies[7]. We introduce the notation $\Gamma \mid e : A \vdash \Delta;\sigma\{\cdot|\dagger\}$ to denote that the dependency to be produced is irrelevant and can be dropped. This trick spares us from defining a second type of sequent $\Gamma \mid e : A \vdash \Delta;\sigma$ to type contexts when dropping the (open) binding $\{\cdot|p\}$. Alternatively, one can think of $\dagger$ as any proof term not in $\mathcal{D}$, which is the same with respect to the list of dependencies. The resulting set of typing rules is given in Figure 7.2, where we assume that every variable bound in the typing context is bound only once (proofs and contexts are considered up to α-conversion).

Note that we work with two-sided sequents here to stay as close as possible to the original presentation of the $\lambda\mu\tilde\mu$-calculus [32]. In particular this means that a type in Δ might depend on a variable previously introduced in Γ and vice versa, so that the split into two contexts makes us lose track of the order of introduction of the hypotheses. In the sequel, to be able to properly define a typed CPS

[7] It is easy to convince ourself that when typing a command $\langle p\|q \cdot \tilde\mu a.c\rangle$ with $\{\cdot|p\}$, the "correct" dependency within c should be $\{a|\mu\alpha\alpha\langle p\|q \cdot \alpha\rangle\}$, where the right proof is not a value. Furthermore, this dependency is irrelevant since there is no way to produce such a command where a type adjustment with respect to a needs to be made in c.

translation, we consider that we can unify both contexts into a single one that is coherent with respect to the order in which the hypothesis have been introduced[8]. We denote this context by $\Gamma \cup \Delta$, where the assumptions of Γ remain unchanged, while the former assumptions $(\alpha : A)$ in Δ are denoted by $(\alpha : A^{\perp\!\!\perp})$.

Example 7.2. The proof $p_1 \triangleq \texttt{subst}\,(\texttt{prf}\ p_0)\ \texttt{refl}$ which was of type $1 = 0$ in Section 5.1.2.2 is now incorrect since the backtracking proof p_0, defined by $\mu\alpha.(0, \mu\_.\langle(1, \texttt{refl})\|\alpha\rangle)$ in our framework, is not a value in $\mathcal{D}$. The proof p_1 should rather be defined by[9] $\mu\alpha.\langle p_0\|\tilde{\mu}a.\langle\texttt{subst}\,(\texttt{prf}\ a)\ \texttt{refl}\|\alpha\rangle\rangle$ which can only be given the type $1 = 1$. ⌟

7.1.4 Subject reduction

We start by giving a few technical lemmas that will be used for proving subject reduction. First, we will show that typing derivations allow weakening on the lists of dependencies. For this purpose, we introduce the notation $\sigma \Rrightarrow \sigma'$ to denote that whenever a judgment is derivable with σ as list of dependencies, then it is derivable using σ':

$$\sigma \Rrightarrow \sigma' \quad \triangleq \quad \forall c\, \forall\Gamma\, \forall\Delta.(c : (\Gamma \vdash \Delta; \sigma) \Rightarrow c : (\Gamma \vdash \Delta; \sigma')).$$

This clearly implies that the same property holds when typing evaluation contexts, *i.e.* if $\sigma \Rrightarrow \sigma'$ then σ can be replaced by σ' in any typing derivation for any context e.

Lemma 7.3 (Dependencies weakening). *For any list of dependencies σ we have:*

$$1.\ \forall V..(\sigma\{V|V\} \Rrightarrow \sigma) \qquad\qquad 2.\ \forall\sigma'.(\sigma \Rrightarrow \sigma\sigma')$$

Proof. The first statement is obvious. The proof of the second is straightforward from the fact that for any p and q, by definition $A_\sigma \subset A_{\sigma\{p|q\}}$. □

As a corollary, we get that † can indeed be replaced by any proof term when typing a context.

Corollary 7.4. *If $\sigma \Rrightarrow \sigma'$, then for any p, e, Γ, Δ:*

$$\Gamma \mid e : A \vdash \Delta; \sigma\{\cdot|\dagger\} \quad\Rightarrow\quad \Gamma \mid e : A \vdash \Delta; \sigma'\{\cdot|p\}.$$

We first state the usual lemmas that guarantee the safety of terms (resp. values, contexts) substitution.

Lemma 7.5 (Safe term substitution). *If $\Gamma \vdash t : \mathbb{N} \mid \Delta; \varepsilon$ then:*

1. $c : (\Gamma, x : \mathbb{N}, \Gamma' \vdash \Delta; \sigma) \quad \Rightarrow \quad c[t/x] : (\Gamma, \Gamma'[t/x] \vdash \Delta[t/x]; \sigma[t/x])$,
2. $\Gamma, x : \mathbb{N}, \Gamma' \vdash q : B \mid \Delta; \sigma \quad \Rightarrow \quad \Gamma, \Gamma'[t/x] \vdash q[t/x] : B[t/x] \mid \Delta[t/x]; \sigma[t/x]$,
3. $\Gamma, x : \mathbb{N}, \Gamma' \mid e : B \vdash \Delta; \sigma \quad \Rightarrow \quad \Gamma, \Gamma'[t/x] \mid e[t/x] : B[t/x] \vdash \Delta[t/x]; \sigma[t/x]$,
4. $\Gamma, x : \mathbb{N}, \Gamma' \vdash u : \mathbb{N} \mid \Delta; \sigma \quad \Rightarrow \quad \Gamma, \Gamma'[t/x] \vdash u[t/x] : \mathbb{N} \mid \Delta[t/x]; \sigma[t/x]$.

Lemma 7.6 (Safe value substitution). *If $\Gamma \vdash V : A \mid \Delta; \varepsilon$ then:*

1. $c : (\Gamma, a : A, \Gamma' \vdash \Delta; \sigma) \quad \Rightarrow \quad c[V/a] : (\Gamma, \Gamma'[V/a] \vdash \Delta[V/a]; \sigma[V/a])$,
2. $\Gamma, a : A, \Gamma' \vdash q : B \mid \Delta; \sigma \quad \Rightarrow \quad \Gamma, \Gamma'[V/a] \vdash q[V/a] : B[V/a] \mid \Delta[V/a]; \sigma[t/x]$,
3. $\Gamma, a : A, \Gamma' \mid e : B \vdash \Delta; \sigma \quad \Rightarrow \quad \Gamma, \Gamma'[V/a] \mid e[V/a] : B[V/a] \vdash \Delta[V/a]; \sigma[V/a]$,

[8] See Section 4.2.3.2 for further details on this point.

[9] That is to say $\texttt{let}\ a = p_0\ \texttt{in}\ \texttt{subst}\ a\ \texttt{refl}$ in natural deduction.

4. $\Gamma, a : A, \Gamma' \vdash u : \mathbb{N} \mid \Delta; \sigma \;\;\Rightarrow\;\; \Gamma, \Gamma'[V/a] \vdash u[V/a] : \mathbb{N} \mid \Delta[V/a]; \sigma[V/a]$.

Lemma 7.7 (Safe context substitution). *If* $\Gamma \mid e : A \vdash \Delta; \varepsilon$ *then:*

1. $c : (\Gamma \vdash \Delta, \alpha : A, \Delta'; \sigma) \qquad \Rightarrow \;\; c[e/\alpha] : (\Gamma \vdash \Delta, \Delta'; \sigma)$,
2. $\Gamma \vdash q : B \mid \Delta, \alpha : A, \Delta'; \sigma \;\;\Rightarrow\;\; \Gamma \vdash q[e/\alpha] : B \mid \Delta, \Delta'; \sigma$,
3. $\Gamma \mid e : B \vdash \Delta, \alpha : A, \Delta'; \sigma \;\;\Rightarrow\;\; \Gamma \mid e[e/\alpha] : B \vdash \Delta, \Delta'; \sigma$.

Proof. The proofs are done by induction on the typing derivation. □

We can now prove the type preservation, using the previous lemmas for rules which perform a substitution, and the list of dependencies to resolve local inconsistencies for dependent types.

Theorem 7.8 (Subject reduction). *If* c, c' *are two commands of* dL *such that* $c : (\Gamma \vdash \Delta; \varepsilon)$ *and* $c \rightsquigarrow c'$, *then* $c' : (\Gamma \vdash \Delta; \varepsilon)$.

Proof. The proof is done by induction on the typing derivation of $c : (\Gamma \vdash \Delta; \varepsilon)$, assuming that for each typing proof, the conversion rules are always pushed down and right as much as possible. To save some space, we sometimes omit the list of dependencies when empty, writing $c : \Gamma \vdash \Delta$ instead of $c : \Gamma \vdash \Delta; \varepsilon$, and we denote the composition of the consecutive $(\equiv_l)$ rules as:

$$\dfrac{\Gamma \mid e : B \vdash \Delta; \sigma}{\Gamma \mid e : A \vdash \Delta; \sigma}\,(\equiv_l)$$

where the hypothesis $A \equiv B$ is implicit.

- **Case** $\langle \lambda x.p \| t \cdot e \rangle \rightsquigarrow \langle p[t/x] \| e \rangle$.

A typing proof for the command on the left-hand side is of the form:

$$\dfrac{\dfrac{\dfrac{\Pi_p}{\Gamma, x : \mathbb{N} \vdash p : A \mid \Delta}}{\Gamma \vdash \lambda x.p : \forall x^{\mathbb{N}}.A \mid \Delta}\,(\forall_r) \qquad \dfrac{\dfrac{\dfrac{\Pi_t}{\Gamma \vdash t : \mathbb{N} \mid \Delta} \quad \dfrac{\Pi_e}{\Gamma \mid e : B[t/x] \vdash \Delta; \{\cdot|\dagger\}}}{\Gamma \mid t \cdot e : \forall x^{\mathbb{N}}.B \vdash \Delta; \{\cdot|\lambda x.p\}}\,(\forall_l)}{\Gamma \mid t \cdot e : \forall x^{\mathbb{N}}.A \vdash \Delta; \{\cdot|\lambda x.p\}}\,(\equiv_l)}{\langle \lambda x.p \| t \cdot e \rangle : \Gamma \vdash \Delta; \varepsilon}\,(\textsc{Cut})$$

We first deduce $A[t/x] \equiv B[t/x]$ from the hypothesis $\forall x^{\mathbb{N}}.A \equiv \forall x^{\mathbb{N}}.B$. Then using that $\Gamma, x : \mathbb{N} \vdash p : A \mid \Delta$ and $\Gamma \vdash t : \mathbb{N} \mid \Delta$, by Lemma 8.4 and the fact that $\Delta[t/x] = \Delta$ we get a proof Π_p' of $\Gamma \vdash p[t/x] : A[t/x] \mid \Delta$. We can thus build the following derivation:

$$\dfrac{\dfrac{\Pi_p'}{\Gamma \vdash p[t/x] : A[t/x] \mid \Delta} \quad \dfrac{\dfrac{\Pi_e}{\Gamma \mid e : B[t/x] \vdash \Delta; \{\cdot|p[t/x]\}}}{\Gamma \mid e : A[t/x] \vdash \Delta; \{\cdot|p[t/x]\}}\,(\equiv_l)}{\langle p[t/x] \| e \rangle : \Gamma \vdash \Delta}\,(\textsc{Cut})$$

using Corollary 7.4 to weaken the binding to $p[t/x]$ in Π_e.

- **Case** $\langle \lambda a.p \| q \cdot e \rangle \rightsquigarrow \langle q \| \tilde{\mu} a.\langle p \| e \rangle \rangle$.

A typing proof for the command on the left-hand side is of the form:

$$\dfrac{\dfrac{\dfrac{\Pi_p}{\Gamma, a : A \vdash p : B \mid \Delta}}{\Gamma \vdash \lambda a.p : \Pi a : A.B \mid \Delta}\,(\rightarrow_r) \qquad \dfrac{\dfrac{\dfrac{\Pi_q}{\Gamma \vdash q : A' \mid \Delta} \quad \dfrac{\Pi_e}{\Gamma \mid e : B'[q/a] \vdash \Delta; \{\cdot|\dagger\}}}{\Gamma \mid q \cdot e : \Pi a : A'.B' \vdash \Delta; \{\cdot|\lambda a.p\}}}{\Gamma \mid q \cdot e : \Pi a : A.B \vdash \Delta; \{\cdot|\lambda a.p\}}\,(\equiv_l)}{\langle \lambda a.p \| q \cdot e \rangle : \Gamma \vdash \Delta}\,(\textsc{Cut})$$

If $q \notin \mathcal{D}$, we define $B'_q \triangleq B'$ which is the only type in $B'_{\{a|q\}}$. Otherwise, we define $B'_q \triangleq B'[q/a]$ which is a type in $B'_{\{a|q\}}$. In both cases, we can build the following derivation:

$$\dfrac{\dfrac{\dfrac{\Pi_q}{\Gamma \vdash q : A' \mid \Delta}}{\Gamma \vdash q : A \mid \Delta}\,(\equiv_l) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_p}{\Gamma, a : A \vdash p : B \mid \Delta}}{\Gamma, a : A \vdash p : B' \mid \Delta}\,(\equiv_r) \quad \dfrac{\Pi_e}{\Gamma, a : A \mid e : B'_q \vdash \Delta; \{a|q\}\{\cdot|p\}}}{\langle p \| e\rangle : \Gamma, a : A \vdash \Delta; \{a|q\}}\,(\textsc{Cut})}{\Gamma \mid \tilde{\mu}a.\langle p\|e\rangle : A \vdash \Delta; \{.|q\}}\,(\tilde{\mu})}{}}{\langle q \| \tilde{\mu}a.\langle p\|e\rangle\rangle : \Gamma \vdash \Delta}\,(\textsc{Cut})$$

using Corollary 7.4 to weaken the dependencies in Π_e.

- **Case** $\langle \mu\alpha.c \| e\rangle \rightsquigarrow c[e/\alpha]$.

A typing proof for the command on the left-hand side is of the form:

$$\dfrac{\dfrac{\dfrac{\Pi_c}{c : \Gamma \vdash \Delta, \alpha : A}}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\,(\mu) \qquad \dfrac{\Pi_e}{\Gamma \mid e : A \vdash \Delta; \{\cdot|\mu\alpha.c\}}}{\langle \mu\alpha.c \| e\rangle : \Gamma \vdash \Delta}\,(\textsc{Cut})$$

We get a proof that $c[e/\alpha] : \Gamma \vdash \Delta; \varepsilon$ is valid by Lemma 7.7.

- **Case** $\langle V \| \tilde{\mu}a.c\rangle \rightsquigarrow c[V/a]$.

A typing proof for the command on the left-hand side is of the form:

$$\dfrac{\dfrac{\Pi_V}{\Gamma \vdash V : A \mid \Delta} \qquad \dfrac{\dfrac{\dfrac{\Pi_c}{c : \Gamma, a : A' \vdash \Delta; \{a|V\}}}{\Gamma \mid \tilde{\mu}a.c : A' \vdash \Delta; \{\cdot|V\}}\,(\tilde{\mu})}{\Gamma \mid \tilde{\mu}a.c : A \vdash \Delta; \{\cdot|V\}}\,(\equiv_l)}{\langle V \| \tilde{\mu}a.c\rangle : \Gamma \vdash \Delta}\,(\textsc{Cut})$$

We first observe that we can derive the following proof:

$$\dfrac{\dfrac{\Pi_V}{\Gamma \vdash V : A \mid \Delta}}{\Gamma \vdash V : A' \mid \Delta}\,(\equiv_l)$$

and get a proof for $c[V/a] : \Gamma \vdash \Delta; \{V|V\}$ by Lemma 7.6. We finally get a proof for $c[V/a] : \Gamma \vdash \Delta; \varepsilon$ by Lemma 7.3.

- **Case** $\langle (t,p) \| e\rangle \rightsquigarrow \langle p \| \tilde{\mu}a.\langle (t,a) \| e\rangle\rangle$, with $p \notin V$.

A proof of the command on the left-hand side is of the form:

$$\dfrac{\dfrac{\dfrac{\Pi_t}{\Gamma \vdash t : \mathbb{N} \mid \Delta} \quad \dfrac{\Pi_p}{\Gamma \vdash p : A[t/x] \mid \Delta}}{\Gamma \vdash (t,p) : \exists x^{\mathbb{N}}.A \mid \Delta}\,(\exists_r) \qquad \dfrac{\Pi_e}{\Gamma \mid e : \exists x^{\mathbb{N}}.A \vdash \Delta; \{\cdot|(t,p)\}}}{\langle (t,p) \| e\rangle : \Gamma \vdash \Delta}\,(\textsc{Cut})$$

We can build the following derivation:

$$\dfrac{\dfrac{\Pi_p}{\Gamma \vdash p : A[t/x] \mid \Delta} \qquad \dfrac{\dfrac{\dfrac{\Pi_{(t,a)}}{\Gamma \vdash (t,a) : \exists x^{\mathbb{N}}.A \mid \Delta}\,(\exists_I) \quad \dfrac{\Pi_e}{\Gamma \mid e : \exists x^{\mathbb{N}}.A \vdash \Delta; \{a|p\}\{\cdot|(t,a)\}}}{\langle (t,a) \| e\rangle : \Gamma, a : A[t/x] \vdash \Delta; \{a|p\}}\,(\textsc{Cut})}{\Gamma \mid \tilde{\mu}a.\langle (t,a) \| e\rangle : A[t/x] \vdash \Delta; \{\cdot|p\}}\,(\tilde{\mu})}{\langle p \| \tilde{\mu}a.\langle (t,a) \| e\rangle\rangle : \Gamma \vdash \Delta}\,(\textsc{Cut})$$

where $\Pi_{(t,a)}$ is as expected, observing that since $p \notin \mathcal{D}$, the binding $\{\cdot|(t,p)\}$ is the same as $\{\cdot|\dagger\}$, and we can apply Corollary 7.4 to weaken dependencies in Π_e.

- **Case** $\langle \mathtt{prf}\,(t,V)\|e\rangle \rightsquigarrow \langle V\|e\rangle$.

This case is easy, observing that a derivation of the command on the left-hand side is of the form:

$$\dfrac{\dfrac{\dfrac{\Pi_t \qquad \dfrac{\Pi_V}{\Gamma \vdash V : A(t) \mid \Delta}}{\Gamma \vdash (t,V) : \exists x^{\mathbb{N}}.A(x) \mid \Delta}\,{\scriptstyle(\exists_r)}}{\Gamma \vdash \mathtt{prf}\,(t,V) : A(\mathtt{wit}\,(t,V)) \mid \Delta}\,{\scriptstyle(\mathtt{prf})} \qquad \dfrac{\Pi_e}{\Gamma \mid e : A(\mathtt{wit}\,(t,V)) \vdash \Delta;\{\cdot|\dagger\}}}{\langle \mathtt{prf}\,(t,V)\|e\rangle : \Gamma \vdash \Delta}\,{\scriptstyle(\textsc{Cut})}$$

Since by definition we have $A(\mathtt{wit}\,(t,V)) \equiv A(t)$, we can derive:

$$\dfrac{\dfrac{\Pi_V}{\Gamma \vdash V : A(t) \mid \Delta} \qquad \dfrac{\dfrac{\Pi_e}{\Gamma \mid e : A(\mathtt{wit}\,(t,V)) \vdash \Delta;\{\cdot|V\}}}{\Gamma \mid e : A(V) \vdash \Delta;\{\cdot|V\}}\,{\scriptstyle(\equiv_l)}}{\langle \mathtt{prf}\,(t,V)\|e\rangle : \Gamma \vdash \Delta}\,{\scriptstyle(\textsc{Cut})}$$

- **Case** $\langle \mathtt{subst}\ \mathtt{refl}\ q\|e\rangle \rightsquigarrow \langle q\|e\rangle$.

This case is straightforward, observing that for any terms t,u, if we have $\mathtt{refl} : t = u$, then $A[t] \equiv A[u]$ for any A.

- **Case** $\langle \mathtt{subst}\ p\,q\|e\rangle \rightsquigarrow \langle p\|\tilde{\mu}a.\langle \mathtt{subst}\ a\,q\|e\rangle\rangle$.

This case is exactly the same as the case $\langle (t,p)\|e\rangle$.

- **Case** $c[t] \rightsquigarrow c[t']$ with $t \to t'$..

Immediate by observing that by definition of the relation $\equiv$, we have $A[t] \equiv A[t']$ for any A.

□

7.1.5 Soundness

We give here a proof of the soundness of dL with a value restriction. The proof is based on an embedding into the $\lambda\mu\tilde{\mu}$-calculus extended with pairs, whose syntax and rules are given in Figure 7.3. A more interesting proof through a continuation-passing translation is presented in Section 8.3.

We first show that typed commands of dL normalize by translating them into the simply-typed $\lambda\mu\tilde{\mu}$-calculus with pairs, that is to say the $\lambda\mu\tilde{\mu}$-calculus extended[10] with proofs of the form (p_1,p_2) and contexts of the form $\tilde{\mu}(a_1,a_2).c$. We do not consider here a particular reduction strategy, and take $\rightarrowtail$ to be the contextual closure of the rules given in Figure 7.3.

The translation essentially consists of erasing the dependencies in types[11], turning the dependent products into arrows and the dependent sum into a pair. The erasure procedure is defined by:

$$\begin{array}{rcl|rcl}
(\forall x^{\mathbb{N}}.A)^* & \triangleq & \mathbb{N} \to A^* & \top^* & \triangleq & \mathbb{N} \to \mathbb{N} \\
(\exists x^{\mathbb{N}}.A)^* & \triangleq & \mathbb{N} \wedge A^* & \bot^* & \triangleq & \mathbb{N} \to \mathbb{N} \\
(\Pi a : A.B)^* & \triangleq & A^* \to B^* & (t = u)^* & \triangleq & \mathbb{N} \to \mathbb{N}
\end{array}$$

and the corresponding translation for terms, proofs, contexts and commands:

[10] This corresponds to the addition of pairs and projections in the λ-calculus to obtain the $\lambda^{\times}$-calculus in Section 2.4.1.

[11] The use of erasure functions is a very standard technique in the systems of the λ-cube, see for instance [132] or [157].

$$\begin{array}{llcl}
\textbf{Proofs} & p & ::= & V \mid \mu\alpha.c \mid (p_1,p_2) \\
\textbf{Values} & V & ::= & a \mid \lambda a.p \mid (V_1,V_2) \\
\textbf{Contexts} & e & ::= & \alpha \mid p\cdot e \mid \tilde\mu a.c \mid \tilde\mu(a_1,a_2).c \\
\textbf{Commands} & c & ::= & \langle p \| e\rangle
\end{array}$$

(a) Syntax

$$\frac{\Gamma \vdash p_1 : A_1 \mid \Delta \qquad \Gamma \vdash p_2 : A_2 \mid \Delta}{\Gamma \vdash (p_1,p_2) : A_1 \wedge A_2 \mid \Delta}\,(\wedge_r)$$

$$\frac{c : \Gamma, a_1 : A_1, a_2 : A_2 \vdash \Delta}{\Gamma \mid \tilde\mu(a_1,a_2).c : A_1 \wedge A_2 \vdash \Delta}\,(\wedge_l)$$

(b) Typing rules

$$\begin{array}{rcl@{\qquad}rcl}
\langle \mu\alpha.c \| e\rangle & \rightarrowtail & c[e/\alpha] & \langle (p_1,p_2) \| \tilde\mu(a_1,a_2).c\rangle & \rightarrowtail & c[p_1/a_1][p_2/a_2] \\
\langle \lambda a.p \| q\cdot e\rangle & \rightarrowtail & \langle q \| \tilde\mu a.\langle p \| e\rangle\rangle & \mu\alpha.\langle p \| \alpha\rangle & \rightarrowtail & p \\
\langle p \| \tilde\mu a.c\rangle & \rightarrowtail & c[p/a] & \tilde\mu a.\langle a \| e\rangle & \rightarrowtail & e
\end{array}$$

(c) Reduction rules

Figure 7.3: $\lambda\mu\tilde\mu$-calculus with pairs

$$\begin{array}{ll|ll|ll}
\langle p \| e\rangle^* & \triangleq \langle p^* \| e^*\rangle & x^* & \triangleq x & (\lambda a.p)^* & \triangleq \lambda a.p^* \\
\alpha^* & \triangleq \alpha & n^* & \triangleq \bar n & (\lambda x.p)^* & \triangleq \lambda x.p^* \\
(t\cdot e)^* & \triangleq t^*\cdot e^* & (\texttt{wit}\ p)^* & \triangleq \pi_1(p^*) & (\mu\alpha.c)^* & \triangleq \mu\alpha.c^* \\
(q\cdot e)^* & \triangleq q^*\cdot e^* & a^* & \triangleq a & (\texttt{prf}\ p)^* & \triangleq \pi_2(p^*) \\
(\tilde\mu a.c)^* & \triangleq \tilde\mu a.c^* & \texttt{refl}^* & \triangleq \lambda x.x & (t,p)^* & \triangleq \mu\alpha.\langle p^* \| \tilde\mu a.\langle (t^*,a) \| \alpha\rangle\rangle
\end{array}$$

$$\begin{array}{ll}
(\texttt{subst}\ V\, q)^* \triangleq \mu\alpha.\langle q^* \| \alpha\rangle & \\
(\texttt{subst}\ p\, q)^* \triangleq \mu\alpha.\langle p^* \| \tilde\mu\_.\langle \mu\alpha.\langle q^* \| \alpha\rangle \| \alpha\rangle\rangle & (p \notin V)
\end{array}$$

where $\pi_i(p) \triangleq \mu\alpha.\langle p \| \tilde\mu(a_1,a_2).\langle a_1 \| \alpha\rangle\rangle$. The term $\bar n$ is defined as any encoding of the natural number n with its type $\mathbb{N}^*$, the encoding being irrelevant here as long as $\bar n \in V$. Note that we translate differently $\texttt{subst}\ V\, q$ and $\texttt{subst}\ p\, q$ to simplify the proof of Proposition 7.11.

We first show that the erasure procedure is adequate with respect to the previous translation.

Lemma 7.9. *The following holds for any types A and B:*

1. *For any terms t and u, $(A[t/u])^* = A^*$.*
2. *For any proofs p and q, $(A[p/q])^* = A^*$.*
3. *If $A \equiv B$ then $A^* = B^*$.*
4. *For any list of dependencies σ, if $A \in B_\sigma$, then $A^* = B^*$.*

Proof. Straightforward: 1 and 2 are direct consequences of the erasure of terms (and thus proofs) from types. 3 follows from 1,2 and the fact that $(t = u)^* = \top^* = \bot^*$. 4 follows from 2. □

We can extend the erasure procedure to typing contexts, and show that it is adequate with respect to the translation of proofs.

Proposition 7.10. *The following holds for any contexts Γ, Δ and any type A:*

1. *For any command c, if $c : \Gamma \vdash \Delta; \sigma$, then $c^* : \Gamma^* \vdash \Delta^*$.*
2. *For any proof p, if $\Gamma \vdash p : A \mid \Delta; \sigma$, then $\Gamma^* \vdash p^* : A^* \mid \Delta^*$.*
3. *For any context e, if $\Gamma \mid e : A \vdash \Delta; \sigma$, then $\Gamma^* \mid e^* : A^* \vdash \Delta^*$.*

Proof. By induction on typing derivations. The fourth item of the previous lemma shows that the list of dependencies becomes useless: since $A \in B_\sigma$ implies $A^* = B^*$, it is no longer needed for the (cut)-rule. Consequently, it can also be dropped for all the other cases. The case of the conversion rule is a direct consequence of the third case. For `refl`, we have by definition, $\texttt{refl}^* = \lambda x.x : \mathbb{N}^* \to \mathbb{N}^*$.

The only non-direct cases are $\texttt{subst}\, p\, q$, with p not a value, and (t,p). To prove the former with $p \notin V$, we have to show that if:

$$\dfrac{\Gamma \vdash p : t = u \mid \Delta; \sigma \qquad \Gamma \vdash q : B[t/x] \mid \Delta; \sigma}{\Gamma \vdash \texttt{subst}\, p\, q : B[u/x] \mid \Delta; \sigma}\;(\texttt{subst})$$

then $\texttt{subst}\, p\, q^* = \mu\alpha.\langle p^* \| \tilde\mu_-.\langle \mu\alpha.\langle q^* \| \alpha\rangle \| \alpha\rangle\rangle : B[u/x]^*$. According to Lemma 7.9, we have $B[u/x]^* = B[t/x]^* = B^*$. By induction hypothesis, we have proofs of $\Gamma^* \vdash p^* : \mathbb{N}^* \to \mathbb{N}^* \mid \Delta^*$ and $\Gamma^* \vdash q^* : B \mid \Delta^*$. Using the notation $\eta_{q^*} \triangleq \mu\alpha.\langle q^* \| \alpha\rangle$, we can derive:

$$\dfrac{\dfrac{\Gamma^* \vdash p^* : \mathbb{N}^* \to \mathbb{N}^* \mid \Delta^* \qquad \dfrac{\dfrac{\dfrac{\Gamma^* \vdash q^* : B^* \mid \Delta^*}{\Gamma^* \vdash \eta_{q^*} : B^* \mid \Delta^*} \quad \dfrac{}{\alpha : B^* \vdash \alpha : B^*}}{\langle \eta_{q^*} \| \alpha\rangle : \Gamma \vdash \Delta^*, \alpha : B^*}\;(\textsc{Cut})}{\Gamma^* \mid \tilde\mu_-.\langle \eta_{q^*} \| \alpha\rangle : B^* \vdash \Delta^*, \alpha : B^*}\;(\tilde\mu)}{\langle p^* \| \tilde\mu_-.\langle \eta_{q^*} \| \alpha\rangle\rangle : \Gamma^* \vdash \Delta^*, \alpha : B^*}\;(\textsc{Cut})}{\Gamma^* \vdash \mu\alpha.\langle p^* \| \tilde\mu_-.\langle \eta_{q^*} \| \alpha\rangle\rangle : B^* \mid \Delta^*}\;(\mu)$$

The case $\texttt{subst}\, V\, q$ is easy since $(\texttt{subst}\, V\, q)^* = [\![q]\!]_p$ has type B^* by induction. Similarly, the proof for the case (t,p) corresponds to the following derivation:

$$\dfrac{\dfrac{\Gamma^* \vdash p^* : A^* \mid \Delta^* \qquad \dfrac{\dfrac{\dfrac{\Gamma^* \vdash t^* : \mathbb{N} \mid \Delta^* \quad \dfrac{}{a : A^* \vdash a : A^*}}{\Gamma^*, a : A^* \vdash (t^*, a) : \mathbb{N} \wedge A^* \mid \Delta^*}\;(\wedge_r) \quad \dfrac{}{\alpha : \mathbb{N} \wedge A^* \vdash \alpha : \mathbb{N} \wedge A^*}}{\langle (t^*, a) \| \alpha\rangle : \Gamma, a : A^* \vdash \Delta^*, \alpha : \mathbb{N} \wedge A^*}\;(\textsc{Cut})}{\Gamma^* \mid \tilde\mu a.\langle (t^*, a) \| \alpha\rangle : A^* \vdash \Delta^*, \alpha : \mathbb{N} \wedge A^*}\;(\tilde\mu)}{\langle p^* \| \tilde\mu a.\langle (t^*, a) \| \alpha\rangle\rangle : \Gamma^* \vdash \Delta^*, \alpha : \mathbb{N} \wedge A^*}\;(\textsc{Cut})}{\Gamma^* \vdash \mu\alpha.\langle p^* \| \tilde\mu a.\langle (t^*, a) \| \alpha\rangle\rangle : \mathbb{N} \wedge A^* \mid \Delta^*}\;(\mu)$$

□

We can then deduce the normalization of dL from the normalization of the $\lambda\mu\tilde\mu$-calculus [140], by showing that the translation preserves the normalization in the sense that if c does not normalize, then neither does c^*.

Proposition 7.11. *If c is a command such that c^* normalizes, then c normalizes.*

Proof. We will actually prove a slightly more precise statement:

$$\forall c_1, c_2, (c_1 \overset{1}{\rightsquigarrow} c_2 \Rightarrow \exists n \geq 1, (c_1)^* \overset{n}{\rightarrowtail} (c_2)^*).$$

Assuming it holds, we get from any infinite reduction path (for $\rightsquigarrow$) starting from c another infinite reduction path (for $\rightarrowtail$) from c^*. Thus, the normalization of c^* implies the one of c.

It remains to prove the previous statement, that is an easy induction on the reduction rule $\rightsquigarrow$.

- **Case** $\texttt{wit}\,(t,V) \to t$:.

$$\begin{array}{rcl}
(\texttt{wit}\,(t,V))^* & = & \pi_1(\mu\alpha.\langle V^* \| \tilde\mu a.\langle (t^*, a) \| \alpha\rangle\rangle) \\
& \rightarrowtail & \pi_1(\mu\alpha.\langle (t^*, V^*) \| \alpha\rangle) \\
& \rightarrowtail & \pi_1(t^*, V^*) \\
& = & \mu\alpha.\langle (t^*, t^*) \| \tilde\mu(a_1, a_2).\langle a_1 \| \alpha\rangle\rangle \\
& \rightarrowtail & \mu\alpha.\langle t^* \| \alpha\rangle \rightarrowtail t^*
\end{array}$$

- **Case** $\langle \mu\alpha.c \| e\rangle \rightsquigarrow c[e/\alpha]$:.

$$(\langle \mu\alpha.c \| e\rangle)^* = \langle \mu\alpha.c^* \| e^*\rangle \rightarrowtail c^*[e^*/\alpha] = c[e/\alpha]^*$$

- **Case** $\langle V \| \tilde{\mu}a.c\rangle \rightsquigarrow c[V/a]$:

.

$$(\langle V \| \tilde{\mu}a.c\rangle)^* = \langle V^* \| \tilde{\mu}a.c^*\rangle \rightarrowtail c^*[V^*/a] = c[V/a]^*$$

- **Case** $\langle \lambda a.p \| q \cdot e\rangle \rightsquigarrow \langle q \| \tilde{\mu}a.\langle p \| e\rangle\rangle$:.

$$\begin{aligned} (\langle \lambda a.p \| q \cdot e\rangle)^* &= \langle \lambda a.p^* \| q^* \cdot e^*\rangle \\ &\rightarrowtail \langle q^* \| \tilde{\mu}a.\langle p^* \| e^*\rangle\rangle \\ &= (\langle q \| \tilde{\mu}a.\langle p \| e\rangle\rangle)^* \end{aligned}$$

- **Case** $\langle \lambda x.p \| t \cdot e\rangle \rightsquigarrow \langle p[t/x] \| e\rangle$:.

$$\begin{aligned} \langle \lambda x.p \| t \cdot e\rangle^* &= \langle \lambda x.p^* \| t^* \cdot e^*\rangle \\ &\rightarrowtail \langle t^* \| \tilde{\mu}x.\langle p^* \| e^*\rangle\rangle \\ &\rightarrowtail \langle p^*[t^*/x] \| e^*\rangle = (\langle p[t/x] \| e\rangle)^* \end{aligned}$$

- **Case** $\langle (t,p) \| e\rangle \rightsquigarrow \langle p \| \tilde{\mu}a.\langle (t,a) \| e\rangle\rangle$:.

$$\begin{aligned} (\langle (t,p) \| e\rangle)^* &= \langle \mu\alpha.\langle p^* \| \tilde{\mu}a.\langle (t^*,a) \| \alpha\rangle\rangle \| e^*\rangle \\ &\rightarrowtail \langle p^* \| \tilde{\mu}a.\langle (t^*,a) \| e^*\rangle\rangle \\ &= (\langle p \| \tilde{\mu}a.\langle (t,a) \| e\rangle\rangle)^*. \end{aligned}$$

- **Case** $\langle \mathtt{prf}\,(t,V) \| e\rangle \rightsquigarrow \langle V \| e\rangle$:.

$$\begin{aligned} (\langle \mathtt{prf}\,(t,V) \| e\rangle)^* &= \langle \pi_2(\mu\alpha.\langle V^* \| \tilde{\mu}a.\langle (t^*,a) \| \alpha\rangle\rangle) \| e^*\rangle \\ &\rightarrowtail \langle \pi_2(\mu\alpha.\langle (t^*,V^*) \| \alpha\rangle) \| e^*\rangle \\ &\rightarrowtail \langle \pi_2(t^*,V^*) \| e^*\rangle \\ &= \langle \mu\alpha.\langle (t^*,V^*) \| \tilde{\mu}(a_1,a_2).\langle a_2 \| \alpha\rangle\rangle \| e^*\rangle \\ &= \langle (t^*,V^*) \| \tilde{\mu}(a_1,a_2).\langle a_2 \| e^*\rangle\rangle \\ &\rightarrowtail \langle V^* \| e^*\rangle = (\langle V \| e\rangle)^* \end{aligned}$$

- **Case** $\langle \mathtt{subst}\ \mathtt{refl}\ q \| e\rangle \rightsquigarrow \langle q \| e\rangle$:.

$$\begin{aligned} (\langle \mathtt{subst}\ \mathtt{refl}\ q \| e\rangle)^* &= \langle \mu\alpha.\langle q^* \| \alpha\rangle \| e^*\rangle \\ &\rightarrowtail \langle q^* \| e^*\rangle = (\langle q \| e\rangle)^* \end{aligned}$$

- **Case** $\langle \mathtt{subst}\ p\, q \| e\rangle \rightsquigarrow \langle p \| \tilde{\mu}a.\langle \mathtt{subst}\ a\, q \| e\rangle\rangle$ (with $p \notin V$):.

$$\begin{aligned} (\langle \mathtt{subst}\ p\, q \| e\rangle)^* &= \langle \mu\alpha.\langle p^* \| \tilde{\mu}\_.\langle \mu\alpha.\langle q^* \| \alpha\rangle \| \alpha\rangle\rangle \| e^*\rangle \\ &\rightarrowtail \langle p^* \| \tilde{\mu}\_.\langle \mu\alpha.\langle q^* \| \alpha\rangle \| e^*\rangle\rangle \\ &\rightarrowtail \langle \mu\alpha.\langle q^* \| \alpha\rangle \| e^*\rangle = (\langle \mathtt{subst}\ a\, q \| e\rangle)^* \end{aligned}$$

□

Theorem 7.12. *If $c : (\Gamma \vdash \Delta; \varepsilon)$, then c normalizes.*

Proof. Proof by contradiction: if c does not normalize, then by Proposition 7.11 neither does c^*. However, by Proposition 7.10 we have that $c^* : \Gamma^* \vdash \Delta^*$. This is absurd since any well-typed command of the $\lambda\mu\tilde{\mu}$-calculus normalizes [140]. □

Using the normalization, we can finally prove the soundness of the system.

Theorem 7.13 (Soundness)**.** *For any* $p \in \mathrm{dL}$*, we have* $\nvdash p : \bot$ *.*

Proof. We actually start by proving by contradiction that a command $c \in \mathrm{dL}$ cannot be well-typed with empty contexts. Indeed, let us assume that there is such a command $c : (\vdash)$. By normalization, we can reduce it to $c' = \langle p' \| e' \rangle$ in normal form and for which we have $c' : (\vdash)$ by subject reduction. Since c' cannot reduce and is well-typed, p' is necessarily a value and cannot be a free variable. Thus, e' cannot be of the shape $\tilde{\mu}a.c''$ and every other possibility is either ill-typed or admits a reduction, which are both absurd.

We can now prove the soundness by contradiction. Assuming that there is a proof p such that $\vdash p : \bot$, we can form the well-typed command $\langle p \| \star \rangle : (\vdash \star : \bot)$ where $\star$ is any fresh α-variable. The previous result shows that p cannot drop the context $\star$ when reducing, since it would give rise to command $c : (\vdash)$. We can still reduce $\langle p \| \star \rangle$ to a command c in normal form, and see that c it has to be of the shape $\langle V \| \star \rangle$ (by the same kind of reasoning, using the fact that c cannot reduce and that $c : (\vdash \star : \bot)$ by subject reduction). Therefore, V is a value of type $\bot$. Since there is no typing rule that can give the type $\bot$ to a value, this is absurd. □

7.1.6 Toward a continuation-passing style translation

The difficulty we encountered while defining our system mostly came from the interaction between classical control and dependent types. Removing one of these two ingredients leaves us with a sound system in both cases. Without dependent types, our calculus amounts to the usual $\lambda\mu\tilde{\mu}$-calculus. And without classical control, we would obtain an intuitionistic dependent type theory that we could easily prove sound.

To prove the correctness of our system, we might be tempted to define a translation to a subsystem without dependent types, or classical control. We will discuss later in Section 7.4 a solution to handle the dependencies. We will focus here on the possibility of removing the classical part from dL, that is to define a translation that gets rid of the classical control. The use of continuation-passing style translations to address this issue is very common, and it was already studied for the simply-typed $\lambda\mu\tilde{\mu}$-calculus [32]. However, as it is defined to this point, dL is not suitable for the design of a CPS translation.

Indeed, in order to fix the problem of desynchronization of typing with respect to the execution, we have added an explicit list of dependencies to the type system of dL. Interestingly, if this solved the problem inside the type system, the very same phenomenon happens when trying to define a CPS-translation carrying the type dependencies.

Let us consider, as discussed in Section 7.1.3, the case of a command $\langle q \| \tilde{\mu}a.\langle p \| e \rangle \rangle$ with $p : B[a]$ and $e : B[q]$. Its translation is very likely to look like:

$$[\![q]\!]\,[\![\tilde{\mu}a.\langle p \| e \rangle]\!] = [\![q]\!]\,(\lambda a.([\![p]\!]\,[\![e]\!])),$$

where $[\![p]\!]$ has type $(B[a] \to \bot) \to \bot$ and $[\![e]\!]$ type $B[q] \to \bot$, hence the sub-term $[\![p]\!]\,[\![e]\!]$ will be ill-typed. Therefore, the fix at the level of typing rules is not satisfactory, and we need to tackle the problem already within the reduction rules.

We follow the idea that the correctness is guaranteed by the head-reduction strategy, preventing $\langle p \| e \rangle$ from reducing before the substitution of a was made. We would like to ensure the same thing happens in the target language (that will also be equipped with a head-reduction strategy), namely that $[\![p]\!]$ cannot be applied to $[\![e]\!]$ before $[\![q]\!]$ has furnished a value to substitute for a. This would correspond informally to the term[12]:

$$([\![q]\!](\lambda a.[\![p]\!]))[\![e]\!].$$

[12] We will see in Section 7.3.4 that such a term could be typed by turning the type $A \to \bot$ of the continuation that $[\![q]\!]$ is

Assuming that q eventually produces a value V, the previous term would indeed reduce as follows:

$$(\llbracket q \rrbracket(\lambda a.\llbracket p \rrbracket))\llbracket e \rrbracket \to ((\lambda a.\llbracket p \rrbracket)\,\llbracket V \rrbracket)\,\llbracket e \rrbracket \to \llbracket p \rrbracket[\llbracket V \rrbracket/a]\,\llbracket e \rrbracket$$

Since $\llbracket p \rrbracket[\llbracket V \rrbracket/a]$ now has a type convertible to $(B[q] \to \bot) \to \bot$, the term that is produced in the end is well-typed.

The first observation is that if q, instead of producing a value, was a classical proof throwing the current continuation away (for instance $\mu\alpha.c$ where $\alpha \notin FV(c)$), this would lead to the unsafe reduction:

$$(\lambda\alpha.\llbracket c \rrbracket(\lambda a.\llbracket p \rrbracket))\llbracket e \rrbracket \to \llbracket c \rrbracket\,\llbracket e \rrbracket.$$

Indeed, through such a translation, $\mu\alpha$ would only be able to catch the local continuation, and the term ends in $\llbracket c \rrbracket\llbracket e \rrbracket$ instead of $\llbracket c \rrbracket$. We thus need to restrict ourselves at least to proof terms that could not throw the current continuation.

The second observation is that such a term suggests the use of delimited continuations[13] to temporarily encapsulate the evaluation of q when reducing such a command:

$$\langle \lambda a.p \| q \cdot e \rangle \;\rightsquigarrow\; \langle \mu\hat{\mathfrak{tp}}.\langle q \| \tilde{\mu}a.\langle p \| \hat{\mathfrak{tp}} \rangle\rangle \| e \rangle.$$

This command is safe under the guarantee that q will not throw away the continuation $\tilde{\mu}a.\langle p \| \hat{\mathfrak{tp}} \rangle$, and will mimic the aforedescribed reduction:

$$\langle \mu\hat{\mathfrak{tp}}.\langle q \| \tilde{\mu}a.\langle p \| \hat{\mathfrak{tp}} \rangle\rangle \| e \rangle \;\rightsquigarrow\; \langle \mu\hat{\mathfrak{tp}}.\langle V \| \tilde{\mu}a.\langle p \| \hat{\mathfrak{tp}} \rangle\rangle \| e \rangle \;\rightsquigarrow\; \langle \mu\hat{\mathfrak{tp}}.\langle p[V/a] \| \hat{\mathfrak{tp}} \rangle \| e \rangle \;\rightsquigarrow\; \langle p[V/a] \| e \rangle.$$

This will also allow us to restrict the use of the list of dependencies to the derivation of judgments involving a delimited continuation, and to fully absorb the potential inconsistency in the type of $\hat{\mathfrak{tp}}$. In Section 7.2, we will extend the language according to this intuition, and see how to design a continuation-passing style translation in Section 8.3.

7.2 Extension of the system

7.2.1 Limits of the value restriction

In the previous section, we strictly restricted the use of dependent types to proof terms that are values. In particular, even though a proof term might be computationally equivalent to some value (say $\mu\alpha.\langle V \| \alpha \rangle$ and V for instance), we cannot use it to eliminate a dependent product, which is unsatisfactory. We will thus relax this restriction to allow more proof terms within dependent types.

We can follow several intuitions. First, we saw in the previous section that we could actually allow any proof terms as long as its CPS translation uses its continuation and uses it only once. We do not have such a translation yet, but syntactically, these are the proof terms that can be expressed (up to α-conversion) in the $\lambda\mu\tilde{\mu}$-calculus with only one continuation variable (that we call $\star$ in Figure 7.4), and which do not contain application[14]. We insist on the fact that this defines a syntactic subset of proofs. Indeed, $\star$ is only a notation and any proof defined with only one continuation variable is α-convertible to denote this continuation variable with $\star$. For instance, $\mu\alpha.\langle \mu\beta\langle V \| \beta \rangle \| \alpha \rangle$ belongs to this category since:

$$\mu\alpha.\langle \mu\beta\langle V \| \beta \rangle \| \alpha \rangle =_\alpha \mu\star.\langle \mu\star.\langle V \| \star \rangle \| \star \rangle$$

waiting for into a (dependent) type $\Pi a : A.R[a]$ parameterized by R. This way we could have $\llbracket q \rrbracket : \forall R.(\Pi a : A.R[a] \to R[q])$ instead of $\llbracket q \rrbracket : ((A \to \bot) \to \bot)$. For $R[a] := (B(a) \to \bot) \to \bot$, the whole term is well-typed. Readers should now be familiar with realizability and also note that such a term is realizable, since it eventually terminates on a correct term $\llbracket p[q/a] \rrbracket\,\llbracket e \rrbracket$.

[13] We stick here to the presentations of delimited continuations in [71, 5], where $\hat{\mathfrak{tp}}$ is used to denote the top-level delimiter.

[14] Indeed, $\lambda a.p$ is a value for any p, hence proofs like $\mu\alpha.\langle \lambda a.p \| q \cdot \alpha \rangle$ can drop the continuation in the end once p becomes the proof in active position.

Proofs

$$p \ ::= \ \cdots \mid \mu\hat{\mathfrak{tp}}.c_{\hat{\mathfrak{tp}}}$$

Delimited continuations

$$\begin{array}{rcl} c_{\hat{\mathfrak{tp}}} & ::= & \langle p_N \| e_{\hat{\mathfrak{tp}}} \rangle \mid \langle p \| \hat{\mathfrak{tp}} \rangle \\ e_{\hat{\mathfrak{tp}}} & ::= & \tilde{\mu}a.c_{\hat{\mathfrak{tp}}} \end{array}$$

NEF fragment

$$\begin{array}{rcl} p_N & ::= & V \mid (t,p_N) \mid \mu\star.c_N \\ & & \mid \texttt{prf}\ p_N \mid \texttt{subst}\ p_N\, q_N \\ c_N & ::= & \langle p_N \| e_N \rangle \\ e_N & ::= & \star \mid \tilde{\mu}a.c_N \end{array}$$

(a) Language

$$\begin{array}{rcl} \langle \mu\alpha.c \| e \rangle & \rightsquigarrow & c[e/\alpha] \\ \langle \lambda a.p \| q \cdot e \rangle & \overset{q \in \text{NEF}}{\rightsquigarrow} & \langle \mu\hat{\mathfrak{tp}}.\langle q \| \tilde{\mu}a.\langle p \| \hat{\mathfrak{tp}} \rangle\rangle \| e \rangle \\ \langle \lambda a.p \| q \cdot e \rangle & \rightsquigarrow & \langle q \| \tilde{\mu}a.\langle p \| e \rangle\rangle \\ \langle \lambda x.p \| V_t \cdot e \rangle & \rightsquigarrow & \langle p[V_t/x] \| e \rangle \\ \langle V_p \| \tilde{\mu}a.c \rangle & \rightsquigarrow & c[V_p/a] \\ \langle (V_t,p) \| e \rangle & \overset{p \notin V}{\rightsquigarrow} & \langle p \| \tilde{\mu}a.\langle (V_t,a) \| e \rangle\rangle \\ \langle \texttt{prf}\ (V_t,V_p) \| e \rangle & \rightsquigarrow & \langle V_p \| e \rangle \end{array}$$

$$\begin{array}{rcl} \langle \texttt{prf}\ p \| e \rangle & \rightsquigarrow & \langle \mu\hat{\mathfrak{tp}}.\langle p \| \tilde{\mu}a.\langle \texttt{prf}\ a \| \hat{\mathfrak{tp}} \rangle\rangle \| e \rangle \\ \langle \texttt{subst}\ p\, q \| e \rangle & \overset{p \notin V}{\rightsquigarrow} & \langle p \| \tilde{\mu}a.\langle \texttt{subst}\ a\, q \| e \rangle\rangle \\ \langle \texttt{subst}\ \texttt{refl}\ q \| e \rangle & \rightsquigarrow & \langle q \| e \rangle \\ \langle \mu\hat{\mathfrak{tp}}.\langle p \| \hat{\mathfrak{tp}} \rangle \| e \rangle & \rightsquigarrow & \langle p \| e \rangle \\ c \to c' & \Rightarrow & \langle \mu\hat{\mathfrak{tp}}.c \| e \rangle \rightsquigarrow \langle \mu\hat{\mathfrak{tp}}.c' \| e \rangle \\ \texttt{wit}\ p \to t & \Leftarrow & \forall \alpha, \langle p \| \alpha \rangle \rightsquigarrow \langle (t,p') \| \alpha \rangle \\ t \to t' & \Rightarrow & c[t] \ \rightsquigarrow \ c[t'] \end{array}$$

where:

$$V_t ::= x \mid n \qquad V_p ::= a \mid \lambda a.p \mid \lambda x.p \mid (V_t,V_p) \mid \texttt{refl} \qquad c[t] ::= \langle (t,p) \| e \rangle \mid \langle \lambda x.p \| t \cdot e \rangle$$

(b) Reduction rules

Figure 7.4: $\text{dL}_{\hat{\mathfrak{tp}}}$: extension of dL with delimited continuations

Interestingly, this corresponds exactly to the so-called *negative-elimination-free* (NEF) proofs of Herbelin [70]. To interpret the axiom of dependent choice, he designed a classical proof system with dependent types in natural deduction, in which the dependent types allow the use of NEF proofs.

Second, Lepigre defined in a recent work [108] a classical proof system with dependent types, where the dependencies are restricted to values. However, the type system allows derivations of judgments up to an observational equivalence, and thus any proof computationally equivalent to a value can be used. In particular, any proof in the NEF fragment is observationally equivalent to a value, and hence is compatible with the dependencies of Lepigre's calculus.

From now on, we consider $\text{dL}_{\hat{\mathfrak{tp}}}$ the system dL of Section 7.1 extended with delimited continuations, and define the fragment of *negative-elimination-free* proof terms (NEF). The syntax of both categories is given by Figure 7.4, the proofs in the NEF fragment are considered up to α-conversion for the context variables[15]. The reduction rules, given in Figure 7.4, are slightly different from the rules in Section 7.1. In the case $\langle \lambda a.p \| q \cdot e \rangle$ with $q \in$ NEF (resp. $\langle \texttt{prf}\ p \| e \rangle$), a delimited continuation is now produced during the reduction of the proof term q (resp. p) that is involved in the list of dependencies. As terms can now contain proofs which are not values, we enforce the call-by-value reduction by requiring that proof values only contain term values. We elude the problem of reducing terms, by defining meta-rules for them[16]. We add standard rules for delimited continuations [71, 5], expressing the fact that when a proof $\mu\hat{\mathfrak{tp}}.c$ is in active position, the current context is temporarily frozen until c is fully reduced.

[15] We actually even consider α-conversion for delimited continuations $\hat{\mathfrak{tp}}$, to be able to insert such terms inside a type, even though it might seem strange it will make sense when proving subject reduction.

[16] Everything works as if when reaching a state where the reduction of a term is needed, we had an extra abstract machine to reduce it. Note that this abstract machine could possibly need another machine itself, etc… We could actually solve this by making the reduction of terms explicit, introducing for instance commands and contexts for terms with the appropriate typing rules. However, this is not necessary from a logical point of view and it would significantly increase the complexity of the proofs, therefore we rather chose to stick to the actual presentation.

Regular mode:

$$\frac{\Gamma \vdash p : A \mid \Delta \quad \Gamma \mid e : A' \vdash \Delta\{\cdot|p\}}{\langle p \| e \rangle : \Gamma \vdash \Delta}\ (\textsc{Cut})$$

$$\frac{(a : A) \in \Gamma}{\Gamma \vdash a : A \mid \Delta}\ (\textsc{Ax}_r) \qquad \frac{(\alpha : A) \in \Delta}{\Gamma \mid \alpha : A \vdash \Delta}\ (\textsc{Ax}_l)$$

$$\frac{c : (\Gamma \vdash \Delta, \alpha : A)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\ (\mu) \qquad \frac{c : (\Gamma, a : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}a.c : A \vdash \Delta}\ (\tilde{\mu})$$

$$\frac{\Gamma, a : A \vdash p : B \mid \Delta}{\Gamma \vdash \lambda a.p : \Pi a : A.B \mid \Delta}\ (\rightarrow_r) \qquad \frac{\Gamma \vdash q : A \mid \Delta \quad \Gamma \mid e : B[q/a] \vdash \Delta \quad q \notin \mathcal{D} \Rightarrow a \notin FV(B)}{\Gamma \mid q \cdot e : \Pi a : A.B \vdash \Delta}\ (\rightarrow_l)$$

$$\frac{\Gamma, x : \mathbb{N} \vdash p : A \mid \Delta}{\Gamma \vdash \lambda x.p : \forall x^{\mathbb{N}}.A \mid \Delta}\ (\forall_l) \qquad \frac{\Gamma \vdash t : \mathbb{N} \vdash \Delta \quad \Gamma \mid e : A[t/x] \vdash \Delta}{\Gamma \mid t \cdot e : \forall x^{\mathbb{N}}.A \vdash \Delta}\ (\forall_r)$$

$$\frac{\Gamma \vdash t : \mathbb{N} \mid \Delta \quad \Gamma \vdash p : A(t) \mid \Delta}{\Gamma \vdash (t,p) : \exists x^{\mathbb{N}}.A(x) \mid \Delta}\ (\exists_r) \qquad \frac{\Gamma \vdash p : \exists x^{\mathbb{N}}.A(x) \mid \Delta \quad p \in \mathcal{D}}{\Gamma \vdash \texttt{prf}\ p : A(\texttt{wit}\ p) \mid \Delta}\ \texttt{prf}$$

$$\frac{\Gamma \vdash p : A \mid \Delta \quad A \equiv B}{\Gamma \vdash p : B \mid \Delta}\ (\equiv_r) \qquad \frac{\Gamma \mid e : A \vdash \Delta \quad A \equiv B}{\Gamma \mid e : B \vdash \Delta}\ (\equiv_l)$$

$$\frac{\Gamma \vdash p : t = u \mid \Delta \quad \Gamma \vdash q : B[t/x] \mid \Delta}{\Gamma \vdash \texttt{subst}\ p\, q : B[u/x] \mid \Delta}\ (\texttt{subst}) \qquad \frac{\Gamma \vdash t : \mathbb{N} \mid \Delta}{\Gamma \vdash \texttt{refl} : t = t \mid \Delta}\ (\texttt{refl})$$

$$\frac{}{\Gamma, x : \mathbb{N} \vdash x : \mathbb{N} \mid \Delta}\ (\textsc{Ax}_t) \qquad \frac{n \in \mathbb{N}}{\Gamma \vdash \overline{n} : \mathbb{N} \mid \Delta}\ (\textsc{Ax}_n) \qquad \frac{\Gamma \vdash p : \exists x A(x) \mid \Delta \quad p \in \mathcal{D}}{\Gamma \vdash \texttt{wit}\ p : \mathbb{N} \mid \Delta}\ (\texttt{wit})$$

Dependent mode:

$$\frac{c : (\Gamma \vdash_d \Delta, \hat{\mathfrak{tp}} : A; \varepsilon)}{\Gamma \vdash \mu\hat{\mathfrak{tp}}.c : A \mid \Delta}\ (\mu\hat{\mathfrak{tp}}) \qquad \frac{\Gamma \vdash p : A \mid \Delta \quad \Gamma \mid e : A \vdash_d \Delta, \hat{\mathfrak{tp}} : B; \sigma\{\cdot|p\}}{\langle p \| e \rangle : \Gamma \vdash_d \Delta, \hat{\mathfrak{tp}} : B; \sigma}\ (\textsc{Cut}_d)$$

$$\frac{B \in A_\sigma}{\Gamma \mid \hat{\mathfrak{tp}} : A \vdash_d \Delta, \hat{\mathfrak{tp}} : B; \sigma\{\cdot|p\}}\ (\hat{\mathfrak{tp}}) \qquad \frac{c : (\Gamma, a : A \vdash_d \Delta, \hat{\mathfrak{tp}} : B; \sigma\{a|p\})}{\Gamma \mid \tilde{\mu}a.c : A \vdash_d \Delta, \hat{\mathfrak{tp}} : B; \sigma\{\cdot|p\}}\ (\tilde{\mu}_d)$$

Figure 7.5: Type system for $\mathrm{dL}_{\hat{\mathfrak{tp}}}$

7.2.2 Delimiting the scope of dependencies

For the typing rules, we can extend the set $\mathcal{D}$ to be the NEF fragment:

$$\mathcal{D} \triangleq \text{NEF}$$

and we now distinguish two modes. The regular mode corresponds to a derivation without dependency issues whose typing rules are the same as in Figure 7.2 without the list of dependencies; plus the new rule of introduction of a delimited continuation $\hat{\mathtt{tp}}_I$. The dependent mode is used to type commands and contexts involving $\hat{\mathtt{tp}}$, and we use the symbol $\vdash_d$ to denote the sequents. There are three rules: one to type $\hat{\mathtt{tp}}$, which is the only one where we use the dependencies to unify dependencies; one to type context of the form $\tilde{\mu}a.c$ (the rule is the same as the former rule for $\tilde{\mu}a.c$ in Section 7.1); and a last one to type commands $\langle p \| e\rangle$, where we observe that the premise for p is typed in regular mode.

Additionally, we need to extend the congruence to make it compatible with the reduction of NEF proof terms (that can now appear in types), thus we add the rules:

$$\begin{array}{rcll} A[p] & \triangleright & A[q] & \text{if } \forall \alpha\, (\langle p \| \alpha\rangle \rightsquigarrow \langle q \| \alpha\rangle) \\ A[\langle q \| \tilde{\mu}a.\langle p \| \star\rangle\rangle] & \triangleright & A[\langle p[q/a] \| \star\rangle] & \text{with } p, q \in \text{NEF} \end{array}$$

Due to the presence of NEF proof terms (which contain a delimited form of control) within types and list of dependenciess, we need the following technical lemma to prove subject reduction.

Lemma 7.14. *For any context* Γ, Δ, *any type* A *and any* $e, \mu\star.c$:

$$\langle \mu\star.c \| e\rangle : \Gamma \vdash_d \Delta, \hat{\mathtt{tp}} : B; \varepsilon \qquad \Rightarrow \qquad c[e/\star] : \Gamma \vdash_d \Delta, \hat{\mathtt{tp}} : B; \varepsilon.$$

Proof. By definition of the NEF proof terms, $\mu\star.c$ is of the general form $\mu\star.c = \mu\star.\langle p_1 \| \tilde{\mu}a_1.\langle p_2 \| \tilde{\mu}a_2.\langle \ldots \| \tilde{\mu}a_{n-1}.\langle p_n \| \star\rangle\rangle\rangle\rangle$. For simplicity reasons, we will only give the proof for the case $n = 2$, so that a derivation for the hypothesis is of the form (we assume the CONV-rules have been pushed to the left of cuts):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_1}{\Gamma \vdash p_1 : A_1 \mid \Delta, \star : A} \qquad \dfrac{\dfrac{\dfrac{\Pi_2}{\Gamma, a_1 : A_1 \vdash p_2 : A \mid \Delta, \star : A} \quad \dfrac{}{\cdots \mid \star : A \vdash \Delta, \star : A}}{\langle p_2 \| \star\rangle : \Gamma, a_1 : A_1 \vdash \Delta, \star : A}\,{\scriptstyle(\text{Cut})}}{\Gamma \mid \tilde{\mu}a_1.\langle p_2 \| \star\rangle : A_1 \vdash \Delta, \star : A}\,{\scriptstyle(\tilde{\mu})}}{\langle p_1 \| \tilde{\mu}a_1.\langle p_2 \| \star\rangle\rangle : \Gamma \vdash \Delta, \star : A}\,{\scriptstyle(\text{Cut})}}{\Gamma \vdash \mu\star.\langle p_1 \| \tilde{\mu}a_1.\langle p_2 \| \star\rangle\rangle : A \mid \Delta}\,{\scriptstyle(\mu)}}{\Gamma \vdash \mu\star.c : A \mid \Delta} \qquad \dfrac{\Pi_e}{\Gamma \mid e : A \vdash_d \Delta, \hat{\mathtt{tp}} : B; \{\cdot | \mu\star.c\}}}{\langle \mu\star.c \| e\rangle : \Gamma \vdash_d \Delta, \hat{\mathtt{tp}} : B; \varepsilon}\,{\scriptstyle(\text{Cut})}$$

Thus, we have to show that we can turn Π_e into a derivation Π_e' of $\Gamma \mid e : A \vdash_d \Delta_{\hat{\mathtt{tp}}}; \{a_1|p_1\}\{\cdot|p_2\}$ with $\Delta_{\hat{\mathtt{tp}}} \triangleq \Delta, \hat{\mathtt{tp}} : B$, since this would allow us to build the following derivation:

$$\dfrac{\dfrac{\Pi_1}{\Gamma \vdash p_1 : A_1 \mid \Delta} \qquad \dfrac{\dfrac{\dfrac{\Pi_2}{\Gamma, a_1 : A_1 \vdash p_2 : A \mid \Delta} \quad \dfrac{\Pi_e'}{\cdots \mid e : A \vdash_d \Delta_{\hat{\mathtt{tp}}}; \{a_1|p_1\}\{\cdot|p_2\}}}{\langle p_2 \| \star\rangle : \Gamma, a_1 : A_1 \vdash \Delta_{\hat{\mathtt{tp}}}; \{a_1|p_1\}}\,{\scriptstyle(\text{Cut})}}{\Gamma \mid \tilde{\mu}a_1.\langle p_2 \| e\rangle : A_1 \vdash_d \Delta_{\hat{\mathtt{tp}}}; \{\cdot|p_1\}}\,{\scriptstyle(\tilde{\mu})}}{\langle p_1 \| \tilde{\mu}a_1.\langle p_2 \| e\rangle\rangle : \Gamma \vdash_d \Delta_{\hat{\mathtt{tp}}}; \varepsilon}\,{\scriptstyle(\text{Cut})}$$

It suffices to prove that if the list of dependencies was used in Π_e to type $\hat{\mathtt{tp}}$, we can still give a derivation with the new one. In practice, it corresponds to showing that for any variable a and any σ:

$$\{a|\mu\star.c\}\sigma \Rrightarrow \{a_1|p_1\}\{a|p_2\}\sigma.$$

For any $A \in B_\sigma$, by definition we have:

$$\begin{aligned} A[\mu\star.\langle p_1 \| \tilde{\mu} a_1.\langle p_2 \| \star\rangle\rangle/b] &\equiv A[\mu\star.\langle p_2[p_1/a_1] \| \star\rangle/b] \\ &\equiv A[p_2[p_1/a_1]/b] = A[p_2/b][p_1/a_1]. \end{aligned}$$

Hence for any $A \in B_{\{a|\mu\star.c\}\sigma}$, there exists $A' \in B_{\{a_1|p_1\}\{a|p_2\}\sigma}$ such that $A \equiv A'$, and we can derive:

$$\dfrac{\dfrac{A' \in B_{\{a_1|p_1\}\{a|p_2\}\sigma}}{\Gamma \mid \hat{\mathtt{tp}} : A' \vdash_d \Delta, \hat{\mathtt{tp}} : B; \{a_1|p_1\}\{b|p_2\}\sigma} \quad A \equiv A'}{\Gamma \mid \hat{\mathtt{tp}} : A \vdash_d \Delta, \hat{\mathtt{tp}} : B; \{a_1|p_1\}\{b|p_2\}\sigma}\,(\equiv_l)$$

□

We can now prove subject reduction for $\mathrm{dL}_{\hat{\mathtt{tp}}}$.

Theorem 7.15 (Subject reduction)**.** *If c, c' are two commands of $\mathrm{dL}_{\hat{\mathtt{tp}}}$ such that $c : (\Gamma \vdash \Delta)$ and $c \rightsquigarrow c'$, then $c' : (\Gamma \vdash \Delta)$.*

Proof. Actually, the proof is slightly easier than for Theorem 7.8, because most of the rules do not involve dependencies. We only give some key cases.

- **Case** $\langle \lambda a.p \| q \cdot e\rangle \rightsquigarrow \langle \mu\hat{\mathtt{tp}}.\langle q \| \tilde{\mu} a.\langle p \| \hat{\mathtt{tp}}\rangle\rangle \| e\rangle$ with $q \in \textsc{nef}$.
  A typing derivation for the command on the left is of the form:

$$\dfrac{\dfrac{\dfrac{\Pi_p}{\Gamma, a : A \vdash p : B \mid \Delta}}{\Gamma \vdash \lambda a.p : \Pi a : A.B \mid \Delta}\,(\to_l) \qquad \dfrac{\dfrac{\Pi_q}{\Gamma \vdash q : A \mid \Delta} \quad \dfrac{\Pi_e}{\Gamma \mid e : B[q/a] \vdash \Delta}}{\Gamma \mid q \cdot e : \Pi a : A.B \vdash \Delta}\,(\to_l)}{\langle \lambda a.p \| q \cdot e\rangle : \Gamma \vdash \Delta}\,(\textsc{Cut})$$

  We can thus build the following derivation for the command on the right:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi_q}{\Gamma \vdash q : A \mid \Delta} \qquad \Pi_p}{\langle q \| \tilde{\mu} a.\langle p \| \hat{\mathtt{tp}}\rangle\rangle : \Gamma \vdash_d \Delta, \hat{\mathtt{tp}} : B[q]; \varepsilon}\,(\textsc{Cut})}{\Gamma \vdash \mu\hat{\mathtt{tp}}.\langle q \| \tilde{\mu} a.\langle p \| \hat{\mathtt{tp}}\rangle\rangle \mid \Delta}\,(\mu\hat{\mathtt{tp}}) \qquad \dfrac{\Pi_e}{\Gamma \mid e : B[q/a] \vdash \Delta}}{\langle \mu\hat{\mathtt{tp}}.\langle q \| \tilde{\mu} a.\langle p \| \hat{\mathtt{tp}}\rangle\rangle \| e\rangle : \Gamma \vdash \Delta}\,(\textsc{Cut})$$

$$\Pi_p = \qquad \dfrac{\dfrac{\dfrac{\Pi_p}{\Gamma, a : A \vdash p : B[a] \mid \Delta} \quad \dfrac{B[q] \in (B[a])_{\{a|q\}}}{\Gamma \mid \hat{\mathtt{tp}} : B[a] \vdash_d \Delta, \hat{\mathtt{tp}} : B[q]; \{a|q\}\{\cdot|\dagger\}}\,(\hat{\mathtt{tp}})}{\langle p \| \hat{\mathtt{tp}}\rangle : \Gamma, a : A \vdash_d \Delta, \hat{\mathtt{tp}} : B[q]; \{a|q\}}\,(\textsc{Cut})}{\Gamma \mid \tilde{\mu} a.\langle p \| \hat{\mathtt{tp}}\rangle : A \vdash_d \Delta, \hat{\mathtt{tp}} : B[q]; \{\cdot|q\}}\,(\tilde{\mu})$$

- **Case** $\langle \mathtt{prf}\ p \| e\rangle \rightsquigarrow \langle \mu\hat{\mathtt{tp}}.\langle p \| \tilde{\mu} a.\langle \mathtt{prf}\ a \| \hat{\mathtt{tp}}\rangle\rangle \| e\rangle$.
  We prove it in the most general case, that is when this reduction occurs under a delimited continuation. A typing derivation for the command on the left has to be of the form:

$$\dfrac{\dfrac{\dfrac{\Pi_p}{\Gamma \vdash p : \exists x.A(x) \mid \Delta}}{\Gamma \vdash \mathtt{prf}\ p : A(\mathtt{wit}\ p) \mid \Delta}\,(\mathtt{prf}) \qquad \dfrac{\Pi_e}{\Gamma \mid e : A(\mathtt{wit}\ p) \vdash_d \Delta, \hat{\mathtt{tp}} : B; \sigma\{\cdot|\mathtt{prf}\ p\}}}{\langle \mathtt{prf}\ p \| e\rangle : \Gamma \vdash_d \Delta, \hat{\mathtt{tp}} : B; \sigma}\,(\textsc{Cut})$$

  The proof p being $\textsc{nef}$, so is $\mu\hat{\mathtt{tp}}.\langle p \| \tilde{\mu} a.\langle \mathtt{prf}\ a \| \hat{\mathtt{tp}}\rangle\rangle$, and by definition of the reduction for types, we have for any type A that:

$$A[\mathtt{prf}\ p] \triangleright A[\mu\hat{\mathtt{tp}}.\langle p \| \tilde{\mu} a.\langle \mathtt{prf}\ a \| \hat{\mathtt{tp}}\rangle\rangle],$$

so that we can prove that for any b:

$$\sigma\{b|\,\texttt{prf}\ p\} \Rrightarrow \sigma\{b|\mu\hat{\mathfrak{tp}}.\langle p\|\tilde\mu a.\langle \texttt{prf}\ a\|\hat{\mathfrak{tp}}\rangle\rangle\}.$$

Thus, we can turn Π_e into Π_e' a derivation of the same sequent except for the list of dependenciess that is changed to $\sigma\{\cdot|\mu\hat{\mathfrak{tp}}.\langle p\|\tilde\mu a.\langle \texttt{prf}\ a\|\hat{\mathfrak{tp}}\rangle\rangle\}$. We conclude the proof of this case by giving the following derivation:

$$\dfrac{\dfrac{\dfrac{\Pi_p}{\Gamma \vdash p : \exists x.A(x) \mid \Delta}}{\langle p\|\tilde\mu a.\langle \texttt{prf}\ a\|\hat{\mathfrak{tp}}\rangle\rangle\Gamma \vdash_d \mid \Delta, \hat{\mathfrak{tp}} : A(\texttt{wit}\ p); \varepsilon}\,(\textsc{Cut}) \qquad \Pi_{\hat{\mathfrak{tp}}}}{\Gamma \vdash \mu\hat{\mathfrak{tp}}.\langle p\|\tilde\mu a.\langle \texttt{prf}\ a\|\hat{\mathfrak{tp}}\rangle\rangle : A(\texttt{wit}\ p) \mid \Delta}\,(\mu\hat{\mathfrak{tp}})$$

with $\Pi_{\hat{\mathfrak{tp}}}$ the following derivation where we removed Γ and Δ when irrelevant:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a : \exists x.A \vdash a : \exists x.A}}{a : \exists x.A \vdash \texttt{prf}\ a : A(\texttt{wit}\ a)}\,(\texttt{prf}) \qquad \dfrac{A(\texttt{wit}\ p) \in (A(\texttt{wit}\ a))_{\{a|p\}}}{\hat{\mathfrak{tp}} : A(\texttt{wit}\ a) \vdash_d \hat{\mathfrak{tp}} : A(\texttt{wit}\ p); \{a|p\}}\,(\hat{\mathfrak{tp}})}{\langle \texttt{prf}\ a\|\hat{\mathfrak{tp}}\rangle : \Gamma, a : \exists x.A(x) \vdash_d \Delta, \hat{\mathfrak{tp}} : A(\texttt{wit}\ p); \{a|p\}}\,(\textsc{Cut})}{\Gamma \mid \tilde\mu a.\langle \texttt{prf}\ a\|\hat{\mathfrak{tp}}\rangle : \exists x.A(x) \vdash_d \Delta, \hat{\mathfrak{tp}} : A(\texttt{wit}\ p); \{\cdot|p\}}\,(\tilde\mu)$$

- **Case** $\langle\mu\hat{\mathfrak{tp}}.\langle p\|\hat{\mathfrak{tp}}\rangle\|e\rangle \rightsquigarrow \langle p\|e\rangle$.

  This case is trivial, because in a typing derivation for the command on the left, $\hat{\mathfrak{tp}}$ is typed with an empty list of dependencies, thus the type of p,e and $\hat{\mathfrak{tp}}$ coincides.

- **Case** $\langle\mu\hat{\mathfrak{tp}}.c\|e\rangle \rightsquigarrow \langle\mu\hat{\mathfrak{tp}}.c'\|e\rangle$ with $c \rightsquigarrow c'$.

  This case corresponds exactly to Theorem 7.8, except for the rule $\langle\mu\alpha.c\|e\rangle \rightsquigarrow c[e/\alpha]$, since $\mu\alpha.c$ is a NEF proof term (remember we are inside a delimited continuation), but this corresponds precisely to Lemma 7.14.

□

Remark 7.16. Interestingly, we could have already taken $\mathcal{D} \triangleq$ NEF in dL and still be able to prove the subject reduction property. The only difference would have been for the case $\langle\mu\alpha.c\|e\rangle \rightsquigarrow c[e/\alpha]$ when $\mu\alpha.c$ is NEF. Indeed, we would have had to prove that such a reduction step is compatible with the list of dependencies, as in the proof for $\mathrm{dL}_{\hat{\mathfrak{tp}}}$, which essentially amounts to Lemma 7.14. This shows that the relaxation to the NEF fragment is valid even without delimited continuations.

To sum up, the restriction to NEF is sufficient to obtain a sound type system, but is not enough to obtain a calculus suitable for a continuation-passing style translation. As we will now see, delimited continuations are crucial for the soundness of the CPS translation. Observe that they also provide us with a type system in which the scope of dependencies is more delimited. ⌟

7.3 A continuation-passing style translation

We shall now see how to define a continuation-passing style translation from $\mathrm{dL}_{\hat{\mathfrak{tp}}}$ to an intuitionistic type theory, and use this translation to prove the soundness of $\mathrm{dL}_{\hat{\mathfrak{tp}}}$. Continuation-passing style translations are indeed very useful to embed languages with classical control into purely functional ones [62, 32]. From a logical point of view, they generally amount to negative translations that allow to embed classical logic into intuitionistic logic [42]. Yet, we know that removing classical control (*i.e.* classical logic) of our language leaves us with a sound intuitionistic type theory. We will now see how to design a CPS translation for our language which will allow us to prove its soundness.

$$\begin{array}{rcll}
t & ::= & x \mid \bar{n} \mid \texttt{wit}\; p & (n \in \mathbb{N}) \\
p & ::= & a \mid \lambda a.p \mid \lambda x.p \mid p\,q \mid p\,t & \\
  &   & \mid (t,p) \mid \texttt{prf}\; p \mid \texttt{refl} \mid \texttt{subst}\; p\,q & \\
  &   & & \\
A,B & ::= & \top \mid \bot \mid t = u \mid \Pi a : A.B & \\
  &   & \mid \forall x^{\mathbb{N}} A \mid \exists x^{\mathbb{N}} A \mid \forall X.A &
\end{array}$$

(a) Language and formulas

$$\begin{array}{rcll}
(\lambda x.p)\,t & \rightarrow_\beta & p[t/x] & \\
(\lambda a.p)\,q & \rightarrow_\beta & p[q/a] & \\
p\,q & \rightarrow_\beta & p'\,q & (\text{if } p \rightarrow_\beta p') \\
k(\texttt{wit}\,(t,p)) & \rightarrow_\beta & k\,t & \\
\texttt{prf}\,(t,p) & \rightarrow_\beta & p & \\
\texttt{subst refl}\; q & \rightarrow_\beta & q &
\end{array}$$

(b) Reduction rules

$$\frac{}{\Gamma \vdash \bar{n} : \mathbb{N}}\,(\mathrm{Ax}_n) \qquad \frac{(x : \mathbb{N}) \in \Gamma}{\Gamma \vdash x : \mathbb{N}}\,(\mathrm{Ax}_t) \qquad \frac{(a : A) \in \Gamma}{\Gamma \vdash a : A}\,(\mathrm{Ax}_p)$$

$$\frac{\Gamma, a : A \vdash p : B}{\Gamma \vdash \lambda a.p : \Pi a : A.B}\,(\rightarrow_I) \qquad \frac{\Gamma \vdash p : \Pi a : A.B \quad \Gamma \vdash q : A}{\Gamma \vdash p\,q : B[q/a]}\,(\rightarrow_E) \qquad \frac{\Gamma, x : \mathbb{N} \vdash p : A}{\Gamma \vdash \lambda x.p : \forall x^{\mathbb{N}} A}\,(\forall_I^1)$$

$$\frac{\Gamma \vdash p : \forall x^{\mathbb{N}} A \quad \Gamma \vdash t : \mathbb{N}}{\Gamma \vdash p\,t : A[t/x]}\,(\forall_E^1) \qquad \frac{\Gamma \vdash p : A \quad X \notin FV(\Gamma)}{\Gamma \vdash p : \forall X.A}\,(\forall_I^2) \qquad \frac{\Gamma \vdash p : \forall X.A}{\Gamma \vdash p : A[P/X]}\,(\forall_E^2)$$

$$\frac{\Gamma \vdash t : \mathbb{N} \quad \Gamma \vdash p : A[u/x]}{\Gamma \vdash (t,p) : \exists x^{\mathbb{N}} A}\,(\exists_I) \qquad \frac{\Gamma \vdash p : \exists x^{\mathbb{N}} A}{\Gamma \vdash \texttt{prf}\; p : A(\texttt{wit}\; p)}\,(\texttt{prf}) \qquad \frac{\Gamma \vdash p : \exists x^{\mathbb{N}} A}{\Gamma \vdash \texttt{wit}\; p : \mathbb{N}}\,(\texttt{wit})$$

$$\frac{}{\Gamma \vdash \texttt{refl} : x = x}\,(\texttt{refl}) \qquad \frac{\Gamma \vdash q : t = u \quad \Gamma \vdash q : A[t]}{\Gamma \vdash \texttt{subst}\; p\,q : A[u]}\,(\texttt{subst}) \qquad \frac{\Gamma \vdash p : A \quad A \equiv B}{\Gamma \vdash p : B}\,(\mathrm{CONV})$$

(c) Type system

Figure 7.6: Target language

7.3.1 Target language

We choose the target language to be an intuitionistic theory in natural deduction that has exactly the same elements as $\mathrm{dL}_{\hat{tp}}$, except the classical control. The language distinguishes between terms (of type $\mathbb{N}$) and proofs, it also includes dependent sums and products for types referring to terms as well as a dependent product at the level of proofs. As it is common for CPS translations, the evaluation follows a head-reduction strategy. The syntax of the language and its reduction rules are given by Figure 7.6.

The type system, also presented in Figure 7.6, is defined as expected, with the addition of a second-order quantification that we will use in the sequel to refine the type of translations of terms and NEF proofs. As for $\mathrm{dL}_{\hat{tp}}$ the type system has a conversion rule, where the relation $A \equiv B$ is the symmetric-transitive closure of $A \triangleright B$, defined once again as the congruence over the reduction $\longrightarrow$ and by the rules:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
0 = 0 & \triangleright & \top & 0 = S(u) & \triangleright & \bot \\
S(t) = 0 & \triangleright & \bot & S(t) = S(u) & \triangleright & t = u.
\end{array}$$

7.3.2 Translation of proofs and terms

We can now define the continuation-passing style translation of terms, proofs, contexts and commands. The translation is given in Figure 7.7, in which we tag some lambdas with a bullet $\lambda^{\bullet}$ for technical reasons. The translation for delimited continuation follows the intuition we presented in Section 7.1.6, and the definition for stacks $t \cdot e$ and $q \cdot e$ (with q NEF) inlines the reduction producing a command with

$$\begin{array}{lll@{\qquad}lll}
\llbracket \mathtt{wit}\ p \rrbracket_t & \triangleq & \lambda k.\llbracket p \rrbracket_p\,(\dot{\lambda} q.k\,(\mathtt{wit}\ q)) & \llbracket n \rrbracket_{V_t} & \triangleq & \bar{n} \\
\llbracket x \rrbracket_t & \triangleq & \lambda k.k\,x & & &
\end{array}$$

$$\begin{array}{lll@{\qquad}lll}
\llbracket a \rrbracket_V & \triangleq & a & \llbracket \mathtt{refl} \rrbracket_V & \triangleq & \mathtt{refl} \\
\llbracket \lambda a.p \rrbracket_V & \triangleq & \dot{\lambda} a.\llbracket p \rrbracket_p & \llbracket \lambda x.p \rrbracket_V & \triangleq & \dot{\lambda} x.\llbracket p \rrbracket_p \\
\llbracket (V_t,V_p) \rrbracket_V & \triangleq & (\llbracket V_t \rrbracket_V, \llbracket V \rrbracket_V) & & &
\end{array}$$

$$\begin{array}{lll@{\qquad}lll}
\llbracket V \rrbracket_p & \triangleq & \lambda k.k\,\llbracket V \rrbracket_V & \llbracket \mu\hat{\mathtt{tp}}.c \rrbracket_p & \triangleq & \lambda k.\llbracket c \rrbracket_{\hat{\mathtt{tp}}} k \\
\llbracket \mu\alpha.c \rrbracket_p & \triangleq & \dot{\lambda}\alpha.\llbracket c \rrbracket_c & & & \\
\llbracket \mathtt{prf}\ p \rrbracket_p & \triangleq & \dot{\lambda} k.(\llbracket p \rrbracket_p\,(\dot{\lambda} q \lambda k'.k'\,(\mathtt{prf}\ q)))\,k & & & \\
\llbracket (t,p) \rrbracket_p & \triangleq & \dot{\lambda} k.\llbracket p \rrbracket_p(\llbracket t \rrbracket_t\,(\lambda x \dot{\lambda} a.k\,(x,a))) & & & \\
\llbracket \mathtt{subst}\ V\,q \rrbracket_p & \triangleq & \lambda k.\llbracket q \rrbracket_p(\dot{\lambda} q'.k\,(\mathtt{subst}\,\llbracket V \rrbracket_V\,q'))) & & & \\
\llbracket \mathtt{subst}\ p\,q \rrbracket_p & \triangleq & \lambda k.\llbracket p \rrbracket_p\,(\dot{\lambda} p'.\llbracket q \rrbracket_p(\dot{\lambda} q'.k\,(\mathtt{subst}\ p'\,q'))) & & & (p \notin V)
\end{array}$$

$$\begin{array}{lll@{\qquad}lll}
\llbracket \alpha \rrbracket_e & \triangleq & \alpha & \llbracket \tilde{\mu} a.c \rrbracket_e & \triangleq & \dot{\lambda} a.\llbracket c \rrbracket_c \\
\llbracket t \cdot e \rrbracket_e & \triangleq & \lambda p.(\llbracket t \rrbracket_t\,(\dot{\lambda} v.p\,v))\,\llbracket e \rrbracket_e & & & \\
\llbracket q_N \cdot e \rrbracket_e & \triangleq & \lambda p.(\llbracket q_N \rrbracket_p\,(\dot{\lambda} v.p\,v))\,\llbracket e \rrbracket_e & & & (q_N \in \textsc{nef}) \\
\llbracket q \cdot e \rrbracket_e & \triangleq & \dot{\lambda} p.\llbracket q \rrbracket_p\,(\dot{\lambda} v.p\,v\,\llbracket e \rrbracket_e) & & & (q \notin \textsc{nef})
\end{array}$$

$$\begin{array}{lll@{\qquad}lll}
\llbracket \langle p \| e \rangle \rrbracket_c & \triangleq & \llbracket e \rrbracket_e\,\llbracket p \rrbracket_p & \llbracket \langle p \| \hat{\mathtt{tp}} \rangle \rrbracket_{\hat{\mathtt{tp}}} & \triangleq & \llbracket p \rrbracket_p \\
\llbracket \langle p \| e \rangle \rrbracket_{\hat{\mathtt{tp}}} & \triangleq & \llbracket p \rrbracket_p\,\llbracket e \rrbracket_{e_{\hat{\mathtt{tp}}}} \qquad (e \neq \hat{\mathtt{tp}}) & \llbracket \tilde{\mu} a.c \rrbracket_{e_{\hat{\mathtt{tp}}}} & \triangleq & \dot{\lambda} a.\llbracket c \rrbracket_{\hat{\mathtt{tp}}}
\end{array}$$

Figure 7.7: Continuation-passing style translation

a delimited continuation. All the other rules are natural[17] in the sense that they reflect the reduction rule $\rightsquigarrow$, except for the translation of pairs (t,p):

$$\llbracket (t,p) \rrbracket_p \triangleq \lambda k.\llbracket p \rrbracket_p(\llbracket t \rrbracket_t\,(\lambda x a.k\,(x,a)))$$

The natural definition would have been $\lambda k.\llbracket t \rrbracket_t\,(\lambda u.\llbracket p \rrbracket_p\,\lambda q.k\,(u,q))$, however such a term would have been ill-typed (while this definition is correct, as we will see in the proof of Lemma 7.25). Indeed, the type of $\llbracket p \rrbracket_p$ depends on t, while the continuation $(\lambda q.k\,(u,q))$ depends on u, but both become compatible once u is substituted by the value return by $\llbracket t \rrbracket_t$. This somewhat strange definition corresponds to the intuition that we reduce $\llbracket t \rrbracket_t$ within a delimited continuation[18], in order to guarantee that we will not reduce $\llbracket p \rrbracket_p$ before $\llbracket t \rrbracket_t$ has returned a value to substitute for u. The complete translation is given in Figure 7.7.

Before defining the translation of types, we first state a lemma expressing the fact that the translations of terms and NEF proof terms use the continuation they are given once and only once. In particular, it makes them compatible with delimited continuations and a parametric return type. This will allow us to refine the type of their translation.

Lemma 7.17. *The translation satisfies the following properties:*

1. *For any term t in $\mathrm{dL}_{\hat{\mathtt{tp}}}$, there exists a. term t^+ such that for any k we have $\llbracket t \rrbracket_t\,k \rightarrow_\beta^* k\,t^+$.*
2. *For any NEF proof p_N, there exists a. proof p_N^+ such that for any k we have $\llbracket p_N \rrbracket_p\,k \rightarrow_\beta^* k\,p_N^+$.*

[17] As usual, we actually obtained the translation from an intermediate step consisting in the definition of an context-free abstract machine. The reader will recognize the usual descent (in call-by-value) through the levels of p, e, V.

[18] In fact, we will see in the next chapter that this requires a kind of co-delimited continuation.

$$\begin{array}{ll|ll|ll}
x^+ & \triangleq x & (\lambda a.p)^+ & \triangleq \lambda a.[\![p]\!]_p & (\mu\star.c)^+ & \triangleq c^+ \\
n^+ & \triangleq \bar{n} & (\lambda x.p)^+ & \triangleq \lambda x.[\![p]\!]_p & (\mu\hat{\mathfrak{tp}}.c)^+ & \triangleq c^+ \\
(\mathtt{wit}\ p)^+ & \triangleq \mathtt{wit}\ p^+ & (t,p)^+ & \triangleq (t^+,p^+) & (\langle p\|\star\rangle)^+ & \triangleq p^+ \\
a^+ & \triangleq a & (\mathtt{prf}\ p)^+ & \triangleq \mathtt{prf}\ p^+ & (\langle p\|\hat{\mathfrak{tp}}\rangle)^+ & \triangleq p^+ \\
\mathtt{refl}^+ & \triangleq \mathtt{refl} & (\mathtt{subst}\ p\,q)^+ & \triangleq \mathtt{subst}\ p^+\ q^+ & (\langle p\|\tilde{\mu}a.c_{\hat{\mathfrak{tp}}}\rangle)^+ & \triangleq c^+[p^+/a]
\end{array}$$

Figure 7.8: Linearity of the translation for NEF proofs

In particular, we have :

$$[\![t]\!]_t\ \lambda x.x \rightarrow_\beta^* t^+ \qquad \textit{and} \qquad [\![p_N]\!]_p\ \lambda a.a \rightarrow_\beta^* p_N^+$$

Proof. Straightforward mutual induction on the structure of terms and NEF proofs, adding similar induction hypothesis for NEF contexts and commands. The terms t^+ and proofs p^+ are given in Figure 7.8. We detail the case (t,p) with $p \in$ NEF to give an insight of the proof.

$$\begin{array}{rll}
[\![(t,p)]\!]_p\,k & \rightarrow_\beta [\![p]\!]_p([\![t]\!]_t\ (\lambda xa.k\ (x,a))) & \text{(by definition)} \\
& \rightarrow_\beta ([\![t]\!]_t\ (\lambda xa.k\ (x,a)))\ p^+ & \text{(by induction)} \\
& \rightarrow_\beta (\lambda xa.k\ (x,a))\ t^+\ p^+ & \text{(by induction)} \\
& \rightarrow_\beta (\lambda a.k\ (t^+,a))\ p^+ & \\
& \rightarrow_\beta k\ (t^+,p^+) &
\end{array}$$

□

Moreover, we can verify by that the translation preserves the reduction:

Proposition 7.18. *If c,c' be two commands of $\mathrm{dL}_{\hat{\mathfrak{tp}}}$ such that $c \rightsquigarrow c'$, then $[\![c]\!]_c =_\beta [\![c']\!]_c$*

Proof. By induction on the reduction rules for $\rightsquigarrow$, using Lemma 7.17 for cases involving a term t. □

We can in fact prove a finer result to show that any infinite reduction sequence in $\mathrm{dL}_{\hat{\mathfrak{tp}}}$ is responsible for an infinite reduction sequence through the translation. Using the preservation of typing (Proposition 7.26) together with the normalization of the target language, this will give us a proof of the normalization of $\mathrm{dL}_{\hat{\mathfrak{tp}}}$ for typed proof terms.

7.3.3 Normalization of $\mathrm{dL}_{\hat{\mathfrak{tp}}}$

We will now prove that the translation is well-behaved with respect to the reduction. In practice, we are mainly interested in the preservation of normalization through the translation. Namely, we want to prove that if the image $[\![c]\!]_c$ of a command c is normalizing in the target language, then the command c is already normalizing in $\mathrm{dL}_{\hat{\mathfrak{tp}}}$. To this purpose, we roughly proceed as follows:

1. we identify a set of reduction steps in $\mathrm{dL}_{\hat{\mathfrak{tp}}}$ which are directly reflected into a strictly positive number of reduction steps through the CPS;
2. we show that the other steps alone can not form an infinite sequence of reduction;
3. we deduce that every infinite sequence of reduction in $\mathrm{dL}_{\hat{\mathfrak{tp}}}$ give rise to an infinite sequence through the translation.

The first point corresponds thereafter to Proposition 7.21, the second one to the Proposition 7.22. As a matter of fact, the most difficult part is somehow anterior to these points. It consists in understanding *how* a reduction step can be reflected through the translation and why it is *enough* to ensure the preservation of normalization (that is the third point). Instead of stating the result directly and give a long and tedious proof of its correctness, we will rather sketch its main steps.

First of all, we split the reduction rule $\to_\beta$ into two different kinds of reduction steps:

- *administrative reductions*, that we denote by $\longrightarrow_{\mathsf{a}}$, which correspond to continuation-passing and computationally irrelevant (w.r.t. to $\mathrm{dL}_{\hat{tp}}$) reduction steps. These are defined as the β-reduction steps of non-annotated λs.
- *distinguished reductions*, that we denote by $\longrightarrow_\bullet$, which correspond to the image of a reduction step through the translation. These are defined as every other rules, that is to say the β-reduction steps of annotated $\lambda^\bullet$'s plus the rules corresponding to redexes formed with `wit`, `prf` and `subst` .

In other words, we define two deterministic reductions $\longrightarrow_\bullet$ and $\longrightarrow_{\mathsf{a}}$, such that the usual weak-head reduction $\to_\beta$ is equal to the union $\longrightarrow_\bullet \cup \longrightarrow_{\mathsf{a}}$. Our goal will be to prove that every infinite reduction sequence in $\mathrm{dL}_{\hat{tp}}$ will be reflected in the existence of an infinite reduction sequence for $\longrightarrow_\bullet$.

Second, let us assume for a while that we can show for any reduction $c \rightsquigarrow c'$ we obtain:

$$\begin{tikzcd}
\llbracket c \rrbracket_c \arrow[dr, "*", "\beta"'] & & & \llbracket c' \rrbracket_c \arrow[dl, "*"', "\mathsf{a}"] \\
& t_0 \arrow[r, "1", "\bullet"'] & t_1 \arrow[r, "*", "\beta"'] & t_2
\end{tikzcd}$$

through the translation. Then by induction, it implies that if a command c_0 produces an infinite reduction sequence $c_0 \rightsquigarrow c_1 \rightsquigarrow c_2 \rightsquigarrow \ldots$, it is reflected through the translation by the following reduction scheme:

$$\begin{tikzcd}
\llbracket c_0 \rrbracket_c \arrow[dr, "*", "\beta"'] & & & \llbracket c_1 \rrbracket_c \arrow[d, "*"', "\mathsf{a}"] \arrow[dr, "*", "\beta"'] & & & \llbracket c_2 \rrbracket_c \arrow[d, "*"', "\mathsf{a}"] \arrow[dr, "*", "\beta"'] & & \\
& t_{00} \arrow[r, "1", "\bullet"'] & t_{01} \arrow[r, "*", "\beta"'] & t_{02} & t_{10} \arrow[r, "1", "\bullet"'] & t_{11} \arrow[r, "*", "\beta"'] & t_{12} & t_{20} \arrow[r, "1", "\bullet"'] & t_{21} \arrow[r, dashed] & {}
\end{tikzcd}$$

Using the fact that all reductions are deterministic, and that the arrow from $\llbracket c_1 \rrbracket_c$ to t_{02} (and $\llbracket c_2 \rrbracket_c$ to t_{12} and so on) can only contain steps of the reduction $\longrightarrow_{\mathsf{a}}$, the previous scheme in fact ensures us that we have:

$$\begin{tikzcd}
\llbracket c_0 \rrbracket_c \arrow[d, "*", "\beta"'] & & \llbracket c_1 \rrbracket_c \arrow[d, "*", "\mathsf{a}"'] & & & \llbracket c_2 \rrbracket_c \arrow[d, "*", "\mathsf{a}"'] & & & \\
t_{00} \arrow[r, "1", "\bullet"'] & t_{01} \arrow[r, "*", "\beta"'] & t_{02} \arrow[r, "*", "\beta"'] & t_{10} \arrow[r, "1", "\bullet"'] & t_{11} \arrow[r, "*", "\beta"'] & t_{12} \arrow[r, "*", "\beta"'] & t_{20} \arrow[r, "1", "\bullet"'] & t_{21} \arrow[r, dashed] & {}
\end{tikzcd}$$

This directly implies that $\llbracket c_0 \rrbracket_c$ produces an infinite reduction sequence and thus is not normalizing. This would be the ideal situation, and if the aforementioned steps were provable as such, the proof would be over. Yet, our situation is more subtle, and we need to refine our analysis to tackle the problem. We shall briefly explain now why we can actually consider a slightly more general reduction scheme, while trying to remain concise on the justification. Keep in mind that it is our goal to preserve the existence of an infinite sequence of distinguished steps.

The first generalization consists to allow distinguished reductions for redexes that are not in head positions. The safety of this generalization follows from this proposition:

Proposition 7.19. *If $u \longrightarrow_\bullet u'$ and $t[u']$ does not normalize, then neither does $t[u]$.*

Proof. By induction on the structure of t, a very similar proof can be found in [84]. □

Following this idea, we define a new arrow $\xrightarrow{?}_{\bullet}$ by:

$$u \longrightarrow_{\bullet} u' \quad \Rightarrow \quad t[u] \xrightarrow{?}_{\bullet} t[u']$$

where $t[] ::= [] \mid t'(t[]) \mid \lambda x.t[]$, expressing the fact that a distinguished step can be performed somewhere in the term. We denote by $\longrightarrow_{\beta^+}$ the reduction relation defined as the union $\longrightarrow_{\beta} \cup \xrightarrow{?}_{\bullet}$, which is no longer deterministic. Coming back to the thread scheme we described above, we can now generalize it with this arrow. Indeed, as we are only interested in getting an infinite reduction sequence from $[\![c_0]\!]_c$, the previous proposition ensures us that if t_{02} (t_{12}, etc.) does not normalize, it is enough to have an arrow $t_{01} \xrightarrow{*}_{\beta^+} t_{02}$ ($t_{11} \xrightarrow{*}_{\beta^+} t_{12}$, etc.) to deduce that t_{01} does not normalize either. Hence it is enough to prove that we have the following thread scheme, where we took advantage of this observation:

$$\begin{tikzcd}
{[\![c_0]\!]_c} \arrow[d, "*", "\beta"'] & & & {[\![c_1]\!]_c} \arrow[dl, "*"', "\mathsf{a}"] \arrow[d, "*", "\beta"'] & & & {[\![c_2]\!]_c} \arrow[dl, "*"', "\mathsf{a}"] \arrow[d, "*", "\beta"'] & & \\
t_{00} \arrow[r, "1"] & t_{01} \arrow[r, "*", "\beta^+"'] & t_{02} & t_{10} \arrow[r, "1"] & t_{11} \arrow[r, "*", "\beta^+"'] & t_{12} & t_{20} \arrow[r, "1"] & t_{21} \arrow[r, dashed] & {}
\end{tikzcd}$$

In the same spirit, if we define $=_{\mathsf{a}}$ to be the congruence over terms induced by the reduction $\longrightarrow_{\mathsf{a}}$, we can show that if a term has a redex for the distinguished relation in head position, then so does any (administratively) congruent term.

Proposition 7.20. *If $t \xrightarrow{1}_{\bullet} u$ and $t =_{\mathsf{a}} t'$, then there exists u' such that $t' \xrightarrow{1}_{\bullet} u'$ and $u =_{\mathsf{a}} u'$.*

Proof. By induction on t, observing that an administrative reduction can neither delete nor create redexes for $\longrightarrow_{\bullet}$. □

In other words, as we are only interested in the distinguished reduction steps, we can take the liberty to reason modulo the congruence $=_{\mathsf{a}}$. Notably, we can generalize one last time our reduction scheme, replacing the left (administrative) arrow from $[\![c_i]\!]_c$ by this congruence:

$$\begin{tikzcd}
{[\![c_0]\!]_c} \arrow[d, "*", "\beta"'] & & & {[\![c_1]\!]_c} \arrow[dl, equal, "\mathsf{a}"] \arrow[d, "*", "\beta"'] & & & {[\![c_2]\!]_c} \arrow[dl, equal, "\mathsf{a}"] \arrow[d, "*", "\beta"'] & & \\
t_{00} \arrow[r, "1"] & t_{01} \arrow[r, "*", "\beta^+"'] & t_{02} & t_{10} \arrow[r, "1"] & t_{11} \arrow[r, "*", "\beta^+"'] & t_{12} & t_{20} \arrow[r, "1"] & t_{21} \arrow[r, dashed] & {}
\end{tikzcd}$$

For all the reasons explained above, such a reduction scheme ensures that there is an infinite reduction sequence from $[\![c_0]\!]_c$. Because of this guarantee, by induction, it is enough to show that for any reduction step $c_0 \rightsquigarrow c_1$, we have:

$$\begin{tikzcd}
{[\![c_0]\!]_c} \arrow[d, "*", "\beta"'] & & & {[\![c_1]\!]_c} \arrow[dl, equal, "\mathsf{a}"] \\
t_0 \arrow[r, "1"] & t_1 \arrow[r, "*", "\beta^+"'] & t_2 &
\end{tikzcd} \tag{1}$$

In fact, as explained in the preamble of this section, not all reduction steps can be reflected this way through the translation. There are indeed 4 reduction rules, that we identify hereafter, that might only be reflected into administrative reductions, and produce a scheme of this shape (which subsumes the former):

$$[\![c_0]\!]_c \xrightarrow{*}_{\beta^+} t =_{\mathsf{a}} [\![c_1]\!]_c \tag{2}$$

This allows us to give a more precise statement about the preservation of reduction through the CPS translation.

Proposition 7.21 (Preservation of reduction). *Let c be two commands of* $\mathrm{dL}_{\hat{\mathfrak{tp}}}$*. If $c_0 \rightsquigarrow c_1$, then it is reflected through the translation into a reduction scheme (1), except for the rules:*

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\langle \mathtt{subst}\; p\, q \| e\rangle & \overset{p\notin V}{\rightsquigarrow} & \langle p \| \tilde\mu a.\langle \mathtt{subst}\; a\, q\|e\rangle\rangle & \langle \mu\hat{\mathfrak{tp}}.\langle p\|\hat{\mathfrak{tp}}\rangle\|e\rangle & \rightsquigarrow & \langle p\|e\rangle \\
\langle \mathtt{subst}\;\mathtt{refl}\; q\|e\rangle & \rightsquigarrow & \langle q\|e\rangle & c[t] & \rightsquigarrow & c[t']
\end{array}$$

which are reflected in the reduction scheme (2).

Proof. The proof is done by induction on the reduction $\rightsquigarrow$ (see Figure 7.4). To ease the notations, we will often write $\dot\lambda v.(\dot\lambda x.[\![p]\!]_p)\; v \longrightarrow_\bullet \dot\lambda x.[\![p]\!]_p$ where we perform α-conversion to identify $\dot\lambda v.[\![p]\!]_p[v/x]$ and $\dot\lambda x.[\![p]\!]_p$. Additionally, to facilitate the comprehension of the steps corresponding to the congruence $=_{\mathsf{a}}$, we use an arrow $\xrightarrow{?}_{\mathsf{a}}$ to denote the possibility of performing an administrative reduction not in head position, defined by:

$$u \longrightarrow_{\mathsf{a}} u' \quad\Rightarrow\quad t[u] \xrightarrow{?}_{\mathsf{a}} t[u']$$

We write $\longrightarrow_{\mathsf{a}^+}$ the union $\longrightarrow_{\mathsf{a}} \cup \xrightarrow{?}_{\mathsf{a}}$.

- **Case** $\langle \mu\alpha.c\|e\rangle \rightsquigarrow c[e/\alpha]$:
  We have:
  $$\begin{array}{rcl}
  [\![\langle\mu\alpha.c\|e\rangle]\!]_c & = & (\dot\lambda\alpha.[\![c]\!]_c)[\![e]\!]_e \\
  & \longrightarrow_\bullet & [\![c]\!]_c[[\![e]\!]_e/\alpha] = [\![c[e/\alpha]]\!]_c
  \end{array}$$

- **Case** $\langle \lambda a.p\|q\cdot e\rangle \rightsquigarrow \langle q\|\tilde\mu a.\langle p\|e\rangle\rangle$:
  We have:
  $$\begin{array}{rcl}
  [\![\langle\lambda a.p\|q\cdot e\rangle]\!]_c & = & (\lambda k.k\,(\dot\lambda a.[\![p]\!]_p))\;\dot\lambda p.[\![q]\!]_p\,(\dot\lambda v.p\,v\,[\![e]\!]_e) \\
  & \longrightarrow_{\mathsf{a}} & (\dot\lambda p.[\![q]\!]_p\,(\dot\lambda v.p\,v\,[\![e]\!]_e))\;\dot\lambda a.[\![p]\!]_p \\
  & \longrightarrow_\bullet & [\![q]\!]_p\,(\dot\lambda v.(\dot\lambda a.[\![p]\!]_p)\,v\,[\![e]\!]_e) \\
  & \xrightarrow{?}_\bullet & [\![q]\!]_p\,(\dot\lambda a.[\![p]\!]_p\,[\![e]\!]_e) = [\![\langle q\|\tilde\mu a.\langle p\|e\rangle\rangle]\!]_c
  \end{array}$$

- **Case** $\langle\lambda a.p\|q_N\cdot e\rangle \overset{q_N\in\textsc{nef}}{\rightsquigarrow} \langle\mu\hat{\mathfrak{tp}}.\langle q_N\|\tilde\mu a.\langle p\|\hat{\mathfrak{tp}}\rangle\rangle\|e\rangle$:
  We know by Lemma 7.17 that q_N being NEF, it will use, and use only once, the continuation it is applied to. Thus, we know that if $k\longrightarrow_\bullet k'$, we have that:
  $$[\![q_N]\!]_p\,k \xrightarrow{*}_\beta k\,q_N^+ \longrightarrow_\bullet k'\,q_N^+ \;{}_\beta\!\longleftarrow [\![q_N]\!]_p\,k'$$
  and we can legitimately write $[\![q_N]\!]_p\,k\longrightarrow_\bullet [\![q_N]\!]_p\,k'$ in the sense that it corresponds to performing now a reduction that would have been performed in the future. Using this remark, we have:
  $$\begin{array}{rcl}
  [\![\langle\lambda a.p\|q_N\cdot e\rangle]\!]_c & = & (\lambda k.k\,(\dot\lambda a.[\![p]\!]_p))\;\lambda p.([\![q_N]\!]_p\,(\dot\lambda v.p\,v))\;[\![e]\!]_e \\
  & \xrightarrow{2}_{\mathsf{a}} & ([\![q_N]\!]_p\,(\dot\lambda v.(\dot\lambda a.[\![p]\!]_p)\,v))\;[\![e]\!]_e \\
  & \longrightarrow_\bullet & ([\![q_N]\!]_p\,(\dot\lambda a.[\![p]\!]_p))[\![e]\!]_e \\
  & {}_{\mathsf{a}}\!\longleftarrow & (\lambda k.([\![q_N]\!]_p\,(\dot\lambda a.[\![p]\!]_p))\,k)\;[\![e]\!]_e = [\![\langle\mu\hat{\mathfrak{tp}}.\langle q_N\|\tilde\mu a.\langle p\|\hat{\mathfrak{tp}}\rangle\rangle\|e\rangle]\!]_c
  \end{array}$$

- **Case** $\langle\lambda x.p\|V_t\cdot e\rangle\rightsquigarrow\langle p[V_t/x]\|e\rangle$:
  Since V_t is a value (*i.e.* x or n), we have $[\![V_t]\!]_t = \lambda k.k\,[\![V_t]\!]_{V_t}$. In particular, it is easy to deduce that $[\![p[V_t/x]]\!]_p = [\![p]\!]_p[[\![V_t]\!]_{V_t}/x]$, and then we have:
  $$\begin{array}{rcl}
  [\![\langle\lambda x.p\|V_t\cdot e\rangle]\!]_c & = & (\lambda k.k\,(\dot\lambda x.[\![p]\!]_p))\lambda p.([\![V_t]\!]_t\,(\dot\lambda v.p\,v))\;[\![e]\!]_e \\
  & \xrightarrow{2}_{\mathsf{a}} & ([\![V_t]\!]_t\,(\dot\lambda v.(\dot\lambda x.[\![p]\!]_p)\,v))\;[\![e]\!]_e \\
  & \longrightarrow_{\mathsf{a}} & ((\dot\lambda v.(\dot\lambda x.[\![p]\!]_p)\,v)\;[\![V_t]\!]_{V_t})\;[\![e]\!]_e \\
  & \longrightarrow_\bullet & ((\dot\lambda x.[\![p]\!]_p)\;[\![V_t]\!]_{V_t})\;[\![e]\!]_e \\
  & \longrightarrow_\bullet & ([\![p]\!]_p[[\![V_t]\!]_{V_t}/x])\;[\![e]\!]_e = [\![p[V_t/x]]\!]_p\;[\![e]\!]_e = \langle p[V_t/x]\|e\rangle
  \end{array}$$

- **Case** $\langle V \| \tilde{\mu} a.c\rangle \rightsquigarrow c[V_p/a]$:
  Similarly to the previous case, we have $\llbracket V\rrbracket_p = \lambda k.k\,\llbracket V\rrbracket_V$ and thus $\llbracket c[V/x]\rrbracket_c = \llbracket p\rrbracket_p[\llbracket V\rrbracket_V/a]$.

$$\begin{array}{rcl}
\llbracket\langle V_p \| \tilde{\mu} a.c\rangle\rrbracket_c & = & (\lambda k.k\,\llbracket V\rrbracket_V)\dot{\lambda} a.\llbracket c\rrbracket_c \\
& \longrightarrow_{\mathsf{a}} & (\dot{\lambda} a.\llbracket c\rrbracket_c)\,\llbracket V\rrbracket_V \\
& \longrightarrow_{\bullet} & \llbracket c\rrbracket_c[\llbracket V\rrbracket_V/a] = \llbracket c[V/a]\rrbracket_c
\end{array}$$

- **Case** $\langle (V_t,p) \| e\rangle \overset{p\notin V}{\rightsquigarrow} \langle p \| \tilde{\mu} a.\langle (V_t,a) \| e\rangle\rangle$:
  We have :

$$\begin{array}{rcl}
\llbracket\langle (V_t,p) \| e\rangle\rrbracket_c & = & (\dot{\lambda} k.\llbracket p\rrbracket_p(\llbracket V_t\rrbracket_t\,(\lambda x\dot{\lambda} a.k\,(x,a)))\,\llbracket e\rrbracket_e \\
& \longrightarrow_{\bullet} & \llbracket p\rrbracket_p\,(\llbracket V_t\rrbracket_t\,(\lambda x\dot{\lambda} a.\llbracket e\rrbracket_e\,(x,a))) \\
& \longrightarrow_{\mathsf{a}^+} & \llbracket p\rrbracket_p\,((\lambda x\dot{\lambda} a.\llbracket e\rrbracket_e\,(x,a))\,\llbracket V_t\rrbracket_{V_t}) \\
& \longrightarrow_{\mathsf{a}^+} & \llbracket p\rrbracket_p\,(\dot{\lambda} a.\llbracket e\rrbracket_e\,(\llbracket V_t\rrbracket_{V_t},a)) \\
& {}_{\mathsf{a}^+}\!\!\longleftarrow & \llbracket p\rrbracket_p\,(\dot{\lambda} a.\llbracket (V_t,a)\rrbracket_p\,\llbracket e\rrbracket_e) \\
& {}_{\mathsf{a}^+}\!\!\longleftarrow & (\lambda k\,\llbracket p\rrbracket_p\,(\dot{\lambda} a.\llbracket (V_t,a)\rrbracket_p\,k))\,\llbracket e\rrbracket_e = \llbracket\langle p \| \tilde{\mu} a.\langle (V_t,a) \| e\rangle\rangle\rrbracket_c
\end{array}$$

- **Case** $\langle \mathtt{prf}\ p \| e\rangle \rightsquigarrow \langle \mu\hat{\mathtt{tp}}.\langle p \| \tilde{\mu} a.\langle \mathtt{prf}\ a \| \hat{\mathtt{tp}}\rangle\rangle \| e\rangle$:
  We have:

$$\begin{array}{rcl}
\llbracket\langle \mathtt{prf}\ p) \| e\rangle\rrbracket_c & = & \dot{\lambda} k.(\llbracket p\rrbracket_p\,(\dot{\lambda} a\lambda k'.k'\,(\mathtt{prf}\ a)))\,k)\,\llbracket e\rrbracket_e \\
& \longrightarrow_{\bullet} & (\llbracket p\rrbracket_p\,(\dot{\lambda} a.\lambda k'.k'\,(\mathtt{prf}\ a)))\,\llbracket e\rrbracket_e \\
& {}_{\mathsf{a}}\!\longleftarrow & (\lambda k.(\llbracket p\rrbracket_p\,(\dot{\lambda} a.\lambda k'.k'\,(\mathtt{prf}\ a)))\,k)\,\llbracket e\rrbracket_e = \llbracket\langle \mu\hat{\mathtt{tp}}.\langle p \| \tilde{\mu} a.\langle \mathtt{prf}\ a \| \hat{\mathtt{tp}}\rangle\rangle \| e\rangle\rrbracket_c
\end{array}$$

- **Case** $\langle \mathtt{prf}\,(V_t,V_p) \| e\rangle \rightsquigarrow \langle V_p \| e\rangle$:
  We have:

$$\begin{array}{rcl}
\llbracket\langle \mathtt{prf}\,(V_t,V_p) \| e\rangle\rrbracket_c & = & \dot{\lambda} k.((\lambda k.k\,(\llbracket V_t\rrbracket_V,\llbracket V_p\rrbracket_V))\,(\dot{\lambda} q\lambda k'.k'\,(\mathtt{prf}\ q)))\,k)\,\llbracket e\rrbracket_e \\
& \longrightarrow_{\bullet} & ((\lambda k.k\,(\llbracket V_t\rrbracket_V,\llbracket V_p\rrbracket_V))\,(\dot{\lambda} q\lambda k'.k'\,(\mathtt{prf}\ q)))\,\llbracket e\rrbracket_e \\
& \longrightarrow_{\mathsf{a}} & ((\dot{\lambda} q\lambda k'.k'\,(\mathtt{prf}\ q))(\llbracket V_t\rrbracket_V,\llbracket V_p\rrbracket_V))\,\llbracket e\rrbracket_e \\
& \longrightarrow_{\bullet} & (\lambda k'.k'\,(\mathtt{prf}\,(\llbracket V_t\rrbracket_V,\llbracket V_p\rrbracket_V)))\,\llbracket e\rrbracket_e \\
& \longrightarrow_{\mathsf{a}} & \llbracket e\rrbracket_e\,(\mathtt{prf}\,(\llbracket V_t\rrbracket_V,\llbracket V_p\rrbracket_V))) \\
& \xrightarrow{?}_{\bullet} & \llbracket e\rrbracket_e\,\llbracket V_p\rrbracket_V\ {}_{\mathsf{a}}\!\longleftarrow\ \llbracket\langle V_p \| e\rangle\rrbracket_c
\end{array}$$

- **Case** $\langle \mathtt{subst}\ p\,q \| e\rangle \overset{p\notin V}{\rightsquigarrow} \langle p \| \tilde{\mu} a.\langle \mathtt{subst}\ a\,q \| e\rangle\rangle$:
  We have:

$$\begin{array}{rcl}
\llbracket\langle \mathtt{subst}\ p\,q \| e\rangle\rrbracket_c & = & (\lambda k.\llbracket p\rrbracket_p\,(\dot{\lambda} a.\llbracket q\rrbracket_p(\dot{\lambda} q'.k\,(\mathtt{subst}\ a\,q'))))\,\llbracket e\rrbracket_e \\
& \longrightarrow_{\mathsf{a}} & \llbracket p\rrbracket_p\,(\dot{\lambda} a.\llbracket q\rrbracket_p(\dot{\lambda} q'.\llbracket e\rrbracket_e\,(\mathtt{subst}\ a\,q'))) \\
& {}_{\mathsf{a}}\!\xleftarrow{?} & \llbracket p\rrbracket_p\,(\dot{\lambda} a.(\lambda k.\llbracket q\rrbracket_p(\dot{\lambda} q'.k\,(\mathtt{subst}\ a\,q')))\,\llbracket e\rrbracket_e) \\
& = & \llbracket\langle p \| \tilde{\mu} a.\langle \mathtt{subst}\ a\,q \| e\rangle\rangle\rrbracket_c
\end{array}$$

- **Case** $\langle \mathtt{subst}\ \mathtt{refl}\ q \| e\rangle \rightsquigarrow \langle q \| e\rangle$:
  We have:

$$\begin{array}{rcl}
\llbracket\langle \mathtt{subst}\ \mathtt{refl}\ q \| e\rangle\rrbracket_c & = & (\lambda k.\llbracket q\rrbracket_p\,(\dot{\lambda} q'.k\,(\mathtt{subst}\ \mathtt{refl}\ q')))\,\llbracket e\rrbracket_e \\
& \longrightarrow_{\mathsf{a}} & \llbracket q\rrbracket_p\,(\dot{\lambda} q'.\llbracket e\rrbracket_e\,(\mathtt{subst}\ \mathtt{refl}\ q')) \\
& \xrightarrow{?}_{\bullet} & \llbracket q\rrbracket_p\,(\dot{\lambda} q'.\llbracket e\rrbracket_e\,q') \\
& \xrightarrow{?}_{\bullet} & \llbracket q\rrbracket_p\,\llbracket e\rrbracket_e = \llbracket\langle q \| e\rangle\rrbracket_c
\end{array}$$

- **Case** $\langle \mu\hat{\mathfrak{tp}}.\langle p \| \hat{\mathfrak{tp}}\rangle \| e\rangle \rightsquigarrow \langle p \| e\rangle$:
  We have:
$$\begin{array}{rcl} \llbracket \langle \mu\hat{\mathfrak{tp}}.\langle p \| \hat{\mathfrak{tp}}\rangle \| e\rangle \rrbracket_c & = & (\lambda k.\llbracket p\rrbracket_p k)\ \llbracket e\rrbracket_e \\ & \longrightarrow_{\mathsf{a}} & \llbracket p\rrbracket_p\ \llbracket e\rrbracket_e = \llbracket \langle p \| e\rangle \rrbracket_c \end{array}$$

- **Case** $c \rightsquigarrow c' \Rightarrow \langle \mu\hat{\mathfrak{tp}}.c \| e\rangle \rightsquigarrow \langle \mu\hat{\mathfrak{tp}}.c' \| e\rangle$:
  By induction hypothesis, we get that $\llbracket c\rrbracket_c \xrightarrow{*}_{\beta^+} t =_{\mathsf{a}} \llbracket c'\rrbracket_c$ for some term t. Therefore we have:
$$\begin{array}{rcl} \langle \mu\hat{\mathfrak{tp}}.c \| e\rangle & = & (\lambda k.\llbracket c\rrbracket_c\ k)\ \llbracket e\rrbracket_e \\ & \longrightarrow_{\mathsf{a}} & \llbracket c\rrbracket_c\ \llbracket e\rrbracket_e \\ & \xrightarrow{*}_{\beta^+} & t\ \llbracket e\rrbracket_e \\ & =_{\mathsf{a}} & \llbracket c'\rrbracket_c\ \llbracket e\rrbracket_e \\ & {}_{\mathsf{a}}\longleftarrow & (\lambda k.\llbracket c'\rrbracket_c\ k)\ \llbracket e\rrbracket_e = \langle \mu\hat{\mathfrak{tp}}.c' \| e\rangle \end{array}$$

- **Case** $t \rightarrow t' \Rightarrow c[t]\ \rightsquigarrow\ c[t']$:
  As such, the translation does not allow an analysis of this case, mainly because we did not give an explicit small-step semantics for terms, and defined terms reduction through a big-step semantics:
$$\forall \alpha, \langle p \| \alpha\rangle \overset{*}{\rightsquigarrow} \langle (t,q) \| \alpha\rangle \Rightarrow \mathtt{wit}\ p \rightarrow t$$
  However, we claim that we could have extended the language of $\mathrm{dL}_{\hat{\mathfrak{tp}}}$ with commands for terms:
$$c_t ::= \langle t \| e_t\rangle \qquad e_t ::= \tilde\mu x.c[t] \qquad c[\,] ::= \langle ([\,],p) \| e\rangle \mid \langle \lambda x.p \| [\,]\cdot e\rangle$$
  and adding dual operators for (co-)delimited continuations to allow for a small-step definition of terms reduction:
$$\begin{array}{rcl|rcl} \langle \lambda x.p \| t\cdot e\rangle & \rightsquigarrow & \langle \mu\hat{\mathfrak{tp}}.\langle t \| \tilde\mu x.\langle p \| \hat{\mathfrak{tp}}\rangle\rangle \| e\rangle & \langle V_t \| \tilde\mu x.c_t\rangle & \rightsquigarrow & c_t[V_t/x] \\ \langle \mathtt{wit}\ p \| e_t\rangle & \rightsquigarrow & \langle p \| \tilde\mu a.\langle \mathtt{wit}\ a \| e_t\rangle\rangle & \langle \mathtt{wit}\,(V_t,V_p) \| e_t\rangle & \rightsquigarrow & \langle V_t \| e_t\rangle \\ \langle (t,p) \| e\rangle & \rightsquigarrow & \langle p \| \tilde\mu\hat{\mathfrak{tp}}.\langle t \| \tilde\mu x.\langle \hat{\mathfrak{tp}} \| \tilde\mu a.\langle (x,a) \| e\rangle\rangle\rangle\rangle & \langle V_p \| \tilde\mu\hat{\mathfrak{tp}}.\langle \hat{\mathfrak{tp}} \| e\rangle\rangle & \rightsquigarrow & \langle V_p \| e\rangle \end{array}$$
$$c \rightsquigarrow c' \Rightarrow \langle p \| \tilde\mu\hat{\mathfrak{tp}}.c\rangle \rightsquigarrow \langle p \| \tilde\mu\hat{\mathfrak{tp}}.c'\rangle$$
  It is worth noting that these rules simulate the big-step definitions we had before while preserving the global call-by-value strategy. Defining the translation for terms in the extended syntax:
$$\begin{array}{lcl@{\qquad\qquad}lcl} \llbracket \mathtt{wit}\ V_t\rrbracket_t & \triangleq & \lambda k.k\,(\mathtt{wit}\ \llbracket V_t\rrbracket_{V_t}) & \llbracket \tilde\mu x.c\rrbracket_t & \triangleq & \dot\lambda x.\llbracket c\rrbracket_c \\ \llbracket \mathtt{wit}\ p\rrbracket_t & \triangleq & \lambda k.\llbracket p\rrbracket_p\,(\dot\lambda q.k\,(\mathtt{wit}\ q)) & \llbracket \langle t \| e_t\rangle\rrbracket_t & \triangleq & \llbracket t\rrbracket_t\ \llbracket e_t\rrbracket_t \\ \llbracket \tilde\mu\hat{\mathfrak{tp}}.c_t\rrbracket_t & \triangleq & \llbracket c_t\rrbracket_t & \llbracket \hat{\mathfrak{tp}}\rrbracket_p & \triangleq & \dot\lambda k.k \end{array}$$
  We can then prove that each reduction rule satisfies the expected scheme.
  Case $\langle \lambda x.p \| t\cdot e\rangle \rightsquigarrow \langle \mu\hat{\mathfrak{tp}}.\langle t \| \tilde\mu x.\langle p \| \hat{\mathfrak{tp}}\rangle\rangle \| e\rangle$:
  We have:
$$\begin{array}{rcl} \langle \lambda x.p \| t\cdot e\rangle & = & (\dot\lambda k.k\ \dot\lambda x.\llbracket p\rrbracket_p)\ (\lambda p.(\llbracket t\rrbracket_t\ (\dot\lambda v.p\,v))\ \llbracket e\rrbracket_e) \\ & \longrightarrow_{\bullet} & (\lambda p.(\llbracket t\rrbracket_t\ (\dot\lambda v.p\,v))\ \llbracket e\rrbracket_e)\ \dot\lambda x.\llbracket p\rrbracket_p \\ & \longrightarrow_{\mathsf{a}} & (\llbracket t\rrbracket_t\ (\dot\lambda v.(\dot\lambda x.\llbracket p\rrbracket_p)\,v))\ \llbracket e\rrbracket_e \\ & \xrightarrow{?}_{\bullet} & (\llbracket t\rrbracket_t\ (\dot\lambda x.\llbracket p\rrbracket))\ \llbracket e\rrbracket_e \\ & {}_{\mathsf{a}^+}\longleftarrow & \lambda k.((\llbracket t\rrbracket_t\ (\dot\lambda x.\llbracket p\rrbracket))\ k)\ \llbracket e\rrbracket_e = \llbracket \langle \mu\hat{\mathfrak{tp}}.\langle t \| \tilde\mu x.\langle p \| \hat{\mathfrak{tp}}\rangle\rangle \| e\rangle\rrbracket_c \end{array}$$
  Case $\langle (t,p) \| e\rangle \rightsquigarrow \langle p \| \tilde\mu\hat{\mathfrak{tp}}.\langle t \| \tilde\mu x.\langle \hat{\mathfrak{tp}} \| \tilde\mu a.\langle (x,a) \| e\rangle\rangle\rangle\rangle$:
  We have:
$$\begin{array}{rcl} \langle (t,p) \| e\rangle & = & (\dot\lambda k.\llbracket p\rrbracket_p\ (\llbracket t\rrbracket_t\ (\lambda x.\dot\lambda a.k\ (x,a))))\ \llbracket e\rrbracket_e \\ & \longrightarrow_{\bullet} & \llbracket p\rrbracket_p\ (\llbracket t\rrbracket_t\ (\lambda x.\dot\lambda a.\llbracket e\rrbracket_e\ (x,a))) \\ & {}_{\mathsf{a}^+}\longleftarrow & \llbracket p\rrbracket_p\ (\llbracket t\rrbracket_t\ (\lambda x.(\lambda k.k)\dot\lambda a.\llbracket e\rrbracket_e\ (x,a))) \\ & {}_{\mathsf{a}^+}\longleftarrow & \llbracket p\rrbracket_p\ (\llbracket t\rrbracket_t\ (\lambda x.(\lambda k.k)\dot\lambda a.(\lambda k.k\ (x,a))\ \llbracket e\rrbracket_e)) \\ & = & \llbracket \langle p \| \tilde\mu\hat{\mathfrak{tp}}.\langle t \| \tilde\mu x.\langle \hat{\mathfrak{tp}} \| \tilde\mu a.\langle (x,a) \| e\rangle\rangle\rangle\rangle\rrbracket_c \end{array}$$

Case $\langle \mathtt{wit}\ p \| e_t \rangle \rightsquigarrow \langle p \| \tilde{\mu} a.\langle \mathtt{wit}\ a \| e_t \rangle\rangle$:
We have:

$$\begin{aligned}
\llbracket \mathtt{wit}\ p \rrbracket_t \llbracket e_t \rrbracket_t &= (\lambda k.\llbracket p \rrbracket_p\, (\lambda^{\bullet} a.k\,(\mathtt{wit}\ a)))\, \llbracket e_t \rrbracket_t \\
&\longrightarrow_{\mathsf{a}} \llbracket p \rrbracket_p\, (\lambda^{\bullet} a.\llbracket e_t \rrbracket_t\, (\mathtt{wit}\ a))) \\
&{}_{\mathsf{a}^+}\!\!\longleftarrow \llbracket p \rrbracket_p\, (\lambda^{\bullet} a.(\lambda k.k\,(\mathtt{wit}\ a))\, \llbracket e_t \rrbracket_t) = \llbracket \langle p \| \tilde{\mu} a.\langle \mathtt{wit}\ a \| e_t \rangle\rangle \rrbracket_c
\end{aligned}$$

Case $\langle V_t \| \tilde{\mu} x.c_t \rangle \rightsquigarrow c_t[V_t/x]$:
We have:

$$\begin{aligned}
\llbracket \mathtt{wit}\,(V_t, V_p) \rrbracket_t \llbracket e_t \rrbracket_t &= (\lambda k.k\,(\mathtt{wit}\,(\llbracket V_t \rrbracket_{V_t}, \llbracket V_p \rrbracket_V)))\, \llbracket e_t \rrbracket_t \\
&\longrightarrow_{\mathsf{a}} \llbracket e_t \rrbracket_t\, (\mathtt{wit}\,(\llbracket V_t \rrbracket_{V_t}, \llbracket V_p \rrbracket_V)) \\
&\longrightarrow_{\bullet} \llbracket e_t \rrbracket_t\, \llbracket V_t \rrbracket_{V_t} \\
&{}_{\mathsf{a}}\!\longleftarrow (\lambda k.k\, \llbracket V_t \rrbracket_{V_t})\, \llbracket e_t \rrbracket_t = \llbracket V_t \rrbracket_t e_t
\end{aligned}$$

Case $\langle \mathtt{wit}\,(V_t, V_p) \| e_t \rangle \rightsquigarrow \langle V_t \| e_t \rangle$:
We have:

$$\begin{aligned}
\llbracket V_t \rrbracket_t \llbracket \tilde{\mu} x.c \rrbracket_t &= (\lambda k.k\, \llbracket V_t \rrbracket_{V_t})\, \lambda^{\bullet} x.\llbracket c \rrbracket_c \\
&\longrightarrow_{\mathsf{a}} (\lambda^{\bullet} x.\llbracket c \rrbracket_c)\, \llbracket V_t \rrbracket_{V_t} \\
&\longrightarrow_{\bullet} \llbracket c \rrbracket_c[\llbracket V_t \rrbracket_{V_t}/x] = \llbracket c[V_t/x] \rrbracket_c
\end{aligned}$$

Case $\langle V \| \tilde{\mu} \hat{\text{tp}}.\langle \hat{\text{tp}} \| e \rangle\rangle \rightsquigarrow \langle V \| e \rangle$:
We have:

$$\begin{aligned}
\llbracket V \rrbracket_p \llbracket \tilde{\mu} \hat{\text{tp}}.\langle \hat{\text{tp}} \| e \rangle \rrbracket_e &= (\lambda k.k\, \llbracket V \rrbracket_V)\, ((\lambda k.k) \llbracket e \rrbracket_e) \\
&\longrightarrow_{\mathsf{a}} ((\lambda k.k) \llbracket e \rrbracket_e)\, \llbracket V \rrbracket_V \\
&\longrightarrow_{\mathsf{a}} \llbracket e \rrbracket_e\, \llbracket V \rrbracket_V \\
&{}_{\mathsf{a}}\!\longleftarrow (\lambda k.k\, \llbracket V \rrbracket_V)\, \llbracket e \rrbracket_e = \llbracket \langle V \| e \rangle \rrbracket_c
\end{aligned}$$

Case $c \rightsquigarrow c' \Rightarrow \langle V \| \tilde{\mu} \hat{\text{tp}}.c \rangle \rightsquigarrow \langle V \| \tilde{\mu} \hat{\text{tp}}.c' \rangle$:
This case is similar to the case for delimited continuations proved before, we only need to use the induction hypothesis for $\llbracket c \rrbracket_c$ to get:

$$\begin{aligned}
\llbracket V \rrbracket_p \llbracket \tilde{\mu} \hat{\text{tp}}.c \rrbracket_e &= (\lambda k.k\, \llbracket V \rrbracket_V)\, \llbracket c \rrbracket_c \\
&\longrightarrow_{\mathsf{a}} \llbracket c \rrbracket_c\, \llbracket V \rrbracket_V \\
&\overset{*}{\longrightarrow}_{\beta^+} t\, \llbracket V \rrbracket_V \\
&=_{\mathsf{a}} \llbracket c' \rrbracket_c\, \llbracket V \rrbracket_V \\
&{}_{\mathsf{a}^+}\!\!\longleftarrow (\lambda k.k\, \llbracket V \rrbracket_V)\, \llbracket c' \rrbracket_c = \llbracket V \rrbracket_p \llbracket \tilde{\mu} \hat{\text{tp}}.c' \rrbracket_e
\end{aligned}$$

□

Proposition 7.22. *There is no infinite sequence only made of reductions:*

$$\begin{array}{llcl@{\qquad}llcl}
(1) & \langle \mathtt{subst}\ p\, q \| e \rangle & \overset{p \notin V}{\rightsquigarrow} & \langle p \| \tilde{\mu} a.\langle \mathtt{subst}\ a\, q \| e \rangle\rangle & (3) & \langle \mu \hat{\text{tp}}.\langle p \| \hat{\text{tp}} \rangle \| e \rangle & \rightsquigarrow & \langle p \| e \rangle \\
(2) & \langle \mathtt{subst}\ \mathtt{refl}\ q \| e \rangle & \rightsquigarrow & \langle q \| e \rangle & (4) & c[t] & \rightsquigarrow & c[t']
\end{array}$$

Proof. It is sufficient to observe that if we define the following quantities:

1. the quantity of `subst` $p\,q$ with p not a value within a command,
2. the quantity of `subst` within a command,
3. the quantity of $\hat{\text{tp}}$ within a command,
4. the quantity of `wit` terms within a command.

then the rule (1) makes the quantity (1) decrease while preserving the others, (2) makes the quantity (2) decrease and preserves the other, and so on. All in all, we have a bound on the maximal number of steps for the reduction restricted to these four rules. □

Proposition 7.23 (Preservation of normalization). *If $[\![c]\!]_c$ normalizes, then c is also normalizing*

Proof. Reasoning by contraposition, let us assume that c is not normalizing. Then in any infinite reduction sequence from c, according to the previous proposition, there are infinitely many steps that are reflected through the CPS into at least one distinguished step (Proposition 7.21). Thus, there is an infinite reduction sequence from $[\![c]\!]_c$ too. □

Theorem 7.24 (Normalization). *If $c : \Gamma \vdash \Delta$, then c normalizes.*

Proof. Using the preservation of typing (Proposition 7.26), we know that if c is typed in $\mathrm{dL}_{\hat{\mathfrak{tp}}}$, then its image $[\![c]\!]_c$ is also typed. Using the fact that typed terms of the target language are normalizing, we can finally apply the previous proposition to deduce that c normalizes. □

7.3.4 Translation of types

We can now define the translation of types in order to show further that the translation $[\![p]\!]_p$ of a proof p of type A is of type [19]$[\![A]\!]^*$. The type $[\![A]\!]^*$ is the double-negation of a type $[\![A]\!]^+$ that depends on the structure of A. Thanks to the restriction of dependent types to NEF proof terms, we can interpret a dependency in p (resp. t) in $\mathrm{dL}_{\hat{\mathfrak{tp}}}$ by a dependency in p^+ (resp. t^+) in the target language. Lemma 7.17 indeed guarantees that the translation of a NEF proof p will eventually return p^+ to the continuation it is applied to. The translation is defined by:

$$\begin{array}{lcl|lcl}
[\![A]\!]^* & \triangleq & ([\![A]\!]^+ \to \bot) \to \bot & [\![t = u]\!]^+ & \triangleq & t^+ = u^+ \\
[\![\forall x^{\mathbb{N}}.A]\!]^+ & \triangleq & \forall x^{\mathbb{N}}.[\![A]\!]^* & [\![\top]\!]^+ & \triangleq & \top \\
[\![\exists x^{\mathbb{N}}.A]\!]^+ & \triangleq & \exists x^{\mathbb{N}}.[\![A]\!]^+ & [\![\bot]\!]^+ & \triangleq & \bot \\
[\![\Pi a : A.B]\!]^+ & \triangleq & \Pi a : [\![A]\!]^+.[\![B]\!]^* & \mathbb{N}^+ & \triangleq & \mathbb{N}
\end{array}$$

Observe that types depending on a term of type T are translated to types depending on a term of the same type T, because terms can only be of type $\mathbb{N}$. As we shall discuss in Section 7.5.2, this will no longer be the case when extending the domain of terms. We naturally extend the translation for types to the translation of contexts, where we consider unified contexts of the form $\Gamma \cup \Delta$:

$$\begin{array}{lcl}
[\![\Gamma, a : A]\!] & \triangleq & [\![\Gamma]\!]^+, a : [\![A]\!]^+ \\
[\![\Gamma, x : \mathbb{N}]\!] & \triangleq & [\![\Gamma]\!]^+, x : \mathbb{N} \\
[\![\Gamma, \alpha : A^{\perp\!\!\perp}]\!] & \triangleq & [\![\Gamma]\!]^+, \alpha : [\![A]\!]^+ \to \bot.
\end{array}$$

As explained informally in Section 7.1.6 and stated by Lemma 7.17, the translation of a NEF proof term p of type A uses its continuation linearly. In particular, this allows us to refine its type to make it parametric in the return type of the continuation. From a logical point of view, it amounts to replace the double-negation $(A \to \bot) \to \bot$ by Friedman's translation [53]: $\forall R.(A \to R) \to R$. It is worth noticing the correspondences with the continuation monad [46] and the codensity monad. Also, we make plain use here of the fact that the NEF fragment is intuitionistic, so to speak. Indeed, it would be impossible to attribute this type to the translation of a (really) classical proof.

Moreover, we can even make the return type of the continuation dependent on its argument (that is a type of the shape $\Pi a : A.R(a)$), so that the type of $[\![p]\!]_p$ will correspond to the elimination rule:

$$\forall R.(\Pi a : A.R(a) \to R(p^+)).$$

This refinement will make the translation of NEF proofs compatible with the translation of delimited continuations.

[19] To follow the notations in the previous chapters, we could have written $[\![A]\!]_p$ and $[\![A]\!]_V$ instead of $[\![A]\!]^*$ and $[\![A]\!]^+$. To avoid confusion, we preferred to stick with the notation p^+ for the translation of NEF proofs, which are of type $[\![A]\!]^+$ and not necessarily values.

Lemma 7.25 (Typing translation for NEF proofs). *The following holds:*

1. *For any term t, if* $\Gamma \vdash t : \mathbb{N} \mid \Delta$ *then* $\llbracket \Gamma \cup \Delta \rrbracket \vdash \llbracket t \rrbracket_t : \forall X.(\forall x^{\mathbb{N}}.X(x) \to X(t^+))$.
2. *For any NEF proof p, if* $\Gamma \vdash p : A \mid \Delta$ *then* $\llbracket \Gamma \cup \Delta \rrbracket \vdash \llbracket p \rrbracket_p : \forall X.(\Pi a : \llbracket A \rrbracket^+.X(a) \to X(p^+)))$.
3. *For any NEF command c, if* $c : (\Gamma \vdash \Delta, \star : B)$ *then* $\llbracket \Gamma \cup \Delta \rrbracket, \star : \Pi b : B^+.X(b) \vdash \llbracket c \rrbracket_c : X(c^+))$.

Proof. The proof is done by induction on the typing derivation. We only give the key cases of the proof.

- **Case** (`wit`). In $\mathrm{dL}_{\hat{\mathfrak{tp}}}$ the typing rule for `wit` p is the following:

$$\dfrac{\Gamma \vdash p : \exists x^{\mathbb{N}}.A(x) \mid \Delta \quad p \in \mathcal{D}}{\Gamma \vdash \mathtt{wit}\ p : \mathbb{N} \mid \Delta}\ (\mathtt{wit})$$

We want to show that:

$$\llbracket \Gamma \cup \Delta \rrbracket \vdash \lambda k.\llbracket p \rrbracket_p\ (\lambda a.k(\mathtt{wit}\ a)) : \forall X.(\forall x^{\mathbb{N}}.X(x) \to X(\mathtt{wit}\ p^+))$$

By induction hypothesis, we have:

$$\llbracket \Gamma \cup \Delta \rrbracket \vdash \llbracket p \rrbracket_p : \forall Z.(\Pi a : \exists x^{\mathbb{N}} \llbracket A \rrbracket^+.Z(a) \to Z(p^+)),$$

hence it amounts to showing that for any X we can build the following derivation, where we write Γ_k for the context $\llbracket \Gamma \cup \Delta \rrbracket, k : \forall x^{\mathbb{N}} X(x)$:

$$\dfrac{\dfrac{\dfrac{}{\Gamma_k \vdash k : \forall x^{\mathbb{N}} X(x)}\ (\mathrm{Ax}_p) \qquad \dfrac{\dfrac{}{\Gamma_k, a : \exists x^{\mathbb{N}}.\llbracket A \rrbracket^+ \vdash a : \exists x^{\mathbb{N}}.\llbracket A \rrbracket^+}\ (\mathrm{Ax}_p)}{\Gamma_k, a : \exists x^{\mathbb{N}}.\llbracket A \rrbracket^+ \vdash \mathtt{wit}\ a : \mathbb{N}}\ (\mathtt{wit})}{\Gamma_k, a : \exists x^{\mathbb{N}}.\llbracket A \rrbracket^+ \vdash k\,(\mathtt{wit}\ a) : X(\mathtt{wit}\ a)}\ (\forall_E^1)}{\Gamma_k \vdash \lambda a.k(\mathtt{wit}\ a) : \Pi a : \exists x^{\mathbb{N}} \llbracket A \rrbracket^+.X(\mathtt{wit}\ a)}\ (\to_I)$$

- **Case** $(\exists_I)$. In $\mathrm{dL}_{\hat{\mathfrak{tp}}}$ the typing rule for (t,p) is the following:

$$\dfrac{\Gamma \vdash t : \mathbb{N} \mid \Delta \quad \Gamma \vdash p : A(t) \mid \Delta}{\Gamma \vdash (t,p) : \exists x^{\mathbb{N}}.A(x) \mid \Delta}\ \exists_i$$

Hence we obtain by induction:

$$\llbracket \Gamma \cup \Delta \rrbracket \vdash \llbracket t \rrbracket_t : \forall X.(\forall x^{\mathbb{N}} X(x) \to X(t^+)) \qquad (IH_t)$$
$$\llbracket \Gamma \cup \Delta \rrbracket \vdash \llbracket p \rrbracket_p : \forall Y.(\Pi a : A(t^+).Y(a) \to Y(p^+)) \qquad (IH_p)$$

and we want to show that for any Z:

$$\llbracket \Gamma \cup \Delta \rrbracket \vdash \lambda k.\llbracket p \rrbracket_p(\llbracket t \rrbracket_t\ (\lambda xa.k\,(x,a))) : \Pi a : \exists x^{\mathbb{N}}.A.Z(a) \to Z(t^+,p^+).$$

So we need to prove that:

$$\llbracket \Gamma \cup \Delta \rrbracket, k : \Pi q : \exists x^{\mathbb{N}}.A.Z(q) \vdash \llbracket p \rrbracket_p(\llbracket t \rrbracket_t\ (\lambda xa.k\,(x,a))) : Z(t^+,p^+)$$

We let the reader check that such a type is derivable by using $X(x) \triangleq \Pi a : A(x).Z(x,a)$ in the type of $\llbracket t \rrbracket_p$, and using $Y(a) \triangleq Z(t^+,a)$ in the type of $\llbracket p \rrbracket_p$:

$$\dfrac{\dfrac{\Gamma' \vdash [\![t]\!]_t : \ldots \qquad \dfrac{\dfrac{\dfrac{}{k : \Pi q : \exists x^{\mathbb{N}}.A.Z(q) \vdash k : \Pi q : \exists x^{\mathbb{N}} A.Z(q)}\,(\mathrm{Ax}_p) \qquad \dfrac{}{x : \mathbb{N}, a : A(x) \vdash (x,a) : \exists x^{\mathbb{N}}.A}\,(\exists_I)}{k : \Pi q : \exists x^{\mathbb{N}}.A.Z(), x : \mathbb{N}, a : A(x) \vdash k\,(x,a) : Z(x,a)}\,(\to_E)}{k : \Pi q : \exists x^{\mathbb{N}}.A.Z(q) \vdash \lambda xa.k\,(x,a) : \forall x.\Pi a : A(x).Z(x,a)}\,(\forall_I)}{\Gamma' \vdash [\![p]\!]_p \Gamma', k.: \Pi a : \exists x^{\mathbb{N}}.A.Z(a) \vdash [\![t]\!]_t\,(\lambda xa.k\,(x,a)) : \Pi a : A(t^+).Z(t^+,a)}\,(\to_E)}{\Gamma', k : \Pi q : \exists x^{\mathbb{N}}.A.Z(q) \vdash [\![p]\!]_p([\![t]\!]_t\,(\lambda xa.k\,(x,a))) : Z(t^+,p^+)}\,(\to_E)$$

- **Case** (μ). For this case, we could actually conclude directly using the induction hypothesis for c. Rather than that, we do the full proof for the particular case $\mu\star.\langle p \| \tilde{\mu}a.\langle q \| \star\rangle\rangle$, which condensates the proofs for $\mu\star.c$ and the two possible cases $\langle p_N \| e_N\rangle$ and $\langle p_N \| \star\rangle$ of NEF commands. This case corresponds to the following typing derivation in $\mathrm{dL}_{\hat{\mathfrak{tp}}}$:

$$\dfrac{\dfrac{\dfrac{\Pi_p}{\Gamma \vdash p : A \mid \Delta} \qquad \dfrac{\dfrac{\dfrac{\Pi_q}{\Gamma, a : A \vdash q : B \mid \Delta} \qquad \dfrac{}{\cdots \mid \star : B \vdash \Delta, \star : B}}{\langle q \| \star\rangle : \Gamma, a : A \vdash \Delta, \star : B}\,(\textsc{Cut})}{\Gamma \mid \tilde{\mu}a.\langle q \| \star\rangle : A \vdash \Delta, \star : B}\,(\tilde{\mu})}{\langle p \| \tilde{\mu}a.\langle q \| \star\rangle\rangle : \Gamma \mid \Delta, \star : B}\,(\textsc{Cut})}{\Gamma \vdash \mu\star.\langle p \| \tilde{\mu}a.\langle q \| \star\rangle \mid \Delta\rangle : B}\,(\mu)$$

We want to show that for any X we can derive:

$$\Gamma' \vdash \lambda k.[\![p]\!]_p\,(\lambda a.[\![q]\!]_p\,k) : \Pi b : B.X(b) \to X(q^+[p^+/a]).$$

By induction, we have:

$$\begin{array}{rcl} \Gamma' & \vdash & [\![p]\!]_p : \forall Y.(\Pi a : A^+.Y(a) \to Y(p^+)) \\ \Gamma', a : A^+ & \vdash & [\![q]\!]_t : \forall Z.(\Pi b : B^+.Z(b) \to Z(q^+)), \end{array}$$

so that by choosing $Z(b) \triangleq X(b)$ and $Y(a) \triangleq X(q^+)$, we get the expected derivation:

$$\dfrac{\Gamma' \vdash [\![p]\!]_p : \ldots \qquad \dfrac{\dfrac{\Gamma', a : A^+ \vdash [\![q]\!]_p : \ldots \qquad \dfrac{}{k : \Pi b : B.X(b) \vdash k : k : \Pi b : B.X(b)}}{\Gamma', k : \Pi b : B.X(b), a : A^+ \vdash [\![q]\!]_p\,k : X(q^+)}\,(\to_E)}{\Gamma', k : \Pi b : B.X(b) \vdash \lambda a.[\![q]\!]_p\,k : \Pi a : A^+.X(q^+)}\,(\to_I)}{\Gamma', k : \Pi b : B.X(b) \vdash [\![p]\!]_p\,(\lambda a.[\![q]\!]_p\,k) : X(q^+[p^+/a])}\,(\to_E)$$

□

Using the previous Lemma, we can now prove that the CPS translation is well-typed in the general case.

Proposition 7.26 (Preservation of typing)**.** *The translation is well-typed, i.e. the following holds:*

1. *if* $\Gamma \vdash p : A \mid \Delta$ *then* $[\![\Gamma \cup \Delta]\!] \vdash [\![p]\!]_p : [\![A]\!]^*$,
2. *if* $\Gamma \mid e : A \vdash \Delta$ *then* $[\![\Gamma \cup \Delta]\!] \vdash [\![e]\!]_e : [\![A]\!]^+ \to \bot$,
3. *if* $c : \Gamma \vdash \Delta$ *then* $[\![\Gamma \cup \Delta]\!] \vdash [\![c]\!]_c : \bot$.

Proof. The proof is done by induction on the typing derivation, distinguishing cases according to the typing rule used in the conclusion. It is clear that for the NEF cases, Lemma 7.25 implies the result by taking $X(a) = \bot$. The rest of the cases are straightforward, except for the delimited continuations that we detail hereafter. We consider a command $\langle \mu\hat{\mathrm{tp}}.\langle q \| \tilde{\mu}a.\langle p \| \hat{\mathrm{tp}}\rangle\rangle \| e\rangle$ produced by the reduction of the command $\langle \lambda a.p \| q \cdot e\rangle$ with $q \in \text{NEF}$. Both commands are translated by a proof reducing to $(\llbracket q \rrbracket_p\,(\lambda a.\llbracket p \rrbracket_p))\,\llbracket e \rrbracket_e$. The corresponding typing derivation in $\mathrm{dL}_{\hat{\mathrm{tp}}}$ is of the form:

$$\dfrac{\dfrac{\dfrac{\Pi_p}{\Gamma, a : A \vdash p : B \mid \Delta}}{\Gamma \vdash \lambda a.p : \Pi a : A.B \mid \Delta}\,(\to_I) \qquad \dfrac{\dfrac{\Pi_q}{\Gamma \vdash q : A \mid \Delta} \quad \dfrac{\Pi_e}{\Gamma \mid e : B[q/a] \vdash \Delta}}{\Gamma \mid q \cdot e : \Pi a : A.B \vdash \Delta}\,(\to_E)}{\langle \lambda a.p \| q \cdot e\rangle : \Gamma \vdash \Delta}\,(\textsc{Cut})$$

By induction hypothesis for e and p we obtain:

$$\begin{array}{rcl} \Gamma' & \vdash & \llbracket e \rrbracket_e : \llbracket B[q^+] \rrbracket^+ \to \bot \\ \Gamma', a : A^+ & \vdash & \llbracket p \rrbracket_p : \llbracket B[a] \rrbracket^* \\ \Gamma' & \vdash & \lambda a.\llbracket p \rrbracket_p : \Pi a : A^+.\llbracket B[a] \rrbracket^*, \end{array}$$

where $\Gamma' = \llbracket \Gamma \cup \Delta \rrbracket$. Applying Lemma 7.25 for $q \in \text{NEF}$ we can derive:

$$\dfrac{\Gamma' \vdash \llbracket q \rrbracket_p : \forall X.(\Pi a : A^+.X(a) \to X(q^+))}{\Gamma' \vdash \llbracket q \rrbracket_p : (\Pi a : A^+.\llbracket B[a] \rrbracket^* \to \llbracket B[q^+] \rrbracket^*}\,(\forall_E^2)$$

We can thus derive that:

$$\Gamma' \vdash \llbracket q \rrbracket_p\,(\lambda a.\llbracket p \rrbracket_p) : \llbracket B[q^+] \rrbracket^*,$$

and finally conclude that:

$$\Gamma' \vdash (\llbracket q \rrbracket_p\,(\lambda a.\llbracket p \rrbracket_p))\,\llbracket e \rrbracket_e : \bot\,.$$

□

We can finally deduce the correctness of $\mathrm{dL}_{\hat{\mathrm{tp}}}$ through the translation:

Theorem 7.27 (Soundness). *For any $p \in \mathrm{dL}_{\hat{\mathrm{tp}}}$, we have: $\nvdash p : \bot$.*

Proof. Any closed proof term of type $\bot$ would be translated in a closed proof of $(\bot \to \bot) \to \bot$. The correctness of the target language guarantees that such a proof cannot exist. □

7.4 Embedding into Lepigre's calculus

In a recent paper [108], Lepigre presented a classical system allowing the use of dependent types with a semantic value restriction. In practice, the type system of his calculus does not contain a dependent product $\Pi a : A.B$ strictly speaking, but it contains a predicate $a \in A$ allowing the decomposition of the dependent product into

$$\forall a.((a \in A) \to B)$$

as it is usual in Krivine's classical realizability [97]. In his system, the relativization $a \in A$ is restricted to values, so that we can only type $V : V \in A$:

$$\dfrac{\Gamma \vdash_{val} V : A}{\Gamma \vdash_{val} V : V \in A}\,\exists_i$$

However typing judgments are defined up to observational equivalence, so that if t is observationally equivalent to V, one can derive the judgment $t : t \in A$.

Interestingly, as highlighted through the CPS translation by Lemma 7.17, any NEF proof $p : A$ is observationally equivalent to some value p^+, so that we could derive $p : (p \in A)$ from $p^+ : (p^+ \in A)$. The NEF fragment is thus compatible with the semantical value restriction. The converse is obviously false, observational equivalence allowing us to type realizers that would be untyped otherwise[20].

We shall now detail an embedding of $\mathrm{dL}_{\hat{\mathrm{tp}}}$ into Lepigre's calculus, and explain how to transfer normalization and correctness properties along this translation. Actually, his language is more expressive than ours, since it contains records and pattern-matching (we will only use pairs, *i.e.* records with two fields), but it is not stratified: no distinction is made between a language of terms and a language of proofs. We only recall here the syntax for the fragment of Lepigre's calculus we use, for the reduction rules and the type system the reader should refer to [108]:

$$\begin{array}{rcl}
v,w & ::= & x \mid \lambda x.t \mid \{l_1 = v_1, l_2 = v_2\} \\
t,u & ::= & a \mid v \mid t\,u \mid \mu\alpha.t \mid p \mid v.l_i \\
\pi,\rho & ::= & \alpha \mid v \cdot \pi \mid [t]\pi \\
p,q & ::= & t * \pi \\
A,B & ::= & X_n(t_1,\dots,t_n) \mid A \to B \mid \forall a.A \mid \exists a.A \\
 & \mid & \forall X_n.A \mid \{l_1 : A_1, l_2 : A_2\} \mid t \in A
\end{array}$$

Even though records are only defined for values, we can define pairs and projections as syntactic sugar:

$$\begin{array}{rcl}
(t_1,t_2) & \triangleq & (\lambda v_1 v_2.\{l_1 = v_1, l_2 = v_2\})\, t_1\, t_2 \\
\mathsf{fst}(t) & \triangleq & (\lambda x.(x.l_1))\, t \\
\mathsf{snd}(t) & \triangleq & (\lambda x.(x.l_2))\, t \\
A_1 \wedge A_2 & \triangleq & \{l_1 : A_1, l_2 : A_2\}
\end{array}$$

Similarly, only values can be pushed on stacks, but we can define processes[21] with stacks of the shape $t \cdot \pi$ as syntactic sugar:

$$t * u \cdot \pi \quad \triangleq \quad tu * \pi$$

We first define the translation for types (extended for typing contexts) where the predicate $\mathsf{Nat}(x)$ is defined as usual in second-order logic:

$$\mathsf{Nat}(x) \triangleq \forall X.(X(0) \to \forall y.(X(y) \to X(S(y))) \to X(x))$$

and $[\![t]\!]_t$ is the translation of the term t given in Figure 7.9:

$$\begin{array}{rl|rl}
(\forall x^{\mathbb{N}}.A)^* & \triangleq \forall x.(\mathsf{Nat}(x) \to A^*) & (\Pi a : A.B)^* & \triangleq \forall a.((a \in A^*) \to B^*) \\
(\exists x^{\mathbb{N}}.A)^* & \triangleq \exists x.(\mathsf{Nat}(x) \wedge A^*) & & \\
(t = u)^* & \triangleq \forall X.(X([\![t]\!]_t) \to X([\![u]\!]_t)) & (\Gamma, x : \mathbb{N})^* & \triangleq \Gamma^*, x : \mathsf{Nat}(x) \\
\top^* & \triangleq \forall X.(X \to X) & (\Gamma, a : A)^* & \triangleq \Gamma^*, a : A^* \\
\bot^* & \triangleq \forall X.Y(X \to Y) & (\Gamma, \alpha : A^{\perp\!\!\perp})^* & \triangleq \Gamma^*, \alpha : \neg A^*
\end{array}$$

Note that the equality is mapped to Leibniz equality, and that the definitions of $\bot^*$ and $\top^*$ are completely ad hoc, in order to make the conversion rule admissible through the translation.

The translation for terms, proofs, contexts and commands of $\mathrm{dL}_{\hat{\mathrm{tp}}}$, given in Figure 7.9 is almost straightforward. We only want to draw the reader's attention on a few points:

- the equality being translated as Leibniz equality, `refl` is translated as the identity $\lambda a.a$, which also matches with $\top^*$,

[20] In particular, Lepigre's semantical restriction is so permissive that it is not decidable, while it is easy to decide whether a proof term of $\mathrm{dL}_{\hat{\mathrm{tp}}}$ is in NEF.

[21] This will allows to ease the definition of the translation to translate separately proofs and contexts. Otherwise, we would need formally to define $[\![\langle p \| q \cdot e\rangle]\!]_c$ all together by $[\![p]\!]_p [\![q]\!]_p * [\![e]\!]_e$.

$$\begin{array}{ll|ll|ll}
\llbracket x\rrbracket_t & \triangleq x & \llbracket (t,p)\rrbracket_p & \triangleq (\llbracket t\rrbracket_t, \llbracket p\rrbracket_p) & \llbracket q\cdot e\rrbracket_e & \triangleq \llbracket q\rrbracket_p \cdot \llbracket e\rrbracket_e \\
\llbracket n\rrbracket_t & \triangleq \lambda zs.s^n(z) & \llbracket \mu\alpha.c\rrbracket_p & \triangleq \mu\alpha.\llbracket c\rrbracket_c & \llbracket t\cdot e\rrbracket_e & \triangleq \llbracket t\rrbracket_t \cdot \llbracket e\rrbracket_e \\
\llbracket \mathtt{wit}\ p\rrbracket_t & \triangleq \pi_1(\llbracket p\rrbracket_p) & \llbracket \mathtt{prf}\ p\rrbracket_p & \triangleq \pi_2(\llbracket p\rrbracket_p) & \llbracket \tilde\mu a.c\rrbracket_e & \triangleq [\lambda a.\llbracket c\rrbracket_c]\bullet \\
\llbracket a\rrbracket_p & \triangleq a & \llbracket \mathtt{refl}\rrbracket_p & \triangleq \lambda a.a & \llbracket \langle p\|e\rangle\rrbracket_c & \triangleq \llbracket p\rrbracket_p * \llbracket e\rrbracket_e \\
\llbracket \lambda a.p\rrbracket_p & \triangleq \lambda a.\llbracket p\rrbracket_p & \llbracket \mathtt{subst}\ p\,q\rrbracket_p & \triangleq \llbracket p\rrbracket_p\,\llbracket q\rrbracket_p & \llbracket \mu\hat{\mathfrak{tp}}.c\rrbracket_p & \triangleq \mu\alpha.\llbracket c\rrbracket_{\hat{\mathfrak{tp}}} \\
\llbracket \lambda x.p\rrbracket_p & \triangleq \lambda x.\llbracket p\rrbracket_p & \llbracket \alpha\rrbracket_e & \triangleq \alpha & \llbracket \langle p\|\hat{\mathfrak{tp}}\rangle\rrbracket_{\hat{\mathfrak{tp}}} & \triangleq \llbracket p\rrbracket_p
\end{array}$$

$$\llbracket \langle p\|\tilde\mu a.c\rangle\rrbracket_{\hat{\mathfrak{tp}}} \triangleq (\mu\alpha.\llbracket p\rrbracket_p * [\lambda a.\llbracket c\rrbracket_{\hat{\mathfrak{tp}}}]\alpha) * \alpha$$

Figure 7.9: Translation of proof terms into Lepigre's calculus

$$\frac{\Gamma\vdash t:A \quad \Gamma\vdash \pi : A^{\perp\!\!\perp}}{\Gamma\vdash t*\pi : B}\,* \qquad \frac{}{\Gamma\vdash \bullet : \bot^{\perp\!\!\perp}}\,\bullet \qquad \frac{}{\Gamma,\alpha : A^{\perp\!\!\perp}\vdash \alpha : A^{\perp\!\!\perp}}\,\alpha \qquad \frac{\Gamma,\alpha : A^{\perp\!\!\perp}\vdash t : A}{\Gamma\vdash \mu\alpha.t : A}\,\mu$$

$$\frac{\Gamma\vdash \pi : (A[x:=t])^{\perp\!\!\perp}}{\Gamma\vdash \pi : (\forall x A)^{\perp\!\!\perp}}\,\forall_l \qquad \frac{\Gamma\vdash t:A \quad \Gamma\vdash \pi : B^{\perp\!\!\perp}}{\Gamma\vdash t\cdot\pi : (A\Rightarrow B)^{\perp\!\!\perp}}\,\Rightarrow_l \qquad \frac{\Gamma\vdash t:A\Rightarrow B \quad \Gamma\vdash \pi : B^{\perp\!\!\perp}}{\Gamma\vdash [t]\pi : A^{\perp\!\!\perp}}\,\mathtt{let}$$

Figure 7.10: Extension of Lepigre's typing rules for stacks

- the strong existential is encoded as a pair, hence `wit` (resp. `prf`) is mapped to the projection π_1 (resp. π_2).

In [108], the coherence of the system is justified by a realizability model, and the type system does not allow us to type stacks. Thus, we cannot formally prove that the translation preserves typing, unless we extend the type system in which case this would imply the adequacy. We might also directly prove the adequacy of the realizability model (through the translation) with respect to the typing rules of $\mathrm{dL}_{\hat{\mathfrak{tp}}}$. We will detail here a proof of adequacy using the former method in the following. We then need to extend Lepigre's system to be able to type stacks. In fact, the proof of adequacy [108, Theorem 6] suggests a way to do so, since any typing rule for typing stacks is valid as long as it is adequate with the realizability model.

We denote by $A^{\perp\!\!\perp}$ the type A when typing a stack, in the same fashion we use to go from a type A in a left rule of two-sided sequent to the type $A^{\perp\!\!\perp}$ in a one-sided sequent (see the remark at the end of Section 7.1.3). We also add a distinguished bottom stack $\bullet$ to the syntax, which is given the most general type $\bot^{\perp\!\!\perp}$. We change the rules $(*)$ and (μ) of the original type system in [108] and add rules for stacks, whose definitions are guided by the proof of the adequacy [108, Theorem 6] in particular by the $(\Rightarrow_e)$-case. These rules are given in Figure 7.10.

We shall now show that these rules are adequate with respect to the realizability model defined in [108, Section 2].

Proposition 7.28 (Adequacy). *Let* Γ *be a (valid) context,* A *be a formula with* $FV(A)\subset dom(\Gamma)$ *and* σ *be a substitution realizing* Γ*. The following statements hold:*

- *if* $\Gamma\vdash_{val} v : A$ *then* $v\sigma\in\llbracket A\rrbracket_\sigma$*;*
- *if* $\Gamma\vdash\pi : A^{\perp\!\!\perp}$ *then* $\pi\sigma\in\llbracket A\rrbracket_\sigma^{\perp}$*;*
- *if* $\Gamma\vdash t : A$ *then* $t\sigma\in\llbracket A\rrbracket_\sigma^{\perp\perp}$*.*

Proof. The proof is done by induction on the typing derivation, we only need to do the proof for the rules we define above (all the other cases correspond to the proof of [108, Theorem 6]).

($\bullet$) By definition, we have $[\![\bot]\!]_\sigma = [\![\forall X.X]\!]_\sigma = \emptyset$, thus for any stack π, we have $\pi \in [\![\bot]\!]_\sigma^\perp = \Pi$. In particular, $\bullet \in [\![\bot]\!]_\sigma^\perp$.

(α) By hypothesis, σ realizes $\Gamma, \alpha : A^{\perp\!\!\!\perp}$ from which we obtain $\alpha\sigma = \sigma(\alpha) \in [\![A]\!]_\sigma^\perp$.

($*$) We need to show that $t\sigma * \pi\sigma \in [\![B]\!]_\sigma^{\perp\perp}$, so we take $\rho \in [\![B]\!]_\sigma^\perp$ and show that $(t\sigma * \pi\sigma) * \rho \in \perp\!\!\!\perp$. By anti-reduction, it is enough to show that $(t\sigma * \pi\sigma) \in \perp\!\!\!\perp$. This is true by induction hypothesis, since $t\sigma \in [\![A]\!]_\sigma^{\perp\perp}$ and $\pi\sigma \in [\![A]\!]_\sigma^\perp$.

(μ) The proof is the very same as in [108, Theorem 6].

($\forall_l$) By induction hypothesis, we have that $\pi\sigma \in [\![A[x := t]]\!]_\sigma^\perp$. We need to show that $[\![A[x := t]]\!]_\sigma^\perp \subseteq [\![\forall x.A]\!]_\sigma^\perp$, which follows from the fact $[\![\forall x.A]\!]_\sigma = \bigcap_{t\in\Lambda}[\![A[x := t]]\!]_\sigma \subseteq [\![A[x := t]]\!]_\sigma$.

($\Rightarrow_l$) If t is a value v, by induction hypothesis, we have that $v\sigma \in [\![A]\!]_\sigma$ and $\pi\sigma \in [\![B]\!]_\sigma^\perp$ and we need to show that $v\sigma \cdot \pi\sigma \in [\![A \Rightarrow B]\!]_\sigma^\perp$. The proof is already done in the case ($\Rightarrow_e$) (see [108, Theorem 6]). Otherwise, by induction hypothesis, we have that $t\sigma \in [\![A]\!]_\sigma^{\perp\perp}$ and $\pi\sigma \in [\![B]\!]_\sigma^\perp$ and we need to show that $t\sigma \cdot \pi\sigma \in [\![A \Rightarrow B]\!]_\sigma^\perp$. So we consider $\lambda x.u \in [\![A \Rightarrow B]\!]_\sigma$, and show that $\lambda x.u * t\sigma \cdot \pi\sigma \in \perp\!\!\!\perp$. We can take a reduction step, and prove instead that $t\sigma * [\lambda x.u]\pi\sigma \in \perp\!\!\!\perp$. This amounts to showing that $[\lambda x.u]\pi \in [\![A]\!]_\sigma^\perp$, which is already proven in the case ($\Rightarrow_e$).

(`let`) We need to show that for all $v \in [\![A]\!]_\sigma$, $v * [t\sigma]\pi\sigma \in \perp\!\!\!\perp$. Taking a step of reduction, it is enough to have $t\sigma * v \cdot \pi\sigma \in \perp\!\!\!\perp$. This is true since by induction hypothesis, we have $t\sigma \in [\![A \Rightarrow B]\!]_\sigma^{\perp\perp}$ and $\pi\sigma \in [\![B]\!]_\sigma^\perp$, thus $v \cdot \pi\sigma \in [\![A \Rightarrow B]\!]_\sigma^\perp$.

□

It only remains to show that the translation we defined in Figure 7.9 preserves typing to conclude the proof of Proposition 7.30.

Lemma 7.29. *If $\Gamma \vdash p : A \mid \Delta$ (in $\mathrm{dL}_{\hat{\mathrm{tp}}}$), then $(\Gamma \cup \Delta)^* \vdash [\![p]\!]_p : A^*$ (in Lepigre's extended system). The same holds for contexts, and if $c : \Gamma \vdash \Delta$ then $(\Gamma \cup \Delta)^* \vdash [\![c]\!]_c : \bot$.*

Proof. The proof is an induction on the typing derivation $\Gamma \vdash p : A \mid \Delta$. Note that in a way, the translation of a delimited continuation decompiles it to simulate in a natural deduction fashion the reduction of the applications of functions to stacks (that could have generated the same delimited continuations in $\mathrm{dL}_{\hat{\mathrm{tp}}}$), while maintaining the frozen context (at top-level) outside of the active command (just like a delimited continuation would do). This trick allows us to avoid the problem of dependencies conflict in the typing derivation. For instance, assuming that $[\![q_1]\!]_p$ (resp. $[\![q_2]\!]_p$) reduces to a value V_1 (resp. V_2)

we have:

$$\begin{array}{rl}
& \llbracket\langle\mu\hat{\mathfrak{tp}}.\langle q_1\|\tilde{\mu}a_1.\langle q_2\|\tilde{\mu}a_2.\langle p\|\hat{\mathfrak{tp}}\rangle\rangle\rangle\|e\rangle\rrbracket_c \\
= & \mu\alpha.(\mu\alpha.(\llbracket q_1\rrbracket_p * [\lambda a_1.\llbracket\langle q_2\|\tilde{\mu}a_2.\langle p\|\hat{\mathfrak{tp}}\rangle\rangle\rrbracket_{\hat{\mathfrak{tp}}}]\alpha) * \alpha) * \llbracket e\rrbracket_e \\
\succ & \mu\alpha.(\llbracket q_1\rrbracket_p * [\lambda a_1.\llbracket\langle q_2\|\tilde{\mu}a_2.\langle p\|\hat{\mathfrak{tp}}\rangle\rangle\rrbracket_{\hat{\mathfrak{tp}}}]\alpha) * \llbracket e\rrbracket_e \\
\succ & \llbracket q_1\rrbracket_p * [\lambda a_1.\llbracket\langle q_2\|\tilde{\mu}a_2.\langle p\|\hat{\mathfrak{tp}}\rangle\rangle\rrbracket_{\hat{\mathfrak{tp}}}]\llbracket e\rrbracket_e \\
\succ^* & \llbracket q_2\rrbracket_p * [\lambda a_2.\llbracket p\rrbracket_p[V_1/a_1]]\llbracket e\rrbracket_e \\
\succ^* & \llbracket p\rrbracket_p[\llbracket V_1\rrbracket_p/a_1][\llbracket V_2\rrbracket_p/a_2] * \llbracket e\rrbracket_e \\
{}^*\!\prec & \llbracket q_2\rrbracket_p * [\lambda a_2.\llbracket p\rrbracket_p[V_1/a_1]]\llbracket e\rrbracket_e \\
{}^*\!\prec & \llbracket q_1\rrbracket_p * [\lambda a_1 a_2.\llbracket p\rrbracket_p)]\llbracket q_2\rrbracket_p \cdot \llbracket e\rrbracket_e \\
{}^*\!\prec & (\lambda a_1 a_2.\llbracket p\rrbracket_p) * \llbracket q_1\rrbracket_p \cdot \llbracket q_2\rrbracket_p \cdot \llbracket e\rrbracket_e = \llbracket\langle\lambda a_1\lambda a_2.p\|q_1\cdot q_2\cdot e\rangle\rrbracket_c
\end{array}$$

where we observe that $\llbracket e\rrbracket_e$ is always kept outside of the computations, and where each command $\langle q_i\|\tilde{\mu}a_i.c_{\hat{\mathfrak{tp}}}\rangle$ is decompiled into $(\mu\alpha.\llbracket q_i\rrbracket_p * [\lambda a_i.\llbracket c_{\hat{\mathfrak{tp}}}\rrbracket_{\hat{\mathfrak{tp}}}].\alpha) * \llbracket e\rrbracket_e$, simulating the (natural deduction style) reduction of $\lambda a_i.\llbracket c_{\hat{\mathfrak{tp}}}\rrbracket_{\hat{\mathfrak{tp}}} * \llbracket q_i\rrbracket_p \cdot \llbracket e\rrbracket_e$. These terms correspond somehow to the translations of former commands typable without types dependencies.

□

As a corollary we get a proof of the adequacy of dL$_{\hat{\mathfrak{tp}}}$ typing rules with respect to Lepigre's realizability model.

Proposition 7.30 (Adequacy). *If* $\Gamma \vdash p : A \mid \Delta$ *and* σ *is a substitution realizing* $(\Gamma\cup\Delta)^*$*, then* $\llbracket p\rrbracket_p\sigma \in \llbracket A^*\rrbracket_\sigma^{\perp\perp}$.

This immediately implies the soundness of dL$_{\hat{\mathfrak{tp}}}$, since a closed proof p of type $\bot$ would be translated as a realizer of $\top \to \bot$, so that $\llbracket p\rrbracket_p\ \lambda x.x$ would be a realizer of $\bot$, which is impossible. Furthermore, the translation clearly preserves normalization (that is that for any c, if c does not normalize then neither does $\llbracket c\rrbracket_c$), and thus the normalization of dL$_{\hat{\mathfrak{tp}}}$ is a consequence of adequacy.

Theorem 7.31 (Soundness). *For any proof p in* dL$_{\hat{\mathfrak{tp}}}$*, we have:* $\nvdash p : \bot$.

It is worth noting that without delimited continuations, we would not have been able to define an adequate translation, since we would have encountered the same problem as for the CPS translation (see Section 7.1.6).

7.5 Toward dLPA$^\omega$: further extensions

As we explained in the preamble of Section 7.1, we defined dL and dL$_{\hat{\mathfrak{tp}}}$ as minimal languages containing all the potential sources of inconsistency we wanted to mix: classical control, dependent types, and a sequent calculus presentation. It had the benefit to focus our attention on the difficulties inherent to the issue, but on the other hand, the language we obtain is far from being as expressive as other usual proof systems. We claimed our system to be extensible, thus we shall now discuss this matter. We will then be ready to define dLPA$^\omega$ in the next chapter, which is the sequent calculus presentation of dPA$^\omega$ using the techniques developed in this chapter.

7.5.1 Intuitionistic sequent calculus

There is not much to say on this topic, but it is worth mentioning that dL and dL$_{\hat{\mathfrak{tp}}}$ could be easily restricted to obtain an intuitionistic framework. Indeed, just like for the passage from LK to LJ, we pretend that it is enough to restrict the syntax of proofs to allow only one continuation variable (that is one conclusion on the right-hand side of sequent) to obtain an intuitionistic calculus. In particular, in such a setting, all proofs will be NEF, and every result we obtained will still hold.

7.5.2 Extending the domain of terms

Throughout the chapter, we only worked with terms of a unique type $\mathbb{N}$, hence it is natural to wonder whether it is possible to extend the domain of terms in $\mathrm{dL}_{\hat{\mathfrak{tp}}}$, for instance with terms in the simply-typed λ-calculus. A good way to understand the situation is to observe what happens through the CPS translation. We saw that a *term* t of type $T = \mathbb{N}$ is translated into a *proof* t^* which is roughly of type $T^* = \neg\neg T^+ = \neg\neg\mathbb{N}$, from which we can extract a *term* t^+ of type $\mathbb{N}$.

However, if T was for instance the function type $\mathbb{N} \to \mathbb{N}$ (resp. $T \to U$), we would only be able to extract a *proof* of type $T^+ = \mathbb{N} \to \neg\neg\mathbb{N}$ (resp. $T^+ \to U^*$). There is no hope in general to extract a function $f : \mathbb{N} \to \mathbb{N}$ from such a term, since such a proof could be of the form $\lambda x.p$, where p might backtrack to a former position, for instance before it was extracted, and furnish another proof. Such a proof is no longer a witness in the usual sense, but rather a realizer of $f \in \mathbb{N} \to \mathbb{N}$ in the sense of Krivine classical realizability. This accounts for a well-know phenomenon in classical logic, where witness extraction is limited to formulas in the Σ^1_0-fragment [119]. It also corresponds to the type we obtain for the image of a dependent product $\Pi a : A.B$, that is translated to a type $\neg\neg\Pi a : A^+.B^*$ where the dependence is in a proof of type A^+. This phenomenon is not surprising and was already observed for other CPS translations for type theories with dependent types [13].

Nevertheless, if the extraction is not possible in the general case, our situation is more specific. Indeed, we only need to consider proofs that are obtained as translation of terms, which can only contains NEF proofs in $\mathrm{dL}_{\hat{\mathfrak{tp}}}$. In particular, the obtained proofs cannot drop continuations, which was the whole point of the restriction to the NEF fragment. Hence we could again refine the translation of types, similarly to what we did in Lemma 7.25. Once more, this refinement would also coincide with a computational property similar to Lemma 7.17, expressing the fact that the extraction can be done simply by passing the identity as a continuation[22]. This witnesses the fact that for any function t in the source language, there exists a term t^+ in the target language which represents the same function, even tough the translation of t is a proof $[\![t]\!]$.

To sum up, this means that we can extend the domain of terms in $\mathrm{dL}_{\hat{\mathfrak{tp}}}$ (in particular, it should affect neither the subject reduction nor the soundness), but the stratification between terms and proofs is to be lost through a CPS translation. If the target language is a non-stratified type theory (most of the presentation of type theories are in this case), then it becomes possible to force the extraction of terms of the same type through the translation.

Another solution would consist to define a separate translation for terms. Indeed, as it was reflected by Lemma 7.17, since neither terms nor the NEFproofs they may contain need continuations, they can be directly translated. The corresponding translation is actually an embedding which maps every pure term (without `wit` p) to itself, and which performs the reduction of NEF proofs p to proofs p^+ so as to eliminate every μ binder. Such a translation would intuitively reflect an abstract machine where the reduction of terms (and the NEF proofs inside) is performed in an external machine. If this solution is arguably a bit *ad hoc*, it is nonetheless correct and is maybe a good way to take advantage of the stratified presentation.

[22] To be precise, for each arrow in the type, a double-negation (or its refinement) would be inserted. For instance, to recover a function of type $\mathbb{N} \to \mathbb{N}$ from a term $t : \neg\neg(\mathbb{N} \to \neg\neg\mathbb{N})$ (where $\neg\neg A$ is in fact more precise, at least $\forall R.(A \to R) \to R$), the continuation need to be forced at each level: $\lambda x.t\,I\,x\,I : \mathbb{N} \to \mathbb{N}$. We do not want to enter into to much details on this here, as it would lead us to much more than a paragraph to define the objects formally, but we claim that we could reproduce the results obtained for terms of type $\mathbb{N}$ in a language with terms representing arithmetic functions in finite types.

7.5.3 Adding expressiveness

From the point of view of the proof language (that is of the tools we have to build proofs), dL$_{\hat{\mathrm{tp}}}$ only enjoys the presence of a dependent sum and a dependent product over terms, as well as a dependent product at the level of proofs (which subsume the non-dependent implication). If this is obviously enough to encode the usual constructors for pairs (p_1,p_2) (of type $A_1 \land A_2$), injections $\iota_i(p)$ (of type $A_1 \lor A_2$), etc..., it seems reasonable to wonder whether such constructors can be directly defined in the language of proofs. In fact, this is the case, and we claim that is possible to define the constructors for proofs (for instance (p_1,p_2)) together with their destructors in the contexts (in that case $\tilde{\mu}(a_1,a_2).c$), with the appropriate typing rules. In practice, it is enough to:

- extend the definitions of the NEF fragment according to the chosen extension,
- extend the call-by-value reduction system, opening if needed the constructors to reduce it to a value,
- in the dependent typing mode, make some pattern-matching within the list of dependencies for the destructors.

The soundness of such extensions can be justified either by extending the CPS translation, or by defining a translation to Lepigre's calculus (which already allows records and pattern-matching over general constructors) and proving the adequacy of the translation with respect to the realizability model.

For instance, for the case of the pairs, we can extend the syntax with:

$$p ::= \cdots \mid (p_1,p_2) \qquad\qquad e ::= \cdots \mid \tilde{\mu}(a_1,a_2).c$$

We then need to add the corresponding typing rules (plus a third rule to type $\tilde{\mu}(a_1,a_2).c$ in regular mode:

$$\frac{\Gamma \vdash p_1 A_1 \mid \Delta \quad \Gamma \vdash p_2 : A_2) \mid \Delta}{\Gamma \vdash (p_1,p_2) : (A_1 \land A_2) \mid \Delta}\land_r \qquad\qquad \frac{c : \Gamma, a_1 : A_1, a_2 : A_2 \vdash_d \Delta, \hat{\mathrm{tp}} : B; \sigma\{(a_1,a_2)|p\}}{\Gamma \mid \tilde{\mu}(a_1,a_2).c : (A_1 \land A_2) \vdash_d \Delta, \hat{\mathrm{tp}} : B; \sigma\{\cdot|p\}}\land_l$$

and the reduction rules:

$$\langle (p_1,p_2) \| e\rangle \rightsquigarrow \langle p_1 \| \tilde{\mu}a_1.\langle p_2 \| \tilde{\mu}a_2.\langle (a_1,a_2) \| e\rangle\rangle\rangle \qquad\qquad \langle (V_1,V_2) \| \tilde{\mu}(a_1,a_2).c\rangle \rightsquigarrow c[V_1/a_1, V_2/a_2]$$

We let the reader check that these rules preserve subject reduction, and suggest the following CPS translations:

$$\begin{array}{rcl}
[\![(p_1,p_2)]\!]_p & \triangleq & \lambda^{\bullet} k.[\![p_1]\!]_p\,(\lambda^{\bullet} a_1.[\![p_2]\!]_p\,(\lambda^{\bullet} a_2.k\,(a_1,a_2))) \\
[\![(V_1,V_2)]\!]_V & \triangleq & \lambda^{\bullet} k.k\,([\![V_1]\!]_V, [\![V_2]\!]_V) \\
[\![\tilde{\mu}(a_1,a_2).c]\!]_e & \triangleq & \lambda p.\,\mathtt{split}\ p\ \mathtt{as}\,(a_1,a_2)\ \mathtt{in}\,[\![c]\!]_c
\end{array}$$

which allows us to prove that the calculus remains correct with these extensions.

We claim that this methodology furnishes in first approximation an approach to the question "*Can I extend this with ... ?*". In particular, it should be enough to get closer to a realistic programming language and extend the language with inductive fix-point operators. We make plain use of these ideas in the next chapter.

7.5.4 A fully sequent-style dependent calculus

While the aim of this chapter was to design a sequent-style calculus embedding dependent types, we only presented the Π-type in sequent-style. Indeed, we wanted to be sure above all else that it was possible to define a sound sequent-calculus with the key ingredients of dependent types, even if these were presented in a natural deduction spirit. Rather than having left-rules, we presented the existential type and the equality type with the following elimination rules:

$$\frac{\Gamma \vdash p : \exists x^{\mathbb{N}}.A(x) \mid \Delta;\sigma \quad p \in \mathcal{D}}{\Gamma \vdash \mathtt{prf}\ p : A(\mathtt{wit}\ p) \mid \Delta;\sigma}\ \mathtt{prf} \qquad\qquad \frac{\Gamma \vdash p : t = u \mid \Delta;\sigma \quad \Gamma \vdash q : B[t/x] \mid \Delta;\sigma}{\Gamma \vdash \mathtt{subst}\ p\,q : B[u/x] \mid \Delta;\sigma}\ \mathtt{subst}$$

However, it is now easy to have both rules in a sequent calculus fashion, for instance we could rather have contexts of the shape $\tilde{\mu}(x,a).c$ (to be dual to proofs (t,p)) and $\tilde{\mu}^{=}.c$ (dual to $\mathtt{refl}$). We could then define the following typing rules (where we add another list of dependencies δ for terms, to compensate the conversion from $A[t]$ to $A[u]$ in the former ($\mathtt{subst}$)-rule):

$$\frac{c : \Gamma, x : \mathbb{N}, a : A(x) \vdash_d \Delta; \sigma\{(x,a)|p\}}{\Gamma \mid \tilde{\mu}(x,a).c : \exists x^{\mathbb{N}}.A(x) \vdash_d \Delta; \sigma\{\cdot|p\}}\ \exists_l \qquad\qquad \frac{c : \Gamma \vdash \Delta; \delta\{t|u\}}{\Gamma \mid \tilde{\mu}^{=}.c : t = u \vdash \Delta; \delta}\ ({=_l})$$

and define $\mathtt{prf}\ p$ and $\mathtt{subst}\ p\,q$ as syntactic sugar:

$$\mathtt{prf}\ p \triangleq \mu\mathfrak{tp}.\langle p \| \tilde{\mu}(x,a).\langle a \| \mathfrak{tp}\rangle\rangle \qquad\qquad \mathtt{subst}\ p\,q \triangleq \mu\alpha.\langle p \| \tilde{\mu}^{=}.\langle q \| \alpha\rangle\rangle.$$

Observe that $\mathtt{prf}\ p$ is now only definable if p is a NEF proof term. For any $p \in$ NEF and any variables a, α, $A(\mathtt{wit}\ p)$ is in $A(\mathtt{wit}\ (x,a))_{\{(x,a)|p\}}$ which allows us to derive (using this in the (Cut)-rule) the admissibility of the former ($\mathtt{prf}$)-rule:

$$\frac{\dfrac{\Gamma \vdash p : \exists x^{\mathbb{N}}.A \mid \Delta;\sigma \qquad \dfrac{\dfrac{\dfrac{\overline{a : A(x) \vdash a : A(x)}}{a : A(x) \vdash a : A(\mathtt{wit}\ (x,a))} \equiv \dfrac{A(\mathtt{wit}\ p) \in A(\mathtt{wit}\ (x,a))_{\{(x,a)|p\}}}{\Gamma \mid \hat{\mathfrak{tp}} : A(\mathtt{wit}\ (x,a)) \vdash_d \hat{\mathfrak{tp}} : A(\mathtt{wit}\ p) \mid \Delta}\ \mathtt{cut}}{\langle a \| \alpha\rangle : \Gamma, x : \mathbb{N}, a : A(x) \vdash_d \Delta, \hat{\mathfrak{tp}} : A(\mathtt{wit}\ p); \sigma\{(x,a)|p\}}}{\Gamma \mid \tilde{\mu}(x,a).\langle a \| \hat{\mathfrak{tp}}\rangle : \exists x^{\mathbb{N}}.A \vdash_d \Delta, \hat{\mathfrak{tp}} : A(\mathtt{wit}\ p); \sigma\{\cdot|p\}}}{\langle p \| \tilde{\mu}(x,a).\langle a \| \alpha\rangle\rangle : \Gamma \vdash_d \Delta, \hat{\mathfrak{tp}} : A(\mathtt{wit}\ p); \sigma\{\cdot|p\}}\ (\textsc{Cut})}{\Gamma \vdash \mu\hat{\mathfrak{tp}}.\langle p \| \tilde{\mu}(x,a).\langle a \| \hat{\mathfrak{tp}}\rangle\rangle : A(\mathtt{wit}\ p) \mid \Delta}$$

Using the fact that $\delta(B[u]) = \delta(B[t])$, we get that the former ($\mathtt{subst}$)-rule is admissible:

$$\frac{\dfrac{\Gamma \vdash p : t = u \mid \Delta; \qquad \dfrac{\dfrac{\Gamma \vdash q : B[t] \mid \Delta;\sigma \quad \dfrac{}{\Gamma \mid \alpha : B[u] \vdash \alpha : B[u] \mid \Delta}\ (\textsc{Ax}_l)}{\langle q \| \alpha\rangle : \Gamma \vdash \Delta, \alpha : B[u]; \delta\{t|u\}}\ (\textsc{Cut})}{\Gamma \mid \tilde{\mu}^{=}.\langle q \| \alpha\rangle : t = u \vdash \Delta, \alpha : B[u]; \delta}\ ({=_l})}{\langle p \| \tilde{\mu}^{=}.\langle q \| \alpha\rangle\rangle : \Gamma \vdash \Delta, \alpha : B[u]; \delta}\ (\textsc{Cut})}{\Gamma \vdash \mu\alpha.\langle p \| \tilde{\mu}^{=}.\langle q \| \alpha\rangle\rangle : B[u] \mid \Delta; \delta}\ (\mu) \qquad .$$

As for the reduction rules, we can define the following (call-by-value) reductions:

$$\langle (V_t, V) \| \tilde{\mu}(x,a).c\rangle \rightsquigarrow c[V_t/x][V/a] \qquad\qquad \langle \mathtt{refl} \| \tilde{\mu}^{=}.c\rangle \rightsquigarrow c$$

and check that they advantageously simulate the previous rules (the expansion rules become useless):

$$\begin{array}{rl} \langle \mathtt{subst}\ \mathtt{refl}\ q \| e\rangle \rightsquigarrow \langle q \| e\rangle & \qquad \langle \mathtt{subst}\ p\,q \| e\rangle \overset{p \notin V}{\rightsquigarrow} \langle p \| \tilde{\mu}a.\langle \mathtt{subst}\ a\,q \| e\rangle\rangle \\ \langle \mathtt{prf}\ (V_t, V_p) \| e\rangle \rightsquigarrow \langle V \| e\rangle & \qquad \langle \mathtt{prf}\ p \| e\rangle \rightsquigarrow \langle \mu\hat{\mathfrak{tp}}.\langle p \| \tilde{\mu}a.\langle \mathtt{prf}\ a \| \hat{\mathfrak{tp}}\rangle\rangle \| e\rangle. \end{array}$$

7.6 Conclusion

In this chapter, we presented dL, a sequent calculus that combines dependent types and classical control by means of a syntactic restriction to values. We proved in Section 7.1 the normalization of dL for typed terms, as well as its soundness. This calculus can be extended with delimited continuations, which permits us to extend the syntactic restriction for dependent types to the fragment of negative-elimination-free proofs. The resulting calculus $\mathrm{dL}_{\hat{\mathtt{tp}}}$, that we presented in Section 7.2, is suitable for the definition of a dependently typed translation to an intuitionistic type theory. As shown in Section 8.3, this translation guarantees both the normalization and the soundness of $\mathrm{dL}_{\hat{\mathtt{tp}}}$. Furthermore, a similar translation can be designed to embedded $\mathrm{dL}_{\hat{\mathtt{tp}}}$ into Lepigre's calculus. As explained in Section 7.4, this provides an alternative way of proving the soundness of $\mathrm{dL}_{\hat{\mathtt{tp}}}$.

Several directions remain to be explored. We plan to investigate possible extensions of the syntactic restriction we defined, and its connections with notions such as with Fürhmann's *thunkability* [54] or Munch-Maccagnoni's *linearity* [127]. Furthermore, it might be of interest to check whether this restriction could make dependent types compatible with other side-effects, in presence of classical logic or not. More generally, we would like to better understand the possible connections between our calculus and the categorical models for dependently typed theory.

On a different perspective, the continuation-passing style translation we defined is at the best of our knowledge a novel contribution, even without considering the classical part. In particular, our translation allows us to use computations (as in the call-by-push value terminology) within dependent types with a call-by-value evaluation strategy, and without any thunking construction. It might be the case that this translation could be adapted to justify extensions of other dependently typed calculi, or provide typed translations between them.

8- dLPA$^\omega$: a sequent calculus with dependent types for classical arithmetic

Drawing on the calculi we studied in the last chapters, we shall now present dLPA$^\omega$, a sequent calculus version of Herbelin's dPA$^\omega$. This calculus provides us with dependent types restricted to the NEF fragment, for which dLPA$^\omega$ is an extension of dL$_{\hat{\mathfrak{tp}}}$. Indeed, in addition to the language of dL$_{\hat{\mathfrak{tp}}}$, dLPA$^\omega$ has terms for classical arithmetic in finite types (PA$^\omega$). More importantly, it includes a lazily evaluated co-fixpoint operator. To this end, the calculus uses a shared store, as in the $\overline{\lambda}_{[l\upsilon\tau\star]}$-calculus.

We first present the language of dLPA$^\omega$ with its type system and its reduction rules. We prove that the calculus verifies the property of subject reduction and that it is as expressive as dPA$^\omega$. In particular, the proof terms for $AC_{\mathbb{N}}$ and DC of dPA$^\omega$ can be directly defined in dLPA$^\omega$. We then apply once again the methodology of Danvy's semantic artifacts to derive a small-step calculus, from which we deduce a continuation-passing-style translation and a realizability interpretation. Both artifacts are somehow a combination of the corresponding ones that we developed for the $\overline{\lambda}_{[l\upsilon\tau\star]}$-calculus and dL$_{\hat{\mathfrak{tp}}}$.

In some sense, there will not be any real novelties in this chapter. In particular, most of the proofs resemble a lot to the corresponding ones in the previous chapters. Yet, as dLPA$^\omega$ gathers all the expressive power and features of the $\overline{\lambda}_{[l\upsilon\tau\star]}$-calculus and dL$_{\hat{\mathfrak{tp}}}$, the different proofs also combined all the tools and tricks used in each case. They are therefore very technical and long, in particular proofs by induction require the tedious verification of multiple cases which are very similar to cases of proofs we already did. We will hence sketch them most of the time, trying to highlight the most interesting parts.

Normalization of dLPA$^\omega$

The main result of this chapter consists in the normalization of dLPA$^\omega$, from which it is easy to convince ourself that dPA$^\omega$ normalizes too[1]. We sketch a proof of normalization through a continuation-passing-style translation, which would rely on the normalization of System F_Υ. We then give a detailed proof through the realizability interpretation.

Nonetheless, we should say before starting this chapter that we already have a guardrail for the normalization. Indeed, we already proved the normalization of a simply-typed classical call-by-need calculus and we explained that the proof was scalable to the same calculus with a second-order type system. Yet, co-fixpoints are definable in a second-order calculus[2], for instance a stream for the infinite conjunction $A(0) \wedge A(1) \wedge \ldots$ can be obtained through the formula $\exists X.[X(0) \wedge \forall x^{\mathbb{N}}.(X(x) \to A(x) \wedge X(S(x)))]$. Besides, the presence of dependent types does not bring any risk of loosing the normalization, since erasing the dependencies in types yield a system with the exact same computational behavior. Hence the normalization of dL$_{\hat{\mathfrak{tp}}}$ and the one of the second-order $\overline{\lambda}_{[l\upsilon\tau\star]}$-calculus should be enough, *a priori*, to guarantee the normalization of dLPA$^\omega$.

[1] As explained in Chapter 5, we will not bother with a formal proof of this statement, neither will we prove any properties on the preservation of dPA$^\omega$ reduction rules through the embedding in dLPA$^\omega$. Indeed, we are already satisfied with the normalization of dLPA$^\omega$, which is as expressive as dPA$^\omega$ and which allows for the same proof terms for dependent and countable choice.

[2] A definition in the framework of dPA$^\omega$ is given in [70].

Another handwavy explanation could consist in arguing that we could authorize infinite stores in the $\overline{\lambda}_{[lv\tau\star]}$-calculus without altering its normalization. Indeed, from the point of view of existing programs (which are finite and typed in finite contexts), they are computing with a finite knowledge of the memory (and we proved that all the terms were suitable for a store extension[3]). Note that in the store, we could theoretically replace any co-fixpoint that produces a stream by the (fully developped) stream in question. Due to the presence of backtracks in co-fixpoints, the store would contain all the possible streams (possibly an infinite number of it) produced when reducing co-fixpoints. In this setting, if a term were to perform an infinite number of reductions steps, it would necessarily have to explore an infinite number of cells in the pre-computed memory, independently from its production. This should not be possible either.

The latter argument is actually quite close from Herbelin's original proof sketch, which this thesis is precisely trying to replace with a more formal proof. So that these unprecise explanations should be taken more as spoilers of the final result than as proof sketches. We shall now present formally dLPA$^\omega$ and prove its normalization, which will then not come as a surprise.

8.1 dLPA$^\omega$: a sequent calculus with dependent types for classical arithmetic

8.1.1 Syntax

The language of dLPA$^\omega$ should not be a surprise either. It is based on the syntax of $\mathrm{dL}_{\hat{\mathfrak{tp}}}$, extended with the expressive power of dPA$^\omega$ and with explicit stores as in the $\overline{\lambda}_{[lv\tau\star]}$-calculus. We stick to a stratified presentation of dependent types, which we find very convenient to separate terms and proof terms which are handled differently.

The syntax of terms is extended as in dPA$^\omega$ to include functions $\lambda x.t$ and applications tu, as well as a recursion operator $\mathtt{rec}^t_{xy}[t_0 \mid t_S]$, so that terms represent objects in arithmetic of finite types.

As for proof terms (and contexts, commands), they are now defined with all the expressiveness of dPA$^\omega$ (see Chapter 5). Each constructor in the syntax of formulas is reflected by a constructor in the syntax of proofs and by the dual co-proof (*i.e.* destructor) in the syntax of evaluation contexts. Namely, the syntax is an extension of $\mathrm{dL}_{\hat{\mathfrak{tp}}}$'s syntax which now includes:

- the usual proofs $\mu\alpha.c$ and contexts $\tilde{\mu}a.c$ of the $\lambda\mu\tilde{\mu}$-calculus;
- pairs (p_1,p_2), which inhabit the conjunction type $A_1 \land A_2$;
- co-pairs $\tilde{\mu}(a_1,a_2).c$, which bind the variables a_1 and a_2 in the command c;
- injections $\iota_i(p)$ for the logical disjunction;
- co-injections or pattern-matching $\tilde{\mu}[a_1.c_1|a_2.c_2]$ which bind the variables a_1 in c_1 and a_2 in c_2;
- pairs (t,p) where t is a term and p a proof, which inhabit the dependent sum type $\exists x^T.A$;
- dual co-pairs $\tilde{\mu}(x,a).c$ which bind the (term and proof) variables x and a in the command c;
- functions $\lambda x.p$, which inhabit the dependent product type $\forall x^T.A$;
- dual stacks $t \cdot e$, where t is a term and e a context whose type might be dependent in t;
- functions $\lambda a.p$, which inhabit the dependent product type $\Pi a : A.B$;
- dual stacks $q \cdot e$, where q is a term and e a context whose type might be dependent in q if q is NEF;
- a proof term $\mathtt{refl}$ which is the proof of atomic equalities $t = t$;
- the dual destructor $\tilde{\mu}{=}.c$ which allows to type the command c modulo an equality of terms;

[3] See Lemma 6.16 for the realizability interpretation and Lemma 6.31 for the CPS translation of the $\overline{\lambda}_{[lv\tau\star]}$-calculus.

Closures	l	$::=$	$c\tau$
Commands	c	$::=$	$\langle p \Vert e \rangle$
Proof terms	p,q	$::=$	$a \mid \iota_i(p) \mid (p,q) \mid (t,p) \mid \lambda x.p \mid \lambda a.p \mid \mathtt{refl}$
		$\mid$	$\mathtt{ind}^t_{ax}[p_0 \mid p_S] \mid \mathtt{cofix}^t_{bx}[p] \mid \mu\alpha.c \mid \mu\hat{\mathtt{tp}}.c_{\hat{\mathtt{tp}}}$
Proof values	V	$::=$	$a \mid \iota_i(V) \mid (V,V) \mid (V_t,V) \mid \lambda x.p \mid \lambda a.p \mid \mathtt{refl}$
Contexts	e	$::=$	$f \mid \alpha \mid \tilde{\mu}a.c\tau$
Forcing contexts	f	$::=$	$[] \mid \tilde{\mu}[a_1.c_1 \mid a_2.c_2] \mid \tilde{\mu}(a_1,a_2).c \mid \tilde{\mu}(x,a).c$
		$\mid$	$t \cdot e \mid p \cdot e \mid \tilde{\mu}{=}.c$
Stores	τ	$::=$	$\varepsilon \mid \tau[a := p_\tau] \mid \tau[\alpha := e]$
Storables	p_τ	$::=$	$V \mid \mathtt{ind}^{V_t}_{ax}[p_0 \mid p_S] \mid \mathtt{cofix}^{V_t}_{bx}[p]$
Terms	t,u	$::=$	$x \mid 0 \mid S(t) \mid \mathtt{rec}^t_{xy}[t_0 \mid t_S] \mid \lambda x.t \mid t\ u \mid \mathtt{wit}\ p$
Terms values	V_t	$::=$	$x \mid S^n(0) \mid \lambda x.t$
Delimited	$c_{\hat{\mathtt{tp}}}$	$::=$	$\langle p_N \Vert e_{\hat{\mathtt{tp}}} \rangle \mid \langle p \Vert \hat{\mathtt{tp}} \rangle$
continuations	$e_{\hat{\mathtt{tp}}}$	$::=$	$\tilde{\mu}a.c_{\hat{\mathtt{tp}}}\tau \mid \tilde{\mu}[a_1.c_{\hat{\mathtt{tp}}} \mid a_2.c'_{\hat{\mathtt{tp}}}] \mid \tilde{\mu}(a_1,a_2).c_{\hat{\mathtt{tp}}} \mid \tilde{\mu}(x,a).c_{\hat{\mathtt{tp}}}$
NEF	c_N	$::=$	$\langle p_N \Vert e_N \rangle$
	p_N,q_N	$::=$	$a \mid \iota_i(p_N) \mid (p_N,q_N) \mid (t,p_N) \mid \lambda x.p \mid \lambda a.p \mid \mathtt{refl}$
		$\mid$	$\mathtt{ind}^t_{ax}[p_N \mid q_N] \mid \mathtt{cofix}^t_{bx}[p_N] \mid \mu\star.c_N \mid \mu\hat{\mathtt{tp}}.c_{\hat{\mathtt{tp}}}$
	e_N	$::=$	$\star \mid \tilde{\mu}[a_1.c_N \mid a_2.c'_N] \mid \tilde{\mu}a.c_N\tau \mid \tilde{\mu}(a_1,a_2).c_N \mid \tilde{\mu}(x,a).c_N$

Figure 8.1: The language of dLPA$^{\omega}$

- operators $\mathtt{ind}^t_{ax}[p_0 \mid p_S]$ and $\mathtt{cofix}^t_{bx}[p]$, as in dPA$^{\omega}$, for inductive and coinductive reasoning;
- delimited continuations through proofs $\mu\hat{\mathtt{tp}}..c_{\mathtt{tp}}$ and the context $\hat{\mathtt{tp}}$;
- a distinguished context $[]$ of type $\bot$, which allows us to reason ex-falso.

As in dL$_{\hat{\mathtt{tp}}}$, the syntax of NEF proofs, contexts and commands is defined as a restriction of the previous syntax. Here again, they are defined (modulo α-conversion) with only one distinguished context variable $\star$ (and consequently only one binder $\mu\star.c$) and without stacks of the shape $t \cdot e$ or $q \cdot e$ (to avoid applications). The commands $c_{\mathtt{tp}}$ within delimited continuations are again defined as commands of the shape $\langle p \Vert \mathtt{tp} \rangle$ or formed by a NEF proof and a context of the shape $\tilde{\mu}a.c_{\hat{\mathtt{tp}}}\tau$, $\tilde{\mu}[a_1.c_{\hat{\mathtt{tp}}}|a_2.c'_{\hat{\mathtt{tp}}}]$, $\tilde{\mu}(a_1,a_2).c_{\hat{\mathtt{tp}}}$ or $\tilde{\mu}(x,a).c_{\hat{\mathtt{tp}}}$.

We adopt a call-by-value evaluation strategy except for fixpoint operators[4] which are evaluated in a lazy way. To this purpose, we use stores in the spirit of the $\overline{\lambda}_{[lv\tau\star]}$-calculus, which are thus defined as lists of bindings of the shape $[a := p]$ where p is a value or a (co-)fixpoint, and of bindings of the shape $[\alpha := e]$ where e is any context. We assume that each variable occurs at most once in a store τ, therefore we reason up to α-reduction and we assume the capability of generating fresh names. Apart from evaluation contexts of the shape $\tilde{\mu}a.c$ and co-variables α, all the contexts are forcing contexts since they eagerly require a value to be reduced. The resulting language is given in Figure 8.1.

[4] To highlight the duality between inductive and coinductive fixpoints, we evaluate both in a lazy way. Even though this is not indispensable for inductive fixpoints, we find this approach more natural in that we can treat both in a similar way in the small-step reduction system and thus through the CPS translation or the realizability interpretation.

Basic rules

$$\begin{array}{lrcl}
 & \langle \lambda x.p \| V_t \cdot e\rangle\tau & \to & \langle p[V_t/x]\|e\rangle\tau \\
(q \in \textsc{nef}) & \langle \lambda a.p \| q \cdot e\rangle\tau & \to & \langle \mu\hat{\mathfrak{tp}}.\langle q\|\tilde{\mu}a.\langle p\|\hat{\mathfrak{tp}}\rangle\rangle\|e\rangle\tau \\
(q \notin \textsc{nef}) & \langle \lambda a.p \| q \cdot e\rangle\tau & \to & \langle q\|\tilde{\mu}a.\langle p\|e\rangle\rangle\tau \\
(e \neq e_{\hat{\mathfrak{tp}}}) & \langle \mu\alpha.c\|e\rangle\tau & \to & c\tau[\alpha := e] \\
 & \langle V\|\tilde{\mu}a.c\tau'\rangle\tau & \to & c\tau[a := V]\tau'
\end{array}$$

Elimination rules

$$\begin{array}{rcl}
\langle \iota_i(V)\|\tilde{\mu}[a_1.c_1 \mid a_2.c_2]\rangle\tau & \to & c_i\tau[a_i := V] \\
\langle (V_1,V_2)\|\tilde{\mu}(a_1,a_2).c\rangle\tau & \to & c\tau[a_1 := V_1][a_2 := V_2] \\
\langle (V_t,V)\|\tilde{\mu}(x,a).c\rangle\tau & \to & (c[t/x])\tau[a := V] \\
\langle \mathtt{refl}\|\tilde{\mu}{=}.c\rangle\tau & \to & c\tau
\end{array}$$

Delimited continuations

$$\begin{array}{lrcl}
(\text{if } c\tau \to c\tau') & \langle \mu\hat{\mathfrak{tp}}.c\|e\rangle\tau & \to & \langle \mu\hat{\mathfrak{tp}}.c\|e\rangle\tau' \\
 & \langle \mu\alpha.c\|e_{\hat{\mathfrak{tp}}}\rangle\tau & \to & c[e_{\hat{\mathfrak{tp}}}/\alpha]\tau \\
 & \langle \mu\hat{\mathfrak{tp}}.\langle p\|\hat{\mathfrak{tp}}\rangle\|e\rangle\tau & \to & \langle p\|e\rangle\tau
\end{array}$$

Call-by-value

$$\begin{array}{lrcl}
(a \text{ fresh}) & \langle \iota_i(p)\|e\rangle\tau & \to & \langle p\|\tilde{\mu}a.\langle \iota_i(a)\|e\rangle\rangle\tau \\
(a_1,a_2 \text{ fresh}) & \langle (p_1,p_2)\|e\rangle\tau & \to & \langle p_1\|\tilde{\mu}a_1.\langle p_2\|\tilde{\mu}a_2.\langle (a_1,a_2)\|e\rangle\rangle\rangle\tau \\
(a \text{ fresh}) & \langle (V_t,p)\|e\rangle\tau & \to & \langle p\|\tilde{\mu}a.\langle (V_t,a)\|e\rangle\rangle\tau
\end{array}$$

Laziness

$$\begin{array}{lrcl}
(a \text{ fresh}) & \langle \mathtt{cofix}^{V_t}_{bx}[p]\|e\rangle\tau & \to & \langle a\|e\rangle\tau[a := \mathtt{cofix}^{V_t}_{bx}[p]] \\
(a \text{ fresh}) & \langle \mathtt{ind}^{V_t}_{bx}[p_0 \mid p_S]\|e\rangle\tau & \to & \langle a\|e\rangle\tau[a := \mathtt{ind}^{V_t}_{bx}[p_0 \mid p_S]]
\end{array}$$

Lookup

$$\begin{array}{lrcl}
 & \langle V\|\alpha\rangle\tau[\alpha := e]\tau' & \to & \langle V\|e\rangle\tau[\alpha := e]\tau' \\
 & \langle a\|f\rangle\tau[a := V]\tau' & \to & \langle V\|a\rangle\tau[a := V]\tau' \\
(b' \text{ fresh}) & \langle a\|f\rangle\tau[a := \mathtt{cofix}^{V_t}_{bx}[p]]\tau' & \to & \langle p[V_t/x][b'/b]\|\tilde{\mu}a.\langle a\|f\rangle\tau'\rangle\tau[b' := \lambda y.\mathtt{cofix}^{y}_{bx}[p]] \\
 & \langle a\|f\rangle\tau[a := \mathtt{ind}^{0}_{bx}[p_0 \mid p_S]]\tau' & \to & \langle p_0\|\tilde{\mu}a.\langle a\|f\rangle\tau'\rangle\tau \\
(b' \text{ fresh}) & \langle a\|f\rangle\tau[a := \mathtt{ind}^{S(t)}_{bx}[p_0 \mid p_S]]\tau' & \to & \langle p_S[t/x][b'/b]\|\tilde{\mu}a.\langle a\|f\rangle\tau'\rangle\tau[b' := \mathtt{ind}^{t}_{bx}[p_0 \mid p_S]]
\end{array}$$

Terms

$$\begin{array}{lrcl}
(\text{if } t \longrightarrow_\beta t') & T[t]\tau & \to & T[t']\tau \\
(\forall\alpha, \langle p\|\alpha\rangle\tau \to \langle (t,p')\|\alpha\rangle\tau) & T[\mathtt{wit}\ p]\tau & \longrightarrow_\beta & T[t] \\
 & (\lambda x.t)V_t & \longrightarrow_\beta & t[V_t/x] \\
 & \mathtt{rec}^{0}_{xy}[t_0 \mid t_S] & \longrightarrow_\beta & t_0 \\
 & \mathtt{rec}^{S(u)}_{xy}[t_0 \mid t_S] & \longrightarrow_\beta & t_S[u/x][\mathtt{rec}^{u}_{xy}[t_0 \mid t_S]/y]
\end{array}$$

where:

$$\begin{array}{rcl}
C_t[\ ] & ::= & \langle ([\ ],p)\|e\rangle \mid \langle \mathtt{ind}^{[\ ]}_{ax}[p_0 \mid p_S]\|e\rangle \mid \langle \mathtt{cofix}^{[\ ]}_{bx}[p]\|e\rangle \mid \langle \lambda x.p\|[\ ]\cdot e\rangle \\
T[\ ] & ::= & C_t[\ ] \mid T[[\ ]u] \mid T[\mathtt{rec}^{[\ ]}_{xy}[t_0 \mid t_S]]
\end{array}$$

Figure 8.2: Reduction rules of dLPA$^\omega$

8.1.2 Reduction rules

Concerning the reduction system of dLPA$^\omega$, which is given in Figure 8.2, there is not much to say. The basic rules are those of the call-by-value $\lambda\mu\tilde{\mu}$-calculus and of $\mathrm{dL}_{\hat{\mathfrak{tp}}}$. The rules for delimited continuations are exactly the same as in $\mathrm{dL}_{\hat{\mathfrak{tp}}}$, except that we have to prevent $\hat{\mathfrak{tp}}$ from being caught and stored by a proof $\mu\alpha.c$. We thus distinguish two rules for commands of the shape $\langle\mu\alpha.c\|e\rangle$, depending on whether e is of the shape $e_{\hat{\mathfrak{tp}}}$ or not. In the former case, we perform the substitution $[e_{\hat{\mathfrak{tp}}}/\alpha]$, which is linear since $\mu\alpha.c$ is necessarily NEF. We should also mention in passing that we abuse the syntax in every other rules, since e should actually refer to e or $e_{\mathfrak{tp}}$ (or the reduction of delimited continuations would be stuck). Elimination rules correspond to commands where the proof is a constructor (say of pairs) applied to values, and where the context is the matching destructor. Call-by-value rules correspond to (ς) rule of Wadler's sequent calculus [161]. The next rules express the fact that (co-)fixpoints are lazily stored, and reduced only if their value is eagerly demanded by a forcing context. Lastly, terms are reduced according to the usual β-reduction, with the operator `rec` computing with the usual recursion rules. It is worth noting that the stratified presentation allows to define the reduction of terms as external: within proofs and contexts, terms are reduced in place. Consequently, as in $\mathrm{dL}_{\hat{\mathfrak{tp}}}$ the very same happen for NEF proofs embedded within terms. Computationally speaking, this corresponds indeed to the intuition that terms are reduced on an external device.

8.1.3 Typing rules

The language of types and formulas is the same as for dPA$^\omega$. As explained, terms are simply typed, with the set of natural numbers as the sole ground type. The formulas are inductively built on atomic equalities of terms, by means of conjunctions, disjunctions, first-order quantifications, dependent products and co-inductive formulas. As in $\mathrm{dL}_{\hat{\mathfrak{tp}}}$, the dependent product $\Pi a : A.B$ corresponds to the usual implication if a does not occur in the conclusion B. Formulas and types are formally defined by:

$$\begin{array}{lrcl}
\textbf{Types} & T,U & ::= & \mathbb{N} \mid T \to U \\
\textbf{Formulas} & A,B & ::= & \top \mid \bot \mid t = u \mid A \wedge B \mid A \vee B \mid \forall x^T.A \mid \exists x^T.A \mid \Pi a : A.B \mid \nu^t_{x,f}A.
\end{array}$$

Formulas are considered up to equational theory on terms, as often in Martin-Löf's intensional type theory. We denote by $A \equiv B$ the reflexive-transitive-symmetric closure of the relation $\triangleright$ induced by the reduction of terms and NEF proofs as follows:

$$\begin{array}{rcll}
A[t] & \triangleright & A[t'] & \text{whenever} \quad t \to_\beta t' \\
A[p] & \triangleright & A[q] & \text{whenever} \quad \forall\alpha\,(\langle p\|\alpha\rangle \to \langle q\|\alpha\rangle)
\end{array}$$

in addition to the reduction rules for equality and for coinductive formulas:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
0 = S(t) & \triangleright & \bot & S(t) = S(u) & \triangleright & t = u \\
S(t) = 0 & \triangleright & \bot & \nu^t_{fx}A & \triangleright & A[t/x][\nu^y_{fx}A/f(y) = 0]
\end{array}$$

We work with one-sided sequents[5] where typing contexts are defined by:

$$\textbf{Typing contexts} \qquad \Gamma,\Gamma' \quad ::= \quad \varepsilon \mid \Gamma, x : T \mid \Gamma, a : A \mid \Gamma, \alpha : A^{\perp\!\!\perp} \mid \Gamma, \hat{\mathfrak{tp}} : A^{\perp\!\!\perp}.$$

using the notation $\alpha : A^{\perp\!\!\perp}$ for an assumption of the refutation of A. This allows us to mix hypotheses over terms, proofs and contexts while keeping track of the order in which they are added (which is necessary because of the dependencies). We assume that a variable occurs at most once in a typing context.

[5] This is essentially an aesthetic choice, which we hope to ease the readability of sequents. On top of that, it avoids us to deal with unified contexts $\Gamma \cup \Delta$ (see Section 4.2.3.2) as we would have done with two-sided sequents.

We define nine syntactic kinds of typing judgments:

- six in regular mode, that we write $\Gamma \vdash^{\sigma} J$:
  1. $\Gamma \vdash^{\sigma} t : T$ for typing terms,
  2. $\Gamma \vdash^{\sigma} p : A$ for typing proofs,
  3. $\Gamma \vdash^{\sigma} e : A^{\perp\!\!\perp}$ for typing contexts,
  4. $\Gamma \vdash^{\sigma} c$ for typing commands,
  5. $\Gamma \vdash^{\sigma} c\tau$ for typing closures,
  6. $\Gamma \vdash^{\sigma} \tau' : (\Gamma';\sigma')$ for typing stores;
- three more for the dependent mode, that we write $\Gamma \vdash_d J;\sigma$:
  7. $\Gamma \vdash_d e : A^{\perp\!\!\perp};\sigma$ for typing contexts,
  8. $\Gamma \vdash_d c;\sigma$ for typing commands,
  9. $\Gamma \vdash_d c\tau;\sigma$ for typing closures.

In each case, σ is a list of dependencies—we explain the presence of a list of dependencies in each case thereafter—, which are still defined from the following grammar:

$$\sigma ::= \varepsilon \mid \sigma\{p|q\}$$

The substitution on formulas according to a list of dependencies σ is defined by:

$$\varepsilon(A) \triangleq \{A\} \qquad\qquad \sigma\{p|q\}(A) \triangleq \begin{cases} \sigma(A[q/p]) & \text{if } q \in \textsc{nef} \\ \sigma(A) & \text{otherwise} \end{cases}$$

Because the language of proof terms now include constructors for pairs, injections, etc, the notation $A[q/p]$ does not refer to usual substitutions properly speaking: p can be a pattern (for instance (a_1, a_2)) and not only a variable.

We shall attract the reader's attention to the fact that all typing judgments include a list of dependencies. As in the $\overline{\lambda}_{[lv\tau\star]}$-calculus, when a proof or a context is caught by a binder, say V and $\tilde{\mu}a$, the substitution $[V/a]$ is not performed but rather put in the store: $\tau[a := V]$. This forces us to slightly change the rules from $\mathrm{dL}_{\hat{\mathfrak{tp}}}$. Indeed, consider for instance the reduction of a dependent function $\lambda a.p$ (of type $\Pi a : A.B$) applied to a stack $V \cdot e$:

$$\langle \lambda a.p \| V \cdot e\rangle\tau \rightarrow \langle \mu\hat{\mathfrak{tp}}.\langle V \| \tilde{\mu}a.\langle p \| \hat{\mathfrak{tp}}\rangle\rangle \| e\rangle\tau \rightarrow \langle \mu\hat{\mathfrak{tp}}.\langle p \| \hat{\mathfrak{tp}}\rangle \| e\rangle\tau[a := V] \rightarrow \langle p \| e\rangle\tau[a := V]$$

which we examined in details in the previous chapter (see Section 7.1.3). In $\mathrm{dL}_{\hat{\mathfrak{tp}}}$, the reduced command was $\langle p[V/a] \| e\rangle$, which was typed with the (Cut) rule over the formula $B[V/a]$. In the present case, p still contains the variable a, whence his type is still $B[a]$, whereas the type of e is $B[V]$. We thus need to compensate the missing substitution.

We are mostly left with two choices. Either we mimic the substitution in the type system, which would amount to the following typing rule:

$$\frac{\Gamma,\Gamma' \vdash \tau(c) \quad \Gamma \vdash \tau : \Gamma'}{\Gamma \vdash c\tau} \qquad \begin{array}{l} \text{where:} \\ \tau[\alpha := e](c) \triangleq \tau(c) \end{array} \quad \left| \quad \begin{array}{ll} \tau[a := p_N](c) \triangleq \tau(c[p_N/a]) & (p \in \textsc{nef}) \\ \tau[a := p](c) \triangleq \tau(c) & (p \notin \textsc{nef}) \end{array} \right.$$

Or we type stores in the spirit of the $\overline{\lambda}_{[lv\tau\star]}$-calculus, and we carry along the derivations all the bindings susceptible to be used in types, which constitutes again a list of dependencies.

The former solution has the advantage of solving the problem before typing the command, but it has the flaw of performing computations which would not occur in the reduction system. For instance, the substitution $\tau(c)$ could duplicate co-fixpoints (and their typing derivations), which would never happen in the calculus. That is the reason why we privilege the other solution, which is closer to the calculus in our opinion. Yet, it has the inconvenient that if forces us to carry a dependencies list even in regular mode. Since this list is fixed (it does not evolve in the derivation except when stores occur), we differentiate the denotation of regular typing judgments, written $\Gamma \vdash^{\sigma} J$, from the one judgments in dependent mode, which we write $\Gamma \vdash_d J;\sigma$ to highlight that σ grows along derivations. The type system we obtain is given in Figure 8.3.

Regular types

$$\frac{\Gamma \vdash^{\sigma} \tau : (\Gamma';\sigma') \quad \Gamma,\Gamma' \vdash^{\sigma\sigma'} p : A}{\Gamma \vdash^{\sigma} \tau[a := p] : (\Gamma', a : A; \sigma'\{a|p\})}\ (\tau_p) \qquad \frac{\Gamma \vdash^{\sigma} \tau : (\Gamma';\sigma') \quad \Gamma,\Gamma' \vdash^{\sigma\sigma'} \alpha : A^{\perp\!\!\perp}}{\Gamma \vdash^{\sigma} \tau[\alpha := e] : (\Gamma', \alpha : A^{\perp\!\!\perp}; \sigma')}\ (\tau_e)$$

$$\frac{\Gamma \vdash^{\sigma} p : A \quad \Gamma \vdash^{\sigma} e : B^{\perp\!\!\perp} \quad \sigma(A) = \sigma(B)}{\Gamma \vdash^{\sigma} \langle p \| e \rangle}\ (\textsc{Cut}) \qquad \frac{\Gamma,\Gamma' \vdash^{\sigma\sigma'} c \quad \Gamma \vdash^{\sigma} \tau : (\Gamma';\sigma')}{\Gamma \vdash c\tau}\ (l)$$

$$\frac{(a : A) \in \Gamma}{\Gamma \vdash^{\sigma} a : A}\ (\textsc{Ax}_r) \qquad \frac{(\alpha : A^{\perp\!\!\perp}) \in \Gamma}{\Gamma \vdash^{\sigma} \alpha : A^{\perp\!\!\perp}}\ (\textsc{Ax}_l) \qquad \frac{\Gamma, \alpha : A^{\perp\!\!\perp} \vdash^{\sigma} c}{\Gamma \vdash^{\sigma} \mu\alpha.c : A}\ (\mu)$$

$$\frac{\Gamma, a : A \vdash^{\sigma} c\tau}{\Gamma \vdash^{\sigma} \tilde{\mu}a.c\tau : A^{\perp\!\!\perp}}\ (\tilde{\mu}) \qquad \frac{\Gamma \vdash^{\sigma} p_1 : A \quad \Gamma \vdash^{\sigma} p_2 : B}{\Gamma \vdash^{\sigma} (p_1,p_2) : A \wedge B}\ (\wedge_r) \qquad \frac{\Gamma, a_1 : A_1, a_2 : A_2 \vdash^{\sigma} c}{\Gamma \vdash^{\sigma} \tilde{\mu}(a_1,a_2).c : (A_1 \wedge A_2)^{\perp\!\!\perp}}\ (\wedge_l)$$

$$\frac{\Gamma \vdash^{\sigma} p : A_i}{\Gamma \vdash^{\sigma} \iota_i(p) : A_1 \vee A_2}\ (\vee_r) \qquad \frac{\Gamma, a_1 : A_1 \vdash^{\sigma} c_1 \quad \Gamma, a_2 : A_2 \vdash^{\sigma} c_2}{\Gamma \vdash^{\sigma} \tilde{\mu}[a_1.c_1 \mid a_2.c_2] : (A_1 \vee A_2)^{\perp\!\!\perp}}\ (\vee_l)$$

$$\frac{\Gamma \vdash^{\sigma} p : A[t/x] \quad \Gamma \vdash^{\sigma} t : T}{\Gamma \vdash^{\sigma} (t,p) : \exists x^{T}.A}\ (\exists_r) \qquad \frac{\Gamma, x : T, a : A \vdash^{\sigma} c}{\Gamma \vdash^{\sigma} \tilde{\mu}(x,a).c : (\exists x^{T}.A)^{\perp\!\!\perp}}\ (\exists_l)$$

$$\frac{\Gamma, x : T \vdash^{\sigma} p : A}{\Gamma \vdash^{\sigma} \lambda x.p : \forall x^{T}.A}\ (\forall_r) \qquad \frac{\Gamma \vdash^{\sigma} t : T \quad \Gamma \vdash^{\sigma} e : A[t/x]^{\perp\!\!\perp}}{\Gamma \vdash^{\sigma} t \cdot e : (\forall x^{T}.A)^{\perp\!\!\perp}}\ (\forall_l) \qquad \frac{\Gamma \vdash^{\sigma} t : \mathbb{N}}{\Gamma \vdash^{\sigma} \mathtt{refl} : t = t}\ \mathtt{refl}$$

$$\frac{\Gamma \vdash^{\sigma} p : A \quad \Gamma \vdash^{\sigma} e : A[u/t]}{\Gamma \vdash^{\sigma} \tilde{\mu}^{=}.\langle p \| e \rangle : (t = u)^{\perp\!\!\perp}}\ (=_l) \qquad \frac{\Gamma \vdash^{\sigma} p : A \quad A \equiv B}{\Gamma \vdash^{\sigma} p : B}\ (\equiv_r) \qquad \frac{\Gamma \vdash^{\sigma} e : A^{\perp\!\!\perp} \quad A \equiv B}{\Gamma \vdash^{\sigma} e : B^{\perp\!\!\perp}}\ (\equiv_l)$$

$$\frac{\Gamma, a : A \vdash^{\sigma} p : B}{\Gamma \vdash^{\sigma} \lambda a.p : \Pi a : A.B}\ (\to_r) \qquad \frac{\Gamma \vdash^{\sigma} q : A \quad \Gamma \vdash^{\sigma} e : B[q/a]^{\perp\!\!\perp} \quad \text{if } q \notin \textsc{nef} \text{ then } a \notin A}{\Gamma \vdash^{\sigma} q \cdot e : (\Pi a : A.B)^{\perp\!\!\perp}}\ (\to_l)$$

$$\frac{}{\Gamma \vdash^{\sigma} [] : \bot^{\perp\!\!\perp}}\ \bot \qquad \frac{\Gamma \vdash^{\sigma} t : \mathbb{N} \quad \Gamma \vdash^{\sigma} p_0 : A[0/x] \quad \Gamma, x : T, a : A \vdash^{\sigma} p_S : A[S(x)/x]}{\Gamma \vdash^{\sigma} \mathtt{ind}_{ax}^{t}[p_0 \mid p_S] : A[t/x]}\ (\mathtt{ind})$$

$$\frac{\Gamma \vdash^{\sigma} t : T \quad \Gamma, f : T \to \mathbb{N}, x : T, b : \forall y^{T}.f(y) = 0 \vdash^{\sigma} p : A \quad f \text{ positive in } A}{\Gamma \vdash^{\sigma} \mathtt{cofix}_{bx}^{t}[p] : \nu_{fx}^{t}A}\ (\mathtt{cofix})$$

Dependent mode

$$\frac{\Gamma, \hat{\mathsf{tp}} : A^{\perp\!\!\perp} \vdash_d c_{\hat{\mathsf{tp}}}; \sigma}{\Gamma \vdash^{\sigma} \mu\hat{\mathsf{tp}}.c_{\hat{\mathsf{tp}}} : A}\ (\mu\hat{\mathsf{tp}}) \qquad \frac{\sigma(A) = \sigma(B)}{\Gamma, \hat{\mathsf{tp}} : A^{\perp\!\!\perp}, \Gamma' \vdash_d \hat{\mathsf{tp}} : B^{\perp\!\!\perp}; \sigma\{\cdot|p\}}\ (\hat{\mathsf{tp}})$$

$$\frac{\Gamma,\Gamma' \vdash_d c_{\hat{\mathsf{tp}}}; \sigma\sigma' \quad \Gamma \vdash^{\sigma} \tau : (\Gamma';\sigma')}{\Gamma \vdash_d c_{\hat{\mathsf{tp}}}\tau; \sigma}\ (l_d) \qquad \frac{\Gamma,\Gamma' \vdash^{\sigma} p : A \quad \Gamma, \hat{\mathsf{tp}} : B^{\perp\!\!\perp}, \Gamma' \vdash_d e : A^{\perp\!\!\perp}; \sigma\{\cdot|p\}}{\Gamma, \hat{\mathsf{tp}} : B^{\perp\!\!\perp}, \Gamma' \vdash_d \langle p \| e \rangle; \sigma}\ (\textsc{Cut}_d)$$

$$\frac{\Gamma, a : A \vdash_d c_{\hat{\mathsf{tp}}}\tau'; \sigma\{a|p_N\}}{\Gamma \vdash_d \tilde{\mu}a.c_{\hat{\mathsf{tp}}}\tau' : A^{\perp\!\!\perp}; \sigma\{\cdot|p_N\}}\ (\tilde{\mu}_d) \qquad \frac{\Gamma, x : T, a : A \vdash_d c_{\hat{\mathsf{tp}}}; \sigma\{(x,a)|p_N\}}{\Gamma \vdash_d \tilde{\mu}(x,a).c_{\hat{\mathsf{tp}}} : (\exists x^{T} A)^{\perp\!\!\perp}; \sigma\{\cdot|p_N\}}\ (\exists_l^d)$$

$$\frac{\Gamma, a_1 : A_1, a_2 : A_2 \vdash_d c_{\hat{\mathsf{tp}}}; \sigma\{(a_1,a_2)|p_N\}}{\Gamma \vdash_d \tilde{\mu}(a_1,a_2).c_{\hat{\mathsf{tp}}} : (A_1 \wedge A_2)^{\perp\!\!\perp}; \sigma\{\cdot|p_N\}}\ (\wedge_l^d) \qquad \frac{\Gamma, a_i : A_i \vdash_d c_{\hat{\mathsf{tp}}}^{i}; \sigma\{\iota_i(a_i)|p_N\}) \quad \forall i \in \{1,2\}}{\Gamma \vdash_d \tilde{\mu}[a_1.c_{\hat{\mathsf{tp}}}^{1} \mid a_2.c_{\hat{\mathsf{tp}}}^{2}] : (A_1 \vee A_2)^{\perp\!\!\perp}; \sigma\{\cdot|p_N\}}\ (\vee_l^d)$$

Terms

$$\frac{}{\Gamma \vdash^{\sigma} 0 : \mathbb{N}}\ (0) \qquad \frac{\Gamma \vdash^{\sigma} t : \mathbb{N}}{\Gamma \vdash^{\sigma} S(t) : \mathbb{N}}\ (S) \qquad \frac{\Gamma, x : U \vdash^{\sigma} t : T}{\Gamma \vdash^{\sigma} \lambda x.t : U \to T}\ (\lambda) \qquad \frac{\Gamma \vdash^{\sigma} t : U \to T \quad \Gamma \vdash^{\sigma} u : U}{\Gamma \vdash^{\sigma} t\,u : T}\ (@)$$

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash^{\sigma} x : T}\ (\textsc{Ax}_t) \qquad \frac{\Gamma \vdash^{\sigma} t : \mathbb{N} \quad \Gamma \vdash^{\sigma} t_0 : U \quad \Gamma, x : \mathbb{N}, y : U \vdash^{\sigma} t_S : U}{\Gamma \vdash^{\sigma} \mathtt{rec}_{xy}^{t}[t_0 \mid t_S] : U}\ (\mathtt{rec}) \qquad \frac{\Gamma \vdash^{\sigma} p : \exists x^{T}.A \quad p\ \textsc{nef}}{\Gamma \vdash^{\sigma} \mathtt{wit}\ p : T}\ (\mathtt{wit})$$

Figure 8.3: Type system for dLPA$^{\omega}$

8.1.4 Subject reduction

It only remains to prove that typing is preserved along reduction. As for the $\overline{\lambda}_{[lv\tau\star]}$-calculus, the proof is simplified by the fact that substitutions are not performed (except for terms), which keeps us from proving the safety of the corresponding substitutions. Yet, we first need to prove some technical lemmas about dependencies. As in the previous chapter, we define a relation $\sigma \Rrightarrow \sigma'$ between lists of dependencies, which expresses the fact that any typing derivation obtained with σ could be obtained as well as with σ':

$$\sigma \Rrightarrow \sigma' \triangleq \sigma(A) = \sigma(B) \Rightarrow \sigma'(A) = \sigma'(B) \qquad \text{(for any } A, B\text{)}$$

We first show that the cases which we encounter in the proof of subject reduction satisfy this relation:

Lemma 8.1 (Dependencies implication). *The following holds for any $\sigma, \sigma', \sigma''$:*

1. $\sigma\sigma'' \Rrightarrow \sigma\sigma'\sigma'$
2. $\sigma\{(a_1,a_2)|(V_1,V_2)\} \Rrightarrow \sigma\{a_1|V_1\}\{a_2|V_2\}$
3. $\sigma\{\iota_i(a)|\iota_i(V)\} \Rrightarrow \sigma\{a|V\}$
4. $\sigma\{(x,a)|(t,V)\} \Rrightarrow \sigma\{a|V\}\{x|t\}$
5. $\sigma\{\cdot|(p_1,p_2)\} \Rrightarrow \sigma\{a_1|p_1\}\{a_2|p_2\}\{\cdot|(a_1,a_2)\}$
6. $\sigma\{\cdot|\iota_i(p)\} \Rrightarrow \sigma\{a|p\}\{\cdot|\iota_i(a)\}$
7. $\sigma\{\cdot|(t,p)\} \Rrightarrow \sigma\{a|p\}\{\cdot|(t,a)\}$

where the fourth item abuse the definition of list of dependencies to include a substitution of terms.

Proof. All the properties are trivial from the definition of the substitution $\sigma(A)$. □

Proposition 8.2 (Dependencies weakening). *If σ, σ' are two dependencies list such that $\sigma \Rrightarrow \sigma'$, then any derivation using σ can be one using σ' instead. In other words, the following rules are admissible:*

$$\frac{\Gamma \vdash^{\sigma} J}{\Gamma \vdash^{\sigma'} J}\,(w) \qquad\qquad \frac{\Gamma \vdash_d J;\sigma}{\Gamma \vdash_d J;\sigma'}\,(w_d)$$

Proof. Simple induction on the typing derivations. The rules $(\hat{\mathrm{tp}})$ and (Cut) where the list of dependencies is used exactly match the definition of $\Rrightarrow$. Every other case is direct using the first item of Lemma 8.1. □

We also need a simple lemma about stores to simplify the proof of subject reduction:

Lemma 8.3. *The following rule is admissible:*

$$\frac{\Gamma \vdash^{\sigma} \tau_0 : (\Gamma_0;\sigma_0) \quad \Gamma,\Gamma_0 \vdash^{\sigma\sigma_0} \tau_1 : (\Gamma_1;\sigma_1)}{\Gamma \vdash^{\sigma} \tau_0\tau_1 : (\Gamma_0,\Gamma_1;\sigma_0,\sigma_1)}\,(\tau\tau')$$

Proof. By induction on the structure of τ_1. □

Lemma 8.4 (Safe term substitution). *If $\Gamma \vdash^{\sigma} t : T$ then for any conclusion J for typing proofs, contexts, terms, etc; the following holds:*

1. *If* $\Gamma, x : T, \Gamma' \vdash^{\sigma} J$ *then* $\Gamma, \Gamma'[t/x] \vdash^{\sigma[t/x]} J[t/x]$.
2. *If* $\Gamma, x : T, \Gamma' \vdash_d J;\sigma$ *then* $\Gamma, \Gamma'[t/x] \vdash_d J[t/x];\sigma[t/x]$.

Proof. By induction on typing rules. □

Theorem 8.5 (Subject reduction). *For any context Γ and any closures $c\tau$ and $c'\tau'$ such that $c\tau \to c'\tau'$, we have:*

1. *If* $\Gamma \vdash c\tau$ *then* $\Gamma \vdash c'\tau'$.
2. *If* $\Gamma \vdash_d c\tau;\varepsilon$ *then* $\Gamma \vdash_d c'\tau';\varepsilon$.

Proof. The proof follows the usual proof of subject reduction, by induction on the typing derivation and the reduction $c\tau \to c'\tau'$. Since there is no substitution but for terms (proof terms and contexts being stored), there is no need for auxiliary lemmas about the safety of substitution. We sketch it by examining all the rules from Figure 8.3 from top to bottom.

- The cases for reductions of λ are identical to the cases proven in the previous chapter for $\mathrm{dL}_{\hat{\mathfrak{tp}}}$.
- The rules for reducing μ and $\tilde\mu$ are almost the same except that elements are stored, which makes it even easier. For instance in the case of $\tilde\mu$, the reduction rule is:

$$\langle V \| \tilde\mu a.c\tau_1\rangle\tau_0 \to c\tau_0[a := V]\tau_1$$

A typing derivation in regular mode for the command on the left-hand side is of the shape:

$$\dfrac{\dfrac{\dfrac{\Pi_V}{\Gamma,\Gamma_0 \vdash^{\sigma\sigma_0} V : A} \qquad \dfrac{\dfrac{\dfrac{\Pi_c}{\Gamma,\Gamma_0,a:A,\Gamma_1 \vdash^{\sigma\sigma_0\sigma_1} c} \quad \dfrac{\Pi_{\tau_1}}{\Gamma,\Gamma_0,a:A \vdash^{\sigma\sigma_0} \tau_1 : (\Gamma_1;\sigma_1)}}{\Gamma,\Gamma_0,a:A \vdash^{\sigma\sigma_0} c\tau_1}\,(l)}{\Gamma,\Gamma_0 \vdash^{\sigma\sigma_0} \tilde\mu a.c\tau_1 : A^{\perp\!\!\perp}}\,(\tilde\mu)}{\Gamma,\Gamma_0 \vdash^{\sigma\sigma_0} \langle V\|\tilde\mu a.c\tau_1\rangle}\,(\textsc{Cut}) \qquad \dfrac{\Pi_{\tau_0}}{\Gamma \vdash^{\sigma} \tau_0 : (\Gamma_0;\sigma_0)}}{\Gamma \vdash^{\sigma} \langle V\|\tilde\mu a.c\tau_1\rangle\tau_0}\,(l)$$

Thus we can type the command on the right-hand side:

$$\dfrac{\dfrac{\dfrac{\Pi_c}{\Gamma,\Gamma_0,a:A,\Gamma_1 \vdash^{\sigma\sigma_0\{a|V\}\sigma_1} c}}{\Gamma,\Gamma_0,a:A,\Gamma_1 \vdash^{\sigma\sigma_0\{a|V\}\sigma_1} c}\,(w) \qquad \dfrac{\dfrac{\dfrac{\Pi_{\tau_0}}{\Gamma \vdash^{\sigma} \tau_0 : (\Gamma_0;\sigma_0)} \quad \dfrac{\Pi_V}{\Gamma,\Gamma_0 \vdash^{\sigma\sigma_0} V : A}}{\Gamma \vdash^{\sigma} \tau_0[a := V] : (\Gamma_0, a:A; \sigma_0,\{a|V\})}\,(\tau_p) \quad \dfrac{\Pi_{\tau_1}}{\Gamma,\Gamma_0,a:A \vdash^{\sigma\sigma_0} \tau_1 : (\Gamma_1;\sigma_1)}}{\Gamma \vdash^{\sigma} \tau_0[a := V]\tau_1 : (\Gamma_0,a:A,\Gamma_1;\sigma_0\{a|V\}\sigma_1)}\,(\tau\tau')}{\Gamma \vdash^{\sigma} c\tau_0[a := V]\tau_1}\,(l)$$

As for the dependent mode, the binding $\{a|p\}$ within the list of dependencies is compensated when typing the store as shown in the last derivation.

- Similarly, elimination rules for contexts $\tilde\mu[a_1.c_1|a_2.c_2]$, $\tilde\mu(a_1,a_2).c$, $\tilde\mu(x,a).c$ or $\tilde\mu{=}.c$ are easy to check, using Lemma 8.1 and the rule (τ_p) in dependent mode to prove the safety with respect to dependencies.

- The cases for delimited continuations are identical to the corresponding cases for $\mathrm{dL}_{\hat{\mathfrak{tp}}}$.

- The cases for the so-called "call-by-value" rules opening constructors are straightforward, using again Lemma 8.1 in dependent mode to prove the consistency with respect to the list of dependencies.

- The cases for the lazy rules are trivial.

- The first case in the "lookup" section is trivial. The three lefts correspond to the usual unfolding of inductive and co-inductive fixpoints. We only sketch the latter in regular mode. The reduction rule is:

$$\langle a\|f\rangle\tau_0[a := \mathtt{cofix}^t_{bx}[p]]\tau_1 \;\to\; \langle p[t/x][b'/b]\|\tilde\mu a.\langle a\|f\rangle\tau_1\rangle\tau_0[b' := \lambda y.\mathtt{cofix}^y_{bx}[p]]$$

The crucial part of the derivation for the left-hand side command is the derivation for the cofix in the store:

$$\dfrac{\dfrac{\Pi_{\tau_0}}{\Gamma \vdash^{\sigma} \tau_0 : (\Gamma_0;\sigma_0)} \qquad \dfrac{\dfrac{\Pi_t}{\Gamma \vdash^{\sigma\sigma_0} t : T} \quad \dfrac{\Pi_p}{\Gamma,\Gamma_0,f : T \to \mathbb{N}, x : T, b : \forall y^T.f(y) = 0 \vdash^{\sigma\sigma_0} p : A}}{\Gamma,\Gamma_0 \vdash^{\sigma\sigma_0} \mathtt{cofix}^t_{bx}[p] : \nu^t_{fx}A}\,(\mathtt{cofix})}{\Gamma \vdash^{\sigma} \tau_0[a := \mathtt{cofix}^t_{bx}[p]] : (\Gamma_0, a : \nu^t_{fx}A;\sigma_0)}\,(\tau_p)$$

Then, using this derivation, we can type the store of the right-hand side command:

$$\dfrac{\dfrac{\Pi_{\tau_0}}{\Gamma \vdash^{\sigma} \tau_0 : (\Gamma_0;\sigma_0)} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma,\Gamma_0,y:T \vdash^{\sigma\sigma_0} y:T} \quad \dfrac{\Pi_p}{\Gamma,\Gamma_0,f:T\to\mathbb{N},x:T,b:\forall y^T.f(y)=0 \vdash^{\sigma\sigma_0} p:A}}{\Gamma,\Gamma_0,y:T\vdash^{\sigma\sigma_0}\texttt{cofix}^y_{bx}[p]:\nu^y_{fx}A}\,(\texttt{cofix})}{\Gamma,\Gamma_0 \vdash^{\sigma\sigma_0}\lambda y.\texttt{cofix}^y_{bx}[p]:\forall y.\nu^t_{fx}A}\,(\forall_r)}{\Gamma\vdash^\sigma \tau_0[b':=\lambda y.\texttt{cofix}^y_{bx}[p]]:\Gamma_0,b':-\forall y.\nu^y_{fx}A}\,(\tau_p)$$

It only remains to type (we avoid the rest of the derivation, which is less interesting) the proof $p[t/x]$ with this new store to ensure us that the reduction is safe (since the variable a will still be of type $\nu^t_{fx}A$ when typing the rest of the command):

$$\dfrac{\dfrac{\Pi_p}{\Gamma,\Gamma_0,b:\forall y.\nu^y_{fx}A\vdash^\sigma p[t/x]:A[t/x][\nu^y_{fx}A/f(y)=0]} \quad \nu^t_{fx}A\equiv A[t/x][\nu^y_{fx}A/f(y)=0]}{\Gamma,\Gamma_0,b:\forall y.\nu^y_{fx}A\vdash^\sigma p[t/x]:\nu^t_{fx}A}\,(\equiv_r)$$

- The cases for reductions of terms are easy. Since terms are reduced in place within proofs, the only things to check is that the reduction of wit preserves types (which is trivial) and that the β-reduction verifies the subject reduction (which is a well-known fact).

□

8.1.5 Natural deduction as macros

We can recover the usual proof terms for elimination rules in natural deduction systems, and in particular the ones from dPA$^\omega$, by defining them as macros in our language. The definitions are straightforward, using delimited continuations for `let`... `in` and the constructors over NEF proofs which might be dependently typed:

$$\begin{array}{rcl|rcl}
\texttt{let}\,a=p\,\texttt{in}\,q & \triangleq & \mu\alpha_p.\langle p\|\tilde\mu a.\langle q\|\alpha_p\rangle\rangle & \texttt{subst}\;p\,q & \triangleq & \mu\alpha.\langle p\|\tilde\mu{=}.\langle q\|\alpha\rangle\rangle \\
\texttt{split}\;p\;\texttt{as}\,(a_1,a_2)\;\texttt{in}\;q & \triangleq & \mu\alpha_p.\langle p\|\tilde\mu(a_1,a_2).\langle q\|\alpha_p\rangle\rangle & \texttt{exfalso}\;p & \triangleq & \mu\alpha.\langle p\|[]\rangle \\
\texttt{case}\;p\;\texttt{of}\,[a_1.p_1\mid a_2.p_2] & \triangleq & \mu\alpha_p.\langle p\|\tilde\mu[a_1.\langle p_1\|\alpha_p\rangle|a_2.\langle p_2\|\alpha_p\rangle]\rangle & \texttt{catch}_\alpha\,p & \triangleq & \mu\alpha.\langle p\|\alpha\rangle \\
\texttt{dest}\;p\;\texttt{as}\,(a,x)\;\texttt{in}\;q & \triangleq & \mu\alpha_p.\langle p\|\tilde\mu(x,a).\langle q\|\alpha_p\rangle\rangle & \texttt{throw}\;\alpha p & \triangleq & \mu_{\_}.\langle p\|\alpha\rangle \\
\texttt{prf}\;p & \triangleq & \mu\hat{\mathfrak{tp}}.\langle p\|\tilde\mu(x,a).\langle a\|\hat{\mathfrak{tp}}\rangle\rangle & & &
\end{array}$$

where $\alpha_p = \hat{\mathfrak{tp}}$ if p is NEF and $\alpha_p = \alpha$ otherwise.

Proposition 8.6 (Natural deduction)**.** *The typing rules from dPA$^\omega$, given in Section 8.1.5), are admissible*

Proof. Straightforward derivations, the cases for `prf` $p\ q$ and `subst` $p\,q$ are given in Section 7.5.4. □

One can even check that the reduction rules in dLPA$^\omega$ for these proofs almost mimic the ones of dPA$^\omega$. To be more precise, the rules of dLPA$^\omega$ do not allow to simulate each rule of dPA$^\omega$, due to the head-reduction strategy, amongst other things. Nonetheless, up to a few details the reduction of a command in dLPA$^\omega$ follows one particular reduction path of the corresponding proof in dPA$^\omega$, or in other word, one reduction strategy.

The main result is that using the macros, the same proof terms are suitable for countable and dependent choice [70]. We do not state it here, but following the approach of [70], we could also extend dLPA$^\omega$ to obtain a proof for the axiom of bar induction.

$$\frac{\Gamma \vdash p : \exists x^T.A \quad \Gamma, x : T, a : A \vdash q : B[(x,a)/\bullet] \quad p \notin \textsc{nef} \Rightarrow \bullet \notin B}{\Gamma \vdash \texttt{dest}\ p\ \texttt{as}\,(x,a)\ \texttt{in}\ q : B[p/\bullet]}\ (\texttt{dest}) \qquad \frac{\Gamma \vdash p : \exists x^T.A(x)}{\Gamma \vdash \texttt{prf}\ p : A(\texttt{wit}\ p)}\ (\texttt{prf})$$

$$\frac{\Gamma \vdash p : A_1 \wedge A_2 \quad \Gamma, a_1 : A_1, a_2 : A_2 \vdash q : B[(a_1,a_2)/\bullet] \quad p \notin \textsc{nef} \Rightarrow \bullet \notin B}{\Gamma \vdash \texttt{split}\ p\ \texttt{as}\,(a_1,a_2)\ \texttt{in}\ q : B[p/\bullet]}\ (\texttt{split}) \qquad \frac{\Gamma \vdash p : A_1 \wedge A_2}{\Gamma \vdash \pi_i(p) : A_i}\ (\wedge_E^i)$$

$$\frac{\Gamma \vdash p : A_1 \vee A_2 \quad \Gamma, a_i : A_i \vdash q : B[\iota_i(a)_i/\bullet] \quad \text{for } i = 1,2 \quad p \notin \textsc{nef} \Rightarrow \bullet \notin B}{\Gamma \vdash \texttt{case}\ p\ \texttt{of}\ [a_1.p_1 \mid a_2.p_2] : B[p/\bullet]}\ (\texttt{case}) \qquad \frac{\Gamma \vdash p : \bot}{\Gamma \vdash \texttt{exfalso}\ p : B}\ (\bot)$$

$$\frac{\Gamma, a : A \vdash q : B[a/\bullet] \quad p \notin \textsc{nef} \Rightarrow \bullet \notin B}{\Gamma \vdash \texttt{let}\, a = p\, \texttt{in}\, q : B[p/\bullet]}\ (\texttt{let}) \qquad \frac{\Gamma, \alpha : A^{\perp\!\!\perp} \vdash p : A}{\Gamma \vdash \texttt{catch}_\alpha\, p : A} \qquad \frac{\Gamma, \alpha : A^{\perp\!\!\perp} \vdash p : A}{\Gamma, \alpha : A^{\perp\!\!\perp} \vdash \texttt{throw}\ \alpha\, p : B}$$

Figure 8.4: Typing rules of dPA$^\omega$

Theorem 8.7 (Countable choice [70]). *We have:*

$$\begin{array}{rcl} AC_{\mathbb{N}} & := & \lambda H.\texttt{let}\, a = \texttt{cofix}^0_{bn}[(Hn, b(S(n))]\ \texttt{in}\,(\lambda n.\,\texttt{wit}\,(\texttt{nth}_n\, a), \lambda n.\,\texttt{prf}\,(\texttt{nth}_n\, a) \\ & : & \forall x^{\mathbb{N}} \exists y^T P(x,y) \to \exists f^{\mathbb{N}\to T} \forall x^{\mathbb{N}} P(x, f(x)) \end{array}$$

where $\texttt{nth}_n\, a := \pi_1(\texttt{ind}^n_{x,c}[a \mid \pi_2(c)])$.

Proof. See Figure 8.5. □

Theorem 8.8 (Dependent choice [70]). *We have:*

$$\begin{array}{rcl} DC & := & \lambda H.\lambda x_0.\,\texttt{let}\ a = (x_0, \texttt{cofix}^0_{bn}[\texttt{dest}\ Hn\ \texttt{as}\,(y,c)\ \texttt{in}\,(y,(c, b\, y)))] \\ & & \texttt{in}\ (\lambda n.\,\texttt{wit}\,(\texttt{nth}_n\, a), (\texttt{refl}, \lambda n.\pi_1(\texttt{prf}\,(\texttt{prf}\,(\texttt{nth}_n\, a))))) \\ & : & \forall x^T.\exists y^T.P(x,y) \to \forall x_0^T.\exists f \in T^{\mathbb{N}}.(f(0) = x_0 \wedge \forall n^{\mathbb{N}}.P(f(n), f(s(n)))) \end{array}$$

where $\texttt{nth}_n\, a := \texttt{ind}^n_{x,d}[a \mid (\texttt{wit}\,(\texttt{prf}\ d), \pi_2(\texttt{prf}\,(\texttt{prf}\,(d))))]$.

Proof. Left to the reader. □

8.2 Small-step calculus

Once more, we follow Danvy's methodology of semantic artifacts to obtain a continuation-passing style translation and a realizability interpretation. We first decompose the reduction system of dLPA$^\omega$ into small-step reduction rules, that we denote by $\rightsquigarrow_s$. This requires a refinement and an extension of the syntax, that we shall now present. To keep us from boring the reader stiff with new (huge) tables for the syntax, typing rules and so forth, we will introduce them step by step. We hope it will help the reader to convince herself of the necessity and of the somewhat naturality of these extensions.

First of all, we need to refine the syntax to distinguish between strong and weak values in the syntax of proof terms. As in the $\overline{\lambda}_{[l v \tau \star]}$-calculus, this refinement is induced by the computational behavior of the calculus: weak values are the ones which are stored by $\tilde{\mu}$ binders, but which are not values enough to be eliminated in front of a forcing context, that is to say variables. Indeed, if we observe the reduction system, we see that in front of a forcing context f, a variable leads a search through the store for a "stronger" value, which could incidentally provoke the evaluation of some fixpoints. On the other

Notations:

- $\mathtt{nth}_t\, p \triangleq \pi_1(\mathtt{ind}^t_{sx}[p \mid \pi_2(s)])$
- $A^n_\infty \triangleq \nu^n_{fx}[A(x) \wedge f(S(x)) = 0]$
- $\mathtt{str}^t_\infty\, H \triangleq \mathtt{cofix}^t_{bn}[(Hn, b(S(n)))]$
- $A(x) \triangleq \exists y^T.P(x,y)$

Typing derivation for nth ($\Pi_{\mathtt{nth}}$):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{n:\mathbb{N}\vdash n:\mathbb{N}}\,(\mathrm{Ax}_n) \quad \dfrac{}{a:A^0_\infty \vdash a:A^0_\infty}\,(\mathrm{Ax}_r) \quad \dfrac{\dfrac{\dfrac{}{s:A^m_\infty \vdash s:A^{S(m)}_\infty}\,(\mathrm{Ax}_r) \quad A^m_\infty \equiv A(m)\wedge A^{S(m)}_\infty}{m:\mathbb{N}, s:A^m_\infty \vdash s: A(m)\wedge A^{S(m)}_\infty}\,(\equiv_r)}{m:\mathbb{N}, s:A^m_\infty \vdash \pi_2(s):A^{S(m)}_\infty}\,(\wedge^2_E)}{a:A^0_\infty, n:\mathbb{N}\vdash \mathtt{ind}^t_{sx}[a \mid \pi_2(s)]:A^n_\infty}\,(\mathtt{ind}) \quad A^n_\infty \equiv A(n)\wedge A^{S(n)}_\infty}{a:A^0_\infty, n:\mathbb{N}\vdash \mathtt{ind}^t_{sx}[a \mid \pi_2(s)]:A(n)\wedge A^{S(n)}_\infty}\,(\equiv_r)}{a:A^0_\infty, n:\mathbb{N}\vdash \pi_1(\mathtt{ind}^t_{sx}[a \mid \pi_2(s)]):A(n)}\,(\wedge^1_E)}{a:A^0_\infty, n:\mathbb{N}\vdash \mathtt{nth}_n\, a:A(n)}\,(\mathtt{def})$$

Typing derivation for $\mathbf{str}^0_\infty$ ($\Pi_{\mathtt{str}_\infty}$):

$$\dfrac{\dfrac{\dfrac{}{\vdash 0:\mathbb{N}} \quad \dfrac{\dfrac{\dfrac{}{H:\forall x^{\mathbb{N}}\exists y^T P(x,y)\vdash H:\forall x^{\mathbb{N}}\exists y^T P(x,y)}\,(\mathrm{Ax}_r) \quad \dfrac{}{n:\mathbb{N}\vdash n:\mathbb{N}}\,(\mathrm{Ax}_r)}{H:\forall x^{\mathbb{N}}\exists y^T P(x,y), n:\mathbb{N}\vdash Hn:\exists y^T.P(n,y)}\,(\forall_r)}{H:\forall x^{\mathbb{N}}\exists y^T P(x,y), n:\mathbb{N}, b:\forall z^N.f(z)=0 \vdash (Hn, b(S(n))):\exists y^T.P(n,y)\wedge f(S(n))=0}}{H:\forall x^{\mathbb{N}}\exists y^T P(x,y)\vdash \mathtt{cofix}^0_{bn}[(Hn,b(S(n)))]:\nu^0_{fx}\exists y^T.P(x,y)\wedge f(S(x))=0}}{H:\forall x^{\mathbb{N}}\exists y^T P(x,y)\vdash \mathtt{str}^0_\infty\, H:A^0_\infty}\,(\mathtt{def})$$

Typing derivation for $AC_{\mathbb{N}}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_{\mathtt{nth}}}{a:A^0_\infty, n:\mathbb{N}\vdash \mathtt{nth}_n\, a:A(n)}}{a:A^0_\infty, n:\mathbb{N}\vdash \mathtt{nth}_n\, a:\exists y^T.P(n,y)}\,(\mathtt{def})}{a:A^0_\infty, n:\mathbb{N}\vdash \mathtt{wit}\,(\mathtt{nth}_n\, a):T}\,(\mathtt{wit})}{a:A^0_\infty\vdash \lambda n.\,\mathtt{wit}\,(\mathtt{nth}_n\, a):\mathbb{N}\to T} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_{\mathtt{nth}}}{a:A^0_\infty, n:\mathbb{N}\vdash \mathtt{nth}_n\, a:A(n)}}{a:A^0_\infty, n:\mathbb{N}\vdash \mathtt{nth}_n\, a:\exists y^T.P(n,y)}\,(\mathtt{def})}{a:A^0_\infty, x:\mathbb{N}\vdash \mathtt{prf}\,(\mathtt{nth}_n\, a):P(x,\mathtt{wit}\,(\mathtt{nth}_x\, a))}\,(\equiv_r)}{a:A^0_\infty, x:\mathbb{N}\vdash \mathtt{prf}\,(\mathtt{nth}_n\, a):P(x,\lambda n.\,\mathtt{wit}\,(\mathtt{nth}_n\, a)x)}\,(\equiv_r)}{a:A^0_\infty\vdash \lambda n.\,\mathtt{prf}\,(\mathtt{nth}_n\, a):\forall x^{\mathbb{N}}.P(x,(\lambda n.\,\mathtt{wit}\,(\mathtt{nth}_n\, a))x)}\,(\forall_r)}{a:A^0_\infty\vdash (\lambda n.\,\mathtt{wit}\,(\mathtt{nth}_n\, a), \lambda n.\,\mathtt{prf}\,(\mathtt{nth}_n\, a)):\exists f^{\mathbb{N}\to T}.\forall x^{\mathbb{N}}.P(x,f(x))}\,(\exists_r)}{H:\forall x^{\mathbb{N}}\exists y^T P(x,y)\vdash \mathtt{let}\, a = \mathtt{str}^0_\infty\, H\,\mathtt{in}\,(\lambda n.\,\mathtt{wit}\,(\mathtt{nth}_n\, a), \lambda n.\,\mathtt{prf}\,(\mathtt{nth}_n\, a)):\exists f^{\mathbb{N}\to T}.\forall x^{\mathbb{N}}.P(x,f(x))}\,(\mathtt{let})}{\vdash \lambda H.\,\mathtt{let}\, a = \mathtt{str}^0_\infty\, H\,\mathtt{in}\,(\lambda n.\,\mathtt{wit}\,(\mathtt{nth}_n\, a), \lambda n.\,\mathtt{prf}\,(\mathtt{nth}_n\, a)):\forall x^{\mathbb{N}}.\exists y^T.P(x,y)\to \exists f^{\mathbb{N}\to T}.\forall x^{\mathbb{N}}.P(x,f(x))}\,(\to_r)$$

where we omit the conversion $P(x,(\lambda n.\,\mathtt{wit}\,(\mathtt{nth}_n\, a))x) \equiv P(x,\mathtt{wit}\,(\mathtt{nth}_x\, a))$ on the right-hand side derivation.

Figure 8.5: Proof of the axiom of countable choice in dLPA$^\omega$

hand, strong values are the ones which can be reduced in front of the matching forcing context, that is to say functions, `refl`, pairs of (weak) values, injections or dependent pairs:

$$\begin{array}{llcl}
\textbf{Weak values} & V & ::= & a \mid v \\
\textbf{Strong values} & v & ::= & \iota_i(V) \mid (V,V) \mid (V_t,V) \mid \lambda x.p \mid \lambda a.p \mid \texttt{refl}
\end{array}$$

This allows to distinguish commands of the shape $\langle v \| f\rangle\tau$, where the forcing context (and next the strong value) are examined to determine whether the command reduces or not; from commands of the shape $\langle a \| f\rangle\tau$ where the focus is put on the variable a, which leads to a lookup for the associated proof in the store.

Next, we need to explicit the reduction of terms. To this purpose, we include a machinery to evaluate terms in a way which resemble the evaluation of proofs. In particular, we define new commands which we write $\langle t \| \pi\rangle$ where t is a term and π is a context for terms (or co-term). Co-terms are either of the shape $\tilde\mu x.c$ or stacks ot the shape $u\cdot\pi$. These constructions are the usual ones of the $\lambda\mu\tilde\mu$-calculus (which are also the ones for proofs). We also extend the definitions of commands with delimited continuations to include the corresponding commands for terms:

$$\begin{array}{llcl|llcl}
\textbf{Commands} & c & ::= & \langle p \| e\rangle \mid \langle t \| \pi\rangle & \textbf{Delimited} & c_{\hat{\mathfrak{tp}}} & ::= & \cdots \mid \langle t \| \pi_{\hat{\mathfrak{tp}}}\rangle \\
\textbf{Co-terms} & \pi & ::= & t\cdot\pi \mid \tilde\mu x.c & \textbf{continuations} & \pi_{\hat{\mathfrak{tp}}} & ::= & t\cdot\pi_{\hat{\mathfrak{tp}}} \mid \tilde\mu x.c_{\hat{\mathfrak{tp}}}
\end{array}$$

We give typing rules for these new constructions, which are the usual rules for typing contexts in the $\lambda\mu\tilde\mu$-calculus:

$$\frac{\Gamma\vdash t:T \quad \Gamma\vdash\pi:U^{\perp\!\!\!\perp}}{\Gamma\vdash t\cdot\pi:(T\to U)^{\perp\!\!\!\perp}}\,(\to_l) \qquad \frac{c:(\Gamma,x:T)}{\Gamma\vdash\tilde\mu x.c:T^{\perp\!\!\!\perp}}\,(\tilde\mu_x) \qquad \frac{\Gamma\vdash^\sigma t:T \quad \Gamma\vdash^\sigma\pi:T^{\perp\!\!\!\perp}}{\Gamma\vdash^\sigma\langle t\|\pi\rangle}\,(\textsc{cut}_t)$$

It is worth noting that the syntax as well as the typing and reduction rules for terms now match exactly the ones for proofs[6]. In other words, with these definitions, we could abandon the stratified presentation without any trouble, since reduction rules for terms will naturally collapse to the ones for proofs.

Finally, in order to maintain typability when reducing dependent pairs of the strong existential type, we need to add what we call *co-delimited continuations*. We saw in the previous chapter that the CPS translation of pairs (t,p) was not the expected one, and we mentioned the fact that it reflected the need for a special reduction rule. Indeed, consider such a pair of type $\exists x^T.A$, the standard way of reducing it would be a rule like:

$$\langle (t,p) \| e\rangle\tau \rightsquigarrow_s \langle t \| \tilde\mu x.\langle p \| \tilde\mu a.\langle (x,a) \| e\rangle\rangle\rangle\tau$$

but such a rule does not satisfy subject reduction. Indeed, consider a typing derivation for the left-hand side command, when typing the pair (t,p), p is of type $A[t]$. On the command on the right-hand side, the variable a will then also be of type $A[t]$, while it should be of type $A[x]$ for the pair (x,a) to be typed. We thus need to compensate this mismatching of types, by reducing t within a context where a is not linked to p but to a co-reset $\check{\mathfrak{tp}}$ (dually to reset $\hat{\mathfrak{tp}}$), whose type can be changed from $A[x]$ to $A[t]$ thanks to a list of dependencies:

$$\langle (t,p) \| e\rangle_p\tau \rightsquigarrow_s \langle p \| \tilde\mu\check{\mathfrak{tp}}.\langle t \| \tilde\mu x.\langle \check{\mathfrak{tp}} \| \tilde\mu a.\langle (x,a) \| e\rangle\rangle\rangle\rangle_p\tau$$

We thus equip the language with new contexts $\tilde\mu\check{\mathfrak{tp}}.c_{\check{\mathfrak{tp}}}$, which we call co-shifts, and where $c_{\check{\mathfrak{tp}}}$ is a command whose last cut is of the shape $\langle\check{\mathfrak{tp}} \| e\rangle$. This corresponds formally to the following syntactic

[6] Except for substitutions of terms, which we could store as well

sets, which are dual to the ones introduced for delimited continuations:

$$\begin{array}{llcl}
\textbf{Contexts} & e & ::= & \cdots \mid \tilde\mu\check{\mathtt{tp}}.c_{\check{\mathtt{tp}}} \\[1ex]
\textbf{Co-delimited} & c_{\check{\mathtt{tp}}} & ::= & \langle p_N \| e_{\check{\mathtt{tp}}}\rangle \mid \langle t \| \pi_{\check{\mathtt{tp}}}\rangle \mid \langle \check{\mathtt{tp}} \| e\rangle \\
\textbf{continuations} & e_{\check{\mathtt{tp}}} & ::= & \tilde\mu a.c_{\check{\mathtt{tp}}} \mid \tilde\mu[a_1.c_{\check{\mathtt{tp}}} \mid a_2.c'_{\check{\mathtt{tp}}}] \mid \tilde\mu(a_1,a_2).c_{\check{\mathtt{tp}}} \mid \tilde\mu(x,a).c_{\check{\mathtt{tp}}} \\
 & \pi_{\check{\mathtt{tp}}} & ::= & t\cdot\pi_{\check{\mathtt{tp}}} \mid \tilde\mu x.c_{\check{\mathtt{tp}}} \\[1ex]
\textbf{NEF} & e_N & ::= & \cdots \mid \tilde\mu\check{\mathtt{tp}}.c_{\check{\mathtt{tp}}}
\end{array}$$

This might seem to be a heavy addition to the language, but we insist on the fact that these artifacts are merely the dual constructions of delimited continuations that we introduced in $\mathrm{dL}_{\hat{\mathtt{tp}}}$, with a very similar intuition. In particular, it might be helpful for the reader to think of the fact that we introduced delimited continuations for type safety of the evaluation of dependent products in $\Pi a : A.B$ (which naturally extends to the case $\forall x^T.A$). Therefore, to maintain type safety of dependent sums in $\exists x^T.A$, we need to introduce the dual constructions of co-delimited continuations. We also give typing rules to these constructions, which are dual to the typing rules for delimited-continuations:

$$\frac{\Gamma,\check{\mathtt{tp}} : A \vdash_d c_{\check{\mathtt{tp}}};\sigma}{\Gamma \vdash^\sigma \tilde\mu\check{\mathtt{tp}}.c_{\check{\mathtt{tp}}} : A^{\perp\!\!\perp}}\,(\tilde\mu\check{\mathtt{tp}}) \qquad\qquad \frac{\Gamma,\Gamma' \vdash^\sigma e : A^{\perp\!\!\perp} \quad \sigma(A) = \sigma(B)}{\Gamma,\check{\mathtt{tp}} : B,\Gamma' \vdash_d \langle\check{\mathtt{tp}}\|e\rangle;\sigma}\,(\check{\mathtt{tp}})$$

Note that we also need to extend the definition of list of dependencies so as to include bindings of the shape $\{x|t\}$ for terms, and that we have to give the corresponding typing rules to type commands of terms in dependent mode:

$$\frac{c : (\Gamma, x : T;\sigma\{x|t\})}{\Gamma \vdash_d \tilde\mu x.c : T^{\perp\!\!\perp};\sigma\{\cdot|t\}}\,(\tilde\mu) \qquad\qquad \frac{\Gamma,\Gamma' \vdash^\sigma t : T \quad \Gamma,\check{\mathtt{tp}} : B,\Gamma' \vdash_d \pi : A^{\perp\!\!\perp};\sigma\{\cdot|t\}}{\Gamma,\check{\mathtt{tp}} : B,\Gamma' \vdash_d \langle t\|\pi\rangle;\sigma}\,(\textsc{Cut})$$

The small-step reduction system is given in Figure 8.6. The rules are written $c_\iota\tau \rightsquigarrow_s c'_o\tau'$ where the annotation ι,p on commands are indices (*i.e.* c,p,e,V,f,t,π,V_t) indicating which part of the command is in control. As in the $\overline{\lambda}_{[lv\tau\star]}$-calculus, we observe an alternation of steps descending from p to f for proofs and from t to V_t for terms. The descent for proofs can be divided in two main phases. During the first phase, from p to e we observe the call-by-value process, which extracts values from proofs, opening recursively the constructors and computing values. In the second phase, the core computation takes place from V to f, with the destruction of constructors and the application of function to their arguments. The laziness corresponds precisely to a skip of the first phase, waiting to possibly reach the second phase before actually going through the first one.

We briefly state the important properties of this system.

Proposition 8.9 (Subject reduction). *The small-step reduction rules satisfy subject reduction.*

Proof. The proof is again a tedious induction on the reduction $\rightsquigarrow_s$. There is almost nothing new in comparison with the cases for the big-step reduction rules: the cases for reduction of terms are straightforward, as well as the administrative reductions changing the focus on a command. We only give the case for the reduction of pairs (t,p). The reduction rule is:

$$\langle (t,p)\|e\rangle_p\tau \rightsquigarrow_s \langle p\|\tilde\mu\check{\mathtt{tp}}.\langle t\|\tilde\mu x.\langle\check{\mathtt{tp}}\|\tilde\mu a.\langle (x,a)\|e\rangle\rangle\rangle\rangle_p\tau$$

Consider a typing derivation for the command on the left-hand side, which is of the shape (we omit the rule (l) and the store for conciseness):

$$\frac{\dfrac{\dfrac{\Pi_t}{\Gamma \vdash^\sigma t : T} \quad \dfrac{\Pi_p}{\Gamma \vdash^\sigma p : A[t/x]}}{\Gamma \vdash^\sigma (t,p) : \exists x^T.A}\,(\exists_r) \qquad \dfrac{\Pi_e}{\Gamma \vdash^\sigma e : (\exists x^T.A)^{\perp\!\!\perp}}}{\Gamma \vdash^\sigma \langle (t,p)\|e\rangle}\,(\textsc{Cut})$$

Commands

$$\begin{array}{rcl}
\langle p\|e\rangle_c\tau & \rightsquigarrow_s & \langle p\|e\rangle_p \\
\langle t\|\pi\rangle_c\tau & \rightsquigarrow_s & \langle t\|\pi\rangle_t
\end{array}$$

Delimited continuations

$$\begin{array}{lrcll}
(\text{for any } \iota,o) & \langle\mu\hat{\mathfrak{tp}}.c\tau''\|e\rangle_p\tau & \rightsquigarrow_s & \langle\mu\hat{\mathfrak{tp}}.c'\tau''\|e\rangle_p\tau' & (\text{if } c_\iota\tau \rightsquigarrow_s c'_o\tau') \\
 & \langle\mu\hat{\mathfrak{tp}}.\langle p\|\hat{\mathfrak{tp}}\rangle\|e\rangle_p\tau & \rightsquigarrow_s & \langle p\|e\rangle_p\tau & \\
(\text{for any } \iota,o) & \langle V\|\tilde{\mu}\check{\mathfrak{tp}}.c\rangle_e\tau & \rightsquigarrow_s & \langle V\|\tilde{\mu}\check{\mathfrak{tp}}.c'\rangle_e\tau' & (\text{if } c_\iota\tau \rightsquigarrow_s c'_o\tau') \\
 & \langle V\|\tilde{\mu}\check{\mathfrak{tp}}.\langle\check{\mathfrak{tp}}\|e\rangle\rangle_e\tau & \rightsquigarrow_s & \langle V\|e\rangle_e\tau &
\end{array}$$

Proofs

$$\begin{array}{lrcl}
(e \neq e_{\hat{\mathfrak{tp}}}) & \langle\mu\alpha.c\|e\rangle_p\tau & \rightsquigarrow_s & c_c\,\tau[\alpha := e] \\
 & \langle\mu\alpha.c\|e_{\hat{\mathfrak{tp}}}\rangle_p\tau & \rightsquigarrow_s & c_c[e_{\hat{\mathfrak{tp}}}/\alpha]\,\tau \\
(a\text{ fresh}) & \langle(p_1,p_2)\|e\rangle_p\tau & \rightsquigarrow_s & \langle p_1\|\tilde{\mu}a_1.\langle p_2\|\tilde{\mu}a_2.\langle(a_1,a_2)\|e\rangle\rangle\rangle_p\tau \\
(a\text{ fresh}) & \langle\iota_i(p)\|e\rangle_p\tau & \rightsquigarrow_s & \langle p\|\tilde{\mu}a.\langle\iota_i(a)\|e\rangle\rangle_p\tau \\
(a\text{ fresh}) & \langle(t,p)\|e\rangle_p\tau & \rightsquigarrow_s & \langle p\|\tilde{\mu}\check{\mathfrak{tp}}.\langle t\|\tilde{\mu}x.\langle\check{\mathfrak{tp}}\|\tilde{\mu}a.\langle(x,a)\|e\rangle\rangle\rangle\rangle_p\tau \\
(y,a\text{ fresh}) & \langle\mathtt{ind}_{bx}^{t}[p\mid q]\|e\rangle_p\tau & \rightsquigarrow_s & \langle\mu\hat{\mathfrak{tp}}.\langle t\|\tilde{\mu}y.\langle a\|\hat{\mathfrak{tp}}\rangle[a := \mathtt{ind}_{bx}^{y}[p\mid q]]\rangle\|e\rangle_p\tau \\
(y,a\text{ fresh}) & \langle\mathtt{cofix}_{bx}^{t}[p]\|e\rangle_p\tau & \rightsquigarrow_s & \langle\mu\hat{\mathfrak{tp}}.\langle t\|\tilde{\mu}y.\langle a\|\hat{\mathfrak{tp}}\rangle\rangle[a := \mathtt{cofix}_{bx}^{y}[p]]\|e\rangle_p\tau \\
 & \langle V\|e\rangle_p\tau & \rightsquigarrow_s & \langle V\|e\rangle_e
\end{array}$$

Contexts

$$\begin{array}{rcl}
\langle V\|\alpha\rangle_e\tau[\alpha := e]\tau' & \rightsquigarrow_s & \langle V\|e\rangle_e\tau[\alpha := e]\tau' \\
\langle V\|\tilde{\mu}a.c\tau'\rangle_e\tau & \rightsquigarrow_s & c_c\tau[a := V]\tau' \\
\langle V\|f\rangle_e\tau & \rightsquigarrow_s & \langle V\|f\rangle_V\tau
\end{array}$$

Values

$$\begin{array}{lrcl}
 & \langle a\|f\rangle_V\tau[a := V]\tau' & \rightsquigarrow_s & \langle V\|f\rangle_V\tau[a := V]\tau' \\
 & \langle v\|f\rangle_V\tau & \rightsquigarrow_s & \langle v\|f\rangle_f\tau \\
(b'\text{ fresh}) & \langle a\|f\rangle_V\tau[a = \mathtt{cofix}_{bx}^{t}[p]]\tau' & \rightsquigarrow_s & \langle p[t/x][b'/b]\|\tilde{\mu}a.\langle a\|f\rangle\tau'\rangle_p\tau[b' := \lambda y.\mathtt{cofix}_{bx}^{y}[p]] \\
 & \langle a\|f\rangle_V\tau[a = \mathtt{ind}_{bx}^{0}[p_0\mid p_S]]\tau' & \rightsquigarrow_s & \langle p_0\|\tilde{\mu}a.\langle a\|f\rangle\tau'\rangle_p\tau \\
(b'\text{ fresh}) & \langle a\|f\rangle_v\tau[a = \mathtt{ind}_{bx}^{S(t)}[p_0\mid p_S]]\tau' & \rightsquigarrow_s & \langle p_S[t/x][b'/b]\|\tilde{\mu}a.\langle a\|f\rangle\tau'\rangle_p\tau[b' := \mathtt{ind}_{bx}^{t}[p_0\mid p_S]]]
\end{array}$$

Forcing contexts

$$\begin{array}{lrcl}
 & \langle\lambda x.p\|t\cdot e\rangle_f\tau & \rightsquigarrow_s & \langle\mu\hat{\mathfrak{tp}}.\langle t\|\tilde{\mu}x.\langle p\|\hat{\mathfrak{tp}}\rangle\rangle\|e\rangle_p\tau \\
(q\in\textsc{nef}) & \langle\lambda a.p\|q\cdot e\rangle_f\tau & \rightsquigarrow_s & \langle\mu\hat{\mathfrak{tp}}.\langle q\|\tilde{\mu}a.\langle p\|\hat{\mathfrak{tp}}\rangle\rangle\|e\rangle_p\tau \\
(q\notin\textsc{nef}) & \langle\lambda a.p\|q\cdot e\rangle_f\tau & \rightsquigarrow_s & \langle q\|\tilde{\mu}a.\langle p\|e\rangle\rangle_p\tau \\
 & \langle\iota_i(V)\|\tilde{\mu}[a_1.c^1\mid a_2.c^2]\rangle_f\tau & \rightsquigarrow_s & c_c^i\tau[a_i := V] \\
 & \langle(V_1,V_2)\|\tilde{\mu}(a_1,a_2).c\rangle_f\tau & \rightsquigarrow_s & c_c\tau[a_1 := V_1][a_2 := V_2] \\
 & \langle(V_t,V)\|\tilde{\mu}(x,a).c\rangle_f\tau & \rightsquigarrow_s & (c[V_t/x])_c\tau[a := V] \\
 & \langle\mathtt{refl}\|\tilde{\mu}{=}.c\rangle_f\tau & \rightsquigarrow_s & c_c\tau
\end{array}$$

Terms

$$\begin{array}{lrcl}
 & \langle tu\|\pi\rangle_t\tau & \rightsquigarrow_s & \langle t\|u\cdot\pi\rangle_t\tau \\
(x\text{ fresh}) & \langle S(t)\|\pi\rangle_t\tau & \rightsquigarrow_s & \langle t\|\tilde{\mu}x.\langle S(x)\|\pi\rangle\rangle_t \\
(x,a\text{ fresh}) & \langle\mathtt{wit}\ p\|\pi\rangle_t\tau & \rightsquigarrow_s & \langle p\|\tilde{\mu}(x,a).\langle x\|\pi\rangle\rangle_p\tau \\
(t\notin V_t) & \langle\mathtt{rec}_{xy}^{t}[t_0\mid t_S]\|\pi\rangle_t\tau & \rightsquigarrow_s & \langle t\|\tilde{\mu}z.\langle\mathtt{rec}_{xy}^{z}[t_0\mid t_S]\|\pi\rangle\rangle_t\tau \\
 & \langle\mathtt{rec}_{xy}^{0}[t_0\mid t_S]\|\pi\rangle_t\tau & \rightsquigarrow_s & \langle t_0\|\pi\rangle_t\tau \\
 & \langle\mathtt{rec}_{xy}^{S(V_t)}[t_0\mid t_S]\|\pi\rangle_t\tau & \rightsquigarrow_s & \langle t_S[V_t/x][\mathtt{rec}_{xy}^{V_t}[t_0\mid t_S]/y]\|\pi\rangle_t\tau \\
 & \langle V_t\|\pi\rangle_t\tau & \rightsquigarrow_s & \langle V_t\|\pi\rangle_\pi\tau \\
 & \langle\lambda x.t\|u\cdot\pi\rangle_\pi\tau & \rightsquigarrow_s & \langle u\|\tilde{\mu}x.\langle t\|\pi\rangle\rangle_t\tau \\
 & \langle V_t\|\tilde{\mu}x.c_t\rangle_\pi\tau & \rightsquigarrow_s & (c_t\tau)[V_t/x] \\
 & \langle V_t\|\tilde{\mu}x.c\rangle_\pi\tau & \rightsquigarrow_s & (c_p\tau)[V_t/x]
\end{array}$$

Figure 8.6: Small-step reduction rules

Then we can type the command on the right-hand side with the following derivation:

$$\dfrac{\dfrac{\Pi_p}{\Gamma \vdash^\sigma p : A[t]} \qquad \dfrac{\dfrac{\dfrac{\Pi_t}{\Gamma, \check{\mathtt{tp}} : A[t] \vdash t : T} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\Pi_{(x,a)} \qquad \Pi_e}{\Gamma, x : T, a : A[x] \vdash^\sigma \langle (x,a) \| e \rangle : A[x]^{\perp\!\!\perp}}\,(\textsc{Cut})}{\Gamma, x : T \vdash^\sigma \tilde{\mu} a.\langle (x,a) \| e \rangle : A[x]^{\perp\!\!\perp}}\,(\tilde{\mu}) \qquad A[t] = (\{x|t\})(A[x])}{\Gamma, \check{\mathtt{tp}} : A[t], x : T \vdash_d \langle \check{\mathtt{tp}} \| \tilde{\mu} a.\langle (x,a) \| e \rangle \rangle ; \sigma\{x|t\}}\,(\textsc{Cut}_d)}{\Gamma, \check{\mathtt{tp}} : A[t/x] \vdash_d \tilde{\mu} x.\langle \check{\mathtt{tp}} \| \tilde{\mu} a.\langle (x,a) \| e \rangle \rangle : T ; \sigma\{\cdot|t\}}\,(\tilde{\mu}_x)}{\Gamma, \check{\mathtt{tp}} : A[t] \vdash \langle t \| \tilde{\mu} x.\langle \check{\mathtt{tp}} \| \tilde{\mu} a.\langle (x,a) \| e \rangle \rangle \rangle ; \sigma}\,(\textsc{Cut}_d)}{\Gamma \vdash^\sigma \tilde{\mu} \check{\mathtt{tp}}.\langle t \| \tilde{\mu} x.\langle \check{\mathtt{tp}} \| \tilde{\mu} a.\langle (x,a) \| e \rangle \rangle \rangle : A[t]^{\perp\!\!\perp}}\,(\tilde{\mu}\check{\mathtt{tp}})}{\Gamma \vdash^\sigma \langle p \| \tilde{\mu} \check{\mathtt{tp}}.\langle t \| \tilde{\mu} x.\langle \check{\mathtt{tp}} \| \tilde{\mu} a.\langle (x,a) \| e \rangle \rangle \rangle \rangle_p}\,(\textsc{Cut})$$

where $\Pi_{(x,a)}$ is as expected. □

It is direct to check that the small-step reduction system simulates the big-step one, and in particular that it preserves the normalization :

Proposition 8.10. *If a closure $c\tau$ normalizes for the reduction $\rightsquigarrow_s$, then it normalizes for $\to$.*

Proof. By contraposition, one proves that if a command $c\tau$ produces an infinite number of steps for the reduction →, then it does not normalize for $\rightsquigarrow_s$ either. This is proved by showing by induction on the reduction → that each step, except for the contextual reduction of terms, is reflected in at least on for the reduction $\rightsquigarrow_s$. The rules for term reductions require a separate treatment, which is really not interesting at this point. We claim that the reduction of terms, which are usual simply-typed λ-terms, is known to be normalizing anyway and does not deserve that we spend another page proving it in this particular setting. □

8.3 A continuation-passing style translation

We present in this section the continuation-passing style translation[7] which arises from the small-step reduction system we defined. In practice, we will not give here a formal proof of normalization for dLPA$^\omega$(we will give one using a realizability interpretation in the next section), so that we will deliberately omit some proofs and details. In particular, we have *a priori* two choices for the target language of this translation.

Either our interest in the translation is only to prove the normalization of dLPA$^\omega$, in which case we can erase the dependencies and use a non-dependently typed target language. Starting from dLPA$^\omega$, embedding terms and proofs in a single syntactic set then removing dependent types would roughly leave us with a first-order language similar to the $\overline{\lambda}_{[lv\tau\star]}$-calculus (but more expressive). A good candidate as a target language for a CPS translation erasing dependencies is hence System F_Υ, possibly enriched[8] with conjunction, disjunction, etc... to recover the same expressiveness as dLPA$^\omega$. In this case, the typability of the translation would be greatly simplified and it would mostly amount to the typability of the CPS translation for the $\overline{\lambda}_{[lv\tau\star]}$-calculus in Chapter 6.

On the other hand, we could be interested in a translation carrying the dependencies, and choose a target language compatible with that. In which case, the proof of typability would concentrate both the difficulties for typing the store-passing part of the translation, and the difficulties related to type

[7] As in for the $\overline{\lambda}_{[lv\tau\star]}$-calculus, it is in fact a continuation-and-store passing style translation, but we refer to it as continuation-passing style for conciseness.

[8] It is folklore that conjunctions, disjunctions and even co-inductive types can be encoded in System F, and thus in System F_Υ. Adding primitive constructions both in the syntax of types and programs is thus just a matter of convenience to simplify the translation. We can thus consider without lost of generality that the language includes these constructions, since alternatively, one could combine the CPS translation with the encoding to obtain a translation to "pure" System F_Υ.

dependencies as for the translation of $\mathrm{dL}_{\hat{\mathrm{tp}}}$. For instance, we could pick the calculus of constructions [28] as a very general target language, in which we would dispose of dependent types and of the expressive power to encode the type of second-order vectors from F_Υ.

We choose to leave the choice of our target language ambiguous, and give the most general translation possible. We thus assume that the proof terms of the target language contain at least constructors for pairs, injections, equality and the same destructors as in dPA^ω (*i.e.* `split`, `case`, `dest`, `subst`, `exfalso`), as well as a way to encode vectors. We do not add substitutions to rename variables, but a thorough definition of the translation should also include an explicit renaming procedure, for the reasons invoked in Section 6.4.1.

This being said, the translation is derived directly from the small-step reduction rules. As for the $\overline{\lambda}_{[lv\tau\star]}$-calculus, the different levels p, e, V, f, v and t, π, V_t are reflected in a translation $[\![\cdot]\!]_\iota$ for each level ι. The main subtlety concerns the way we handle inductive and co-inductive fixpoints, and more generally the store. Observe that in dLPA^ω we managed to delimit the unfolding of fixpoints to the store, everything happening as if they were special cells producing computations. In other words, we could have been one step further to remove fixpoints from the syntax of proofs, limiting their occurrences strictly to the store. This is actually what is done through the translation, where we mark some cells with `IND` and `COFIX`. The computational content of the fixpoint is thus decomposed step by step, each step being produced by the `lookup` function, that is defined (in pseudo-code) as follows:

```
lookup $\kappa\,\tau_1[\kappa := p]\tau_2\,k$  :=  match $(\kappa,p)$ with
  | $\alpha, e$                                   $\mapsto$  $e\,\tau_1[a := V]\tau_2\,k$
  | $a, V$                                   $\mapsto$  $V\,\tau_1[a := V]\tau_2\,k$
  | $a, \mathtt{COFIX}_{bx}^{t} p$                        $\mapsto$  $(p[t/x][b'/b])\,\tau_1[b' := [\![\lambda y.\mathtt{cofix}_{bx}^{y}[p]]\!]_v]\,(\lambda\tau q.q\,\tau[a := q]\tau_2\,k)$
  | $a, \mathtt{IND}_{bx}^{0}[p_0 \mid p_S]$                 $\mapsto$  $p_0\,\tau_1\,(\lambda\tau q.q\,\tau[a := q]\tau_2\,k)$
  | $a, \mathtt{IND}_{bx}^{S(t)}[p_0 \mid p_S]$              $\mapsto$  $(p_S[t/x][b'/b])\,\tau_1[b' := \mathtt{IND}_{bx}^{t}[p_0 \mid p_S]]\,(\lambda\tau q.q\,\tau[a := q]\tau_2\,k)$
```

where in each case b' is fresh. In practice, this simply corresponds to a store where cells include a flag so that the `lookup` function given above could be implemented in the target language by means of pattern-matching using injections and `case`. The lookup function is now the only piece of the whole translation which actually has the computational content of a fixpoint.

The full translation is given by Figure 8.7, and is by construction correct with respect to reduction. In particular, we could again prove by a tedious induction on the reduction $\rightsquigarrow_s$ that the normalization is preserved:

Proposition 8.11. *If $[\![c\tau]\!]_l$ normalizes, then so does $c\tau$ for $\rightsquigarrow_s$.*

In what concerns the typing of the translation, in the case where we erase the dependencies, it would simply amount to the typing of the translation for the $\overline{\lambda}_{[lv\tau\star]}$-calculus, that is to say that the translation of typing judgments for proofs (resp. contexts, etc) will be of the shape:

$$[\![\Gamma \vdash^\sigma p : A]\!] \quad \triangleq \quad (\vdash\ [\![p]\!]_p : [\![\Gamma]\!]_\Gamma \rhd_p \iota(A))$$

where again:

$$\Upsilon \rhd_p A \quad \triangleq \quad \forall Y <: \Upsilon.\, Y \to (Y \rhd_e A) \to \bot\,.$$

The only novelty with respect to the CPS translation for the $\overline{\lambda}_{[lv\tau\star]}$-calculus sites in the `lookup` function in the cases of `IND` and `COFIX`. However, it is easy to check that in both cases, through the translation (and already in the small-step reduction system) these elements take a continuation at level f and put in active position a proof at level p in front of a continuation which is built to be at level e. In particular, types are respected in the sense that `lookup` $a\,(\tau[a := \mathtt{COFIX}_{bx}^{t}[\![p]\!]_p])\,k_f$ is indeed of type $\bot$. We claim

Commands

$$\begin{aligned}
\llbracket\langle p\|e\rangle\rrbracket_c\,\tau &= \llbracket p\rrbracket_p\,\tau\,\llbracket e\rrbracket_e \\
\llbracket\langle p\|\hat{\mathfrak{tp}}\rangle\rrbracket_c\,\tau &= \llbracket p\rrbracket_p\,\tau & \qquad \llbracket\langle t\|\pi\rangle\rrbracket_c\,\tau &= \llbracket t\rrbracket_t\,\tau\,\pi \\
\llbracket\langle\check{\mathfrak{tp}}\|e\rangle\rrbracket_c\,\tau &= \llbracket e\rrbracket_e\,\tau
\end{aligned}$$

Proofs

$$\begin{aligned}
&& \llbracket\mu\hat{\mathfrak{tp}}.c\rrbracket_p\,\tau\,k &= (\llbracket c\rrbracket_c\,\tau)\,k \\
&& \llbracket\mu\alpha.c\rrbracket_p\,\tau\,k &= \llbracket c\rrbracket_c\,\tau[\alpha := k] \\
&& \llbracket(p_1,p_2)\rrbracket_p\,\tau\,k &= \llbracket p_1\rrbracket_p\,\tau\,(\lambda\tau_1 q_1.\llbracket p_2\rrbracket_p\,\tau_1[a_1 := q_1]\,(\lambda\tau_2 q_2.k\,\tau_2[a_2 := q_2]\,\llbracket(a_1,a_2)\rrbracket_V)) \\
&& \llbracket\iota_i(p)\rrbracket_p\,\tau\,k &= \llbracket p\rrbracket_p\,\tau\,(\lambda\tau q.k\,\tau[a := q]\,\llbracket\iota_i(a)\rrbracket_V) \\
&& \llbracket(t,p)\rrbracket_p\,\tau\,k &= \llbracket p\rrbracket_p\,\tau\,(\lambda\tau.\llbracket t\rrbracket_t\,\tau\,(\lambda\tau xa.k\,\tau\,\llbracket(x,a)\rrbracket_V)) \\
(a\text{ fresh}) && \llbracket\mathtt{cofix}^t_{bx}[p]\rrbracket_p\,\tau\,k &= (\llbracket t\rrbracket_t\,\tau\,(\lambda\tau y.\llbracket a\rrbracket_p\,\tau[a := \mathtt{COFIX}^y_{bx}\llbracket p\rrbracket_p]))\,k \\
(a\text{ fresh}) && \llbracket\mathtt{ind}^t_{bx}[p\,|\,p_s]\rrbracket_p\,\tau\,k &= (\llbracket t\rrbracket_t\,\tau\,(\lambda\tau y.\llbracket a\rrbracket_p\,\tau[a := \mathtt{IND}^y_{bx}[\llbracket p\rrbracket_p\,|\,\llbracket q\rrbracket_p]]))\,k \\
&& \llbracket V\rrbracket_p\,\tau\,k &= k\,\tau\,\llbracket V\rrbracket_V
\end{aligned}$$

Contexts

$$\begin{aligned}
\llbracket\tilde\mu a.c\tau'\rrbracket_e\,\tau\,V &= \llbracket c\rrbracket_c\,(\tau[a := V]\llbracket\tau'\rrbracket_\tau) & \qquad \llbracket\alpha\rrbracket_e\,\tau\,V &= \mathtt{lookup}\;\alpha\,\tau\,V \\
\llbracket\tilde\mu\check{\mathfrak{tp}}.c\rrbracket_e\,\tau\,V &= (\llbracket c\rrbracket_c\,\tau)\,V & \qquad \llbracket f\rrbracket_e\,\tau\,V &= V\,\tau\,\llbracket f\rrbracket_f
\end{aligned}$$

Weak values

$$\begin{aligned}
\llbracket a\rrbracket_V\,\tau\,k_v &= \mathtt{lookup}\;a\,\tau\,k_v & \qquad \llbracket v\rrbracket_V\,\tau\,k_v &= k_v\,\tau\,\llbracket V\rrbracket_v
\end{aligned}$$

Forcing contexts

$$\begin{aligned}
&& \llbracket t\cdot e\rrbracket_f\,\tau\,v &= (\llbracket t\rrbracket_t\,\tau\,(\lambda\tau x.v\,\tau\,x))\,\llbracket e\rrbracket_e \\
(q_N\in\textsc{nef}) && \llbracket q_N\cdot e\rrbracket_f\,\tau\,v &= (\llbracket q_N\rrbracket_p\,\tau\,(\lambda\tau q'.v\,\tau\,q'))\,\llbracket e\rrbracket_e \\
(q\notin\textsc{nef}) && \llbracket q\cdot e\rrbracket_f\,\tau\,v &= \llbracket q\rrbracket_p\,\tau\,(\lambda\tau q'.v\,\tau\,q'\,\llbracket e\rrbracket_e) \\
(b_i\text{ fresh}) && \llbracket\tilde\mu(a_1,a_2).c\rrbracket_f\,\tau\,v &= \mathtt{split}\;v\;\mathtt{as}\,(b_1,b_2)\;\mathtt{in}\,(\llbracket c\rrbracket_c\,\tau[a_1 := b_1][a_2 := b_2]) \\
(b_i\text{ fresh}) && \llbracket\tilde\mu[a_1.c_1\,|\,a_2.c_2]\rrbracket_f\,\tau\,v &= \mathtt{case}\;v\;\mathtt{of}\,[b_1.\llbracket c_1\rrbracket_c\,\tau[a_1 := b_1]\,|\,b_2.\llbracket c_2\rrbracket_c\,\tau[a_2 := b_2,]] \\
(b\text{ fresh}) && \llbracket\tilde\mu(x,a).c\rrbracket_f\,\tau\,v &= \mathtt{dest}\;v\;\mathtt{as}\,(y,b)\;\mathtt{in}\,(\lambda x.\llbracket c\rrbracket_c)y\,\tau[a := b] \\
&& \llbracket\tilde\mu^{=}.c\rrbracket_f\,\tau\,v &= \mathtt{subst}\;v\,\llbracket c\rrbracket_c\,\tau \\
&& \llbracket[]\rrbracket_f\,\tau\,v &= \mathtt{exfalso}\;v
\end{aligned}$$

Strong values

$$\begin{aligned}
\llbracket\lambda x.p\rrbracket_v\,\tau\,V_t\,e &= \llbracket p\rrbracket_p[V_t/x]\,\tau\,e & \qquad \llbracket(a_1,a_2)\rrbracket_v &= (\llbracket a_1\rrbracket_V,\llbracket a_2\rrbracket_V) \\
\llbracket\lambda a.p\rrbracket_v\,\tau\,V\,e &= \llbracket p\rrbracket_p\,\tau[a := V]\,e & \qquad \llbracket\iota_i(a)\rrbracket_v &= \iota_i(\llbracket a\rrbracket_V) \\
\llbracket\mathtt{refl}\rrbracket_v &= \mathtt{refl} & \qquad \llbracket(V_t,a)\rrbracket_v &= (\llbracket V_t\rrbracket_{V_t},\llbracket a\rrbracket_V)
\end{aligned}$$

Environments

$$\begin{aligned}
&& \llbracket\varepsilon\rrbracket_\tau &= \varepsilon \\
\llbracket\tau[a := \mathtt{cofix}^{V_t}_{bx}[p]]\rrbracket_\tau &= \llbracket\tau\rrbracket_\tau[a := \mathtt{COFIX}^{\llbracket V_t\rrbracket_{V_t}}_{bx}\llbracket p\rrbracket_p] & \qquad \llbracket\tau[a := V]\rrbracket_\tau &= \llbracket\tau\rrbracket_\tau[a := \llbracket V\rrbracket_v] \\
\llbracket\tau[a := \mathtt{ind}^{V_t}_{bx}[p\,|\,q]]\rrbracket_\tau &= \llbracket\tau\rrbracket_\tau[a := \mathtt{IND}^{\llbracket V_t\rrbracket_{V_t}}_{bx}[\llbracket p\rrbracket_p\,|\,\llbracket q\rrbracket_p]] & \qquad \llbracket\tau[\alpha := e]\rrbracket_\tau &= \llbracket\tau\rrbracket_\tau[\alpha := \llbracket e\rrbracket_e]
\end{aligned}$$

Terms

$$\begin{aligned}
\llbracket V_t\rrbracket_t\,\tau\,k_t &= k_t\,\tau\,\llbracket V_t\rrbracket_{V_t} \\
\llbracket S(u)\rrbracket_t\,\tau\,k_t &= \llbracket u\rrbracket_t\,\tau\,(\lambda\tau x.k_t\,\tau\,\llbracket S(x)\rrbracket_{V_t})) \\
\llbracket t\,u\rrbracket_t\,\tau\,k_t &= \llbracket t\rrbracket_t\,\tau\,\lambda\tau v.\llbracket u\rrbracket_t\,\tau\,(\lambda\tau w.v\,w\,\tau\,k_t)) \\
\llbracket\mathtt{wit}\,(p)\rrbracket_t\,\tau\,k_t &= \llbracket p\rrbracket_p\,\tau\,(\lambda\tau q.q\,\tau\,(\lambda\alpha.(\llbracket\tilde\mu(x,a).\langle x\|\alpha\rangle\rrbracket_f)\,k_t)) \\
\llbracket\mathtt{rec}^t_{xy}[u_0\,|\,u_S]\rrbracket_t\,\tau\,k_t &= \llbracket t\rrbracket_t\,\tau\,(\lambda\tau z.\mathtt{rec}^z_{xy}[\llbracket u_0\rrbracket_t\,|\,\llbracket u_S\rrbracket_t]\,\tau\,k_t)
\end{aligned}$$

$$\begin{aligned}
\llbracket u\cdot\pi\rrbracket_\pi\,\tau\,v &= \llbracket u\rrbracket_t\,\tau\,(\lambda\tau w.v\,\tau\,w\,\llbracket\pi\rrbracket_\pi) & \qquad \llbracket x\rrbracket_{V_t} &= x \\
\llbracket\tilde\mu x.c\rrbracket_\pi\,\tau\,v &= (\llbracket c\rrbracket_c\,\tau)[v/x] & \qquad \llbracket S(V_t)\rrbracket_{V_t} &= S(\llbracket V_t\rrbracket_{V_t}) \\
\llbracket 0\rrbracket_{V_t} &= 0 & \qquad \llbracket\lambda x.t\rrbracket_{V_t} &= \lambda\tau xk.\llbracket t\rrbracket_t\,\tau\,k
\end{aligned}$$

Figure 8.7: Continuation-and-store-passing style translation

than once we understood how the translation of the $\overline{\lambda}_{[lv\tau\star]}$-calculus was typed, this setting is more or less the same and should not give us a hard time.

However, in the case where we would like to obtain a translation compatible with dependent types, we know that we need to refine the typing of terms and NEF proof terms, as we did in dL$_{\hat{\mathfrak{tp}}}$. This is certainly possible, in particular given a NEF proof term p, it is still possible to pass the continuation $\lambda\tau a.a$ to $[\![p_N]\!]_p$ to force the extraction of a proof p_N^+. This should allow us to refine the type of $[\![p_N]\!]_p$ to obtain something like:

$$\Upsilon \triangleright_p A \quad \triangleq \quad \forall Y <: \Upsilon.\, Y \to \forall R.'\Pi a : (Y \triangleright_e A).R(a) \to R(p_N^+))\,.$$

However, due to the laziness and the two layers of alternation between proof and contexts, we should probably process to a second extraction to obtain a strong value, and cleverly handle the store while doing so. In the absence of a real motivation for such a translation, we did not take the time to study the question more in depth. However we are confident in the fact that the main difficulties has been studied in the previous chapters, so that if it was worthwhile, with time and rigor, it should be possible to methodically obtain a translation of types compatible with the dependencies.

8.4 Normalization of dLPA$^\omega$

8.4.1 A realizability interpretation of dLPA$^\omega$

We shall now present the realizability interpretation of dLPA$^\omega$, which will finally give us a proof of its normalization. Here again, the interpretation combines ideas of the interpretations for the $\overline{\lambda}_{[lv\tau\star]}$-calculus (Chapter 6) and for dL$_{\hat{\mathfrak{tp}}}$ through the embedding in Lepigre's calculus (Chapter 7). Namely, as for the $\overline{\lambda}_{[lv\tau\star]}$-calulus, formulas will be interpreted by sets of proofs-in-store of the shape $(p|\tau)$, and the orthogonality will be defined between proofs-in-store $(p|\tau)$ and contexts-in-store $(e|\tau')$ such that the stores τ an τ' are compatible.

We recall the main definition necessary to the realizability interpretation:

Definition 8.12 (Proofs-in-store)**.** We call *closed proof-in-store* (resp. *closed context-in-store*, *closed term-in-store*, etc) the combination of a proof p (resp. context e, term t, etc) with a closed store τ such that $FV(p) \subseteq \mathrm{dom}(\tau)$. We use the notation $(p|\tau)$ to denote such a pair. In addition, we denote by Λ_p (resp. Λ_e, etc.) the set of all proofs and by Λ_p^τ (resp. Λ_e^τ, etc.) the set of all proofs-in-store. ⌟

We denote the sets of closed closures by C_0, and we identify again $(c|\tau)$ with the closure $c\tau$ when c is closed in τ. We can now define the notion of pole, which has to satisfy an extra condition due to the presence of delimited continuations

Definition 8.13 (Pole)**.** A subset $\perp\!\!\!\perp \in C_0$ is said to be *saturated* or *closed by anti-reduction* whenever for all $(c|\tau), (c'|\tau') \in C_0$, we have

$$(c'\tau' \in \perp\!\!\!\perp) \;\wedge\; (c\tau \to c'\tau') \;\Rightarrow\; (c\tau \in \perp\!\!\!\perp)$$

It is said to be *closed by store extension* if whenever $c\tau$ is in $\perp\!\!\!\perp$, for any store τ' extending τ, $c\tau'$ is also in $\perp\!\!\!\perp$:

$$(c\tau \in \perp\!\!\!\perp) \;\wedge\; (\tau \lhd \tau') \;\Rightarrow\; (c\tau' \in \perp\!\!\!\perp)$$

It is said to be *closed under delimited continuations* if whenever $c[e/\hat{\mathfrak{tp}}]\tau$ (resp. $c[V/\check{\mathfrak{tp}}]\tau$) is in $\perp\!\!\!\perp$, then $\langle \mu\hat{\mathfrak{tp}}.c \| e\rangle\tau$ (resp .$\langle V \| \tilde{\mu}\check{\mathfrak{tp}}.c\rangle\tau$) belongs to $\perp\!\!\!\perp$:

$$(c[e/\hat{\mathfrak{tp}}]\tau \in \perp\!\!\!\perp) \;\Rightarrow\; (\langle \mu\hat{\mathfrak{tp}}.c \| e\rangle\tau \in \perp\!\!\!\perp) \qquad\qquad (c[V/\check{\mathfrak{tp}}]\tau \in \perp\!\!\!\perp) \;\Rightarrow\; (\langle V \| \tilde{\mu}\check{\mathfrak{tp}}.c\rangle\tau \in \perp\!\!\!\perp)$$

A *pole* is defined as any subset of C_0 that is closed by anti-reduction, by store extension and under delimited continuations. ⌟

We can verify that the set of normalizing command is indeed a pole:

Proposition 8.14. *The set* $\bot\!\!\!\bot_{\Downarrow} = \{c\tau \in C_0 : c\tau \text{ normalizes }\}$ *is a pole.*

Proof. The first two conditions have already been verified for the $\overline{\lambda}_{[lv\tau\star]}$-calculus. The third one is straightforward, since if a closure $\langle\mu\hat{\mathfrak{tp}}.c\|e\rangle\tau$ is not normalizing, it is easy to verify that $c[e/\hat{\mathfrak{tp}}]$ is not normalizing either. Roughly, there is only two possible reduction steps for a command $\langle\mu\hat{\mathfrak{tp}}.c\|e\rangle\tau$: either it reduces to $\langle\mu\hat{\mathfrak{tp}}.c'\|e\rangle\tau'$, in which case $c[e/\hat{\mathfrak{tp}}]\tau$ also reduces to a closure which is almost $(c'\tau')[e/\hat{\mathfrak{tp}}]$; or c is of the shape $\langle p\|\hat{\mathfrak{tp}}\rangle$ and it reduces to $c[e/\hat{\mathfrak{tp}}]\tau$. In both cases, if $\langle\mu\hat{\mathfrak{tp}}.c\|e\rangle\tau$ can reduce, so can $c[e/\hat{\mathfrak{tp}}]\tau$. The same reasoning allows us to show that if $c[V/\check{\mathfrak{tp}}]\tau$ normalizes, then so does $\langle V\|\tilde{\mu}\check{\mathfrak{tp}}.c\rangle\tau$ for any value sV. □

We now recall the notion of compatible stores, which allows us to define an orthogonality relation between proofs- and contexts-in-store.

Definition 8.15 (Compatible stores and union)**.** Let τ and τ' be stores, we say that:

- they are *independent* and note $\tau\#\tau'$ when $\mathrm{dom}(\tau)\cap\mathrm{dom}(\tau')=\emptyset$.
- they are *compatible* and note $\tau\diamond\tau'$ whenever for all variables a (resp. co-variables α) present in both stores: $a\in\mathrm{dom}(\tau)\cap\mathrm{dom}(\tau')$; the corresponding proofs (resp. contexts) in τ and τ' coincide.
- τ' is an *extension* of τ and we write $\tau\lhd\tau'$ whenever $\mathrm{dom}(\tau)\subseteq\mathrm{dom}(\tau')$ and $\tau\diamond\tau'$.
- the compatible union $\overline{\tau\tau'}$ of compatible closed stores τ and τ', is defined as $\mathsf{join}(\tau,\tau')$, which itself given by:

$$\begin{array}{rcll}
\mathsf{join}(\tau_0[a:=p]\tau_1,\tau_0'[a:=p]\tau_1') & \triangleq & \tau_0\tau_0'[a:=p]\mathsf{join}(\tau_1,\tau_1') & (\text{if } \tau_0\#\tau_0')\\
\mathsf{join}(\tau_0[\alpha:=e]\tau_1,\tau_0'[\alpha:=e]\tau_1') & \triangleq & \tau_0\tau_0'[\alpha:=e]\mathsf{join}(\tau_1,\tau_1') & (\text{if } \tau_0\#\tau_0')\\
\mathsf{join}(\tau,\tau') & \triangleq & \tau\tau' & (\text{if } \tau\#\tau')
\end{array}$$ ⌟

The next lemma (which follows from the previous definition) states the main property we will use about union of compatible stores.

Lemma 8.16. *If τ and τ' are two compatible stores, then $\tau\lhd\overline{\tau\tau'}$ and $\tau'\lhd\overline{\tau\tau'}$. Besides, if τ is of the form $\tau_0[x:=t]\tau_1$, then $\overline{\tau\tau'}$ is of the form $\overline{\tau_0}[x:=t]\overline{\tau_1}$ with $\tau_0\lhd\overline{\tau_0}$ and $\tau_1\lhd\overline{\tau_1}$.*

We recall the definition of the orthogonality relation with respect to a pole, which is identical to the one for the $\overline{\lambda}_{[lv\tau\star]}$-calculus:

Definition 8.17 (Orthogonality)**.** Given a pole $\bot\!\!\!\bot$, we say that a proof-in-store $(p|\tau)$ is *orthogonal* to a context-in-store $(e|\tau')$ and write $(p|\tau)\bot\!\!\!\bot(e|\tau')$ if τ and τ' are compatible and $\langle p\|e\rangle\overline{\tau\tau'}\in\bot\!\!\!\bot$. The orthogonality between terms and coterms is defined identically. ⌟

We are now equipped to define the realizability interpretation of dLPA$^\omega$. Firstly, in order to simplify the treatment of coinductive formulas, we extend the language of formulas with second-order variables $X,Y,\ldots$ and we replace $\nu_{fx}^{t}A$ by $\nu_{Xx}^{t}A[X(y)/f(y)=0]$. The typing rule for co-fixpoint operators then becomes:

$$\frac{\Gamma\vdash^{\sigma} t:T \quad \Gamma,x:T,b:\forall y^{T}.X(y)\vdash^{\sigma} p:A \quad X\text{ positive in }A \quad X\notin FV(\Gamma)}{\Gamma\vdash^{\sigma}\mathtt{cofix}_{bx}^{t}[p]:\nu_{Xx}^{t}A}\ (\mathtt{cofix})$$

Secondly, as in the interpretation of dL$_{\hat{\mathfrak{tp}}}$ through Lepigre's calculus, we introduce two new predicates, $p\in A$ for NEF proofs and $t\in T$ for terms. This allows us to decompose the dependent products and sums into:

$$\begin{array}{rcl@{\qquad}rcll}
\forall x^{T}.A & \triangleq & \forall x.(x\in T\to A) & \Pi a:A.B & \triangleq & \forall a.(a\in A\to B) & (\text{if } a\in FV(B))\\
\exists x^{T}.A & \triangleq & \exists x.(x\in T\to A) & \Pi a:A.B & \triangleq & A\to B & (\text{if } a\notin FV(B))
\end{array}$$

This corresponds to the language of formulas and types defined by:

$$\begin{array}{lrcl}
\textbf{Types} & T,U & ::= & \mathbb{N} \mid T \to U \mid t \in T \\
\textbf{Formulas} & A,B & ::= & \top \mid \bot \mid X(t) \mid t = u \mid A \wedge B \mid A \vee B \mid a \in A \mid \forall x.A \mid \exists x.A \mid \forall a.A \mid \nu^t_{Xx}A
\end{array}$$

and to the following inference rules:

$$\frac{\Gamma \vdash^\sigma v : A \quad a \notin FV(\Gamma)}{\Gamma \vdash^\sigma v : \forall a.A}\,(\forall^a_r) \qquad \frac{\Gamma \vdash^\sigma v : A \quad x \notin FV(\Gamma)}{\Gamma \vdash^\sigma v : \forall x.A}\,(\forall^x_r) \qquad \frac{\Gamma \vdash^\sigma v : A[t/x]}{\Gamma \vdash^\sigma v : \exists x.A}\,(\exists^x_r)$$

$$\frac{\Gamma \vdash^\sigma e : A[q/a] \quad q\ \textsc{nef}}{\Gamma \vdash^\sigma e : (\forall a.A)^{\perp\!\!\!\perp}}\,(\forall^a_l) \qquad \frac{\Gamma \vdash^\sigma e : A[t/x]}{\Gamma \vdash^\sigma e : (\forall x.A)^{\perp\!\!\!\perp}}\,(\forall^x_l) \qquad \frac{\Gamma \vdash^\sigma e : A \quad x \notin FV(\Gamma)}{\Gamma \vdash^\sigma e : (\exists x.A)^{\perp\!\!\!\perp}}\,(\exists^x_l)$$

$$\frac{\Gamma \vdash^\sigma p : A \quad p\ \textsc{nef}}{\Gamma \vdash^\sigma p : p \in A}\,(\in^p_r) \qquad \frac{\Gamma \vdash^\sigma e : A^{\perp\!\!\!\perp}}{\Gamma \vdash^\sigma e : (q \in A)^{\perp\!\!\!\perp}}\,(\in^p_l) \qquad \frac{\Gamma \vdash^\sigma t : T}{\Gamma \vdash^\sigma t : t \in T}\,(\in^t_r) \qquad \frac{\Gamma \vdash^\sigma \pi : T^{\perp\!\!\!\perp}}{\Gamma \vdash^\sigma \pi : (t \in T)^{\perp\!\!\!\perp}}\,(\in^t_l)$$

These rules are exactly the same as in Lepigre's calculus [108] up to our stratified presentation in a sequent calculus fashion and modulo our syntactic restriction to NEF proofs instead of his semantical restriction. It is a straightforward verification to check that the typability is maintained through the decomposition of dependent products and sums.

Another similarity with Lepigre's realizability model is that truth/falsity values will be closed under observational equivalence of proofs and terms. To this purpose, for each store τ we introduce the relation $\equiv_\tau$, which we define as the reflexive-transitive-symmetric closure of the relation $\triangleright_\tau$:

$$\begin{array}{rclll}
t & \triangleright_\tau & t' & \text{whenever} & \exists \tau', \forall \pi, (\langle t \| \pi \rangle \tau \to \langle t' \| \pi \rangle \tau' \\
p & \triangleright_\tau & q & \text{whenever} & \exists \tau', \forall f\ (\langle p \| f \rangle \tau \to \langle q \| f \rangle \tau')
\end{array}$$

All this being settled, it only remains to determine how to interpret coinductive formulas. While it would be natural to try to interpret them by fixpoints in the semantics, this poses difficulties for the proof of adequacy. We will discuss this matter in the next section, but as for now, we will give a simpler interpretation. We stick to the intuition that since `cofix` operators are lazily evaluated, they actually are realizers of every finite approximation of the (possibly infinite) coinductive formulas. Consider for instance the case of a stream

$$\mathtt{str}^0_\infty p \triangleq \mathtt{cofix}^0_{bx}[(px, b(S(x)))]$$

of type $\nu^0_{fx}A(x) \wedge f(S(x)) = 0$. Such stream will produce on demand any tuple $(p0,(p1,\ldots(pn,\square)\ldots)$ where $\square$ denotes the fact that it could be any term, in particular $\mathtt{str}^{n+1}_\infty p$. So that $\mathtt{str}^0_\infty p$ should be a successful defender of the formula

$$(A(0) \wedge (A(1) \wedge \ldots(A(n) \wedge \top)\ldots)$$

Since `cofix` operators only reduce when they are bound to a variable in front of a forcing context, it suggests to interpret the coinductive formula $\nu^0_{fx}A(x) \wedge X(S(x))$ at level f as the union of all the opponents to a finite approximation.

To this end, given a coinductive formula $\nu^0_{Xx}A$ where X is positive in A, we define its finite approximations by:

$$F^0_{A,t} \triangleq \top \qquad\qquad F^{n+1}_{A,t} \triangleq A[t/x][F^n_{A,y}/X(y)]$$

Since f is positive in A, we have for any integer n and any term t that $\|F^n_{A,t}\|_f \subseteq \|F^{n+1}_{A,t}\|_f$. We can finally define the interpretation of coinductive formulas by:

$$\|\nu^t_{Xx}A\|_f \triangleq \bigcup_{n \in \mathbb{N}} \|F^n_{A,t}\|_f$$

$$\begin{array}{rcl}
\|\bot\|_f & \triangleq & \Lambda_f^\tau \\
\|\top\|_f & \triangleq & \emptyset \\
\|\dot{F}(t)\|_f & \triangleq & F(t) \\
\|t = u\|_f & \triangleq & \begin{cases} \{(\tilde{\mu}\!=\!.c|\tau) : c\tau \in \perp\!\!\!\perp\} & \text{if } t \equiv_\tau u \\ \Lambda_f^\tau & \text{otherwise} \end{cases} \\
\|p \in A\|_f & \triangleq & \{(V|\tau) \in |A|_V : V \equiv_\tau p\}^{\perp\!\!\!\perp_f} \\
\|T \to B\|_f & \triangleq & \{(V_t \cdot e|\tau) : (V_t|\tau) \in |t \in T|_{V_t} \wedge (e|\tau) \in \|B\|_e\} \\
\|A \to B\|_f & \triangleq & \{(V \cdot e|\tau) : (V|\tau) \in |A|_V \wedge (e|\tau) \in \|B\|_e\} \\
\|T \wedge A\|_f & \triangleq & \{(\tilde{\mu}(x,a).c|\tau) : \forall \tau', V_t \in |T|_{V_t}^{\tau'}, V \in |A|_V^{\tau'}, \tau \diamond \tau' \Rightarrow c[V_t/x]\overline{\tau\tau'}[a := V] \in \perp\!\!\!\perp\} \\
\|A_1 \wedge A_2\|_f & \triangleq & \{(\tilde{\mu}(a_1,a_2).c|\tau) : \forall \tau', V_1 \in |A_1|_V^{\tau'}, V_2 \in |A_2|_V^{\tau'}, \tau \diamond \tau' \Rightarrow c\overline{\tau\tau'}[a_1 := V_1][a_2 := V_2] \in \perp\!\!\!\perp\} \\
\|A_1 \vee A_2\|_f & \triangleq & \{(\tilde{\mu}[a_1.c_1|a_2.c_2]|\tau) : \forall \tau', V \in |A_i|_V^{\tau'}, \tau \diamond \tau' \Rightarrow c\overline{\tau\tau'}[a_i := V] \in \perp\!\!\!\perp\} \\
\|\exists x.A\|_f & \triangleq & \bigcap_{t \in \Lambda_t} \|A[t/x]\|_f \\
\|\forall x.A\|_f & \triangleq & (\bigcap_{t \in \Lambda_t} \|A[t/x]\|_f^{\perp\!\!\!\perp_v})^{\perp\!\!\!\perp_f} \\
\|\forall a.A\|_f & \triangleq & (\bigcap_{t \in \Lambda_p} \|A[p/a]\|_f^{\perp\!\!\!\perp_v})^{\perp\!\!\!\perp_f} \\
\|\nu_{fx}^t A\|_f & \triangleq & \bigcup_{n \in \mathbb{N}} \|F_{A,t}^n\|_f \\
|A|_V & \triangleq & \|A\|_f^{\perp\!\!\!\perp_V} \;=\; \{(V|\tau) : \forall f\tau', \tau \diamond \tau' \wedge (f|\tau') \in \|A\|_f \Rightarrow (V|\tau) \perp\!\!\!\perp (F|\tau')\} \\
\|A\|_e & \triangleq & |A|_V^{\perp\!\!\!\perp_e} \;=\; \{(E|\tau) : \forall V\tau', \tau \diamond \tau' \wedge (V|\tau') \in |A|_V \Rightarrow (V|\tau') \perp\!\!\!\perp (E|\tau)\} \\
|A|_p & \triangleq & \|A\|_e^{\perp\!\!\!\perp_p} \;=\; \{(p|\tau) : \forall E\tau', \tau \diamond \tau' \wedge (E|\tau') \in \|A\|_E \Rightarrow (t|\tau) \perp\!\!\!\perp (E|\tau')\} \\
 & & \\
|\mathbb{N}|_{V_t} & \triangleq & \{(S^n(0)|\tau), n \in \mathbb{N}\} \\
|t \in T|_{V_t} & \triangleq & \{(V_t|\tau) \in |T|_{V_t} : V_t \equiv_\tau t\} \\
|T \to U|_{V_t} & \triangleq & \{(\lambda x.t|\tau) : \forall V_t\tau', \tau \diamond \tau' \wedge (V_t|\tau') \in |T|_{V_t} \Rightarrow (t[V_t/x]|\overline{\tau\tau'}) \in |U|_t\} \\
|T|_\pi & \triangleq & |A|_{V_t}^{\perp\!\!\!\perp_\pi} \;=\; \{(F|\tau) : \forall v\tau', \tau \diamond \tau' \wedge (v|\tau') \in |A|_v \Rightarrow (v|\tau') \perp\!\!\!\perp (F|\tau)\} \\
|T|_t & \triangleq & |A|_\pi^{\perp\!\!\!\perp_t} \;=\; \{(V|\tau) : \forall F\tau', \tau \diamond \tau' \wedge (F|\tau') \in \|A\|_F \Rightarrow (V|\tau) \perp\!\!\!\perp (F|\tau')\}
\end{array}$$

where:

- $p \in S^\tau$ (resp. e,V,etc.) denote $(p|\tau) \in S$ (resp. $(e|\tau)$,$(V|\tau)$,etc.),
- F is a function from Λ_t to $\mathcal{P}(\Lambda_f^\tau)_{/\equiv_\tau}$.

Figure 8.8: Realizability interpretation for dLPA$^\omega$

The realizability interpretation of closed formulas and types is defined in Figure 8.8 by induction on the structure of formulas at level f, and by orthogonality at levels V,e,p. When S is a subset of $\mathcal{P}(\Lambda_p^\tau)$ (resp. $\mathcal{P}(\Lambda_e^\tau)$,$\mathcal{P}(\Lambda_t^\tau)$,$\mathcal{P}(\Lambda_\pi^\tau)$), we use the notation $S^{\perp\!\!\!\perp_f}$ (resp. $S^{\perp\!\!\!\perp_V}$, etc.) to denote its orthogonal set restricted to Λ_f^τ (resp. Λ_V^τ, etc.):

$$S^{\perp\!\!\!\perp_f} \triangleq \{(f|\tau) \in \Lambda_f^\tau : \forall (p|\tau') \in S, \tau \diamond \tau' \Rightarrow \langle p \| f \rangle \overline{\tau\tau'} \in \perp\!\!\!\perp\}$$

At level f, closed formulas are interpreted by sets of strong forcing contexts-in-store $(f|\tau)$. As observed in the previous section, these sets are besides closed under the relation $\equiv_\tau$ along their component τ, we thus denote them by $\mathcal{P}(\Lambda_f^\tau)_{/\equiv_\tau}$. Second-order variables $X, Y, \ldots$ are then interpreted by functions from the set of terms Λ_t to $\mathcal{P}(\Lambda_f^\tau)_{/\equiv_\tau}$ and as usual for each such function F we add a predicate symbol $\dot{F}$ in the language.

We shall now prove the adequacy of the interpretation with respect to the type system. To this end, we need to recall a few definitions and lemmas. Since stores only contain proof terms, we need to define valuations for term variables in order to close formulas[9]. These valuations are defined by the usual grammar:

$$\rho ::= \varepsilon \mid \rho[x \mapsto V_t] \mid \rho[X \mapsto \dot{F}]$$

We denote by $(p|\tau)_\rho$ (resp. p_ρ, A_ρ) the proof-in-store $(p|\tau)$ where all the variables $x \in \mathrm{dom}(\rho)$ (resp. $X \in \mathrm{dom}(\rho)$) have been substituted by the corresponding term $\rho(x)$ (resp. falsity value $\rho(x)$).

Definition 8.18. Given a closed store τ, a valuation ρ and a fixed pole $\perp\!\!\!\perp$, we say that the pair (τ,ρ) *realizes* Γ, which we write[10] $(\tau,\rho) \Vdash \Gamma$, if:

1. for any $(a : A) \in \Gamma$, $(a|\tau)_\rho \in |A_\rho|_V$
2. for any $(\alpha : A_\rho^{\perp\!\!\!\perp}) \in \Gamma$, $(\alpha|\tau)_\rho \in \|A_\rho\|_e$
3. for any $\{a|p\} \in \sigma$, $a \equiv_\tau p$
4. for any $(x : T) \in \Gamma$, $x \in \mathrm{dom}(\rho)$ and $(\rho(x)|\tau) \in |T_\rho|_{V_t}$

$\lrcorner$

We recall two key properties of the interpretation, whose proofs are similar to the proofs for the corresponding statement in the $\overline{\lambda}_{[lv\tau\star]}$-calculus (Lemma 6.16 and Proposition 6.14):

Lemma 8.19 (Store weakening). *Let τ and τ' be two stores such that $\tau \lhd \tau'$, let Γ be a typing context, let $\perp\!\!\!\perp$ be a pole and ρ a valuation. The following statements hold:*

1. $\overline{\tau\tau'} = \tau'$
2. *If $(p|\tau)_\rho \in |A_\rho|_p$ for some closed proof-in-store $(p|\tau)_\rho$ and formula A, then $(p|\tau')_\rho \in |A_\rho|_p$. The same holds for each level e, E, V, f, t, π, V_t of the interpretation.*
3. *If $(\tau,\rho) \Vdash \Gamma$ then $(\tau',\rho) \Vdash \Gamma$.*

Proposition 8.20 (Monotonicity). *For any closed formula A, any type T and any given pole $\perp\!\!\!\perp$, we have the following inclusions:*

$$|A|_V \subseteq |A|_p \qquad\qquad \|A\|_f \subseteq \|A\|_e \qquad\qquad |T|_{V_t} \subseteq |T|_t$$

Finally we can check that the interpretation is indeed defined up to the relations $\equiv_\tau$:

Lemma 8.21. *For any store τ and any valuation ρ, the component along τ of the truth and falsity values defined in Figure 8.8 are closed under the relation $\equiv_\tau$:*

1. *if $(f|\tau)_\rho \in \|A_\rho\|_f$ and $A_\rho \equiv_\tau B_\rho$, then $(f|\tau)_\rho \in \|B_\rho\|_f$,*
2. *if $(V_t|\tau)_\rho \in |A_\rho|_{V_t}$ and $A_\rho \equiv_\tau B_\rho$, then $(V_t|\tau)_\rho \in |B_\rho|_v$.*

The same applies with $|A_\rho|_p$, $\|A_\rho\|_e$, etc.

Proof. By induction on the structure of A_ρ and the different levels of interpretation. The different base cases ($p \in A_\rho$, $t \in T$, $t = u$) are direct since their components along τ are defined modulo $\equiv_\tau$, the other cases are trivial inductions. □

Proposition 8.22 (Adequacy). *The typing rules are adequate with respect to the realizability interpretation. In other words, if Γ is a typing context, $\perp\!\!\!\perp$ a pole, ρ a valuation and τ a store such that $(\tau,\rho) \Vdash \Gamma;\sigma$, then the following hold:*

1. *If v is a strong value such that $\Gamma \vdash^\sigma v : A$ or $\Gamma \vdash_d v : A;\sigma$, then $(v|\tau)_\rho \in |A_\rho|_V$.*

[9] Alternatively, we could have modified the small-step reduction rules to include substitutions of terms.

[10] Once again, we should formally write $(\tau,\rho) \Vdash_{\perp\!\!\!\perp} \Gamma$ but we will omit the annotation by $\perp\!\!\!\perp$ as often as possible.

2. *If f is a forcing context such that $\Gamma \vdash^{\sigma} f : A^{\perp\!\!\!\perp}$ or $\Gamma \vdash_d f : A^{\perp\!\!\!\perp}; \sigma$, then $(f|\tau)_\rho \in \|A_\rho\|_f$.*
3. *If V is a weak value such that $\Gamma \vdash^{\sigma} V : A$ or $\Gamma \vdash_d V : A; \sigma$, then $(V|\tau)_\rho \in |A_\rho|_V$.*
4. *If e is a context such that $\Gamma \vdash^{\sigma} e : A^{\perp\!\!\!\perp}$ or $\Gamma \vdash_d e : A^{\perp\!\!\!\perp}; \sigma$, then $(e|\tau)_\rho \in \|A_\rho\|_e$.*
5. *If p is a proof term such that $\Gamma \vdash^{\sigma} p : A$ or $\Gamma \vdash_d p : A; \sigma$, then $(p|\tau)_\rho \in |A_\rho|_p$.*
6. *If V_t is a term value such that $\Gamma \vdash^{\sigma} V_t : T$, then $(V_t|\tau)_\rho \in |T_\rho|_{V_t}$.*
7. *If π is a term context such that $\Gamma \vdash^{\sigma} \pi : T$, then $(\pi|\tau)_\rho \in |T_\rho|_\pi$.*
8. *If t is a term such that $\Gamma \vdash^{\sigma} t : T$, then $(t|\tau)_\rho \in |T_\rho|_t$.*
9. *If τ' is a store such that $\Gamma \vdash^{\sigma} \tau' : (\Gamma';)\sigma'$, then $(\tau\tau', \rho) \Vdash (\Gamma, \Gamma'; \sigma\sigma')$.*
10. *If c is a command such that $\Gamma \vdash^{\sigma} c$ or $\Gamma \vdash_d c; \sigma$, then $(c\tau)_\rho \in \perp\!\!\!\perp$.*
11. *If $c\tau'$ is a closure such that $\Gamma \vdash^{\sigma} c\tau'$ or $\Gamma \vdash_d c\tau'; \sigma$, then $(c\tau\tau')_\rho \in \perp\!\!\!\perp$.*

Proof. The proof is done by induction on the typing derivation such as given in the system extended with the small-step reduction $\rightsquigarrow_s$. Most of the cases correspond to the proof of adequacy for the interpretation of the $\overline{\lambda}_{[lv\tau\star]}$-calculus, so that we only give the most interesting cases. To lighten the notations, we omit the annotation by the valuation ρ whenever it is possible.

- **Case** $(\exists_r)$. We recall the typing rule through the decomposition of dependent sums:

$$\frac{\Gamma \vdash^{\sigma} t : u \in T \qquad \Gamma \vdash^{\sigma} p : A[u/x]}{\Gamma \vdash^{\sigma} (t,p) : (u \in T \wedge A[u])}$$

By induction hypothesis, we obtain that $(t|\tau) \in |u \in T|_t$ and $(p|\tau) \in |A[u]|_p$. Consider thus any context-in-store $(e|\tau') \in \|u \in T \wedge A[u]\|_e$ such that τ and τ' are compatible, and let us denote by τ_0 the union $\overline{\tau\tau'}$. We have:

$$\langle (t,p) \| e \rangle_p \tau_0 \rightsquigarrow_s \langle p \| \tilde{\mu}\check{\mathrm{tp}}.\langle t \| \tilde{\mu}x.\langle \check{\mathrm{tp}} \| \tilde{\mu}a.\langle (x,a) \| e \rangle\rangle\rangle\rangle_p \tau_0$$

so that by anti-reduction, we need to show that $\tilde{\mu}\check{\mathrm{tp}}.\langle t \| \tilde{\mu}x.\langle \check{\mathrm{tp}} \| \tilde{\mu}a.\langle (x,a) \| e \rangle\rangle\rangle \in \|A[u]\|_e$. Let us then consider a value-in-store $(V|\tau_0') \in |A[u]|_V$ such that τ_0 and τ_0' are compatible, and let us denote by τ_1 the union $\overline{\tau_0\tau_0'}$. By closure under delimited continuations, to show that $\langle V \| \tilde{\mu}\check{\mathrm{tp}}.\langle t \| \tilde{\mu}x.\langle \check{\mathrm{tp}} \| \tilde{\mu}a.\langle (x,a) \| e \rangle\rangle\rangle\rangle_p \tau_1$ is in the pole it is enough to show that the closure $\langle t \| \tilde{\mu}x.\langle V \| \tilde{\mu}a.\langle (x,a) \| e \rangle\rangle\rangle \tau_1$ is in $\perp\!\!\!\perp$,. Thus it suffices to show that the coterm-in-store $(\tilde{\mu}x.\langle V \| \tilde{\mu}a.\langle (x,a) \| e \rangle\rangle | \tau_1)$ is in $|u \in T|_\pi$.

Consider a term value-in-store $(V_t|\tau_1') \in |u \in T|_{V_t}$, such that τ_1 and τ_1' are compatible, and let us denote by τ_2 the union $\overline{\tau_1\tau_1'}$. We have:

$$\langle V_t \| \tilde{\mu}x.\langle V \| \tilde{\mu}a.\langle (x,a) \| e \rangle\rangle\rangle \tau_2 \rightsquigarrow_s \langle V \| \tilde{\mu}a.\langle (V_t,a) \| e \rangle\rangle \tau_2 \rightsquigarrow_s \langle (V_t,a) \| e \rangle \tau_2[a := V]$$

It is now easy to check that $((V_t,a)|\tau_2[a := V]) \in |u \in T \wedge A[u]|_V$ and to conclude, using Lemma 8.19 to get $(e|\tau_2[a := V]) \in \|u \in T \wedge A[u]\|_e$, that this closure is finally in the pole.

- **Case** $(\equiv_r)$,$(\equiv_l)$. These cases are direct consequences of Lemma 8.21 since if A, B are two formulas such that $A \equiv B$, in particular $A \equiv_\tau B$ and thus $|A|_v = |B|_v$.

- **Case** (`refl`),$(=_l)$. The case for `refl` is trivial, while it is trivial to show that $(\tilde{\mu}\!=\!.\langle p \| e \rangle | \tau)$ is in $\|t = u\|_f$ if $(p|\tau) \in |A[t]|_p$ and $(e|\tau) \in \|A[u]\|_e$. Indeed, either $t \equiv_\tau u$ and thus $A[t] \equiv_\tau A[u]$ (Lemma 8.21, or $t \not\equiv_\tau u$ and $\|t = u\|_f = \Lambda_f^\tau$.

- **Case** $(\forall_r^x)$. This case is standard in a call-by-value language with value restriction. We recall the typing rule:

$$\frac{\Gamma \vdash^\sigma v : A \quad x \notin FV(\Gamma)}{\Gamma \vdash^\sigma v : \forall x.A}\ (\forall_r^x)$$

The induction hypothesis gives us that $(v|\tau)_\rho$ is in $|A_\rho|_V$ for any valuation $\rho[x \mapsto t]$. Then for any t, we have $(v|\tau)_\rho \in \|A_\rho[t/x]\|_f^{\perp\!\!\perp_v}$ so that $(v|\tau)_\rho \in (\bigcap_{t\in\Lambda_t} \|A[t/x]\|_f^{\perp\!\!\perp_v})$. Therefore if $(f|\tau')_\rho$ belongs to $\|\forall x.A_\rho\|_f = (\bigcap_{t\in\Lambda_t} \|A[t/x]\|_f^{\perp\!\!\perp_v})^{\perp\!\!\perp_f}$, we have by definition that $(v|\tau)_\rho \perp\!\!\perp (f|\tau')_\rho$.

- **Case** $(\mathtt{ind})$. We recall the typing rule:

$$\frac{\Gamma \vdash^\sigma t : \mathbb{N} \quad \Gamma \vdash^\sigma p_0 : A[0/x] \quad \Gamma, x : T, a : A \vdash^\sigma p_S : A[S(x)/x]}{\Gamma \vdash^\sigma \mathtt{ind}_{ax}^t[p_0 \,|\, p_S] : A[t/x]}\ (\mathtt{ind})$$

We want to show that $(\mathtt{ind}_{ax}^t[p_0 \,|\, p_S]|\tau) \in |A[t]|_p$, let us then consider $(e|\tau') \in \|A[t]\|_e$ such that τ and τ' are compatible, and let us denote by τ_0 the union $\overline{\tau\tau'}$. By induction hypothesis, we have[11] $t \in |t \in \mathbb{N}|_t$ and we have:

$$\langle \mathtt{ind}_{bx}^t[p_0 \,|\, p_S] \| e\rangle_p \tau_0 \rightsquigarrow_s \langle \mu\hat{\mathfrak{tp}}.\langle t \| \tilde{\mu} y.\langle a \| \hat{\mathfrak{tp}}\rangle[a := \mathtt{ind}_{bx}^y[p_0 \,|\, p_S]]\rangle \| e\rangle_p \tau_0$$

so that by anti-reduction and closure under delimited continuations, it is enough to show that the coterm-in-store $(\tilde{\mu} y.\langle a\|e\rangle[a := \mathtt{ind}_{bx}^y[p_0 \,|\, p_S]]|\tau_0)$ is in $|t \in \mathbb{N}|_\pi$. Let us then consider $(V_t|\tau_0') \in |t \in \mathbb{N}|_{V_t}$ such that τ_0 and τ_0' are compatible, and let us denote by τ_1 the union $\overline{\tau_0\tau_0'}$. By definition, $V_t = S^n(0)$ for some $n \in \mathbb{N}$ and $t \equiv_{\tau_1} S^n(0)$, and we have:

$$\langle S^n(0) \| \tilde{\mu} y.\langle a\|e\rangle[a := \mathtt{ind}_{bx}^y[p_0 \,|\, p_S]]\rangle \tau_1 \rightsquigarrow_s \langle a\|e\rangle \tau_1[a := \mathtt{ind}_{bx}^{S^n(0)}[p_0 \,|\, p_S]]$$

We conclude by showing by induction on the natural numbers that for any $n \in N$, the value-in-store $(a|\tau_1[a := \mathtt{ind}_{bx}^{S^n(0)}[p_0 \,|\, p_S]])$ is in $|A[S^n(0)]|_V$. Let us consider $(f|\tau_1') \in \|A[S^n(0)]\|_f$ such that the store $\tau_1[a := \mathtt{ind}_{bx}^{S^n(0)}[p_0 \,|\, p_S]]$ and τ_1' are compatible, and let us denote by $\tau_2[a := \mathtt{ind}_{bx}^{S^n(0)}[p_0 \,|\, p_S]]\tau_2'$ their union.

- If $n = 0$, we have:

  $$\langle a\|f\rangle \tau_2[a := \mathtt{ind}_{bx}^0[p_0 \,|\, p_S]]\tau_2' \rightsquigarrow_s \langle p_0 \| \tilde{\mu} a.\langle a\|f\rangle \tau_2'\rangle \tau_2$$

  We conclude by anti-reduction and the induction hypothesis for p_0, since it is easy to show that $(\tilde{\mu} a.\langle a\|f\rangle\tau_2'|\tau_2) \in \|A[0]\|_e$.

- If $n = S(m)$, we have:

  $$\langle a\|f\rangle \tau_2[a := \mathtt{ind}_{bx}^{S(S^m(0))}[p_0 \,|\, p_S]]\tau_2' \rightsquigarrow_s \langle p_S[S^m(0)/x][b'/b] \| \tilde{\mu} a.\langle a\|f\rangle\tau_2'\rangle_p \tau_2[b' := \mathtt{ind}_{bx}^{S^m(0)}[p_0 \,|\, p_S]]$$

  Since we have by induction that $(b'|\tau_2[b' := \mathtt{ind}_{bx}^{S^m(0)}[p_0 \,|\, p_S]])$ is in $|A[S^m(0)]|_V$, we can conclude by anti-reduction, using the induction hypothesis for p_S and the fact that $(\tilde{\mu} a.\langle a\|f\rangle\tau_2'|\tau_2)$ belongs to $\|A[S(S^m(0))]\|_e$.

[11] Recall that any term t of type T can be given the type $t \in T$.

- **Case** (`cofix`). We recall the typing rule:

$$\frac{\Gamma \vdash^{\sigma} t : T \quad \Gamma, x : T, b : \forall y^{T}.X(y) \vdash^{\sigma} p : A \quad X \text{ positive in } A \quad X \notin FV(\Gamma)}{\Gamma \vdash^{\sigma} \mathtt{cofix}_{bx}^{t}[p] : \nu_{Xx}^{t} A}\ (\mathtt{cofix})$$

We want to show that $(\mathtt{cofix}_{bx}^{t}[p]|\tau) \in |\nu_{Xx}^{t}A|_p$, let us then consider $(e|\tau') \in \|\nu_{Xx}^{t}A\|_e$ such that τ and τ' are compatible, and let us denote by τ_0 the union $\overline{\tau\tau'}$. By induction hypothesis, we have $t \in |t \in T|_t$ and we have:

$$\langle \mathtt{cofix}_{bx}^{t}[p] \| e \rangle_p \tau_0 \rightsquigarrow_s \langle \mu \hat{\mathfrak{tp}}.\langle t \| \tilde{\mu} y.\langle a \| \hat{\mathfrak{tp}} \rangle [a := \mathtt{cofix}_{bx}^{y}[p]] \rangle \| e \rangle_p \tau_0$$

so that by anti-reduction and closure under delimited continuations, it is enough to show that the coterm-in-store $(\tilde{\mu} y.\langle a \| e \rangle [a := \mathtt{cofix}_{bx}^{y}[p]] | \tau_0)$ is in $|t \in \mathbb{N}|_{\pi}$. Let us then consider $(V_t|\tau_0') \in |t \in T|_{V_t}$ such that τ_0 and τ_0' are compatible, and let us denote by τ_2 the union $\overline{\tau_0\tau_0'}$. We have:

$$\langle V_t \| \tilde{\mu} y.\langle a \| e \rangle [a := \mathtt{cofix}_{bx}^{y}[p]] \rangle \tau_1 \rightsquigarrow_s \langle a \| e \rangle \tau_1 [a := \mathtt{cofix}_{bx}^{V_t}[p]]$$

It suffices to show now that the value-in store $(a|\tau_1[a := \mathtt{cofix}_{bx}^{V_t}[p]])$ is in $|\nu_{Xx}^{V_t}A|_V$. By definition, we have:

$$|\nu_{Xx}^{V_t}A|_V = (\bigcup_{n \in \mathbb{N}} \|F_{A,V_t}^{n}\|_f)^{\perp\!\!\!\perp_V} = \bigcap_{n \in \mathbb{N}} \|F_{A,V_t}^{n}\|_f^{\perp\!\!\!\perp_V} = \bigcap_{n \in \mathbb{N}} |F_{A,V_t}^{n}|_V$$

We conclude by showing by induction on the natural numbers that for any $n \in N$ and any V_t, the value-in-store $(a|\tau_1[a := \mathtt{cofix}_{bx}^{V_t}[p]])$ is in $|F_{A,V_t}^{n}|_V$.

The case $n = 0$ is trivial since $|F_{A,V_t}^{0}|_V = |\top|_V = \Lambda_V^{\tau}$. Let then n be an integer and any V_t be a term value. Let us consider $(f|\tau_1') \in \|F_{A,V_t}^{n+1}A\|_f$ such that $\tau_1[a := \mathtt{cofix}_{bx}^{V_t}[p]]$ and τ_1' are compatible, and let us denote by $\tau_2[a := \mathtt{cofix}_{bx}^{V_t}[p]]\tau_2'$ their union. By definition, we have:

$$\langle a \| f \rangle \tau_2 [a := \mathtt{cofix}_{bx}^{V_t}[p]] \tau_2' \rightsquigarrow_s \langle p[V_t/x][b'/b] \| \tilde{\mu} a.\langle a \| f \rangle \tau_2' \rangle \tau_2 [b' := \lambda y.\mathtt{cofix}_{bx}^{y}[p]]$$

It is straightforward to check, using the induction hypothesis for n, that $(b'|\tau_2[b' := \lambda y.\mathtt{cofix}_{bx}^{y}[p]])$ is in $|\forall y.y \in T \to F_{A,y}^{n}|_V$. Thus we deduce by induction hypothesis for p, denoting by S the function $t \mapsto \|F_{A,t}^{n}\|_f$, that:

$$(p[V_t/x][b'/b]|\tau_2[b' := \lambda y.\mathtt{cofix}_{bx}^{y}[p]]) \in |A[V_t/x][\dot{S}/X]|_p = |A[V_t/x][F_{A,y}^{n}/X(y)]|_p = |F_{A,V_t}^{n+1}|_p$$

It only remains to show that $(\tilde{\mu} a.\langle a \| f \rangle \tau_2' | \tau_2) \in \|F_{A,V_t}^{n+1}\|_e$, which is trival from the hypothesis for f. □

We can finally deduce from Propositions 8.14 and 8.22 that dLPA$^{\omega}$ is normalizable and sound.

Theorem 8.23 (Normalization). *If $\Gamma \vdash^{\sigma} c$, then c is normalizable.*

Theorem 8.24 (Consistency). $\nvdash_{dLPA^{\omega}} p : \bot$

Proof. Assume there is such a proof p, by adequacy $(p|\varepsilon)$ is in $|\bot|_p$ for any pole. Yet, the set $\perp\!\!\!\perp \triangleq \emptyset$ is a valid pole, and with this pole, $|\bot|_p = \emptyset$, which is absurd. □

8.4.2 About the interpretation of coinductive formulas

While our realizability interpretation finally gave us a proof of normalization and soundness for dLPA$^{\omega}$, it has two aspects that we could find unsatisfactory. First, regarding the small-step reduction system, one could have expected the lowest level of interpretation to be v instead of f. Moreover, if we observe our definition, we notice that most of the cases of $\|\cdot\|_f$ are in fact defined by orthogonality to a subset of strong values. Indeed, except for coinductive formulas, we could indeed have defined instead an interpretation $|\cdot|_v$ of formulas at level v and then the interpretation $\|\cdot\|_f$ by orthogonality:

$$\begin{array}{rcl}
|\bot|_v & \triangleq & \emptyset \\
|t = u|_v & \triangleq & \begin{cases} \texttt{refl} & \text{if } t \equiv u \\ \emptyset & \text{otherwise} \end{cases} \\
|p \in A|_v & \triangleq & \{(v|\tau) \in |A|_v : v \equiv_\tau p\} \\
|T \to B|_v & \triangleq & \{(\lambda x.p|\tau) : \forall V_t \tau', \tau \diamond \tau' \wedge (V_t|\tau') \in |T|_V \Rightarrow (p[V_t/x]|\overline{\tau\tau'}) \in |B|_p\} \\
|A \to B|_v & \triangleq & \{(\lambda a.p|\tau) : \forall V \tau', \tau \diamond \tau' \wedge (V|\tau') \in |A|_V \Rightarrow (p|\overline{\tau\tau'}[a := V]) \in |B|_p\} \\
|T \wedge A|_v & \triangleq & \{((V_t, V)|\tau) : (V_t|\tau) \in |T|_{V_t} \wedge (V|\tau) \in |A_2|_V\} \\
|A_1 \wedge A_2|_v & \triangleq & \{((V_1, V_2)|\tau) : (V_1|\tau) \in |A_1|_V \wedge (V_2|\tau) \in |A_2|_V\} \\
|A_1 \vee A_2|_v & \triangleq & \{(\iota_i(V)|\tau) : (V|\tau) \in |A_i|_V\} \\
|\exists x.A|_v & \triangleq & \bigcup_{t\in\Lambda_t} |A[t/x]|_v \\
|\forall x.A|_v & \triangleq & \bigcap_{t\in\Lambda_t} |A[t/x]|_v \\
|\forall a.A|_v & \triangleq & \bigcap_{p\in\Lambda_p} |A[p/x]|_v \\
\|A\|_f & \triangleq & \{(f|\tau) : \forall v \tau', \tau \diamond \tau' \wedge (v|\tau') \in |A|_v \Rightarrow (v|\tau') \perp\!\!\!\perp (F|\tau)\}
\end{array}$$

If this definition is somewhat more natural, it poses a problem for the definition of coinductive formulas. Indeed, there is a priori no strong value in the orthogonal of $\|\nu^t_{fv} A\|_f$, which is:

$$(\|\nu^t_{fv} A\|_f)^{\perp\!\!\!\perp_v} = (\bigcup_{n\in\mathbb{N}} \|F^n_{A,t}\|_f)^{\perp\!\!\!\perp_v} = \bigcap_{n\in\mathbb{N}} (\|F^n_{A,t}\|_f)^{\perp\!\!\!\perp_v})$$

For instance, consider again the case of a stream of type $\nu^0_{fx} A(x) \wedge f(S(x)) = 0$, a strong value in the intersection should be in every $|A(0) \wedge (A(1) \wedge \ldots (A(n) \wedge \top) \ldots)|_v$, which is not possible due to the finiteness of terms[12] Thus the definition $|\nu^t_{fv} A|_v \triangleq \bigcap_{n\in\mathbb{N}} |F^n_{A,t}|_v$ would give $|\nu^t_{fx} A|_v = \emptyset = |\bot|_v$.

Interestingly, and this is the second aspect that we do not find completely satisfactory, we could have define instead the truth value of coinductive formulas directly by :

$$|\nu^t_{fx} A|_v \triangleq |A[t/x][\nu^y_{fx} A / f(y) = 0]|_v$$

Let us sketch the proof that such a definition is well-founded. We consider the language of formulas without coinductive formulas and extended with formulas of the shape $X(t)$ where $X, Y, \ldots$ are parameters. At level v, closed formulas are interpreted by sets of strong values-in-store $(v|\tau)$, and as we already observed, these sets are besides closed under the relation $\equiv_\tau$ along their component τ. If $A(x)$ is a formula whose only free variable is x, the function which associates to each term t the set $|A(t)|_v$ is thus a function from Λ_t to $\mathcal{P}(\Lambda^\tau_v)_{\equiv_\tau}$, let us denote the set of these functions by $\mathscr{L}$.

Proposition 8.25. *The set $\mathscr{L}$ is a complete lattice with respect to the order $\leq_{\mathscr{L}}$ defined by:*

$$F \leq_{\mathscr{L}} G \triangleq \forall t \in \Lambda_t . F(t) \subseteq G(t)$$

Proof. Trivial since the order on functions is defined pointwise and the co-domain $\mathcal{P}(\Lambda^\tau_v)$ is itself a complete lattice. □

[12] Yet, it might possible to consider interpretation with infinite proof terms, the proof of adequacy for proofs and contexts (which are finite) will still work exactly the same. However, another problem will arise for the adequacy of the `cofix` operator. Indeed, with the interpretation above, we would obtain the inclusion $\bigcup_{n\in\mathbb{N}}(\|F^n_{A,t}\|_f) \subset (\bigcap_{n\in\mathbb{N}} |F^n_{A,t}|_t)^{\perp\!\!\!\perp_f} = \|\nu^t_{fx} A\|_f$ which is strict in general. By orthogonality, this gives us that $|\nu^t_{fx} A|_V \subseteq \bigcup_{n\in\mathbb{N}}(\|F^n_{A,t}\|_f))^{\perp\!\!\!\perp_V}$, while the proof of adequacy only proves that $(a|\tau[a := \texttt{cofix}^t_b[x]p])$ belongs to the latter set.

We define valuations, which we write ρ, as functions mapping each parameter X to a function $\rho(X) \in \mathscr{L}$. We then define the interpretations $|A|_v^\rho, \|A\|_f^\rho, \ldots$ of formulas with parameters exactly as above with the additional rule[13]:

$$|X(t)|_v^\rho \triangleq \{(v|\tau) \in \rho(X)(t)\}$$

Let us fix a formula A which has one free variable x and a parameter X such that sub-formulas of the shape $X\,t$ only occur in positive positions in A.

Lemma 8.26. *Let $B(x)$ is a formula without parameters whose only free variable is x, and let ρ be a valuation which maps X to the function $t \mapsto |B(t)|_v$. Then $|A|_v^\rho = |A[B(t)/X(t)]|_v$*

Proof. By induction on the structure of A, all cases are trivial, and this is true for the basic case $A \equiv X(t)$:

$$|X(t)|_v^\rho = \rho(X)(t) = |B(t)|_v$$

□

Let us now define φ_A as the following function:

$$\varphi_A : \left\{ \begin{array}{ccc} \mathscr{L} & \to & \mathscr{L} \\ F & \mapsto & t \mapsto |A[t/x]|_v^{[X\mapsto F]} \end{array} \right.$$

Proposition 8.27. *The function φ_A is monotone.*

Proof. By induction on the structure of A, where X can only occur in positive positions. The case $|X(t)|_v$ is trivial, and it is easy to check that truth values are monotonic with respect to the interpretation of formulas in positive positions, while falsity values are anti-monotonic. □

We can thus apply Knaster-Tarski theorem to φ_A, and we denote by $\mathtt{gfp}(\varphi_A)$ its greatest fixpoint. We can now define:

$$|\nu^t_{Xx} A|_v \triangleq \mathtt{gfp}(\varphi_A)(t)$$

This definition satisfies the expected equality:

Proposition 8.28. *We have:*

$$|\nu^t_{Xx} A|_v = |A[t/x][\nu^y_{Xx} A/X(y)]|_v$$

Proof. Observe first that by definition, the formula $B(z) = |\nu^z_{Xx} A|_v$ satisfies the hypotheses of Lemma 8.26 and that $\mathtt{gfp}(\varphi_A) = t \mapsto B(t)$. Then we can deduce :

$$|\nu^t_{Xx} A|_v = \mathtt{gfp}(\varphi_A)(t) = \varphi_A(\mathtt{gfp}(\varphi_A))(t) = |A[t/x]|_v^{[X\mapsto \mathtt{gfp}(\varphi_A)]} = |A[t/x][\nu^y_{Xx} A/X(y)]|_v$$

□

Back to the original language, it only remains to define $|\nu^t_{fx} A|_v$ as the set $|\nu^t_{Xx} A[X(y)/f(y) = 0]|_v$ that we just defined. This concludes our proof that the interpretation of coinductive formulas through the equation in Proposition 8.28 is well-founded.

We could also have done the same reasoning with the interpretation from the previous section, by defining $\mathscr{L}$ as the set of functions from Λ_t to $\mathcal{P}(\Lambda_f^\tau)_{\equiv_\tau}$. The function φ_A, which is again monotonic, is then:

$$\varphi_A : \left\{ \begin{array}{ccc} \mathscr{L} & \to & \mathscr{L} \\ F & \mapsto & t \mapsto |A[t/x]|_v^{[X\mapsto F]} \end{array} \right.$$

[13] Observe that this rule is exactly the same as in the previous section (see Figure 8.8).

We recognize here the definition of the formula $F^n_{A,t}$. Defining f^0 as the function $t \mapsto \|\top\|_f$ and $f^{n+1} \triangleq \varphi_A(f^n)$ we have:

$$\forall n \in \mathbb{N}, \|F^n_{A,t}\|_f = f^n(t) = \varphi^n_A(f^0)(t)$$

However, in both cases (defining primitively the interpretation at level v or f), this definition does not allow us to prove[14] the adequacy of the (cofix) rule. In the case of an interpretation defined at level f, the best that we can do is to show that for any $n \in \mathbb{N}$, f^n is a post-fixpoint since for any term t, we have:

$$f^n(t) = \|F^n_{A,t}\|_f \subseteq \|F^{n+1}_{A,t}\|_f = f^{n+1}(t) = \varphi_A(f^n)(t)$$

With $\|\nu^t_{fx}A\|_f$ defined as the greatest fixpoint of φ_A, for any term t and any $n \in \mathbb{N}$ we have the inclusion $f^n(t) \subseteq \mathtt{gfp}(\varphi_A)(t) = \|\nu^t_{fx}A\|_f$ and thus:

$$\bigcup_{n\in\mathbb{N}} \|F^n_{A,t}\|_f = \bigcup_{n\in\mathbb{N}} f^n(t) \subseteq \|\nu^t_{fx}A\|_f$$

By orthogonality, we get:

$$|\nu^t_{fx}A|_V \subseteq \bigcap_{n\in\mathbb{N}} |F^n_{A,t}|_V$$

and thus our proof of adequacy from the last section is not enough to conclude that $\mathtt{cofix}^t_{bx}[p] \in |\nu^t_{fx}A|_p$. For this, we would need to prove that the inclusion is an equality. An alternative to this would be to show that the function $t \mapsto \bigcup_{n\in\mathbb{N}} \|F^n_{A,t}\|_f$ is a fixpoint for φ_A. In that case, we could stick to this definition and happily conclude that it satisfies the equation:

$$\|\nu^t_{Xx}A\|_f = \|A[t/x][\nu^y_{Xx}A/X(y)]\|_f$$

This would be the case if the function φ_A was Scott-continuous on $\mathscr{L}$ (which is a dcpo), since we could then apply Kleene fixed-point theorem[15] to prove that $t \mapsto \bigcup_{n\in\mathbb{N}} \|F^n_{A,t}\|_f$ is the stationary limit of $\varphi^n_A(f_0)$. However, φ_A is not Scott-continuous[16] (the definition of falsity values involves double-orthogonal sets which do not preserve supremums), and this does not apply.

8.5 Conclusion and perspectives

Recap of the journey In the end, we met our main objective, namely proving the soundness and the normalization of a language which includes proof terms for dependent and countable choice in a classical setting. This language, which we called dLPA$^\omega$, provides us with the same computational features as dPA$^\omega$ but in a sequent-calculus fashion. The calculus indeed includes co-fixpoint operators, which are lazily evaluated. To this end, dLPA$^\omega$ uses the design of the $\overline{\lambda}_{[lv\tau\star]}$-calculus of Ariola *et al.* [4], which we equipped in Chapter 6 with a type system and which we proved to be normalizing. In addition, the proof terms of dLPA$^\omega$ are dependently typed thanks to a restriction of dependencies to the negative-elimination-free fragment which makes them compatible with classical logic. These computational features allows dLPA$^\omega$ to internalize the realizability approach of [15, 40] as a direct proofs-as-programs interpretation: both proof terms for countable and dependent choices furnish a

[14] To be honest, we should rather say that we could not manage to find a proof, and that we would welcome any suggestion from insightful readers.

[15] In fact, Cousot and Cousot proved a constructive version of Kleene fixed-point theorem which states that without any continuity requirement, the transfinite sequence $(\varphi^\alpha_A(f^0))_{\alpha\in O_n}$ is stationary [30]. Yet, we doubt that the gain of the desired equality is worth a transfinite definition of the realizability interpretation.

[16] In fact, this is nonetheless a good news about our interpretation. Indeed, it is well-know that the more "regular" a model is, the less interesting it is. For instance, Streicher showed that the realizability model induced by Scott domains (using it as a realizability structure) was not only a forcing model by also equivalent to the ground model.

lazy witness for the ideal choice function which is evaluated on demand. At the risk of repeating ourself, this interpretation is in line with the slogan that with new programing principles—here the lazy evaluation and the co-inductive objects—come new reasoning principles—here the axioms $AC_{\mathbb{N}}$ and DC.

In our search for a proof of normalization for dLPA$^{\omega}$, we developed novel tools to study these side-effects and dependent types in presence of classical logic. On the one hand, we set out in Chapter 7 the difficulties related to the definition of a sequent calculus with dependent types. We proposed a formalism, dL$_{\hat{\text{tp}}}$, which restricts the dependencies to proofs in the NEF fragment together. This restriction is strengthened with an evaluation of dependently typed computations within delimited continuations; while the type system is enriched with an explicit list of dependencies. This provides us with a calculus whose reduction is safe, and which has the advantage of being suitable for a typed continuation-passing style translation carrying the dependencies.

On the other hand, we defined a typed continuation-and-store passing style translation for the $\overline{\lambda}_{[lv\tau\star]}$-calculus, which we related to Kripke forcing semantics. Besides, we saw how to handle laziness and explicit stores in a realizability interpretation *à la* Krivine. This might be a first step toward new interpretations of different axioms using laziness within Krivine classical realizability. In a long term perspective, it would be interesting to understand the impact of laziness and stores on the corresponding realizability algebras. More generally, the algebraic models that we study in the last part of this thesis are oriented toward the call-by-name and the call-by-value evaluation strategies. While it is probably the case that these structures could be modified to obtain call-by-need algebras, the structure of such algebras is not obvious *a priori*.

Comparison with Krivine's interpretations of dependent choice At the end of the day, we presented a calculus, dLPA$^{\omega}$, with the nice property of allowing for the direct definition of a proof term for the axiom of dependent choice. Beside, we prove the normalization and soundness of dLPA$^{\omega}$ by means of a realizability interpretation *à la* Krivine. Yet, the computational content we give to the axiom of dependent choice is pretty different of Krivine's usual realizer of the same [94]. Indeed, our proof uses dependent types to get witnesses of existential formulas, and represents the choice function through the lazily evaluated stream of its values. In turn, Krivine realizes a statement which is logically equivalent to the axiom of dependent choice thanks to the instruction quote, which injectively associates a natural number to each closed λ_c-term. In particular, such an instruction allows to compare the codes of two programs, so that terms of the λ_c-calculus extended with quote can reduce differently according to the code of the arguments they are given. In a more recent work [102], Krivine proposes a realizability model which has a bar-recursor and where the axiom of dependent choice is realized using the bar-recursion. This realizability model satisfies the continuum hypothesis and many more properties, in particular the real numbers have the same properties as in the ground model. However, the very structure of this model, where Λ is of cardinal $\aleph_1$ (in particular infinite streams of integer are terms), makes it incompatible with the instruction quote.

It is clear that the three approaches are different in terms of programming languages. Nonetheless, it could be interesting to compare them from the point of view of the realizability models they give rise to. We did not study at all this question for dLPA$^{\omega}$, especially we do not know whether it is suitable to define the same model of ZF + ¬ AC + ¬ CH (set theory with the negation of the axiom of choice and the negation of continuum hypothesis). Neither do we know if the induced model is compatible with the quote instruction (we conjecture that it is). It might be the case that our approach can be related with the one with a bar-recursor in [102]. In particular, our analysis of the interpretation of co-inductive formulas may suggest that the interest of lazy co-fixpoints is precisely to approximate the limit situation where Λ has infinite objects.

The study of the structures of Krivine realizability models is already a hard question, and so is in general the problem of determining the consequences that a particular set of instructions or a specific pole might have on on the model[17]. Nonetheless, the question still holds.

Reduction of the consistency of classical arithmetic in finite types with dependent choice to the consistency of second-order arithmetic The standard approach to the computational content of classical dependent choice in the classical arithmetic in finite types is via realizability as initiated by Spector [150] in the context of Gödel's functional interpretation, and later adapted to the context of modified realizability by Berardi *et al* [15]. In the different settings of second-order arithmetic [97] and classical realizability, Krivine [94] gives a realization of a formulation of dependent choice over sets of numbers using side-effects (a clock or a quote operator).

In all these approaches, the correctness of the realizer, which implies consistency of the system, is itself justified by a use at the meta-level of a principle classically equivalent to dependent choice (dependent choice itself in Krivine, bar induction or update induction [16] in the case of Spector or Berardi *et al*).

Our approach is here different. Not only we directly interpret proofs of dependent choice in classical arithmetic computationally but we propose a path to a computational reduction of the consistency of classical arithmetic in finite types (PA^ω) to the one of the target language F_Υ. This system is an extension of system F, but it is not clear whether its consistency is conservative of not over system F. Ultimately, we would be interested in a computational reduction of the consistency of dPA^ω or dLPA^ω to the one of PA2, that is to the consistency of second-order arithmetic. While it is well-known that DC is conservative over second-order arithmetic with full comprehension (see [149, Theorem VII.6.20]), it would nevertheless be very interesting to have such a direct computational reduction. The converse direction has been recently studied by Valentin Blot, who presented in [18] a translation of System F into a simply-typed total language with a variant of bar recursion.

[17] To quote the last PhD student in date who attempted to define purpose-oriented realizability models [2], they are like Forrest's Gump chocolates boxes, *"you never know what you're gonna get"*.

Part III

Algebraic models of classical realizability

9- Algebraization of realizability

In the first parts of this thesis, we introduced several calculi for which we gave a Krivine realizability interpretation. Namely, in addition to Krivine's λ_c-calculus, we presented interpretations for the call-by-name, call-by-value and call-by-need $\lambda\mu\tilde{\mu}$-calculi, for $\mathrm{dL}_{\hat{\mathfrak{tp}}}$ and for dLPA^{ω}. Amongst others, we could cite Munch-Maccagnoni's interpretation for System L [126], Lepigre's interpretation for his call-by-value calculus with a semantical value restriction [108], or Jaber's interpretation of SECD machine code [80]. Since classical realizability interpretations provide powerful tools for computational analysis of programs, it naturally raises the question of knowing what is, in a calculus, the structure necessary to the definition of a classical realizability interpretation.

The structures of classical realizability Additionally, as we briefly mentioned in Section 3.5.3, the recent work of Krivine revealed impressive new perspectives in using realizability from a model-theoretic point of view. In [98] and [99], Krivine introduced the notion of *realizability algebras*, which constitute the classical counterpart of *partial combinatory algebras* for intuitionistic realizability. He showed how these structures allow for the construction of models of ZF. Relying on realizability algebras, he defined in particular a model in which neither the continuum hypothesis nor the axiom of choice are valid (see Section 3.5.3), bringing then new perspectives from a model-theoretic point of view.

Roughly speaking, a realizability algebra contains the minimal structure to be a suitable target for compiling the λ_c-calculus. It consists of three sets: a set of *terms* Λ (which contains a certain set of combinators[1]), a set of *stacks* Π and a set of *processes* $\Lambda \star \Pi$ together with a preorder relation $\succ$ on $\Lambda \star \Pi$. These elements are axiomatized in such a way that the relation $\succ$ behaves like the reduction of the abstract machine for the λ_c-calculus. Such a structure is indeed present in each of the cases we studied in this thesis.

The structures of intuitionistic realizability On the other hand, in the continuity of Kleene and Troelstra's tradition of intuitionistic realizability (see [159] for an historical overview), Hyland, Johnstone and Pitts introduced in the 1980s the notion of tripos [79, 135]. A major application of triposes is the effective topos $\mathcal{E}\!f\!f$, later introduced by Hyland in [78], which allows for an analysis of realizability in the general framework of toposes. Let us briefly outline the tripos underlying Kleene realizability. Recall that in Kleene realizability, a formula is realized by natural numbers (see Chapter 3). To each closed formula φ we can then associate the set of its realizers $\{n \in \mathbb{N} : n \Vdash \varphi\}$, which belongs to $\mathcal{P}(\mathbb{N})$. This structure can be generalized to interpret a predicate $\varphi(x)$, where the free variable x ranges over a set X, as a function from X to $\mathcal{P}(\mathbb{N})$ which associates to each $x \in X$ the set $\{n \in \mathbb{N} : n \Vdash \varphi(x)\}$. For instance, given a set X, we can define the equality $=_X$ as the function:

$$=_X\colon (x,y) \in (X \times X) \mapsto \begin{cases} \mathbb{N} & \text{if } x = y \\ \emptyset & \text{otherwise} \end{cases}$$

[1] See [98] for the full definition. The key point is that the set of combinators is complete with respect to the λ-calculus and contains $\boldsymbol{cc}$.

Following the realizability interpretation, we can interpret predicate logic, for instance we define[2]:

$$(\varphi \to \psi)(x) \triangleq \{n \in \mathbb{N} : \forall m \in \varphi(x), n(m) \in \psi(x)\}\,.$$

This naturally induces an entailment relation $\vdash_X$ on predicates for each set X. Given φ, ψ two predicates over X, we say that $\varphi \vdash_X \psi$ if there exists $n \in \mathbb{N}$ such that for all $x \in X$, n realizes $(\varphi \to \psi)(x)$, that is to say:

$$\varphi \vdash_X \psi \quad \triangleq \quad \bigcap_{x \in X} (\varphi \to \psi)(x) \neq \emptyset\,.$$

The entailment relation $\vdash_X$ defines in fact a preorder on predicates. Moreover, the set of predicates equipped with this preorder $(\mathcal{P}(\mathbb{N})^X, \vdash_X)$ broadly defines a Heyting algebra[3]. Indeed, in addition to the arrow $\to$, the connectives $\wedge$, $\vee$ can be defined as in Kleene realizability. It is almost direct[4] to show that for any set X:

$$\chi \wedge \varphi \vdash_X \psi \quad \Leftrightarrow \quad \chi \vdash_X \varphi \to \psi$$

Given two sets X, Y, any function $f : X \to Y$ induces a function f^* from $\mathcal{P}(\mathbb{N})^Y$ to $\mathcal{P}(\mathbb{N})^X$ by precomposing any $\varphi : Y \to \mathcal{P}(\mathbb{N})$ by f: $\varphi \circ f : X \to \mathcal{P}(\mathbb{N})$. In terms of logic, f^* corresponds to the operation of reindexing the variables of a predicate φ along f.

Before turning to a more formal introduction, the last logical notions we want to mention in this context are the quantifiers, whose presentation is due to Lawvere's work [105]. Consider the particular case of a projection $\pi : \Gamma \times X \to \Gamma$. It gives rise to a function $\pi^* : \mathcal{P}(\mathbb{N})^\Gamma \to \mathcal{P}(\mathbb{N})^{\Gamma \times X}$, which turns any predicate φ on Γ into a predicate $\pi^*(\varphi)$ on $\Gamma \times X$. On the contrary, since existential and universal quantifiers on X bind a variable, they are defined as functions from $\mathcal{P}(\mathbb{N})^{\Gamma \times X} \to \mathcal{P}(\mathbb{N})^\Gamma$, in such a way[5] that the following equivalences hold for all $\varphi \in \mathcal{P}(\mathbb{N})^\Gamma$ and for all $\psi \in \mathcal{P}(\mathbb{N})^{\Gamma \times X}$:

$$\begin{array}{rcll} \psi & \vdash_{\Gamma \times X} & \pi^*(\varphi) & \text{if and only if} \quad \exists_X(\psi) \vdash_\Gamma \varphi \\ \pi^*(\varphi) & \vdash_{\Gamma \times X} & \psi & \text{if and only if} \quad \varphi \vdash_\Gamma \forall_X(\psi) \end{array}$$

Up to this point, the structure we exhibited is called a *hyperdoctrine*, due to F. William Lawvere [105]. In broad terms, a hyperdoctrine is defined by a similar structure where the sets $\mathcal{P}(\mathbb{N})^X$ are generalized to arbitrary Heyting algebras $(H_X, \vdash_X)$. A *tripos*, as we will see, is a hyperdoctrine with the extra-assumption that there exists a set $\texttt{Prop}$ (here $\mathcal{P}(\mathbb{N})$) of "propositions" and a generic "truth predicate" $\texttt{tr} \in H_{\texttt{Prop}}$ (here the identity function $\texttt{id}_{\mathcal{P}(\mathbb{N})}$), such that for any predicate φ in H_X, there exists a function $\chi_\varphi : X \to \texttt{Prop}$ which verifies:

$$\varphi \dashv\vdash_X \chi_\varphi^*(\texttt{tr})$$

Triposes, which were studied and defined by Andrew Pitts during his PhD thesis [135, 136], have been conducive to the categorical analysis of realizability.

Towards a categorical presentation of classical realizability For a long time, Krivine classical realizability and the categorical approach to realizability seemed to have no connections. The situation changed in the past ten years, notably thanks to Thomas Streicher who built an important bridge in [151]. After reformulating the Krivine's abstract machine of the λ_c-calculus as an *abstract Krivine*

[2] Remember that a natural number n is identified with the n^{th} recursive function of a fixed enumeration.

[3] Strictly speaking, it actually defines a Heyting prealgebra, that is to say a Heyting algebra whose career is a preorder (whitout the property of anti-symmetry) instead of a poset.

[4] In terms of recursive functions, the left-to-right implication is merely curryfication and vice-versa.

[5] We let the reader check that in the general case of a function $f : X \to Y$, we can define the quantifiers by

$$\exists_f(\varphi)(y) \triangleq \textstyle\bigcup_{x \in X}(f(x) =_Y y \wedge \varphi(x)) \qquad \forall_f(\varphi)(y) \triangleq \textstyle\bigcap_{x \in X}(f(x) =_Y y \to \varphi(x))$$

structure (AKS), Streicher proved that from each AKS one may construct a *filtered ordered partial combinatory algebra* and a tripos. Later on, in a series of papers from 2013-2015 [44, 44, 45] Walter Ferrer Santos, Jonas Frey, Mauricio Guillermo, Octavio Malherbe and Alexandre Miquel developed the theory of *Krivine ordered combinatory algebras* ($^{\mathcal{K}}$OCA) for classical realizability. Their main purpose was to try to abstract as much as possible the essence of abstract Krivine structures, in order to get a structure which is as general as possible and which captures the necessary ingredients to generate Krivine models (*i.e.* triposes).

This part of the thesis is in line with this general purpose. In the next chapters, we will introduce the notion of *implicative algebras*, developed by Alexandre Miquel [121]. As we shall see, these are structures which encompass all the structure necessary to the definition of classical realizability models. In particular, the λ_c-calculus and the ordered combinatory algebras are definable within implicative algebras. In addition, they allow for simple criteria to determine whether the induced realizability tripos collapses to a forcing tripos. Based on the arrow connective, implicative algebras somewhat reflect the enriched lattice structure underlying Krivine realizability interpretation of logic. After introducing these structures, we will present the notion of *disjunctive algebras* and *conjunctive algebras*. Respectively based on the 'par' $⅋$ and the tensor $\otimes$ connectives together with a negation, these structures reflects the corresponding decompositions of the arrow in linear logic. As we will explain, these decompositions can be interpreted in terms of evaluation strategies: disjunctive algebras naturally arise from a call-by-name fragment of Munch-Maccagnoni's System L [126], while conjunctive algebras correspond to a call-by-value fragment of the same.

9.1 The underlying lattice structure

9.1.1 Classical realizability

Let us start by arguing that through the Curry-Howard interpretation of logic, and especially in realizability, there is an omnipresent lattice structure. This structure is reminiscent of the concept of subtyping, which makes concrete, in a programming language, a well-known fact in mathematics: if f is a function whose domain is a set X (say the set $\mathbb{R}$), and if S is a subset of X (say $\mathbb{N} \subset \mathbb{R}$), then f can be restricted to a function $f_{|S}$ of domain S. Similarly, in object-oriented programming, if a program p takes as input any object in a class C, if D is a class which inherits of the structure of C, p can be applied to any object in D. This idea is usually reflected in the theory of typed calculus by a *subtyping relation*, often denoted by $<:$, where $T <: U$ means that T is more precise as a type. For instance, type systems including a subtyping relation (see [22] for instance) usually have the rules:

$$\frac{\Gamma \vdash p : T \quad T <: U}{\Gamma \vdash p : U}\ (\textsc{Sub}) \qquad\qquad \frac{U_1 <: T_1 \quad T_2 <: U_2}{T_1 \to T_2 <: U_1 \to U_2}\ (\textsc{S-Arr})$$

The first rule, called *subsumption*, says that we can always replace a type by a supertype. The second one expresses that the arrow is contravariant on its left-hand side and covariant on its right-hand side. To say it differently, if we think of $T <: U$ as "T is more constrained than U is", and consider the rule `nat` $<:$ `real`, a function of type `real` $\to$ `nat` is indeed more constrained than a function of type `real` $\to$ `real`, itself more constrained than the type `nat` $\to$ `real`. Besides, as suggested by the notation, the subtyping relation is reflexive and anti-symmetric, it thus induces a preorder on types.

This relation is implicit in classical realizability, in the sense that the subsumption rule is always adequate: if $A <: B$, for any pole, if $t \Vdash A$ then $t \Vdash B$ (see [144, Proposition 3.1.1]). In terms of truth values, this means that if $A <: B$, then $\|A\| \supseteq \|B\|$ (and hence $|A| \subseteq |B|$). We said that this relation was implicit, and indeed, even when the relation is not syntactically defined, given a pole $\perp\!\!\!\perp$ it is always possible to define a semantic notion of subtyping[6]:

Subtyping $\qquad A \leq_{\perp\!\!\!\perp} B \triangleq$ for all valuations ρ, $\|B\|_\rho \subseteq \|A\|_\rho$

In this case, the relation $\leq$ being induced from (reversed) set inclusions, it comes with a richer structure of complete lattice, where the meet $\wedge$ is defined as a union and the join $\vee$ as an intersection. Observe that in particular, this corresponds to the interpretation of universal quantifiers in classical realizability:

$$\|\forall x.A\|_\rho \triangleq \bigcup_{n\in\mathbb{N}} \|A\|_{\rho[x\mapsto n]} = \curlywedge\{\|A\|_{\rho[x\mapsto n]} : n \in \mathbb{N}\}$$

In this lattice structure, quantifiers are thus naturally defined as meets and joins, while the logical connectives $\wedge$ and $\vee$, in the case of realizability, are interpreted in terms of products and sums. To sum up, classical realizability then correspond to the following picture:

Realizability: $\qquad \forall = \curlywedge \qquad \wedge = \times \qquad \exists = \curlyvee \qquad \vee = +$

9.1.2 Forcing

In turn, in the cases of semantics given by Heyting algebras (for intuitionistic logic) or Boolean algebras (for classical logic), quantifiers and connectives are both interpreted in terms of meets and joins. To put it differently, the universal quantifier is semantically defined as an infinite conjunction, while the existential one is defined as an infinite union. These cases are not different from Kripke semantics for intuitionistic logic or Cohen forcing in the case of classical logic.

Let us first examine the case of Kripke models to show that they induce Heyting algebras. Consider indeed a Kripke model $(\mathcal{W}, \leq, D, V)$ (see Chapter 1). Then let us denote by $\mathcal{U}$ the set of upward closed subsets of $\mathcal{W}$:

$$\mathcal{U} \triangleq \{U \subseteq \mathcal{W} : \forall v, w \in \mathcal{W}, v \in U \wedge v \leq w \Rightarrow w \in U\}$$

The intersection (resp. the union) of upward closed sets being itself upward closed, $(\mathcal{U}, \subseteq)$ defines a lattice structure, whose higher element $\top$ is $\mathcal{W}$. In fact, this structure is even a Heyting algebra, where for any sets $U, V \in \mathcal{U}$, the arrow is defined by:

$$U \to V \triangleq \{w \in \mathcal{W} : \forall v \in \mathcal{W}, w \leq v \wedge v \in U \Rightarrow v \in V\}$$

It is routine to check that $U \to V$ belongs to $\mathcal{U}$ and that it satisfies the properties of the implication operation in Heyting algebras[7]. Moreover, it can be shown[8] that the validity under Kripke semantics in the model $(\mathcal{W}, \leq, D, V)$ corresponds to the interpretation in the Heyting algebra $(\mathcal{U}, \subseteq)$:

$$[\![\varphi]\!]^{\mathcal{U}} = \{w \in \mathcal{W} : w \Vdash \varphi\}$$

and thus $\mathcal{U} \vDash \varphi$, that is to say $[\![\varphi]\!]^{\mathcal{U}} = \top$, if and only if $\forall w \in \mathcal{W}, w \Vdash \varphi$.

Regarding Cohen forcing, a very similar construction allows us to reduce it to the case of Boolean-valued models [14]. Loosely speaking, Cohen forcing is a construction which, starting from a ground model $\mathcal{M}$ of set theory and a poset $(P, \leq)$ of forcing conditions, defines a new model $\mathcal{M}[G]$ where G is a generic filter on P. Without entering into the definition of $\mathcal{M}[G]$, we can briefly explain how the validity in $\mathcal{M}[G]$ can be understood in terms of Boolean algebras. First, any poset $(P, \leq_P)$ can be

[6] Note that this definition is specific to classical realizability, in the intuitionistic case, semantic subtyping $A <: B$ is defined as the inclusion $|A| \subseteq |B|$ of truth value. In the classical setting, semantic subtyping is thus defined as the reversed inclusion of falsity values $\|B\| \subseteq \|A\|$, which is a strictly stronger condition (in fact, the inclusion of truth value $|A| \subseteq |B|$ does not constitute a valid definition of subtyping in the classical case).

[7] Both direction of the equivalence $U \cap X \subseteq V \Leftrightarrow X \subseteq U \to V$ are simple exercises.

[8] See for instance [47] for a complete proof.

embedded by an order-preserving morphism to $RO(P)$ the complete Boolean algebra of regular open sets[9] of P. The embedding e in question maps every forcing condition p to the interior of the closure of the following open set:

$$O_p = \{q \in P : q \leq p\}\,.$$

Writing B for the Boolean algebra $RO(P)$, the forcing relation can then be defined by:

$$p \Vdash \varphi \;\; \triangleq \;\; e(p) \leq [\![\varphi]\!]^{\mathcal{B}}$$

where $[\![\cdot]\!]^{\mathcal{B}}$ is the interpretation in the Boolean-valued model $\mathcal{M}^{\mathcal{B}}$. Finally, the validity of a formula φ in $\mathcal{M}[G]$ is broadly[10] defined by the existence of a condition $p \in G$ which forces φ. The truth value under the forcing translation can thus be interpreted in terms of Boolean algebras.

For these reasons, we can say that the interpretation of connectives and quantifications in intuitionistic (Kripke) and classical (Cohen) forcing amount to their interpretations in Heyting and Boolean algebras, respectively. This situation can be summarized by:

Forcing:	$\forall = \wedge = \curlywedge$	$\exists = \vee = \curlyvee$

In this sense, the realizability interpretation is therefore, *a priori*, more general than the forcing one.

9.2 A types-as-programs interpretation

Let us put the focus back on the lattice structure in realizability, and more specifically to the subtyping relation. Given a fixed pole $\perp\!\!\!\perp$, the semantic definition of the subtyping relation that we gave is equivalent to:

$$A \leq_{\perp\!\!\!\perp} B \text{ if for all } t\text{, whenever } t \Vdash A \text{ then } t \Vdash B$$

Formulas are thus ordered according to their truth values, which are set of realizers. Loosely speaking, we are identifying formulas with their realizers. On the other hand, many semantics allows us to associate terms with their principal types. For instance, the identity $I = \lambda x.x$ can be identified to its principal type $\forall X.X \to X$; doing so, the fact that $I \Vdash \texttt{nat} \to \texttt{nat}$ can be read as $\forall X.X \to X \leq \texttt{nat} \to \texttt{nat}$ at the level of formulas. Identifying terms with their principal type allows us to associate to each realizer the truth value of its principal types (to which it belongs). In other words, it corresponds to the following informal inclusion:

$$\text{Realizers} \subseteq \text{Truth values}$$

But what can be said about the reverse inclusion? In order to consider truth values as realizers we should be able to lift the operations of λ-abstraction and application at the level of truth values. As we shall see in the next chapters, this is in fact perfectly feasible in simple algebraic structures, called *implicative structures*. In these structures, that we present in Chapter 10, truth values can be regarded as generalized realizers and manipulated as such. In particular, it suggests that the previous inclusion of realizers into truth values could actually be turned into an equality:

$$\text{Realizers} = \text{Truth values}$$

An important feature of implicative structures is thus that they allow to formalize this identification. In particular, any truth value a will be identified with the realizer whose principal type is a itself. Implicative structures are complete lattices equipped with a binary operation $a \to b$ verifying properties

[9] For the order topology. Regular open sets are open sets which are equal to the interior of their closure.

[10] To be more accurate, a formula $\phi(x_1,...x_n)$ is valid in $\mathcal{M}[G]$ if there exists a condition p in G which forces $\phi(\underline{x}_1,...\underline{x}_n)$ where $\underline{x}_i$ is a name in $\mathcal{M}^B$ for x_i. We really do not want to formally introduced forcing here, an introduction in terms of Boolean-valued model is given in [14].

coming from the logical implication. As we will see, they indeed allow us to interpret both the formulas and the terms in the same structure. For instance, the ordering relation $a \leq b$ will encompass different intuitions depending on whether we regard a and b as formulas or as terms. Namely, $a \leq b$ will be given the following meanings:

- the formula a is a subtype of the formula b;
- the term a is a realizer of the formula b;
- the realizer a is more defined than the realizer b.

The last item correspond to the intuition that if a is a realizer of all the formulas of which b is a realizer, a is more precise than b, or more powerful as a realizer. Therefore, a and b should be ordered.

In terms of the Curry-Howard correspondence, this means that not only do we identify types with formulas and proofs with programs, but we also identify types and programs. Visually, this corresponds to the following situation:

$$\begin{tikzcd}
\text{Types} \arrow[r, equal] \arrow[d, equal] & \text{Formulas} \arrow[d, equal] \\
\lambda\text{-terms} \arrow[r, equal] & \text{Proofs}
\end{tikzcd}$$

which is to be compared with the corresponding diagram in Section 2.3.

Because we consider formulas as realizers, any formula will be at least realized by itself. In particular, the lowest formula $\bot$ is realized. While this can be dazzling at first sight, it merely reflects that implicative structures do not come with an intrinsic criterion of consistency. To this purpose, we will introduce the notion of *separator*, which is similar to the usual notion of filter for Boolean algebras. *Implicative algebras* will be defined as implicative structures equipped with a separator. As we shall see, they capture the algebraic essence of classical realizability models. In particular, we will embed both the λ_c-calculus and its type system in such a way that the adequacy is preserved. Furthermore, we will see that they give rise to the usual realizability triposes, and that they provide us with simple criteria to determine whether the induced triposes collapse to forcing triposes. Implicative algebras therefore appear to be the adequate algebraic structure to study classical realizability and the models it induces.

9.3 Organization of the third part

Above all, we shall warn the reader that the very concept of implicative algebras—as well as the different results that we present about it—in this manuscript are not ours. They are due to Alexandre Miquel, who have been giving numerous talks on the topic [121], but they are unpublished for the time being. In particular, the next chapter should not be taken as a scientific contribution peculiar to this thesis, even our presentation of the subject is deeply influenced by Miquel's own presentation. Our only contribution about implicative algebras is the Coq formalization that we will mention in the next chapter.

First, we recall in the next section some definitions of basic algebraic structures and some vocabulary from category theory that are used in the sequel. Next, in the last section of this chapter, we present the algebraic structures prior this work which are used in the study of realizability from a categorical point of view. This last section is intended to be a brief survey of the work of Streicher [151] and Ferrer, Frey, Guillermo, Malherbe and Miquel [45] on the topic. This will naturally lead us to the definitions of implicative algebras in the following chapter.

Chapter 10 is then devoted to the presentation of implicative algebras. We first introduce the notion of implicative structures and give a few examples. Next, we show how to embed both the λ_c-calculus and its second-order type system while proving the adequacy of the embedding. We then introduce the notion of separators and implicative algebras, and show how they induce realizability triposes.

In Chapter 11, we present a similar structure which is based on the decomposition of the arrow $a \to b$ as $\neg a \vee b$. We first give a computational account for this decomposition in a fragment of Munch-Maccagnoni's system L, and explain how it is related to the choice of a call-by-name evaluation strategy for the λ-calculus. We then introduce the notion of disjunctive algebras, which we relate to the implicative ones. Similarly, we present in Chapter 12 a structure based on the decomposition of the arrow $a \to b$ as $\neg(a \wedge \neg b)$ and follow the same process towards the definition of conjunctive algebras.

This part of the thesis is supported by a Coq development[11], in which most of the results are proved. My motivation for this development was twofold. First, I should confess that I started it as an (amusing) exercise to better understand implicative algebras. Because I was probably the first in the position of checking Miquel's definitions and results, I thought that the best way to do it might be to formalize everything. Second, insofar as implicative algebras aim, on a long-term perspective, at providing a foundational ground for the algebraic analysis of realizability models, a Coq formalization also seemed to be a good way of laying the foundations of these structures.

9.4 Categories and algebraic structures

9.4.1 Lattices

We recall some definitions and properties about lattices. Since the proofs are very standard, we omit them and refer the reader to the Coq formalization if needed.

Definition 9.1 (Lattice)**.** A *lattice* is a partially ordered set $(\mathcal{L}, \leq)$ such that that any pair of elements $a, b \in \mathcal{L}$ admits:

1.• a greatest lower bound, which we write $a \wedge b$;
2.• a lowest upper bound, which we write $a \vee b$.

⌟

In order to interpret the quantifications, we will pay attention to arbitrary meets and joins, hence to complete lattices:

Definition• 9.2. A lattice $\mathcal{L}$, is said to be *meet-complete* (resp. *join-complete*) if any subset $A \subseteq \mathcal{L}$ admits a greatest lower bound (resp. lowest upper bound), written $\bigwedge_{a \in A} a$ or simply $\bigwedge A$ (resp. $\bigvee_{a\in A} a$ and $\bigvee A$). It is said to be *complete* if it is both meet- and join-complete. ⌟

The following theorem states that any meet-complete lattice is also join-complete and vice-versa:

Theorem• 9.3. *If $\mathcal{L}$ is a meet-complete lattice, then $\mathcal{L}$ is a complete lattice with the join operation defined by:*

$$\bigvee_{a \in A} a \triangleq \bigwedge_{a \in ub(A)} a$$

where $ub(A)$ is the set of upper-bounds of A. The converse direction is similar.

Any complete lattice has a lowest and a highest element, which we write $\bot$ and $\top$:

Proposition 9.4. *In any complete lattice $\mathcal{L}$, the following holds:*

[11] The source of the Coq development can be browsed or downloaded from here• [122]. We use the bullet to denote the statements which are formalized in the development. In the electronic version of the manuscript, these statements are given with an hyperlink pointing directly to their Coq counterpart.

1.$^\bullet$ $\top = \bigwedge \emptyset = \bigvee \mathcal{L}$ *2.*$^\bullet$ $\bot = \bigvee \emptyset = \bigwedge \mathcal{L}$

Finally, we recall that reversing the order of a (complete) lattice still gives a (complete) lattice where meet and join are exchanged:

Proposition$^\bullet$ 9.5. *If $(\mathcal{L},\leq)$ is a complete lattice, then $(\mathcal{L},\triangleleft)$ where $a \triangleleft b \triangleq b \leq a$ is a complete lattice.*

9.4.2 Boolean algebras

We recall the definition and some key properties of Boolean algebras.

Definition$^\bullet$ 9.6. A *Boolean algebra* is a quadruple $(\mathcal{B},\leq,\bot,\top)$ such that:

- $(\mathcal{B},\leq,\vee,\wedge)$ is a bounded lattice, $\top$ being the upper bound of $\mathcal{B}$ and $\bot$ its lower bound
- $\mathcal{B}$ is distributive, in the sense that:

$$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c) \qquad a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c) \qquad (\forall a,b,c \in \mathcal{B})$$

- every element $a \in \mathcal{B}$ has a complement, which we write $\neg a$, in the sense that:

$$a \vee \neg a = \top \qquad a \wedge \neg a = \bot \qquad (\forall a \in \mathcal{B})$$

A Boolean algebra is said to be *complete* if it is complete as a lattice. ⌟

We state some properties of Boolean algebras, in particular the commutation of the negation with the other internal laws:

Proposition 9.7. *If $\mathcal{B}$ is a complete Boolean algebra, the following hold:*

1.$^\bullet$ $b = \neg a$ *if and only if* $(a \vee b = \top)$ *and* $(a \wedge b = \bot)$ $(\forall a,b \in \mathcal{B})$

2.$^\bullet$ $\neg\neg a = a$ $(\forall a \in \mathcal{B})$

3.$^\bullet$ $\neg(a \vee b) = (\neg a) \wedge (\neg b)$ *and* $\neg(a \wedge b) = (\neg a) \vee (\neg b)$ $(\forall a,b \in \mathcal{B})$

Finally, we recall the commutation of the negation with arbitrary joins and meets in complete Boolean algebras:

Theorem 9.8. *If $\mathcal{B}$ is a complete Boolean algebra, then the following holds for any $A \subseteq \mathcal{B}$:*

1.$^\bullet$ $\neg \bigwedge\{a : a \in A\} = \bigvee\{\neg a : a \in A\}$ *2.*$^\bullet$ $\neg \bigvee\{a : a \in A\} = \bigwedge\{\neg a : a \in A\}$

All these commutations can be interpreted in terms of logical commutation in Boolean-valued models. The first ones indicate that the internal logic of Boolean-valued models (and in particular of forcing models) has an involutive negation and that De Morgan's laws are satisfied. The former theorem indicate that negation commutes with quantifiers as follows:

$$\neg\forall = \exists\neg \qquad \neg\exists = \forall\neg$$

These equalities will not hold in general in implicative algebras. Better, they will precisely characterize the collapse of the induced realizability triposes to forcing ones. In this sense, these commutations show that implicative algebras are a strict refinement of Boolean algebras. As such, they also are the sign that implicative algebras might provide us with models which are *a priori* more general than Boolean-valued models.

9.4.3 Categories

We briefly introduce some standard notions of category theory in order to further define the notions of hyperdoctrine and tripos.

Definition 9.9. A *category* C is given by a class of *objects* together with a class of *morphisms* $C(a,b)$ for each pair $a,b \in C$ of objects, as well as:

- an associative composition of morphism, which is written $g \circ f$ for all $f \in C(a,b), g \in C(b,c)$,
- a morphism $\mathtt{id}_a \in C(a,a)$ (identity) for each $a \in C$, such that:

$$\forall f \in C(a,b), f \circ \mathtt{id}_a = \mathtt{id}_b \circ f = f$$

⌟

The property required for the identity and the associativity of the composition can be expressed in terms of diagrams, by requiring that the following diagrams commute[12]:

$$\begin{tikzcd}
a \arrow[r, "f"] \arrow[d, "\mathtt{id}_a"'] \arrow[dr, "f"] & b \arrow[d, "\mathtt{id}_b"] \\
a \arrow[r, "f"'] & b
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
 & b \arrow[dr, "h \circ g"] \arrow[dd, "g"] & \\
a \arrow[ur, "f"] \arrow[dr, "g \circ f"'] & & d \\
 & c \arrow[ur, "h"'] &
\end{tikzcd}$$

In the sequel, we will often express properties by means of diagrams. Most of the algebraic structures that we mentioned until here can be regarded as particular categories with extra structure. The class of a given structure (say the Boolean algebras, the lattices) also form a category in general, whose morphisms are the structure-preserving functions. For instance, the following structures are categories:

- **Set**, the category of sets, whose objects are sets and whose morphisms are the functions between sets;
- **Poset**, the category whose objects are posets and whose morphisms are order-preserving functions;
- any poset $(P,\leq)$ can be regarded as a category whose objects are its elements, and where there is morphism between two objects x and y when $x \leq y$;
- **Lat**, the category of lattices, is formed with lattices as objects and functions preserving the meet $\wedge$ and the join $\vee$ as morphisms;
- any lattice $(\mathcal{L},\leq)$ can be considered in itself as a category;
- etc.

We recall some standard definitions about objects and morphisms:

Definition 9.10. Let C be a category:

- A morphism $f : a \to b$ is said to be *invertible* if there exists a morphism $g : b \to a$ such that $g \circ f = \mathtt{id}_a$ et $f \circ g = \mathtt{id}_b$
- a and b are said to be *isomorphic* if there exists $f \in C(a,b)$ invertible
- an object t is said to be *terminal* if $\forall a \in C, \exists! f : a \to t$
- an object i is said to be *initial* if $\forall a \in C, \exists! f : i \to a$

⌟

[12] That is to say that if we take an element of the object a, the images we will obtain by two paths leading to the same object will be equal.

Definition 9.11 (Dual category)**.** Let $\mathcal{C}$ and $\mathcal{D}$ be two categories. We define:

- $\mathcal{C}^{op}$ the *dual category* of $\mathcal{C}$ as being the category with the same objects in which morphisms and the composition are reversed: $\mathcal{C}^{op}(a,b) = \mathcal{C}(b,a)$, $f \circ_{\mathcal{C}^{op}} g = g \circ_{\mathcal{C}} f$
- $\mathcal{C} \times \mathcal{D}$ the *product category* of $\mathcal{C}$ and $\mathcal{D}$, whose objects are pairs of objects $(c \in \mathcal{C}, d \in \mathcal{D})$, and whose morphisms are pairs of morphisms, identities pairs of identities and where the composition is defined componentwise. ⌟

9.4.4 Functors

The notion of (covariant) functor is a natural generalization of the usual notion of morphism:

Definition 9.12 (Functor)**.** Let $\mathcal{C}$ and $\mathcal{D}$ be two categories. A *covariant functor F* from $\mathcal{C}$ to $\mathcal{D}$ is a correspondence that maps each object a of $\mathcal{C}$ to an object $F(a)$ of $\mathcal{D}$, and each morphism f in $\mathcal{C}(a,b)$ to a morphism $F(f)$ in $\mathcal{D}(F(a),F(b))$ for all $a,b \in \mathcal{C}$, which preserves:

- the identity: $\forall a \in \mathcal{C}, F(\mathtt{id}_a) = \mathtt{id}_{F(a)}$
- the composition: $\forall f \in \mathcal{C}(a,b), g \in \mathcal{C}(b,c), F(g \circ f) = F(g) \circ F(f)$

⌟

Example 9.13. For instance, we can define the powerset functor $\mathcal{P}$: **Set** → **Set** which constructs the subsets of a set:

$$\mathcal{P} : \begin{cases} x \mapsto \mathcal{P}(x) \\ (f : x \to y) \mapsto \mathcal{P}f : \begin{cases} \mathcal{P}(x) \to \mathcal{P}(y) \\ s \mapsto f(s) \end{cases} \end{cases}$$

⌟

The composition of functors is defined canonically. An *isomorphism of categories* is as a functor which is bijective both on objects and on morphisms (or equivalently as a functor which is invertible for the composition of functors). This allows us to define **Cat**, the category whose objects are categories and whose morphisms are functors.

The previous definition can be extended to the notion of *contravariant functors*, which reverse morphisms and the composition:

Definition 9.14 (Contravariant functor)**.** A *contravariant functor F* from $\mathcal{C}$ to $\mathcal{D}$ from $\mathcal{C}$ to $\mathcal{D}$ is a correspondence that maps each object a of $\mathcal{C}$ to an object $F(a)$ of $\mathcal{D}$, and each morphism f in $\mathcal{C}(a,b)$ to a morphism $F(f)$ in $\mathcal{D}(F(b),F(a))$ for all $a,b \in \mathcal{C}$, such that:

$$\forall f \in \mathcal{C}(a,b), \forall g \in \mathcal{C}(b,c), F(g \circ f) = F(f) \circ F(g)$$

Equivalently, a contravariant functor is a functor from $\mathcal{C}^{op}$ to $\mathcal{D}$. ⌟

Being given two categories, we can thus study the class of functors between these two categories. Actually, we can even equip this class with operators, which are called natural transformations:

Definition 9.15 (Natural transformation)**.** Let $\mathcal{C}$ and $\mathcal{D}$ be two categories, and $F,G : \mathcal{C} \to \mathcal{D}$ two functors. A *natural transformation α* from F to G is a family of morphisms $(\alpha_a)_{a \in \mathcal{C}}$, with $\alpha_a \in \mathcal{D}(F(a),G(a))$ for all $a \in \mathcal{C}$ and such that for all $f \in \mathcal{C}(a,b)$, the following diagram commutes:

$$\begin{tikzcd}
F(a) \arrow[r, "F(f)"] \arrow[d, "\alpha_a"'] & F(b) \arrow[d, "\alpha_b"] \\
G(a) \arrow[r, "G(f)"'] & G(b)
\end{tikzcd}$$

If in addition, for any object $a \in \mathcal{C}$, the morphism α_a is invertible, we say that α is a natural bijection. A functor $F : \mathcal{C} \to \mathcal{D}$ is then called an *equivalence of categories* when there exists a functor $G : \mathcal{D} \to \mathcal{C}$ and two natural bijections from $F \circ G$ (resp. $G \circ F$) to the identity functor of C (resp. the one of $\mathcal{D}$). This notion generalizes the one of isomorphisms of categories. ⌟

Definition 9.16 (Adjunction)**.** Let $\mathcal{C}$ and $\mathcal{D}$ be categories, an *adjunction* between $\mathcal{C}$ and $\mathcal{D}$ is a triple (F, G, φ) where:

- F is a functor from $\mathcal{D}$ to $\mathcal{C}$;
- G is a functor from $\mathcal{C}$ to $\mathcal{D}$;
- for all $c \in \mathcal{C}, d \in \mathcal{D}$, $\varphi_{c,d}$ is a bijection from $\mathcal{C}(F(d), c)$ to $\mathcal{D}(d, G(c))$, natural in c and d.

We denote it by $F \dashv G$, F is said to be the *left adjoint* (of G), and vice-versa. ⌟

We introduce a last definition describing a broad class of categories. These categories allow for instance to give a categorical counterpart to the λ-calculus, see for instance [8] for an introductory presentation.

Definition 9.17 (Cartesian category)**.** Let $\mathcal{C}$ be a category, $a, b \in \mathcal{C}$. A product of a and b is a triple $(a \times b, \pi_a, \pi_b)$, where $a \times b \in \mathcal{C}$, $\pi^1_{a\times b} \in \mathcal{C}(a \times b, a)$ and $\pi^2_{a\times b} \in \mathcal{C}(a \times b, b)$ are such that for all $f \in \mathcal{C}(c, a), g \in \mathcal{C}(c, b)$, there exists a unique morphism $\langle f, g\rangle \in \mathcal{C}(c, a \times b)$ such that the following diagrams commutes:

$$\begin{tikzcd}
 & c \arrow[ld, "f"'] \arrow[d, "\langle f,g\rangle"] \arrow[rd, "g"] & \\
a & a \times b \arrow[l, "\pi^1_{a\times b}"] \arrow[r, "\pi^2_{a\times b}"'] & b
\end{tikzcd}$$

A category is said *Cartesian* if it contains a terminal object $\top$ and if every pair of objects has a product. A Cartesian category is said to be *closed* if for any object $c \in \mathcal{C}$, the functor $(\cdot) \times c : \mathcal{C} \to \mathcal{C}$ has a right-adjoint, which we write $c \to (\cdot)$. ⌟

9.4.5 Hyperdoctrines and triposes

We can now define the structures which allow for a categorical approach of realizability. First, we recall the definition of Heyting algebras:

Definition$^\bullet$ 9.18. A Heyting algebra $\mathcal{H}$ is a bounded lattice such that for all $a, b \in \mathcal{H}$ there is a greatest element x of $\mathcal{H}$ such that $a \wedge x \leq b$. This element is denoted $a \to b$. ⌟

In any Heyting algebra, one defines the pseudo-complement $\neg a$ of any element a by setting $\neg a \triangleq (a \to \bot)$. By definition, $a \wedge \neg a = \bot$ and $\neg a$ is the largest element having this property. However, it is not true in general that $a \vee \neg a = \top$, thus $\neg$ is only a pseudo-complement, not a real complement, as would be the case in a Boolean algebra. A *complete Heyting algebra* is a Heyting algebra that is complete as a lattice. Observe that Heyting algebras form a category[13] **HA** whose morphisms $F : \mathcal{H} \to \mathcal{H}'$ are the morphisms of the underlying lattice structure preserving Heyting's implication: $F(a \to b) = F(a) \to F(b)$ for all $a, b \in \mathcal{H}$.

In the category of Heyting algebras, we have a particular notion of adjunction, which is peculiar to partially ordered sets:

[13] Formally, **HA** is a subcategory of the category **Ord** of pre-orders. This category is sometimes called of Heyting prealgebras since the equality is induced by the preorder relation $a = b \triangleq a \leq b \wedge b \leq a$. In the literature this equality is sometimes written $a \cong b$ and called an isomorphism to distinguish it from the equality of pre-ordered sets.

Definition 9.19 (Galois connection). A *Galois connection* between two posets A, B is a pair of function $f : A \to B, g : B \to A$ such that:

$$f(x) \leq y \Leftrightarrow x \leq g(y)$$

⌟

For instance, the following examples are Galois connections:

- the natural injection and the floor form a Galois connection between $\mathbb{N}$ and $\mathcal{R}$:

$$\forall n \in \mathbb{N}, \forall x \in \mathcal{R}, (n \leq x \Leftrightarrow n \leq \lfloor x \rfloor)$$

- in any Heyting algebra $\mathcal{H}$, given $a \in \mathcal{H}$, we have:

$$\forall x, y \in \mathcal{H}, (a \wedge x \leq y \Leftrightarrow x \leq a \to y)$$

- in any lattice $\mathcal{L}$, (binary) meets and joins are respectively the left and right adjoints of a Galois connection formed with the diagonal morphism $\Delta : \mathcal{L} \to \mathcal{L} \times \mathcal{L}$.

Proposition 9.20. *If (f, g) is a Galois connection between two ordered sets A, B, then:*

1. *f and g are monotonic functions,*
2. *g is fully determined by f (and thus unique) and vice-versa.*

Proof. It is easy to check that indeed, f is uniquely determined by g:

$$f(x) = \min \{y \in B : x \leq g(y)\} \qquad \text{(for all } x \in A)$$

and vice-versa. □

We are now ready to define the key notion of (first-order) hyperdoctrine, due to Lawvere [105]. While there are many definitions of this notion in the literature, they are not always equivalent. Here, we follow Pitt's presentation [136] by adopting a minimal definition. This definition captures exactly the notion of first-order theory with equality.

Definition 9.21 (Hyperdoctrine). Let C be a Cartesian closed category. A *first-order hyperdoctrine* over C is a contravariant functor $\mathcal{T} : C^{op} \to \mathbf{HA}$ with the following properties:

1. For each diagonal morphism $\delta_X : X \to X \times X$ in C, the left adjoint to $\mathcal{T}(\delta_X)$ at the top element $\top \in \mathcal{T}(X)$ exists. In other words, there exists an element $=_X \in \mathcal{T}(X \times X)$ such that for all $\varphi \in \mathcal{T}(X \times X)$:

$$\top \leq \mathcal{T}(\delta_X)(\varphi) \quad \Leftrightarrow \quad =_X \leq \varphi$$

2. For each projection $\pi^1_{\Gamma,X} : \Gamma \times X \to \Gamma$ in C, the monotonic function $\mathcal{T}(\pi^1_{\Gamma,X}) : \mathcal{T}(\Gamma) \to \mathcal{T}(\Gamma \times X)$ has both a left adjoint $(\exists X)_\Gamma$ and a right adjoint $(\forall X)_\Gamma$:

$$\begin{array}{ccc} \varphi \leq \mathcal{T}(\pi^1_{\Gamma,X})(\psi) & \Leftrightarrow & (\exists X)_\Gamma(\varphi) \leq \psi \\ \mathcal{T}(\pi^1_{\Gamma,X})(\varphi) \leq \psi & \Leftrightarrow & \varphi \leq (\forall X)_\Gamma(\psi) \end{array}$$

3. These adjoints are natural in Γ, i.e. given $s : \Gamma \to \Gamma'$ in C, the following diagrams commute:

$$\begin{tikzcd} \mathcal{T}(\Gamma' \times X) \arrow[r, "\mathcal{T}(s \times \mathrm{id}_X)"] \arrow[d, "(\exists X)_{\Gamma'}"'] & \mathcal{T}(\Gamma \times X) \arrow[d, "(\exists X)_\Gamma"] \\ \mathcal{T}(\Gamma') \arrow[r, "\mathcal{T}(s)"'] & \mathcal{T}(\Gamma) \end{tikzcd} \qquad \begin{tikzcd} \mathcal{T}(\Gamma' \times X) \arrow[r, "\mathcal{T}(s \times \mathrm{id}_X)"] \arrow[d, "(\forall X)_{\Gamma'}"'] & \mathcal{T}(\Gamma \times X) \arrow[d, "(\forall X)_\Gamma"] \\ \mathcal{T}(\Gamma') \arrow[r, "\mathcal{T}(s)"'] & \mathcal{T}(\Gamma) \end{tikzcd}$$

This condition is also called the *Beck-Chevaley conditions*.

The elements of $\mathcal{T}(X)$, as X ranges over the objects of C, are called the $\mathcal{T}$*-predicates*. ⌟

Let us give some intuitions about this definition, which are related to the informal introduction of hyperdoctrine we did at the beginning of the chapter:

- The base category C is the *domain of discourse*, that is to say that its elements are types or contexts (whence the suggestive notations X and Γ) on which the predicates range. Its morphisms thus correspond to substitutions, while products $\Gamma \times \Gamma'$ should be understood as the concatenations of contexts.
- The functor $\mathcal{T}$ associates to each context $\Gamma \in C$ the sets of predicates over Γ. It might be helpful to think of the elements of $\mathcal{T}(\Gamma)$ as formulas $\varphi(x_1,\ldots,x_n)$ of free variables $x_1 : X_1,\ldots,x_n : X_n$ with $\Gamma \equiv X_1,\ldots,X_n$. The structure of Heyting algebra means that predicates can be compound by means of the connectives $\wedge, \vee, \rightarrow$ and that these operations respect the laws of intuitionistic propositional logic.
- The functoriality of $\mathcal{T}$, that is the fact that each morphism $s : \Gamma \rightarrow \Gamma'$ in C induces a morphism $\mathcal{T}(s) : \mathcal{T}(\Gamma') \rightarrow \mathcal{T}(\Gamma)$, is to be understood as the existence of substitutions on formulas. In other words, if $\varphi(x)$ is a predicate ranging over Γ and s is as above, then $\mathcal{T}(s)(\varphi)$ is intuitively the predicate $\varphi(s(y))$.
- The ordering on formulas corresponds to the inclusion of predicates in the sense of the associated theory, that is to say:
$$\varphi \leq \psi \quad \equiv \quad \forall (x : \Gamma).(\varphi(x) \Rightarrow \psi(x))$$
The induced equality on formulas is then extensional or, to put it differently, a relation of equiprovability:
$$\varphi = \psi \quad \equiv \quad \forall (x : \Gamma).(\varphi(x) \Leftrightarrow \psi(x))$$
- With these intuitions in mind, the diagonal morphism δ_X is nothing more than the function which duplicates variables, and the first condition simply means that:
$$\forall (x : X).(\top \Rightarrow \varphi(x,x)) \qquad \Leftrightarrow \qquad \forall (x,y : X).(x = y \Rightarrow \varphi(x,y))$$
- As explained in the introduction, since both quantifiers $\exists x : X.$ and $\forall x : X.$ bind the variable x, turning any formula ranging over $\Gamma \times X$ into a formula ranging over Γ, it is natural to interpret them as morphism from $\mathcal{T}(\Gamma \times X)$ to $\mathcal{T}(\Gamma)$. As for their definitions as left and right adjoints of the projection $\pi^1_{\Gamma \times X}$, *i.e.*:
$$\begin{array}{rcl} \varphi \leq \mathcal{T}(\pi^1_{\Gamma\times X})(\psi) & \Leftrightarrow & (\exists X)_\Gamma(\varphi) \leq \psi \\ \mathcal{T}(\pi^1_{\Gamma\times X})(\varphi) \leq \psi & \Leftrightarrow & \varphi \leq (\forall X)_\Gamma(\psi) \end{array}$$
they correspond to the following logical equivalences which characterize them:
$$\begin{array}{rcl} \forall (y : \Gamma, x : X).(\varphi(y,x) \Rightarrow \psi(y)) & \Leftrightarrow & \forall (y : \Gamma).(\exists (x : X).\varphi(y,x)) \Rightarrow \psi(y) \\ \forall (y : \Gamma, x : X).(\varphi(y) \Rightarrow \psi(y,x)) & \Leftrightarrow & \forall (y : \Gamma).\varphi(y) \Rightarrow \forall (x : X).\psi(y,x) \end{array}$$
- Using the equality predicates and the adjoints for first projections, one can show that in fact for every morphism $f : X \rightarrow Y$, $\mathcal{T}(f) : \mathcal{T}(Y) \rightarrow \mathcal{T}(X)$ has left and right adjoints, which for any $y \in Y$ are intuitively given by:
$$\begin{array}{rcl} \exists(f)(\varphi)(y) & \equiv & \exists (x : X).(f(x) = y \wedge \varphi(x)) \\ \forall(f)(\varphi)(y) & \equiv & \forall (x : X).(f(x) = y \Rightarrow \varphi(x)) \end{array}$$

- Finally, the Beck-Chevaley conditions simply express that the quantifiers are compatible with the substitution. For instance, in the left-hand side diagram for the existential quantifier, given $\Gamma, \Gamma', X \in C$ and a morphism $s : \Gamma \to \Gamma'$, the commutation of the diagram requires that:

$$\mathcal{T}(s) \circ (\exists X)_{\Gamma'} \quad = \quad (\exists X)_{\Gamma} \circ (\mathcal{T}(s \times \mathtt{id}_X))$$

  In terms of substitutions, the previous equality is nothing more than the requirement that for any $\varphi \in \mathcal{T}(\Gamma' \times X)$ and any $y' \in \Gamma'$:

$$(\exists (x : X).\varphi(y,x))[y := s(y')] \quad = \quad \exists (x : X).(\varphi(s(y'),x))$$

  The commutation of the other diagram gives the same equality for the universal quantifier.

Remembering the introduction of this chapter, the definition of Kleene's realizability naturally induces a hyperdoctrine structure where each set X is associated to the Heyting algebra $(\mathcal{P}(\mathbb{N})^X, \vdash_X)$. Actually, any complete Heyting algebra gives rise to a hyperdoctrine whose structure is very similar:

Example 9.22 (Hyperdoctrine of a complete Heyting algebra)**.** Let $\mathcal{H}$ be a complete Heyting algebra. The functor $\mathcal{T} : \mathbf{Set}^{op} \to \mathbf{HA}$ given by:

$$\mathcal{T}(X) = \mathcal{H}^X \qquad \text{and} \qquad \mathcal{T}(f) : \left\{ \begin{array}{l} \mathcal{H}^Y \to \mathcal{H}^X \\ g \mapsto (x \mapsto g(f(x))) \end{array} \right. \quad \text{for any } f \in X \to Y$$

defines a hyperdoctrine. The $\mathcal{T}$-predicates are indexed families of elements of $\mathcal{H}$, ordered componentwise. The equality predicates are given by:

$$=_X (x,x') \triangleq \begin{cases} \top & \text{if } x = x' \\ \bot & \text{if } x \neq x' \end{cases}$$

where $\top$ (resp. $\bot$) is the greatest (resp. least) element of $\mathcal{H}$. The adjoints are defined thanks to the completeness of $\mathcal{H}$:

$$(\exists X)_\Gamma(\varphi)(y) = \bigvee_{x \in X} \varphi(y,x) \qquad\qquad (\forall X)_\Gamma(\varphi)(y) = \bigwedge_{x \in X} \varphi(y,x)$$

The Beck-Chevaley conditions are easily verified. In the case of the existential quantifier, for all $\Gamma, \Gamma', X \in C$, any $\varphi \in \mathcal{H}^{\Gamma \times X}$ and any $s : \Gamma \to \Gamma'$, we have:

$$\begin{aligned} (\mathcal{T}(s) \circ (\exists X)_{\Gamma'})(\varphi) \; &= \mathcal{T}(s)(y' \mapsto \textstyle\bigvee_{x \in X} \varphi(y',x)) \\ &= y \mapsto \textstyle\bigvee_{x \in X} \varphi(s(y),x) \\ &= y \mapsto \textstyle\bigvee_{x \in X} \mathcal{T}(s \times \mathtt{id}_X)(\varphi) \\ &= ((\exists X)_\Gamma \circ \mathcal{T}(s \times \mathtt{id}_X))(\varphi) \end{aligned}$$

⌟

Hyperdoctrines are thus tailored to furnish a categorical representation of theories in first-order intuitionistic predicate logic. It was then observed that when a hyperdoctrine has enough structure, the model it gives can be somewhat internalized into a topos[14]. The hyperdoctrines for which this construction is possible were called *triposes* by Hyland, Johnstone and Pitts in [79].

[14] We will not introduce toposes in this thesis. A topos can regarded as a generalization of the category of sets, as such, the set-theoretic foundations of mathematics can expressed in terms of toposes. Toposes are useful structures for the categorical analysis of (high-order) logic. The standard reference for logic interpretation through toposes is Johnstone's book *Sketches of an elephant* [85].

Definition 9.23 (Tripos)**.** A *tripos* over a Cartesian closed category $\mathcal{C}$ is a first-order hyperdoctrine $\mathcal{T} : \mathcal{C}^{op} \to \mathbf{HA}$ which has a *generic predicate*, *i.e.* there exists an object $\mathtt{Prop} \in \mathcal{C}$ and a predicate $\mathtt{tr} \in \mathcal{T}(\mathtt{Prop})$ such that for any object $\Gamma \in \mathcal{C}$ and any predicate $\varphi \in \mathcal{T}(\Gamma)$, there exists a (not necessarily unique) morphism $\chi_\varphi \in \mathcal{C}(\Gamma, \mathtt{Prop})$ such that:

$$\varphi = \mathcal{T}(\chi_\varphi)(\mathtt{tr})$$

⌟

Before giving some examples, we shall say that:

- the object $\mathtt{Prop} \in \mathcal{C}$, as the notation suggests, is the type of *propositions*;
- the generic predicate $\mathtt{tr} \in \mathcal{T}(\mathtt{Prop})$ is the *truth predicate*;
- for each predicate $\varphi \in \mathcal{T}(\Gamma)$, the arrow $\chi_\varphi \in \mathcal{C}(\Gamma, \mathtt{Prop})$ is then a propositional function representing φ, since for any $x \in \Gamma$, we intuitively have:

$$\mathtt{tr}(\chi_\varphi(x)) \equiv \varphi(x)$$

Example 9.24.

1. The example described in the introduction for Kleene's realizability indeed defines a tripos.
2. Given a complete Heyting algebra, the hyperdoctrine given by the functor $\mathcal{T}(X) = \mathcal{H}^X$ (see Example 9.22) is a tripos, with $\mathtt{Prop}$ being defined as (the underlying set of) $\mathcal{H}$, and the truth predicate being given by $\mathtt{tr} \triangleq \mathtt{id}_\mathcal{H} \in \mathcal{T}(\mathcal{H})$.

⌟

9.5 Algebraic structures for (classical) realizability

9.5.1 OCA: ordered combinatory algebras

Finally, we recall in this section the different algebraic structures arising from realizability. We first present the notion of *ordered combinatory algebras*, abbreviated in OCA, which is a variant[15] of Hofstra and Van Oosten's notion of ordered partial combinatory algebras [76].

Definition 9.25 (OCA)**.** An *ordered combinatory algebra* is a quintuple $(\mathcal{A}, \leq, \mathsf{app}, \boldsymbol{k}, \boldsymbol{s})$, which we simply write $\mathcal{A}$, where:

- $\leq$ is a partial order over $\mathcal{A}$,
- $\mathsf{app} : (a,b) \mapsto ab$ is a monotonic function[16] from $\mathcal{A} \times \mathcal{A}$ to $\mathcal{A}$,
- $\boldsymbol{k} \in \mathcal{A}$ is such that $\boldsymbol{k}ab \leq a$ for all $a, b \in \mathcal{A}$,
- $\boldsymbol{s} \in \mathcal{A}$ is such that $\boldsymbol{s}abc \leq ac(bc)$ for all $a, b, c \in \mathcal{A}$.

⌟

Given an ordered combinatory algebra $\mathcal{A}$, we define the set of downward closed subsets of $\mathcal{A}$, which we write $D(\mathcal{A})$:

$$D(\mathcal{A}) \triangleq \{S \subset \mathcal{A} : \forall a \in \mathcal{A}, \forall b \in S, a \leq b \Rightarrow a \in S\}$$

The *standard realizability tripos* on $\mathcal{A}$ is defined by the functor $\mathcal{T}$ which associates to each set $X \in \mathbf{Set}^{op}$ the set of functions $D(\mathcal{A})^X$, which is equipped with the ordering:

$$\varphi \vdash_X \psi \quad \triangleq \quad \exists a \in \mathcal{A}. \forall x \in X. \forall b \in \mathcal{A}. (b \in \varphi(x) \Rightarrow ab \in \psi(x))$$

[15] In partial combinatory algebras, the application is defined as a partial function.

[16] Observe that the application, which is written as a product, is neither commutative nor associative in general.

The type of propositions Prop is defined as $D(\mathcal{A})$ itself and the generic predicate is defined as the identity of $D(\mathcal{A})$. While this definition is standard[17] in the framework of intuitionistic realizability [160]—the reader might in particular recognize the structure underlying the example we gave in the introduction—, its counterpart for classical logic is slightly different.

In his paper [151], Streicher exhibits the notion of *abstract Krivine structure* (which we write AKS), which he shows to be a particular case of OCA. Yet,the so-called *Krivine tripos* he constructs afterwards is defined as a functor mapping any set X to the set of functions $\mathcal{A}^X$ with values in $\mathcal{A}$ (instead of a powerset like $\mathcal{D}(A)$). To this purpose, he considers *filtered ordered combinatory algebras*, which are the given of an OCA with a filter:

Definition 9.26 (Filter)**.** If $\mathcal{A}$ is an OCA, a *filter* over $\mathcal{A}$ is a subset $\Phi \subseteq \mathcal{A}$ such that:

- $\boldsymbol{k} \in \Phi$ and $\boldsymbol{s} \in \Phi$,
- Φ is closed under application, *i.e.* if $a, b \in \Phi$ then $ab \in \Phi$.

⌟

Remark 9.27. It is a well-known fact that Hilbert's combinators K and S are complete with respect to the λ-calculus, in the sense that any closed λ-terms can be encoded as a combination of K and S which is adequate with the β-reduction. Similarly, in an ordered combinatory algebra, any λ-terms t can be encoded as a combination t^* of $\boldsymbol{k}$ and $\boldsymbol{s}$ such that the β-reduction is reflected through the ordering: for any λ-terms $t(x)$ and u, we have[18]:

$$((\lambda x.t)u)^* \leq (t[u/x])^*$$

We shall thus abuse the notation to write closed λ-terms as if they were elements $\mathcal{A}$. Besides, by definition of the notion of filter, any filter Φ contains all the closed λ-terms. ⌟

9.5.2 AKS: abstract Krivine structures

Krivine abstract structures are merely an axiomatization of the Krivine abstract machine viewed as an algebraic structure:

Definition 9.28 (AKS)**.** An *abstract Krivine structure* is a septuple $(\mathbf{\Lambda}, \mathbf{\Pi}, \mathtt{app}, \mathtt{push}, \mathsf{k}_{\_}, \boldsymbol{k}, \boldsymbol{s}, \boldsymbol{cc}, \mathbf{PL}, \perp\!\!\!\perp)$ where:

1. $\mathbf{\Lambda}$ and $\mathbf{\Pi}$ are non-empty sets, respectively called the *terms* and *stacks* of the AKS;
2. $\mathtt{app} : t, u \mapsto tu$ if a function (called *application*) from $\mathbf{\Lambda} \times \mathbf{\Lambda}$ to $\mathbf{\Lambda}$;
3. $\mathtt{push} : t, \pi \mapsto t \cdot \pi$ if a function (called *push*) from $\mathbf{\Lambda} \times \mathbf{\Pi}$ to $\mathbf{\Pi}$;
4. $\mathsf{k}_{\_} : \pi \mapsto \mathsf{k}_\pi$ if a function from $\mathbf{\Pi}$ to $\mathbf{\Lambda}$ (k_π is called a *continuation*);
5. $\boldsymbol{k}$, $\boldsymbol{s}$ and $\boldsymbol{cc}$ are three distinguished terms of $\mathbf{\Lambda}$;
6. $\perp\!\!\!\perp \subseteq \mathbf{\Lambda} \times \mathbf{\Pi}$ (called the *pole*) is a relation between terms and stacks, also written $t \star \pi \in \perp\!\!\!\perp$. This relation fulfills the following axioms for all terms $t, u, v \in \mathbf{\Lambda}$ and all stacks $\pi, \pi' \in \mathbf{\Lambda}$:

$$\begin{array}{rcc}
tu \star \pi \in \perp\!\!\!\perp & \text{whenever} & t \star u \cdot \pi \in \perp\!\!\!\perp \\
\boldsymbol{k} \star t \cdot u \cdot \pi \in \perp\!\!\!\perp & \text{whenever} & t \star \pi \in \perp\!\!\!\perp \\
\boldsymbol{s} \star t \cdot u \cdot v \cdot \pi \in \perp\!\!\!\perp & \text{whenever} & tv(uv) \star \pi \in \perp\!\!\!\perp \\
\boldsymbol{cc} \star t \cdot \pi \in \perp\!\!\!\perp & \text{whenever} & t \star \mathsf{k}_\pi \cdot \pi \in \perp\!\!\!\perp \\
\mathsf{k}_\pi \star t \cdot \pi' \in \perp\!\!\!\perp & \text{whenever} & t \star \pi \in \perp\!\!\!\perp
\end{array}$$

[17] To be exact, the very central notion is the one of *partial combinatory algebras* [160], which is not ordered and where app is defined as a partial function. In this case, the tripos associates to each sets the set of functions $\mathcal{P}(\mathcal{A})^X$ with values in the powerset of $\mathcal{A}$ rather than in $D(A)$.

[18] See [45] for instance for a proof.

7. $\mathbf{PL} \subseteq \mathbf{\Lambda}$ is a subset of $\mathbf{\Lambda}$ (whose elements are called the *proof-like terms*), which contains $\boldsymbol{k}, \boldsymbol{s}, \boldsymbol{cc}$ and is closed under application.

⌟

It is obvious that any realizability model (in the sense given in Chapter 3) induces an abstract Krivine structure. In fact, almost all the definitions that we used in the previous chapters when defining realizability interpretations can be restated in terms of abstract Krivine structures. Given any subset of stacks $X \subseteq \mathbf{\Pi}$ (which we call a *falsity value*), we write $X^{\perp\!\!\!\perp}$ for its orthogonal set with respect to the pole:

$$X^{\perp\!\!\!\perp} \triangleq \{t \in \mathbf{\Lambda} : \forall \pi \in X, t \star \pi \in \perp\!\!\!\perp\}$$

Orthogonality for subsets $X \subseteq \mathbf{\Lambda}$ (*i.e.* a *truth value*) is defined identically. As usual we write $t \perp\!\!\!\perp \pi$ for $t \star \pi \in \perp\!\!\!\perp$ and $t \perp\!\!\!\perp X$ (resp. $X \perp\!\!\!\perp \pi$) for $t \in X^{\perp\!\!\!\perp}$ (resp. $\pi \in X^{\perp\!\!\!\perp}$). The set of falsity values closed under bi-orthogonality is then defined by:

$$\mathcal{P}_{\perp\!\!\!\perp}(\mathbf{\Pi}) \triangleq \{X \in \mathcal{P}(\mathbf{\Pi}) : X = X^{\perp\!\!\!\perp\perp\!\!\!\perp}\}$$

With these definitions, from any abstract Krivine structure can be constructed a filtered ordered combinatory algebra:

Proposition 9.29 (From AKS to OCA). *If* $(\mathbf{\Lambda}, \mathbf{\Pi}, \mathsf{app}, \mathsf{push}, \mathsf{k}\_, \boldsymbol{k}, \boldsymbol{s}, \boldsymbol{cc}, \mathbf{PL}, \perp\!\!\!\perp)$ *is an abstract Krivine structure, then the quintuple* $(\mathcal{P}_{\perp\!\!\!\perp}(\mathbf{\Pi}), \leq, \mathsf{app}', \{\boldsymbol{k}\}^{\perp\!\!\!\perp}, \{\boldsymbol{s}\}^{\perp\!\!\!\perp})$ *is an OCA, with:*

- $X \leq Y \triangleq X \supseteq Y$
- $\mathsf{app}'(X, Y) \triangleq \{\pi \in \Pi : \forall t \in Y^{\perp\!\!\!\perp}. t \cdot \pi \in X\}^{\perp\!\!\!\perp\perp\!\!\!\perp}$

Besides, $\Phi \triangleq \{X \in \mathcal{P}_{\perp\!\!\!\perp}(\Pi) : \exists t \in \mathbf{PL}. t \perp\!\!\!\perp X\}$ *defines a filter for this OCA.*

Proof. See [151] or [45]. □

Given a filtered ordered combinatory algebra $(\mathcal{A}, \Phi)$, one can define the functor $\mathcal{T} : \mathbf{Set}^{op} \to \mathcal{A}$:

$$\mathcal{T}(X) = \mathcal{A}^X \qquad \text{and} \qquad \mathcal{T}(f) : \left\{ \begin{array}{l} \mathcal{A}^Y \to \mathcal{A}^X \\ g \mapsto (x \mapsto g(f(x))) \end{array} \right. \quad \text{for any } f \in X \to Y$$

endowed with the following *entailment* relation:

$$\varphi \vdash_X \psi \triangleq \exists a \in \Phi. \forall x \in X. a\varphi(x) \leq \psi(x) \qquad \text{(for all } X \in \mathbf{Set})$$

In such a case, we shall refer to a as a *realizer.* It is easy to show that the entailment relation $\vdash_X$ actually defines an order relation on $\mathcal{T}(X)$. Therefore, this functor always defines what is called an indexed preorder. In the particular case where the filtered OCA arises from an AKS, it can even be shown that the functor $\mathcal{T}$ actually defines a tripos, which Streicher calls a *Krivine tripos* [151, Theorem 5.10].

9.5.3 $^{\mathcal{I}}$OCA: implicative ordered combinatory algebras

In the continuity of Streicher's work, Ferrer *et al.* defined a subclass of ordered combinatory algebras which possess precisely the additional structure necessary to make of the previous functor a tripos [45]. These algebras, which they call *Krivine ordered combinatory algebras* ($^{\mathcal{K}}$OCA), thus provide us with an algebraic interpretation of Krivine classical realizability. It turns out that they are naturally definable as a particular case of a slightly more general class of algebras, called *implicative ordered combinatory algebras* ($^{\mathcal{I}}$OCA). As we shall see, a $^{\mathcal{K}}$OCA, which is the classical counterpart of an $^{\mathcal{I}}$OCA, is obtained by adding to the latter a combinator corresponding to the usual `call/cc` operator.

Definition 9.30 (${}^{\mathcal{I}}$OCA)**.** An *implicative ordered combinatory algebra* consists of an octuple of the shape $(\mathcal{A},\leq,\mathtt{app},\mathtt{imp},\boldsymbol{k},\boldsymbol{s},\boldsymbol{e},\Phi)$, which we simply write $\mathcal{A}$ or $(\mathcal{A},\Phi)$, where:

- $\leq$ is a partial order over $\mathcal{A}$, and $\mathcal{A}$ is meet-complete as a poset;
- $\mathtt{app} : (a,b) \mapsto ab$ is a monotonic function from $\mathcal{A}\times A$ to $\mathcal{A}$,
- $\mathtt{imp} : a,b \mapsto a \to b$ is a monotonic function from $\mathcal{A}^{op}\times\mathcal{A}\to\mathcal{A}$ (*i.e.* $\mathtt{imp}$ is monotonic in its second component, antitonic in the first);
- $\Phi\subseteq\mathcal{A}$ is a *filter*, closed by application and such that $\boldsymbol{k},\boldsymbol{s},\boldsymbol{e}\in\Phi$;
- the following holds for all $a,b,c\in\mathcal{A}$:
  - $\boldsymbol{k}ab\leq a$
  - $\boldsymbol{s}abc\leq ac(bc)$
  - if $a\leq b\to c$ then $ab\leq c$
  - if $ab\leq c$ then $\boldsymbol{e}a\leq b\to c$

⌟

Observe that in particular, any ${}^{\mathcal{I}}$OCA is a filtered OCA. The extra requirement of an arrow, as the reader might have guessed, equips the sets $(\mathcal{A}^X,\vdash_X)$ with a structure of Heyting algebra. In other words, when $\mathcal{A}$ is an ${}^{\mathcal{I}}$OCA, the functor $\mathcal{T} : X\mapsto\mathcal{A}^X$ is a tripos. Indeed, thanks to combinatorial completeness of $\boldsymbol{k}$ and $\boldsymbol{s}$, we can define a meet through the usual encoding of pairs in λ-calculus. We define:

$$\mathsf{t}\triangleq\lambda xy.x \qquad \mathsf{f}\triangleq\lambda xy.y \qquad \mathsf{p}\triangleq\lambda xyz.zxy \qquad \mathsf{p}_0\triangleq\lambda x.(x\mathsf{t}) \qquad \mathsf{p}_1\triangleq\lambda x.(x\mathsf{f})$$

which ensures that $\mathsf{p}_0(\mathsf{p}ab)\leq a$ and $\mathsf{p}_1(\mathsf{p}ab)\leq b$. This allows us to define a map $\wedge:\mathcal{A}\times\mathcal{A}\to\mathcal{A}$ by $a\wedge b\triangleq\mathsf{p}ab$. As for the arrow, the imp operations naturally induces an arrow on formulas such that for any $X\in\mathbf{Set}$, and any $\varphi,\psi,\theta\in\mathcal{A}^X$, we have:

$$\varphi\vdash_X\psi\to\theta \qquad \text{if and only if} \qquad \varphi\wedge\psi\vdash_X\theta$$

Since we believe it might help the reader to see the connection with realizability, we sketch the proof of this statement. From left to right, the implication is trivial since if there exists $u\in\Phi$ such that for all $a\in\varphi(x),b\in\psi(x)$ and $c\in\theta(x)$, $ua\leq b\to c$, then by definition of the arrow $(ua)b\leq c$. Therefore, we can define the realizer $r\triangleq\lambda x.(xu)$ which belongs to Φ and verifies that $r(\mathsf{p}ab)\leq c$.

From right to left, the proof is very similar: if there exists $u\in\Phi$ such that for all $a\in\varphi(x),b\in\psi(x)$ and $c\in\theta(x)$, $u(\mathsf{p}ab)\leq c$, in particular we have $(\lambda y.u(\mathsf{p}ay))b\leq c$. Therefore, by definition of the arrow, we have that $\boldsymbol{e}(\lambda y.u(\mathsf{p}ay))\leq b\to c$ and thus $\lambda x.\boldsymbol{e}(\lambda y.u(\mathsf{p}xy))$ is the expected realizer.

The complete proof that the functor $\mathcal{T}$ is a tripos can be found in [45].

9.5.4 ${}^{\mathcal{K}}$OCA: Krivine ordered combinatory algebras

This notion of ${}^{\mathcal{I}}$OCA can be slightly enforced to obtain the notion of *Krivine ordered combinatory algebras*, that should be simply understood as the usual addition of `call/cc` to go from an intuitionistic setting to the classical one:

Definition 9.31 (${}^{\mathcal{K}}$OCA)**.** A *Krivine ordered combinatory algebra* is an implicative combinatory algebra equipped with a distinguished element $\boldsymbol{c}\in\Phi$ such that for all $a,b\in\mathcal{A}$:

$$\boldsymbol{c}\leq((a\to b)\to a)\to a$$

⌟

Example 9.32. Any complete Boolean algebra $\mathcal{B}$ induces a ${}^{\mathcal{K}}$OCA by defining:

$$ab\triangleq a\wedge b \qquad a\to b\triangleq\neg a\vee b \qquad \Phi\triangleq\{\top\} \qquad \boldsymbol{s}\triangleq\boldsymbol{k}\triangleq\boldsymbol{e}\triangleq\boldsymbol{c}\triangleq\top$$

Broadly, Boolean algebras are trivial ${}^{\mathcal{K}}$OCA where all the realized elements are collapsed to $\top$. ⌟

Interestingly, any abstract Krivine structure gives rise to a Krivine ordered combinatory algebra, and vice-versa. In both cases, the induced triposes (by the AKS and the ${}^{\mathcal{K}}$OCA) are equivalent. This justifies the claim that the latter indeed captures the necessary additional structure that allows an OCA induced from an AKS to be a tripos. These results are a refinement of Proposition 9.29:

Proposition 9.33 (From AKS to ${}^{\mathcal{K}}$OCA)**.** *If* $(\mathbf{\Lambda},\mathbf{\Pi},\mathtt{app},\mathtt{push},\mathtt{k}_{\_},\boldsymbol{k},\boldsymbol{s},\boldsymbol{cc},\mathbf{PL},\perp\!\!\!\perp)$ *is an abstract Krivine structure, then the nonuple* $(\mathcal{P}_{\perp\!\!\!\perp}(\mathbf{\Pi}),\leq,\mathtt{app}',\mathtt{imp}',\{\boldsymbol{k}\}^{\perp\!\!\!\perp},\{\boldsymbol{s}\}^{\perp\!\!\!\perp},\{\boldsymbol{cc}\}^{\perp\!\!\!\perp},\{\boldsymbol{e}\}^{\perp\!\!\!\perp},\Phi)$ *is a* ${}^{\mathcal{K}}$*OCA, with:*

- $X \leq Y \;\triangleq\; X \supseteq Y$;
- $\mathtt{app}'(X,Y) \triangleq \{\pi \in \Pi : \forall t \in Y^{\perp\!\!\!\perp}. t\cdot\pi \in X\}^{\perp\!\!\!\perp\perp\!\!\!\perp}$;
- $\mathtt{imp}'(X,Y) \triangleq \{t\cdot\pi \in \Pi : t \in X^{\perp\!\!\!\perp} \wedge \pi \in Y\}^{\perp\!\!\!\perp\perp\!\!\!\perp}$;
- $\boldsymbol{e} \triangleq \boldsymbol{s}(\boldsymbol{k}(\boldsymbol{skk}))$;

Besides, $\Phi \triangleq \{X \in \mathcal{P}_{\perp\!\!\!\perp}(\Pi) : \exists t \in \mathbf{PL}. t \perp\!\!\!\perp X\}$ *defines a filter for this OCA.*

Proposition 9.34 (From ${}^{\mathcal{K}}$OCA to AKS)**.** *If* $(\mathcal{A},\leq,\mathtt{app}_{\mathcal{A}},\mathtt{imp}_{\mathcal{A}},\boldsymbol{k},\boldsymbol{s},\boldsymbol{c},\boldsymbol{e},\Phi)$ *is a* ${}^{\mathcal{K}}$*OCA, then the septuple defined by* $(\mathcal{A},\mathcal{A},\mathtt{app},\mathtt{push},\mathtt{k}_{\_},\kappa,s,c,\mathbf{PL},\perp\!\!\!\perp)$ *is an abstract Krivine structure, where:*

- $\perp\!\!\!\perp \triangleq\leq$ i.e. $t \perp\!\!\!\perp \pi \triangleq t \leq \pi$;
- $\mathtt{app}(t,u) \triangleq \mathtt{app}_t(t,u) = tu$;
- $\mathtt{push}(t,\pi) \triangleq \mathtt{imp}(t,\pi) = t\cdot\pi$;
- $\mathtt{k}_\pi \triangleq \pi \to \bot$;
- $\mathbf{PL} \triangleq \Phi$;
- $\kappa \triangleq \boldsymbol{e}(\boldsymbol{bek})$, $s \triangleq \boldsymbol{e}(\boldsymbol{b}(\boldsymbol{be}(\boldsymbol{be}))\boldsymbol{s})$, $c \triangleq \boldsymbol{e}\,\boldsymbol{c}$,

where $\boldsymbol{b}$ *is an abbreviation for* $\boldsymbol{s}(\boldsymbol{ks})\boldsymbol{k}$.

Proof. See [45, Theorem 5.11] for the first proposition, [45, Theorem 5.13] for the second. □

Without considering in details the proofs of the correspondences between AKS and ${}^{\mathcal{K}}$OCA or their associated triposes, it is worth noting that when going from a ${}^{\mathcal{K}}$OCA $\mathcal{A}$ to a AKS, both sets $\mathbf{\Lambda}$ and $\mathbf{\Pi}$ are defined as $\mathcal{A}$. This means in particular that realizers and their opponents live in the same world, and the orthogonality relation is simply reflected by the order. That is $t \perp\!\!\!\perp \pi$ if $t \leq \pi$, and more generally if $X \subseteq \mathcal{P}(\mathbf{\Pi})$, $t \perp\!\!\!\perp X$ if for any $x \in X$, $t \leq x$. If, as advocated in Section 9.2, we identify a closed formula A with its falsity values $\|A\|$, we recover the intuition that $t \Vdash A$ is reflected by the ordering $t \leq \|A\|$. With these ideas in mind, we are now ready to see the more general notion of implicative algebra.

10- Implicative algebras

In this chapter, we present Alexandre Miquel's *implicative algebras*[1], which aim at providing an algebraic framework for classical realizability. We first introduce the notion of *implicative structures* on which implicative algebras rely. Then, we will show that most of the structures we introduced in Chapter 9 (Complete Heyting/Boolean algebras, AKSs, OCAs) are particular cases of implicative structures. Next, we show how to embed both the λ_c-calculus in a manner which is adequate with its second-order type system. Finally, we introduce the notion of separators and implicative algebras, and show how they induce realizability triposes.

Most of the results in this chapter are supported by a Coq development$^\bullet$ *[122]. All along the chapter, we use the bullet to denote the statements that are formalized.*

10.1 Implicative structures

10.1.1 Definition

Intuitively, *implicative structures* are tailored to represent both the formulas of second-order logic and realizers arising from Krivine's λ_c-calculus. We shall see in the sequel how they indeed allow us to define λ-terms, but let us introduce them by focusing on their logical facet. We are interested in formulas of second-order logic, that is to say of system F, which are defined by a simple grammar:

$$A, B ::= X \mid A \Rightarrow B \mid \forall X.A$$

Implicative structures are therefore defined as meet-complete lattices (for the universal quantification) with an internal binary operation satisfying the properties of the implication:

Definition$^\bullet$ 10.1. An *implicative structure* is a complete meet-semilattice $(\mathcal{A}, \preccurlyeq)$ equipped with a binary operation $(a, b) \mapsto (a \to b)$, called the *implication* of $\mathcal{A}$, that fulfills the following axioms:

1. Implication is anti-monotonic with respect to its first operand and monotonic with respect to its second operand, in the sense that for all $a, a_0, b, b_0 \in \mathcal{A}$:

   (Variance) $\quad$ if $a_0 \preccurlyeq a$ and $b \preccurlyeq b_0$ then $(a \to b) \preccurlyeq (a_0 \to b_0)$

2. Arbitrary meets distribute over the second operand of implication, in the sense that for all $a \in \mathcal{A}$ and for all subsets $B \subseteq \mathcal{A}$:

   (Distributivity) $$\bigcurlywedge_{b \in B} (a \to b) = a \to \bigcurlywedge_{b \in B} b$$

⌟

[1] We insist on the fact that all the results presented in this chapters are his. Most of them are given in [121]. Independently, structures that are very similar to implicative structures can be found in Frédéric Ruyer's Ph.D. thesis [147] under the name of *applicative lattices*.

Remark 10.2. 1. The distributivity axiom of implicative structures should not be confused with the property of distributivity for lattices (see the definition of Boolean algebras). In general, the underlying lattice of an implicative structure does not have to be distributive.

2.$^\bullet$ In the particular case where $B = \emptyset$, the axiom of distributivity states that $a \to \top = \top$ for all $a \in \mathcal{A}$. ⌟

10.1.2 Examples of implicative structures

10.1.2.1 Complete Heyting algebras

The first example of implicative structures is given by complete Heyting algebras. Indeed, the axioms of implicative structures are intuitionistic tautologies verified by any complete Heyting algebra:

Proposition 10.3. *If $(\mathcal{H}, \preccurlyeq, \to)$ is a complete Heyting algebra, then for all $a, a', b, b', c \in \mathcal{H}$ and for all subsets $B \subseteq \mathcal{H}$, the following holds:*

1.$^\bullet$ *if $a \preccurlyeq a'$, then $a' \to b \preccurlyeq a \to b$;*

2.$^\bullet$ *if $b \preccurlyeq b'$, then $a \to b \preccurlyeq a \to b'$;*

3.$^\bullet$ *$a \curlywedge c \preccurlyeq b \Leftrightarrow a \preccurlyeq c \to b$*

4.$^\bullet$ *$a \to \bigcurlywedge_{b \in B} b = \bigcurlywedge_{b \in B}(a \to b)$.*

Proof. Observe first that since $\mathcal{H}$ is complete, by definition we have $a \to b = \bigcurlyvee\{x \in \mathcal{H} : a' \wedge x \preccurlyeq b\}$.

1. Let $a, a', b \in \mathcal{H}$ be fixed. Using this observation above for $a' \to b$, it suffices to show that $a \to b$ is an upper bound of the set $\{x \in \mathcal{H} : a' \curlywedge x \preccurlyeq b\}$. Let then $x \in \mathcal{H}$ be such that $a' \curlywedge x \preccurlyeq b$. To show that $x \preccurlyeq a \to b$, it suffices to show that $a \curlywedge x \preccurlyeq b$. This follows from the transitivity of the order: $a \curlywedge x \preccurlyeq a' \curlywedge x \preccurlyeq b$.

2. Similar to 1.

3. Let $a, b, c \in \mathcal{H}$ be fixed. The left-to-right implication is trivial from the observation above. From right to left, we show that $a \curlywedge c \preccurlyeq c \curlywedge (c \to b) \preccurlyeq b$. The first inequality follows from the monotonicity of $\curlywedge$, the second one follows from the definition of $c \to b$.

4. Let $a \in \mathcal{H}$ and $B \subseteq \mathcal{H}$ be fixed. By definition, this amounts to showing that:

$$\bigcurlyvee\{x \in \mathcal{H} : a \curlywedge x \preccurlyeq \bigcurlywedge_{b \in B} b\} = \bigcurlywedge_{b \in B} \bigcurlyvee\{x \in \mathcal{H} : a \curlywedge x \preccurlyeq b\}$$

   which we show by anti-symmetry. To show that the term on the left hand-side term is inferior to the one on the right-hand side, it suffices to show that $\bigcurlyvee\{x \in \mathcal{H} : a \curlywedge x \preccurlyeq \bigcurlywedge_{b \in B} b\} \preccurlyeq a \to b$ for any $b \in B$. Let thus $x \in \mathcal{H}$ be such that $a \bigcurlywedge x \preccurlyeq \bigcurlywedge_{b \in B} b$, we need to show that $x \preccurlyeq a \to b$. This follows from the third item and the inequality $a \curlywedge x \preccurlyeq \bigcurlywedge_{b \in B} b \preccurlyeq b$. The converse inequality is proved similarly.

□

We deduce that every complete Heyting algebra induces an implicative structure with the same arrow:

Proposition$^\bullet$ 10.4. *Every complete Heyting algebra is an implicative structure.*

The converse is obviously false, since the implication of an implicative structure $\mathcal{A}$ is in general not determined by the lattice structure of $\mathcal{A}$.

10.1.2.2 Complete Boolean algebras

Since any (complete) Boolean algebra is in particular a (complete) Heyting algebra, *a fortiori* any complete Boolean algebra induces an implicative structure:

Proposition• 10.5. *If $\mathcal{B}$ is a (complete) Boolean algebra, then $\mathcal{B}$ is a (complete) Heyting algebra where the implication is defined for all $a,b \in \mathcal{B}$ by $a \to b \triangleq (\neg a) \curlyvee b$.*

Proof. Let $a,b \in \mathcal{B}$ be fixed. We show that $(\neg a) \curlyvee b$ is the supremum of $\{x \in \mathcal{B} : a \curlywedge x \preccurlyeq b\}$, *i.e.* that it belongs to this set and that it is an upper bound of the same. The first part is trivial, since the distributivity implies that $a \curlywedge (\neg a \curlyvee b) = (a \curlywedge \neg a) \curlyvee (a \curlywedge b) = a \curlywedge b \preccurlyeq b$. For the second part of the statement, let $c \in \mathcal{B}$ be such that $a \curlywedge c \preccurlyeq b$. Then we have: $c = (c \curlywedge \neg a) \curlyvee (c \curlywedge a) \preccurlyeq (c \curlywedge \neg a) \curlyvee b \preccurlyeq \neg a \curlyvee b$, which concludes the proof. □

Proposition• 10.6. *If $\mathcal{B}$ is a (complete) Boolean algebra, then $\mathcal{B}$ induces an implicative structure where the implication is defined for all $a,b \in \mathcal{B}$ by $a \to b \triangleq \neg a \curlyvee b$.*

10.1.2.3 Dummy structures

Given a complete lattice $\mathcal{L}$, there are at least two possible definitions of dummy implicative structures:

Proposition 10.7. *If $\mathcal{L}$ is a complete lattice, the following definitions give rise to implicative structures:*

1.• $a \to b \triangleq \top$ for all $a,b \in \mathcal{L}$

2.• $a \to b \triangleq b$ for all $a,b \in \mathcal{L}$

Proof. Trivial in both cases. □

Both definitions induce implicative structures which are meaningless from the point of view of logic. Nonetheless, they will provide us with useful counter-examples.

10.1.2.4 Ordered combinatory algebras

Any ordered combinatory algebra (see Definition 9.25) also induces an implicative structure, whose definition is related with the definition of the realizability tripos. Indeed, remember that given an OCA $\mathcal{A}$ and a set X, the ordering on predicates of $\mathcal{P}(\mathcal{A})^X$ is defined by:

$$\varphi \vdash_X \psi \quad \triangleq \quad \exists r \in \mathcal{A}.\forall x \in X.\forall a \in \mathcal{A}.(a \in \varphi(x) \Rightarrow ra \in \psi(x))$$

where r is broadly a *realizer* of $\forall x \in X.\varphi(x) \Rightarrow \psi(x)$. Similarly, we can define an implication on the complete lattice $\mathcal{P}(\mathcal{A})$ which give rise to an implicative structure:

Proposition 10.8. *If $\mathcal{A}$ is an ordered combinatory algebra, then the complete lattice $\mathcal{P}(\mathcal{A})$ equipped with the implication:*

$$A \to B \quad \triangleq \quad \{r \in \mathcal{A} : \forall a \in A. ra \in B\} \qquad\qquad (\forall A,B \subseteq \mathcal{A})$$

is an implicative structure

Proof. Both conditions (variance and distributivity) are trivial from the definition. □

In particular, the powerset of any ${}^{\mathcal{I}}$OCA or ${}^{\mathcal{K}}$OCA induces an implicative structure with the same construction.

10.1.2.5 Implicative structure of classical realizability

Our final example of implicative structure—which is the main motivation of this work—is given by classical realizability. As we saw in Chapter 9, the construction of classical realizability models, whether it be from Krivine's realizability algebras [98, 99, 100] in a set-theoretic like fashion or in Streicher's AKS [151], takes place in a structure of the form $(\Lambda, \Pi, \cdot, \perp\!\!\!\perp)$ where:

- Λ is the set of realizers;
- Π is the set of stacks (or opponents);
- $(\cdot) : \Lambda \times \Pi \to \Pi$ is a binary operation for pushing a realizer onto a stack;
- $\perp\!\!\!\perp \subseteq \Lambda \times \Pi$ is the pole.

Given such a quadruple, we can define:

- $\mathcal{A} \triangleq \mathcal{P}(\Pi)$;
- $a \preccurlyeq b \triangleq a \supseteq b$ (for all $a, b \in \mathcal{A}$)
- $a \to b \triangleq a^{\perp\!\!\!\perp} \cdot b = \{t \cdot \pi : t \in a^{\perp\!\!\!\perp}, \pi \in b\}$ (for all $a, b \in \mathcal{A}$)

where as usual $a^{\perp\!\!\!\perp}$ is $\{t \in \Lambda : \forall \pi \in a, (t, \pi) \in \perp\!\!\!\perp\} \in \mathcal{P}(\Lambda)$, the orthogonal set of $a \in \mathcal{P}(\Pi)$ with respect to the pole $\perp\!\!\!\perp$. Here again, it is easy to verify that this defines an implicative structure.

Proposition 10.9. *The triple $(\mathcal{A}, \preccurlyeq, \to)$ is an implicative structure.*

Proof. The proof is again trivial. Variance conditions correspond to the usual monotonicity of truth and falsity values, while the distributivity follows directly by unfolding the definitions. □

Remark 10.10. 1. Actually, in this particular case the implication satisfies two additional laws:

$$(\bigwedge_{a \in A} a) \to b = \bigvee_{a \in A} (a \to b) \qquad \text{and} \qquad a \to (\bigvee_{b \in B} b) = \bigvee_{b \in B} (a \to b)$$

for all $a, b \in \mathcal{A}, A, B \subseteq \mathcal{A}$. These extra properties also follow directly from the definition, however, they are almost never used in classical realizability.

2. Unlike Streicher's definition of the OCA used for the construction of Krivine's tripos (see Proposition 9.29), where $\mathcal{A}$ is defined as $\mathcal{P}_{\perp\!\!\!\perp}(\Pi)$, we consider $\mathcal{A}$ to be all the sets of $\mathcal{P}(\Pi)$. In this sense, we are in line with Krivine's usual definitions, where falsity values are not necessarily closed by double orthogonal. We will see that this presents an advantage over Streicher's OCAs (and thus Ferrer *et al.* ${}^{\mathcal{I}}$OCAs and ${}^{\mathcal{K}}$OCAs), namely that we will have the full adjunction:

$$a \leq b \to c \quad \Leftrightarrow \quad ab \leq c \qquad\qquad (\forall a, b, c \in \mathcal{A})$$

On the contrary, in ${}^{\mathcal{I}}$OCAs and ${}^{\mathcal{K}}$OCAs an adjunctor $\boldsymbol{e}$ is required for the right-to-left implication, which becomes:

$$ab \leq c \quad \Rightarrow \quad \boldsymbol{e}a \leq b \to c \qquad\qquad (\forall a, b, c \in \mathcal{A})$$

⌟

10.2 Interpreting the λ-calculus

10.2.1 Interpretation of λ-terms

We motivated the definition of implicative structures with the aim of obtaining a common framework for the interpretation both of types and programs. We shall now see how λ-terms can indeed be defined in implicative structures.

From now on, let $\mathcal{A} = (\mathcal{A}, \preccurlyeq, \rightarrow)$ denotes an arbitrary implicative structure.

Definition$^\bullet$ 10.11 (Application). Given two elements $a, b \in \mathcal{A}$, we call the *application* of a to b and write ab the element of $\mathcal{A}$ that is defined by

$$ab \triangleq \bigwedge\{c \in \mathcal{A} : a \preccurlyeq (b \rightarrow c)\}.$$

As usual, we write $ab_1b_2 \cdots b_n$ for $((ab_1)\, b_2) \cdots b_n$ (for all $a, b_1, b_2, \ldots, b_n \in \mathcal{A}$). ⌟

If we think of the order relation $a \preccurlyeq b$ as "a is more precise than b", the above definition actually defines the application ab as the meet of all the elements c such that $b \rightarrow c$ is an approximation of a. This definition fulfills the usual properties of the λ-calculus:

Proposition 10.12 (Properties of application). *For all $a, a', b, b', c \in \mathcal{A}$:*

1.$^\bullet$ *If $a \preccurlyeq a'$ and $b \preccurlyeq b'$, then $ab \preccurlyeq a'b'$* *(Monotonicity)*
2.$^\bullet$ $(a \rightarrow b)a \preccurlyeq b$ *(β-reduction)*
3.$^\bullet$ $a \preccurlyeq (b \rightarrow ab)$ *(η-expansion)*
4.$^\bullet$ $ab = \min\{c \in A : a \preccurlyeq (b \rightarrow c)\}$ *(Minimum)*
5.$^\bullet$ $ab \preccurlyeq c \quad \Leftrightarrow \quad a \preccurlyeq (b \rightarrow c)$ *(Adjunction)*

Proof. For all $a, b \in \mathcal{A}$, let us write $\mathsf{App}_{a,b} = \{c \in \mathcal{A} : a \preccurlyeq (b \rightarrow c)\}$, so that $ab = \bigwedge \mathsf{App}_{a,b}$.

1. We prove the monotonicity w.r.t. to the left operand a, the monotonicity w.r.t. to the right one is very similar. Let a, a', b be elements of $\mathcal{A}$, and assume that $a \preccurlyeq a'$. We want to prove:

$$\bigwedge \mathsf{App}_{a,b} \preccurlyeq \bigwedge \mathsf{App}_{a',b}$$

It is thus enough to show that $\mathsf{App}_{a,b} \subseteq \mathsf{App}_{a',b}$, which is trivial.

2. For any $a, b \in \mathcal{A}$, we have by definition that $b \in \mathsf{App}_{a\rightarrow b,a}$, thus $\bigwedge \mathsf{App}_{a\rightarrow b,a} \preccurlyeq b$.
3. Let a, b be elements of $\mathcal{A}$. By distributivity, we have $b \rightarrow \bigwedge \mathsf{App}_{a,b} = \bigwedge\{b \rightarrow c : c \in \mathsf{App}_{a,b}\}$. To prove the desired inequality, it is enough to show that for any $c \in \mathsf{App}_{a,b}$, we have $a \preccurlyeq b \rightarrow c$, which is a tautology.
4. Follows from 3.
5. From left to right, we prove that $a \preccurlyeq (b \rightarrow ab) \preccurlyeq (b \rightarrow c)$ using 3 and the covariance of the implication. From right to left, it is clear that if $c \in \mathsf{App}_{a,b}$, then $ab = \bigwedge \mathsf{App}_{a,b} \preccurlyeq c$.

□

Remark$^\bullet$ 10.13 (Galois connection). The adjunction $ab \preccurlyeq c \Leftrightarrow a \preccurlyeq (b \rightarrow c)$ expresses the existence of a family of Galois connections $f_b \dashv g_b$ indexed by all $b \in \mathcal{A}$, where the left and right adjoints $f_b, g_b : \mathcal{A} \rightarrow \mathcal{A}$ are defined by:

$$f_b : a \mapsto ab \qquad \text{and} \qquad g_b : c \mapsto (b \rightarrow c) \qquad\qquad \text{(for all } a, b, c \in \mathcal{A})$$

Recall that in a Galois connection, the left adjoint is fully determined by the right one (and vice-versa, see Proposition 9.20). In the particular case of a complete Heyting algebra $(\mathcal{H}, \preccurlyeq, \rightarrow)$, this implies that the application is characterized by $ab = a \curlywedge b$ for all $a, b \in \mathcal{H}$. Indeed, in any Heyting algebra, the adjunction $a \curlywedge b \preccurlyeq c \Leftrightarrow a \preccurlyeq (b \rightarrow c)$ holds for all $a, b, c \in \mathcal{H}$ (Proposition 10.3), by uniqueness of the left adjoint, ab and $a \curlywedge b$ are thus equal. ⌟

Definition$^\bullet$ 10.14 (Abstraction). Given a function $f : \mathcal{A} \rightarrow \mathcal{A}$, we call *abstraction* of f and write λf the element of $\mathcal{A}$ defined by:

$$\lambda f \triangleq \bigcurlywedge_{a \in \mathcal{A}} (a \rightarrow f(a))$$

⌟

Once again, if we think of the order relation $a \preccurlyeq b$ as "a is more precise than b", the meet of the elements of a set S is an element containing the union of all the informations given by the elements of S. With this in mind, the above definition sets λf as the union of all the step functions $a \rightarrow f(a)$. This definition, together with the definition of the application, fulfills again properties expected from the λ-calculus:

Proposition 10.15 (Properties of the abstraction). *The following holds for any $f, g : \mathcal{A} \rightarrow \mathcal{A}$:*

1.$^\bullet$ *If for all $a \in \mathcal{A}$, $f(a) \preccurlyeq g(a)$, then $\lambda f \preccurlyeq \lambda g$.* *(Monotonicity)*
2.$^\bullet$ *For all $a \in \mathcal{A}$, $(\lambda f)a \preccurlyeq f(a)$.* *($\beta$-reduction)*
3.$^\bullet$ *For all $a \in \mathcal{A}$, $a \preccurlyeq \lambda(x \mapsto ax)$.* *($\eta$-expansion)*

Proof. Let $a \in \mathcal{A}$ be fixed.

1. By hypothesis, we have for all $b \in \mathcal{A}$ that $\bigcurlywedge_{a \in \mathcal{A}}(a \rightarrow f(a)) \preccurlyeq b \rightarrow f(b) \preccurlyeq b \rightarrow g(b)$. We can thus conclude that $\lambda f = \bigcurlywedge_{a \in \mathcal{A}}(a \rightarrow f(a)) \preccurlyeq \bigcurlywedge_{a \in \mathcal{A}}(a \rightarrow g(a)) = \lambda g$.
2. By definition of the application, in order to show that $(\lambda f)a \preccurlyeq f(a)$ it is enough to prove the inequality $\lambda f \preccurlyeq a \rightarrow f(a)$, which is obvious.
3. By definition of the abstraction, to show that $a \preccurlyeq \lambda(x \mapsto ax)$ it is enough to show that for any $x \in A$ we have $a \preccurlyeq x \rightarrow ax$. By distributivity, we have:

$$x \rightarrow ax = x \rightarrow \bigcurlywedge\{b \in \mathcal{A} : a \preccurlyeq (x \rightarrow b)\} = \bigcurlywedge_{x, b \in \mathcal{A}} \{x \rightarrow b : a \preccurlyeq (x \rightarrow b)\}$$

   We conclude by proving that a is a lower bound of the set on the right hand-side, which is a tautology.

□

We call a *λ-term with parameters (in $\mathcal{A}$)* any term defined from the following grammar:

$$t, u ::= x \mid a \mid \lambda x.t \mid tu$$

where x is a variable and a is an element of $\mathcal{A}$. We can thus associate to each closed λ-term with parameters t an element $t^{\mathcal{A}}$ of $\mathcal{A}$, defined by induction on the size of t as follows:

$$\begin{array}{rcll} a^{\mathcal{A}} & \triangleq & a & \qquad (\text{if } a \in \mathcal{A}) \\ (tu)^{\mathcal{A}} & \triangleq & (t^{\mathcal{A}})u^{\mathcal{A}} & \\ (\lambda x.t)^{\mathcal{A}} & \triangleq & \lambda(a \mapsto (t[a/x])^{\mathcal{A}}) & \end{array}$$

Thanks to the properties of the application and of the abstraction in implicative structures that we proved, we can check that the embedding of λ-term is sound with respect to the β-reduction and the η-expansion:

$$\frac{(x:a)\in\Gamma}{\Gamma\vdash x:a}\ (\mathrm{Ax}) \qquad \frac{}{\Gamma\vdash a:a}\ (\mathcal{A}) \qquad \frac{FV(t)\subseteq \mathrm{dom}(\Gamma)}{\Gamma\vdash t:\top}\ (\top)$$

$$\frac{\Gamma\vdash t:a \quad a\preccurlyeq a'}{\Gamma\vdash t:a'}\ (\preccurlyeq) \qquad \frac{\Gamma\vdash t:a \quad \Gamma'\preccurlyeq\Gamma}{\Gamma'\vdash t:a}\ (w) \qquad \frac{\Gamma,x:a\vdash t:b}{\Gamma\vdash\lambda x.t:a\to b}\ (\lambda)$$

$$\frac{\Gamma\vdash t:a\to b \quad \Gamma\vdash u:a}{\Gamma\vdash tu:b}\ (@) \qquad \frac{\Gamma\vdash t:a_i \quad \text{for all } i\in I}{\Gamma\vdash t:\bigcurlywedge_{i\in I}a_i}\ (\curlywedge)$$

Figure 10.1: Semantic typing rules

Lemma 10.16. *The substitution of variable by parameter is monotonic, that is to say: for each λ-term t with free variables $x_1,\ldots,x_n$, and for all parameters $a_1,b_1,\ldots,a_n,b_n$, if $a_i\preccurlyeq b_1$ for all $i\leq n$, then:*

$$(t[a_1/x_1,\ldots,a_n/x_n])^{\mathcal{A}}\preccurlyeq(t[b_1/x_1,\ldots,b_n/x_n])^{\mathcal{A}}$$

Proof. By induction on the structure of t, using Propositions 10.12 and 10.15. □

Proposition 10.17. *For all closed λ-terms t and u with parameters in $\mathcal{A}$, the following holds:*

1. *If $t\to_\beta u$, then $t^{\mathcal{A}}\preccurlyeq u^{\mathcal{A}}$.*
2. *If $t\to_\eta u$, then $u^{\mathcal{A}}\preccurlyeq t^{\mathcal{A}}$.*

Proof. Straightforward from Proposition 10.15 and Lemma 10.16. □

Again, if we think of the order relation $a\preccurlyeq b$ as "a is more precise than b", it makes sense that the β-reduction $t\to_\beta u$ is reflected in the ordering $t^{\mathcal{A}}\preccurlyeq u^{\mathcal{A}}$: the result of a computation contains indeed less information than the computation itself[2].

10.2.2 Adequacy

We now dispose of a structure in which we can interpret types and λ-terms. We saw that the interpretation of terms was intuitively sound with respect to the β-reduction. We shall now prove that the typing rules of System F are adequate with respect to the interpretation of terms, that is to say that if t is a closed λ-terms of type T, then $t^{\mathcal{A}}\preccurlyeq T^{\mathcal{A}}$. The last statement can again be understood as the fact that a term (*i.e.* a computation) carries more information than its type, just like a realizer of a formula is more informative about the formula than the formula itself.

10.2.2.1 Semantic typing rules

To this aim, we start by defining a semantic type system, that is a set of inference rules where terms are typed with elements of $\mathcal{A}$. Typing judgments are thus of the shape $\Gamma\vdash t:a$ where:

- t a λ-term with parameters;
- a is an element of the implicative structure $\mathcal{A}$;
- Γ is a finite list of the shape $\Gamma\equiv x_1:a_1,\ldots,x_n:a_n$, where the x_i are variables and the a_i are elements of $\mathcal{A}$.

[2] For instance, 0 contains less information than $15-(3\times 5)$ or than $\mathbb{1}_{\mathbb{Q}}(\sqrt{(2)})$.

Since elements of $\mathcal{A}$ are also their own realizers, we can also identify typing contexts with substitutions whose values are in $\mathcal{A}$. The ordering relation naturally extends to typing contexts: we write $\Gamma' \preccurlyeq \Gamma$ when for every binding $(x : a) \in \Gamma$, there exists a binding $(x : a') \in \Gamma'$ such that $a' \preccurlyeq a$. In other words, the relation $\Gamma' \preccurlyeq \Gamma$ means that $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Gamma')$ and that Γ' restricted to $\mathrm{dom}(\Gamma)$ is lower than Γ component-wise.

Using the notation $t[\Gamma]$ to denote the term t under the substitution Γ, we can finally define the sequents $\Gamma \vdash t : a$ as shorthands for:

$$\Gamma \vdash t : a \quad \triangleq \quad FV(t) \subseteq \mathrm{dom}(\Gamma) \ \wedge \ (t[\Gamma])^{\mathcal{A}} \preccurlyeq a$$

We can now prove that:

Proposition 10.18 (Semantic typing)**.** *The typing rules in Figure 10.1 are sound,* i.e. *for each inference rule, we can deduce the conclusion from its hypotheses.*

Proof. Simple proof by case analysis.

- **Cases** (Ax),($\mathcal{A}$),($\top$). Obvious from the definition.

- **Case** ($\preccurlyeq$). Direct by transitivity of the order: if $(t[\Gamma])^{\mathcal{A}} \preccurlyeq a$ and $a \preccurlyeq a'$ then $(t[\Gamma])^{\mathcal{A}} \preccurlyeq a'$.

- **Case** (w). Follows from the definition of $\Gamma' \preccurlyeq \Gamma$ and the monotonicity of the substitution(Lemma 10.16).

- **Case** (λ). Assume that t is a term, that a, b are elements of $\mathcal{A}$ and that Γ is a context such that $FV(t) \subseteq \mathrm{dom}(\Gamma) \cup \{a\}$ and $(t[\Gamma, x : a])^{\mathcal{A}} \preccurlyeq b$. Then we have:

$$(\lambda x.t[\Gamma])^{\mathcal{A}} = \bigcurlywedge_{c \in \mathcal{A}} (c \to (t[\Gamma, x : c])^{\mathcal{A}}) \preccurlyeq a \to (t[\Gamma, x : a])^{\mathcal{A}} \preccurlyeq a \to b$$

- **Case** (@). Assume that t, u are terms, that a, b are elements of $\mathcal{A}$, and that Γ is a context such that:

$$FV(t), FV(u) \subseteq \mathrm{dom}(\Gamma) \qquad (t[\Gamma])^{\mathcal{A}} \preccurlyeq a \to b \qquad (u[\Gamma])^{\mathcal{A}} \preccurlyeq u$$

Then by definition and adjunction, we have:

$$(tu[\Gamma])^{\mathcal{A}} = (t[\Gamma])^{\mathcal{A}} (u[\Gamma])^{\mathcal{A}} \qquad \text{and} \qquad (t[\Gamma])^{\mathcal{A}} (u[\Gamma])^{\mathcal{A}} \preccurlyeq b \Leftrightarrow (t[\Gamma])^{\mathcal{A}} \preccurlyeq (u[\Gamma])^{\mathcal{A}} \to b$$

We conclude by anti-monotonicity of the implication:

$$(t[\Gamma])^{\mathcal{A}} \preccurlyeq a \to b \preccurlyeq (u[\Gamma])^{\mathcal{A}} \to b$$

- **Case** ($\bigcurlywedge$). This case is obvious since the meet is the greatest lower bound. □

This finally formalizes the intuition that $t \preccurlyeq a$ could be read as "t realizes a". Indeed, if t is a closed λ-term, and A a formula of system F, the adequacy lemma (Proposition 3.14) of Krivine classical realizability gives us that $t \Vdash A$, while the previous corollary somewhat gives us $t^{\mathcal{A}} \preccurlyeq A^{\mathcal{A}}$. Nonetheless, to justify formally such a statement, we should define an embedding of formulas and to prove the adequacy of the translations of terms and types with respect to the typing rules of System F.

$$\frac{(x:A)\in\Gamma}{\Gamma\vdash x:A}\,(\text{Ax}) \qquad \frac{\Gamma,x:A\vdash t:B}{\Gamma\vdash\lambda x\,.\,t:A\to B}\,(\to_I) \qquad \frac{\Gamma\vdash t:A\to B \qquad \Gamma\vdash t:A}{\Gamma\vdash tu:B}\,(\to_E)$$

$$\frac{\Gamma\vdash t:A \quad X\notin FV(\Gamma)}{\Gamma\vdash t:\forall X.A}\,(\forall_I) \qquad \frac{\Gamma\vdash t:\forall X.A}{\Gamma\vdash t:A\{X:=B\}}\,(\forall_E) \qquad \frac{}{\Gamma\vdash \boldsymbol{cc}:((A\to B)\to A)\to A}\,(\boldsymbol{cc})$$

Figure 10.2: Type system$^\bullet$ for the λ_c-calculus

10.2.2.2 Adequacy of the interpretation

For the formalization of the former result, we chose a slightly different approach that we shall now sketch. First, we extend the usual formulas of System F by defining second-order formulas with parameters as:

$$A,B ::= a \mid X \mid A\Rightarrow B \mid \forall X.A \qquad\qquad (a\in\mathcal{A})$$

We can then embed closed formulas with parameters into the implicative structure $\mathcal{A}$. The embedding is trivially defined by:

$$\begin{array}{rcll} a^{\mathcal{A}} & \triangleq & a & \text{(if } a\in\mathcal{A}\text{)}\\ (A\Rightarrow B)^{\mathcal{A}} & \triangleq & A^{\mathcal{A}}\to B^{\mathcal{A}} & \\ (\forall X.A)^{\mathcal{A}} & \triangleq & \bigcurlywedge_{a\in\mathcal{A}}(A\{X:=a\})^{\mathcal{A}} & \end{array}$$

We define a type system for the λ_c-calculus with parameters[3] (that is λ-terms with parameter plus an instruction $\boldsymbol{cc}$). Typing contexts$^\bullet$ are defined as usual by finite lists of hypotheses of the shape $(x:A)$ where x is a variable and A a formula with parameters. The inference rules, given in Figure 10.2, are the same as in System F (with the extended syntaxes of terms and formulas with parameters), plus the additional rules for cc.

In order to prove the adequacy of the type system with respect to the embedding, we define substitutions$^\bullet$, which we write σ, as functions mapping variables (of terms and types) to element of $\mathcal{A}$:

$$\sigma ::= \varepsilon \mid \sigma[x\mapsto a] \mid \sigma[X\mapsto a] \qquad\qquad (a\in\mathcal{A},\ x,X \text{ variables})$$

In the spirit of the proof of adequacy in classical realizability, we say that a substitution σ realizes$^\bullet$ a typing context Γ, which we write $\sigma\Vdash\Gamma$, if for all bindings $(x:A)\in\Gamma$ we have $\sigma(x)\preccurlyeq(A[\sigma])^{\mathcal{A}}$.

Theorem$^\bullet$ 10.19. *The typing rules of Figure 10.2 are adequate with respect to the interpretation of terms and formulas: if t is a λ_c-term with parameters, A a formula with parameters and Γ a typing context such that $\Gamma\vdash t:A$ then for all substitutions $\sigma\Vdash\Gamma$, we have $(t[\sigma])^{\mathcal{A}}\preccurlyeq(A[\sigma])^{\mathcal{A}}$.*

Proof. The proof resembles the usual proof of adequacy in classical realizability, and most of the cases are very similar to cases of Proposition 10.18. The additional case for the instruction $\boldsymbol{cc}$ is trivial since we define $\boldsymbol{cc}^{\mathcal{A}}\triangleq\bigcurlywedge_{a,b\in\mathcal{A}}(((a\to b)\to a)\to a)=(\forall XY.(((X\Rightarrow Y)\Rightarrow X)\Rightarrow X))^{\mathcal{A}}$ (we shall come back later to this definition). □

In the particular case where t is a closed term typed by A in the empty context, we obtain that $t^{\mathcal{A}}\preccurlyeq A^{\mathcal{A}}$. This result will be fundamental in the next section.

Corollary$^\bullet$ 10.20. *For all λ-terms t, if $\vdash t:A$, then $t^{\mathcal{A}}\preccurlyeq A^{\mathcal{A}}$.*

[3] In practice, we use Charguéraud's locally nameless representation [23] for terms and formulas. Without giving too much details, we actually define pre-terms$^\bullet$ and pre-types$^\bullet$ which allow both for names (for free variables) and De Bruijn indices (for bounded variables). Terms$^\bullet$ and types$^\bullet$ are then defined as pre-terms and pre-types without free De Bruijn indices. Such a representation is particularly convenient to prevent from name clashes to arise.

10.2.3 Combinators

The previous results indicates that any closed λ-terms is, through the interpretation, lower than the interpretation of its principal type. We give here some examples of closed λ-terms which are in fact equal to their principal types through the interpretation in $\mathcal{A}$. Let us now consider the following combinators:

$$\mathbf{I} \triangleq \lambda x.x \qquad \mathbf{K} \triangleq \lambda xy.x \qquad \mathbf{S} \triangleq \lambda xyz.xz(yz) \qquad \mathbf{W} \triangleq \lambda xy.xyy$$

It is well-known that these combinators can be given the following polymorphic types:

$$\begin{array}{rcl} \mathbf{I} & : & \forall X.X \Rightarrow X \\ \mathbf{K} & : & \forall XY.X \Rightarrow Y \Rightarrow X \\ \mathbf{S} & : & \forall XYZ.(X \Rightarrow Y \Rightarrow Z) \Rightarrow (X \Rightarrow Y) \Rightarrow X \Rightarrow Z \\ \mathbf{W} & : & \forall XY.(X \Rightarrow X \Rightarrow Y) \Rightarrow X \Rightarrow Y \end{array}$$

Through the interpretation these combinators are identified with their types:

Proposition 10.21. *The following equalities hold in any implicative structure $\mathcal{A}$:*

1.$^\bullet$ $\mathbf{I}^{\mathcal{A}} = \curlywedge_{a\in\mathcal{A}}(a \to a)$ 3.$^\bullet$ $\mathbf{S}^{\mathcal{A}} = \curlywedge_{a,b,c\in\mathcal{A}}((a \to b \to c) \to (a \to b) \to a \to c)$

2.$^\bullet$ $\mathbf{K}^{\mathcal{A}} = \curlywedge_{a,b\in\mathcal{A}}(a \Rightarrow b \Rightarrow a)$ 4.$^\bullet$ $\mathbf{W}^{\mathcal{A}} = \curlywedge_{a,b,c\in\mathcal{A}}((a \to a \to b) \to a \to b)$

Proof. The inequality from left to right are consequences of the adequacy.

1. By definition, $\mathbf{I}^{\mathcal{A}} = (\lambda x.x)^{\mathcal{A}} = \curlywedge_{a\in\mathcal{A}}(a \to a)$
2. By definition, $\mathbf{K}^{\mathcal{A}} = (\lambda xy.x)^{\mathcal{A}} = \curlywedge_{a\in\mathcal{A}}(a \to (\lambda y.a)^{\mathcal{A}}) = \curlywedge_{a\in\mathcal{A}}(a \to (\curlywedge_{b\in\mathcal{A}}(b \to a))$. We obtain the desired equality by distributivity.
3. By definition, $\mathbf{S}^{\mathcal{A}} = (\lambda xyz.xy(zy))^{\mathcal{A}} = \curlywedge_{x,y,z\in\mathcal{A}}(x \to y \to z \to xz(yz))$. We thus need to show that for any $x, y, z \in \mathcal{A}$, we have:

   $$\underset{a,b,c\in\mathcal{A}}{\curlywedge}((a \to b \to c) \to (a \to b) \to a \to c) \preccurlyeq x \to y \to z \to xz(yz)$$

   We use the transitivity to show that (the other inequality is trivial):

   $$\underset{c\in\mathcal{A}}{\curlywedge}((z \to yz \to c) \to (z \to yz) \to z \to c) \preccurlyeq x \to y \to z \to xz(yz) = \underset{c\in\mathcal{A}:\ xz\preccurlyeq yz\to c}{\curlywedge}(x \to y \to z \to c)$$

   where we obtain the equality by unfolding the definition of the application and by using the distributivity. We conclude by showing that for any $c \in \mathcal{A}$ such that $xz \preccurlyeq yz \to c$, we have:

   $$(z \to yz \to c) \to (z \to yz) \to z \to c \preccurlyeq x \to y \to z \to c$$

   This follows from the monotony of the arrow, using the adjunction of the implication. For instance, we have:

   $$x \preccurlyeq (z \to yz \to c) \quad \Leftrightarrow \quad xz \preccurlyeq yz \to c$$

4. The case for $\mathbf{W}$ is similar.

□

Finally, in the spirit of the previous equality, we define the interpretation of $\boldsymbol{cc}$ by the interpretation of its principal type, that is:

$$\boldsymbol{cc}^{\mathcal{A}} \triangleq \mathbf{cc} = \underset{a,b}{\curlywedge}(((a \to b) \to a) \to a)$$

Remark 10.22. It is not always the case that a term is equal to its principal type. Consider for instance a dummy implicative structure $\mathcal{A}$ where $a \to b = \top$ for all elements $a, b \in \mathcal{A}$. Suppose in addition that $\mathcal{A}$ has at least two distinct elements, so that $\bot \neq \top$. Then the following holds:

1.• For any $a, b \in \mathcal{A}$, we have $ab = \bigcurlywedge\{c : a \preccurlyeq b \to c\} = \bigcurlywedge \mathcal{A} = \bot$.

2.• For any $f : \mathcal{A} \to \mathcal{A}$, we have $\lambda f = \bigcurlywedge_{a\in\mathcal{A}}(a \to f(a)) = \bigcurlywedge_{a\in\mathcal{A}} \top = \top$.

3.• **ii** $: \forall X.X \to X$, yet $(\mathbf{ii})^{\mathcal{A}} = \bot \neq \top = (\forall X.X \to X)^{\mathcal{A}}$.

4.• $\mathbf{i}^{\mathcal{A}} = \top \neq \bot = (\mathbf{skk})^{\mathcal{A}}$.

⌟

10.2.4 The problem of consistency

The last remark shows us that not all implicative structures are suitable for interpreting intuitionistic or classical logic. We thus need to introduce a criterion of consistency:

Definition 10.23 (Consistency). We say that an implicative structure is:

- *intuitionistically consistent* if $t^{\mathcal{A}} \neq \bot$ for all closed λ-terms;
- *classically consistent* if $t^{\mathcal{A}} \neq \bot$ for all closed λ_c-terms.

⌟

We verify that non trivial complete Heyting algebras are consistent as implicative algebras. To this aim, we first show that:

Proposition 10.24. *In any complete Heyting algebra $\mathcal{A}$, all closed pure λ-terms t arei nterpreted as the maximal element: $t^{\mathcal{A}} = \top$.*

Proof. Remember from Remark 10.13 that the application in the associated implicative structure is characterized by $ab = a \curlywedge b$ for all $a, b \in \mathcal{H}$. We prove a more general proposition, namely that for any closed λ-term t with parameters $a_1, \ldots, a_n \in \mathcal{A}$, we have:

$$t^{\mathcal{A}} \succcurlyeq a_1 \curlywedge \ldots \curlywedge a_n$$

In the particular case where t is a pure λ-term (*i.e.* without any parameter), it indeed implies that $t^{\mathcal{A}} = \top$. We proceed by induction on t. The cases for the application and parameters are trivial, for the abstraction we have:

$$(\lambda x.t)^{\mathcal{A}} = \bigcurlywedge_{a\in\mathcal{A}} (a \to (t[a/x])^{\mathcal{A}}) \succcurlyeq \bigcurlywedge_{a\in\mathcal{A}} (a \to a \curlywedge a_1 \curlywedge \ldots \curlywedge a_n)$$

We conclude by showing that for any a, we have:

$$a_1 \curlywedge \ldots \curlywedge a_n \preccurlyeq a \to a \curlywedge a_1 \curlywedge \ldots \curlywedge a_n$$

which follows by adjunction. □

The proposition above enforces the observation (see Example 9.32) that Heyting algebras and Boolean algebras provide us with an interpretation of logic that is degenerated with respect to the computation. In other words, all proofs collapse to the maximal element $\top$. Nonetheless, this ensures that any non-degenerated Heyting algebra induces an intuitionistically consistent implicative structure:

Proposition 10.25. *Every non-degenerated Heyting algebras gives rise to an intuitionistically consistent implicative structure.*

We shall now relate the previous definition to the usual definition of consistency in classical realizability. Recall that any abstract Krivine structures $\mathcal{K} = (\mathbf{\Lambda}, \mathbf{\Pi}, \mathsf{app}, \mathsf{push}, \mathsf{k}_{\_}, \boldsymbol{k}, \boldsymbol{s}, \boldsymbol{cc}, \mathbf{PL}, \perp\!\!\!\perp)$ induces an implicative structure $(\mathcal{A}, \preccurlyeq, \to)$ where $\mathcal{A} = \mathcal{P}(\mathbf{\Pi})$, $a \preccurlyeq b \Leftrightarrow a \supseteq b$ and $a \to b = a^{\perp\!\!\!\perp} \cdot b$. Remember that a realizability model is said to be consistent when there is no proof-like term realizing $\bot$. Rephrased in terms of abstract Krivine structures, a falsity value $a \in \mathcal{P}(\mathbf{\Pi})$ is said to be realized by $t \in \mathbf{PL}$, which we write $t \Vdash a$, if $t \in a^{\perp\!\!\!\perp}$. The consistency can then be expressed by this simple criterion:

$$\mathcal{K} \text{ is consistent} \quad \text{if and only if} \quad \{\bot\}^{\perp\!\!\!\perp} \cap \mathbf{PL} = \mathbf{\Pi}^{\perp\!\!\!\perp} \cap \mathbf{PL} = \emptyset$$

We thus need to check that this criterion of consistency for the AKS implies the consistency of the induced implicative algebra, *i.e.* that if t is a closed λ_c-term, then $t^{\mathcal{A}} \neq \bot$. By definition of the implicative algebra $\mathcal{A}$ induced the AKS, we have that $t^{\mathcal{A}} \in \mathcal{A} = \mathcal{P}(\mathbf{\Pi})$. Therefore, $t^{\mathcal{A}}$ is a falsity value from the point of view of the AKS. To ensure that it is not equal to $\bot$ (*i.e.* $\mathbf{\Pi}$), it is enough to find a realizer of $t^{\mathcal{A}}$ in the AKS. The consistency of the AKS precisely states that $\bot$ does not have any realizer.

Our strategy to find a realizer for $t^{\mathcal{A}}$ in the AKS is to use t itself. First, we reduce the problem to the set of terms that are identifiable with the combinatory terms of the AKS. We call a *combinatory term* any term that is obtained by combination of the previous combinators. To each combinatory term t we associate a term t^{Λ} in $\mathbf{\Lambda}$, whose definition by induction is trivial:

$$\mathbf{K}^{\Lambda} \triangleq \boldsymbol{k} \qquad\qquad \mathbf{S}^{\Lambda} \triangleq \boldsymbol{s} \qquad\qquad \mathbf{CC}^{\Lambda} \triangleq \boldsymbol{cc} \qquad\qquad (tu)^{\Lambda} \triangleq \mathsf{app}(t^{\Lambda}, u^{\Lambda})$$

Since the set $\mathbf{PL}$ is closed under application, for any combinatory term t, its interpretation t^{Λ} is in $\mathbf{PL}$. The combinatory completeness of $(\mathbf{K}, \mathbf{S}, \mathbf{CC})$ with respect to closed λ_c-terms ensures us that there exists a combinatory term t_0 (viewed as a λ-term) such that $t_0 \to_\beta t$. By Proposition 10.17, we thus have $t_0^{\mathcal{A}} \preccurlyeq t^{\mathcal{A}}$. It is thus enough to show that $t_0^{\mathcal{A}} \neq \bot$: we reduced the original problem for closed λ_c-terms to combinatory terms.

It thus only remains to show that for any combinatory term t_0, its interpretation $t_0^{\mathcal{A}}$ is not $\bot$. For the reason detailed above, it is sufficient to prove that $t_0^{\mathcal{A}}$ is realized. We prove that $t_0^{\mathcal{A}}$ is in fact realized by t_0^{Λ}:

Lemma 10.26. *For any combinatory term t, t^{Λ} realizes $t^{\mathcal{A}}$,* i.e. $t^{\Lambda} \Vdash t^{\mathcal{A}}$

Proof. We proceed by induction on the structure of t, by combining usual results of classical realizability and properties of the implicative structures:

- For the three combinators $\mathbf{K}, \mathbf{S}, \mathbf{CC}$, we have that their interpretations in $\mathcal{A}$ are equal to their principal types (see Proposition 10.21), which their associated combinators in the AKS realize. For instance, $\mathbf{K}^{\mathcal{A}} = \bigwedge_{a,b \in \mathcal{A}} (a \to b \to a)$ and $\mathbf{K}^{\Lambda} = \boldsymbol{k} \Vdash \|\forall AB.A \to B \to A\|$. By definition of the implicative structures, we have $\bigwedge_{a,b \in \mathcal{A}} (a \to b \to a) = \|\forall AB.A \to B \to A\|$. Thus $\mathbf{K}^{\Lambda} \Vdash \mathbf{K}^{\mathcal{A}}$.
- If $t = t_1 t_2$, we have by induction hypothesis $t_1^{\Lambda} \Vdash t_1^{\mathcal{A}}$. By η-expansion (Proposition 10.17), we get that $t_1^{\mathcal{A}} \preccurlyeq t_2^{\mathcal{A}} \to t_1^{\mathcal{A}} t_2^{\mathcal{A}}$, and thus by subtyping $t_1^{\Lambda} \Vdash t_2^{\mathcal{A}} \to t_1^{\mathcal{A}} t_2^{\mathcal{A}}$. Since we have $t_2^{\Lambda} \Vdash t_2^{\mathcal{A}}$ by induction hypothesis, we can conclude that $t_1^{\Lambda} t_2^{\Lambda} \Vdash t_1^{\mathcal{A}} t_2^{\mathcal{A}}$.

□

We can thus conclude that the consistency of the AKS induces the one (in the sense of Definition 10.23) of the associated implicative structures:

Proposition 10.27. *If $\mathcal{K}$ is a consistent abstract Krivine structure, then the implicative structure it induces is classically consistent.*

Proof. Let t be any closed λ_c-term. We want to show that $t^{\mathcal{A}} \neq \bot = \mathbf{\Pi}$. We show that $t^{\mathcal{A}}$, which belongs to $\mathcal{P}(\mathbf{\Pi})$ is realized by a proof-like term □

It is worth noting that the previous reasoning also applies to Krivine ordered combinatory algebras, since they induce abstract Krivine structures. Besides, the criterion of consistency is defined in both case with respect to the set **PL** (the filter for $^{\mathcal{K}}$OCAs, recall that both are identified through the passage from $^{\mathcal{K}}$OCA to AKS). Beyond that, this set (together with the pole in the case of AKS) is the key ingredient in the definition of the realizability tripos. It is already at the heart of the definition of Krivine's realizabilty models, where valid formulas are precisely the formulas realized by a proof-like term. We shall then introduce the corresponding ingredient for implicative structures.

10.3 Implicative algebras

10.3.1 Separation

Definition$^\bullet$ 10.28 (Separator)**.** Let $(\mathcal{A},\preccurlyeq,\to)$ be an implicative structure. We call a *separator* over $\mathcal{A}$ any set $\mathcal{S} \subseteq \mathcal{A}$ such that for all $a,b \in A$, the following conditions hold:

1. $\mathbf{K}^{\mathcal{A}} \in \mathcal{S}$, and $\mathbf{S}^{\mathcal{A}} \in \mathcal{S}$. (Combinators)
2. If $a \in \mathcal{S}$ and $a \preccurlyeq b$, then $b \in \mathcal{S}$. (Upwards closure)
3. If $(a \to b) \in \mathcal{S}$ and $a \in \mathcal{S}$, then $b \in \mathcal{S}$. (Closure under modus ponens)

A separator $\mathcal{S}$ is said to be *classical* if besides $\mathbf{cc}^{\mathcal{A}} \in \mathcal{S}$ and *consistent* if $\bot \notin \mathcal{S}$. ⌟

Remark$^\bullet$ 10.29 (Alternative definition)**.** In presence of condition (2), condition (3) is equivalent to the following condition:

(3') If $a \in \mathcal{S}$ and $b \in \mathcal{S}$ then $ab \in \mathcal{S}$ (Closure under application)

The proof uses basic properties of application:

- (3) ⇒(3'): If $a \in \mathcal{S}$ and $b \in \mathcal{S}$, since $a \preccurlyeq b \to ab$ (Proposition 10.17) by upward closure we have $b \to ab \in \mathcal{S}$, and thus $ab \in \mathcal{S}$ by modus ponens .
- (3')⇒(3): If $a \in \mathcal{S}$ and $a \to b \in \mathcal{S}$, then $(a \to b)a \in \mathcal{S}$ by closure under application. Since $(a \to b)a \preccurlyeq b$ (Proposition 10.17) by upward closure we conclude that $b \in \mathcal{S}$.

⌟

Intuitively, thinking of elements of an implicative structure as truth values, a separator should be understood as the set which distinguishes the valid formulas. Considering the elements as terms, it should rather be viewed as the set of valid realizers. Indeed, conditions (1) and (3') ensure that all λ-terms are in any separator. Reading $a \preccurlyeq b$ as "*the formula a is a subtype of the formula b*", condition (2) ensures the validity of semantic subtyping. Thinking of the ordering as "*a is a realizer of the formula b*", condition (2) states that if a formula is realized, then it is in the separator.

Definition$^\bullet$ 10.30 (Implicative algebra)**.** We call *implicative algebra* any quadruple $(\mathcal{A},\preccurlyeq,\to,\mathcal{S})$ where $(\mathcal{A},\preccurlyeq,\to)$ is an implicative structure and $\mathcal{S}$ is a separator over $\mathcal{A}$. We say that an implicative algebra is *classical* if its separator is. ⌟

Example$^\bullet$ 10.31 (Complete Boolean algebras)**.** If $\mathcal{B}$ is a complete Boolean algebra, then $\mathcal{B}$ induces an implicative structure. Besides, the interpretation of any closed λ-term is equal to $\top$ (Proposition 10.24), and it is easy to verify that for all $a,b \in \mathcal{B}$, $(((a \to b) \to a) \to a) = (\neg(\neg(\neg a \curlyvee b) \curlyvee a) \curlyvee a) = \top$, so that in particular $\mathbf{cc}^{\mathcal{B}} = \top$. Therefore, the singleton $\{\top\}$ is a classical separator for the induced implicative structure (it is obviously closed under modus ponens and upward closure). Any non-degenerated complete Boolean algebras thus induces a classically consistent implicative algebra.

Alternatively, any filter for $\mathcal{B}$ defines a separator: a filter is upward closed and closed under (binary) meets by definition. Since the application ab in Boolean algebras coincide with the binary meet $a \curlywedge b$ (Remark 10.13), any filter satisfies conditions (2) and (3') ⌟

Example 10.32 (Abstract Krivine structure)**.** Recall that any AKS $(\Lambda, \Pi, \mathsf{app}, \mathsf{push}, \mathsf{k}_\_, \boldsymbol{k}, \boldsymbol{s}, \boldsymbol{cc}, \mathbf{PL}, \perp\!\!\!\perp)$ induces an implicative structure $(\mathcal{A}, \preccurlyeq, \rightarrow)$ where $\mathcal{A} = \mathcal{P}(\Pi)$, $a \preccurlyeq b \Leftrightarrow a \supseteq b$ and $a \rightarrow b = a^{\perp\!\!\!\perp} \cdot b$. The sets of realized formulas, namely $\mathcal{S} = \{a \in \mathcal{A} : a^{\perp\!\!\!\perp} \cap \mathbf{PL} \neq \emptyset\}$, defines a valid separator. The condition of upward-closure is obvious by subtyping and we saw in Lemma 10.26 that $\mathbf{K}^{\mathcal{A}}, \mathbf{S}^{\mathcal{A}}, \mathbf{CC}^{\mathcal{A}}$ were realized respectively by $\boldsymbol{k}, \boldsymbol{s}$ and $\boldsymbol{cc}$. As for the closure under modus ponens, for any $a, b \in \mathcal{A}$, if $(a \rightarrow b) \in \mathcal{S}$ and $a \in \mathcal{S}$, by definition there exist $t, u \in \Lambda$ such that $t \Vdash a \rightarrow b$ and $u \Vdash a$. Therefore, $tu \Vdash b$ and thus $b \in S$. ⌟

10.3.2 λ_c-terms

The first property that we shall state about classical separators is that they contain the interpretation of all closed λ_c-terms. This follows again from the combinatorial completeness of the basis $(\mathbf{K}, \mathbf{S}, \mathbf{CC})$ for the λ_c-calculus. Indeed, if $\mathcal{S}$ is a classical separator over an implicative structure $(\mathcal{A}, \preccurlyeq, \rightarrow)$, it is clear that any combinatory term is in the separator. Again, by combinatory completeness, if t is a closed λ_c-term, there exists a combinatory term t_0 such that $t_0 \rightarrow_\beta t$, and therefore $t_0^{\mathcal{A}} \preccurlyeq t^{\mathcal{A}}$ (by Proposition 10.17). By upward closure of separators, we deduce that:

Proposition• 10.33. *If $(\mathcal{A}, \preccurlyeq, \rightarrow, \mathcal{S})$ is a (classical) implicative algebra and t is a closed λ-term (resp. λ_c-term), then $t^{\mathcal{A}} \in \mathcal{S}$.*

From the previous proposition and the adequacy of second-order typing rules for the λ_c-calculus (Theorem 10.19), we obtain that:

Corollary• 10.34. *If $(\mathcal{A}, \preccurlyeq, \rightarrow, \mathcal{S})$ is a (classical) implicative algebra, t is a closed λ-term (resp. λ_c-term) and A is a formula such that $\vdash t : A$, then $A^{\mathcal{A}} \in \mathcal{S}$.*

Remark 10.35. The latter corollary provides us with a methodology for proving that an element of a given implicative algebra is in the separator. In the spirit of realizability, where the standard methodology to prove that a formula is realized consists in using typed terms and adequacy as much as possible, we can use typed terms to prove automatically that the corresponding formulas belongs to the separator. We shall use this methodology[4] abundantly in the sequel. ⌟

10.3.3 Internal logic

In order to be able to define triposes from implicative algebras, we first need to equip them with a structure of Heyting algebra. To this end, we begin with defining an entailment relation in the spirit of filtered OCAs. We then define quantifiers and connectives as usual in classical realizability (see Section 3.3.1.1), and we verify that they satisfy the usual logical rules. This will lead us to the definition of the implicative tripos.

10.3.3.1 Entailment

In the rest of this section, we work within a fixed implicative algebra $(\mathcal{A}, \preccurlyeq, \rightarrow, \mathcal{S})$.

Definition• 10.36 (Entailment)**.** For all $a, b \in \mathcal{A}$, we say that a *entails* b and write $a \vdash_{\mathcal{S}} b$ if $a \rightarrow b \in \mathcal{S}$. We say that a and b are equivalent and write $a \cong_{\mathcal{S}} b$ if $a \vdash_{\mathcal{S}} b$ and $b \vdash_{\mathcal{S}} a$. ⌟

Proposition 10.37 (Properties of $\vdash_{\mathcal{S}}$)**.** *For any $a, b, c \in \mathcal{A}$, the following holds:*

1.• $a \vdash_{\mathcal{S}} a$ *(Reflexivity)*
2.• *if $a \vdash_{\mathcal{S}} b$ and $b \vdash_{\mathcal{S}} c$ then $a \vdash_{\mathcal{S}} c$* *(Transitivity)*

[4] In the Coq development, this corresponds to the tactic called `realizer`• which we indeed use a lot.

3.$^\bullet$ *if $a \preccurlyeq b$ then $a \vdash_{\mathcal{S}} b$* *(Subtyping)*

4.$^\bullet$ *if $a \cong_{\mathcal{S}} b$ then $a \in \mathcal{S}$ if and only if $b \in \mathcal{S}$* *(Closure under $\cong_{\mathcal{S}}$)*

5.$^\bullet$ *if $a \vdash_{\mathcal{S}} b \to c$ then $a \curlywedge b \vdash_{\mathcal{S}} c$* *(Half-adjunction property)*

6.$^\bullet$ *$\bot \vdash_{\mathcal{S}} a$* *(Ex falso quod libet)*

7.$^\bullet$ *$a \vdash_{\mathcal{S}} \top$* *(Maximal element)*

Proof. 2. We go once and for all through all the steps of the methodology described in Remark 10.35. If $a \vdash b$ and $b \vdash c$, we have by definition that $a \to b \in \mathcal{S}$ and $b \to c \in \mathcal{S}$. We use the closure under modus ponens and prove that $(a \to b) \to (b \to c) \to (a \to c) \in \mathcal{S}$. Besides, let us define $t \triangleq \lambda xyz.y(xz)$. It is clear that we can derive $\vdash t : \forall abb'.(a \to b) \to (b \to c) \to (a \to c)$ in System F, whence by Theorem 10.19 we have:

$$t^{\mathcal{A}} \preccurlyeq \min_{a,b,c\in\mathcal{A}} (a \to b) \to (b \to c) \to (a \to c)$$

Since $t^{\mathcal{A}} \in \mathcal{S}$ (Proposition 10.33) and $\mathcal{S}$ is upward closed, we get the expected result. In the sequel, we shall simply say that the formula is realized by $\lambda xyz.y(xz)$.

3. This is realized by the identity (by subtyping).
4. Direct from the definition of $\cong_{\mathcal{S}}$ and the closure under modus ponens.
5. The formula $(a \to b \to c) \to a \curlywedge b \vdash_{\mathcal{S}} c$ is realized (using the fact that $a \curlywedge b \preccurlyeq a, b)$ $W = \lambda xy.xyy$.

1,6,7. Direct from 3. □

Besides, the entailment relation is compatible with respect to the monotonicity of the arrow:

Proposition 10.38 (Compatibility with $\to$). *The following hold for all $a, a', b, b' \in \mathcal{A}$:*

1.$^\bullet$ *If $b \vdash b'$ then $a \to b \vdash a \to b'$*

2.$^\bullet$ *If $a \vdash a'$ then $a' \to b \vdash a \to b$*

Proof. 1. If $b \vdash b'$, we have by definition $b \to b' \in \mathcal{S}$. We use the closure under modus ponens and prove that $(b \to b') \to (a \to b) \to (a \to b') \in \mathcal{S}$. This formula is realized by $\lambda xyz.x(yz)$.

2. Similarly, we prove that $(a \to b') \to (a' \to b) \to (a \to b) \in \mathcal{S}$ since it is realized by $\lambda xyz.y(xz)$. □

Therefore, the arrow behaves like Heyting's arrow with respect to the preorder relation $\vdash_{\mathcal{S}}$ in terms of monotonicity. Nonetheless, we only have half the adjunction with the meet. Indeed, the other direction (if $a \curlywedge b \vdash_{\mathcal{S}} c$ then $a \vdash_{\mathcal{S}} b \to c$) does not make sense computationally, since the meet does not reflect a logical connective. This should not come as a surprise, since we explained in Section 9.1.1 that in realizability, the conjunction was interpreted by the product type rather than the meet.

10.3.3.2 Negation

Recall that the negation is defined by $\neg a \triangleq a \to \bot$. If additionally the separator is classical, we can prove that for any $a \in \mathcal{A}$, we have:

Proposition 10.39 (Double negation). *If $\mathcal{S}$ is a classical separator, the following holds for any $a \in \mathcal{A}$:*

1.$^\bullet$ *$a \vdash_{\mathcal{S}} \neg\neg a$*

2.$^\bullet$ *$\neg\neg a \vdash_{\mathcal{S}} a$*

Proof. 1. Trivial, since it is realized by $\lambda xk.kx$.

2. Follows from the inequality $((a \to \bot) \to a) \to a \preccurlyeq ((a \to \bot) \to \bot) \to a$, whose left member is realized by **cc**.

□

10.3.3.3 Quantifiers

Following the usual definition in classical realizability (see Section 9.1.1), the universal quantification of a family of truth values is naturally defined as its meet. Therefore, we introduce the convenient notation:

$$\bigforall_{i\in I} a_i \triangleq \bigcurlywedge_{i\in I} a_i$$

It is clear that this definition is compatible with the expected semantic rules:

Proposition• 10.40 (Universal quantifier)**.** *The following semantic typing rules are valid in any implicative structures:*

$$\frac{\Gamma \vdash t : a_i \quad \textit{for all } i \in I}{\Gamma \vdash t : \forall_{i\in I} a_i} \qquad\qquad \frac{\Gamma \vdash t : \forall_{i\in I} a_i \quad i_0 \in I}{\Gamma \vdash t : a_{i_0}}$$

Dually, we follow the usual encodings of the existential quantification (see Section 3.3.1.1), and we define:

$$\exists_{i\in I} a_i \triangleq \bigcurlywedge_{c\in A}(\bigcurlywedge_{i\in I}(a_i \to c) \to c)$$

While it could have seemed more natural to define existential quantifiers through joins, we should recall that the arrow does not commute with joins in general. We shall see in Section 10.4.4.2 that when it does, the realizability tripos precisely collapses to a forcing tripos. Once more, the expected semantic typing rules are satisfied:

Proposition• 10.41 (Existential quantifier)**.** *The following semantic typing rules are valid in any implicative structures:*

$$\frac{\Gamma \vdash t : a_{i_0} \quad i_0 \in I}{\Gamma \vdash \lambda x.xt : \exists_{i\in I} a_i} \qquad\qquad \frac{\Gamma \vdash t : \exists_{i\in I} a_i \quad \Gamma, x : a_i \vdash u : c \quad (\textit{for all } i \in I)}{\Gamma \vdash t(\lambda x.u) : c}$$

Proof. Straightforward using the adjunction of the application (Proposition 10.12) and lattices properties. For instance, for the introduction rule, assume that $(t[\Gamma])^{\mathcal{A}} \preccurlyeq a_i$ for some $i \in I$. Then we have to prove that $(\lambda x.xt[\Gamma])^{\mathcal{A}} \preccurlyeq \bigcurlywedge_{c\in A}(\bigcurlywedge_{i\in I}(a_i \to c) \to c)$. Let then c be in $\mathcal{A}$, using the adjunction it suffices to prove that:

$$(\lambda x.xt[\Gamma])^{\mathcal{A}}(\bigcurlywedge_{i\in I}(a_i \to c)) \preccurlyeq c$$

Using the property of β-reduction (Proposition 10.17) and the transitivity, it is enough to show that:

$$(\bigcurlywedge_{i\in I}(a_i \to c)(t[\Gamma])^{\mathcal{A}}) \preccurlyeq c \quad\Leftrightarrow\quad \bigcurlywedge_{i\in I}(a_i \to c) \preccurlyeq (t[\Gamma])^{\mathcal{A}} \to c$$

We conclude using the hypothesis for t and the anti-monotonicity of the arrow. The proof for the elimination rule is very similar. Observe that we really consider the elements of the implicative structure as λ-terms, that is to say that we compute with truth values. □

10.3.3.4 Sum and product

We define it by the usual encodings in System F:

$$a \times b \triangleq \bigcurlywedge_{c\in\mathcal{A}}((a \to b \to c) \to c)$$

Recall that the pair $\langle a, b\rangle$ is encoded by the λ-term $\lambda x.xab$, while first and second projection are respectively defined by $\pi_1 \triangleq \lambda xy.x$ and $\pi_2 \triangleq \lambda xy.y$. We can check that the expected semantic typing rules for pairs are valid

Proposition• 10.42 (Product). *The following semantic typing rules are valid:*

$$\frac{\Gamma \vdash t : a \quad \Gamma \vdash u : b}{\Gamma \vdash \lambda z.ztu : a \times b} \qquad \frac{\Gamma \vdash t : a \times b}{\Gamma \vdash t\pi_1 : a} \qquad \frac{\Gamma \vdash t : a \times b}{\Gamma \vdash t\pi_2 : b}$$

Proof. Straightforward lattice manipulation, similar to the proof for the existential quantifier. □

Similarly, we can define a sum type through the usual encoding:

$$a + b \triangleq \bigcurlywedge_{c \in \mathcal{A}} ((a \to c) \to (b \to c) \to c)$$

We check again that the expected semantic typing rules for pairs are valid:

Proposition• 10.43 (Sum). *The following semantic typing rules are valid:*

$$\frac{\Gamma \vdash t : a}{\Gamma \vdash \lambda lr.lt : a + b} \qquad \frac{\Gamma \vdash t : b}{\Gamma \vdash \lambda lr.rt : a + b} \qquad \frac{\Gamma \vdash t : a + b \quad \Gamma, x : a \vdash u : c \quad \Gamma, y : b \vdash v : c}{\Gamma \vdash t(\lambda x.u)(\lambda y.v) : c}$$

Proof. Straightforward lattice manipulation. □

We are now ready to verify that the entailment relation together with the sum and products induce a structure of Heyting algebra. We will then focus to the construction of the implicative tripos.

10.4 Implicative triposes

10.4.1 Induced Heyting algebra

The natural candidate which computationally represents a "meet" of a and b is the product type $a \times b$. We can verify that it satisfies the expected property (in Heyting algebras) w.r.t. to the arrow:

Proposition• 10.44 (Adjunction). *For any $a, b, c \in \mathcal{A}$, we have:*

$$a \vdash_{\mathcal{S}} b \to c \qquad \textit{if and only if} \qquad a \times b \vdash_{\mathcal{S}} c$$

Proof. Both directions are proofs using the expected realizer and subtyping: from left to right, we use $\lambda xy.yx$ to realize $(a \to b \to c) \to a \times b \to c$; from right to left, we realize $(a \times b \to c) \to a \to b \to c$ with $\lambda pxy.p(\lambda z.zxy)$. □

Corollary 10.45 (Heyting prealgebra). *For any implicative algebra $(\mathcal{A}, \preccurlyeq, \to, \mathcal{S})$, the induced quintuple $(\mathcal{A}, \vdash_{\mathcal{S}}, \times, +, \to)$ is a Heyting prealgebra.*

The former is only a Heyting prealgebra and not a Heyting algebra because the entailment relation is a preorder (instead of an order). We thus consider the quotient $\mathcal{A}/_{\cong_{\mathcal{S}}}$ of the former Heyting prealgebra by the relation $\cong_{\mathcal{S}}$, which we write $\mathcal{A}/\mathcal{S}$ (and $\mathcal{H}$ hereafter). We equip $\mathcal{H}$ with an order relation:

$$[a] \leq_{\mathcal{H}} [b] \;\triangleq\; a \vdash_{\mathcal{S}} b \qquad \text{(for all } a, b \in \mathcal{A}\text{)}$$

where we write $[a]$ for the equivalence class of $a \in \mathcal{A}$. We define:

$$\begin{array}{rcl}
[a] \to_{\mathcal{H}} [b] & \triangleq & [a \to b] \\
{[a]} \wedge_{\mathcal{H}} [b] & \triangleq & [a \times b] \\
{[a]} \vee_{\mathcal{H}} [b] & \triangleq & [a + b] \\
\top_{\mathcal{H}} & \triangleq & [\top] = \mathcal{S} \\
\bot_{\mathcal{H}} & \triangleq & [\bot] = \{a \in \mathcal{A} : \neg a \in \mathcal{S}\}
\end{array}$$

Proposition 10.46 (Induced Heyting algebra)**.** *The quintuple* $(\mathcal{H}, \leq_{\mathcal{H}}, \wedge_{\mathcal{H}}, \vee_{\mathcal{H}}, \rightarrow_{\mathcal{H}})$ *is a Heyting algebra.*

Proof. We first show that $(\mathcal{H}, \leq_{\mathcal{H}}, \wedge_{\mathcal{H}}, \wedge_{\mathcal{H}})$ is a lattice. It is clear that $(\mathcal{H}, \leq_{\mathcal{H}})$ is a poset, we then have to prove that $\wedge_H$ and $\vee_H$ indeed defines binary meets and joins. We thus need to prove that for all $a, b, c \in \mathcal{A}$, we have:

1. • $[a] \times [b] \leq_{\mathcal{H}} [a]$ and $[a] \times [b] \leq_{\mathcal{H}} [b]$. In $\mathcal{A}$, the corresponding implications are realized respectively by $\lambda xy.x$ and $\lambda xy.y$.
2. • If $[c] \leq_{\mathcal{H}} [a]$ and $[c] \leq_{\mathcal{H}} [b]$, then $[c] \leq_{\mathcal{H}} [a] \times [b]$. Let us assume that $c \rightarrow a \in \mathcal{S}$ and $c \rightarrow b \in \mathcal{S}$. Then by closure of the separator under modus ponens, it suffices to show that $(c \rightarrow a) \rightarrow (c \rightarrow b) \rightarrow c \rightarrow (a \times b) \in \mathcal{S}$. This formula is realized by $\lambda tucz.z(tc)(uc)$.
3. • $[a] \leq_{\mathcal{H}} [a] + [b]$ and $[b] \leq_{\mathcal{H}} [a] + [b]$. The corresponding implications in $\mathcal{A}$ are realized respectively by $\lambda xtu.tx$ and $\lambda xtu.ux$.
4. • If $[a] \leq_{\mathcal{H}} [c]$ and $[b] \leq_{\mathcal{H}} [c]$, then $[c] \leq_{\mathcal{H}} [a] + [b]$. Let us assume that $c \rightarrow a \in \mathcal{S}$ and $c \rightarrow b \in \mathcal{S}$. Then by closure of the separator under modus ponens, it suffices to show that $(a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow (a + b) \rightarrow c \in \mathcal{S}$. This formula is realized by $\lambda xyt.txy$.

We already know from Proposition 10.37 that $\top$ and $\bot$ are respectively the maximal and minimal elements of $\mathcal{A}$ for $\leq_{\mathcal{H}}$. Thus $(\mathcal{H}, \leq_{\mathcal{H}}, \wedge_{\mathcal{H}}, \wedge_{\mathcal{H}})$ is a bounded lattice.

Finally, we need to prove that the adjunction $[a] \wedge_{\mathcal{H}} [b] \leq_{\mathcal{H}} [c] \Leftrightarrow [a] \leq_{\mathcal{H}} [b] \rightarrow_{\mathcal{H}} [c]$ holds for any $a, b, c \in \mathcal{A}$. This is a direct consequence of the corresponding adjunction that we proved in $\mathcal{A}$ for $\vdash_S$ and $\rightarrow$ (Proposition 10.44). □

Remark 10.47. If the implicative algebra is classical, for all $a \in \mathcal{A}$ we have $\neg\neg a \cong_{\mathcal{S}} a$ (Proposition 10.39). Through the quotient, this implies that $\neg\neg[a] = [a]$ for all $a \in \mathcal{A}$. This means that in the case of a classical implicative algebra, the induced Heyting algebra is actually a Boolean algebra. ⌟

We are almost ready to define the implicative tripos. Following the construction of triposes associated to AKSs and $^{\mathcal{K}}$OCAs, we want to define a functor roughly of the form $\mathcal{P} : I \in \mathbf{Set}^{op} \mapsto \mathcal{A}^I$. However, as we saw that the implicative algebra $\mathcal{A}$ gives rise to a Heyting algebra through a quotient by (the equivalence relation induced by) the separator. We first need to check that the indexed family $\mathcal{A}^I$ is an implicative structure. Then we will need to quotient $\mathcal{A}^I$ by an appropriate separator.

10.4.2 Product of implicative structures

Let I be a set and $(A_i)_{i \in I}$ be a family of implicative structures, which we write $(\mathcal{A}_i, \preccurlyeq_i, \rightarrow_i)$. The Cartesian product $\mathcal{A} \triangleq \prod_{i \in I} A_i$ is naturally equipped with a structure of implicative structure, using the order and implication defined componentwise:

$$(a_i)_{i \in I} \preccurlyeq (b_i)_{i \in I} \triangleq \forall i \in I.(a_i \preccurlyeq_i b_i) \qquad\qquad (a_i)_{i \in I} \rightarrow (b_i)_{i \in I} \triangleq (a_i \rightarrow_i b_i)_{i \in I}$$

Proposition 10.48 (Product of structures)**.** *The triple* $(\mathcal{A}, \preccurlyeq, \rightarrow)$ *is an implicative structure.*

Proof. Straightforward, since the variance and the distributivity are verified for each component. □

Since the order relation is defined componentwise, in particular the meet of a set of family is the family of the meet componentwise:

$$\bigcurlywedge_{(a_i)_{i \in I} \in \mathcal{A}} (a_i)_{i \in I} = \Big(\bigcurlywedge_{a_i \in \mathcal{A}_i} a_i \Big)_{i \in I}$$

As a consequence, all the definitions are compatible with the corresponding definitions componentwise, namely for all $a, b \in \mathcal{A}$ and any $f = (f_i : \mathcal{A}_i \to \mathcal{A}_i)_{i\in I}$ we have:

$$ab = (a_i b_i)_{i\in I} \qquad \lambda f = (\lambda f_i)_{i\in I} \qquad \mathbf{K}^{\mathcal{A}} = (\mathbf{K}^{\mathcal{A}_i})_{i\in I} \qquad \mathbf{S}^{\mathcal{A}} = (\mathbf{S}^{\mathcal{A}_i})_{i\in I}$$
$$a \times b = (a_i \times b_i)_{i\in I} \qquad a + b = (a_i + b_i)_{i\in I} \qquad \mathbf{cc}^{\mathcal{A}} = (\mathbf{cc}^{\mathcal{A}_i})_{i\in I}$$

In the same spirit, it is clear that if $(\mathcal{S}_i)_{i\in I} \subseteq \mathcal{A}_i$ is a family of separators (*i.e.* for each $i \in I$, $\mathcal{S}_i$ is a separator for $\mathcal{A}_i$), then the Cartesian product $\mathcal{S} = \prod_{i\in I} \mathcal{S}_i$ is a separator for the implicative structure $\mathcal{A}$. Besides, the entailment relation induced by this separator product corresponds again to the induced relation componentwise, since for all $a, b \in \mathcal{A}$ we have:

$$a \vdash_{\mathcal{S}} b \triangleq a \to b \in \mathcal{S} \quad \Leftrightarrow \quad \forall i \in I.(a_i \to_i b_i \in \mathcal{S}_i) \quad \Leftrightarrow \quad \forall i \in I.(a_i \vdash_{\mathcal{S}_i} b_i)$$

10.4.3 Implicative tripos

We are now ready to define the implicative tripos. Let $(\mathcal{A}, \preccurlyeq, \to, \mathcal{S})$ be a fixed implicative algebra. For each set I, the Cartesian product $\mathcal{A}^I$ gives rise to an implicative structure which we write $(\mathcal{A}^I, \preccurlyeq^I, \to^I)$. As explained in the previous section, the Cartesian product $\mathcal{S}^I$ defines a separator for the implicative structure $\mathcal{A}^I$, which we call the *power separator*. By definition, an element a of $\mathcal{A}^I$ belongs to the power separator $\mathcal{S}^I$, if for each $i \in I$, a_i belongs to $\mathcal{S}$. In terms of realizability, this intuitively means that for each $i \in I$, a_i is realized.

As we shall see further, this separator is too permissive in the sense that it contains too many elements and that the corresponding quotient collapses to a forcing tripos. Yet, the separator $\mathcal{S}$ induces another separator, which we write $\mathcal{S}[I]$ and call *uniform separator*, which is defined by:

$$\mathcal{S}[I] \triangleq \{a \in \mathcal{A}^I : \exists s \in S. \forall i \in I. s \preccurlyeq a_i\}$$

An element $a \in \mathcal{A}$ is thus in the uniform separator if it is uniformly realized by the same s in each component. We clearly have the following inclusion:

$$\mathcal{S}[I] \subseteq \mathcal{S}^I \subseteq \mathcal{A}^I$$

We write $(\mathcal{A}^I/\mathcal{S}[I], \leq_{\mathcal{S}[I]}, \to_{\mathcal{S}[I]})$ the associated Heyting algebra.

Theorem 10.49 (Implicative tripos). *Let $(\mathcal{A}, \preccurlyeq, \to, \mathcal{S})$ be an implicative algebra. The following functor :*

$$\mathcal{T} : I \mapsto \mathcal{A}^I/\mathcal{S}[I] \qquad\qquad \mathcal{T}(f) : \left\{ \begin{array}{ccc} \mathcal{A}^I/\mathcal{S}[I] & \to & \mathcal{A}^J/\mathcal{S}[J] \\ {[(a_i)_{i\in I}]} & \mapsto & {[(a_{f(j)})_{j\in J}]} \end{array} \right. \qquad\qquad (\forall f \in J \to I)$$

defines[5] *a tripos.*

Proof. We verify that $\mathcal{T}$ satisfies all the necessary conditions to be a tripos.

- The functoriality of $\mathcal{T}$ is clear.
- For each $I \in \mathbf{Set}$, the image of the corresponding diagonal morphism $\mathcal{T}(\delta_I)$ associates to any element $[(a_{ij})_{(i,j)\in I\times I}] \in \mathcal{T}(I \times I)$ the element $[(a_{ii})_{i\in I}] \in \mathcal{T}(I)$. We define[6]:

$$(=_I) \;:\; i, j \mapsto \begin{cases} \curlywedge_{a\in\mathcal{A}}(a \to a) & \text{if } i = j \\ \bot \to \top & \text{if } i \neq j \end{cases}$$

[5] Note that the definition of the functor on functions $f : J \to I$ assumes implicitly the possibility of picking a representative in any equivalent class $[a] \in \mathcal{A}/\mathcal{S}[I]$, *i.e.* the full axiom of choice.

[6] The reader familiar with classical realizability might recognize the usual interpretation of Leibniz's equality.

and we need to prove that for all $[a] \in \mathcal{T}(I \times I)$:

$$[\top]_I \leq_{\mathcal{S}[I]} \mathcal{T}(\delta_I)(a) \qquad \Leftrightarrow \qquad [=_I] \leq_{\mathcal{S}[I\times I]} [a]$$

Let then $[(a_{ij})_{i,j\in I}]$ be an element of $\mathcal{T}(I\times I)$. From left to right, assume that $[\top]_I \leq_{\mathcal{S}[I]} \mathcal{T}(\delta_I)(a)$, that is to say that there exists $s \in \mathcal{S}$ such that for any $i \in I$, $s \preccurlyeq \top \to a_{ii}$. Then it is easy to check that for all $i,j \in I$, $\lambda z.z(s(\lambda x.x)) \preccurlyeq i =_I j \to a_{ij}$. Indeed, using the adjunction and the β-reduction it suffices to show that for all $i,j \in I$, $(i =_I j) \preccurlyeq (s(\lambda x.x)) \to a_{ij}$. If $i = j$, this follows from the fact that $(s(\lambda x.x)) \preccurlyeq a_{ii}$. If $i \neq j$, this is clear by subtyping.

From right to left, if there exists $s \in \mathcal{S}$ such that for any $i,j \in I$, $s \preccurlyeq i =_I j \to a_{ij}$, then in particular for all $i \in I$ we have $s \preccurlyeq (\lambda x.x) \to a_{ii}$, and then $\lambda\_.s(\lambda x.x) \preccurlyeq \top \to a_{ii}$ which concludes the case.

- For each projection $\pi^1_{I\times J} : I \times J \to I$ in C, the monotone function $\mathcal{T}(\pi^1_{I,J}) : \mathcal{T}(I) \to \mathcal{T}(I\times J)$ has both a left adjoint $(\exists J)_I$ and a right adjoint $(\forall J)_I$ which are defined by:

$$(\forall J)_I\Big(\big[(a_{ij})_{i,j\in I\times J}\big]\Big) \triangleq \big[(\bigvee_{j\in J} a_{ij})_{i\in I}\big] \qquad\qquad (\exists J)_I\Big(\big[(a_{ij})_{i,j\in I\times J}\big]\Big) \triangleq \big[(\bigexists_{j\in J} a_{ij})_{i\in I}\big]$$

The proofs of the adjointness of this definition are again easy manipulation of λ-calculus. We only give the case of $\exists$, the case for $\forall$ is easier. We need to show that for any $[(a_{ij})_{(i,j)\in I\times J}] \in \mathcal{T}(I\times J)$ and for any $[(b_i)_{i\in I}]$, we have:

$$[(a_{ij})_{(i,j)\in I\times J}] \leq_{\mathcal{S}[I\times J]} [(b_i)_{(i,j)\in I}] \qquad \Leftrightarrow \qquad \big[(\bigexists_{j\in J} a_{ij})_{i\in I}\big] \leq_{\mathcal{S}[I]} [(b_i)_{i\in I}]$$

Let us fix some $[a]$ and $[b]$ as above. From left to right, assume that there exists $s \in \mathcal{S}$ such that for all $i \in I, j \in J$, $s \preccurlyeq a_{ij} \to b_i$, and thus $sa_{ij} \preccurlyeq b_i$. Using the semantic elimination rule of the existential quantifier, we deduce that for all $i \in I$, if $t \preccurlyeq \exists_{j\in J} a_{ij}$, then $t(\lambda x.sx) \preccurlyeq b_i$. Therefore, for all $i \in I$ we have $\lambda y.y(\lambda x.sx) \preccurlyeq \exists_{j\in J} a_{ij} \to b_i$.

From right to left, assume that there exists $s \in \mathcal{S}$ such that for all $i \in I$, $s \preccurlyeq \exists_{j\in J} a_{ij} \to b_i$. For any $j \in J$, using the semantic introduction rule of the existential quantifier, we deduce that for all $i \in I$, $\lambda x.xa_{ij} \preccurlyeq \exists_{j\in J} a_{ij}$. Therefore, for all $i \in I$ we have $\lambda x.s(\lambda z.zx) \preccurlyeq a_{ij} \to b_i$.

- These adjoints clearly satisfy the Beck-Chevaley condition. For instance, for the existential quantifier, we have for all I, I', J, for any $[(a_{i'j})_{(i',j)\in I'\times J}] \in \mathcal{T}(I' \times J)$ and any $s : I \to I'$,

$$\begin{aligned}(\mathcal{T}(s) \circ (\exists J)_{I'})([(a_{i'j})_{(i',j)\in I'\times J}]) &= \mathcal{T}(s)(\big[(\exists_{j\in J} a_{i'j})_{i'\in I'}\big]) \\ &= \big[(\exists_{j\in J} a_{s(i)j})_{i\in I}\big] \\ &= ((\exists J)_I)([(a_{s(i)j})_{ij\in I\times J}]) \\ &= ((\exists J)_I \circ \mathcal{T}(s \times \mathtt{id}_J)([(a_{ij})_{i,j\in I\times J}])\end{aligned}$$

- Finally, we define $\mathtt{Prop} \triangleq \mathscr{A}$ and verify that $\mathtt{tr} \triangleq [\mathtt{id}_{\mathscr{A}}] \in \mathcal{T}(\mathtt{Prop})$ is a generic predicate. Let then I be a set, and $a = [(a_i)_{i\in I}] \in \mathcal{T}(I)$. We let $\chi_a : i \mapsto a_i$ be the characteristic function of a (it is in $I \to \mathtt{Prop}$), which obviously satisfies that for all $i \in I$:

$$\mathcal{T}(\chi_a)(\mathtt{tr}) = [(\chi_a(i))_{i\in I})] = [(a_i)_{i\in I}]$$

□

10.4.4 Relation with forcing triposes

10.4.4.1 The fundamental diagram

We shall now briefly present a criterion to determine whether an implicative tripos is equivalent to a forcing tripos. By forcing tripos, we refer to a tripos of the shape $\mathcal{T} : I \mapsto \mathcal{H}^I$ where $\mathcal{H}$ is a complete

Heyting (or Boolean in classical case) algebra (see Example 9.22). In particular, recall that in the case of forcing (see Section 9.1.2), we have:

$$\forall = \curlywedge = \wedge$$

while it is worth observing that the definition of the implicative tripos is in adequacy with the usual situation of in realizability, that is to say that we have:

$$\forall = \curlywedge \qquad\qquad \wedge = \times$$

In the case of the implicative tripos, the algebra $\mathcal{T}(I)$ of predicates associated to the set I is defined by $\mathcal{T}(I) = \mathcal{A}^I/\mathcal{S}[I]$, that is: as the quotient of the power implicative algebra $\mathcal{A}^I$ by the uniform power separator $\mathcal{S}[I]$. Note that here, we used the uniform power separator $\mathcal{S}[I]$ and not the pointwise power separator $\mathcal{S}^I$, precisely to avoid a trivialization of the form $\mathcal{A}^I/\mathcal{S}^I = (\mathcal{A}/S)^I$ that would amount to a forcing tripos, based on the Heyting algebra $\mathcal{H} = \mathcal{A}/\mathcal{S}$.

Indeed, we saw in Section 10.4.2 that the separator product $\mathcal{S}^I$ also defines a separator for the algebra $\mathcal{A}^I$. We could have considered instead the quotient $\mathcal{A}^I/\mathcal{S}^I$. Since $\mathcal{S}[I] \subseteq \mathcal{S}^I$, in particular we have that if a and b are two elements of $\mathcal{A}^I$ and if besides $a \cong_{S[I]} b$, then $a \cong_{S^I} b$. In other words, the map which associates to each equivalence class w.r.t. $\mathcal{S}[I]$ the equivalence class of its representative w.r.t. $\mathcal{S}^I$:

$$\iota_I : \left\{ \begin{array}{ccc} \mathcal{A}^I/\mathcal{S}[I] & \to & \mathcal{A}^I/\mathcal{S}^I \\ {[a]}_{/\mathcal{S}[I]} & \mapsto & {[a]}_{/\mathcal{S}^I} \end{array} \right.$$

is surjective onto $\mathcal{A}^I/\mathcal{S}^I$.

Moreover, we could have directly defined a tripos by taking the quotient $\mathcal{A}/\mathcal{S}$ (which defines a Heyting algebra $\mathcal{H}$), and considered the functor which associates to each I the product $(\mathcal{A}/\mathcal{S})^I$. This situation corresponds precisely to a forcing tripos. Here again, we can define the map which associates to each equivalence class $[(a_i)_{i\in I}]$ w.r.t. $\mathcal{S}[I]$ the sequence of equivalence classes of the a_i w.r.t. $\mathcal{S}$:

$$\rho_I : \left\{ \begin{array}{ccc} \mathcal{A}^I/\mathcal{S}[I] & \to & (\mathcal{A}/\mathcal{S})^I \\ {[a]}_{/\mathcal{S}[I]} & \mapsto & {[a_i]}_{/\mathcal{S}} \end{array} \right.$$

which is surjective onto $(\mathcal{A}/\mathcal{S})^I$. Finally, it is clear that $\mathcal{A}^I/\mathcal{S}^I$ and $(\mathcal{A}/\mathcal{S})^I$ are in bijection: in $\mathcal{A}^I/\mathcal{S}^I$, two elements $[(a_i)_{i\in I}]$ and $[(v_i)_{i\in I}]$ are in the same equivalence class if they are equivalent componentwise, that is for all $i \in I$, a_i and b_i are equivalent:

$$[(a_i)_{i\in I}] \cong_{\mathcal{S}^I[(b_i)_{i\in I}]} \quad\Leftrightarrow\quad \forall i \in I.[a_i] \cong_{\mathcal{S}} [b_i]$$

Denoting by ϱ_I the corresponding bijection from $\mathcal{A}^I/\mathcal{S}^I$ to $(\mathcal{A}/\mathcal{S})^I$, the situation can then be summarized by the following diagram:

$$\begin{tikzcd}
\mathcal{A}^I \arrow[r, two heads, "{[\cdot]_{/\mathcal{S}[I]}}"] \arrow[d, two heads, "{[\cdot]_{/\mathcal{S}^I}}"'] & \mathcal{A}^I/\mathcal{S}[I] \arrow[dl, two heads, "\iota_I"] \arrow[d, two heads, "\rho_I"] & = \mathcal{T}(I) \\
\mathcal{A}^I/\mathcal{S}^I \arrow[r, two heads, "\sim", "\varrho_I"'] & (\mathcal{A}/\mathcal{S})^I & = \mathcal{T}(1)^I
\end{tikzcd}$$

In this diagram, all the maps are surjective, the top right corner corresponds to the implicative tripos while the bottom right one corresponds to a forcing tripos. We shall now make use of the diagram to precise the situation. To this purpose, we first need to prove a lemma about morphisms of Heyting algebras.

Lemma 10.50. *Let $\mathcal{H}$,$\mathcal{H}'$ be two Heyting algebras. If $f : \mathcal{H} \to \mathcal{H}'$ be a morphism of Heyting algebras, then f is an isomorphism if and only if f is bijective.*

Proof. The left to right implication is trivial, we thus have to prove that if f is a one-to-one morphism, then f^{-1} is a morphism. It is easy to see that f^{-1} preserves the lattice structure and the implication because f does. For instance for the preservation of meets, for all $a, b \in \mathcal{H}'$ we have:

$$f^{-1}(a \curlywedge b) = f^{-1}(f(f^{-1}(a)) \curlywedge f(f^{-1}(b))) = f^{-1}(f(f^{-1}(a) \curlywedge f^{-1}(b))) = f^{-1}(a) \curlywedge f^{-1}(b)$$

As for the preservation of the order, if $a, b \in \mathcal{H}'$ are such that $a \preccurlyeq b$, then $a = a \curlywedge b$ and we have:

$$f^{-1}(a) = f^{-1}(a \curlywedge b) = f^{-1}(a) \curlywedge f^{-1}(b) \preccurlyeq f^{-1}(b)$$

Therefore, we can conclude that $f^{-1}(a) \preccurlyeq f^{-1}(b)$. □

Using the previous lemma, we obtain the following characterization:

Proposition 10.51. *The following are equivalent:*

1. *The map: $\rho_I : (\mathcal{A}^I/\mathcal{S}[I]) \to (\mathcal{A}/\mathcal{S})^I$ is an isomorphism (of Heyting Algebras).*
2. *The map: $\rho_I : (\mathcal{A}^I/\mathcal{S}[I]) \to (\mathcal{A}/\mathcal{S})^I$ is injective.*
3. *$\mathcal{S}[I] = \mathcal{S}^I$.*
4. *The separator $\mathcal{S} \subseteq A$ is closed under all I-indexed meets.*

Proof. The equivalence between the first three conditions follows from the above characterization of isomorphisms in **HA**. If $(a_i)_{i\in I} \in \mathcal{S}^I$ and $\mathcal{S}[I] = \mathcal{S}^I$, then there exists an $s \in \mathcal{S}$ such that for all $i \in I$, $s \preccurlyeq a_i$. Then $s \preccurlyeq \bigcurlywedge_{i\in I} a_i$ and the latter belongs to $\mathcal{S}$ by upward closure. Therefore, $\mathcal{S}$ is closed under I-indexed meets. For the converse direction, it suffices to see that if $(a_i)_{i\in I} \in \mathcal{S}^I$, then by closure under I-indexed meets $\bigcurlywedge_{i\in I} a_i$ is in $\mathcal{S}$ and is a uniform realizer for $(a_i)_{i\in I}$, which thus belongs to $\mathcal{S}[I]$. □

This diagram is thus the cornerstone on the study of implicative tripos. In particular, the most interesting realizability models (*i.e.* those which can not be obtained by forcing) are the ones occurring in the top right corner when the map ρ_I is not an isomorphism.

10.4.4.2 Collapse criteria

We shall briefly present some criteria which characterizes the situations where implicative triposes are isomorphic to forcing triposes. As we do not want to enter into too much detail here (we leave it for the forthcoming paper of Alexandre Miquel on the topic), let us loosely use notions that we do not formally define. Our goal here is mainly to give some intuitions, and to highlight some phenomena that were already known in Krivine realizability algebras.

First of all, as we mentioned in Section 3.5.3, the construction of Krivine's realizability models for the negation of the axiom of choice and the continuum hypothesis deeply relies on the fact that the formula $\textsc{ind} \equiv \forall x.\mathrm{Nat}(x)$ is not realized. In our framework, this formula can be defined by:

$$\textsc{ind}^{\mathcal{A}} \triangleq \bigcurlywedge_{n\in\mathbb{N}} \bigcurlywedge_{a\in\mathcal{A}^{\mathbb{N}}} (a_0 \to \bigcurlywedge_{i\in\mathbb{N}} (a_i \to a_{i+1}) \to a_n)$$

In fact, this can be reduced to the formula called *parallel-or* (**p-or**), which is defined in any implicative structure by:

$$\textbf{p-or}^{\mathcal{A}} \triangleq (\top \to \bot \to \bot) \curlywedge (\bot \to \top \to \bot)$$

in the sense that if this formula is realized if and only if IND is[7]. Besides, in the case where the realizability algebra (*i.e.* the λ_c-calculus) contains an instruction ⫛ of non-deterministic choice, it is easy to define a realizer for the formula **p-or**. In which case, the realizability models collapses to a forcing model.

This phenomenon can be rephrased directly within implicative algebras. First, the operator ⫛ is naturally interpreted in any implicative structure $\mathcal{A}$ by:

$$⫛^{\mathcal{A}} \triangleq \bigcurlywedge_{a,b\in A} (a \to b \to a \curlywedge b)$$

and it is an easy exercise of λ_c-calculus to show that:

Proposition 10.52. *If* $(\mathcal{A},\preccurlyeq,\to,\mathcal{S})$ *is a classical implicative algebra, then:*

$$⫛^{\mathcal{A}} \cong_{\mathcal{S}} \textit{\textbf{p-or}}^{\mathcal{A}} \cong_{\mathcal{S}} \textit{IND}^{\mathcal{A}}$$

Then it is possible to show that an implicative tripos is isomorphic to a forcing tripos if and only if its separator contains $⫛^{\mathcal{A}}$ and is *finitely generated* (*i.e.* it is defined as the closure under application and upwards of a finite subset of the implicative structure $\mathcal{A}$).

Theorem 10.53 (Characterization of forcing triposes). *Let* $\mathcal{T} : \mathbf{Set}^{op} \to \mathbf{HA}$ *be an implicative tripos induced by an implicative algebra* $(\mathcal{A},\preccurlyeq,\to,\mathcal{S})$*. The following are equivalent:*

1. *The tripos* $\mathcal{T}$ *is isomorphic to a forcing tripos*
2. *The separator* $\mathcal{S} \subseteq \mathcal{A}$ *is a principal filter of* $\mathcal{A}$*.*
3. *The separator* $\mathcal{S} \subseteq \mathcal{A}$ *is finitely generated and* $⫛ \in \mathcal{S}$*.*

Proof. See [121]. □

Furthermore, in the case where the arrow commutes with arbitrary joins, that is if for all $b \in \mathcal{A}$ the following holds:

$$\bigcurlywedge_{a\in\mathcal{A}} (a \to b) = (\bigcurlyvee_{a\in\mathcal{A}} a) \to b$$

the interpretation of **p-or** belongs to any separator. Indeed, since $\bot = \bigcurlyvee \emptyset$, the previous equality implies that $\bot \to a = \top$ for any $a \in \mathcal{A}$, and in particular:

$$\mathbf{p\text{-}or}^{\mathcal{A}} = (\bot \to \top \to \bot) \curlywedge (\top \to \bot \to \bot) = \top \curlywedge (\top \to \top) = \top \to \top$$

Therefore, in the previous situation, $⫛^{\mathcal{A}}$ and $\textsc{ind}^{\mathcal{A}}$ also belong to all classical separators. The previous equation is not meaningless, because when it holds, it allows to define the existential quantifier as a join, and it can be read as:

$$\bigforall_{a\in\mathcal{A}} (a \to b) = (\bigexists_{a\in\mathcal{A}} a) \to b$$

In other words, we can not expect an implicative algebra which is "too" commutative to induce triposes which are not isomorphic to a forcing tripos.

[7] In Krivine's article, the fact that the algebra ∇_2 is not trivial precisely relies on the fact that there is no term which realizes both $\top \to \bot \to \bot$ and $\bot \to \top \to \bot$ (see [99, Theorem 31]).

10.5 Conclusion

We presented in this section the concept of *implicative algebra*, that relies on the primitive notion of implicative structure. These structures are defined as a particular class of meet-complete lattices equipped with an arrow, where this arrow satisfies commutations with arbitrary meets which are the counterpart of the logical commutation between the universal quantification and the implication. We showed that implicative structures are a generalization Streicher's AKSs and Ferrer *et al.*'s $^{\mathcal{K}}$OCAs. In particular, they allow us to defined triposes arising from Krivine classical realizability models, and they provide us with simple criteria to determine whether the induced triposes are equivalent to forcing triposes. As such, implicative algebras appear to be a promising framework for the algebraic analysis of classical realizability.

This presentation is totally in line with Krivine's usual presentation of his realizability models, and in particular it takes position on a presentation of logic through universal quantification and the implication. The computational counterpart of this choice is that the presentation relies on the call-by-name λ_c-calculus. This raises the question of knowing whether it is possible to have alternative presentations with similar structures based on different connectives (and thus different calculi).

In the last two chapters of this manuscript, we will present an attempt in this perspective. Firstly, we will introduce the so-called notion of *disjunctive algebras*, which are primitively defined in disjunctive structures relying on a disjunction $⅋$ and a negation $\neg$. We will relate these connectives to a fragment of Munch-Maccagnoni's System L, which amounts to a call-by-name decomposition of the λ-calculus. In particular, we will see that any disjunctive algebra induces an implicative algebra.

Secondly, we will introduce the dual notion of *conjunctive algebras*, based on conjunctive structures whose connectives are a conjunction $\otimes$ and a negation $\neg$. Here again, this decomposition of the arrow corresponds to a fragment of Munch-Maccagnoni's System L, which amounts to a call-by-value λ-calculus. We will see that such a structure can naturally be obtained by duality from a disjunctive algebra.

These two different presentations are not as accomplished as the study of implicative algebras. In particular, we do not dispose of the full embeddings of the corresponding calculus, and we are still missing some correspondences between the three presentations. Yet, they should rather be taken as a first step toward a complete zoology of the implicative-like algebras. We conjecture that implicative algebras constitute the more general framework.

11- Disjunctive algebras

We shall now introduce the notion of disjunctive algebra, which is a structure primarily based on disjunction, negation (for the connectives) and meets (for the universal quantifier). Our main purpose is to draw the comparison with implicative algebras, as an attempt to justify eventually that the latter are at least as general as the former. All along this chapter, we will follow the same rationale which guided the definition of implicative structures, separators, etc... If we will not be able to recover all the disjunctive counterpart of the properties of implicative algebras, we should anyway be convinced in the end that disjunctive algebras do not bring any benefits over the implicative one, in the sense that disjunctive algebras are particular cases of implicative algebras.

The first step in this direction is the definition of disjunctive structures. Our starting point is the fact that in classical logic, the following equivalence holds for all formulas A and B:

$$A \to B \qquad \Leftrightarrow \qquad \neg A \vee B$$

In particular, this equivalence suggests that as long as we are interested in a classical framework, we could as well define the logic with the disjunction and negation as ground connectives. This is for instance the choice of Bourbaki in his *Éléments de mathématique* [21]. The first volume of the famous treatise begins precisely with the introduction of the logical symbols, which are ¬, ∨ plus two others used to handle substitutions. The first symbolic shorthand which is defined is precisely the implication, and logic is axiomatized by the following schemes:

$$\begin{array}{lcl@{\qquad\qquad}lcl} S1 & : & (A \vee A) \to A & S3 & : & (A \vee B) \to (B \vee A) \\ S2 & : & A \to (A \vee B) & S4 & : & (A \to B) \to ((C \vee A) \to (C \vee B)) \end{array}$$

These logical schemes should give us a guideline in the definition of separators for disjunctive structures.

In the seminal paper introducing linear logic [58], Jean-Yves Girard refines the structure of the sequent calculus LK, introducing in particular two connectives for the disjunctions: ⅋ and ⊕. The first one is said to be multiplicative, while the second one is said to be additive, due to the treatment of contexts in the corresponding rules:

$$\frac{\Gamma \vdash A_1, A_2, \Delta}{\Gamma \vdash A_1 \parr A_2, \Delta}\,(\parr_r) \quad \frac{\Gamma_1, A \vdash \Delta_1 \quad \Gamma_2, B \vdash \Delta_2}{\Gamma_1, \Gamma_2, A_1 \parr A_2 \vdash \Delta_1, \Delta_2}\,(\parr_l) \qquad \frac{\Gamma \vdash A_i, \Delta}{\Gamma \vdash A_1 \oplus A_2, \Delta}\,(\oplus_r) \quad \frac{\Gamma, A_1 \vdash \Delta \quad \Gamma, A_2 \vdash \Delta}{\Gamma, A_1 \parr A_2 \vdash \Delta}\,(\oplus_l)$$

In the (multiplicative) rules for ⅋, contexts are indeed juxtaposed, while they are identified in the (additive) rule for ⊕. With this finer set of connectives, Girard shows that the usual implication[1] can be retrieved using the multiplicative disjunction:

$$A \to B \quad \triangleq \quad \neg A \parr B$$

[1] To do justice to Girard's approach, the implication which is considered in linear logic, written ⊸, is different from the usual one. The difference between both implications is not relevant in our framework.

Dually to these two connectives for the disjunction, two connectives are also introduced for the conjunction! $\otimes$ (multiplicative) and & (additive). Disjunctive and conjunctive connectives are related through some laws of duality which are very similar to De Morgan's laws for classical logic. For instance, the multiplicative connectives verify that $\neg(A \parr B) = \neg A \otimes \neg B$ and $\neg(A \otimes B) = \neg A \parr \neg B$. In particular, this give rises to a second decomposition of the arrow:

$$A \to B \quad \triangleq \quad \neg(A \otimes \neg B)$$

In 2009, Guillaume Munch-Maccagnoni gave a computational account of Girard's presentation for classical logic with a division between multiplicative and additive connectives [126]. In his calculus, named L, each connective corresponds to the type of a particular constructor (or destructor). While L is in essence close to Curien and Herbelin's $\lambda\mu\tilde{\mu}$-calculus (in particular it is presented with the same paradigm of duality between proofs and contexts), the syntax of terms does not include λ-abstraction (and neither does the syntax of formulas includes an implication). The two decompositions of the arrow evoked above are precisely reflected in a decomposition of λ-abstractions (and dually, of stacks) in terms of L constructors.

Notably, the choice of a decomposition corresponds to a particular choice of an evaluation strategy for the encoded λ-calculus. When picking the $\parr$ connective, the corresponding λ-terms are evaluated according to a call-by-name evaluation strategy whose machinery resembles the one of the call-by-name $\lambda\mu\tilde{\mu}$-calculus (see Chapter 4). On the other hand, if the implication is defined through the $\otimes$ connective, the corresponding λ-calculus is reduced in a call-by-value fashion.

We shall begin by considering the call-by-name case, which is closer to the situation of implicative algebras. We start with the presentation of the corresponding fragment of Munch-Maccagnoni's calculus, which we call $\mathrm{L}^{\parr}$. In particular, we will see how this calculus induces a realizability model whose structure leads us to the definition of *disjunctive structures.* We will observe that the encoding of λ-terms into $\mathrm{L}^{\parr}$ can be directly reflected as an implicative structure induced by each disjunctive structure. Finally, we shall define the notions of (disjunctive) *separator* and *disjunctive algebra.* We will see that, again, any disjunctive algebra can be viewed as an implicative algebra.

11.1 The $\mathrm{L}^{\parr}$ calculus

We present here the fragment of L induced by the negative connectives $\parr$, $\neg$ and $\forall$, in order to present afterwards the realizability model it induces. Since this calculus has a lot of similarities with respect to the $\lambda\mu\tilde{\mu}$-calculus, and since the realizability interpretation is akin to the one we gave for the call-by-name $\lambda\mu\tilde{\mu}$-calculus (see Section 4.4.5), we shall try to be concise. In particular, we skip some proofs which can be found either in [126] or in previous chapters.

11.1.1 The $\mathrm{L}^{\parr}$ calculus

The $\mathrm{L}^{\parr}$-calculus is a subsystem of Munch-Maccagnoni's system L [126], restricted to the negative fragment corresponding to the connectives $\parr$, $\neg^-$ (which we simply write $\neg$ since there is no ambiguity here) and $\forall$. To ease the connection with the syntax of the $\lambda\mu\tilde{\mu}$-calculus, we slightly change the notations of the original paper. The syntax is given by:

$$\begin{array}{llcl}
\textbf{Contexts} & e^+ & ::= & \alpha \mid (e_1^+, e_2^+) \mid [t^-] \mid \mu x.c \\
\textbf{Terms} & t^- & ::= & x \mid \mu(\alpha_1,\alpha_2).c \mid \mu[x].c \mid \mu\alpha.c \\
\textbf{Commands} & c & ::= & \langle t^- \| e^+ \rangle
\end{array}$$

Observe in particular that we only have positive contexts and negative terms. We write $\mathcal{E}$ for the set of contexts, $\mathcal{T}$ for the set of terms, C for the set of commands, and $\mathcal{E}_0$, $\mathcal{T}_0$, C_0 for the sets of closed contexts, terms and commands. As for values, they are defined by the following fragment of the syntax[2]:

Values $\qquad V ::= \alpha \mid (V_1,V_2) \mid [t^-]$

We denote by $\mathcal{V}_0$ the corresponding set of closed values.

Since the notations might be a bit confusing regarding the ones we used in previous chapters (especially with respect to the $\lambda\mu\tilde{\mu}$-calculus), we shall say a few words about it:

- (e^+,e^+) are pairs of positive contexts, which we will relate to usual stacks;
- $\mu(\alpha_1,\alpha_2).c$, which binds the co-variables α_1,α_2, is the dual destructor;
- $[t^-]$ is a constructor for the negation, which allows us to embed a negative term into a positive context;
- $\mu[x].c$, which binds the variable x, is the dual destructor;
- $\mu\alpha.c$ and $\mu x.c$ correspond respectively to $\mu\alpha$ and $\tilde{\mu}x$ in the $\lambda\mu\tilde{\mu}$-calculus.

Remark 11.1 (Notations)**.** We shall explain that in (full) L, the same syntax allows us to define terms t and contexts e (thanks to the duality between them). In particular, no distinction is made between t and e, which are both written t, and commands are indifferently of the shape $\langle t^+\|t^-\rangle$ or $\langle t^-\|t^+\rangle$. For this reason, in [126] is considered a syntax where a notation $\bar{x}$ is used to distinguish between the positive variable x (that can appear in the left-member $\langle x|$ of a command) and the positive co-variable $\bar{x}$ (resp. in the right member $|x\rangle$ of a command). In particular, the $\mu\alpha$ binder of the $\lambda\mu\tilde{\mu}$-calculus would have been written $\mu\bar{x}$ and the $\tilde{\mu}x$ binder would have been denoted by $\mu\alpha$ (see [126, Appendix A.2]). We thus switched the x and α of L (and removed the bar), in order to stay coherent with the notations in the rest of this manuscript. ⌟

The reduction rules correspond to what could be expected from the syntax of the calculus: destructors reduce in front of the corresponding constructors, both μ binders catch values in front of them and pairs of contexts are expanded if they are not values. As for the η-expansion rules, they are also quite natural:

$$\begin{array}{rcl}
\langle \mu[x].c\|[t]\rangle & \to_\beta & c[t/x] \\
\langle t\|\mu x.c\rangle & \to_\beta & c[t/x] \\
\langle \mu\alpha.c\|V\rangle & \to_\beta & c[V/\alpha] \\
\langle \mu(\alpha_1,\alpha_2).c\|(V_1,V_2)\rangle & \to_\beta & c[V_1/\alpha_1,V_2/\alpha_2] \\
\langle t\|(e,e')\rangle & \to_\beta & \langle \mu\alpha.\langle \mu\alpha'.\langle t\|(\alpha,\alpha')\rangle\|e'\rangle\|e\rangle
\end{array}
\qquad
\begin{array}{rcl}
c & \to_\eta & \langle \mu\alpha.c\|\alpha\rangle \\
c & \to_\eta & \langle \mu(\alpha_1,\alpha_2).c\|(\alpha_1,\alpha_2)\rangle \\
c & \to_\eta & \langle \mu[x].c\|[x]\rangle \\
c & \to_\eta & \langle x\|\mu x.c\rangle
\end{array}$$

where in the last $\to_\beta$ rule, $(e,e') \notin V$.

Finally, we shall present the type system of L$^{⅋}$. In the continuity of the presentation of implicative algebras, we are interested in a second-order settings. Formulas are then defined by the following grammar:

Formulas $\qquad A,B := X \mid A ⅋ B \mid \neg A \mid \forall X.A$

Once again, the type system is similar to the one of the $\lambda\mu\tilde{\mu}$-calculus, in the sense that it is presented in a sequent calculus fashion. We work with two-sided sequents, where typing contexts are defined as usual as finite lists of bindings between variable and formulas:

$$\Gamma ::= \varepsilon \mid \Gamma, x : A \qquad\qquad \Delta ::= \varepsilon \mid \Delta, \alpha : A$$

Sequents are of three kinds, as in the $\lambda\mu\tilde{\mu}$-calculus:

[2] The reader may observe that in this setting, values are defined as contexts, so that we may have called them *covalues* rather than values. We stick to this denomination to stay coherent with the terminology in Munch-Maccagnoni's paper [126].

$$\frac{\Gamma \vdash t : A \mid \Delta \quad \Gamma \mid e : A \vdash \Delta}{\langle t \| e \rangle : \Gamma \vdash \Delta}\,(\textsc{Cut})$$

$$\frac{(\alpha : A) \in \Delta}{\Gamma \mid \alpha : A \vdash \Delta}\,(ax\vdash) \qquad \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A \mid \Delta}\,(\vdash ax)$$

$$\frac{c : \Gamma, x : A \vdash \Delta}{\Gamma \mid \mu x.c : A \vdash \Delta}\,(\mu\vdash) \qquad \frac{c : \Gamma \vdash \Delta, \alpha : A}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\,(\vdash\mu)$$

$$\frac{\Gamma \mid e_1 : A \vdash \Delta \quad \Gamma \mid e_2 : B \vdash \Delta}{\Gamma \mid (e_1, e_2) : A \parr B \vdash \Delta}\,(\parr\vdash) \qquad \frac{c : \Gamma \vdash \Delta, \alpha_1 : A, \alpha_2 : B}{\Gamma \vdash \mu(\alpha_1, \alpha_2).c : A \parr B \mid \Delta}\,(\vdash\parr)$$

$$\frac{\Gamma \vdash t : A \mid \Delta}{\Gamma \mid [t] : \neg A \vdash \Delta}\,(\neg\vdash) \qquad \frac{c : \Gamma, x : A \vdash \Delta}{\Gamma \vdash \mu[x].c : \neg A \mid \Delta}\,(\vdash\neg)$$

$$\frac{\Gamma \mid e : A[B/X] \vdash \Delta}{\Gamma \mid e : \forall X.A \vdash \Delta}\,(\forall\vdash) \qquad \frac{\Gamma \vdash t : A \mid \Delta \quad X \notin FV(\Gamma, \Delta)}{\Gamma \vdash t : \forall X.A}\,(\vdash\forall)$$

Figure 11.1: Typing rules for the $\mathrm{L}_{\parr,\neg}$-calculus

- $\Gamma \vdash t : A \mid \Delta$ for typing terms,
- $\Gamma \mid e : A \vdash \Delta$ for typing contexts,
- $c : \Gamma \vdash \Delta$ for typing commands.

Just like both connectives $\parr$ and $\neg$ are reflected by a constructor and a destructor in the syntax, in the type system each connective corresponds to a left rule (the introduction rule, for typing the constructor) and to a right rule (the elimination rule, for typing the destructor), in addition to the usual rules for typing variables, μ binders and commands. The type system is given in Figure 11.1.

Remark 11.2 (Universal quantifier)**.** In L, the universal quantification is also reflected by constructors in the syntax. This has the benefits of avoiding the problems of value restriction for the introduction rule. In our particular setting, since all terms are values, the introduction rule does not cause any problem. Beyond that, the realizability model we are going to define is only a pretext to the introduction of disjunctive structures, in which we will interpret the universal quantification by meets. Thus, it would be meaningless for us to introduce a syntactic constructor for the universal quantifier. ⌟

Remark 11.3 (Multiplicativity)**.** We simplified a bit the type system of L to avoid structural rules. Therefore, the rule ($\parr\ \vdash$) uses the same contexts in both hypotheses and the conclusion, instead of juxtaposing contexts in the conclusion. Both presentations are equivalent since both type systems allow for weakening and contraction. ⌟

11.1.2 Embedding of the λ-calculus

Following Munch-Maccagnoni's paper [126, Appendix E], we can embed the λ-calculus into the $\mathrm{L}^{\parr}$-calculus. To this end, we are guided by the expected definition of the arrow:

$$A \to B \ \triangleq\ \neg A \parr B$$

It is easy to see that with this definition, a stack $u \cdot e$ in $A \to B$ (that is with u a term of type A and e a context of type B) is naturally defined as a shorthand for the pair $([u], e)$, which indeed inhabits the type $\neg A \parr B$. Starting from there, the rest of the definitions are straightforward:

$$\begin{array}{rcl} u\cdot e &\triangleq& ([u],e)\\ \mu([x],\beta).c &\triangleq& \mu(\alpha,\beta).\langle \mu[x].c\|\alpha\rangle \\ \lambda x.t &\triangleq& \tilde\mu([x],\beta).\langle t\|\beta\rangle\\ t\,u &\triangleq& \mu\alpha.\langle t\|u\cdot\alpha\rangle\end{array}$$

These definitions are sound with respect to the typing rules expected from the $\lambda\mu\tilde\mu$-calculus:

Proposition 11.4. *The following typing rules are admissible:*

$$\frac{\Gamma,x:A\vdash t:B}{\Gamma\vdash\lambda x.t:A\to B}\qquad \frac{\Gamma\vdash u:A\mid\Delta\quad \Gamma\mid e:B\vdash\Delta}{\Gamma\mid u\cdot e:A\to B\vdash\Delta}\qquad \frac{\Gamma\vdash t:A\to B\mid\Delta\quad\Gamma\vdash u:A\mid\Delta}{\Gamma\vdash t\,u:B\mid\Delta}$$

Proof. Each case is directly derivable from $L^{\parr}$ type system. We abuse the notation to denote by (*def*) a rule which simply consists in unfolding the shorthands defining the λ-terms.

- **Case** $\mu([x],\alpha).c$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma\mid\alpha:\neg A\vdash\Delta,\alpha:\neg A,\beta:B}{\scriptstyle(ax\vdash)}\qquad\dfrac{c:(\Gamma,x:A\vdash\Delta,\beta:B)}{\Gamma\vdash\mu[x].c:\neg A\mid\Delta,\beta:B}{\scriptstyle(\vdash\mu)}}{\langle\mu[x].c\|\alpha\rangle:(\Gamma\vdash\Delta,\alpha:\neg A,\beta:B)}{\scriptstyle(\textsc{Cut})}}{\Gamma\vdash\mu(\alpha,\beta).\langle\mu[x].c\|\alpha\rangle:\neg A\parr B\mid\Delta}{\scriptstyle(\vdash\parr)}}{\Gamma\vdash\mu([x],\beta).c:\neg A\parr B\mid\Delta}{\scriptstyle(\mathit{def})}$$

- **Case** $\lambda x.t$:

$$\dfrac{\dfrac{\dfrac{\Gamma,x:A\vdash t:B\mid\Delta\qquad\dfrac{}{\Gamma\mid\beta:B\vdash\Delta,\beta:B}{\scriptstyle(ax\vdash)}}{\langle t\|\beta\rangle:(\Gamma,x:A\vdash\beta:B,\Delta}{\scriptstyle(\textsc{Cut})}}{\Gamma\vdash\mu([x],\beta).\langle t\|\beta\rangle:\neg A\parr B\mid\Delta}{\scriptstyle(\vdash\parr)}}{\Gamma\vdash\lambda x.t:A\to B\mid\Delta}{\scriptstyle(\mathit{def})}$$

- **Case** $u\cdot e$:

$$\dfrac{\dfrac{\dfrac{\Gamma\vdash u:A\mid\Delta}{\Gamma\mid[u]:A\vdash\Delta}{\scriptstyle(\neg\vdash)}\qquad\Gamma\mid e:B\vdash\Delta}{\Gamma\mid([u],e):\neg A\parr B\vdash\Delta}{\scriptstyle(\parr\vdash)}}{\Gamma\mid u\cdot e:A\to B\vdash\Delta}{\scriptstyle(\mathit{def})}$$

- **Case** $t\,u$:

$$\dfrac{\dfrac{\dfrac{\Gamma\vdash t:A\to B\mid\Delta\qquad\dfrac{\Gamma\vdash u:A\mid\Delta\quad\dfrac{}{\Gamma\mid\alpha:B\vdash\Delta,\alpha:B}}{\Gamma\mid u\cdot\alpha:A\to B\vdash\Delta,\alpha:B)}}{\langle t\|u\cdot\alpha\rangle:(\Gamma\vdash\Delta,\alpha:B)}{\scriptstyle(\textsc{Cut})}}{\Gamma\vdash\mu\alpha.\langle t\|u\cdot\alpha\rangle:B\mid\Delta}{\scriptstyle(\vdash\mu)}}{\Gamma\vdash t\,u:B\mid\Delta}{\scriptstyle(\mathit{def})}$$

□

In addition, the above definitions of λ-terms induce the usual rules of β-reduction for the call-by-name evaluation strategy in the Krivine abstract machine (notice that in the KAM, all stacks are values):

Proposition 11.5 (β-reduction). *We have the following reduction rules:*

$$\begin{array}{rcll} \langle t\,u\|\pi\rangle &\to_\beta& \langle t\|u\cdot\pi\rangle & (\pi\in V^+)\\ \langle\lambda x.t\|u\cdot\pi\rangle &\to_\beta& \langle t[u/x]\|\pi\rangle & (\pi\in V^+)\end{array}$$

Proof. If $\pi \in V^+$, we have indeed:

$$\langle t\,u \| \pi\rangle = \langle \mu\alpha.\langle t \| u \cdot \alpha\rangle \| \pi\rangle \to_\beta \langle t \| u \cdot \pi\rangle$$

and:

$$\begin{aligned}
\langle \lambda x.t \| u \cdot \pi\rangle &= \langle \tilde{\mu}([x],\beta).\langle t \| \beta\rangle \| ([u],\pi)\rangle \\
&= \langle \mu(\alpha,\beta).\langle \alpha \| \mu[x].\langle t \| \beta\rangle\rangle \| ([u],\pi)\rangle \\
&\to_\beta \langle [u] \| \mu[x].\langle t \| \pi\rangle\rangle \\
&\to_\beta \langle t[u/x] \| \pi\rangle
\end{aligned}$$

□

At this stage, it is clear that the structure of $\mathrm{L}^{\parr}$ allows us to recover all the computational strength of the call-by-name $\lambda\mu\tilde{\mu}$-calculus. As we explained in Section 4.2.4, this also means that we can encode the term ***cc*** of the λ_c-calculus, and simulate the Krivine abstract machine. Therefore, $\mathrm{L}^{\parr}$ is suitable for the definition of a realizability interpretation through these encodings, but as for the full system L, we can also directly define a realizability model for $\mathrm{L}^{\parr}$.

11.1.3 A realizability model based on the $\mathrm{L}^{\parr}$-calculus

We briefly go through the definition of the realizability interpretation *à la* Krivine for $\mathrm{L}^{\parr}$. The reader should observe that in the end, this interpretation is very similar to the one of the call-by-name $\lambda\mu\tilde{\mu}$-calculus (see Section 4.4.5). As usual, we begin with the definition of a pole:

Definition 11.6 (Pole)**.** A subset $\perp\!\!\!\perp \in C$ is said to be saturated whenever for all $c,c' \in C$, if $c \to_\beta c'$ then $c \in \perp\!\!\!\perp$. A *pole* is defined as any saturated subset of C_0. ⌟

As it is common in Krivine's call-by-name realizability, falsity values are defined primitively as sets of contexts. Truth values are then defined by orthogonality to the corresponding falsity values. We say that a term t is *orthogonal* (with respect to the pole $\perp\!\!\!\perp$) to a context e we denote by $t \perp\!\!\!\perp e$ when $\langle t \| e\rangle \in \perp\!\!\!\perp$. A term t (resp. a context e) is said to be orthogonal to a set $S \subseteq \mathcal{E}_0$ (resp. $S \subseteq \mathcal{T}_0$), which we write $t \perp\!\!\!\perp S$, when for all $e \in S$, t is orthogonal to e.

Orthogonality satisfies the expected properties of monotonicity:

Proposition 11.7 (Monotonicity)**.** *For any subset S of $\mathcal{T}_0$ (resp. $\mathcal{E}_0$) and any subset $\mathcal{U} \in \mathcal{P}(\mathcal{T}_0)$ (resp. any subset of $\mathcal{P}(\mathcal{E}_0)$), the following holds:*

1. $S \subseteq S^{\perp\!\!\!\perp\perp\!\!\!\perp}$
2. $S^{\perp\!\!\!\perp} = S^{\perp\!\!\!\perp\perp\!\!\!\perp\perp\!\!\!\perp}$
3. $(\bigcap_{S\in\mathcal{U}} S)^{\perp\!\!\!\perp} = \bigcup_{S\in\mathcal{U}}(S^{\perp\!\!\!\perp})$
4. $(\bigcup_{S\in\mathcal{U}} S)^{\perp\!\!\!\perp} \supseteq \bigcap_{S\in\mathcal{U}}(S^{\perp\!\!\!\perp})$

As we explained in more details in chapter 4, the realizability interpretation *à la* Krivine of a calculus given in a sequent calculus presentation (that is whose reduction rules are presented in the shape of an abstract machine) can be derived mechanically from a small-step reduction system. We will not do it in the present case, but it amounts to the case of the call-by-name $\lambda\mu\tilde{\mu}$-calculus. Because of this evaluation strategy (which is induced here by the choice of connectives), a formula A is primitively interpreted by its "falsity value of values", which we write $\|A\|_V$ and call *primitive falsity value*, which is a set in $\mathcal{P}(\mathcal{V}_0)$ (and thus in $\mathcal{P}(\mathcal{E}_0)$). Its *truth value* $|A|$ is then defined by orthogonality to $\|A\|_V$ (and is a set in $\mathcal{P}(\mathcal{T}_0)$), while its *falsity value* $\|A\| \in \mathcal{P}(\mathcal{E}_0)$ is again obtained by orthogonality to $|A|$. Therefore, a universal formula $\forall X.A$ is interpreted by the union over all the possible instantiations for the primitive falsity value of the variable X by a set $S \in \mathcal{P}(\mathcal{V}_0)$. As it is usual in Krivine realizability, to ease the definitions we assume that for each subset S of $\mathcal{P}(\mathcal{V}_0)$, there is a constant symbol $\dot{S}$ in the syntax. The interpretation is given by:

$$\begin{array}{rcl}
\|\dot{S}\|_V & \triangleq & S \\
\|\forall X.A\|_V & \triangleq & \bigcup_{S\in\mathcal{P}(\mathcal{V}_0)} \|A\{X := \dot{S}\}\|_V \\
\|A \parr B\|_V & \triangleq & \{(V_1,V_2) : V_1 \in \|A\|_V \land V_2 \in \|B\|_V\} \\
\|\neg A\|_V & \triangleq & \{[t] : t \in |A|\} \\
|A| & \triangleq & \{t : \forall V \in \|A\|_V, t \perp\!\!\!\perp V\} \\
\|A\| & \triangleq & \{e : \forall t \in |A|, t \perp\!\!\!\perp e\}
\end{array}$$

Remark 11.8. One could alternatively prefer to consider the following primitive falsity value:

$$\|A \parr B\|_V \triangleq \{(e_1,e_2) : e_1 \in \|A\| \land e_2 \in \|B\|\}$$

As highlighted by Dagand and Scherer [35], the design choice for primitive falsity value results in constraints on the definition of the reduction rules to make them adequate with the definitions. A short Coq development on the proof of adequacy of $\mathrm{L}^{\parr}$ typing rules (for the propositional fragment) viewed as an evaluating machine is given to support this claim[3]. In particular, it makes very clear the impact that the choice of definition for $\|A \parr B\|_V$ has on the reduction system. $\lrcorner$

We shall now verify that the type system of $\mathrm{L}^{\parr}$ is indeed adequate with this interpretation. We first prove the following simple lemma:

Lemma 11.9 (Substitution)**.** *Let A be a formula whose only free variable is X. For any closed formula B, if $S = \|B\|_V$, then $\|A[B/X]\|_V = \|A[\dot{S}/X]\|_V$.*

Proof. Easy induction on the structure of formulas, with the observation that the statement for primitive falsity values implies the same statement for truth values ($|A[B/X]| = |A[\dot{S}/X]|$) and falsity values ($\|A[B/X]\| = \|A[\dot{S}/X]\|$). The key case is for the atomic formula $A \equiv X$, where we easily check that:

$$\|X[B/X]\|_V = \|B\|_V = S = \|\dot{S}\|_V = \|X[\dot{S}/X]\|_V$$

□

The last step before proving adequacy consists in defining substitutions and valuations. We say that a *valuation*, which we write ρ, is a function mapping each second-order variable to a primitive falsity value $\rho(X) \in \mathcal{P}(\mathcal{V}_0)$. A *substitution*, which we write σ, is a function mapping each variable x to a closed term c and each variable α to a closed value $V \in \mathcal{V}_0$:

$$\sigma ::= \varepsilon \mid \sigma, x \mapsto t \mid \sigma, \alpha \mapsto V^+$$

We say that a substitution σ realizes a context Γ and note $\sigma \Vdash \Gamma$ when for each binding $(x : A) \in \Gamma$, $\sigma(x) \in |A|$. Similarly, we say that σ realizes a context Δ if for each binding $(\alpha : A) \in \Delta$, $\sigma(\alpha) \in \|A\|_V$.

We can now state the property of adequacy of the realizability interpretation:

Proposition 11.10 (Adequacy)**.** *Let Γ, Δ be typing contexts, ρ be a valuation and σ be a substitution such that $\sigma \Vdash \Gamma[\rho]$ and $\sigma \Vdash \Delta[\rho]$. We have:*

1. *If V^+ is a positive value such that $\Gamma \mid V^+ : A \vdash \Delta$, then $V^+[\sigma] \in \|A[\rho]\|_V$.*
2. *If t is a term such that $\Gamma \vdash t : A \mid \Delta$, then $t[\sigma] \in |A[\rho]|$.*
3. *If e is a context such that $\Gamma \mid e : A \vdash \Delta$, then $e[\sigma] \in \|A[\rho]\|$.*
4. *If c is a command such that $c : (\Gamma \vdash \Delta)$, then $c[\sigma] \in \perp\!\!\!\perp$.*

Proof. The proof is almost the same as for the proof of adequacy for the call-by-name $\lambda\mu\tilde{\mu}$-calculus. We only give some key cases which are peculiar to this setting. We proceed by induction over the typing derivations. Let σ be a substitution realizing $\Gamma[\rho]$ and $\Delta[\rho]$.

[3] See `https://www.irif.fr/~emiquey/these/coq/Real.RealLPar.html`.

- **Case** ($\vdash \neg$). Assume that we have:

$$\frac{c : \Gamma, x : A \vdash \Delta}{\Gamma \vdash \tilde{\mu}[x].c : \neg A}\ (\vdash\neg)$$

and let $[t]$ be a term in $\|A[\rho]\|_V$, that is to say that $t \in |A[\rho]|$. We know by induction hypothesis that for any valuation $\sigma' \Vdash (\Gamma, x : A)[\rho]$, $c[\sigma'] \in \perp\!\!\!\perp$ and we want to show that $\mu[x].c[\sigma]\perp\!\!\!\perp[t]$. We have that:

$$\mu[x].c\perp\!\!\!\perp[t] \quad \rightarrow_\beta \quad c[\sigma][t/x] = c[\sigma, x \mapsto t]$$

hence it is enough by saturation to show that $c[\sigma][u/x] \in \perp\!\!\!\perp$. Since $t \in |A[\rho]|$, $\sigma[x \mapsto t] \Vdash (\Gamma, x : A)[\rho]$ and we can conclude by induction hypothesis. The cases for ($\mu\ \vdash$), ($\vdash\ \mu$) and ($\vdash\ \parr$) proceed similarly.

- **Cases** ($\neg \vdash$). Trivial by induction hypotheses.
- **Case** ($\parr\ \vdash$). Assume that we have:

$$\frac{\Gamma \mid e_1 : A \vdash \Delta \quad \Gamma \mid u : B \vdash \Delta}{\Gamma \mid (e_1, e_2) : A \parr B \vdash \Delta}\ (\parr\vdash)$$

Let then t be a term in $|(A \parr B)[\rho]|$, to show that $\langle t \| (e_1, e_2)\rangle \in \perp\!\!\!\perp$, we proceed by anti-reduction:

$$\langle t \| (e, e')\rangle \rightarrow_\beta \langle \mu\alpha.\langle \mu\alpha'.\langle t \| (\alpha, \alpha')\rangle \| e'\rangle \| e\rangle$$

It now easy to show, using the induction hypotheses for e and e' that this command is in the pole: it suffices to show that the term $\mu\alpha.\langle\mu\alpha'.\langle t\|(\alpha,\alpha')\rangle\|e'\rangle \in |A|$, which amounts to showing that for any value $V_1 \in \|A\|_V$:

$$\langle\mu\alpha.\langle\mu\alpha'.\langle t\|(\alpha,\alpha')\rangle\|V\rangle\|\rightarrow_\beta\rangle\langle\mu\alpha'.\langle t\|(V,\alpha')\rangle\|e'\rangle \in \perp\!\!\!\perp$$

Again this holds by showing that for any $V' \in |B|$,

$$\langle\mu\alpha'.\langle t\|(V,\alpha')\rangle\|V'\rangle \rightarrow_\beta \langle t\|(V,V')\rangle \in \perp\!\!\!\perp$$

- **Case** ($\vdash \forall$). Trivial.

- **Case** ($\forall\ \vdash$). Assume that we have:

$$\frac{\Gamma \mid e : A[B/X] \vdash \Delta}{\Gamma \mid e : \forall X.A \vdash \Delta}\ (\forall\vdash)$$

By induction hypothesis, we obtain that $e[\sigma] \in \|A[B/X][\rho]\|$; so that if we denote $\|B[\rho]\|_V \in \mathcal{P}(\mathcal{V}_0)$ by S, we have:

$$e[\sigma] \in \|A[\dot{S}/X]\| \subseteq \bigcup_{S\in\mathcal{P}(\mathcal{V}_0)} \|A[\dot{S}/X][\rho]\|_V^{\perp\!\!\!\perp\perp\!\!\!\perp} \subseteq (\bigcup_{S\in\mathcal{P}(\mathcal{V}_0)} \|A[\dot{S}/X][\rho]\|_V)^{\perp\!\!\!\perp\perp\!\!\!\perp} = \|\forall X.A[\rho]\|$$

where we make implicit use of Lemma 11.9. □

As a consequence of the former result and Proposition 11.4, we deduce that the typing rules for the encoded $\lambda\mu\tilde{\mu}$-rules also are adequate with the realizability interpretation.

Corollary 11.11. *The typing rules for $\lambda\mu\tilde{\mu}$-terms are adequate.*

In particular, this means that the realizability interpretation for $\mathrm{L}^{\parr}$ is a particular case of the one we define for the call-by-name $\lambda\mu\tilde{\mu}$-calculus in Section 11.1.3.

11.2 Disjunctive structures

Let us examine for a minute the situation to which we arrived. First, insofar as the call-by-name machinery of the $\lambda\mu\tilde{\mu}$-calculus was embeddable into $\mathrm{L}^{⅋}$, in particular the Krivine abstract machine for the λ_c-calculus can be recovered in this setting. Therefore, we could have used these embedding to make use of the realizability interpretation for the λ_c-calculus. Schematically, this would have corresponded to the following path:

$$\mathrm{L}^{⅋} \dashrightarrow \text{(Call-by-name) } \lambda\mu\tilde{\mu}\text{-calculus} \dashrightarrow \lambda_c\text{-calculus KAM} \dashrightarrow \text{Realizability model}$$

In particular, thinking of this construction from the point of view of implicative structures, this implies that we could have defined an implicative algebra by proceeding as follows:

$$\mathrm{L}^{⅋} \dashrightarrow \lambda_c\text{-calculus KAM} \dashrightarrow \text{Implicative structure} \dashrightarrow \text{Implicative algebra}$$

On the other hand, we saw in the previous section that the $\mathrm{L}^{⅋}$ calculus was suitable for the direct definition of a realizability model. The interpretation is induced by the reduction system of $\mathrm{L}^{⅋}$, which directly reflects the choice of connectives. Instead of embedding an arrow to obtain in the end an implicative structure, we should expect a direct algebraic counterpart for the structure of the calculus, and obtain a direct algebraic interpretation looking like:

$$\mathrm{L}^{⅋} \dashrightarrow \text{Disjunctive structure} \dashrightarrow \text{Disjunctive algebra}$$

Finally, we know that the realizability model obtained directly from the $\mathrm{L}^{⅋}$ calculus somehow contains the realizability model that would have been constructed with the arrow. In other words, the interpretation of $\mathrm{L}^{⅋}$ is a particular case of interpretation for a λ_c-calculus enriched with some additional structure. Therefore, we expect that, at the level of algebraic structures, any disjunctive algebra should induce an implicative algebra:

$$\text{Disjunctive algebra} \dashrightarrow \text{Implicative algebra}$$

11.2.1 Disjunctive structures

Following the rationale guiding the definition of implicative structure and algebras, we should now define the notion of *disjunctive structure.* Such a structure will then contain two internal laws to reflect the negation and the disjunction from the language of formulas. Regarding the expected commutations, as we choose negative connectives and in particular a universal quantifier, we should define commutations with respect to arbitrary meets. The following properties of the realizability interpretation for $\mathrm{L}^{⅋}$ provides us with a safeguard for the definition to come:

Proposition 11.12 (Commutations). *In any $L^{⅋}$ realizability model (that is to say for any pole $\perp\!\!\!\perp$), the following equalities hold:*

1. *If $X \notin FV(B)$, then $\|\forall X.(A ⅋ B)\|_V = \|(\forall X.A) ⅋ B\|_V$.*
2. *If $X \notin FV(A)$, then $\|\forall X.(A ⅋ B)\|_V = \|A ⅋ (\forall X.B)\|_V$.*
3. *$\|\neg(\forall X.A)\|_V = \bigcap_{S\in\mathcal{P}(\mathcal{V}_0)} \|\neg A\{X := \dot{S}\}\|_V$*

Proof. 1. Assume the $X \notin FV(B)$, then we have:

$$\begin{aligned}
\|\forall X.(A ⅋ B)\|_V &= \bigcup_{S\in\mathcal{P}(\mathcal{V}_0)} \|A\{X := \dot{S}\} ⅋ B\|_V \\
&= \bigcup_{S\in\mathcal{P}(\mathcal{V}_0)} \{(V_1,V_2) : V_1 \in \|A\{X := \dot{S}\}\|_V \wedge V_2 \in \|B\|_V\} \\
&= \{(V_1,V_2) : V_1 \in \bigcup_{S\in\mathcal{P}(\mathcal{V}_0)} \|A\{X := \dot{S}\}\|_V \wedge V_2 \in \|B\|_V\} \\
&= \{(V_1,V_2) : V_1 \in \|\forall X.A\|_V \wedge V_2 \in \|B\|\} \;=\; \|(\forall X.A) ⅋ B\|_V
\end{aligned}$$

2. Identical.
3. The proof is again a simple unfolding of the definitions:

$$\begin{aligned}\|\neg(\forall X.A)\|_V = \{[t] : t \in |\forall X.A|\} &= \{[t] : t \in \bigcap_{S\in\mathcal{P}(\mathcal{V}_0)} |A\{X := \dot{S}\}|\} \\ = \bigcap_{S\in\mathcal{P}(\mathcal{V}_0)} \{[t] : t \in |A\{X := \dot{S}\}]|\} &= \bigcap_{S\in\mathcal{P}(\mathcal{V}_0)} \|\neg A\{X := \dot{S}\}\|_V\end{aligned}$$

□

In terms of algebraic structure, the previous proposition advocates for the following equalities:

$$1.\ \mathop{\curlywedge}_{b\in B}(a \parr b) = a \parr (\mathop{\curlywedge}_{b\in B} b) \qquad 2.\ \mathop{\curlywedge}_{b\in B}(b \parr a) = (\mathop{\curlywedge}_{b\in B} b) \parr a \qquad 3.\ \neg \mathop{\curlywedge}_{a\in A} a = \mathop{\curlyvee}_{a\in A} \neg a$$

(recall that the order is defined as the reversed inclusion of primitive falsity values (whence $\cap$ is $\curlyvee$) and that the $\forall$ quantifier is interpreted by $\curlywedge$.)

Definition$^\bullet$ 11.13 (Disjunctive structure). A *disjunctive structure* is a complete meet-semilattice $(\mathcal{A}, \preccurlyeq)$ equipped with a binary operation $(a,b) \mapsto a \parr b$, called the *disjunction of* $\mathcal{A}$ together with a unary operation $a \mapsto \neg a$ called the *negation of* $\mathcal{A}$, which fulfill the following axioms:

1. Negation is anti-monotonic in the sense that for all $a, a' \in \mathcal{A}$:

   (Contravariance) $\qquad$ if $a \preccurlyeq a'$ then $\neg a' \preccurlyeq \neg a$

2. Disjunction is monotonic in the sense that for all $a, a', b, b' \in \mathcal{A}$:

   (Variance) $\qquad$ if $a \preccurlyeq a'$ and $b \preccurlyeq b'$ then $a \parr b \preccurlyeq a' \parr b'$

3. Arbitrary meets distributes over both operands of disjunction, in the sense that for all $a \in \mathcal{A}$ and for all subsets $B \subseteq \mathcal{A}$:

   (Distributivity) $\qquad \mathop{\curlywedge}_{b\in B}(a \parr b) = a \parr (\mathop{\curlywedge}_{b\in B} b) \qquad \mathop{\curlywedge}_{b\in B}(b \parr a) = (\mathop{\curlywedge}_{b\in B} b) \parr a$

4. Negation of the meet of set is equal to the join of the set of negated elements, in the sense that for all subsets $A \subseteq \mathcal{A}$:

   (Commutation) $\qquad \neg \mathop{\curlywedge}_{a\in A} a = \mathop{\curlyvee}_{a\in A} \neg a$

⌟

As in the case of implicative structures, the commutation laws imply the value of the internal laws when applied to the maximal element $\top$:

Proposition 11.14. *If $(\mathcal{A}, \preccurlyeq, \parr, \neg)$ is a disjunctive structure, then the following hold for all $a \in \mathcal{A}$:*

1.$^\bullet$ $\top \parr a = \top$ $\qquad$ *2.$^\bullet$* $a \parr \top = \top$ $\qquad$ *3.$^\bullet$* $\neg\top = \bot$

Proof. Using Proposition 9.4 and the axioms of disjunctive structures, we prove:

1. for all $a \in \mathcal{A}$, $\top \parr a = (\curlywedge \emptyset) \parr a = \curlywedge_{x,a\in\mathcal{A}}\{x \parr a : x \in \emptyset\} = \curlywedge \emptyset = \top$
2. for all $a \in \mathcal{A}$, $a \parr \top = a \parr (\curlywedge \emptyset) = \curlywedge_{x,a\in\mathcal{A}}\{a \parr x : x \in \emptyset\} = \curlywedge \emptyset = \top$
3. $\neg\top = \neg(\curlywedge \emptyset) = \curlyvee_{x\in\mathcal{A}}\{\neg x : x \in \emptyset\} = \curlyvee \emptyset = \bot$

□

11.2.2 Examples of disjunctive structures

11.2.2.1 Dummy structure

Example$^\bullet$ 11.15 (Dummy disjunctive structure)**.** Given a complete lattice $(\mathcal{L}, \preccurlyeq)$, the following definitions give rise to a dummy structure that fulfills the axioms of Definition 11.13:

$$a \parr b \triangleq \top \qquad\qquad \neg a \triangleq \bot \qquad\qquad (\forall a, b \in \mathcal{A})$$

The verification of the different axioms is straightforward. ⌟

11.2.2.2 Complete Boolean algebra

Example$^\bullet$ 11.16 (Complete Boolean algebras)**.** Let $\mathcal{B}$ be a complete Boolean algebra. It encompasses a disjunctive structure, that is defined by:

- $\mathcal{A} \triangleq \mathcal{B}$
- $a \preccurlyeq b \triangleq a \preccurlyeq b$
- $a \parr b \triangleq a \vee b$
- $\neg a \triangleq \neg a$

$(\forall a, b \in \mathcal{A})$

The different axioms are direct consequences of Proposition 9.7. ⌟

11.2.3 Disjunctive structure of classical realizability

If we abstract the structure of the realizability interpretation of $L^{\parr}$ (see Section 11.1.3), it is a structure of the form $(\mathcal{T}_0, \mathcal{E}_0, \mathcal{V}_0, (\cdot,\cdot), [\cdot], \perp\!\!\!\perp)$ where $\mathcal{V}_0 \subseteq \mathcal{E}_0$ is the distinguished subset of (positive) values, $(\cdot,\cdot)$ is a binary map from $\mathcal{E}_0^2$ to $\mathcal{E}_0$ (whose restriction to $\mathcal{V}_0$ has values in $\mathcal{V}_0$), $[\cdot]$ is an operation from $\mathcal{T}_0$ to $\mathcal{V}_0$, and $\perp\!\!\!\perp \subseteq \mathcal{T}_0 \times \mathcal{E}_0$ is a relation[4]. From this sextuple, we can define:

- $\mathcal{A} \triangleq \mathcal{P}(\mathcal{V}_0)$
- $a \preccurlyeq b \triangleq a \supseteq b$
- $a \parr b \triangleq (a, b) = \{(V_1, V_2) : V_1 \in a \wedge V_2 \in b\}$
- $\neg a \triangleq [a^{\perp\!\!\!\perp}] = \{[t] : t \in a^{\perp\!\!\!\perp}\}$

$(\forall a, b \in \mathcal{A})$

Proposition 11.17. *The quadruple* $(\mathcal{A}, \preccurlyeq, \parr, \neg)$ *is a disjunctive structure.*

Proof. We show that the axioms of Definition 11.13 are satisfied.

1. (Contravariance) Let $a, a' \in \mathcal{A}$, such that $a \preccurlyeq a'$ ie $a' \subseteq a$. Then $a^{\perp\!\!\!\perp} \subseteq a'^{\perp\!\!\!\perp}$ and thus
$$\neg a = \{[t] : t \in a^{\perp\!\!\!\perp}\} \subseteq \{[t] : t \in a'^{\perp\!\!\!\perp}\} = \neg a'$$
i.e. $\neg a' \preccurlyeq \neg a$.
2. (Covariance) Let $a, a', b, b' \in \mathcal{A}$ such that $a' \subseteq a$ and $b' \subseteq b$. Then we have
$$a \parr b = \{(V_1, V_2) : V_1 \in a \wedge V_2 \in b\} \subseteq \{(V_1, V_2) : V_1 \in a' \wedge V_2 \in b'\} = a' \parr b'$$
i.e. $a \parr b \preccurlyeq a' \parr b'$.
3. (Distributivity) Let $a \in \mathcal{A}$ and $B \subseteq \mathcal{A}$, we have:
$$\bigcurlywedge_{b \in B} (a \parr b) = \bigcurlywedge_{b \in B} \{(V_1, V_2) : V_1 \in a \wedge e_2 \in b\} = \{(V_1, V_2) : V_1 \in a \wedge V_2 \in \bigcurlywedge_{b \in B} b\} = a \parr (\bigcurlywedge_{b \in B} b)$$
4. (Commutation) Let $B \subseteq \mathcal{A}$, we have (recall that $\bigcurlyvee_{b \in B} b = \bigcap_{b \in B} b$):
$$\bigcurlyvee_{b \in B} (\neg b) = \bigcurlyvee_{b \in B} \{[t] : t \in b^{\perp\!\!\!\perp}\} = \{[t] : t \in \bigcurlyvee_{b \in B} b^{\perp\!\!\!\perp}\} = \{[t] : t \in (\bigcurlywedge_{b \in B} b)^{\perp\!\!\!\perp}\} = \neg(\bigcurlywedge_{b \in B} b)$$

□

Remark 11.18. The same definitions taking $\mathcal{A} \triangleq \mathcal{P}(\mathcal{E}_0)$ instead of $\mathcal{P}(\mathcal{V}_0)$ also satisfy the same properties. ⌟

[4] We could also abstract the different properties axiomatizing the pole and the different sets to obtain some kind of "abstract $L^{\parr}$ structure", but there is no point in doing this, since it would be less general than the notion of disjunctive structure anyway.

11.2.4 Interpreting $\mathrm{L}^{\parr}$

Following the interpretation of the λ-calculus in implicative structures, we shall now see how $\mathrm{L}^{\parr}$ commands can be recovered from disjunctive structures. From now on, we assume given a disjunctive structure $(\mathcal{A}, \preccurlyeq, \parr, \neg)$.

11.2.4.1 Commands

We shall begin with the interpretation of commands. This poses a novel difficulty with respect to the definition of λ-terms in implicative structures. Indeed, we are looking for an interpretation of terms and contexts, that is to say for both the realizers and the opponents (while in implicative structures we only interpreted realizers). Therefore, we first need to understand what it means for a command (in terms of the disjunctive structure) to be well-formed, *i.e.* to be in the pole. For this, we follow the intuition of the passage from a $^{\mathcal{K}}$OCA to an AKS (see Proposition 9.34). This translation indeed defines the embedding of a one-sided structure (the $^{\mathcal{K}}$OCA, with a set $\mathcal{A}$ of combinators) to a two-sided structure (the AKS, with a set $\mathbf{\Lambda}$ of realizers and a set $\mathbf{\Pi}$ of opponents). The induced AKS is indeed defined with the same domain for terms and stacks $\mathbf{\Lambda} = \mathbf{\Pi} = \mathcal{A}$. In this setting, the pole $\perp\!\!\!\perp$ is simply defined as the order relation on the $^{\mathcal{K}}$OCA: a term $t \in \mathbf{\Lambda}$ is orthogonal to a stack $\pi \in \mathbf{\Pi}$ if $t \preccurlyeq \pi$. This definition is in accordance with the intuition that the order reflect the quantity of information that a term (resp. stack, formula, etc...) carries: if the term t can defeat its opponent π, *i.e.* if $t \star \pi \in \perp\!\!\!\perp$, it means indeed that t is more defined than π.

We thus define the *commands* of the disjunctive structure $\mathcal{A}$ as the pair (a, b) (which we continue to write $\langle a \| b \rangle$) with $a, b \in \mathcal{A}$, and we define the pole $\perp\!\!\!\perp$ as the ordering relation $\preccurlyeq$. We write $C_{\mathcal{A}} = \mathcal{A} \times \mathcal{A}$ for the set of commands in $\mathcal{A}$ and $(a, b) \in \perp\!\!\!\perp$ for $a \preccurlyeq b$. Besides, we define an ordering on commands which extends the intuition that the order reflect the "definedness" of objects: given two commands c, c' in $C_{\mathcal{A}}$, we say that c is lower than c' and we write $c \trianglelefteq c'$ if $c \in \perp\!\!\!\perp$ implies that $c' \in \perp\!\!\!\perp$. It is straightforward to check that:

Proposition• 11.19. *The relation $\trianglelefteq$ is a preorder.*

Besides, the relation $\trianglelefteq$ verifies the following property of variance with respect to the order $\preccurlyeq$:

Proposition• 11.20 (Commands ordering)**.** *For all $t, t', \pi, \pi' \in \mathcal{A}$, if $t \preccurlyeq t'$ and $\pi' \preccurlyeq \pi$, then $\langle t \| \pi \rangle \trianglelefteq \langle t' \| \pi' \rangle$.*

Proof. Trivial by transitivity of $\preccurlyeq$. □

Finally, it is worth noting that meets are covariant with respect to $\trianglelefteq$ and $\preccurlyeq$, while joins are contravariant:

Lemma• 11.21. *If c and c' are two functions associating to each $a \in \mathcal{A}$ the commands $c(a)$ and $c'(a)$ such that $c(a) \trianglelefteq c'(a)$, then we have:*

$$\bigcurlywedge_{a \in \mathcal{A}} \{a : c(a) \in \perp\!\!\!\perp\} \preccurlyeq \bigcurlywedge_{a \in \mathcal{A}} \{a : c'(a) \in \perp\!\!\!\perp\} \qquad\qquad \bigcurlyvee_{a \in \mathcal{A}} \{a : c'(a) \in \perp\!\!\!\perp\} \preccurlyeq \bigcurlyvee_{a \in \mathcal{A}} \{a : c(a) \in \perp\!\!\!\perp\}$$

Proof. Assume c, c' are such that for all $a \in \mathcal{A}$, $ca \trianglelefteq c'a$. Then it is clear that by definition we have the inclusion $\{a \in \mathcal{A} : c(a) \in \perp\!\!\!\perp\} \subseteq \{a \in \mathcal{A} : c'(a) \in \perp\!\!\!\perp\}$, whence the expected results. □

11.2.4.2 Contexts

We are now ready to define the interpretation of $\mathrm{L}^{\parr}$ contexts in the disjunctive structure $\mathcal{A}$. The interpretation for the contexts corresponding to the connectives is very natural:

Definition$^\bullet$ **11.22** (Pairing)**.** For all $a,b \in \mathcal{A}$, we let $(a,b) \triangleq a ⅋ b$. ⌟

Definition$^\bullet$ **11.23** (Boxing)**.** For all $a \in \mathcal{A}$, we let $[a] \triangleq \neg a$. ⌟

Note that with these definitions, the encodings of pairs and boxes directly inherit of the properties of the internal law ⅋ and $\neg$ in disjunctive structures. As for the binder $\mu x.c$, which we write $\tilde{\mu}^+ c$, it should be defined in such a way that if c is a function mapping each $a \in \mathcal{A}$ to a command $c(a) \in C_{\mathcal{A}}$, then $\mu^+.c$ should be "compatible" with any a such that $c(a)$ is well-formed (*i.e.* $c(a) \in \perp\!\!\!\perp$). As it belongs to the side of opponents, the "compatibility" means that it should be greater than any such a, and we thus define it as a join.

Definition$^\bullet$ **11.24** (μ^+)**.** For all $c : \mathcal{A} \to C_{\mathcal{A}}$, we define:

$$\mu^+.c := \bigcurlyvee_{a \in \mathcal{A}} \{a : c(a) \in \perp\!\!\!\perp\}$$

⌟

These definitions enjoy the following properties with respect to the β-reduction and the η-expansion (compare with Proposition 10.17):

Proposition 11.25 (Properties of μ^+)**.** *For all functions $c, c' : \mathcal{A} \to C_{\mathcal{A}}$, the following hold:*

1.$^\bullet$ *If for all $a \in \mathcal{A}$, $c(a) \trianglelefteq c'(a)$, then $\mu^+.c' \preccurlyeq \mu^+.c$* *(Variance)*

2.$^\bullet$ *For all $t \in \mathcal{A}$, then $\langle t \| \mu^+.c\rangle \trianglelefteq c(t)$* *($\beta$-reduction)*

3.$^\bullet$ *For all $e \in \mathcal{A}$, then $t = \mu^+.(a \mapsto \langle a \| e\rangle)$* *($\eta$-expansion)*

Proof. 1. Direct consequence of Proposition 11.21.

2,3. Trivial by definition of μ^+.

□

Remark 11.26 (Subject reduction)**.** The β-reduction $c \to_\beta c'$ is reflected by the ordering relation $c \trianglelefteq c'$, which reads *"if c is well-formed, then so is c'"*. In other words, this corresponds to the usual property of subject reduction. In the sequel, we will see that β-reduction rules of L$^{⅋}$ will always been reflected in this way through the embedding in disjunctive structures. ⌟

11.2.4.3 Terms

Dually to the definitions of (positive) contexts μ^+ as a join, we define the embedding of (negative) terms, which are all binders, by arbitrary meets:

Definition$^\bullet$ **11.27** (μ^-)**.** For all $c : \mathcal{A} \to C_{\mathcal{A}}$, we define:

$$\mu^-.c := \bigcurlywedge_{a \in \mathcal{A}} \{a : c(a) \in \perp\!\!\!\perp\}$$

⌟

Definition$^\bullet$ **11.28** ($\mu^{()}c$)**.** For all $c : \mathcal{A}^2 \to C_{\mathcal{A}}$, we define:

$$\mu^{()}.c := \bigcurlywedge_{a,b \in \mathcal{A}} \{a ⅋ b : c(a,b) \in \perp\!\!\!\perp\}$$

⌟

Definition$^\bullet$ **11.29** ($\mu^{[]}$)**.** For all $c : \mathcal{A} \to C_{\mathcal{A}}$, we define:

$$\mu^{[]}.c := \bigcurlywedge_{a \in \mathcal{A}} \{\neg a : c(a) \in \perp\!\!\!\perp\}$$

⌟

These definitions also satisfy some variance properties with respect to the preorder $\trianglelefteq$ and the order relation $\preccurlyeq$, namely, negative binders for variable ranging over positive contexts are covariant, while negative binders intended to catch negative terms are contravariant.

Proposition 11.30 (Variance). *For any functions c, c' with the corresponding arities, the following hold:*

1.$^{\bullet}$ *If $c(a) \trianglelefteq c'(a)$ for all $a \in \mathcal{A}$, then $\mu^-.c \preccurlyeq \mu^-.c'$*

2.$^{\bullet}$ *If $c(a,b) \trianglelefteq c'(a,b)$ for all $a,b \in \mathcal{A}$, then $\mu^{()}.c \preccurlyeq \mu^{()}.c'$*

3.$^{\bullet}$ *If $c(a) \trianglelefteq c'(a)$ for all $a \in \mathcal{A}$, then $\mu^{[]}.c' \preccurlyeq \mu^{[]}.c$*

Proof. Direct consequences of Proposition 11.21. □

The η-expansion is also reflected as usual by the ordering relation $\preccurlyeq$:

Proposition 11.31 (η-expansion). *For all $t \in \mathcal{A}$, the following holds:*

1.$^{\bullet}$ $t = \mu^-.(a \mapsto \langle t \| a \rangle)$

2.$^{\bullet}$ $t \preccurlyeq \mu^{()}.(a,b \mapsto \langle t \| (a,b) \rangle)$

3.$^{\bullet}$ $t \preccurlyeq \mu^{[]}.(a \mapsto \langle t \| [a] \rangle)$

Proof. Trivial from the definitions. □

The β-reduction is reflected by the preorder $\trianglelefteq$:

Proposition 11.32 (β-reduction). *For all $e, e_1, e_2, t \in \mathcal{A}$, the following holds:*

1.$^{\bullet}$ $\langle \mu^-.c \| e \rangle \trianglelefteq c(e)$

2.$^{\bullet}$ $\langle \mu^{()}.c \| (e_1, e_2) \rangle \trianglelefteq c(e_1, e_2)$

3.$^{\bullet}$ $\langle \mu^{[]}.c \| [t] \rangle \trianglelefteq c(t)$

Proof. Trivial from the definitions. □

Finally, we call a *$L^{\parr}$ term with parameters in $\mathcal{A}$* (resp. context, command) any $\mathrm{L}^{\parr}$ term (possibly) enriched with constants taken in the set $\mathcal{A}$. Commands with parameters are equipped with the same rules of reduction as in $\mathrm{L}^{\parr}$, considering parameters as inert constants. To every closed $\mathrm{L}^{\parr}$ term t (resp. context e,command c) we associate an element $t^{\mathcal{A}}$ (resp. $e^{\mathcal{A}}$, $c^{\mathcal{A}}$) of $\mathcal{A}$, defined by induction on the structure of t as follows:

Contexts :

$$\begin{array}{rcl} a^{\mathcal{A}} & \triangleq & a \\ (e_1, e_2)^{\mathcal{A}} & \triangleq & (e_1^{\mathcal{A}}, e_2^{\mathcal{A}}) \\ [t]^{\mathcal{A}} & \triangleq & [t^{\mathcal{A}}] \\ (\mu x.c)^{\mathcal{A}} & \triangleq & \mu^-(a \mapsto (c[x := a])^{\mathcal{A}}) \end{array}$$

Terms :

$$\begin{array}{rcl} a^{\mathcal{A}} & \triangleq & a \\ (\mu\alpha.c)^{\mathcal{A}} & \triangleq & \mu^-(a \mapsto (c[\alpha := a])^{\mathcal{A}}) \\ (\mu(\alpha_1,\alpha_2).c)^{\mathcal{A}} & \triangleq & \mu^{()}(a,b \mapsto (c[\alpha_1 := a, \alpha_2 := b])^{\mathcal{A}}) \\ (\mu[x].c)^{\mathcal{A}} & \triangleq & \mu^{[]}(a \mapsto (c[x := a])^{\mathcal{A}}) \end{array}$$

Commands: $\langle t \| e \rangle^{\mathcal{A}} \triangleq \langle t^{\mathcal{A}} \| e^{\mathcal{A}} \rangle)$

In particular, this definition has the nice property of making the pole $\perp\!\!\!\perp$ (*i.e.* the order relation $\preccurlyeq$) closed under anti-reduction, as reflected by the following property of $\trianglelefteq$:

Proposition 11.33 (Subject reduction). *For any closed commands c_1, c_2 of $L^{\parr}$, if $c_1 \rightarrow_\beta c_2$ then $c_1^{\mathcal{A}} \trianglelefteq c_2^{\mathcal{A}}$,* i.e. *if $c_1^{\mathcal{A}}$ belongs to $\perp\!\!\!\perp$ then so does $c_2^{\mathcal{A}}$.*

Proof. Direct consequence of Propositions 11.25 and 12.22. □

11.2.5 Adequacy

We shall now prove that the interpretation of $\mathrm{L}^{\parr}$ is adequate with respect to its type system. Again, we extend the syntax of formulas to define second-order formulas with parameters by:

$$A,B ::= a \mid X \mid \neg A \mid A \parr B \mid \forall X.A \qquad (a \in \mathscr{A})$$

This allows us to embed closed formulas with parameters into the disjunctive structure $\mathscr{A}$. The embedding is trivially defined by:

$$\begin{array}{rcll} a^{\mathscr{A}} & \triangleq & a & \text{(if } a \in \mathscr{A}) \\ (\neg A)^{\mathscr{A}} & \triangleq & \neg A^{\mathscr{A}} & \\ (A \parr B)^{\mathscr{A}} & \triangleq & A^{\mathscr{A}} \parr B^{\mathscr{A}} & \\ (\forall X.A)^{\mathscr{A}} & \triangleq & \curlywedge_{a\in\mathscr{A}} (A\{X := a\})^{\mathscr{A}} & \end{array}$$

As for the adequacy of the interpretation for the second-order λ_c-calculus, we define substitutions, which we write σ, as functions mapping variables (of terms, contexts and types) to element of $\mathscr{A}$:

$$\sigma ::= \varepsilon \mid \sigma[x \mapsto a] \mid \sigma[\alpha \mapsto a] \mid \sigma[X \mapsto a] \qquad (a \in \mathscr{A},\ x,X \text{ variables})$$

In the spirit of the proof of adequacy in classical realizability, we say that a substitution σ realizes a typing context Γ, which write $\sigma \Vdash \Gamma$, if for all bindings $(x : A) \in \Gamma$ we have $\sigma(x) \preccurlyeq (A[\sigma])^{\mathscr{A}}$. Dually, we say that σ realizes Δ if for all bindings $(\alpha : A) \in \Delta$, we have $\sigma(\alpha) \succcurlyeq (A[\sigma])^{\mathscr{A}}$. We can now prove

Theorem 11.34 (Adequacy)**.** *The typing rules of $L^{\parr}$ (Figure 11.1) are adequate with respect to the interpretation of terms (contexts,commands) and formulas. Indeed, for all contexts Γ,Δ, for all formulas with parameters A then for all substitutions σ such that $\sigma \Vdash \Gamma$ and $\sigma \Vdash \Delta$, we have:*

1. *for any term t, if $\Gamma \vdash t : A \mid \Delta$, then $(t[\sigma])^{\mathscr{A}} \preccurlyeq A[\sigma]^{\mathscr{A}}$;*
2. *for any context e, if $\Gamma \mid e : A \vdash \Delta$, then $(e[\sigma])^{\mathscr{A}} \succcurlyeq A[\sigma]^{\mathscr{A}}$;*
3. *for any command c, if $c : (\Gamma \vdash \Delta)$, then $(c[\sigma])^{\mathscr{A}} \in \perp\!\!\!\perp$.*

Proof. By induction over the typing derivations.

- **Case** (Cut). Assume that we have:

$$\dfrac{\Gamma \vdash t : A \mid \Delta \quad \Gamma \mid e : A \vdash \Delta}{\langle t \| e \rangle : \Gamma \vdash \Delta}\ (\textsc{Cut})$$

By induction hypotheses, we have $(t[\sigma])^{\mathscr{A}} \preccurlyeq A[\sigma]^{\mathscr{A}}$ and $(e[\sigma])^{\mathscr{A}} \succcurlyeq A[\sigma]^{\mathscr{A}}$. By transitivity of the relation $\preccurlyeq$, we deduce that $(t[\sigma])^{\mathscr{A}} \preccurlyeq (e[\sigma])^{\mathscr{A}}$, so that $((\langle t \| e \rangle[\sigma])^{\mathscr{A}} \in \perp\!\!\!\perp$.

- **Case** ($\vdash ax$). Straightforward, since if $(x : A) \in \Gamma$, then $(x[\sigma])^{\mathscr{A}} \preccurlyeq (A[\sigma])^{\mathscr{A}}$. The case ($ax \vdash$) is identical.

- **Case** ($\vdash \mu$). Assume that we have:

$$\dfrac{c : \Gamma \vdash \Delta, \alpha : A}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\ (\vdash\mu)$$

By induction hypothesis, we have that $(c[\sigma,\alpha \mapsto (A[\sigma])^{\mathscr{A}}])^{\mathscr{A}} \in \perp\!\!\!\perp$. Then, by definition we have:

$$((\mu\alpha.c)[\sigma])^{\mathscr{A}} = (\mu\alpha.(c[\sigma]))^{\mathscr{A}} = \bigcurlywedge_{b\in\mathscr{A}} \{b : (c[\sigma,\alpha \mapsto b])^{\mathscr{A}} \in \perp\!\!\!\perp\} \preccurlyeq (A[\sigma])^{\mathscr{A}}$$

- **Case** ($\mu\ \vdash$). Similarly, assume that we have:

$$\dfrac{c : \Gamma, x : A \vdash \Delta}{\Gamma \mid \mu x.c : A \vdash \Delta}\ {\scriptstyle(\mu\vdash)}$$

By induction hypothesis, we have that $(c[\sigma, x \mapsto (A[\sigma])^{\mathcal{A}}])^{\mathcal{A}} \in \perp\!\!\!\perp$. Therefore, we have:

$$((\mu x.c)[\sigma])^{\mathcal{A}} = (\mu x.(c[\sigma]))^{\mathcal{A}} = \bigcurlyvee_{b\in\mathcal{A}} \{b : (c[\sigma, x \mapsto b])^{\mathcal{A}} \in \perp\!\!\!\perp\} \succcurlyeq (A[\sigma])^{\mathcal{A}}.$$

- **Case** ($⅋\ \vdash$). Assume that we have:

$$\dfrac{\Gamma \mid e_1 : A_1 \vdash \Delta \quad \Gamma \mid e_2 : A_2 \vdash \Delta}{\Gamma \mid (e_1, e_2) : A_1 ⅋ A_2 \vdash \Delta}\ {\scriptstyle(⅋\vdash)}$$

By induction hypotheses, we have that $(e_1[\sigma])^{\mathcal{A}} \succcurlyeq (A_1[\sigma])^{\mathcal{A}}$ and $(e_2[\sigma])^{\mathcal{A}} \succcurlyeq (A_2[\sigma])^{\mathcal{A}}$. Therefore, by monotonicity of the $⅋$ operator, we have:

$$((e_1, e_2)[\sigma])^{\mathcal{A}} = (e_1[\sigma], e_2[\sigma])^{\mathcal{A}} = (e_1[\sigma])^{\mathcal{A}} ⅋ (e_2[\sigma])^{\mathcal{A}} \succcurlyeq (A_1[\sigma])^{\mathcal{A}} ⅋ (A_2[\sigma])^{\mathcal{A}}.$$

- **Case** ($\vdash\ ⅋$). Assume that we have:

$$\dfrac{c : \Gamma \vdash \Delta, \alpha_1 : A_1, \alpha_2 : A_2}{\Gamma \vdash \mu(\alpha_1, \alpha_2).c : A_1 ⅋ A_2 \mid \Delta}\ {\scriptstyle(\vdash⅋)}$$

By induction hypothesis, we get that $(c[\sigma, \alpha_1 \mapsto (A_1[\sigma])^{\mathcal{A}}, \alpha_2 \mapsto (A_2[\sigma])^{\mathcal{A}}])^{\mathcal{A}} \in \perp\!\!\!\perp$. Then by definition we have

$$((\mu(\alpha_1, \alpha_2).c)[\sigma])^{\mathcal{A}} = \bigcurlywedge_{a,b\in\mathcal{A}} \{a ⅋ b : (c[\sigma, \alpha_1 \mapsto a, \alpha_2 \mapsto b])^{\mathcal{A}} \in \perp\!\!\!\perp\} \preccurlyeq (A_1[\sigma])^{\mathcal{A}} ⅋ (A_2[\sigma])^{\mathcal{A}}.$$

- **Case** ($\neg\ \vdash$). Assume that we have:

$$\dfrac{\Gamma \vdash t : A \mid \Delta}{\Gamma \mid [t] : \neg A \vdash \Delta}\ {\scriptstyle(\neg\vdash)}$$

By induction hypothesis, we have that $(t[\sigma])^{\mathcal{A}} \preccurlyeq (A[\sigma])^{\mathcal{A}}$. Then by definition of $[\,]^{\mathcal{A}}$ and covariance of the $\neg$ operator, we have:

$$([t[\sigma]])^{\mathcal{A}} = \neg(t[\sigma])^{\mathcal{A}} \succcurlyeq \neg(A[\sigma])^{\mathcal{A}}.$$

- **Case** ($\vdash\ \neg$). Assume that we have:

$$\dfrac{c : \Gamma, x : A \vdash \Delta}{\Gamma \vdash \mu[x].c : \neg A \mid \Delta}\ {\scriptstyle(\vdash\neg)}$$

By induction hypothesis, we have that $(c[\sigma, x \mapsto (A[\sigma])^{\mathcal{A}}])^{\mathcal{A}} \in \perp\!\!\!\perp$. Therefore, we have:

$$((\mu[x].c)[\sigma])^{\mathcal{A}} = (\mu[x].(c[\sigma]))^{\mathcal{A}} = \bigcurlywedge_{b\in\mathcal{A}} \{\neg b : (c[\sigma, x \mapsto b])^{\mathcal{A}} \in \perp\!\!\!\perp\} \preccurlyeq \neg(A[\sigma])^{\mathcal{A}}.$$

- **Case** ($\forall\ \vdash$) . Assume that we have:

$$\dfrac{\Gamma \vdash e : A\{X := B\} \mid \Delta}{\Gamma \mid e : \forall X.A \vdash \Delta}\ {\scriptstyle(\forall\vdash)}$$

By induction hypothesis, we have that $(e[\sigma])^{\mathcal{A}} \succcurlyeq ((A\{X := B\})[\sigma])^{\mathcal{A}} = (A[\sigma, X \mapsto (B[\sigma])^{\mathcal{A}}])^{\mathcal{A}}$. Therefore, we have that $(e[\sigma])^{\mathcal{A}} \succcurlyeq (A[\sigma, X \mapsto (B[\sigma])^{\mathcal{A}}])^{\mathcal{A}} \succcurlyeq \bigcurlywedge_{b\in\mathcal{A}}\{A\{X := b\}[\sigma]^{\mathcal{A}}\}$.

- **Case** ($\vdash \forall$) . Similarly, assume that we have:

$$\frac{\Gamma \vdash t : A \mid \Delta \quad X \notin FV(\Gamma,\Delta)}{\Gamma \vdash t : \forall X.A}\ {\scriptstyle(\vdash\forall)}$$

By induction hypothesis, we have that $(t[\sigma])^{\mathcal{A}} \preccurlyeq (A[\sigma,X \mapsto b])^{\mathcal{A}}$ for any $b \in A$. Therefore, we have that $(t[\sigma])^{\mathcal{A}} \preccurlyeq \curlywedge_{b\in\mathcal{A}}(A\{X := b\}[\sigma]^{\mathcal{A}})$. □

11.3 From disjunctive to implicative structures

11.3.1 The induced implicative structure

Recall that the implication is defined in terms of the disjunction and the negation by:

$$a \overset{\parr}{\rightarrow} b \triangleq \neg a \parr b$$

This definition can be reflected at the level of disjunctive structures in the sense that it directly induces an implicative structure:

Proposition• 11.35. *If $(\mathcal{A},\preccurlyeq,\parr,\neg)$ is a disjunctive structure, then $(\mathcal{A},\preccurlyeq,\overset{\parr}{\rightarrow})$ is an implicative structure.*

Proof. We need to show that the definition of the arrow fulfills the expected axioms:

1. (Variance) Let $a,b,a',b' \in \mathcal{A}$ be such that $a' \preccurlyeq a$ and $b \preccurlyeq b'$, then we have:

$$a \overset{\parr}{\rightarrow} b = \neg a \parr b \preccurlyeq \neg a' \parr b' = a' \overset{\parr}{\rightarrow} b'$$

since $\neg a \preccurlyeq \neg a'$ by contra-variance of the negation and $b \preccurlyeq b'$.

2. (Distributivity) Let $a \in \mathcal{A}$ and $B \subseteq \mathcal{A}$, then we have:

$$\curlywedge_{b\in B}(a \overset{\parr}{\rightarrow} b) = \curlywedge_{b\in B}(\neg a \parr b) = \neg a \parr (\curlywedge_{b\in B} b) = a \overset{\parr}{\rightarrow} (\curlywedge_{b\in B} b)$$

by distributivity of the infimum over the disjunction.

□

Therefore, we can again define for all a,b of $\mathcal{A}$ the application ab as well as the abstraction λf for any function f from $\mathcal{A}$ to $\mathcal{A}$;

$$ab \triangleq \curlywedge\{c \in \mathcal{A} : a \preccurlyeq b \overset{\parr}{\rightarrow} c\} \qquad\qquad \lambda f \triangleq \curlywedge_{a\in\mathcal{A}}(a \overset{\parr}{\rightarrow} fa)$$

We get for free the properties of these encodings in implicative structures:

Proposition 10.15 (Properties of abstraction and application). *The following properties hold for all $a,a',b,b',c \in \mathcal{A}$ and for all $f,g : \mathcal{A} \to \mathcal{A}$,*

1.• If $a \preccurlyeq a'$ and $b \preccurlyeq b'$, then $ab \preccurlyeq a'b'$. *(Monotonicity of application)*

2.• If $f(a) \preccurlyeq g(a)$ for all $a \in \mathcal{A}$, then $\lambda f \preccurlyeq \lambda g$. *(Monotonicity of abstraction)*

3.• $(\lambda f)a \preccurlyeq fa$. *($\beta$-reduction)*

4.• $a \preccurlyeq \lambda(x \mapsto ax)$. *($\eta$-expansion)*

5.• If $ab \preccurlyeq c$ then $a \preccurlyeq b \overset{\parr}{\rightarrow} c$. *(Adjunction)*

11.3.2 Interpretation of the λ-calculus

Up to this point, we defined two ways of interpreting a λ-term into a disjunctive structures, either through the implicative structure which is induced by the disjunctive one, or by embedding into the $L^{\parr}$-calculus which is then interpreted within the disjunctive structure. As a sanity check, we verify that both coincide.

Lemma• 11.36. *The shorthand $\mu([x],\alpha).c$ is interpreted in $\mathcal{A}$ by:*

$$(\mu([x],\alpha).c)^{\mathcal{A}} = \bigcurlywedge_{a,b\in\mathcal{A}} \{(\neg a) \parr b : c[x := a, \alpha := b] \in \preccurlyeq\}$$

Proof.

$$\begin{aligned}
\mu([x],\alpha).c)^{\mathcal{A}} &= (\mu(x_0,\alpha).\langle\mu[x].c \| x_0\rangle)^{\mathcal{A}} \\
&= \bigcurlywedge_{a',b\in\mathcal{A}} \{a' \parr b : (\langle\mu[x].c[\alpha := b] \| a'\rangle)^{\mathcal{A}} \in \preccurlyeq\} \\
&= \bigcurlywedge_{a',b\in\mathcal{A}} \{a' \parr b : (\bigcurlywedge_{a\in\mathcal{A}} \{\neg a : c^{\mathcal{A}}[x := a, \alpha := b] \in \preccurlyeq\} \preccurlyeq a'\} \\
&= \bigcurlywedge_{a,b\in\mathcal{A}} \{(\neg a) \parr b : c^{\mathcal{A}}[x := a, \alpha := b] \in \preccurlyeq\}
\end{aligned}$$

□

Proposition 11.37 (λ-calculus). *Let $\mathcal{A}^{\parr} = (\mathcal{A}, \preccurlyeq, \parr, \neg)$ be a disjunctive structure, and $\mathcal{A}^{\rightarrow} = (\mathcal{A}, \preccurlyeq, \xrightarrow{\parr})$ the implicative structure it canonically defines, we write ι for the corresponding inclusion. Let t be a closed λ-term (with parameter in $\mathcal{A}$), and $[\![t]\!]$ his embedding in $L^{\parr}$. Then we have*

$$\iota(t^{\mathcal{A}^{\rightarrow}}) = [\![t]\!]^{\mathcal{A}^{\parr}}$$

where $t^{\mathcal{A}^{\rightarrow}}$ (resp. $t^{\mathcal{A}^{\parr}}$) is the interpretation of t within $\mathcal{A}^{\rightarrow}$ (resp. $\mathcal{A}^{\parr}$).

In other words, this proposition expresses the fact that the following diagram commutes:

$$\begin{tikzcd}
\lambda\text{-calculus} \arrow[r, "{[\![\ ]\!]}"] \arrow[d, "{[\ ]^{\mathcal{A}^{\rightarrow}}}"'] & L^{\parr} \arrow[d, "{[\ ]^{\mathcal{A}^{\parr}}}"'] \\
(\mathcal{A}^{\rightarrow}, \preccurlyeq, \rightarrow) \arrow[r, "\iota"] & (\mathcal{A}^{\parr}, \preccurlyeq, \parr, \neg)
\end{tikzcd}$$

Proof. By induction over the structure of terms.

- **Case** a for some $a \in \mathcal{A}^{\parr}$. This case is trivial as both terms are equal to a.

- **Case** $\lambda x.u$. We have $[\![\lambda x.u]\!] = \mu([x],\alpha).\langle[\![t]\!] \| \alpha\rangle$ and

$$\begin{aligned}
(\mu([x],\alpha).\langle[\![t]\!] \| \alpha\rangle)^{\mathcal{A}^{\parr}} &= \bigcurlywedge_{a,b\in\mathcal{A}} \{\neg a \parr b : ([\![t[x := a]]\!]^{\mathcal{A}^{\parr}}, b) \in \perp\!\!\!\perp\} \\
&= \bigcurlywedge_{a,b\in\mathcal{A}} \{\neg a \parr b : [\![t[x := a]]\!]^{\mathcal{A}^{\parr}} \preccurlyeq b\} \\
&= \bigcurlywedge_{a\in\mathcal{A}} (\neg a \parr [\![t[x := a]]\!]^{\mathcal{A}^{\parr}})
\end{aligned}$$

On the other hand,

$$\iota([\lambda x.t]^{\mathcal{A}^{\to}}) = \iota(\bigcurlywedge_{a\in\mathcal{A}} (a \overset{\parr}{\to} (t[x := a])^{\mathcal{A}^{\to}})) = \bigcurlywedge_{a\in\mathcal{A}} (\neg a \parr \iota(t[x := a]^{\mathcal{A}^{\to}}))$$

Both terms are equal since $\llbracket t[x := a]\rrbracket^{\mathcal{A}^{\parr}} = \iota(t[x := a])^{\mathcal{A}^{\to}})$ by induction hypothesis.

- **Case** $u\,v$.

On the one hand, we have $\llbracket u\,v\rrbracket = \mu(\alpha).\langle\llbracket u\rrbracket\|([\llbracket v\rrbracket],\alpha)\rangle$ and

$$\begin{aligned}(\mu(\alpha).\langle\llbracket u\rrbracket\|([\llbracket v\rrbracket],\alpha)\rangle)^{\mathcal{A}^{\parr}} &= \bigcurlywedge_{a\in\mathcal{A}} \{a : (\llbracket u\rrbracket^{\mathcal{A}^{\parr}}, (\neg\llbracket v\rrbracket^{\mathcal{A}^{\parr}} \parr a)) \in \perp\!\!\!\perp\} \\ &= \bigcurlywedge_{a\in\mathcal{A}} \{a : \llbracket u\rrbracket^{\mathcal{A}^{\parr}} \preccurlyeq (\neg\llbracket v\rrbracket^{\mathcal{A}^{\parr}} \parr a)\}\end{aligned}$$

On the other hand,

$$\iota([u\,v]^{\mathcal{A}^{\to}}) = \iota(\bigcurlywedge_{a\in\mathcal{A}} \{a : (u^{\mathcal{A}^{\to}}) \preccurlyeq (v^{\mathcal{A}^{\to}}) \overset{\parr}{\to} a\}) = \bigcurlywedge_{a\in\mathcal{A}} \{a : \iota(u^{\mathcal{A}^{\to}}) \preccurlyeq \neg(\iota(v^{\mathcal{A}^{\to}}) \parr a\}))$$

Both terms are equal since $\llbracket u\rrbracket^{\mathcal{A}^{\parr}} = \iota(u^{\mathcal{A}^{\to}})$ and $\llbracket v\rrbracket^{\mathcal{A}^{\parr}} = \iota(v^{\mathcal{A}^{\to}})$ by induction hypotheses. □

11.4 Disjunctive algebras

11.4.1 Separation in disjunctive structures

We shall now introduce the notion of disjunctive separator. To this purpose, we adapt the definition of implicative separators, using Bourbaki's axioms for the disjunction and the negation instead of Hilbert's combinators **s** and **k**. We recall these axioms, which are taken from [21, p.25], to which we added the fifth one:

$$\begin{array}{lcl} S1 & : & (A \vee A) \to A \\ S2 & : & A \to (A \vee B) \\ S3 & : & (A \vee B) \to (B \vee A) \\ S4 & : & (A \to B) \to ((C \vee A) \to (C \vee B)) \\ S5 & : & (A \vee (B \vee C)) \to ((A \vee B) \vee C) \end{array}$$

Remark 11.38 (About S5)**.** The last axiom will mostly be used to swap the premises of an arrow from $A \to B \to C$ to $B \to A \to C$. In his book, Bourbaki does not need such an operation since he is interested in the provability of such an arrow, for which he can introduce A and B as hypotheses and try to prove C using these hypotheses in an arbitrary order. Therefore, the order of the premises is somehow irrelevant in his approach. On the opposite, we shall now contemplate the notion of separation (just like in the previous chapter). Typically, we will have to determine whether an element $a \to b$ belongs to a given separator, which is different from determining if b belongs to the separator knowing that a is in it. In this sense, we are facing a situation which is different from Bourbaki's setting.

Besides, viewed as a combinator, the fifth axiom is clearly independent from the others: it is the only one that allows us to decompose the operand of a disjunction as a disjunction itself (S1-S4 only consider premises/conclusions of the form A, $A \vee B$ or $(\neg A) \vee B$). Even though this informal argument could appear as not enough convincing, we believe that the question of knowing whether S5 is an axiom properly speaking is not of big interest here. If it is, then there is no point in considering the stronger notion of (non-associative) disjunctive algebra since all the realizability algebras are associative. If it is not, this simply means that there is a way to compile the corresponding combinator thanks to the first four, just like **i** can be retrieved by **skk** in implicative algebras. ⌟

Let $(\mathcal{A}, \preccurlyeq, \parr, \neg)$ be a fixed disjunctive structure. We thus define the combinators that canonically correspond to the previous axioms:

$$\begin{array}{rcl} s_1^{\parr} & \triangleq & \curlywedge_{a\in\mathcal{A}} [(a \parr a) \to a] \\ s_2^{\parr} & \triangleq & \curlywedge_{a,b\in\mathcal{A}} [a \to (a \parr b)] \\ s_3^{\parr} & \triangleq & \curlywedge_{a,b\in\mathcal{A}} [(a \parr b) \to b \parr a] \\ s_4^{\parr} & \triangleq & \curlywedge_{a,b,c\in\mathcal{A}} [(a \to b) \to (c \parr a) \to (c \parr b)] \\ s_5^{\parr} & \triangleq & \curlywedge_{a,b,c\in\mathcal{A}} [(a \parr (b \parr c)) \to ((a \parr b) \parr c)] \end{array}$$

Separators for $\mathcal{A}$ are defined similarly to the separators for implicative structures, replacing the combinators $\mathbf{K}, \mathbf{S}$ and $\mathbf{cc}$ by the previous ones.

Definition• 11.39 (Separator)**.** We call *separator* for the disjunctive structure $\mathcal{A}$ any subset $\mathcal{S} \subseteq \mathcal{A}$ that fulfills the following conditions for all $a, b \in \mathcal{A}$:

(1) If $a \in \mathcal{S}$ and $a \preccurlyeq b$ then $b \in \mathcal{S}$ (upward closure)

(2) s_1, s_2, s_3, s_4 and s_5 are in $\mathcal{S}$ (combinators)

(3) If $a \to b \in \mathcal{S}$ and $a \in \mathcal{S}$ then $b \in \mathcal{S}$ (closure under modus ponens)

A separator $\mathcal{S}$ is said to be *consistent* if $\bot \notin \mathcal{S}$. ⌟

Remark• 11.40 (Alternative definition)**.** As for implicative structures (Remark 10.29), in presence of condition (1), condition (3) is equivalent to the following condition:

(3') If $a \in \mathcal{S}$ and $b \in \mathcal{S}$ then $ab \in \mathcal{S}$ (closure under application)

The proof is exactly the same:

- (3) ⇒(3'): If $a \in \mathcal{S}$ and $b \in \mathcal{S}$, since $a \preccurlyeq b \to ab$ (Section 11.3.1) by upward closure we have $b \to ab \in \mathcal{S}$, and thus $ab \in \mathcal{S}$ by modus ponens .
- (3')⇒(3): If $a \in \mathcal{S}$ and $a \to b \in \mathcal{S}$, then $(a \to b)a \in \mathcal{S}$ by closure under application. Since $(a \to b)a \preccurlyeq b$ (Section 11.3.1) by upward closure we conclude that $b \in \mathcal{S}$.

⌟

Definition• 11.41 (Disjunctive algebra)**.** We call *disjunctive algebra* the given of a disjunctive structure $(\mathcal{A}, \preccurlyeq, \parr, \neg)$ together with a separator $\mathcal{S} \subseteq \mathcal{A}$. A disjunctive algebra is said to be consistent if its separator is. ⌟

Remark 11.42. The reader may notice that in this chapter, we do not distinguish between classical and intuitionistic separators. Indeed, $\mathrm{L}^{\parr}$ and the corresponding fragment of the sequent calculus are intrinsically classical. As we shall see thereafter, so are the disjunctive algebras: the negation is always involutive modulo the equivalence $\cong_{\mathcal{S}}$ (Proposition 11.58). ⌟

Example• 11.43 (Complete Boolean algebras)**.** Once again, if $\mathcal{B}$ is a complete Boolean algebra, $\mathcal{B}$ induces a disjunctive structure in which it is easy to verify that the combinators $s_1^{\parr}, s_3^{\parr}, s_3^{\parr}, s_4^{\parr}$ and $s_5^{\parr}$ are equal to the maximal element $\top$. Therefore, the singleton $\{\top\}$ is a valid separator for the induced disjunctive structure and any non-degenerated complete Boolean algebras thus induces a consistent disjunctive algebra. In fact, the filters for $\mathcal{B}$ are exactly its separators. ⌟

11.4.2 Disjunctive algebra from classical realizability

Recall that any model of classical realizability based on the $\mathrm{L}^{\parr}$-calculus induces a disjunctive structure, where:

- $\mathcal{A} \triangleq \mathcal{P}(\mathcal{V}_0)$
- $a \preccurlyeq b \triangleq a \supseteq b$
- $a \parr b \triangleq (a,b) = \{(e_1,e_2) : e_1 \in a \land e_2 \in b\}$
- $\neg a \triangleq [a^{\perp\!\!\perp}] = \{[v] : v \in a^{\perp\!\!\perp}\}$

$(\forall a,b \in \mathcal{A})$

As in the implicative case, the set of formulas realized by a closed term[5], that is to say:

$$\mathcal{S}_{\perp\!\!\perp} \triangleq \{a \in \mathcal{P}(V_0^+) : a^{\perp\!\!\perp} \cap \mathcal{T}_0 \neq \emptyset\}$$

defines a valid separator. The conditions (1) and (3) are clearly verified (for the same reasons as in the implicative case), but we should verify that the formulas corresponding to the combinators are indeed realized.

Let us then consider the following closed terms:

$$\begin{array}{lcl} PS_1 & \triangleq & \mu([x],\alpha).\langle x \| (\alpha,\alpha)\rangle \\ PS_2 & \triangleq & \mu([x],\alpha).\langle \mu(\alpha_1,\alpha_2).\langle x\|\alpha_1\rangle \| \alpha\rangle \\ PS_3 & \triangleq & \mu([x],\alpha).\langle \mu(\alpha_1,\alpha_2).\langle x\|(\alpha_2,\alpha_1)\rangle \| \alpha\rangle \\ PS_4 & \triangleq & \mu([x],\alpha).\langle \mu([y],\beta).\langle \mu(\gamma,\delta).\langle y\|(\gamma,\mu z.\langle x\|([z],\delta)\rangle)\rangle \|\beta\rangle\|\alpha\rangle \\ PS_5 & \triangleq & \mu([x],\alpha).\langle \mu(\beta,\alpha_3).\langle \mu(\alpha_1,\alpha_2).\langle x\|(\alpha_1,(\alpha_2,\alpha_3))\rangle\rangle\|\beta\rangle\|\alpha\rangle \end{array}$$

Proposition 11.44. *The previous terms have the following types in $\mathrm{L}^{\parr}$:*

1. $\vdash PS_1 : \forall A.(A \parr A) \to A \mid$
2. $\vdash PS_2 : \forall AB.A \to A \parr B \mid$
3. $\vdash PS_3 : \forall AB.A \parr B \to B \parr A \mid$
4. $\vdash PS_4 : \forall ABC.(A \to B) \to (C \parr A \to C \parr B) \mid$
5. $\vdash PS_5 : \forall ABC.(A \parr (B \parr C)) \to ((A \parr B) \parr C) \mid$

Proof. Straightforward typing derivations in $\mathrm{L}^{\parr}$. □

We deduce that $\mathcal{S}_{\perp\!\!\perp}$ is a valid separator:

Proposition 11.45. *The quintuple $(\mathcal{P}(\mathcal{V}_0),\preccurlyeq,\parr,\neg,\mathcal{S}_{\perp\!\!\perp})$ as defined above is a disjunctive algebra.*

Proof. Conditions (1) and (3) are trivial. Condition (2) follows from the previous proposition and the adequacy lemma for the realizability interpretation of $\mathrm{L}^{\parr}$ (Proposition 11.10). □

11.4.3 About the combinators

The interpretations of the terms PS_1, PS_2, PS_3 and PS_5 are equal to their principal types.

Proposition• 11.46. *We have:*

$$(PS_1)^{\mathcal{A}} = \bigcurlywedge_{a\in\mathcal{A}} ((a \parr a) \to a)$$

[5] As in the $\lambda\mu\tilde{\mu}$-calculus (see Section 4.4.5), proof-like terms in $\mathrm{L}^{\parr}$ simply correspond to closed terms.

Proof. By definition, we have:

$$(PS_1)^{\mathcal{A}} = (\mu([x],\alpha).\langle x \| (\alpha,\alpha)\rangle)^{\mathcal{A}} = \bigcurlywedge_{\alpha,x\in\mathcal{A}} \{x \to \alpha : x \preccurlyeq (\alpha \parr \alpha)\}$$

Let α, x be elements of $\mathcal{A}$ such that $x \preccurlyeq \alpha \parr \alpha$. Then by covariance of the arrow and definition of the meet, we deduce that:

$$\bigcurlywedge_{a\in\mathcal{A}} \{(a \parr a) \to a\} \preccurlyeq (\alpha \parr \alpha) \to \alpha \preccurlyeq x \to \alpha$$

and this being true for any $\alpha, x \in \mathcal{A}$, we obtain:

$$\bigcurlywedge_{a\in\mathcal{A}} \{(a \parr a) \to a\} \preccurlyeq \bigcurlywedge_{\alpha,x\in\mathcal{A}} \{x \to \alpha : x \preccurlyeq (\alpha \parr \alpha)\} = (PS_1)^{\mathcal{A}}$$

The converse inequality can be proved the same way, or can be directly deduced using Proposition 12.29 and the adequacy $\mathrm{L}^{\parr}$ typing rules (Proposition 11.34). □

Proposition• 11.47. *We have:*

$$(PS_2)^{\mathcal{A}} = \bigcurlywedge_{a,b\in\mathcal{A}} (a \to a \parr b)$$

Proof. By definition, we have:

$$(PS_2)^{\mathcal{A}} = (\mu([x],\alpha).\langle \mu(\alpha_1,\alpha_2).\langle x \| \alpha_1\rangle \| \alpha\rangle)^{\mathcal{A}} = \bigcurlywedge_{\alpha,x\in\mathcal{A}} \{x \to \alpha : \bigcurlywedge_{\alpha_1,\alpha_2\in\mathcal{A}} \{\alpha_1 \parr \alpha_2 : x \preccurlyeq \alpha_1\} \preccurlyeq \alpha\}$$

Using the distributivity of meets over the disjunction, one observe that for any fixed a:

$$\bigcurlywedge_{\alpha_1,\alpha_2\in\mathcal{A}} \{\alpha_1 \parr \alpha_2 : x \preccurlyeq \alpha_1\} = \Big(\bigcurlywedge_{\alpha_1\in\mathcal{A}} \{\alpha_1 : x \preccurlyeq \alpha_1\}\Big) \parr \Big(\bigcurlywedge_{\alpha_2\in\mathcal{A}} \{\alpha_2\}\Big) = x \parr \bot$$

Therefore, we can directly prove that:

$$(PS_2)^{\mathcal{A}} = \bigcurlywedge_{\alpha,x\in\mathcal{A}} \{x \to \alpha : x \parr \bot \preccurlyeq \alpha\} = \bigcurlywedge_{\alpha,x\in\mathcal{A}} \{x \to x \parr \bot\} = \bigcurlywedge_{a,b\in\mathcal{A}} \{a \to (a \parr b)\}$$

□

Proposition• 11.48. *We have:*

$$(PS_3)^{\mathcal{A}} = \bigcurlywedge_{a,b\in\mathcal{A}} (a \parr b \to b \parr a)$$

Proof. We want to prove the inequality from right to left, the other one being a consequence of semantic typing. By definition, we have:

$$(PS_3)^{\mathcal{A}} = (\mu([x],\alpha).\langle \mu(\alpha_1,\alpha_2).\langle x \| (\alpha_2,\alpha_1)\rangle \| \alpha\rangle)^{\mathcal{A}} = \bigcurlywedge_{\alpha,x\in\mathcal{A}} \Big\{x \to \alpha : \bigcurlywedge_{\alpha_1,\alpha_2\in\mathcal{A}} \{\alpha_1 \parr \alpha_2 : x \preccurlyeq \alpha_2 \parr \alpha_1\} \preccurlyeq \alpha\Big\}$$

Let α, x be elements of $\mathcal{A}$ such that $\curlywedge_{\alpha_1,\alpha_2\in\mathcal{A}}\{\alpha_1 \parr \alpha_2 : x \preccurlyeq \alpha_2 \parr \alpha_1\} \preccurlyeq \alpha$. Using the variance of the arrow we obtain that:

$$x \to \bigcurlywedge_{\alpha_1,\alpha_2\in\mathcal{A}} \{\alpha_1 \parr \alpha_2 : x \preccurlyeq \alpha_2 \parr \alpha_1\} \preccurlyeq x \to \alpha$$

Using the commutation of meet and par, we have:

$$x \to \bigcurlywedge_{\alpha_1,\alpha_2 \in \mathcal{A}} \{\alpha_1 \parr \alpha_2 : x \preccurlyeq \alpha_2 \parr \alpha_1\} = \bigcurlywedge_{\alpha_1,\alpha_2 \in \mathcal{A}} \{x \to \alpha_1 \parr \alpha_2 : x \preccurlyeq \alpha_2 \parr \alpha_1\}$$

Let then α_1, α_2 be elements of $\mathcal{A}$ such that $x \preccurlyeq \alpha_2 \parr \alpha_1$, using the variance of the arrow, we deduce that:

$$\bigcurlywedge_{a,b \in \mathcal{A}} (a \parr b \to b \parr a) \preccurlyeq \alpha_1 \parr \alpha_2 \to \alpha_2 \parr \alpha_1 \preccurlyeq x \to \alpha_2 \parr \alpha_1$$

Recollecting the pieces, we deduce (by introduction of the meet over α_1, α_2) that:

$$\bigcurlywedge_{a,b \in \mathcal{A}} (a \parr b \to b \parr a) \preccurlyeq x \to \bigcurlywedge_{\alpha_1,\alpha_2 \in \mathcal{A}} \{\alpha_1 \parr \alpha_2 : x \preccurlyeq \alpha_2 \parr \alpha_1\} \preccurlyeq x \to \alpha$$

and finally (by introduction of the meet over α, x) that:

$$\bigcurlywedge_{a,b \in \mathcal{A}} (a \parr b \to b \parr a) \preccurlyeq \bigcurlywedge_{\alpha,x \in \mathcal{A}} \left\{x \to \alpha : \bigcurlywedge_{\alpha_1,\alpha_2 \in \mathcal{A}} \{\alpha_1 \parr \alpha_2 : x \preccurlyeq \alpha_2 \parr \alpha_1\} \preccurlyeq \alpha\right\} = (PS_3)^{\mathcal{A}}$$

□

Proposition$^\bullet$ 11.49. *We have:*

$$(PS_5)^{\mathcal{A}} = \bigcurlywedge_{a,b,c \in \mathcal{A}} ((a \parr (b \parr c)) \to ((a \parr b) \parr c)$$

Proof. Once more, we only want to prove the inequality from right to left, the other one being a consequence of semantic typing. By definition, we have:

$$\begin{aligned}
(PS_5)^{\mathcal{A}} &= (\mu([x],\alpha).\langle \mu(\beta,\alpha_3).\langle \mu(\alpha_1,\alpha_2).\langle x \| (\alpha_1,(\alpha_2,\alpha_3))\rangle \| \beta\rangle \| \alpha\rangle)^{\mathcal{A}} \\
&= \textstyle\bigcurlywedge_{\alpha,x \in \mathcal{A}} \left\{x \to \alpha : \bigcurlywedge_{\beta,\alpha_3 \in \mathcal{A}} \left\{\beta \parr \alpha_3 : \bigcurlywedge_{\alpha_1,\alpha_2 \in \mathcal{A}} \{\alpha_1 \parr \alpha_2 : x \preccurlyeq \alpha_1 \parr (\alpha_2 \parr \alpha_3)\} \preccurlyeq \beta\right\} \preccurlyeq \alpha\right\} \\
&= \textstyle\bigcurlywedge_{x,\beta,\alpha_3 \in \mathcal{A}} \left\{x \to (\beta \parr \alpha_3) : \bigcurlywedge_{\alpha_1,\alpha_2 \in \mathcal{A}} \{\alpha_1 \parr \alpha_2 : x \preccurlyeq \alpha_1 \parr (\alpha_2 \parr \alpha_3)\} \preccurlyeq \beta\right\} \\
&= \textstyle\bigcurlywedge_{x,\alpha_3,\alpha_1,\alpha_2 \in \mathcal{A}} \{x \to (\alpha_1 \parr \alpha_2) \parr \alpha_3 : x \preccurlyeq \alpha_1 \parr (\alpha_2 \parr \alpha_3)\}
\end{aligned}$$

Let $x, \alpha_3, \alpha_1, \alpha_2$ be elements of $\mathcal{A}$ such that $x \preccurlyeq \alpha_1 \parr (\alpha_2 \parr \alpha_3)$. Using the covariance of the arrow on the left, and by definition of meets, we get that:

$$\bigcurlywedge_{a,b,c \in \mathcal{A}} ((a \parr (b \parr c)) \to ((a \parr b) \parr c) \preccurlyeq \alpha_1 \parr (\alpha_2 \parr \alpha_3) \to (\alpha_1 \parr \alpha_2) \parr \alpha_3 \preccurlyeq x \to (\alpha_1 \parr \alpha_2) \parr \alpha_3$$

Thus, we can conclude (by introduction of the meet over $x, \alpha_3, \alpha_2, \alpha_1$) that:

$$\bigcurlywedge_{a,b,c \in \mathcal{A}} ((a \parr (b \parr c)) \to ((a \parr b) \parr c) \preccurlyeq \bigcurlywedge_{x,\alpha_3,\alpha_1,\alpha_2 \in \mathcal{A}} \{x \to (\alpha_1 \parr \alpha_2) \parr \alpha_3 : x \preccurlyeq \alpha_1 \parr (\alpha_2 \parr \alpha_3)\} = (PS_5)^{\mathcal{A}}$$

□

Remark$^\bullet$ 11.50. Before turning to the study of the internal logic of disjunctive algebras, we should say a word on the missing equality for PS_4 and $s_4^{\parr}$. In contrast with the other four L$^{\parr}$ terms, PS_4 makes use of a context $\mu x.c$. Through the embedding, this binder is translated into a join and we get:

$$PS_4^{\mathcal{A}} = \bigcurlywedge_{x,\alpha \in \mathcal{A}} \{x \to \alpha : \bigcurlywedge_{y,\beta \in \mathcal{A}} \{y \to \beta : \bigcurlywedge_{\gamma,\delta \in \mathcal{A}} \{\gamma \parr \delta : y \preccurlyeq \gamma \parr \bigcurlyvee_{z \in \mathcal{A}} \{z : x \preccurlyeq z \to \delta\}\} \preccurlyeq \beta\} \preccurlyeq \alpha\}$$

By manipulation of the meets with their commutation, we can reduce it to:

$$PS_4^{\mathcal{A}} = \bigcurlywedge_{x,b,c\mathcal{A}} \{x \to (c \parr \bigcurlyvee_{z\in\mathcal{A}} \{z : x \preccurlyeq z \to b\}\} \to (c \parr b)\}$$

Nonetheless, this is *a priori* the best we can do, in the absence of commutation law for the join. In particular, there is no way to prove that $s_4^{\parr} = \curlywedge_{a,b,c\in\mathcal{A}}((a \to b) \to (c \parr a) \to (c \parr b))$ is lower than this term, given a fixed x, there is no way to find two elements a and b such that $x \preccurlyeq a \to b$. Of course, if the disjunctive algebra has extra commutations (of joins with the negation and the disjunction), the equality holds, but in this case the disjunctive algebra is in fact a Boolean algebra. ⌟

11.4.4 Internal logic

11.4.4.1 Entailment

As in the case of implicative algebras, we define a relation of entailment:

Definition$^\bullet$ 11.51 (Entailment). For all $a,b \in \mathcal{A}$, we say that a *entails* b and write $a \vdash_{\mathcal{S}} b$ if $a \to b \in \mathcal{S}$. We say that a and b are equivalent and write $a \cong_{\mathcal{S}} b$ if $a \vdash_{\mathcal{S}} b$ and $b \vdash_{\mathcal{S}} a$. ⌟

From the combinators, we directly get that:

Proposition 11.52 (Combinators). *For all $a,b,c \in \mathcal{A}$, the following holds:*

1.$^\bullet$ $(a \parr a) \vdash a$

2.$^\bullet$ $a \vdash (a \parr b)$

3.$^\bullet$ $(a \parr b) \vdash (b \parr a)$

4.$^\bullet$ $(a \to b) \vdash (c \parr a) \to (c \parr b)$

5.$^\bullet$ $a \parr (b \parr c) \vdash (a \parr b) \parr c$

Proposition 11.53 (Preorder). *For any $a,b,c \in \mathcal{A}$, the following holds:*

1.$^\bullet$ $a \vdash_{\mathcal{S}} a$ *(Reflexivity)*

2.$^\bullet$ if $a \vdash_{\mathcal{S}} b$ and $b \vdash_{\mathcal{S}} c$ then $a \vdash_{\mathcal{S}} c$ *(Transitivity)*

Proof. We first that (2) holds by applying twice the closure by modus ponens, then we use it with the relation $a \vdash_{\mathcal{S}} a \parr a$ and $a \parr a \vdash_{\mathcal{S}}$ proven above to get 1. □

We could pursue our investigation about the properties of the entailment relation as we did in implicative algebras. Unfortunately, in comparison with the implicative setting, we are lacking of a powerful proof tool. Indeed, remember that for implicative algebras, we were able to compute directly with truth values, mainly thanks to the fact that any separator contains all closed λ-terms. This statement was proven using the combinatorial completeness of the separators **K** and **S** with respect to the λ-calculus. Here, we are in a situation drastically different: first of all, we do not have any clue about a potential completeness of $PS_1, \dots, PS_5$ with respect to $\mathrm{L}^{\parr}$. And even if we were having such a result, since $PS_4^{\mathcal{A}}$ is not equal to $s_4^{\parr}$, we still could not use it to prove that every closed $\mathrm{L}^{\parr}$ term is in the separator.

In a nutshell, we are in a situation where we have to do realizability with only a finite set of realizers, and the possibility of examining the structure of falsity values case by case. In particular, most of the proof we present thereafter rely on technical lemmas requiring tedious and boring proofs. We shall skip some details, taking advantage of our formalization which should help the reader to convince himself that we are not hiding difficulties under the carpet. The key lemma in this situation is the closure of the separator under application (condition (3')). Indeed, it allows us to prove the following technical lemmas, which are generalized forms of modus ponens and transitivity, compatible with meets:

Lemma• 11.54 (Generalized modus ponens). *For all subsets $A, B \subseteq \mathcal{A}$, if $\bigcurlywedge_{a \in A, b \in B}(a \to b) \in \mathcal{S}$ and $(\bigcurlywedge_{a \in A} a) \in \mathcal{S}$, then $(\bigcurlywedge_{b:B} b) \in \mathcal{S}$.*

Proof. Let $A, B \subseteq A$ be two subsets of $\mathcal{A}$ such that $t_{ab} \triangleq \bigcurlywedge_{a \in A, b \in B}(a \to b) \in \mathcal{S}$ and $t_a \triangleq (\bigcurlywedge_{a \in A} a) \in \mathcal{S}$. Then by closure under application, we have $t_{ab}t_a \in \mathcal{S}$. Using the upward closure, it only remains to prove that:

$$t_{ab}t_a \preccurlyeq (\bigcurlywedge_{b:B} b)$$

which is an easy manipulation of meets using the adjunction. □

Lemma• 11.55 (Generalized transitivity). *For any subsets $A, B, C \subseteq \mathcal{A}$, if $\bigcurlywedge_{a \in A, b \in B}(a \to b) \in \mathcal{S}$ and $\bigcurlywedge_{b \in B, c \in C}(b \to c) \in \mathcal{S}$, then $\bigcurlywedge_{a \in A, c \in C}(a \to c) \in \mathcal{S}$.*

Proof. Let $A, B, C \subseteq \mathcal{A}$ be some fixed sets, such that $t_{ab} \triangleq \bigcurlywedge_{a \in A, b \in B}(a \to b) \in \mathcal{S}$ and $t_{bc} = \bigcurlywedge_{b \in B, c \in C}(b \to c) \in \mathcal{S}$. Then we have $s_4^{\parr}\, t_{ab}\, t_{bc} \in \mathcal{S}$, and it suffices to show that

$$s_4^{\parr}\, t_{bc}\, t_{ab} = \left(\bigcurlywedge_{a,b,c \in \mathcal{A}} (a \to b) \to (c \parr a) \to (c \parr b) \right) t_{bc}\, t_{ab} \preccurlyeq \bigcurlywedge_{a \in A, c \in C} (a \to c)$$

This is proved again by a straightforward manipulation of the meets using the adjunction. □

As a corollary, we can for instance use the previous lemma to show that:

Proposition• 11.56 (I). *We have $I^{\mathcal{A}} = \bigcurlywedge_{a \in A}(a \to a) \in \mathcal{S}$.*

Proof. Simple application of Lemma 11.55 to compose $s_2^{\parr}$ and $s_1^{\parr}$. □

11.4.4.2 Negation

We can relate the primitive negation to the one induced by the underlying implicative structure:

Proposition 11.57 (Implicative negation). *For all $a \in \mathcal{A}$, the following holds:*

1.• $\neg a \vdash_{\mathcal{S}} a \to \bot$ *2.•* $a \to \bot \vdash_{\mathcal{S}} \neg a$

Proof. We prove in both cases a slightly more general statement, namely that the meet over all a, b or the corresponding implication belongs to the separator. The first item follows directly from the fact that $s_2^{\parr}$ belongs to the separator, since $\bigcurlywedge_{a \in \mathcal{A}}(\neg a) \to (a \to \bot) = \bigcurlywedge_{a \in \mathcal{A}}(\neg a) \to (\neg a \parr \bot)$.

For the second item, the first step is to apply Lemma 11.55 with the following hypotheses:

$$\bigcurlywedge_{a \in \mathcal{A}} ((a \to \bot) \to a \to \neg a) \in \mathcal{S} \qquad \bigcurlywedge_{a \in \mathcal{A}} (a \to \neg a) \to \neg a \in \mathcal{S}$$

The statement on the left hand-side is proved by subtyping from the identity. On the right hand-side, we use twice Lemma 11.54 to prove that:

$$\bigcurlywedge_{a \in \mathcal{A}} (a \to a) \to (\neg a \to \neg a) \to (a \to \neg a) \to \neg a \ \in \mathcal{S}$$

The two extra hypotheses are trivially subtypes of the identity again. This statement follows from this more general property (recall that $a \to a = \neg a \parr a$):

$$\bigcurlywedge_{a,b \in \mathcal{A}} (a \parr b) \to a + b$$

that we shall prove thereafter (see Proposition 11.59). □

Additionally, we can show that the principle of double negation elimination is valid with respect to any separator:

Proposition 11.58 (Double negation)**.** *For all $a \in \mathcal{A}$, the following holds:*

1.$^\bullet$ $a \vdash_{\mathcal{S}} \neg\neg a$ *2.*$^\bullet$ $\neg\neg a \vdash_{\mathcal{S}} a$

Proof. The first item is easy since for all $a \in \mathcal{A}$, we have $a \to \neg\neg a = (\neg a) \parr \neg\neg a \cong_{\mathcal{S}} \neg\neg a \parr \neg a = \neg a \to \neg a$. As for the second item, we use Lemma 11.55 and Proposition 11.57 to it reduce to the statement:

$$\curlywedge_{a\in\mathcal{A}} ((\neg a) \to \bot) \to a \quad \in \mathcal{S}$$

We use again Lemma 11.55 to prove it, by showing that:

$$\curlywedge_{a\in\mathcal{A}} ((\neg a) \to \bot) \to (\neg a) \to a \quad \in \mathcal{S} \qquad\qquad \curlywedge_{a\in\mathcal{A}} ((\neg a) \to a) \to (\neg a) \to a \quad \in \mathcal{S}$$

where the statement on the left hand-side from by subtyping from the identity. For the one on the right hand-side, we use the same trick as in the last proof in order to reduce it to:

$$\curlywedge_{a\in\mathcal{A}} (a \to \neg a) \to (a \to a) \to (\neg a \to a) \to a) \ \in \mathcal{S}$$

□

11.4.4.3 Sum type

As in implicative structures, we can define the sum type by:

$$a + b \triangleq \curlywedge_{c\in\mathcal{A}} ((a \to c) \to (b \to c) \to c) \qquad\qquad (\forall a, b \in \mathcal{A})$$

We can prove that the disjunction and this sum type are equivalent from the point of view of the separator:

Proposition 11.59 (Implicative sum type)**.** *For all $a, b \in \mathcal{A}$, the following holds:*

1.$^\bullet$ $a \parr b \vdash_{\mathcal{S}} a + b$ *2.*$^\bullet$ $a + b \vdash_{\mathcal{S}} a \parr b$

Proof. We prove in both cases a slightly more general statement, namely that the meet over all a, b or the corresponding implication belongs to the separator. For the first item, we have:

$$\curlywedge_{a,b\in\mathcal{A}} (a \parr b) \to a + b = \curlywedge_{a,b,c\in\mathcal{A}} (a \parr b) \to (a \to c) \to (b \to c) \to c$$

Swapping the order of the arguments, we prove that $\curlywedge_{a,b,c\in\mathcal{A}}(b \to c) \to (a \parr b) \to (a \to c) \to c \in \mathcal{S}$. For this, we use Lemma 11.55 and the fact that:

$$\curlywedge_{a,b,c\in\mathcal{A}} (b \to c) \to (a \parr b) \to (a \parr c) \ \in \mathcal{S} \qquad\qquad \curlywedge_{a,c\in\mathcal{A}} (a \parr c) \to (a \to c) \to c \ \in \mathcal{S}$$

The left hand-side statement is proved using $s_4^{\parr}$, while on the right hand-side we prove it from the fact that:

$$\curlywedge_{a,c\in\mathcal{A}} (a \to c) \to (a \parr c) \to c \parr c \ \in \mathcal{S}$$

which is a subtype of $\mathbf{s}_4^{\parr}$, by using Lemma 11.55 again with $\mathbf{s}_1^{\parr}$ and by manipulation on the order of the argument.

The second item is easier to prove, using Lemma 11.55 again and the fact that:

$$\bigcurlywedge_{a,b\in\mathcal{A}} a + b \to (a \to (a \parr b)) \to (b \to (a \parr b)) \to (a \parr b) \in \mathcal{S}$$

which is a subtype of $\mathbf{I}^{\mathcal{A}}$ (which belongs to $\mathcal{S}$). The other part, which is to prove that:

$$\bigcurlywedge_{a,b\in\mathcal{A}} ((a \to (a \parr b)) \to (b \to (a \parr b)) \to (a \parr b)) \to (a \parr b) \quad \in \mathcal{S}$$

follows from Lemma 11.54 and the fact that $\bigcurlywedge_{a,b\in\mathcal{A}}(a \to (a \parr b))$ and $\bigcurlywedge_{a,b\in\mathcal{A}}(b \to (a \parr b))$ are both in the separator.

□

11.4.5 Induced implicative algebras

We shall now prove that the combinators defining implicative separators also belong to any disjunctive separator. Since conditions (1) and (3) of disjunctive and implicative separators are equal, this will in particular prove that any disjunctive algebra is a particular case of implicative algebra.

Proposition• 11.60 (Combinator $\mathbf{K}^{\mathcal{A}}$)**.** *For any disjunctive algebra $(\mathcal{A},\preccurlyeq,\parr,\neg,\mathcal{S})$, we have $\mathbf{K}^{\mathcal{A}} \in \mathcal{S}$.*

Proof. This directly follows by upwards closure from the fact that $\bigcurlywedge_{a,b\in\mathcal{A}} a \to (b \parr a) \in \mathcal{S}$. □

Proposition• 11.61 (Combinator $\mathbf{S}^{\mathcal{A}}$)**.** *For any disjunctive algebra $(\mathcal{A},\preccurlyeq,\parr,\neg,\mathcal{S})$, we have $\mathbf{S}^{\mathcal{A}} \in \mathcal{S}$.*

Proof. We make several applications of Lemmas 11.55 and 11.54 consecutively. We prove that:

$$\bigcurlywedge_{a,b,c\in\mathcal{A}} ((a \to b \to c) \to (a \to b) \to a \to c) \quad \in \mathcal{S}$$

is implied by Lemma 11.55 and:

$$\bigcurlywedge_{a,b,c\in\mathcal{A}} ((a \to b \to c) \to (b \to a \to c)) \in \mathcal{S} \qquad \text{and} \qquad \bigcurlywedge_{a,b,c\in\mathcal{A}} ((b \to a \to c) \to (a \to b) \to a \to c) \in \mathcal{S}$$

The statement on the left hand-side is an ad-hoc lemma, while the other is proved by generalized transitivity (Lemma 11.54), using a subtype of $\mathbf{s}_4^{\parr}$ as hypothesis, from:

$$\bigcurlywedge_{a,b,c\in\mathcal{A}} ((a \to b) \to (a \to a \to c)) \to (a \to b) \to a \to c \in \mathcal{S}$$

The latter is proved, using again generalized transitivity with a subtype of $\mathbf{s}_4^{\parr}$ as premise, from:

$$\bigcurlywedge_{a,b,c\in\mathcal{A}} (a \to a \to c) \to (a \to c) \in \mathcal{S}$$

This is proved using again Lemmas 11.55 and 11.54 with $\mathbf{s}_5^{\parr}$ and a variant of $\mathbf{s}_4^{\parr}$. □

Proposition• 11.62 (Combinator $\mathbf{cc}^{\mathcal{A}}$)**.** *For any disjunctive algebra $(\mathcal{A},\preccurlyeq,\parr,\neg,\mathcal{S})$, we have $\mathbf{cc}^{\mathcal{A}} \in \mathcal{S}$.*

Proof. We make several applications of Lemmas 11.55 and 11.54 consecutively. We prove that:

$$\bigcurlywedge_{a,b\in\mathcal{A}} ((a \to b) \to a) \to a \ \in \mathcal{S}$$

is implied by generalized modus ponens (Lemma 11.55) and:

$$\begin{array}{ll} & \bigcurlywedge_{a,b\in\mathcal{A}}((a \to b) \to a) \to (\neg a \to a \to b) \to \neg a \to a \ \in \mathcal{S} \\ \text{and} & \bigcurlywedge_{a,b\in\mathcal{A}}((\neg a \to a \to b) \to \neg a \to a) \to a \ \in \mathcal{S} \end{array}$$

The statement above is a subtype of $s_4^{\parr}$, while the other is proved, by Lemma 11.55, from:

$$\begin{array}{ll} & \bigcurlywedge_{a,b\in\mathcal{A}}((\neg a \to a \to b) \to \neg a \to a) \to \neg a \to a \ \in \mathcal{S} \\ \text{and} & \bigcurlywedge_{a\in\mathcal{A}}((\neg a) \to a) \to a \ \in \mathcal{S} \end{array}$$

The statement below is proved as in Proposition 11.58, while the statement above is proved by a variant of the modus ponens and:

$$\bigcurlywedge_{a,b\in\mathcal{A}} (\neg a \to a \to b) \ \in \mathcal{S}$$

We conclude by proving this statement using the connections between $\neg a$ and $a \to \bot$, reducing the latter to:

$$\bigcurlywedge_{a,b\in\mathcal{A}} (a \to \bot) \to a \to b \ \in \mathcal{S}$$

which is a subtype of the identity. □

As a consequence, we get the expected theorem:

Theorem• 11.63. *Any disjunctive algebra is a classical implicative algebra.*

Proof. The conditions of upward closure and closure under modus ponens coincide for implicative and disjunctive separators, and the previous propositions show that **k**, **s** and **cc** belong to the separator of any disjunctive algebra. □

Corollary 11.64. *If t is a closed λ-term and $(\mathcal{A}, \preccurlyeq, \parr, \neg, \mathcal{S})$ a disjunctive algebra, then $t^{\mathcal{A}} \in \mathcal{S}$.*

11.4.6 From implicative to disjunctive algebras

On the converse direction, we could wonder whether it is possible to get a disjunctive algebra from an implicative one. The first step in this direction would be to define a disjunctive structure from an implicative structure, and to this end, the natural candidates for the disjunction and the negation are:

$$a \parr b \triangleq a + b \qquad\qquad \neg a \triangleq a \to \bot$$

Indeed, we saw that in the implicative algebra underlying any disjunctive algebra $(\mathcal{A}, \preccurlyeq, \parr, \neg, \mathcal{S})$, we had the equivalences $a \parr b \cong_{\mathcal{S}} a + b$ and $\neg a \cong_{\mathcal{S}} a \to \bot$ (Propositions 11.57 and 11.59).

However, there is no reason for the required laws of commutation:

$$\bigcurlywedge_{b\in B} (a + b) = a + (\bigcurlywedge_{b\in B} b) \qquad \bigcurlywedge_{b\in B} (b + a) = (\bigcurlywedge_{b\in B} b) + a \qquad (\bigcurlywedge_{a\in A} a) \to \bot = \bigcurlyvee_{a\in A} (a \to \bot)$$

to hold in an implicative structure. If we focus on the particular case of implicative algebras arising from an abstract Krivine structure (or alternatively in any Krivine realizability model), the equality for the negation holds, but the equalities for the sum type are not true in general. More precisely, they hold in the case were the arrow commutes with the joins, in which case we know that any separator on such a structure will induce a forcing tripos. Nonetheless, in the case where these equalities hold, it is easy to see that any implicative algebra induces a disjunctive algebra since the axioms $s_1^{\parr}, s_2^{\parr}, s_3^{\parr}, s_4^{\parr}, s_5^{\parr}$ are all realized by closed λ-terms. Writing $\neg_\bot a$ for $a \to \bot$, we have:

Proposition 11.65. *If* $(\mathcal{A}, \preccurlyeq, \to, \mathcal{S})$ *is an implicative algebra and* $(\mathcal{A}, \preccurlyeq, +, \neg_\bot)$ *is a disjunctive structure, then* $(\mathcal{A}, \preccurlyeq, +, \neg_\bot, \mathcal{S})$ *is a disjunctive algebra.*

Proof. The conditions of upward closure and closure under modus ponens coincide for implicative and disjunctive separators, and finding realizers for $s_1^{\parr}, s_2^{\parr}, s_3^{\parr}, s_4^{\parr}, s_5^{\parr}$ (with $\parr = +$) is an easy exercise of λ-calculus. □

In other words, implicative algebras which induce disjunctive algebras through[6] $+$ and $\cdot \to \bot$ are particular cases of implicative algebras satisfying extra properties of commutation.

11.5 Conclusion

Since any disjunctive algebra is a particular case of implicative algebra, it is clear that the construction leading to the implicative tripos can be rephrased in this framework. In particular, the same criterion allows us to determine whether the implicative tripos is isomorphic to a forcing tripos. Notably, a disjunctive algebra with extra-commutations for the disjunction $\parr$ and the negation $\neg$ with arbitrary joins will induce an implicative algebra where the arrow commutes with arbitrary joins. Therefore, the induced tripos would collapse to a forcing situation (see Section 10.4.4.2).

Of course, we could reproduce the whole construction (that is studying the product of disjunctive structures, then the quotient by the uniform separator, and verifying the necessary conditions for the functor $\mathcal{T} : I \mapsto \mathcal{A}^I/\mathcal{S}[I]$ to be a tripos) directly in the setting of disjunctive algebras. Nonetheless, insofar as we are interested in the most general framework (and especially in existence of triposes which are not isomorphic to forcing triposes), there is no point in doing this. Indeed, the main conclusion that we draw from this chapter is the following slogan:

Implicative algebras are more general than disjunctive algebras.

In particular, even though we are still missing some properties which would be convenient to be able to use disjunctive algebras in practice, the former slogan dissuades us to pursue in this direction. Nonetheless, we should point out the main feature that is missing in our analysis of disjunctive algebras, namely a computational completeness with respect to $\mathrm{L}^{\parr}$. We obtained in the end that any closed λ-term is in the separator of any disjunctive algebra, which provides us with the possibility of proving that a given element belongs to the separator by finding the adequate realizer. Especially, since we know that the disjunction $a \parr b$ is equivalent, with respect to separators, to the sum type $a + b$ (and similarly for the negation $\neg a$ and the implication $a \to \bot$), any formula can be realized by a λ-term for the equivalent formula encoded with $+$ and $\neg_\bot$. However, this is not really convenient in practice and it would be nice to be able to realize formulas directly through $\mathrm{L}^{\parr}$ terms. We do not know if this is possible in the absolute. It would make sense to prove that the combinators $s_1^{\parr}, s_2^{\parr}, s_3^{\parr}, s_4^{\parr}, s_5^{\parr}$ are complete with respect to $\mathrm{L}^{\parr}$ terms, but all our attempts in this direction have shown to be unsuccessful.

[6] Of course, one could still argue that there are maybe better candidates for embedding a negation and a disjunction into implicative structures. Inasmuch as the disjunction and negation that are obtained in the construction of the implicative tripos are $+$ and $\neg_\bot$, we believe this choice to be legitimate.

12- Conjunctive algebras

In the previous chapter, we studied disjunctive algebras, which we introduced as a result of the decomposition of the implication with a disjunction and a negation. In particular, we saw that this decomposition canonically corresponds to the $\mathrm{L}^{\parr}$ calculus, into which the λ-calculus can be embedded. Notably, the so-defined λ-calculus is equipped with a call-by-name evaluation strategy, as in the Krivine abstract machine for the λ_c-calculus. We showed that this correspondence has a direct algebraic counterpart, since disjunctive algebras are in fact particular cases of implicative algebras.

We shall now study the dual case of structures resulting of the decomposition of the arrow into primitive negations and conjunctions. We mentioned in particular that Girard's decomposition of the arrow in linear logic can be expressed in terms of the multiplicative law of conjunction, written $\otimes$, by:

$$A \to B \ \triangleq\ \neg(A \otimes \neg B)$$

The connective $\otimes$ is indeed related to the disjunction $\parr$ by duality through the laws $\neg(A \parr B) = \neg A \otimes \neg B$ and $\neg(A \otimes B) = \neg A \parr \neg B$. The typing rules for this connective in linear logic are given by:

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma \mid A \otimes B \vdash \Delta} \qquad\qquad \frac{\Gamma \vdash A \mid \Delta \quad \Gamma \vdash B \mid \Delta}{\Gamma \vdash A \otimes B \mid \Delta}$$

which are again dual to the rules for the disjunction.

We shall now follow the same process as in the previous chapter, but with the conjunction $\otimes$ as a primitive connective. First, we will present $\mathrm{L}^{\otimes}$, the fragment of Munch-Maccagnoni's L calculus [126] which corresponds to the connectives $\neg$ and $\otimes$. We will observe that this fragment allows for the encoding of a call-by-value λ-calculus. Next, we will give the realizability interpretation *à la* Krivine for this calculus. Then, based on the structure of this realizability model, we will introduce the notion of conjunctive structure. We will show that these structures are dual to the disjunctive structures we formerly introduced. Again, we will show how to embed terms and contexts of $\mathrm{L}^{\otimes}$ into conjunctive structures. Finally, we will define the notion of a separator for conjunctive structures, leading to the definition of conjunctive algebras. We shall prove that any disjunctive algebra induces a conjunctive algebra by duality.

Unfortunately, we did not achieve to prove the converse, namely that disjunctive algebras could be obtained by duality from conjunctive algebras. In fact, beyond that, we are lacking some basic results to be able to manipulate elements of conjunctive structures in the same computational fashion as in implicative or disjunctive algebras. As a consequence, we do not prove that disjunctive algebras can be recovered from conjunctive algebras by duality. As such, our study of conjunctive algebras thus remains incomplete. We shall come back to this aspect in the conclusion of this chapter.

12.1 A call-by-value decomposition of the arrow

We begin with the presentation of the fragment of L induced by the positive connectives $\otimes, \neg^{+}$ and $,\exists$. Next we shall see the realizability interpretation it induces, with the purpose of justifying afterwards the definition of conjunctive structures. Again, since this calculus has a lot of similarities with the call-by-value $\lambda\mu\tilde{\mu}$-calculus (see Section 4.5) in addition to being dual to $L^{⅋}$, we shall try to be concise in this section.

12.1.1 The $L^{\otimes}$ calculus

The $L^{\otimes}$ calculus is thus a subsystem of L. It corresponds exactly to the restriction of L to its positive fragment induced by the connectives $\otimes, \neg$ and $, \exists$. The syntax of terms, contexts and commands is given by:

Contexts	e^-	$::=$	$\alpha \mid \mu(x,y).c \mid \mu[\alpha].c \mid \mu x.c$
Terms	t^+	$::=$	$x \mid (t,t) \mid [e] \mid \mu\alpha.c$
Commands	c	$::=$	$\langle t^+ \| e^- \rangle$

We write $\mathcal{T}_0$, $\mathcal{E}_0$, C_0 for the sets of closed terms, contexts and commands. In this framework, values are defined by:

Values $\quad V ::= x \mid (V,V) \mid [e^-]$

Observe in particular that any (negative) context is a value. We denote by $\mathcal{V}_0$ the set of closed values. The syntax is really close to the one of $L^{⅋}$ (it has the same constructors, but terms are now positive while contests are negative), we recall the meanings of the different constructions:

- (t^+,t^+) are pairs of positive terms;
- $\mu(x_1,x_2).c$, which binds the variables x_1,x_2, is the dual destructor;
- $[e^-]$ is a constructor for the negation, which allows us to embed a negative context intro a positive term;
- $\mu[x].c$, which binds the variable x, is the dual destructor;
- $\mu\alpha.c$ and $\mu x.c$ correspond respectively to $\mu\alpha$ and $\tilde{\mu}x$ in the $\lambda\mu\tilde{\mu}$-calculus.

Remark 12.1 (Notations)**.** As we explained in the previous chapter, in L [126] is considered a syntax where a notation $\bar{x}$ is used to distinguish between the positive variable x (that can appear in the left-member $\langle x|$ of a command) and the co-variable $\bar{x}$ (resp. in the right member $|x\rangle$ of a command). The positive variable that we write x is also written x in [126], while the negative co-variable α is denoted by $\bar{\alpha}$. ⌟

The reduction rules correspond to the intuition one could have from the syntax of the calculus: all destructors and binders reduce in front of the corresponding values, while pairs of terms are expanded if needed. The rules are given by:

$$\begin{array}{rcl} \langle \mu\alpha.c \| e\rangle & \to_\beta & c[e/\alpha] \\ \langle [e] \| \mu[\alpha].c\rangle & \to_\beta & c[e/\alpha] \\ \langle V \| \mu x.c\rangle & \to_\beta & c[V/x] \\ \langle (V,V') \| \mu(x,x').c\rangle & \to_\beta & c[V/x,V'/x'] \\ \langle (t,u) \| e\rangle & \to_\beta & \langle t \| \mu x.\langle u \| \mu y.\langle (x,y) \| e\rangle\rangle\rangle \end{array} \qquad \begin{array}{rcl} c & \to_\eta & \langle \mu\alpha.c \| \alpha\rangle \\ c & \to_\eta & \langle [\alpha] \| \mu[\alpha].c\rangle \\ c & \to_\eta & \langle x \| \mu x.c\rangle \\ c & \to_\eta & \langle (x_1,x_2) \| \mu(x_1,x_2).c\rangle \end{array}$$

where $(t,u) \notin V$ in the last β-reduction rule.

Lastly, we shall present the type system of $L^{\otimes}$. Second-order formulas are defined from the positive connectives by:

Formulas $\quad A,B := X \mid A \otimes B \mid \neg A \mid \exists X.A$

$$\frac{\Gamma \vdash t : A \mid \Delta \quad \Gamma \mid e : A \vdash \Delta}{\langle t \| e \rangle : \Gamma \vdash \Delta}\ (\textsc{Cut})$$

$$\frac{(\alpha : A) \in \Delta}{\Gamma \mid \alpha : A \vdash \Delta}\ (ax\vdash) \qquad \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A \mid \Delta}\ (\vdash ax)$$

$$\frac{c : \Gamma \vdash \Delta, x : A}{\Gamma \mid \mu x.c : A \vdash \Delta}\ (\mu\vdash) \qquad \frac{c : \Gamma, \alpha : A \vdash \Delta}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\ (\vdash\mu)$$

$$\frac{c : (\Gamma, x : A, x' : B \vdash \Delta)}{\Gamma \mid \mu(x,x').c : A \otimes B \vdash \Delta}\ (\otimes\vdash) \qquad \frac{\Gamma \vdash t : A \mid \Delta \quad \Gamma \vdash u : B \mid \Delta}{\Gamma \vdash (t,u) : A \otimes B \mid \Delta}\ (\vdash\otimes)$$

$$\frac{c : \Gamma, x : A \vdash \Delta}{\Gamma \mid \mu[\alpha].c : \neg A}\ (\neg\vdash) \qquad \frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \vdash [e] : \neg A \vdash \Delta}\ (\vdash\neg)$$

$$\frac{\Gamma \vdash e : A \mid \Delta \quad X \notin FV(\Gamma,\Delta)}{\Gamma \mid e : \exists X.A \vdash \Delta}\ (\exists_l) \qquad \frac{\Gamma \vdash V : A[B/X] \mid \Delta}{\Gamma \vdash V : \exists X.A}\ (\exists_r)$$

Figure 12.1: Typing rules for the $\mathrm{L}^{\otimes}$-calculus

We still work with two-sided sequents, where typing contexts are defined as finite lists of bindings between variable and formulas:

$$\Gamma ::= \varepsilon \mid \Gamma, x : A \qquad\qquad \Delta ::= \varepsilon \mid \Delta, \alpha : A$$

Sequents are again of three kinds, as in the $\lambda\mu\tilde{\mu}$-calculus and $\mathrm{L}^{⅋}$:

- $\Gamma \vdash t : A \mid \Delta$ for typing terms,
- $\Gamma \mid e : A \vdash \Delta$ for typing contexts,
- $c : \Gamma \vdash \Delta$ for typing commands.

The type system is given in Figure 12.1, where each connective corresponds to a left and a right rule.

Remark 12.2 (Existential quantifier)**.** As in the type system of $\mathrm{L}^{⅋}$, we do not associate the existential quantifier to a constructor. Indeed, since our primary motivation is the definition of conjunctive structures, in which this quantifier will simply be expressed by arbitrary joins, it would be irrelevant to add a constructor now. In turn, observe that we restrict the introduction of the existential quantifier to values. ⌟

12.1.2 Embedding of the λ-calculus

Guided by the expected definition of the arrow:

$$A \to B \ \triangleq\ \neg(A \otimes \neg B)$$

we can follow Munch-Maccagnoni's paper [126, Appendix E], to embed the λ-calculus into $\mathrm{L}^{\otimes}$.

With this definition, a stack $u \cdot e$ in $A \to B$ (that is with u a term of type A and e a context of type B) is naturally embedded as a term $(u,[e])$, which is turn into the context $\mu[\alpha].\langle (u,[e]) \| \alpha \rangle$ which indeed inhabits the "arrow" type $\neg(A \otimes \neg B)$. Starting from this, the rest of the definitions are direct:

$$\begin{aligned}
\mu(x,[\alpha]).c &\ \triangleq\ \mu(x,x').\langle x' \| \mu[\alpha].c \rangle \\
\lambda x.t &\ \triangleq\ [\mu(x,[\alpha]).\langle t \| \alpha \rangle] \\
t \cdot e &\ \triangleq\ \mu[\alpha].\langle (t,[e]) \| \alpha \rangle \\
t\,u &\ \triangleq\ \mu\alpha.\langle t \| u \cdot \alpha \rangle
\end{aligned}$$

These shorthands allow for the expected typing rules:

Proposition 12.3. *The following typing rules are admissible:*

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash u : A \mid \Delta \quad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid u \cdot e : A \to B \vdash \Delta} \qquad \frac{\Gamma \vdash t : A \to B \mid \Delta \quad \Gamma \vdash u : A \mid \Delta}{\Gamma \vdash t\,u : B \mid \Delta}$$

Proof. Each case is directly derivable from $\mathrm{L}^{\otimes}$ type system. We abuse the notations to denote by (*def*) a rule which simply consists in unfolding the shorthands defining the λ-terms.

- **Case** $\mu(x, [\alpha]).c$:

$$\dfrac{\dfrac{\dfrac{\dfrac{c : (\Gamma, x : A \vdash \Delta, \alpha : B)}{\Gamma \vdash \mu[x].c : \neg A \mid \Delta, \beta : B}\,(\mu\vdash) \qquad \dfrac{}{\Gamma, x : A, x' : \neg B \vdash x' : \neg B \mid \Delta}\,(\vdash ax)}{\langle x' \| \mu[\alpha].c\rangle : (\Gamma, x : A, x' : \neg B \vdash \Delta)}\,(\textsc{Cut})}{\Gamma \mid \mu(x, x').\langle x' \| \mu[\alpha].c\rangle : A \otimes \neg B \vdash \Delta}\,(\otimes\vdash)}{\Gamma \mid \mu(x, [\alpha]).c : A \otimes \neg B \vdash \Delta}\,(\mathit{def})$$

- **Case** $\lambda x.t$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, x : A \vdash t : B \mid \Delta \qquad \dfrac{}{\Gamma \mid \beta : B \vdash \Delta, \beta : B}\,(ax\vdash)}{\langle t \| \beta\rangle : (\Gamma, x : A \vdash \beta : B, \Delta)}\,(\textsc{Cut})}{\Gamma \mid \mu(x, [\beta]).\langle t \| \beta\rangle : A \otimes \neg B \vdash \Delta}}{\Gamma \vdash [\mu(x, [\beta]).\langle t \| \beta\rangle] : \neg(A \otimes \neg B) \mid \Delta}\,(\vdash\neg)}{\Gamma \vdash \lambda x.t : A \to B \mid \Delta}\,(\mathit{def})$$

- **Case** $u \cdot e$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash u : A \vdash \Delta \qquad \dfrac{\Gamma \mid e : B \vdash \Delta}{\Gamma \vdash [e] : \neg B \mid \Delta}\,(\vdash\neg)}{\Gamma \vdash (u, [e]) : A \otimes \neg B \mid \Delta}\,(\vdash\otimes) \qquad \dfrac{}{\Gamma \mid \alpha : (A \otimes \neg B) \vdash \Delta, \alpha : (A \otimes \neg B)}\,(ax\vdash)}{\langle (u, [e]) \| \alpha\rangle : (\Gamma \vdash \Delta, \alpha : A \otimes \neg B)}\,(\textsc{Cut})}{\Gamma \mid \mu[\alpha].\langle (u, [e]) \| \alpha\rangle : \neg(A \otimes \neg B) \vdash \Delta}\,(\neg\vdash)}{\Gamma \mid u \cdot e : A \to B \vdash \Delta}\,(\mathit{def})$$

- **Case** $t\,u$:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash t : A \to B \mid \Delta \qquad \dfrac{\Gamma \vdash u : A \mid \Delta \quad \dfrac{}{\Gamma \mid \alpha : B \vdash \Delta, \alpha : B}}{\Gamma \mid u \cdot \alpha : A \to B \vdash \Delta, \alpha : B}}{\langle t \| u \cdot \alpha\rangle : (\Gamma \vdash \Delta, \alpha : B)}\,(\textsc{Cut})}{\Gamma \vdash \mu\alpha.\langle t \| u \cdot \alpha\rangle : B \mid \Delta}\,(\vdash\mu)}{\Gamma \vdash t\,u : B \mid \Delta}\,(\mathit{def})$$

□

Besides, the usual rules of β-reduction for the call-by-value evaluation strategy are simulated through the reduction of $\mathrm{L}^{\otimes}$:

Proposition 12.4 (β-reduction)**.** *We have the following reduction rules:*

$$\begin{array}{rcl} \langle t\,u \| e\rangle & \to_\beta & \langle t \| u \cdot e\rangle \\ \langle \lambda x.t \| u \cdot e\rangle & \to_\beta & \langle u \| \mu x.\langle t \| e\rangle\rangle \\ \langle V \| \mu x.c\rangle & \to_\beta & c[V/x] \end{array}$$

Proof. The third rule is included in L$^{\otimes}$ reduction system, the first follows from:

$$\langle tu \| e\rangle = \langle \mu\alpha.\langle t \| u\cdot\alpha\rangle \| e\rangle \to_\beta \langle t \| u\cdot e\rangle$$

For the second rule, we first check that we have:

$$\langle (V,[e]) \| \mu(x,[\alpha]).c\rangle = \langle (V,[e]) \| \mu(x,x').\langle x' \| \mu[\alpha].c\rangle\rangle \to_\beta \langle [e] \| \mu[\alpha].c[V/X]\rangle \to_\beta c[V/x][e/\alpha]$$

from which we deduce:

$$\begin{aligned}\langle \lambda x.t \| u\cdot e\rangle &= \langle [\mu(x,[\alpha]).\langle t\|\alpha\rangle] \| \mu[\alpha].\langle (u,[e]) \| \alpha\rangle\rangle\\ &\to_\beta \langle (u,[e]) \| \mu(x,[\alpha]).\langle t\|\alpha\rangle\rangle\\ &\to_\beta \langle u \| \mu y.\langle (y,[e]) \| \mu(x,[\alpha]).\langle t\|\alpha\rangle\rangle\rangle\\ &\to_\beta \langle u \| \mu x.\langle t \| e\rangle\rangle\end{aligned}$$

□

Therefore, L$^{\otimes}$ allows us to recover the full computation strength of the call-by-value $\lambda\mu\tilde{\mu}$-calculus. We shall now see that it is suitable for a realizability interpretation which is very similar to the corresponding interpretation for the call-by-value $\lambda\mu\tilde{\mu}$-calculus (see Section 4.5.4).

12.1.3 A realizability model based on the L$^{\otimes}$-calculus

We briefly recall the definitions necessary to the realizability interpretation *à la* Krivine of L$^{\otimes}$. Most of the properties being the same as for L$^{⅋}$ or any of the several interpretations we gave in the previous chapters, we spare the reader from a useless copy-paste and go straight to the point.

A *pole* is defined as usual as any subset of C_0 closed by anti-reduction. We write $\perp\!\!\!\perp$ for the pole, and $t \perp\!\!\!\perp e$ for the orthogonality relation it induces. As it is common in call-by-value realizability model (see Section 4.5.4), formulas as interpreted as *truth values of values*, which we call *primitive truth values. Falsity values* are then defined by orthogonality to the corresponding primitive truth values, and *truth values* are defines by orthogonality to falsity values. Therefore, an existential formula $\exists X.A$ is interpreted by the union over all the possible instantiations for the primitive truth value of the variable X by a set $S \in \mathcal{P}(\mathcal{V}_0)$. As it is usual in Krivine realizability, in order to ease the definition we assume that for each subset S of $\mathcal{P}(\mathcal{V}_0)$, there is a constant symbol $\dot{S}$ in the syntax. The interpretation is given by:

$$\begin{array}{rcl}|\dot{S}|_V & \triangleq & S\\ |A\otimes B|_V & \triangleq & \{(t,u) : t\in |A|_V \wedge u \in |B|_V\}\\ |\neg A|_V & \triangleq & \{[e] : e \in \|A\|\}\\ |\exists X.A|_V & \triangleq & \bigcup_{S\in\mathcal{P}(\mathcal{V}_0)} |A\{X := \dot{S}\}|_V\\ \|A\| & \triangleq & \{e : \forall V \in |A|_V, V \perp\!\!\!\perp e\}\\ |A| & \triangleq & \{t : \forall e \in \|A\|, t \perp\!\!\!\perp e\}\end{array}$$

We define again *valuations*, which we write ρ, as functions mapping each second-order variable to a primitive falsity value $\rho(X) \in \mathcal{P}(\mathcal{V}_0)$. In this framework, we say that a *substitution*, which we denote by σ, is a function mapping each variable x to a closed value $V \in \mathcal{V}_0$ and each variable α to a closed context $e \in \mathcal{E}_0$:

$$\sigma ::= \varepsilon \mid \sigma, x \mapsto V \mid \sigma, \alpha \mapsto e$$

We write $\sigma \Vdash \Gamma$ and we say that a substitution σ realizes a context Γ, when for each binding $(x : A) \in \Gamma$, we have $\sigma(x) \in |A|_V$. Similarly, we say that σ realizes a context Δ if for each binding $(\alpha : A) \in \Delta$, we have $\sigma(\alpha) \in \|A\|$.

Lemma 12.5 (Adequacy)**.** *Let Γ, Δ be typing contexts, ρ be a valuation and σ be a substitution which verifies that $\sigma \Vdash \Gamma[\rho]$ and $\sigma \Vdash \Delta[\rho]$. We have:*

1. *If V^+ is a value such that $\Gamma \vdash V^+ : A \mid \Delta$, then $V^+[\sigma] \in |A[\rho]|_V$.*
2. *If e is a context such that $\Gamma \mid e : A \vdash \Delta$, then $e[\sigma] \in \|A[\rho]\|$.*
3. *If t is a term such that $\Gamma \vdash t : A \mid \Delta$, then $t[\sigma] \in |A[\rho]|$.*
4. *If c is a command such that $c : (\Gamma \vdash \Delta)$, then $c[\sigma] \in \bot\!\!\!\bot$.*

Proof. The proof is again an induction over typing derivations. The proof being very similar to the one for $\mathrm{L}^{\parr}$ (Proposition 11.10), the call-by-value $\lambda\mu\tilde{\mu}$-calculus (Proposition 4.23) or L [126], we leave it to the reader. □

12.2 Conjunctive structures

We shall now introduce the notion of *conjunctive structure.* Following the methodology from the previous chapter, we begin by observing the existing commutations in the realizability models induced by $\mathrm{L}^{\otimes}$. Since we are in a structure centered on positive connectives, we should pay attention to the commutations with joins:

Proposition 12.6 (Commutations)**.** *In any $L^{\otimes}$ realizability model (that is to say for any pole $\bot\!\!\!\bot$), the following equalities hold:*

1. *If $X \notin FV(B)$, then $|\exists X.(A \otimes B)|_V = |(\exists X.A) \otimes B|_V$.*
2. *If $X \notin FV(A)$, then $|\exists X.(A \otimes B)|_V = |A \otimes (\exists X.B)|_V$.*
3. *$|\neg(\exists X.A)|_V = \bigcap_{S\in\mathcal{P}(\mathcal{V}_0)} |\neg A\{X := \dot{S}\}|_V$*

Proof. 1. Assume the $X \notin FV(B)$, then we have:

$$\begin{aligned}
|\exists X.(A \otimes B)|_V &= \bigcup_{S\in\mathcal{P}(\mathcal{V}_0)} |A\{X := \dot{S}\} \otimes B|_V \\
&= \bigcup_{S\in\mathcal{P}(\mathcal{V}_0)} \{(V_1, V_2) : V_1 \in |A\{X := \dot{S}\}|_V \wedge V_2 \in |B|_V\} \\
&= \{(e_1, e_2) : e_1 \in \bigcup_{S\in\mathcal{P}(\mathcal{V}_0)} |A\{X := \dot{S}\}|_V \wedge e_2 \in |B|_V\} \\
&= \{(e_1, e_2) : e_1 \in |\exists X.A|_V \wedge e_2 \in \|B\|\} \quad = \quad |(\exists X.A) \otimes B|_V
\end{aligned}$$

2. Identical.
3. The proof is again a simple unfolding of the definitions:

$$\begin{aligned}
|\neg(\exists X.A)|_V = \{[t] : t \in |\exists X.A|\} \quad &= \quad \{[t] : t \in \bigcap_{S\in\mathcal{P}(\mathcal{V}_0)} |A\{X := \dot{S}\}|\} \\
= \bigcap_{S\in\mathcal{P}(\mathcal{V}_0)} \{[t] : t \in |A\{X := \dot{S}\}]|\} &= \bigcap_{S\in\mathcal{P}(\mathcal{V}_0)} |\neg A\{X := \dot{S}\}|_V
\end{aligned}$$

□

Since we are interested in primitive truth values, which are logically ordered by inclusion (in particular, the existential quantifier is interpreted by unions, thus joins), in terms of algebraic structures, the previous proposition advocates for the equalities:

1. $\displaystyle\curlyvee_{b\in B} (a \otimes b) = a \otimes (\curlyvee_{b\in B} b)$ 2. $\displaystyle\curlyvee_{b\in B} (b \otimes a) = (\curlyvee_{b\in B} b) \otimes a$ 3. $\displaystyle\neg \curlyvee_{a\in A} a = \curlywedge_{a\in A} \neg a$

Definition• 12.7 (Conjunctive structure)**.** A *conjunctive structure* is a complete join-semilattice $(\mathcal{A}, \preccurlyeq)$ equipped with a binary operation $(a, b) \mapsto a \otimes b$, called the *conjunction of* $\mathcal{A}$, and a unary operation $a \mapsto \neg a$ called the *negation of* $\mathcal{A}$, that fulfill the following axioms:

1. Negation is anti-monotonic in the sense that for all $a, a' \in \mathcal{A}$:

   (Variance) $\qquad$ if $a \preccurlyeq a'$ then $\neg a' \preccurlyeq \neg a$

2. Conjunction is monotonic in the sense that for all $a, a', b, b' \in \mathcal{A}$:

   (Variance) $\qquad$ if $a \preccurlyeq a'$ and $b \preccurlyeq b'$ then $a \otimes b \preccurlyeq a' \otimes b'$

3. Arbitrary meets distributes over both operands of conjunction, in the sense that for all $a \in \mathcal{A}$ and for all subsets $B \subseteq \mathcal{A}$:

   (Distributivity) $$\curlyvee_{b \in B} (a \otimes b) = a \otimes (\curlyvee_{b \in B} b) \qquad \curlyvee_{b \in B} (b \otimes a) = (\curlyvee_{b \in B} b) \otimes a$$

4. Negation of an arbitrary join is equal to the meet of the set of negated elements, in the sense that for all subsets $A \subseteq \mathcal{A}$:

   (Commutation) $$\neg \curlyvee_{a \in A} a = \curlywedge_{a \in A} \neg a$$

⌟

Remark 12.8. Recall that a complete join-semilattice is a complete lattice (Theorem 9.3). Therefore, conjunctive structures also have arbitrary meets. The novelty, in comparison with implicative and disjunctive structures, is that the definition of conjunctive separators will make use of arbitrary meets (while the properties of distributivity and commutation are given for arbitrary joins). This mismatch is at the origin of most of the difficulties that we will meet in the sequel. ⌟

As in the cases of implicative and disjunctive structures, the commutations imply that:

Proposition 12.9. *If* $(\mathcal{A}, \preccurlyeq, \otimes, \neg)$ *is a conjunctive structure, then the following hold for all* $a \in \mathcal{A}$*:*

1.• $\bot \otimes a = \bot$

2.• $a \otimes \bot = \bot$

3.• $\neg \bot = \top$

Proof. Using proposition 9.4 and the axioms of conjunctive structures, one can prove:

1. $\bot \otimes a = (\curlyvee \emptyset) \otimes a = \curlyvee_{x, a \in \mathcal{A}} \{x \otimes a : x \in \emptyset\} = \curlyvee \emptyset = \bot$
2. Identical.
3. $\neg \bot = \neg(\curlyvee \emptyset) = \curlywedge_{x \in \mathcal{A}} \{\neg x : x \in \emptyset\} = \curlywedge \emptyset = \top$

□

12.2.1 Examples of conjunctive structures

12.2.1.1 Dummy structure

Following the constraints given by the lemma above, we have at least one way to define a dummy structure:

Example• 12.10 (Dummy conjunctive structure)**.** Given a complete lattice L, the following definitions give rise to a dummy structure that fulfills the axioms of Definition 11.13:

$$a \otimes b \triangleq \bot \qquad \neg a \triangleq \top \qquad (\forall a, b \in \mathcal{A})$$

The verification of the different axioms is straightforward. ⌟

12.2.1.2 Complete Boolean algebras

Example• 12.11 (Complete Boolean algebras)**.** Let $\mathcal{B}$ be a complete Boolean algebra. It embodies a conjunctive structure, that is defined by:

- $\mathcal{A} \triangleq \mathcal{B}$
- $a \preccurlyeq b \triangleq a \preccurlyeq b$
- $a \otimes b \triangleq a \wedge b$
- $\neg a \triangleq \neg a$

$(\forall a,b \in \mathcal{A})$

The different axioms are direct consequence of proposition 9.7. ⌟

12.2.2 Conjunctive structure of classical realizability

As for the disjunctive case, we can abstract the structure of the realizability interpretation of $L^{\otimes}$ into a structure of the form $(\mathcal{T}_0,\mathcal{E}_0,\mathcal{V}_0,(\cdot,\cdot),[\cdot],\perp\!\!\!\perp)$, where $\mathcal{V}_0 \subseteq \mathcal{T}_0$ is the distinguished subset of values, $(\cdot,\cdot)$ is a map from ${\mathcal{T}_0}^2$ to $\mathcal{T}_0$ (whose restriction to $\mathcal{V}_0$ has values in $\mathcal{V}_0$), $[\cdot]$ is an operation from $\mathcal{E}_0$ to $\mathcal{V}_0$, and $\perp\!\!\!\perp \subseteq \mathcal{T}_0 \times \mathcal{E}_0$ is a relation. From this sextuple we can define:

- $\mathcal{A} \triangleq \mathcal{P}(\mathcal{V}_0)$
- $a \preccurlyeq b \triangleq a \subseteq b$
- $a \otimes b \triangleq (a,b) = \{(V_1,V_2) : V_1 \in a \wedge V_2 \in b\}$
- $\neg a \triangleq [a^{\perp\!\!\!\perp}] = \{[e] : e \in a^{\perp\!\!\!\perp}\}$

$(\forall a,b \in \mathcal{A})$

Proposition 12.12. *The quadruple* $(\mathcal{A},\preccurlyeq,\otimes,\neg)$ *is a conjunctive structure.*

Proof. We show that the axioms of Definition 12.7 are satisfied.

1. Anti-monotonicity. Let $a,a' \in \mathcal{A}$, such that $a \preccurlyeq a'$ ie $a \subseteq a'$. Then $a'^{\perp\!\!\!\perp} \subseteq a^{\perp\!\!\!\perp}$ and thus
$$\neg a' = \{[t] : t \in a'^{\perp\!\!\!\perp}\} \subseteq \{[t] : t \in a^{\perp\!\!\!\perp}\} = \neg a$$
i.e. $\neg a' \preccurlyeq \neg a$.

2. Covariance of the conjunction. Let $a,a',b,b' \in \mathcal{A}$ such that $a' \subseteq a$ and $b' \subseteq b$. Then we have
$$a \otimes b = \{(t,u) : t \in a \wedge u \in b\} \subseteq \{(t,u) : t \in a' \wedge u \in b'\} = a' \otimes b'$$
i.e. $a \otimes b \preccurlyeq a' \otimes b'$

3. Distributivity. Let $a \in \mathcal{A}$ and $B \subseteq \mathcal{A}$, we have:
$$\bigcurlyvee_{b\in B} (a \otimes b) = \bigcurlyvee_{b\in B} \{(v,u) : t \in a \wedge u \in b\} = \{(t,u) : t \in a \wedge u \in \bigcurlyvee_{b\in B} b\} = a \otimes (\bigcurlyvee_{b\in B} b)$$

4. Commutation. Let $B \subseteq \mathcal{A}$, we have (recall that $\bigcurlywedge_{b\in B} b = \bigcap_{b\in B} b$):
$$\bigcurlywedge_{b\in B} \{\neg b\} = \bigcurlywedge_{b\in B} \{[t] : t \in b^{\perp\!\!\!\perp}\} = \{[t] : t \in \bigcurlywedge_{b\in B} b^{\perp\!\!\!\perp}\} = \{[t] : t \in (\bigcurlyvee_{b\in B} b)^{\perp\!\!\!\perp}\} = \neg(\bigcurlyvee_{b\in B} b)$$

□

12.2.3 Interpreting $L^{\otimes}$ terms

We shall now see how to embed $L^{\otimes}$ commands, contexts and terms into any conjunctive structure. For the rest of the section, we assume given a conjunctive structure $(\mathcal{A},\preccurlyeq,\otimes,\neg)$.

12.2.3.1 Commands

Following the same intuition as for the embedding of $\mathrm{L}^{\parr}$ into disjunctive structures, we define the *commands* $\langle a \| b\rangle$ of the conjunctive structure $\mathcal{A}$ as the pairs (a,b), and we define the pole $\perp\!\!\!\perp$ as the ordering relation $\preccurlyeq$. We write $C_{\mathcal{A}} = \mathcal{A} \times \mathcal{A}$ for the set of commands in $\mathcal{A}$ and $(a,b) \in \perp\!\!\!\perp$ for $a \preccurlyeq b$.

We consider the same relation $\trianglelefteq$ over $C_{\mathcal{A}}$, which was defined by:

$$c \trianglelefteq c' \quad \triangleq \quad \text{if } c \in \perp\!\!\!\perp \text{ then } c' \in \perp\!\!\!\perp \qquad\qquad (\forall c, c' \in C_{\mathcal{A}})$$

Since the definition of commands only relies on the underlying lattice of $\mathcal{A}$, the relation $\trianglelefteq$ has the same properties as in disjunctive structures and in particular it defines a preorder (see Section 11.2.4.1).

12.2.3.2 Terms

The definitions of terms are very similar to the corresponding definitions for the dual contexts in disjunctive structures.

Definition• 12.13 (Pairing). For all $a, b \in \mathcal{A}$, we let $(a,b) \triangleq a \otimes b$. ⌟

Definition• 12.14 (Boxing). For all $a \in \mathcal{A}$, we let $[a] \triangleq \neg a$. ⌟

Definition• 12.15 (μ^+).

$$\mu^+.c \triangleq \bigcurlywedge_{a \in \mathcal{A}} \{a : c(a) \in \perp\!\!\!\perp\}$$

⌟

We have the following properties for μ^+:, whose proofs are trivial:

Proposition 12.16 (Properties of μ^+). *For any functions $c, c' : \mathcal{A} \to C_{\mathcal{A}}$, the following hold:*

1.• *If for all $a \in \mathcal{A}$, $c(a) \trianglelefteq c'(a)$, then $\mu^+.c' \preccurlyeq \mu^+.c$* *(Variance)*
2.• *For all $t \in \mathcal{A}$, then $t = \mu^+.(a \mapsto \langle t \| a\rangle)$* *($\eta$-expansion)*
3.• *For all $e \in \mathcal{A}$, then $\langle \mu^+.c \| e\rangle \trianglelefteq c(e)$* *($\beta$-reduction)*

Proof. 1. Direct consequence of Proposition 11.21.

2,3. Trivial by definition of μ^+. □

12.2.3.3 Contexts

Dually to the definitions of the (positive) contexts μ^+ as a meet, we define the embedding of (negative) terms, which are all binders, by arbitrary joins:

Definition• 12.17 (μ^-). For all $c : \mathcal{A} \to C_{\mathcal{A}}$, we define:

$$\mu^-.c \triangleq \bigcurlyvee_{a \in \mathcal{A}} \{a : c(a) \in \perp\!\!\!\perp\}$$

⌟

Definition• 12.18 ($\mu^{()}$). For all $c : \mathcal{A}^2 \to C_{\mathcal{A}}$, we define:

$$\mu^{()}.c \triangleq \bigcurlyvee_{a,b \in \mathcal{A}} \{a \otimes b : c(a,b) \in \perp\!\!\!\perp\}$$

⌟

Definition$^\bullet$ 12.19 ($\mu^{[]}$)**.** For all $c : \mathcal{A} \to C_{\mathcal{A}}$, we define:

$$\mu^{[]}.c \triangleq \bigcurlyvee_{a \in \mathcal{A}} \{\neg a : c(a) \in \perp\!\!\!\perp\}$$

$\lrcorner$

Again, these definitions satisfy variance properties with respect to the preorder $\trianglelefteq$ and the order relation $\preccurlyeq$. Observe that the $\mu^{()}$ and μ^{-} binders, which are negative binders catching positive terms, are contravariant with respect to these relations while the $\mu^{[]}$ binder, which catches a negative context, is covariant.

Proposition 12.20 (Variance)**.** *For any functions c, c' with the corresponding arities, the following hold:*

1.$^\bullet$ *If $c(a) \trianglelefteq c'(a)$ for all $a \in \mathcal{A}$, then $\mu^{-}.c' \preccurlyeq \mu^{-}.c$*
2.$^\bullet$ *If $c(a,b) \trianglelefteq c'(a,b)$ for all $a, b \in \mathcal{A}$, then $\mu^{()}.c' \preccurlyeq \mu^{()}.c$*
3.$^\bullet$ *If $c(a) \trianglelefteq c'(a)$ for all $a \in \mathcal{A}$, then $\mu^{[]}.c \preccurlyeq \mu^{[]}.c'$*

Proof. Direct consequences of Proposition 11.21. □

The η-expansion is also reflected by the ordering relation $\preccurlyeq$:

Proposition 12.21 (η-expansion)**.** *For all $t \in \mathcal{A}$, the following holds:*

1.$^\bullet$ $\mu^{-}.(a \mapsto \langle t \| a \rangle) = t$
2.$^\bullet$ $\mu^{()}.(a, b \mapsto \langle t \| (a,b) \rangle) \preccurlyeq t$
3.$^\bullet$ $\mu^{[]}.(a \mapsto \langle t \| [a] \rangle) \preccurlyeq t$

Proof. Trivial from the definitions. □

The β-reduction is again reflected by the preorder $\trianglelefteq$ as the property of subject reduction:

Proposition 12.22 (β-reduction)**.** *For all $e, e_1, e_2, t \in \mathcal{A}$, the following holds:*

1.$^\bullet$ $\langle \mu^{-}.c \| e \rangle \trianglelefteq c(e)$
2.$^\bullet$ $\langle \mu^{()}.c \| (e_1, e_2) \rangle \trianglelefteq c(e_1, e_2)$
3.$^\bullet$ $\langle \mu^{[]}.c \| [t] \rangle \trianglelefteq c(t)$

Proof. Trivial from the definitions. □

12.2.4 Adequacy

We shall now prove that the interpretation of $\mathrm{L}^{\otimes}$ is adequate with respect to its type system. Again, we extend the syntax of formulas to define second-order formulas with parameters by:

$$A, B ::= a \mid X \mid \neg A \mid A \otimes B \mid \exists X.A \qquad\qquad (a \in \mathcal{A})$$

This allows us to define an embedding of closed formulas with parameters into the conjunctive structure $\mathcal{A}$;

$$\begin{array}{rcll} a^{\mathcal{A}} & \triangleq & a & \text{(if } a \in \mathcal{A}\text{)} \\ (\neg A)^{\mathcal{A}} & \triangleq & \neg A^{\mathcal{A}} & \\ (A \otimes B)^{\mathcal{A}} & \triangleq & A^{\mathcal{A}} \otimes B^{\mathcal{A}} & \\ (\exists X.A)^{\mathcal{A}} & \triangleq & \curlyvee_{a \in \mathcal{A}} (A\{X := a\})^{\mathcal{A}} & \end{array}$$

As in the previous chapter, we define substitutions, which we write σ, as functions mapping variables (of terms, contexts and types) to element of $\mathcal{A}$:

$$\sigma ::= \varepsilon \mid \sigma[x \mapsto a] \mid \sigma[\alpha \mapsto a] \mid \sigma[X \mapsto a] \qquad\qquad (a \in \mathcal{A},\ x,X \text{ variables})$$

We say that a substitution σ realizes a typing context Γ, which write $\sigma \Vdash \Gamma$, if for all bindings $(x : A) \in \Gamma$ we have $\sigma(x) \preccurlyeq (A[\sigma])^{\mathcal{A}}$. Dually, we say that σ realizes Δ if for all bindings $(\alpha : A) \in \Delta$, we have $\sigma(\alpha) \succcurlyeq (A[\sigma])^{\mathcal{A}}$.

Theorem 12.23 (Adequacy). *The typing rules of* $L^{\otimes}$ *(Figure 12.1) are adequate with respect to the interpretation of terms (contexts,commands) and formulas: for all contexts* Γ,Δ*, for all formulas with parameters* A *and for all substitutions* σ *such that* $\sigma \Vdash \Gamma$ *and* $\sigma \Vdash \Delta$*, we have:*

1. *For any term* t*, if* $\Gamma \vdash t : A \mid \Delta$*, then* $\ (t[\sigma])^{\mathcal{A}} \preccurlyeq A[\sigma]^{\mathcal{A}}$*;*
2. *For any context* e*, if* $\Gamma \mid e : A \vdash \Delta$*, then* $\ (e[\sigma])^{\mathcal{A}} \succcurlyeq A[\sigma]^{\mathcal{A}}$*;*
3. *For any command* c*, if* $\ c : (\Gamma \vdash \Delta)$*, then* $\ (c[\sigma])^{\mathcal{A}} \in \bot\!\!\!\bot$*.*

Proof. By induction on the typing derivations. Since most of the cases are similar to the corresponding cases for the adequacy of the embedding of $L^{⅋}$ into disjunctive structures, we only give some key cases.

- **Case** ($\vdash \otimes$). Assume that we have:

$$\frac{\Gamma \vdash t_1 : A_1 \mid \Delta \quad \Gamma \vdash t_2 : A_2 \mid \Delta}{\Gamma \vdash (t_1,t_2) : A_1 \otimes A_2 \mid \Delta}\ {\scriptstyle (\vdash \otimes)}$$

By induction hypotheses, we have that $(t_1[\sigma])^{\mathcal{A}} \preccurlyeq (A_1[\sigma])^{\mathcal{A}}$ and $(t_2[\sigma])^{\mathcal{A}} \preccurlyeq (A_2[\sigma])^{\mathcal{A}}$. Therefore, by monotonicity of the $\otimes$ operator, we have:

$$((t_1,t_2)[\sigma])^{\mathcal{A}} = (t_1[\sigma],t_2[\sigma])^{\mathcal{A}} = (t_1[\sigma])^{\mathcal{A}} \otimes (t_2[\sigma])^{\mathcal{A}} \preccurlyeq (A_1[\sigma])^{\mathcal{A}} ⅋ (A_2[\sigma])^{\mathcal{A}}\,.$$

- **Case** ($\otimes\ \vdash$). Assume that we have:

$$\frac{c : \Gamma, x_1 : A_1, x_2 : A_2 \vdash \Delta}{\Gamma \mid \mu(x_1,x_2).c : A_1 \otimes A_2 \vdash \Delta}\ {\scriptstyle (\otimes \vdash)}$$

By induction hypothesis, we get that $(c[\sigma,x_1 \mapsto (A_1[\sigma])^{\mathcal{A}},x_2 \mapsto (A_2[\sigma])^{\mathcal{A}}])^{\mathcal{A}} \in \bot\!\!\!\bot$. Then by definition we have

$$((\mu(x_1,x_2).c)[\sigma])^{\mathcal{A}} = \curlyvee_{a,b \in \mathcal{A}} \{a ⅋ b : (c[\sigma,x_1 \mapsto a, x_2 \mapsto b])^{\mathcal{A}} \in \bot\!\!\!\bot\} \succcurlyeq (A_1[\sigma])^{\mathcal{A}} \otimes (A_2[\sigma])^{\mathcal{A}}.$$

- **Case** ($\exists \vdash$). Assume that we have:

$$\frac{\Gamma \mid e : A \vdash \Delta \quad X \notin FV(\Gamma,\Delta)}{\Gamma \mid e : \exists X.A \vdash \Delta}\ {\scriptstyle (\exists \vdash)}$$

By induction hypothesis, we have that for all $a \in \mathcal{A}$, $(e[\sigma])^{\mathcal{A}} \succcurlyeq ((A)[\sigma,x \mapsto a])^{\mathcal{A}}$. Therefore, we have that $(e[\sigma])^{\mathcal{A}} \succcurlyeq \curlyvee_{a\in\mathcal{A}}(A\{X := a\}[\sigma])^{\mathcal{A}}$.

- **Case** ($\vdash \exists$). Similarly, assume that we have:

$$\frac{\Gamma \vdash t : A\{X := B\} \mid \Delta}{\Gamma \vdash t : \exists X.A \mid \Delta}\ {\scriptstyle (\vdash \exists)}$$

By induction hypothesis, we have that $(t[\sigma])^{\mathcal{A}} \preccurlyeq (A[\sigma,X \mapsto (B[\sigma])^{\mathcal{A}}])^{\mathcal{A}}$. Therefore, we have that $(t[\sigma])^{\mathcal{A}} \preccurlyeq \curlyvee_{b\in\mathcal{A}}\{A\{X := b\}[\sigma]^{\mathcal{A}}\}$. □

12.2.5 Duality between conjunctive and disjunctive structures

We now show how disjunctive structures and conjunctive structures are connected by a form of duality. Per se, this connection only reflects the well-known duality between call-by-value and call-by-name [32]. In fact, the passage from one structure to the other exactly reflects the dual translation from the $\lambda\mu\tilde{\mu}$-calculus to itself [32, Section 7] which sends terms to contexts and vice-versa. This duality is also reflected in L [126] already in its syntax, in which the same constructors are used both for terms and contexts. Here, since the term t and the context e of a well-formed command are connected by $t^{\mathcal{A}} \preccurlyeq e^{\mathcal{A}}$, we materialize the duality by reversing the order relation. We know that reversing the order in a complete lattice yields a complete lattice in which meets and joins are exchanged (Proposition 9.5). Therefore, it only remains to prove that the axioms of disjunctive and conjunctive structures can be deduced through this duality one from each other.

12.2.5.1 From disjunctive to conjunctive structures

Let $(\mathcal{A}, \preccurlyeq, ⅋, \neg)$ be a disjunctive structure. We define:

$$\begin{array}{llll} \bullet\ \mathcal{A}^{\otimes} \triangleq \mathcal{A}^{⅋} & \bullet\ \curlywedge^{\otimes} \triangleq \curlyvee^{⅋} & \bullet\ a \otimes b \triangleq a ⅋ b & \\ \bullet\ a \triangleleft b \triangleq b \preccurlyeq a & \bullet\ \curlyvee^{\otimes} \triangleq \curlywedge^{⅋} & \bullet\ \ \neg a\ \ \triangleq \neg a & (\forall a, b \in \mathcal{A}) \end{array}$$

As expected, we have that:

Theorem• 12.24. *The structure $(\mathcal{A}^{\otimes}, \triangleleft, \otimes, \neg)$ defined above is a conjunctive structure.*

Proof. We check that for all $a, a', b, b' \in \mathcal{A}$ and for all subsets $A \subseteq \mathcal{A}$, we have:

1.• If $a \triangleleft a'$ then $\neg a' \triangleleft \neg a$ (Variance)
2.• If $a \triangleleft a'$ and $b \triangleleft b'$ then $a \otimes b \triangleleft a' \otimes b'$. (Variance)
3.• $(\curlywedge^{\otimes}_{a\in A} a) \otimes b = \curlywedge^{\otimes}_{a\in A}(a \otimes b)$ and $b \otimes (\curlywedge^{\otimes}_{a\in A} a) = \curlywedge^{\otimes}_{a\in A}(b \otimes a)$ (Distributivity)
4.• $\neg(\curlyvee^{\otimes}_{a\in A} a) = \curlywedge^{\otimes}_{a\in A}(\neg a)$ (Commutation)

All the proof are trivial from the corresponding properties of disjunctive structures. □

12.2.5.2 From conjunctive to disjunctive structures

Let $(\mathcal{A}, \preccurlyeq, \otimes, \neg)$ be a conjunctive structure. We define:

$$\begin{array}{llll} \bullet\ \mathcal{A}^{⅋} \triangleq \mathcal{A}^{\otimes} & \bullet\ \curlywedge^{⅋} \triangleq \curlyvee^{\otimes} & \bullet\ a ⅋ b \triangleq a \otimes b & \\ \bullet\ a \triangleleft b \triangleq b \preccurlyeq a & \bullet\ \curlyvee^{⅋} \triangleq \curlywedge^{\otimes} & \bullet\ \ \neg a\ \ \triangleq \neg a & (\forall a, b \in \mathcal{A}) \end{array}$$

Again, we have that:

Theorem• 12.25. *The structure $(\mathcal{A}^{\otimes}, \triangleleft, \otimes, \neg)$ defined above is a conjunctive structure.*

Proof. We check that for all $a, a', b, b' \in \mathcal{A}$ and for all subsets $A \subseteq \mathcal{A}$, we have:

1.• If $a \triangleleft a'$ then $\neg a' \triangleleft \neg a$. (Variance)
2.• If $a \triangleleft a'$ and $b \triangleleft b'$ then $a ⅋ b \triangleleft a' ⅋ b'$. (Variance)
3.• $(\curlywedge^{⅋}_{a\in A} a) ⅋ b = \curlywedge^{⅋}_{a\in A}(a ⅋ b)$ and $a ⅋ (\curlywedge^{⅋}_{b\in B} b) = \curlywedge^{⅋}_{b\in B}(a ⅋ b)$ (Distributivity)
4.• $\neg(\curlywedge^{⅋} A) = \curlyvee^{⅋}_{a\in A}(\neg a)$ (Commutation)

All the proof are trivial from the corresponding properties of conjunctive structures. □

12.3 Conjunctive algebras

12.3.1 Separation in conjunctive structures

We shall now define the notion of separator for conjunctive structures. To this end, we consider axioms (*i.e.* combinators) which correspond to the dual properties axiomatizing the disjunction $\parr$ in disjunctive algebras. Remember that in a conjunctive structure, the arrow is defined:

$$a \overset{\otimes}{\rightarrow} b \triangleq \neg(a \otimes \neg b) \qquad\qquad (\forall a, b \in \mathcal{A})$$

We thus define the following combinators:

$$\begin{array}{lcl} \mathbf{s}_1^{\otimes} & \triangleq & \curlywedge_{a \in \mathcal{A}} \, [a \overset{\otimes}{\rightarrow} (a \otimes a)] \\ \mathbf{s}_2^{\otimes} & \triangleq & \curlywedge_{a,b \in \mathcal{A}} \, [(a \otimes b) \overset{\otimes}{\rightarrow} a] \\ \mathbf{s}_3^{\otimes} & \triangleq & \curlywedge_{a,b \in \mathcal{A}} \, [(a \otimes b) \overset{\otimes}{\rightarrow} (b \otimes a)] \\ \mathbf{s}_4^{\otimes} & \triangleq & \curlywedge_{a,b,c \in \mathcal{A}} \, [(a \overset{\otimes}{\rightarrow} b) \overset{\otimes}{\rightarrow} (c \otimes a) \overset{\otimes}{\rightarrow} (c \otimes b)] \\ \mathbf{s}_5^{\otimes} & \triangleq & \curlywedge_{a,b,c \in \mathcal{A}} \, [(a \otimes (b \otimes c)) \overset{\otimes}{\rightarrow} ((a \otimes b) \otimes c))] \end{array}$$

which leads us to the expected definition of a separator:

Definition• 12.26 (Separator)**.** Given a conjunctive algebra $(\mathcal{A}, \preccurlyeq, \otimes, \neg)$, we call separator for $\mathcal{A}$ any subset $\mathcal{S} \subseteq \mathcal{A}$ that fulfills the following conditions for all $a, b \in \mathcal{A}$:

(1) If $a \in \mathcal{S}$ and $a \preccurlyeq b$ then $b \in \mathcal{S}$ (upward closure)

(2) $\mathbf{s}_1^{\otimes}, \mathbf{s}_2^{\otimes}, \mathbf{s}_3^{\otimes}, \mathbf{s}_4^{\otimes}$ and $\mathbf{s}_5^{\otimes}$ are in $\mathcal{S}$ (combinators)

(3) If $a \overset{\otimes}{\rightarrow} b \in \mathcal{S}$ and $a \in \mathcal{S}$ then $b \in \mathcal{S}$ (closure under modus ponens)

A separator $\mathcal{S}$ is said to be *consistent* if $\bot \notin \mathcal{S}$. ⌟

Example• 12.27 (Complete Boolean algebras)**.** Once again, if $\mathcal{B}$ is a complete Boolean algebra, $\mathcal{B}$ induces a disjunctive structure in which it is easy to verify that the combinators $\mathbf{s}_1^{\otimes}, \mathbf{s}_3^{\otimes}, \mathbf{s}_3^{\otimes}, \mathbf{s}_4^{\otimes}$ and $\mathbf{s}_5^{\otimes}$ are equal to $\top$. Therefore, the singleton $\{\top\}$ or any filter for $\mathcal{B}$ are valid separators for the induced conjunctive structure. ⌟

12.3.2 Conjunctive algebra from classical realizability

Remember that any model of classical realizability based on $\mathrm{L}^{\otimes}$ induces a conjunctive structure, where:

- $\mathcal{A} \triangleq \mathcal{P}(\mathcal{V}_0)$
- $a \preccurlyeq b \triangleq a \subseteq b$
- $a \otimes b \triangleq (a, b) = \{(V_1, V_2) : V_1 \in a \wedge V_2 \in b\}$
- $\neg a \triangleq [a^{\perp\!\!\!\perp}] = \{[e] : e \in a^{\perp\!\!\!\perp}\}$

$(\forall a, b \in \mathcal{A})$

As in the implicative and disjunctive cases, the set of formulas realized by a closed term[1], that is to say:

$$\mathcal{S}_{\perp\!\!\!\perp} \triangleq \{a \in \mathcal{P}(V_0^+) : a^{\perp\!\!\!\perp\perp\!\!\!\perp} \cap \mathcal{T}_0 \neq \emptyset\}$$

defines a valid separator. The condition (1) and (3) are clearly verified (for the same reasons as in the disjunctive and implicative cases), but we should verify that the formulas corresponding to the combinators are indeed realized. Let us then consider the following closed terms:

$$\begin{array}{lcl} TS_1 & \triangleq & \lambda a.(a, a) \\ TS_2 & \triangleq & \lambda(a, b).a \\ TS_3 & \triangleq & \lambda(a, b).(b.a) \end{array} \qquad \begin{array}{lcl} TS_4 & \triangleq & \lambda f.(\lambda(c, a).(c, f a)) \\ TS_5 & \triangleq & \lambda(a, (b, c)).((a, b), c \end{array}$$

[1] As in the $\lambda\mu\tilde{\mu}$-calculus (see Section 4.4.5) and $\mathrm{L}^{\parr}$, proof-like terms in $\mathrm{L}^{\otimes}$ simply correspond to closed terms.

where we use the shorthands:

$$\begin{array}{rcl}\lambda x.t & \triangleq & [\mu(x,[\alpha]).\langle t \| \alpha\rangle] \\ \lambda(a,b).t & \triangleq & \lambda x.\mu\alpha.\langle x \| \mu(a,b).\langle t\|\alpha\rangle\rangle \\ \lambda(a,(b,c)).t & \triangleq & \lambda(a,x).\mu\alpha.\langle x\|\mu(b,c).\langle t\|\alpha\rangle\rangle\end{array}$$

To show that these terms indeed realize the expected formulas, we need to introduce the additional rule for the universal quantifier and to give its realizability interpretation:

$$\frac{\Gamma \vdash V : A \mid \Delta \quad X \notin FV(\Gamma,\Delta)}{\Gamma \vdash V : \forall X.A}\ (\vdash\forall) \qquad \frac{\Gamma \vdash e : A[B/X] \mid \Delta}{\Gamma \mid e : \forall X.A \vdash \Delta}\ (\forall\vdash) \qquad |\forall X.A|_V \triangleq \textstyle\bigcap_{S\subseteq\mathcal{P}(\mathcal{V}_0)} |A\{X := \dot{S}\}|_V$$

Lemma 12.28. *The typing rules above are adequate with respect to the realizability interpretation of $L^{\otimes}$.*

Proof. The proof, which relies on the value restriction for the right rule, is the same as for L or $L^{\parr}$. □

Proposition 12.29. *The previous terms have the following types in $L^{\parr}$:*

1. $\vdash TS_1 : \forall A.A \to (A\otimes A) \mid$
2. $\vdash TS_2 : \forall AB.(A\otimes B) \to A \mid$
3. $\vdash TS_3 : \forall AB.A\otimes B \to B\otimes A \mid$
4. $\vdash TS_4 : \forall ABC.(A \to B) \to (C\otimes A \to C\otimes B) \mid$
5. $\vdash TS_5 : \forall ABC.(A\otimes(B\otimes C)) \to ((A\otimes B)\otimes C) \mid$

Proof. Straightforward typing derivations in $L^{\otimes}$. □

We deduce that $\mathcal{S}_{\perp\!\!\perp}$ is a valid separator for the conjunctive structure, and thus that any realizability model based on $L^{\otimes}$ induces a conjunctive algebra:

Proposition 12.30. *The quintuple $(\mathcal{P}(\mathcal{V}_0),\preccurlyeq,\otimes,\neg,\mathcal{S}_{\perp\!\!\perp})$ as defined above is a conjunctive algebra.*

Proof. Conditions (1) and (3) are trivial. Condition (2) follows from the previous propositions and the adequacy of the realizability interpretation of $L^{\otimes}$, observing that by definition of the conjunctive structure, we have $|\forall X.A|_V = \curlywedge_{a\in\mathcal{A}} |A\{X := \dot{a}\}|_V$. □

12.3.3 From disjunctive to conjunctive algebras

We shall now prove that any disjunctive algebra induces by duality a conjunctive algebra, using the construction we presented before to obtain a conjunctive structure from the underlying disjunctive structures. The key of this construction was to consider the reversed lattice, inversing thus meets and joins:

$$\begin{array}{llll}\bullet\ \mathcal{A}^{\otimes} \triangleq \mathcal{A}^{\parr} & \bullet\ \curlywedge^{\otimes} \triangleq \curlyvee^{\parr} & \bullet\ a\otimes b \triangleq a \parr b & \\ \bullet\ a \lessdot b \triangleq b \preccurlyeq a & \bullet\ \curlyvee^{\otimes} \triangleq \curlywedge^{\parr} & \bullet\ \neg a \triangleq \neg a & (\forall a,b\in\mathcal{A})\end{array}$$

Since both structures have the same career and disjunction, we will adopt the following notation to distinguish the conjunctive and disjunctive arrows:

$$a \overset{\parr}{\to} b \triangleq \neg a \parr b \qquad\qquad a \overset{\otimes}{\to} b \triangleq \neg(a\otimes\neg b) \qquad\qquad (\forall a,b\in\mathcal{A})$$

The question is now to determine, given a separator $\mathcal{S}^{\parr}$ for the disjunctive structure, how to define a separator $\mathcal{S}^{\otimes}$ for the conjunctive structure. Since separator are upwards closed and the lattice underlying the disjunctive structure is reversed in the conjunctive one, we should consider a set which is downward closed with respect to the order $\preccurlyeq$. To this purpose, we use the only contravariant operation we have at hands, and we define $\mathcal{S}^{\otimes}$ as the pre-image of $\mathcal{S}^{\parr}$ through the negation:

$$\mathcal{S}^{\otimes} \ \triangleq\ \neg^{-1}(\mathcal{S}^{\parr}) = \{a\in\mathcal{A} : \neg a \in \mathcal{S}^{\parr}\}$$

By definition, we thus have the following lemma:

Lemma• 12.31. *For all $a \in \mathcal{A}$, $a \in \mathcal{S}^{\otimes}$ if and only if $\neg a \in \mathcal{S}^{\parr}$.*

Besides, it is easy to show that the so-defined $\mathcal{S}^{\otimes}$ is indeed upward closed with respect to the reversed order:

Lemma• 12.32. *For all $a, b \in \mathcal{A}$, if $a \triangleleft b$ and $a \in \mathcal{S}^{\otimes}$ then $b \in \mathcal{S}^{\otimes}$.*

Proof. Straightforward: if $a \triangleleft b$ and $a \in \mathcal{S}^{\otimes}$, then $\neg a \in \mathcal{S}^{\parr}$ and $\neg a \preccurlyeq \neg b$, thus $\neg b \in \mathcal{S}^{\parr}$ and $b \in \mathcal{S}^{\otimes}$. □

Therefore, it remains to prove that $\mathcal{S}^{\otimes}$ contains the expected combinators, and that it is closed under modus ponens. For both proofs, the following proposition is fundamental:

Proposition• 12.33 (Contraposition)**.** *For all $a, b \in \mathcal{A}$, we have:*

$$a \overset{\otimes}{\rightarrow} b \in \mathcal{S}^{\otimes} \quad \Leftrightarrow \quad \neg a \overset{\parr}{\rightarrow} \neg b \in \mathcal{S}^{\parr}$$

Proof. Let $a, b \in \mathcal{A}$ be fixed. We do the proof directly by equivalence, since all the required equivalences hold for disjunctive algebras:

$$\begin{array}{rcll} a \overset{\otimes}{\rightarrow} b \in \mathcal{S}^{\otimes} & \Leftrightarrow & \neg(a \otimes \neg b) \in \mathcal{S}^{\otimes} & \text{(by definition)} \\ & \Leftrightarrow & \neg\neg(a \parr \neg b) \in \mathcal{S}^{\parr} & \text{(by definition)} \\ & \Leftrightarrow & (a \parr \neg b) \in \mathcal{S}^{\parr} & \text{(by DNE + Modus ponens)} \\ & \Leftrightarrow & (\neg\neg a \parr \neg b) \in \mathcal{S}^{\parr} & \text{(by DNI + } \parr\text{-compatible)} \\ & \Leftrightarrow & \neg a \overset{\parr}{\rightarrow} \neg b \in \mathcal{S}^{\parr} & \text{(by definition)} \end{array}$$

where DNE and DNI refer to the elimination and introduction of double negation (Proposition 11.58). The $\parr$-compatibility refers to the possibility of applying arrows of the shape $(a \rightarrow b) \in \mathcal{S}^{\parr}$ to get $(b \parr c) \in \mathcal{S}^{\parr}$ from $(a \parr c) \in \mathcal{S}^{\parr}$ (by application of $s_4^{\parr}$). The detailed proof is given in the Coq development. □

In particular, we can now deduce that $\mathcal{S}^{\otimes}$ is closed under modus ponens. The proof is straightforward from the previous lemma and Lemma 12.31.

Corollary• 12.34 (Modus Ponens)**.** *For all $a, b \in \mathcal{A}$, if $a \in \mathcal{S}^{\otimes}$ and $a \overset{\otimes}{\rightarrow} b \in \mathcal{S}^{\otimes}$, then $b \in \mathcal{S}^{\otimes}$.*

We now prove that $s_1^{\otimes}$, $s_1^{\otimes}$, $s_1^{\otimes}$, $s_1^{\otimes}$ and $s_1^{\otimes}$ belong to $\mathcal{S}^{\otimes}$. In each case, the proof somewhat consists in using the previous lemmas to be able to make use of the fact the dual combinator which is in $\mathcal{S}^{\parr}$.

Proposition• 12.35 ($s_1^{\otimes}$)**.** $s_1^{\otimes} \in \mathcal{S}^{\otimes}$

Proof. We want to show that $s_1^{\otimes} = \curlywedge^{\otimes}_{a \in \mathcal{A}} a \overset{\otimes}{\rightarrow} a \otimes a$ is in $\mathcal{S}^{\otimes}$. By definition of $\overset{\otimes}{\rightarrow}$ and commutation of the negation, we have $s_1^{\otimes} = \curlywedge^{\otimes}_{a \in \mathcal{A}} \neg(a \otimes \neg(a \otimes a)) = \neg \curlyvee^{\otimes}_{a \in \mathcal{A}}(a \otimes \neg(a \otimes a))$. To prove that the former is in the store, it suffices to prove that:

$$\neg\neg \mathop{\curlyvee}^{\otimes}_{a \in \mathcal{A}} (a \parr \neg(a \otimes a)) \in \mathcal{S}^{\parr} \qquad \textit{i.e.} \qquad \neg\neg \mathop{\curlywedge}^{\parr}_{a \in \mathcal{A}} (a \parr \neg(a \parr a)) \in \mathcal{S}^{\parr}$$

We conclude by double negation introduction (Proposition 11.58) and generalized modus ponens (Lemma 11.54) with $s_3^{\parr}$ and $s_1^{\parr}$. □

Proposition• 12.36 ($s_2^{\otimes}$)**.** $s_2^{\otimes} \in \mathcal{S}^{\otimes}$

Proof. We want to show that $s_2^{\otimes} = \curlywedge_{a,b \in \mathcal{A}} (a \otimes b) \overset{\otimes}{\rightarrow} a$ is in $\mathcal{S}^{\otimes}$. By definition of $\overset{\otimes}{\rightarrow}$ and commutation of the negation, we have $s_2^{\otimes} = \curlywedge_{a,b \in \mathcal{A}} \neg((a \otimes b) \otimes \neg a) = \neg \curlyvee^{\otimes}_{a,b \in \mathcal{A}}((a \otimes b) \otimes \neg a)$. To prove that the former is in the store, it suffices to prove that:

$$\neg\neg \mathop{\curlyvee}^{\otimes}_{a \in \mathcal{A}} ((a \otimes b) \otimes \neg a) \qquad \textit{i.e.} \qquad \neg\neg \mathop{\curlywedge}^{\parr}_{a \in \mathcal{A}} ((a \otimes b) \otimes \neg a)$$

We conclude by double negation introduction (Proposition 11.58) and generalized modus ponens (Lemma 11.54) with $s_3^{\parr}$ and $s_2^{\parr}$. □

The three other proofs for $s_3^{\otimes}$, $s_4^{\otimes}$ and $s_5^{\otimes}$ are identical and left to the reader.

Proposition• 12.37. $s_3^{\otimes} \in \mathcal{S}^{\otimes}$

Proposition• 12.38. $s_4^{\otimes} \in \mathcal{S}^{\otimes}$

Proposition• 12.39. $s_5^{\otimes} \in \mathcal{S}^{\otimes}$

We can thus conclude that $\mathcal{S}^{\otimes}$ is indeed a separator for the conjunctive structure, or, in other words:

Theorem• 12.40. *The quintuple* $(\mathcal{A}^{\otimes}, \triangleleft, \otimes, \neg, \mathcal{S}^{\otimes})$ *defines a conjunctive algebra.*

12.4 Conclusion

12.4.1 On conjunctive algebras

First, we should say is that we are still missing many things in the understanding of conjunctive algebras. In particular, as such we are not able to prove the converse direction, that is that disjunctive algebras can be obtained from conjunctive algebras by duality. Neither are we in the position of defining a conjunctive tripos to study its connection with the implicative and disjunctive cases. The main reason for this is that in conjunctive structures, the application induced by the λ-calculus does not satisfy[2] the usual adjunction:

$$a \preccurlyeq b \to c \quad \Leftrightarrow \quad ab \preccurlyeq c$$

This property being crucial in most of the proofs we presented for implicative and disjunctive algebras, we are not able to follow the same track. In particular, the adjunction is central in the definition of the induced Heyting algebra (thus of the induced tripos).

In fact, the absence of this property is in itself a reassuring fact. Indeed, one of the lesson we learned from the $\lambda\mu\tilde{\mu}$-calculus, is that through the duality of computation, on the side of terms, the call-by-name evaluation strategy computes as the call-by-value evaluation strategy does on the side of contexts, and vice-versa. Therefore, it is not that surprising that the application (on the side of terms) does not satisfy the same properties in disjunctive and conjunctive structures. Actually, we can say more, namely that in a structure with all commutations (of the connectives with meets and joins), the adjunction holds[3]. But again, such a structure can only induce triposes[4] which are necessarily isomorphic to forcing triposes. As such, it is thus a feature for conjunctive structures not to satisfy the (call-by-name) adjunction.

We did not have the time to explore this question much in depth, but at first sight, it reminds us of the situation in Streicher's AKSs or Ferrer *et al.* $^{\mathcal{K}}$OCAs, where an adjunctor is needed for the equivalence to holds. In these particular settings, the problem is due to the fact that (call-by-name) falsity values are restricted to those which are closed under bi-orthogonality. It is worth notice that one of the usual interest of considering this particular shape of falsity values is related to value restriction (see [126] for a discussion on the topic). While we saw how to circumvent this difficulty in implicative and disjunctive structures, it might be the case that it is unavoidable in a call-by-value fashion. Anyway, if the necessity of an adjunctor has the downside of complicating proofs, it does not prevent from inducing triposes. Therefore, this could be on solution to obtain a notion of conjunctive tripos. Another solution may consist in defining another application for which the adjunction holds. To this purpose, one track to follow could be to observe the behavior of the usual application (in disjunctive structure) on elements of the conjunctive through the embedding given in Section 12.2.5.2.

[2] The left to right implication is trivially satisfied, the not satisfied implication it the right to left one.

[3] This only a sufficient condition, but we conjecture having extra-commutations to obtain the adjunction is also necessary.

[4] To be precise, since we were not able to define conjunctive triposes, we should rather say that a conjunctive structure with all the commutations would probably induce disjunctive structures with the same commutations. These disjunctive structures would only induce triposes isomorphic to forcing triposes. Yet, we believe that in the case where a canonical notion conjunctive triposes could be defined, the very same would happen.

12.4.2 On the algebraization of Krivine classical realizability

In the last three chapters, we have shown that the underlying algebraic structures of classical realizability can be reified into algebras whose structures depend on the choice of logical connectives. Realizability models based on the λ_c-calculus, whose type system is defined with an arrow as logical connective, are particular instances of implicative algebras; models based on $\mathrm{L}^{\parr}$, whose type system is defined with a disjunction and a negation as logical connectives, are particular cases of disjunctive algebras; models based on $\mathrm{L}^{\otimes}$, whose connectives are a conjunction and a negation, are particular cases of conjunctive algebras. We highlighted the fact that the choice of connective (and therefore the corresponding algebraic structure) was related to the choice of a strategy of evaluation for the λ-calculus: call-by-name naturally corresponds to implicative and disjunctive algebras, while conjunctive algebras canonically embodies a call-by-value λ-calculus.

In the continuity of classical realizability, one of the main features of these algebraic structures is to give different semantics to the logical connectives $\wedge, \vee$ and to the quantifiers. For instance, the conjunction $a \wedge b$ is interpreted by the product type $a \times b$ in implicative algebras; whereas the universal quantification $\forall X.A(X)$ is interpreted by a meet $\bigcurlywedge_{b\in\mathcal{A}} A(b)$. This distinction between both interpretations leaves the door open to the definition of triposes that reflect Krivine realizability models [98, 99, 100, 101]. In particular, these models are more general than the models one can obtain by means of a forcing construction. It is worth noting that in the construction of realizability triposes from an implicative algebra $\mathcal{A}$, the structure of Heyting algebra which is obtained through the quotient $(\mathcal{A}^I/\mathcal{S}[I], \vdash_{\mathcal{S}[I]})$ (and therefore, the hyperdoctrine and the tripos) ignores the former order relation $\preccurlyeq$ and the former meets and joins $\curlywedge, \curlyvee$. More, whenever the underlying algebraic structure $\mathcal{A}$ has too many commutation properties, then the connective $\times$ (resp $+$) becomes equivalent to $\wedge$ (resp $\vee$). As a consequence, everything happens as if they were the same in $\mathcal{A}$, that is as if $\mathcal{A}$ were a Boolean algebra: the induced tripos is isomorphic to a forcing tripos. Schematically, the situation can be summed up by the following diagram[5]:

$$\begin{tikzcd}
\begin{array}{c}\textbf{Implicative algebras}\\ \begin{array}{c|c|c|c} \Rightarrow & \wedge & \forall & \mathcal{M}\vDash \_ \\ \hline \to & \times & \bigcurlywedge & \_\in\mathcal{S}\end{array}\end{array}
& \begin{array}{c}\textbf{Disjunctive algebras}\\ \begin{array}{c|c|c|c} \neg & \vee & \forall & \mathcal{M}\vDash \_ \\ \hline \neg & \parr & \bigcurlywedge & \_\in\mathcal{S}^{\parr}\end{array}\end{array} \arrow[l, bend right, "a\to b=\neg a\parr b"'] \arrow[r, dashed, bend left, "a \triangleleft b = b \preccurlyeq a,\ \mathcal{S}^{\otimes}=\neg^{-1}(\mathcal{S}^{\parr})"]
& \begin{array}{c}\textbf{Conjunctive algebras}\\ \begin{array}{c|c|c|c} \neg & \wedge & \exists & \mathcal{M}\vDash \_ \\ \hline \neg & \otimes & \bigcurlyvee & \_\in\mathcal{S}^{\otimes}\end{array}\end{array} \\
& \begin{array}{c}\textbf{Boolean algebras}\\ \begin{array}{c|c|c|c} \neg & \wedge & \forall & \mathcal{M}\vDash \_ \\ \hline \neg & \curlywedge & \bigcurlywedge & \_\in\mathcal{F}\end{array}\end{array} \arrow[ul, "a\to b=\neg a\curlyvee b"] \arrow[u, "\parr=\curlyvee"'] \arrow[ur, "\otimes=\curlywedge"'] &
\end{tikzcd}$$

In this diagram, plain arrows $A \to B$ indicate that the structure A is a particular case of B, while the dashed one $A \dashrightarrow B$ means that B can be obtained from A through a construction. We annotate the arrow with the key definitions in the passage from one structure to another.

As we explained in Chapters 10 and 11, the left part of this diagram can be reflected at the level of the induced triposes. Indeed, if a structure A is particular case of a class of structures B (*i.e.* for an arrow $A \to B$ above), then the tripos $\mathcal{T}_A$ that A induces is also a particular case of tripos $\mathcal{T}_B$: formally, this is reflected by a surjective map $\mathcal{T}_B(I) \twoheadrightarrow \mathcal{T}_A(I)$ for all $I \in \mathbf{Set}^{op}$ (see the diagram in Section 10.4.4.1).

Up to now, the conclusion from the last chapters is that implicative algebras appear as the more

[5] Where we write $\mathcal{M} \vDash \_$ to represent the criterion of validity and where $\mathcal{F}$ denotes a filter of Boolean algebra.

general setting. Nonetheless, we did not achieve yet a complete study of conjunctive algebras. In particular, we are lacking the definition of an application (from the point of view of λ-calculus) satisfying the adjunction necessary to obtain a Heyting algebra (and thus a tripos). Besides, we are also missing an arrow in the previous diagram, from conjunctive to disjunctive algebra. We conjecture that there should be a way to prove that from any conjunctive algebra can be obtained a disjunctive algebra through the same duality, that is by reversing the order (see Section 12.2.5.2) and taking as (disjunctive) separator the preimage $\neg^{-1}(\mathcal{S}^{\otimes})$ of the (conjunctive) separator. In particular, we believe that the induced triposes should be proved to be isomorphic. In addition to giving a proof to support the claim that implicative algebras provide us with the more general framework, such a result would have a particular significance, showing that call-by-name and call-by-value calculi induce equivalent realizability models.

In a long-term perspective, several directions of investigation emerge. First, implicative algebras appear as a promising new tool from a model-theoretic point of view. They indeed provide us with a framework whose ground structure is as simple as Boolean algebras, while carrying all the computational power of the λ-calculus. In particular, they seem easier to manipulate than Krivine's realizability algebras while providing us with the same expressiveness. Since Krivine's realizability models seem to bring novel possibilities with respect to the traditional models of set theory, implicative algebras might be the more convenient structure to develop the model-theoretic analysis of classical realizability.

Second, we saw that implicative algebras identify types and programs, somewhat performing the last step of unification in the proofs-as-programs correspondence. As such, implicative algebras are tailored to the second-order λ_c-calculus, that is to say the second-order classical logic, but they clearly scale to high-order classical logic. On its computational facet, following the leitmotiv of the second part of this thesis, it raises the question of extending the calculus with side-effects. For instance, we wonder how our interpretations for the (call-by-need) $\overline{\lambda}_{[lv\tau\star]}$-calculus or—which is more ambitious—for dLPA$^{\omega}$ may be interpreted algebraically. In particular, an interpretation of dLPA$^{\omega}$ in terms of implicative algebras might help us to answer the questions we raised in Section 8.5 about the structure of the induced model. Especially, we could hope to take advantage of the criteria of collapsing so as to determine whether dLPA$^{\omega}$ allows for realizability models which are not equivalent to forcing constructions.

Furthermore, in the continuity of the study of disjunctive and conjunctive algebras, it would be interesting to determine how much of these structures can be combined without collapsing to a forcing situation. To put it differently, we saw that an implicative (resp. disjunctive) algebra in which arbitrary meets and joins distribute over all the connectives can only induce a tripos which is isomorphic to a forcing tripos. Yet, it is not clear whether it is possible to define an algebra which is both disjunctive and conjunctive without collapsing to a Boolean algebra. Such a structure would make sense to model the call-by-push-value paradigm [109], whose evaluation is directed by the polarity of terms (and thus requires a syntax with connectives of both polarities). Among other things, call-by-push value has shown to be a conducing setting for the study of side-effects in the realm of the proofs-as-programs correspondence.

Last, all along this manuscript we have been using several times Krivine realizability as a tool to prove properties for different calculi. Even if this perspective is at first sight fuzzier than the previous ones, it could be interesting to determine whether the reasoning process—*i.e.* defining a realizability interpretation and proving its adequacy in order to finally deduce theorems (mainly normalization and consistence properties)—can be transposed algebraically. In other words, we wonder whether, given a calculus, one could hope to define an embedding of this given calculus into an implicative algebra, next prove the adequacy of the embedding; then consider, for instance, the "separator" of normalizing terms to prove the normalization of the calculus. In itself, such an approach would probably be very closed from the usual one, but having a unifying framework might bring us some benefits.

For all these reasons, I am convinced that implicative algebras have a bright future ahead. We hope that this thesis would have done its bit towards a broader diffusion of their potentialities and features. *I have a dream that one day, we will all compute with formulas as if they were λ-terms...*

Bibliography

[1] D. Ahman, N. Ghani, and G. D. Plotkin. *Dependent Types and Fibred Computational Effects*, pages 36–54. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. URL: `http://dx.doi.org/10.1007/978-3-662-49630-5_3`, `doi:10.1007/978-3-662-49630-5_3`.

[2] S. Amini. *Logique classique et calcul.* PhD thesis, Université Paris-Diderot, 2015.

[3] Z. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7(3):265–301, 1993. `doi:10.1017/S0956796897002724`.

[4] Z. M. Ariola, P. Downen, H. Herbelin, K. Nakata, and A. Saurin. Classical call-by-need sequent calculi: The unity of semantic artifacts. In T. Schrijvers and P. Thiemann, editors, *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings*, Lecture Notes in Computer Science, pages 32–46. Springer, 2012. `doi:10.1007/978-3-642-29822-6`.

[5] Z. M. Ariola, H. Herbelin, and A. Sabry. A type-theoretic foundation of delimited continuations. *Higher-Order and Symbolic Computation*, 22(3):233–273, 2009. URL: `http://dx.doi.org/10.1007/s10990-007-9006-0`, `doi:10.1007/s10990-007-9006-0`.

[6] Z. M. Ariola, H. Herbelin, and A. Saurin. Classical call-by-need and duality. In C.-H. L. Ong, editor, *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*, volume 6690 of *Lecture Notes in Computer Science*, pages 27–44. Springer, 2011. `doi:10.1007/978-3-642-21691-6\_6`.

[7] Z. M. Ariola, J. Maraist, M. Odersky, M. Felleisen, and P. Wadler. A call-by-need lambda calculus. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 233–246, New York, NY, USA, 1995. ACM. `doi:10.1145/199448.199507`.

[8] S. Awodey. *Category Theory.* Oxford Logic Guides. OUP Oxford, 2010. `doi:10.1093/acprof:oso/9780198568612.001.0001`.

[9] S. Banach and A. Tarski. Sur la décomposition des ensembles de points en parties respectivement congruentes. *Fundamenta Mathematicae*, 6(1):244–277, 1924.

[10] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and The Foundations of Mathematics.* North-Holland, 1984.

[11] H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991. `doi:10.1017/S0956796800020025`.

[12] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 2)*, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992. URL: `http://dl.acm.org/citation.cfm?id=162552.162561`.

[13] G. Barthe, J. Hatcliff, and M. H. B. Sørensen. CPS translations and applications: The cube and beyond. *Higher-Order and Symbolic Computation*, 12(2):125–170, 1999. URL: http://dx.doi.org/10.1023/A:1010000206149, doi:10.1023/A:1010000206149.

[14] J. L. Bell. *Set Theory: Boolean-Valued Models and Independence Proofs*. Oxford: Clarendon Press, 2005.

[15] S. Berardi, M. Bezem, and T. Coquand. On the computational content of the axiom of choice. *J. Symb. Log.*, 63(2):600–622, 1998. URL: http://dx.doi.org/10.2307/2586854, doi:10.2307/2586854.

[16] U. Berger. A computational interpretation of open induction. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*, page 326. IEEE Computer Society, 2004.

[17] V. Blot. Hybrid realizability for intuitionistic and classical choice. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 575–584, New York, NY, USA, 2016. ACM. doi:10.1145/2933575.2934511.

[18] V. Blot. An interpretation of system f through bar recursion. In *LICS 2017, Reijkavik, Iceland*, 2017.

[19] J. Bolyai. *The Science Absolute of Space: Independent of the Truth Or Falsity of Euclid's Axiom XI (which Can Never be Decided a Priori)...* Neomonic series. Neomon, 1896. URL: http://real-eod.mtak.hu/2790/.

[20] G. Boole. *Investigation of The Laws of Thought On Which Are Founded the Mathematical Theories of Logic and Probabilities.* 1853. URL: http://www.gutenberg.org/etext/15114.

[21] N. Bourbaki. *Éléments de mathématique. Théorie des ensembles.* Hermann, Paris, 1970. URL: http://opac.inria.fr/record=b1078957, doi:10.1007/978-3-540-34035-5.

[22] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. *An extension of system F with subtyping*, pages 750–770. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991. URL: http://dx.doi.org/10.1007/3-540-54415-1_73, doi:10.1007/3-540-54415-1_73.

[23] A. Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, Oct 2012. doi:10.1007/s10817-011-9225-2.

[24] A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932. doi:10.2307/1968337.

[25] A. Church. A note on the entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936. doi:10.2307/2269326.

[26] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[27] T. Coquand. A semantics of evidence for classical arithmetic. *Journal of Symbolic Logic*, 60(1):325–337, 1995. doi:10.2307/2275524.

[28] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2):95 – 120, 1988. URL: http://www.sciencedirect.com/science/article/pii/0890540188900053, doi:http://dx.doi.org/10.1016/0890-5401(88)90005-3.

[29] T. Coquand and C. Paulin. *Inductively defined types*, pages 50–66. Springer Berlin Heidelberg, Berlin, Heidelberg, 1990. URL: http://dx.doi.org/10.1007/3-540-52335-9_47, doi:10.1007/3-540-52335-9_47.

[30] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.

[31] T. Crolard. A confluent lambda-calculus with a catch/throw mechanism. *J. Funct. Program.*, 9(6):625–647, 1999. doi:10.1017/S0956796899003512.

[32] P.-L. Curien and H. Herbelin. The duality of computation. In *Proceedings of ICFP 2000*, SIGPLAN Notices 35(9), pages 233–243. ACM, 2000. doi:10.1145/351240.351262.

[33] H. B. Curry. Functionality in Combinatory Logic. *Proceedings of the National Academy of Science*, 20:584–590, Nov. 1934. doi:10.1073/pnas.20.11.584.

[34] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.

[35] P.-É. Dagand and G. Scherer. Normalization by realizability also evaluates. In D. Baelde and J. Alglave, editors, *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, Le Val d'Ajol, France, Jan. 2015. URL: https://hal.inria.fr/hal-01099138.

[36] N. Daniels. Thomas reid's discovery of a non-euclidean geometry. *Philosophy of Science*, 39(2):219–234, 1972. URL: http://www.jstor.org/stable/186723.

[37] O. Danvy, K. Millikin, J. Munk, and I. Zerny. *Defunctionalized Interpreters for Call-by-Need Evaluation*, pages 240–256. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. URL: http://dx.doi.org/10.1007/978-3-642-12251-4_18, doi:10.1007/978-3-642-12251-4_18.

[38] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972. doi:http://dx.doi.org/10.1016/1385-7258(72)90034-0.

[39] P. Downen, L. Maurer, Z. M. Ariola, and S. P. Jones. Sequent calculus as a compiler intermediate language. In *ICFP 2016*, 2016. URL: http://research.microsoft.com/en-us/um/people/simonpj/papers/sequent-core/scfp_ext.pdf.

[40] M. H. Escardó and P. Oliva. Bar recursion and products of selection functions. *CoRR*, abs/1407.7046, 2014. URL: http://arxiv.org/abs/1407.7046.

[41] M. Felleisen, D. P. Friedman, E. E. Kohlbecker, and B. F. Duba. Reasoning with continuations. In *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*, pages 131–141. IEEE Computer Society, 1986.

[42] G. Ferreira and P. Oliva. On various negative translations. In S. van Bakel, S. Berardi, and U. Berger, editors, *Proceedings Third International Workshop on Classical Logic and Computation, CL&C 2010, Brno, Czech Republic, 21-22 August 2010.*, volume 47 of *EPTCS*, pages 21–33, 2010. URL: http://dx.doi.org/10.4204/EPTCS.47.4, doi:10.4204/EPTCS.47.4.

[43] W. Ferrer Santos, J. Frey, M. Guillermo, O. Malherbe, and A. Miquel. Ordered combinatory algebras and realizability. *Mathematical Structures in Computer Science*, 27(3):428–458, 2017. doi:10.1017/S0960129515000432.

[44] W. Ferrer Santos, M. Guillermo, and O. Malherbe. A Report on Realizability. *ArXiv e-prints*, 2013. arXiv:1309.0706.

[45] W. Ferrer Santos, M. Guillermo, and O. Malherbe. Realizability in OCAs and AKSs. *ArXiv e-prints*, 2015. `arXiv:1512.07879`.

[46] A. Filinski. Representing monads. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457. ACM Press, 1994.

[47] M. Fitting. *Intuitionistic Logic, Model Theory and Forcing*. Amsterdam: North-Holland Pub. Co., 1969.

[48] G. Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Halle, 1879. URL: `http://gallica.bnf.fr/ark:/12148/bpt6k65658c/`.

[49] G. Frege. *Posthumous Writings*. Wiley-Blackwell, 1991.

[50] J. Frey. Realizability Toposes from Specifications. In T. Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*, volume 38 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 196–210, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: `http://drops.dagstuhl.de/opus/volltexte/2015/5164`, `doi:10.4230/LIPIcs.TLCA.2015.196`.

[51] J. Frey. Classical realizability in the cps target language. *Electronic Notes in Theoretical Computer Science*, 325(Supplement C):111 – 126, 2016. The Thirty-second Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXII). `doi:https://doi.org/10.1016/j.entcs.2016.09.034`.

[52] D. Fridlender and M. Pagano. Pure type systems with explicit substitutions. *J. Funct. Program.*, 25, 2015. URL: `http://dx.doi.org/10.1017/S0956796815000210`, `doi:10.1017/S0956796815000210`.

[53] H. Friedman. *Classically and intuitionistically provably recursive functions*, pages 21–27. Springer Berlin Heidelberg, Berlin, Heidelberg, 1978. URL: `http://dx.doi.org/10.1007/BFb0103100`, `doi:10.1007/BFb0103100`.

[54] C. Führmann. Direct models for the computational lambda calculus. *Electr. Notes Theor. Comput. Sci.*, 20:245–292, 1999. URL: `http://dx.doi.org/10.1016/S1571-0661(04)80078-1`, `doi:10.1016/S1571-0661(04)80078-1`.

[55] J. Garrigue. Relaxing the value restriction. In Y. Kameyama and P. J. Stuckey, editors, *Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004, Proceedings*, volume 2998 of *Lecture Notes in Computer Science*, pages 196–213. Springer, 2004. URL: `http://dx.doi.org/10.1007/978-3-540-24754-8_15`, `doi:10.1007/978-3-540-24754-8_15`.

[56] G. Gentzen. Untersuchungen über das logische schließen. i. *Mathematische Zeitschrift*, 39(1):176–210, 1935. URL: `http://dx.doi.org/10.1007/BF01201353`, `doi:10.1007/BF01201353`.

[57] G. Gentzen. Untersuchungen über das logische schließen. ii. *Mathematische Zeitschrift*, 39(1):405–431, 1935. URL: `http://dx.doi.org/10.1007/BF01201363`, `doi:10.1007/BF01201363`.

[58] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1 – 101, 1987. `doi:10.1016/0304-3975(87)90045-4`.

[59] J.-Y. Girard. A new constructive logic: classic logic. *Mathematical Structures in Computer Science*, 1(3):255–296, 1991. `doi:10.1017/S0960129500001328`.

[60] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[61] K. Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931. `doi:10.1007/BF01700692`.

[62] T. G. Griffin. A formulae-as-type notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 47–58, New York, NY, USA, 1990. ACM. URL: `http://doi.acm.org/10.1145/96709.96714`, `doi:10.1145/96709.96714`.

[63] M. Guillermo. *Jeux de réalisabilité en arithmétique classique.* PhD thesis, Université Paris 7, 2008.

[64] M. Guillermo and A. Miquel. Specifying peirce's law in classical realizability. *Mathematical Structures in Computer Science*, 26(7):1269–1303, 2016. `doi:10.1017/S0960129514000450`.

[65] M. Guillermo and Étienne Miquey. Classical realizability and arithmetical formulæ. *Mathematical Structures in Computer Science*, page 1–40, 2016. `doi:10.1017/S0960129515000559`.

[66] K. Gödel. *Consistency of the Continuum Hypothesis.* Princeton University Press, 1940.

[67] R. Harper and M. Lillibridge. Polymorphic type assignment and CPS conversion. *LISP and Symbolic Computation*, 6(3):361–379, 1993.

[68] H. Herbelin. *C'est maintenant qu'on calcule: au cœur de la dualité.* Habilitation thesis, University Paris 11, Dec. 2005.

[69] H. Herbelin. On the degeneracy of sigma-types in presence of computational classical logic. In P. Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 209–220. Springer, 2005. URL: `http://dx.doi.org/10.1007/11417170_16`, `doi:10.1007/11417170_16`.

[70] H. Herbelin. A constructive proof of dependent choice, compatible with classical logic. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 365–374. IEEE Computer Society, 2012. URL: `http://dx.doi.org/10.1109/LICS.2012.47`, `doi:10.1109/LICS.2012.47`.

[71] H. Herbelin and S. Ghilezan. An approach to call-by-name delimited continuations. In G. C. Necula and P. Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 383–394. ACM, Jan. 2008.

[72] A. Heyting. *Mathematische Grundlagenforschung. Intuitionismus. Beweistheorie.* Springer-Verlag, Berlin, 1934. `doi:10.1007/978-3-642-65617-0`.

[73] D. Hilbert. Mathematical problems. *Bulletin ot the American Mathematical Society*, 8(10):437–479, 1902. `doi:10.1090/S0002-9904-1902-00923-3`.

[74] D. Hilbert. Königsberg's radio address, September 1930. URL: `https://www.maa.org/book/export/html/326610`.

[75] D. Hilbert and W. Ackermann. *Das Entscheidungsproblem*, pages 119–131. Springer, Berlin, Heidelberg, 1928. `doi:10.1007/978-3-642-52789-0`.

[76] P. HOFSTRA and J. VAN OOSTEN. Ordered partial combinatory algebras. *Mathematical Proceedings of the Cambridge Philosophical Society*, 134(3):445–463, 2003. doi:10.1017/S0305004102006424.

[77] W. A. Howard. The formulae-as-types notion of construction. Privately circulated notes, 1969.

[78] J. Hyland. The effective topos. *Studies in Logic and the Foundations of Mathematics*, 110:165 – 216, 1982. The L. E. J. Brouwer Centenary Symposium. doi:10.1016/S0049-237X(09)70129-6.

[79] J. Hyland, P. Johnstone, and A. Pitts. Tripos theory. *Math. Proc. Camb. Phil. Soc.*, 88:205–232, 1980.

[80] G. Jaber. Krivine Realizability for Compiler Correctness. Master's thesis, Ecole des Mines de Nantes/LINA/INRIA, June 2010. URL: https://dumas.ccsd.cnrs.fr/dumas-00530710.

[81] G. Jaber, G. Lewertowski, P.-M. Pédrot, M. Sozeau, and N. Tabareau. The definitional side of the forcing. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 367–376, New York, NY, USA, 2016. ACM. doi:10.1145/2933575.2935320.

[82] G. Jaber, N. Tabareau, and M. Sozeau. Extending type theory with forcing. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, LICS '12, pages 395–404, Washington, DC, USA, 2012. IEEE Computer Society. doi:10.1109/LICS.2012.49.

[83] T. J. Jech. *The Axiom of Choice.* Studies in Logic. North-Holland Publishing Company, 1973.

[84] F. Joachimski and R. Matthes. Short proofs of normalization for the simply- typed λ-calculus, permutative conversions and gödel's t. *Archive for Mathematical Logic*, 42(1):59–87, 2003. URL: http://dx.doi.org/10.1007/s00153-002-0156-9, doi:10.1007/s00153-002-0156-9.

[85] P. Johnstone. *Sketches of an elephant: a Topos theory compendium.* Oxford logic guides. Oxford Univ. Press, New York, NY, 2002.

[86] D. Kesner. *Reasoning About Call-by-need by Means of Types*, pages 424–441. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. doi:10.1007/978-3-662-49630-5_25.

[87] S. C. Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, 10:109–124, 1945.

[88] A. Kolmogoroff. Zur deutung der intuitionistischen logik. *Mathematische Zeitschrift*, 35(1):58–65, Dec 1932. URL: http://dx.doi.org/10.1007/BF01186549, doi:10.1007/BF01186549.

[89] S. A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.

[90] S. A. Kripke. Semantical analysis of intuitionistic logic i. *Studies in Logic and the Foundations of Mathematics*, 40:92 – 130, 1965. doi:http://dx.doi.org/10.1016/S0049-237X(08)71685-9.

[91] J.-L. Krivine. Un algorithme non-typable dans le système f. *Comptes rendus de l'Académie des sciences. Série 1, Mathématiques*, 304(5):123–126, February 1987. URL: http://gallica.bnf.fr/ark:/12148/bpt6k57447206.

[92] J.-L. Krivine. *Lambda-calculus, types and models.* Masson, 1993.

[93] J.-L. Krivine. Typed lambda-calculus in classical Zermelo-Fraenkel set theory. *Arch. Math. Log.*, 40(3):189–205, 2001.

[94] J.-L. Krivine. Dependent choice, 'quote' and the clock. *Th. Comp. Sc.*, 308:259–276, 2003.

[95] J.-L. Krivine. A call-by-name lambda-calculus machine. In *Higher Order and Symbolic Computation*, 2004.

[96] J.-L. Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, Sep 2007. URL: `http://dx.doi.org/10.1007/s10990-007-9018-9`, `doi:10.1007/s10990-007-9018-9`.

[97] J.-L. Krivine. Realizability in classical logic. In interactive models of computation and program behaviour. *Panoramas et synthèses*, 27, 2009.

[98] J.-L. Krivine. Realizability algebras: a program to well order r. *Logical Methods in Computer Science*, 7(3), 2011.

[99] J.-L. Krivine. Realizability algebras II : new models of ZF + DC. *Logical Methods in Computer Science*, 8(1):10, Feb. 2012. 28 p.

[100] J.-L. Krivine. Quelques propriétés des modèles de réalisabilité de ZF, Feb. 2014. URL: `http://hal.archives-ouvertes.fr/hal-00940254`.

[101] J.-L. Krivine. On the Structure of Classical Realizability Models of ZF. In H. Herbelin, P. Letouzey, and M. Sozeau, editors, *20th International Conference on Types for Proofs and Programs (TYPES 2014)*, volume 39 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 146–161, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.TYPES.2014.146`.

[102] J.-L. Krivine. Bar Recursion in Classical Realisability: Dependent Choice and Continuum Hypothesis. In J.-M. Talbot and L. Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:11, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.CSL.2016.25`.

[103] Y. Lafont, B. Reus, and T. Streicher. Continuations semantics or expressing implication by negation. Technical Report 9321, Ludwig-Maximilians-Universität, München, 1993.

[104] F. Lang. Explaining the lazy krivine machine using explicit substitution and addresses. *Higher-Order and Symbolic Computation*, 20(3):257–270, Sep 2007. `doi:10.1007/s10990-007-9013-1`.

[105] F. Lawvere. Adjointness in foundations. *Dialectica*, 23:281–296, 1969.

[106] G. W. Leibniz. *Die philosophischen Schriften*, volume 7. Karl Immanuel Gerhardt, 1890. URL: `https://archive.org/details/diephilosophisc00gerhgoog`.

[107] G. W. Leibniz. The art of discovery (1685). In P. Wiener, editor, *Leibniz: Selections*, Lyceum editions: Philosophy series. 1951.

[108] R. Lepigre. A classical realizability model for a semantical value restriction. In P. Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 476–502. Springer, 2016.

[109] P. B. Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*, volume 2 of *Semantics Structures in Computation*. Springer, 2004. `doi:10.1007/978-94-007-0954-6`.

[110] N. Lobatchevsky. Géométrie imaginaire. *Journal für die reine und angewandte Mathematik*, pages 295–320, 1837.

[111] S. MacLane and I. Moerdijk. *Sheaves in Geometry and Logic.* Springer, 1992. doi:10.1007/978-1-4612-0927-0.

[112] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8:275–317, 1994.

[113] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *J. Funct. Program.*, 8(3):275–317, 1998. doi:10.1017/S0956796898003037.

[114] P. Martin-Löf. Constructive mathematics and computer programming. In *Proc. Of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*, pages 167–184, Upper Saddle River, NJ, USA, 1985. Prentice-Hall, Inc. URL: http://dl.acm.org/citation.cfm?id=3721.3731.

[115] P. Martin-Löf. An intuitionistic theory of types. In twenty-five years of constructive type theory. *Oxford Logic Guides*, 36:127–172, 1998.

[116] P.-A. Melliès. *Local States in String Diagrams*, pages 334–348. Springer International Publishing, Cham, 2014. doi:10.1007/978-3-319-08918-8_23.

[117] A. Miquel. Classical program extraction in the calculus of constructions. In *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2007.

[118] A. Miquel. Existential witness extraction in classical realizability and via a negative translation. *Logical Methods for Computer Science*, 2010.

[119] A. Miquel. Existential witness extraction in classical realizability and via a negative translation. *Logical Methods in Computer Science*, 7(2):188–202, 2011. URL: http://dx.doi.org/10.2168/LMCS-7(2:2)2011, doi:10.2168/LMCS-7(2:2)2011.

[120] A. Miquel. Forcing as a program transformation. In *LICS*, pages 197–206. IEEE Computer Society, 2011.

[121] A. Miquel. Implicative algebras: a new foundation for realizability and forcing, 2017. URL: https://www.p\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{e\global\mathchardef\accent@spacefactor\spacefactor}\accent19e\egroup\spacefactor\accent@spacefactordrot.fr/montevideo2016/miquel-slides.pdf.

[122] É. Miquey. Coq development on implicative algebras. URL: https://www.irif.fr/~emiquey/these/index.html#coqia.

[123] I. Moerdijk and J. van Oosten. Topos theory, 2007. URL: http://www.staff.science.uu.nl/~ooste110/syllabi/toposmoeder.pdf.

[124] E. Moggi. Computational lambda-calculus and monads. Technical Report ECS-LFCS-88-66, Edinburgh Univ., 1988. doi:10.1109/LICS.1989.39155.

[125] G. Muller. The axiom of choice is wrong. URL: https://cornellmath.wordpress.com/2007/09/13/the-axiom-of-choice-is-wrong/.

[126] G. Munch-Maccagnoni. Focalisation and Classical Realisability. In E. Grädel and R. Kahle, editors, *Computer Science Logic '09*, volume 5771 of *Lecture Notes in Computer Science*, pages 409–423. Springer, Heidelberg, 2009.

[127] G. Munch-Maccagnoni. *Models of a Non-associative Composition*, pages 396–410. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. URL: http://dx.doi.org/10.1007/978-3-642-54830-7_26, doi:10.1007/978-3-642-54830-7_26.

[128] C. Okasaki, P. Lee, and D. Tarditi. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, 7(1):57–82, 1994. doi:10.1007/BF01019945.

[129] P. Oliva and T. Streicher. On Krivine's realizability interpretation of classical second-order arithmetic. *Fundam. Inform.*, 84(2):207–220, 2008.

[130] M. Parigot. Free deduction: An analysis of "computations" in classical logic. In A. Voronkov, editor, *Proceedings of LPAR*, volume 592 of *LNCS*, pages 361–380. Springer, 1991. URL: http://dx.doi.org/10.1007/3-540-55460-2_27.

[131] M. Parigot. Proofs of strong normalisation for second order classical natural deduction. *J. Symb. Log.*, 62(4):1461–1479, 1997.

[132] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the calculus of constructions. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 89–104, New York, NY, USA, 1989. ACM. doi:10.1145/75277.75285.

[133] P.-M. Pédrot. *A Materialist Dialectica.* Theses, Paris Diderot, Sept. 2015. URL: https://hal.archives-ouvertes.fr/tel-01247085.

[134] P.-M. Pédrot and A. Saurin. *Classical By-Need*, pages 616–643. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. URL: http://dx.doi.org/10.1007/978-3-662-49498-1_24, doi:10.1007/978-3-662-49498-1_24.

[135] A. Pitts. *The Theory of Triposes.* PhD thesis, Cambridge University, 1981.

[136] A. M. Pitts. Tripos theory in retrospect. *Mathematical Structures in Computer Science*, 12(3):265–279, 2002. doi:10.1017/S096012950200364X.

[137] G. Plotkin. Lcf considered as a programming language. *Theoretical Computer Science*, 5(3):223 – 255, 1977. doi:10.1016/0304-3975(77)90044-5.

[138] G. Plotkin and J. Power. *Notions of Computation Determine Monads*, pages 342–356. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. doi:10.1007/3-540-45931-6_24.

[139] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975. doi:10.1016/0304-3975(75)90017-1.

[140] E. Polonovski. Strong normalization of lambda-bar-mu-mu-tilde-calculus with explicit substitutions. In *FOSSACS*, volume 2987 of *Lecture Notes in Computer Science*, pages 423–437, Barcelona, Spain, 2004. Springer-Verlag. URL: https://hal.archives-ouvertes.fr/hal-00004321.

[141] P.-M. Pédrot and N. Tabareau. An effectful way to eliminate addiction to dependence. In *Proc. Of LICS2017*, 2017.

[142] T. Reid. Géométrie des visibles. In *Œuvres complètes*, pages 186–203. Théophile Jouffroy, Pierre Paul Royer-Collard, 1828. URL: https://books.google.fr/books?id=KtOZDb_UB_kC.

[143] L. Rieg. Extracting Herbrand trees in classical realizability using forcing. In S. R. D. Rocca, editor, *Computer Science Logic 2013 (CSL 2013)*, volume 23 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 597–614, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2013/4221, doi:10.4230/LIPIcs.CSL.2013.597.

[144] L. Rieg. *On Forcing and Classical Realizability*. Theses, Ecole normale supérieure de lyon - ENS LYON, June 2014. URL: https://tel.archives-ouvertes.fr/tel-01061442.

[145] B. Riemann. Über die hypothesen, welche der geometrie zu grunde liegen. *Abhandlungen der Königlichen Gesellschaft der Wissenschaften zu Göttingen*, 1867.

[146] B. Russell. *Introduction to Mathematical Philosophy*. Dover Publications, 1919.

[147] F. Ruyer. *Proofs, Types and Subtypes*. PhD thesis, Université de Savoie, Nov. 2006. URL: https://tel.archives-ouvertes.fr/tel-00140046.

[148] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993. doi:10.1007/BF01019462.

[149] S. G. Simpson. *Subsystems of Second Order Arithmetic*. Perspectives in Logic. Cambridge University Press, 2 edition, 2009. doi:10.1017/CBO9780511581007.

[150] C. Spector. Provably recursive functionals of analysis: A consistency proof of analysis by an extension of principles in current intuitionistic mathematics. In F. D. E. Dekker, editor, *Recursive function theory: Proceedings of symposia in pure mathematics*, volume 5, page 1–27, Providence, Rhode Island, 1962. American Mathematical Society.

[151] T. Streicher. Krivine's classical realisability from a categorical perspective. *Mathematical Structures in Computer Science*, 23(6):1234–1256, 2013. doi:10.1017/S0960129512000989.

[152] G. J. Sussman and G. L. Steele, Jr. An interpreter for extended lambda calculus. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1975.

[153] A. Tarski. *The concept of truth in the languages of the deductive sciences*, pages 152–278. Oxford at the Clarendon Press, 1956, 1953.

[154] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. doi:10.1112/plms/s2-42.1.230.

[155] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. a correction. *Proceedings of the London Mathematical Society*, s2-43(1):544–546, 1938. URL: http://dx.doi.org/10.1112/plms/s2-43.6.544, doi:10.1112/plms/s2-43.6.544.

[156] M. Vákár. *In Search of Effectful Dependent Types*. PhD thesis, University of Oxford, 2017. URL: http://arxiv.org/abs/1706.07997.

[157] S. van Bakel, L. Liquori, S. R. della Rocca, and P. Urzyczyn. Comparing cubes of typed and type assignment systems. *Annals of Pure and Applied Logic*, 86(3):267 – 303, 1997. doi:10.1016/S0168-0072(96)00036-X.

[158] D. van Dalen. *Intuitionistic logic*, pages 224–257. Lou Gobble, Blackwell, Oxford, 2001.

[159] J. Van Oosten. Realizability: a historical essay. *Mathematical Structures in Computer Science*, 12(3):239–263, 2002. doi:10.1017/S0960129502003626.

[160] J. van Oosten. Studies in logic and the foundations of mathematics. In *Realizability: An Introduction to its Categorical Side*, volume 152 of *Studies in Logic and the Foundations of Mathematics*, pages ii –. Elsevier, 2008. doi:10.1016/S0049-237X(13)72046-9.

[161] P. Wadler. Call-by-value is dual to call-by-name. In C. Runciman and O. Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 189–201. ACM, 2003. URL: http://doi.acm.org/10.1145/944705.944723, doi:10.1145/944705.944723.

[162] A. Wright. Simple imperative polymorphism. In *LISP and Symbolic Computation*, pages 343–356, 1995.

[163] E. Zermelo. Beweis, dass jede menge wohlgeordnet werden kann. (aus einem an herrn hilbert gerichteten briefe). *Mathematische Annalen*, 59:514–516, 1904. URL: http://eudml.org/doc/158167.