Programmes, preuves et fonctions : le ménage à trois de Curry-Howard

Emmanuel Beffara Lionel Vaux

13 juin 2013

Résumé

Ce cours est une introduction à la théorie de la démonstration, et plus particulièrement à cette articulation entre logique mathématique et théorie des langages de programmation qu'est la correspondance de Curry-Howard. Cette dernière énonce que démonstrations et programmes sont des objets de même nature, y compris d'un point de vue calculatoire : l'exécution des programmes a pour contrepartie une procédure de normalisation des preuves, appelée *élimination des coupures*.

La première partie présente cette correspondance dans le cas de la logique propositionnelle minimale et du λ-calcul, qui en illustre les idées fondatrices. On introduit ensuite la notion de sémantique dénotationnelle au moyen du modèle relationnel du λ-calcul qui, bien qu'étonnamment simple, révèle déjà une structure riche. En regardant de près cette sémantique et son rapport avec le comportement du λ-calcul, on explique comment apparaissent la logique linéaire, ses règles de déduction et sa procédure de normalisation.

La deuxième partie explore les moyens d'étendre cette approche au-delà du calcul fonctionnel pur et de la logique propositionnelle minimale. On montre d'abord comment l'introduction de la quantification permet de représenter les types de données usuels. Le cas des mécanismes de contrôle en programmation impérative est également bien établi, bien que la recherche y soit active. D'autres aspects sont plus exploratoires : effets, calcul concurrent, parallélisme. On donne un aperçu des apports de la réalisabilité classique et de la logique linéaire dans ces directions.

1 Introduction

Les liens entre logique et informatique sont profonds et anciens : le théorème d'incomplétude de Gödel lui-même ne peut être compris sans le concept de fonction calculable. En forçant le trait, et si on fixe sa naissance au moment où Church, Gödel et Turing font surgir l'unicité et l'universalité de la notion de calcul, on peut dire que l'informatique est apparue comme une branche de la logique, avant que la réalisation matérielle des ordinateurs

et l'invention des langages de programmation n'en fasse une science à part entière.

L'incomplétude constitue un obstacle indépassable au programme de Hilbert, qui visait à une présentation « finitiste » des mathématiques. Loin d'un échec, cette découverte a renouvelé les questionnements de la logique, en mettant en évidence l'importance du système de raisonnement dans lequel on se place : elle a fait de la démonstration un objet d'étude.

La théorie de la démonstration formalise les preuves comme des objets finis, syntaxiques, sur lesquels on peut « calculer ». Gentzen a par exemple ramené la cohérence de l'arithmétique à la terminaison d'une certaine procédure de normalisation des preuves, définie par des transformations locales : l'élimination des coupures. Très tôt, on a également exploré un possible contenu calculatoire des démonstrations elles-mêmes : la réalisabilité de Kleene associe des fonctions calculables aux preuves de l'arithmétique intuitionniste de Heyting.

Cette approche a connu un essor considérable dans les années 1970, avec la correspondance de Curry–Howard, initialement formulée pour la logique minimale et le λ-calcul simplement typé : dans ce cadre les preuves *sont* les programmes typés par des formules. Les développements issus de cette découverte sont nombreux : de nouveaux résultats de cohérence ; la mise au point de systèmes de types riches et expressifs ; l'invention de nouveaux systèmes de démonstration avec de bonnes propriétés ; le développement de la preuve automatique ou assistée ; l'extraction de programmes certifiés corrects ; etc.

Ce cours propose un panorama de la correspondance entre preuves et programmes et des notions, outils et résultats principaux qui y sont développés, avec un biais vers deux thématiques nées en France : la logique linéaire de Girard et la réalisabilité classique de Krivine. On survolera également quelques sujets de recherche actuels dans ces domaines. L'esprit général du cours est d'exposer les grandes idées dans un cadre simple et autosuffisant… quitte à mentir par omission ou volontairement réécrire l'histoire. On émaille toutefois le récit de références plus orthodoxes.

On commence par détailler la correspondance pour la logique minimale propositionnelle et le λ-calcul : c'est le cas fondateur, pour lequel la théorie est la mieux établie. Dans ce cadre, on introduit la notion de sémantique dénotationnelle : il s'agit de capturer le comportement global d'un programme ou d'une preuve, vu comme une fonction. On montre ensuite comment, par une étude fine de la sémantique, on peut inventer la logique linéaire, ses règles de déduction et sa procédure de normalisation.

La suite du cours est consacrée aux extensions possibles de la correspondance au delà de la logique minimale propositionnelle et du calcul purement fonctionnel. On verra qu'enrichir le système logique revient à étendre le langage de programmation sous-jacent : types de données, polymorphisme, contrôle du flot d'exécution. On reviendra également à la logique linéaire, pour proposer une interprétation des preuves dans les algèbres de processus.

En guise de conclusion, la dernière partie apporte un éclairage possible des idées précédentes, à travers la notion de sémantique quantitative, qui est l'origine véritable de la logique linéaire.

2 Preuves et programmes

Dans cette première partie, on décrit la correspondance de Curry-Howard telle que formalisée à la fin des années 1970 [17] : la déduction naturelle minimale est isomorphe au λ-calcul simplement typé. On commence par introduire les acteurs en présence.

2.1 Rappels : calcul propositionnel et valeurs booléennes

On considère les formules engendrées par les connecteurs de la logique usuelle : $\wedge$ (et), $\vee$ (ou), $\neg$ (non), $\Rightarrow$ (implique), $\top$ (vrai), $\bot$ (faux). Plus précisément, on se donne un ensemble dénombrable $\mathcal{V}$ de variables propositionnelles $X, Y, Z, \ldots$ et on considère les expressions données par la grammaire :

$$A, B, C, \ldots ::= X \mid A \Rightarrow B \mid A \wedge B \mid A \vee B \mid \neg A \mid \top \mid \bot.$$

Cette notation un peu cryptique au premier abord est une manière concise de dire les choses suivantes : [1]

- l'ensemble des formules est le plus petit ensemble d'expressions (c'est-à-dire d'arbres finis étiquetés) tel que, si A, B, C sont des formules, alors X, $A \Rightarrow B$, $A \wedge B$, $\ldots$ sont des formules ;
- on annonce que dans la suite, on utilisera les lettres A, B, C, $\ldots$ pour noter les formules (il est déjà implicite que X est une variable propositionnelle).

En termes d'arbres : un nœud d'étiquette « X » (une variable propositionnelle) est une feuille, un nœud d'étiquette « $\Rightarrow$ » a deux fils, $\ldots$

À un moment dans tout cursus d'informatique ou de mathématiques, on subit la définition de la vérité par les tables. Souvenez-vous, ça ressemblait à la table 1. Ces tables synthétisent l'information nécessaire pour calculer la valeur de vérité d'une formule à partir de celles des variables propositionnelles : pour toute distribution de valeurs de vérité $d : \mathcal{V} \to \{0, 1\}$ (l'adresse d'une ligne dans la table), on définit ainsi la « vérité sous les hypothèses d », par induction sur les formules. Les « théorèmes » sont alors les formules qui sont vraies indépendamment de la distribution : on parle de tautologies.

1. Dans les livres de logique un peu poussiéreux, aussi excellents soient-ils par ailleurs, on commence par considérer des mots, écrits sur un unique alphabet contenant les variables propositionnelles, les parenthèses et les connecteurs, puis on définit les formules bien parenthésées, le connecteur principal d'une formule, la relation de sous-formule, *etc.* Mais en 2013 et vu le public de cette école, on peut bien ne plus considérer le lecteur comme une machine de Turing à qui il faudrait enseigner l'analyse syntaxique !

A	B	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \vee (A \Rightarrow B)$
0	0	0	0	1	1
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	1	1

TABLE 1 – Exemples de tables de vérité : les connecteurs binaires et une tautologie

De là on déduit par exemple : la validité du tiers-exclu $A \vee \neg A$; les lois de De Morgan $\neg(A \Rightarrow B) \iff A \wedge \neg B$, $\neg(A \wedge B) \iff \neg A \vee \neg B$, etc. ; le codage de l'implication $A \Rightarrow B \iff \neg A \vee B$. Le connecteur d'équivalence $\iff$ est ici une « macro » : $(A \iff B) \coloneqq (A \Rightarrow B) \wedge (B \Rightarrow A)$. C'est une tautologie ssi les colonnes de A et de B dans une table sont les mêmes.

Cette interprétation booléenne est bien pratique pour présenter les connecteurs logiques au débutant, mais ça ne va pas bien loin et, surtout, ça n'a pas grand chose à voir avec la démonstration : qui voudrait systématiquement démontrer l'implication $A \Rightarrow B$ en vérifiant que A est faux ou que B est vrai, pour chacune des distributions de valeurs de vérité ?

2.2 Déduction naturelle

La théorie de la démonstation est la science des preuves, en tant qu'objets mathématiques. C'est-à-dire qu'on introduit des représentations formelles des démonstrations, sur lesquelles on peut ensuite raisonner.

L'idée est de considérer une preuve comme un arbre, dont les nœuds seraient les étapes de raisonnement. Par exemple, la manière la plus naturelle de prouver $A \wedge B$, c'est de prouver séparément A et B. On aurait donc une règle de déduction $\frac{A \qquad B}{A \wedge B}\ \langle\wedge\rangle$: de haut en bas, on construit une preuve, et on lit « si on a démontré A et B, on peut en déduire $A \wedge B$ » ; de bas en haut, on recherche une preuve, et on lit « pour démontrer $A \wedge B$, il suffit de démontrer A et B ».

Pour prouver $A \Rightarrow B$, on commence généralement par « supposons A » et on cherche à démontrer B avec cette hypothèse. On aurait donc quelque chose comme $\frac{A \vdash B}{A \Rightarrow B}\ \langle\Rightarrow\rangle$, où le signe « $\vdash$ » (on lit généralement « thèse ») n'est pas un connecteur mais un séparateur entre les hypothèses courantes et la formule qu'on démontre. C'est-à-dire qu'un état de démonstration n'est pas simplement constitué de la formule de conclusion, mais également des hypothèses sous lesquelles on travaille. Plus formellement :

Définition 2.1. *On appelle séquent la donnée $\Gamma \vdash A$ d'une formule A (la conclusion) et d'une famille finie Γ de formules (les hypothèses). Une règle de déduction R est alors la donnée*

$$\frac{}{\Gamma, A \vdash A}\ (\text{Hyp}) \qquad \frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\ (\bot)$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ \langle\Rightarrow\rangle \qquad \frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \wedge A_2}\ \langle\wedge\rangle$$

$$\frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ [\Rightarrow] \qquad \frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_1}\ [\wedge_1] \qquad \frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_2}\ [\wedge_2]$$

TABLE 2 – Règles de la déduction naturelle classique.

$$\frac{\Gamma_1 \vdash A_1 \qquad \cdots \qquad \Gamma_n \vdash A_n}{\Gamma \vdash A}\ R$$

d'une famille de séquents prémisses, et d'un séquent conclusion. Enfin, une preuve est un arbre dont les nœuds sont des instances de règles, et tel que les séquents prémisses d'un nœud sont les séquents conclusions de ses parents.

En pratique, n vaut 0, 1 ou 2. Afin de ne pas alourdir la présentation, on restreint dans la suite le jeu de connecteurs à $\{\Rightarrow, \wedge, \bot\}$. Tous les autres peuvent s'écrire à partir de ceux-là, en posant $\neg A \coloneqq A \Rightarrow \bot$, $A \vee B \coloneqq \neg(\neg A \wedge \neg B)$ et $\top \coloneqq \neg\bot$.

La déduction naturelle classique est alors le système de déduction comportant les règles données en table 2. La règle (Hyp) est la seule règle possible pour une feuille : c'est l'utilisation directe d'une hypothèse courante (le contexte Γ, A contient A en plus des formules de Γ). On a deux jeux de règles pour chaque connecteur binaire :

- les règles $\langle\Rightarrow\rangle$ et $\langle\wedge\rangle$, qu'on a vues plus haut, sont appelées règles d'introduction : elles disent comment on démontre directement une formule en fonction de son connecteur principal ;
- les règles $[\Rightarrow]$, $[\wedge_1]$ et $[\wedge_2]$ sont appelées règles d'élimination : elles disent comment on peut utiliser une formule (par exemple une hypothèse déchargée via (Hyp)) selon son connecteur principal.

La plus petite démonstration d'un théorème qu'on puisse concevoir est :

$$\dfrac{\dfrac{}{A \vdash A}\ (\text{Hyp})}{\vdash A \Rightarrow A}\ \langle\Rightarrow\rangle$$

qui établit la réflexivité de l'implication. Ce n'est bien sûr pas la plus intéressante mais, comme l'égalité ou la fonction identité, elle joue un rôle essentiel.

Un peu à part, la règle $(\bot)$ correspond au raisonnement par l'absurde : pour démontrer A, il suffit d'établir que $\neg A$ (c'est-à-dire $A \Rightarrow \bot$) est contradictoire. Dans le discours mathématique courant, on fait assez difficilement la distiction entre l'introduction d'une négation et le raisonnement par l'absurde, c'est pourtant bien différent :

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\,\langle\Rightarrow\rangle \quad vs \quad \frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\,(\bot) \quad .$$

Là où les règles précédentes s'intéressaient à la structure des formules, la règle ($\bot$) permet de raisonner sur la vérité : c'est vrai, puisque ce n'est pas faux. C'est par exemple celle qui permet de démontrer $\neg\neg A \Rightarrow A$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg\neg A, \neg A \vdash \neg\neg A}\,\text{(Hyp)} \quad (\text{c.-à-d. } \neg A \Rightarrow \bot)}{\neg\neg A, \neg A \vdash \bot}\,[\Rightarrow] \qquad \dfrac{}{\neg\neg A, \neg A \vdash \neg A}\,\text{(Hyp)}}{\neg\neg A \vdash A}\,(\bot)}{\vdash \neg\neg A \Rightarrow A}\,\langle\Rightarrow\rangle \quad .$$

Notez que ce type de raisonnement n'est d'ailleurs pas restreint aux formules portant sur la négation : la règle ($\bot$) est nécessaire pour répondre à l'exercice suivant (on y reviendra).

Exercice 2.2. *La formule* $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ *est appelée* loi de Peirce. *Il est facile de voir que c'est une tautologie : donnez-en une preuve en déduction naturelle classique.*

Théorème 2.3. *La déduction naturelle classique est correcte et complète pour la vérité booléenne : le séquent* $\vdash A$ *est prouvable si et seulement si* A *est une tautologie.*

Démonstration. Il est assez facile d'établir la correction (on ne prouve que des tautologies), en démontrant par induction sur les preuves un énoncé plus général : si $\Gamma \vdash A$ est prouvable alors toute distribution de valeurs de vérité qui valide les formules de Γ valide également A.

Établir la complétude (toutes les tautologies sont prouvables) demande un petit peu plus de travail mais reste élémentaire : on est loin du théorème de complétude de Gödel pour le calcul des prédicats. [2] Essentiellement, on montre qu'on peut raisonner sur les valeurs de vérité. Détailler la suite de la démonstration constitue d'ailleurs un exercice accessible et intéressant.

Soient V un ensemble fini de variables propositionnelles, et $d : V \to \{0, 1\}$ une distribution de valeurs de vérité sur V. On peut construire le contexte qui représente l'hypothèse « d est vraie » : $\Gamma_{d,V} = (\epsilon_d X_1, \ldots, \epsilon_d X_n)$, avec $\epsilon_d X \coloneqq X$ si $d(X) = 1$ et $\epsilon_d X \coloneqq \neg X$ sinon. On montre alors par induction sur la formule A que, pour tout V contenant les variables de A et toute distribution d sur V, si $d(A) = 1$ alors $\Gamma_{d,V} \vdash A$ et sinon $\Gamma_{d,V} \vdash \neg A$.

D'autre part, on construit une preuve de la disjonction des $\Gamma_{d,V}$ pour V fixé : $D_V = \bigvee_{d:V\to\{0,1\}} \bigwedge_{X\in V} \epsilon_d X$. Cette formule généralise le tiers exclu,

2. Comme son modeste cousin 2.3, le théorème de complétude de Gödel établit que le système de preuve qu'on se donne est suffisant, cette fois pour la logique du premier ordre : tout ce qui est vrai dans une théorie est démontrable à partir des axiomes de la théorie. Par contraste, le théorème d'incomplétude pose une limite à la notion de théorie : une théorie récursivement énumérable, non contradictoire et contenant l'arithmétique élémentaire ne peut pas être complète, c'est-à-dire qu'il existe une formule A telle que ni A ni $\neg A$ ne s'en déduit.

et exprime intuitivement qu'une des distributions de valeurs de vérité est nécessairement validée : sa preuve consiste à énumérer toutes les possibilités de vérité pour les variables, et elle est de taille exponentielle en celle de V.

Enfin, on assemble le tout : si A est une tautologie, alors $d(A) = 1$ pour tout d, donc on a une preuve de $\Gamma_{d,V} \vdash A$, et donc de $\bigwedge_{X \in V} \epsilon_d X \vdash A$. On compose ces 2^V preuves avec celle de D_V pour obtenir une preuve de A. □

Remarque 2.4. *Plutôt que de coder les autres connecteurs grâce aux lois de De Morgan, on aurait pu les conserver et donner les règles d'introduction et d'élimination correspondantes : on aurait toujours un résultat de complétude.*

2.3 Intuitionnisme : les preuves comme fonctions

Le statut un peu particulier du raisonnement par l'absurde suggère de considérer le système privé de la règle ($\bot$), qu'on appelle déduction naturelle minimale. Ce système reste bien sûr correct, mais il n'est plus complet, et par exemple $\neg\neg A \Rightarrow A$ et $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ n'y sont plus démontrable.

Pourquoi accorder la moindre attention à un système si cruellement estropié ? Initialement, c'était justifié par une exigence d'ordre philosophique, concernant la constructivité de la logique. Plutôt que d'entrer dans le détail de cette préoccupation, par ailleurs d'une portée quasi-nulle en logique propositionnelle, on la ramène à une tentative d'« explication » des preuves : en plus d'établir qu'une formule A est vraie, une preuve de A devrait dire *comment* A est vraie. On parle d'*intuitionnisme*.

Plus explicitement, on considère les preuves d'une formule d'une manière procédurale : une preuve de $A \Rightarrow B$, c'est une fonction qui à toute preuve de A associe une preuve de B ; une preuve de $A \wedge B$, c'est une preuve de A et une preuve de B ; il n'y a pas de preuve de $\bot$. On associe ainsi une opération ensembliste à chaque connecteur : $A \Rightarrow B = B^A$, $A \wedge B = A \times B$, $\bot = \emptyset$. Ce point de vue est connu sous le nom d'interprétation BHK, des noms de Brouwer, Heyting et Kolmogorov.

L'approche est intéressante mais, sans même pousser la formalisation plus avant, on en voit rapidement la limite : on n'a pas grand chose à dire sur la négation. En effet, si A admet une preuve en ce sens, $\neg A = A \Rightarrow \bot$ n'en admet pas ; et si A n'a pas de preuve, $\neg A$ a pour unique preuve l'identité sur $\emptyset$. Bref, on n'est pas sorti du modèle booléen, sauf à ne considérer que le fragment sans absurdité ni négation.

Autre faille, même dans le fragment $\{\Rightarrow, \wedge\}$: une preuve de $A \wedge B$ n'est en général pas directement issue d'une preuve de A et d'une preuve de B.

Exemple 2.5. *Si on a une preuve π de $\vdash A$, on peut construire :*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,(Hyp) \quad \dfrac{}{A \vdash A}\,(Hyp)}{A \vdash A \wedge A}\,\langle\wedge\rangle}{\vdash A \Rightarrow A \wedge A}\,\langle\Rightarrow\rangle \qquad \begin{array}{c}\pi \\ \nabla \\ \vdash A\end{array}}{\vdash A \wedge A}\,[\Rightarrow]$$

On voit tout de même poindre le début d'une idée : en prenant en compte le contexte, on peut étendre le point de vue de BHK et considérer toute preuve d'un séquent $A_1, \ldots, A_n \vdash B$ comme définissant une fonction de $A_1 \times \cdots \times A_n$ dans B. L'axiome (Hyp) s'interprète naturellement comme une projection ; l'introduction $\langle\wedge\rangle$ est bien la règle de formation d'un couple ; la partie gauche de l'exemple 2.5, de conclusion $A \Rightarrow A \wedge A$, décrit la fonction diagonale $(x \mapsto (x, x))$ de A ; la règle $[\Rightarrow]$ n'est autre que l'application d'une fonction à son argument. La preuve ci-dessus peut donc être interprétée comme *calculant* le couple $\langle\pi,\pi\rangle$, en appliquant la diagonale à π.

Dans sa version initiale, la correspondance de Curry–Howard n'est rien de plus que la formalisation des idées du paragraphe précédent : les preuves en déduction naturelle minimale correspondent exactement aux termes d'un langage de programmation fonctionnel, le λ-calcul simplement typé.

2.4 λ-calcul

Le λ-calcul est l'archétype des langages de programmation fonctionnels. Il correspond à une théorie naïve des fonctions : à partir d'un terme t qui dépend d'une variable x (une fonction), on peut toujours former le terme $\lambda x\, t$ qui représente $x \mapsto t$ (l'abstraction de la fonction) ; de même, on peut écrire l'application formelle d'un terme s à un autre terme t, notée $(s)\, t$.[3] On considère également la possibilité de former des couples et des projections.

Définition 2.6. *Étant donné un ensemble (infini dénombrable) de variables de termes $x, y, z, \ldots$, les expressions du λ-calcul sont donnés par la grammaire :*

$$s, t, \ldots ::= x \mid \lambda x\, t \mid (s)\, t \mid \langle s,t\rangle \mid \pi_1 s \mid \pi_2 s \quad .$$

Les termes du λ-calcul sont les expressions dans lesquelles on considère que $\lambda x\, s$ lie les occurrences de la variable x dans s : on identifie les expressions qui ne diffèrent que par le choix des variables liées.[4]

3. Ceci n'est pas une erreur de frappe : on écrit $(s)\, t$ pour l'application de la *fonction* s à l'*argument* t. La notation usuelle $s(t)$ présenterait le double inconvénient de conserver l'ambigüité avec « la valeur de s en t » et d'encourager la multiplication des parenthèses en cas d'applications successives : $(s(t))(u)$ ou, au mieux, $s(t)(u)$. Avec la notation proposée, promue par Jean-Louis Krivine, on peut en faire l'économie : on écrira par exemple $(s)\, t\, u \coloneqq ((s)\, t)\, u$ et $\lambda x\, (s)\, t \coloneqq \lambda x\ ((s)\, t)$.

4. Formellement, il faudrait ici introduire la notion d'α-équivalence, qui correspond à autoriser le renommage des variables liées lorsque cela ne crée pas de confusion avec les variables déjà présentes. Ce serait beaucoup d'efforts pour une notion transparente pour les mathématiciens ($\sum_{i=1}^{n} \sum_{j=1}^{p} a_{i,j} = \sum_{j=1}^{n} \sum_{i=1}^{p} a_{j,i}$) comme pour les programmeurs.

Si s et t sont des termes, et x est une variable, on note $s[t/x]$ le terme obtenu en substituant toutes les occurrences libres (:= non liées) de x dans s par t, quitte à renommer les variables liées pour éviter les captures.[5] Le mécanisme fondamental du calcul, appelé β-réduction, revient alors à une sorte de paraphrase :

$$(\lambda x\, s)\, t \rightsquigarrow s[t/x]$$

soit « le résultat de l'application de $x \mapsto s$ à t, c'est la valeur de s en t ».

En étendant la réduction aux couples et projections, on obtient :

Définition 2.7. *La β-réduction est la plus petite relation contextuelle*[6] *telle que*

$$(\lambda x\, s)\, t \to_\beta s[t/x], \quad \pi_1\langle s,t\rangle \to_\beta s \quad \textit{et} \quad \pi_2\langle s,t\rangle \to_\beta t$$

pour tous termes s et t. On note $\to_\beta^*$ sa clôture réflexive et transitive, et on appelle β-équivalence l'équivalence engendrée.

Exercice 2.1. *Réduisez autant que possible le terme $(\lambda x\, \lambda y\, (x)\, (x)\, y)\, \lambda x\, x$.*

Remarque 2.8. *Alors qu'on a l'impression de n'avoir rien fait, on peut montrer que cette notion de calcul est Turing-puissante : on peut coder les entiers comme des fonctionnelles d'itération $\underline{n} := \lambda x\, \lambda y\, (x)^n\, y$, où $(x)^n\, y$ est l'application itérée n fois de x à y, et montrer que les fonctions représentables sur ces* entiers de Church *sont exactement les fonctions calculables. En particulier, on parle ici de fonctions partielles : le calcul ne termine pas toujours (considérer par exemple le terme $\Omega := (\lambda x\, (x)\, x)\, \lambda x\, (x)\, x$, qui se réduit en lui-même).*

Exercice 2.2. *Donnez des λ-termes qui représentent l'addition et la multiplication sur les entiers. Par exemple, pour l'addition, on cherche un terme $\underline{add}$ tel que $(\underline{add})\, \underline{m}\, \underline{n}$ se réduise en $\underline{m+n}$.*

Théorème 2.9. *La β-réduction a la propriété de Church-Rosser : si s et t sont β-équivalents, alors il existe u tel que $s \to_\beta^* u$ et $t \to_\beta^* u$.*

C'est le théorème fondamental du λ-calcul, il indique que le choix de l'ordre des réductions ne change pas la valeur calculée : tout terme est équivalent à au plus un terme irréductible, qu'on appelle alors sa *forme normale*.

Esquisse de démonstration. La technique standard pour établir ce résultat est attribuée à Tait et Martin-Löf : on introduit une version parallèle de la β-réduction, qui permet de réduire en un seul pas un nombre arbitraire

5. Autrement dit, on s'arrange pour que les occurrences libres des variables de t restent libres dans $s[t/x]$: simple mesure d'hygiène syntaxique, ennuyeuse mais indispensable.

6. Par relation contextuelle, on entend une relation qui passe à travers les constructeurs, c'est-à-dire qu'on peut appliquer les étapes de réduction dans n'importe quel sous-terme.

$$\frac{}{\Gamma, x : A \vdash x : A}\ (\text{Var}) \qquad \frac{\Gamma, x : A \vdash s : B}{\Gamma \vdash \lambda x\, s : A \Rightarrow B}\ (\lambda)$$

$$\frac{\Gamma \vdash s : A \Rightarrow B \quad \Gamma \vdash t : A}{\Gamma \vdash (s)\, t : B}\ (\text{App})$$

$$\frac{\Gamma \vdash s_1 : A_1 \qquad \Gamma \vdash s_2 : A_2}{\Gamma \vdash \langle s_1, s_2 \rangle : A_1 \wedge A_2}\ (\text{Cpl}) \qquad \frac{\Gamma \vdash s : A_1 \wedge A_2}{\Gamma \vdash \pi_1 s : A_1}\ (\pi_1)$$

$$\frac{\Gamma \vdash s : A_1 \wedge A_2}{\Gamma \vdash \pi_2 s : A_2}\ (\pi_2)$$

TABLE 3 – Règles de typage simple pour le λ-calcul avec types produits.

d'expressions réductibles (on dit *redex*) simultanément présentes dans un terme. Pour cette variante, on obtient la confluence forte : tous les réduits (en un pas) d'un terme donné admettent un réduit commun. La confluence et donc la propriété de Church-Rosser pour la β-réduction s'en déduisent. □

2.5 Correspondance de Curry-Howard

La déduction naturelle minimale peut maintenant être vue comme un système de typage qui assure que les λ-termes dénotent des fonctions totales : la β-réduction des termes typés mène à une forme normale. Formellement :

Définition 2.10. *Un contexte de typage est une famille finie de formules Γ, indexée par des variables. On dit que le terme t est de type A dans le contexte Γ si le jugement $\Gamma \vdash t : A$ est dérivable suivant les règles de la table 3. règle !de typage*

En particulier Γ doit définir le type des variables libres de t. Si t est clos (sans variable libre), on peut parler de son type sans contexte : t a le type A, et on note $t : A$, si $\vdash t : A$ est dérivable.

Propriété 2.11. *Le séquent $\Gamma \vdash A$ est prouvable en déduction naturelle minimale si et seulement s'il existe un λ-terme t tel que $\Gamma \vdash t : A$ (quitte à réindexer Γ).*

Cette conséquence immédiate de la définition précédente est moins importante que sa justification : les règles d'inférence de la déduction naturelle minimale *sont* les règles de typage des constructeurs du λ-calcul. C'est en ce sens qu'on parle d'isomorphisme entre preuves et programmes.

Modulo cet isomorphisme, la β-réduction devient une procédure de calcul sur les preuves : l'*élimination des coupures*. On entend ici par coupure un détour apparemment inutile : l'introduction d'un connecteur, qu'on élimine aussitôt. Ces coupures prennent donc toutes la forme d'un redex, qu'on peut réduire. Par exemple, une coupure sur une implication $A \Rightarrow B$ correspond à l'utilisation d'un résultat intermédiaire A pour prouver B, et l'élimination de la coupure donne une preuve directe de B.

Exemple 2.12. *La sous-preuve gauche de l'exemple 2.5 donne le terme* $\lambda x\,\langle x,x\rangle : A \Rightarrow A \wedge A$. *De plus, si* $t : A$ *est le terme typé correspondant à* π, *la preuve de* $A \wedge A$ *devient* $(\lambda x\,\langle x,x\rangle)\,t$ *qui se* β*-réduit en* $\langle t,t\rangle : A \wedge A$.

Cette procédure d'élimination des coupures termine toujours et mène donc à une preuve sans coupure. C'est le résultat annoncé : le typage assure l'existence de formes normales.

Théorème 2.13. *Les termes typés normalisent fortement : si* $\Gamma \vdash t : A$ *est dérivable, les chemins de réduction depuis* t *sont de longueur bornée.*

Ce théorème et ses extensions à des systèmes plus riches justifient l'intérêt porté par les logiciens à la correspondance de Curry-Howard : ils permettent de relier la cohérence d'un système logique à son contenu calculatoire.

En effet, pour établir la cohérence d'un système logique, le premier réflexe est de faire référence à une notion de vérité préexistante, un modèle : on ne peut démontrer que des résultats vrais. Cette approche n'est cependant pas très satisfaisante dès qu'on se pose la question du fondement : qu'est-ce qui justifie le modèle ? Un méta-modèle ? On stagne.

Comment alors s'assurer de la cohérence d'un système sans faire référence à un modèle ? En 1934, Gentzen a démontré son *Hauptsatz* : toute démonstration dans l'arithmétique de Peano peut être ramenée à une démonstration *sans coupures*. Or il est facile de vérifier structurellement que les démonstrations sans coupures ne permettent pas d'obtenir de contradiction, ce qui établit la cohérence de l'arithmétique sans jamais faire mention de son interprétation standard dans les entiers. Pour l'occasion Gentzen invente deux représentations formelles des démonstrations : la déduction naturelle, qui sert de référence, et le calcul des séquents, logiquement équivalent, et pour lequel il démontre le théorème. [7]

Dans le cas du calcul propositionnel, on n'a aucun doute sur l'existence ou la nature du modèle sous-jacent : le modèle booléen à deux éléments ne recèle aucun monstre. On peut tout de même illustrer ici la portée du théorème de normalisation forte : il permet de distinguer la logique classique de la logique minimale, sans faire référence à une notion de modèle.

Exercice 2.14. *On a annoncé en section 2.3 que la loi de Peirce* $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ *n'était pas démontrable en déduction naturelle minimale. On peut maintenant justifier cette affirmation : montrez qu'il n'existe pas de terme clos de ce type. La première étape est de se ramener à un terme en forme normale.*

Les preuves du théorème de normalisation forte pour le λ-calcul abondent dans la littérature. Celle qu'on donne ici est dûe à René David, et elle a l'intérêt de n'utiliser qu'un argument purement arithmétique.

7. Cette approche ne permet bien sûr pas de s'affranchir du théorème d'incomplétude de Gödel : la preuve de Gentzen, si on la formalise, utilise plus que l'arithmétique.

Preuve du théorème 2.13. On commence par établir :

Lemme 2.15. *Si s, t, u_1, …, u_p sont fortement normalisables et $v = ((s)\,t)\,u_1 \cdots u_p$ ne l'est pas, alors il existe w tel que $(w[t/x])\,u_1 \cdots u_p$ ne soit pas fortement normalisable et $s \to_\beta^* \lambda x\, w$.*

La preuve se fait par récurrence sur $\ell(s) + \ell(t) + \sum_j \ell(u_j)$, où $\ell(s)$ est la longueur maximale d'une suite de β-réductions depuis s. Ensuite :

Lemme 2.16. *Si on a $\Gamma, x : A \vdash s : B$ et $\Gamma \vdash t : A$, avec s et t fortement normalisables, alors $s[t/x]$ est fortement normalisable.*

La preuve se fait par induction sur le triplet $(A, \ell(s), s)$, ordonné lexicographiquement. On utilise le lemme précédent pour le cas où s est une application. La preuve du théorème en découle, par induction sur les termes. □

Remarque 2.17. *En écho à la remarque 2.4, on pourrait également proposer une extension du λ-calcul avec des types sommes pour la disjonction, et un type vide pour l'absurde. Les propriétés calculatoires du système obtenu seraient bien moins satisfaisantes, notamment parce que les formes normales de ce système n'ont pas grand chose de canonique. À ce sujet, voir le chapitre 10 du* Proofs and Types *[15].*

3 Programmes et fonctions

La correspondance entre preuves et programmes nous a permis de relier la cohérence logique et la prouvabilité à une propriété du calcul. C'était déjà le cas chez Gentzen, sans que le procédé de calcul soit perçu comme l'exécution d'un programme, et c'était embryonnaire chez Curry dès 1934 [4]. C'est toutefois depuis l'établissement par Howard à la fin des années 1960 d'un isomorphisme entre λ-calcul et déduction naturelle minimale [17] que cette correspondance est devenue un axe majeur de communication entre la logique mathématique et l'informatique. D'un côté on a pu voir les formules logiques comme étant les spécifications de programmes, ce qui a par exemple permis de développer des assistants de preuve, comme Coq, avec capacités d'extraction de programmes fonctionnels à partir de démonstrations formelles. De l'autre, les concepts développés pour l'étude des langages de programmation sont venus nourrir l'étude des systèmes logiques. La sémantique dénotationnelle est l'un de ces outils issus de la théorie de la programmation.

3.1 Sémantique dénotationnelle

Cette approche initiée par Dana Scott et Christopher Strachey [26], là encore dans les annés 1960, consiste à décrire formellement les comportements entrée-sortie des programmes comme des fonctions : deux programmes équivalents pour le calcul dénotent la même fonction. On remplace donc les objets

finis et inductifs que sont les programmes, avec une dynamique potentiellement complexe,[8] par des objets infinis, mais statiques.

Dans le cas du λ-calcul, la solution naïve inspirée par BHK échoue : si on interprète les λ-termes comme les éléments d'un ensemble Λ, n'importe quelle abstraction $\lambda x\, s$ peut être vue comme une fonction $\Lambda \to \Lambda$, mais c'est aussi un terme, qui doit être interprété dans Λ. Or l'ensemble des fonctions $\Lambda \to \Lambda$ ne peut pas être plongé dans Λ, pour une simple raison de cardinalité !

Pour contourner cette difficulté, il faut raffiner le modèle en s'appuyant sur des propriétés nécessairement vérifiées par les dénotations de programmes. Scott a proposé d'utiliser les fonctions continues entre domaines. Les domaines sont des ensembles ordonnés particuliers : l'ordre y représente une hiérarchie d'information. Les fonctions continues sont celles qui n'utilisent qu'une information finie sur leur argument pour produire une sortie : c'est le cas des programmes, qui répondent en temps fini (s'ils répondent).

Des raffinements successifs de ces notions ont été étudiés depuis, notamment la stabilité, dûe à Gérard Berry [3], qui approche une forme de séquentialité. Pour un « atome » d'information β produit dans la sortie, à partir de l'information finie a de l'entrée, il y a une information minimum $a_0 \subset a$ qui suffit à produire β : ce minimum a_0 représente l'état d'information au moment où β est produit (et parler de moment n'a de sens que pour un processus séquentiel).

En étudiant la stabilité dans une famille de domaines particuliers, Girard est parvenu à la notion d'espace cohérent [10], qui permet une simplification drastique de la présentation de la sémantique : les fonctions stables entre espaces cohérents sont représentables par leur trace, c'est-à-dire l'ensemble des couples (a_0, β) tels que a_0 est une information minimale pour produire β. C'est en étudiant cette sémantique que Girard a découvert la logique linéaire.

L'importance de toutes ces notions n'est pas qu'historique, et leur maîtrise donne certainement un éclairage indispensable de la logique moderne. Cependant ce cours n'a pas une vocation encyclopédique, et la littérature sur le sujet abonde. Surtout, la connaissance des modèles dénotationnels a bien progressé depuis cet âge héroïque, et nous pouvons ici prendre le parti de réécrire l'histoire, en nous appuyant sur un modèle particulièrement simple issu de ce folklore : le modèle relationnel multiensembliste.

3.2 Modèle relationnel du λ-calcul

Celui-ci reprend dans un cadre très épuré les intuitions précédentes, et peut être expliqué en termes de ressources. On va associer un ensemble $[\![A]\!]$ à chaque type A : un élément de $[\![A]\!]$ représente une information atomique sur un objet de type A, et un programme produit un atome d'information en « consommant » des atomes de son entrée.

8. Rappelons que le λ-calcul pur et non typé est Turing puissant.

$$\frac{}{\Gamma^{[]}, x^{[\alpha]} : A \vdash x^{\alpha} : A}\ [\![\text{Var}]\!] \qquad \frac{\Gamma^{\overline{\gamma}}, x^{\overline{\alpha}} : A \vdash s^{\beta} : B}{\Gamma^{\overline{\gamma}} \vdash \lambda x\, s^{(\overline{\alpha},\beta)} : A \Rightarrow B}\ [\![\lambda]\!]$$

$$\frac{\Gamma^{\overline{\gamma}_0} \vdash s^{([\alpha_1,\ldots,\alpha_k],\beta)} : A \Rightarrow B \qquad \Gamma^{\overline{\gamma}_1} \vdash t^{\alpha_1} : A \quad \cdots \quad \Gamma^{\overline{\gamma}_k} \vdash t^{\alpha_k} : A}{\Gamma^{\sum_{j=0}^{k} \overline{\gamma}_j} \vdash (s)\, t^{\beta} : B}\ [\![\text{App}]\!]$$

$$\frac{\Gamma^{\overline{\gamma}} \vdash s_i{}^{\alpha} : A_i}{\Gamma^{\overline{\gamma}} \vdash \langle s_1, s_2 \rangle^{(i,\alpha)} : A_1 \wedge A_2}\ [\![\text{Cpl}_i]\!] \qquad \frac{\Gamma^{\overline{\gamma}} \vdash s^{(i,\alpha)} : A_1 \wedge A_2}{\Gamma^{\overline{\gamma}} \vdash \pi_i s^{\alpha} : A_i}\ [\![\pi_i]\!]$$

TABLE 4 – Règles d'inférence de la sémantique relationnelle du λ-calcul.

Ainsi, la sémantique d'un terme typé $x : A \vdash s : B$ sera un ensemble de couples $(\overline{\alpha}, \beta)$ où $\overline{\alpha} = [\alpha_1, \ldots, \alpha_n]$ est un multiensemble fini d'élements de $[\![A]\!]$ (on note $\overline{\alpha} \in \mathcal{M}_f[\![A]\!]$) et $\beta \in [\![B]\!]$: on la notera $[\![x : A \vdash s : B]\!]$, voire $[\![s]\!]$ si le contexte et le type sont évidents. Intuitivement, $(\overline{\alpha}, \beta) \in [\![s]\!]$ si s produit β en consommant les atomes $\alpha_1, \ldots, \alpha_n$ (on prend en compte les multiplicités). Par exemple si $\alpha \in [\![A]\!]$, $([\alpha], \alpha) \in [\![x : A \vdash x : A]\!]$: l'identité se contente de reproduire son entrée, atome par atome. On généralise facilement à un nombre quelconque de variables libres ou d'arguments : $([\alpha], [], \alpha) \in [\![x : A, y : B \vdash x : A]\!]$ car y n'est pas utilisé ici. Naturellement, les termes de type $A \Rightarrow B$ sont interprétés comme des parties de $\mathcal{M}_f([\![A]\!]) \times [\![B]\!]$: on travaille moralement sur des traces de fonctions.

Dans la suite, on définit cette interprétation en introduisant des annotations sur les dérivations de typage, de sorte que $(\overline{\alpha}_1, \ldots, \overline{\alpha}_n, \beta) \in [\![s]\!]$ si et seulement si $x_1^{\overline{\alpha}_1} : A_1, \ldots, x_n^{\overline{\alpha}_n} : A_n \vdash s^{\beta} : B$ est dérivable.

Définition 3.1. *Supposons donné un ensemble $[\![X]\!]$ pour chaque variable propositionnelle X. On étend cette interprétation aux formules et aux séquents par :*

$$[\![A \Rightarrow B]\!] \coloneqq \mathcal{M}_f([\![A]\!]) \times [\![B]\!] \quad \textit{et} \quad [\![A \wedge B]\!] \coloneqq [\![A]\!] + [\![B]\!]$$
$$\textit{puis} \quad [\![A_1, \ldots, A_n \vdash B]\!] \coloneqq \mathcal{M}_f([\![A_1]\!]) \times \cdots \times \mathcal{M}_f([\![A_n]\!]) \times [\![B]\!].$$

On définit la sémantique d'un terme typé $\Gamma \vdash s : A$ comme un sous-ensemble de $[\![\Gamma \vdash A]\!]$ au moyen des règles de la table 4 : en notant $\Gamma = x_1 : A_1, \ldots, x_n : A_n$,

$$[\![\Gamma \vdash s : A]\!] \coloneqq \left\{ (\overline{\alpha}_1, \ldots, \overline{\alpha}_n, \beta);\ x_1^{\overline{\alpha}_1} : A_1, \ldots, x_n^{\overline{\alpha}_n} : A_n \vdash s^{\beta} : B \right\} \quad .$$

Il faut expliquer quelques raccourcis de notations. Dans la règle $[\![\text{Var}]\!]$, $\Gamma^{[]}$ signifie que toutes les variables de Γ ont reçu l'annotation vide $[]$. C'est le comportement attendu : on consulte la variable x une fois et on reproduit l'information à l'identique, les autres variables ne sont pas consultées.

Dans les autres règles, $\Gamma^{\overline{\gamma}}$ est une annotation générique de la forme $x_1^{\overline{\alpha}_1} : A_1, \ldots, x_n^{\overline{\alpha}_n} : A_n$. La règle $[\![\lambda]\!]$ revient alors à un simple changement

de parenthésage. Ainsi, le terme clos identité $\vdash \lambda x\, x : A \Rightarrow A$ reçoit la sémantique $[\![\lambda x\, x]\!] = \{([\alpha], \alpha);\ \alpha \in [\![A]\!]\}$.

L'opération ensembliste qui interprète le type produit $A \wedge B$ est la somme $[\![A]\!] + [\![B]\!] \coloneqq (\{1\} \times [\![A]\!]) \cup (\{2\} \times [\![B]\!])$: on force l'union à être disjointe, et c'est le sens de l'indice $i \in \{1, 2\}$ dans les règles $[\![\mathrm{Cpl}_i]\!]$ et $[\![\pi_i]\!]$. Rappelons en effet que la sémantique d'un terme est un ensemble d'atomes : une partie de $[\![A]\!] + [\![B]\!]$ est alors définie de manière univoque par la donnée d'une partie de $[\![A]\!]$ et d'une partie de $[\![B]\!]$, c'est-à-dire par un couple.

La règle la plus significative est celle de l'application. Pour produire β :

- le terme $(s)\, t$ utilise une instance $(\overline{\alpha}, \beta)$ de la fonction s ;
- le terme t doit alors fournir chacun des atomes de $\overline{\alpha} = [\alpha_1, \ldots, \alpha_n]$;
- l'instance de s et les n utilisations de t consomment chacune des informations sur le contexte Γ, qui sont collectées dans la somme $\sum_{i=0}^{n} \overline{\gamma}_i$.

Remarque 3.2. *Cette explication met en valeur une particularité de la sémantique qu'on a choisie : dans $(s)\, t$, le contrôle est donné à la fonction s, qui duplique, efface ou évalue librement son argument pour produire son résultat. Ceci correspond à une stratégie d'évaluation en* appel par nom*, contrairement à l'*appel par valeur *qui évaluerait l'argument avant de donner le contrôle à la fonction.*

Exercice 3.3. *Vérifiez que le terme clos $\vdash \lambda x\, \lambda y\, (x)\, y : (A \Rightarrow B) \Rightarrow (A \Rightarrow B)$ a pour sémantique $\{([(\overline{\alpha}, \beta)], \overline{\alpha}, \beta);\ (\overline{\alpha}, \beta) \in [\![A \Rightarrow B]\!]\}$. Calculez celle de $\vdash \lambda x\, \lambda y\, (x)\, (x)\, y : (A \Rightarrow A) \Rightarrow (A \Rightarrow A)$.*

Théorème 3.4. *La sémantique est préservée par la β-réduction : si s est typé et $s \to_\beta t$ alors $[\![s]\!] = [\![t]\!]$.*

Démonstration. Pour le modèle relationnel qu'on vient de décrire, ce résultat s'obtient sans difficulté par induction sur s, en passant par le lemme suivant :

Lemme 3.5. *Si $\Gamma, x : A \vdash s : B$ et $\Gamma \vdash t : A$, alors $(\overline{\gamma}, \beta) \in [\![s[t/x]]\!]$ ssi on peut écrire $\overline{\gamma} = \sum_{i=0}^{n} \overline{\gamma}_i$ et s'il existe $\overline{\alpha} = (\alpha_1, \ldots, \alpha_n)$ avec $(\overline{\gamma}_0, \overline{\alpha}, \beta) \in [\![s]\!]$ et $(\overline{\gamma}_i, \alpha_i) \in [\![t]\!]$ pour tout i.*

qu'on démontre par induction sur s. □

On a traité ici le cas du λ-calcul simplement typé. On peut en déduire un modèle dénotationnel du λ-calcul non typé, à condition de résoudre une équation sur les types : $D = D \Rightarrow D$, avec laquelle tous les termes deviennent typables de type D. [9] Celle-ci se traduit sémantiquement par : $D = \mathcal{M}_f(D) \times D$. Malheureusement, le plus petit point fixe de cette équation est $D = \emptyset$. On suivra plutôt la construction suivante, dûe à Thomas Ehrhard.

9. On ne traite en fait ici que le fragment du λ-calcul *pur*, réduit à l'abstraction et à l'application, sans les couples. C'est suffisant puisqu'on peut coder $\langle s,t\rangle \coloneqq \lambda x\, (x)\, s\, t$, $\pi_1 s \coloneqq (s)\, \lambda x\, \lambda y\, x$ et $\pi_2 s \coloneqq (s)\, \lambda x\, \lambda y\, y$.

Définition 3.6. *Si X est un ensemble, on note $\mathcal{M}_f(X)^{(\omega)}$ l'ensemble des suites* presque toujours vides *de multiensembles finis sur X. On pose alors $D_0 = \emptyset$, $D_{n+1} = \mathcal{M}_f(D_n)^{(\omega)}$ et $D = \bigcup_{n\geq 0} D_n$.*

On a alors une bijection évidente entre D et $\mathcal{M}_f(D) \times D$, et on peut reprendre les règles de la table 4, *en oubliant les types*. Il n'y a ensuite rien à changer à la preuve du théorème 3.4 pour vérifier que cette sémantique relationnelle du λ-calcul pur est stable par β-réduction. Rappelons que cette manipulation serait complètement inaccessible avec une sémantique à la BHK, où $D \Rightarrow D$ serait l'espace des fonctions : D^D a toujours un cardinal strictement supérieur à celui de D. La proposition de Scott de ne considérer que les fonctions continues avait précisément pour but d'éliminer cet obstacle.

Remarque 3.7. *On peut en fait donner une sémantique du λ-calcul simplement typé dans toute* catégorie cartésienne fermée. *On ne donne pas les détails ici (la référence est le Lambek–Scott [21]), mais cela signifie qu'on considère des morphismes entre espaces mathématiquement structurés, pour lesquels on a une construction d'espace des morphismes et une notion de couple compatible avec la* curryfication *: $(A \wedge B) \Rightarrow C \cong A \Rightarrow (B \Rightarrow C)$.*

Ici les espaces sont des ensembles et les morphismes de A dans B sont les « multirelations », c'est-à-dire les parties de $\mathcal{M}_f(A) \times B$: ce sont les interprétations potentielles de termes à une variable libre $x : A \vdash s : B$. La composition de deux multirelations $f \subseteq \mathcal{M}_f(A) \times B$ et $g \subseteq \mathcal{M}_f(B) \times C$ se calcule :

$$g \circ f = \left\{ \left(\sum_{j=1}^{k} \alpha_j, \gamma \right) ;\ \exists \overline{\beta} = [\beta_1, \ldots, \beta_k],\ (\overline{\beta}, \gamma) \in g \text{ et } \forall j,\ (\overline{\alpha}_j, \beta_j) \in f \right\}$$

qui généralise à toutes les multirelations la sémantique de la substitution $[\![x : A \vdash s[t/y] : C]\!] = [\![y : B \vdash s : C]\!] \circ [\![x : A \vdash t : B]\!] = [\![x : A \vdash (\lambda y\, s)\, t : C]\!]$. La curryfication se ramène à la bijection $\mathcal{M}_f(A + B) \cong \mathcal{M}_f(A) \times \mathcal{M}_f(B)$.

3.3 On a inventé la logique linéaire !

Dans la section précédente, on a donné une explication calculatoire de la règle [[App]]. En y regardant de plus près, on peut simplifier la présentation en distinguant deux opérations : d'une part t est utilisé autant de fois que nécessaire pour produire un multiensemble d'atomes ; d'autre part ce multiensemble est mis en relation avec une utilisation possible de s. On décompose donc la règle [[App]] en deux règles plus élémentaires :

$$\frac{\left(\Gamma^{\overline{\gamma}_j} \vdash t^{\alpha_j} : A\right)_{j=1}^{k}}{\Gamma^{\sum_{j=1}^{k} \overline{\gamma}_j} \vdash t^{[\alpha_1,\ldots,\alpha_k]} : !A}\ [\![!]\!] \qquad \text{et}$$

$$\frac{\Gamma^{\overline{\gamma}} \vdash s^{(\overline{\alpha},\beta)} : A \Rightarrow B \qquad \Gamma^{\overline{\gamma}'} \vdash t^{\overline{\alpha}} : !A}{\Gamma^{\overline{\gamma}+\overline{\gamma}'} \vdash (s)\, t^{\beta} : B}\ [\![@]\!]$$

où on introduit une forme intermédiaire de jugement de typage $\Gamma \vdash t : !A$ qui distingue le cas où t est utilisé comme argument de type A.

Remarque 3.8. *Si $y : B$ (resp. $x : A$) est la seule variable libre de s (resp. t), on retrouve le calcul de la composition de la remarque 3.7 en dérivant :*

$$\dfrac{\dfrac{y^{\overline{\beta}} : B \vdash s^{\gamma} : C}{\vdash \lambda y\, s^{(\overline{\beta},\gamma)} : B \Rightarrow C}\,[\![\lambda]\!] \qquad \dfrac{\left(x^{\overline{\alpha}_j} : A \vdash t^{\beta_j} : B\right)_{j=1}^{k}}{x^{\sum_{j=1}^{k}\overline{\alpha}_j} : A \vdash t^{\overline{\beta}} : !B}\,[\![!]\!]}{x^{\sum_{j=1}^{k}\overline{\alpha}_j} : A \vdash (\lambda y\, s)\, t^{\gamma} : C}\,[\![@]\!]$$

On voit que la règle $[\![@]\!]$ se ramène dans ce cas à une simple composition de relations, au sens suivant.

Définition 3.9. *On appelle relation de A dans B toute partie $s \subset A \times B$. Soient s et t des relations, respectivement de A dans B et de B dans C. On définit la composition $t \cdot s$ par : $(\alpha, \gamma) \in t \cdot s$ s'il existe $\beta \in B$, tel que $(\alpha, \beta) \in s$ et $(\beta, \gamma) \in t$.*

On a alors : $[\![x : A \vdash (s)\, t : C]\!] = [\![y : B \vdash s : C]\!] \cdot [\![x : A \vdash t : !B]\!]$.

On vient de faire le premier pas vers la décomposition de la logique minimale par la *logique linéaire*. En effet, la structure des relations est bien plus simple et naturelle que celle des multirelations. Il est donc certainement intéressant de définir un système logique qui décrit d'une manière précise les relations qui servent à l'interprétation des termes.

Certains termes de type $A \Rightarrow B$ peuvent en fait déjà être interprétés comme des relations de A dans B : ceux dont la sémantique ne contient que des éléments de la forme $([\alpha], \beta)$. Intuitivement, ceci revient à exiger que la fonction représentée évalue exactement une fois son argument : on dit alors que le programme ou la preuve est *linéaire*. L'exemple canonique est l'identité $\lambda x\, x : A \Rightarrow A$. Plus intéressant : si on fixe un terme clos $t : A$, on peut définir l'opérateur d'application à t, $\lambda x\, (x)\, t : (A \Rightarrow B) \Rightarrow B$, qui est linéaire en la « fonction » x, en accord avec la remarque 3.2.

La décomposition de l'application vue plus haut permet maintenant de considérer n'importe quel programme comme un programme linéaire. Sémantiquement, on raffine l'implication $A \Rightarrow B$ en $!A \multimap B$: l'*implication linéaire* $A \multimap B$ désigne le type des relations de A dans B ; la modalité (connecteur unaire) « ! », appellée *bien-sûr*, correspond à la construction de l'ensemble des multiensembles finis. De même, une preuve en logique linéaire du séquent $A_1, \ldots, A_n \vdash B$ dénotera une relation de $A_1 \times \cdots \times A_n$ dans B.

Remarque 3.10. *Contrairement à la situation habituelle en logique, on n'aura plus de moyen générique de contracter ou d'affaiblir les hypothèses : en logique minimale, il était facile de construire une preuve de $A \vdash B$ à partir d'une preuve de $\vdash B$ (on n'utilise pas l'hypothèse A) ou d'une preuve de $A, A \vdash B$ (on utilise deux fois la même hypothèse). Mais on ne voit pas*

de manière canonique de construire une relation de A dans B à partir d'une partie de B, ou bien d'une relation de $A \times A$ dans B. Le choix des contextes des règles est donc particulièrement significatif, ce qui fait qu'on présente souvent la logique linéaire comme une logique sensible aux s : une preuve doit utiliser chaque hypothèse exactement une fois.

La construction de la logique linéaire est guidée par la décomposition attendue de la logique minimale : une preuve de $A_1, \ldots, A_n \vdash B$ en logique minimale, se traduira en une preuve de $!A_1, \ldots, !A_n \vdash B$ en logique linéaire. Comme on construit le système en partant du modèle dénotationnel, et par souci de concision, on annotera directement les règles avec l'opération correspondante sur la sémantique. On omet les annotations laissées inchangées.

Considérons la règle la plus élémentaire :

$$\frac{}{A^\alpha \vdash A^\alpha}\ (\text{Id})\ .$$

Elle construit la relation diagonale sur A : c'est l'identité linéaire. Remarquez que son contexte est nécessairement vide.

Rappelons ici le rôle primitif de la composition des relations souligné dans la relecture de l'application. En logique, la composition est représentée par la règle de coupure $\frac{A \vdash B \qquad B \vdash C}{A \vdash C}\ (\text{Cut})$ et plus généralement :

$$\frac{\Gamma \vdash B^\beta \qquad \Delta, B^\beta \vdash C}{\Gamma, \Delta \vdash C}\ (\text{Cut})\ .$$

Cette règle est au cœur du calcul des séquents imaginé par Gentzen. En l'incluant dans un système logique, on peut y concentrer tous les détours et indirections des démonstrations, car elle permet de remplacer les règles d'élimination des connecteurs par des règles d'introduction *dans les hypothèses*. On l'adopte donc en logique linéaire. Ainsi, on aura pour l'implication linéaire :

$$\frac{\Gamma, A^\alpha \vdash B^\beta}{\Gamma \vdash A \multimap B^{(\alpha,\beta)}}\ (\vdash\multimap) \quad \text{et} \quad \frac{\Gamma \vdash A^\alpha \qquad \Delta, B^\beta \vdash C}{\Gamma, \Delta, A \multimap B^{(\alpha,\beta)} \vdash C}\ (\multimap\vdash)\ .$$

La première est exactement de la même nature que $\langle\Rightarrow\rangle$, tandis que la seconde introduit une implication *à gauche*. Elle permet, avec la coupure de dériver :

$$\dfrac{\Gamma \vdash A \multimap B \qquad \dfrac{\Gamma \vdash A \qquad \dfrac{}{\Gamma, B \vdash B}\ (\text{Id})}{\Gamma, A \multimap B \vdash B}\ (\multimap\vdash)}{\Gamma, \Delta \vdash B}\ (\text{Cut})$$

qui correspondrait à une règle d'élimination, similaire à $[\![@]\!]$.

Les règles de la modalité bien-sûr représentent les différentes opérations sur les multiensembles finis qu'on a utilisées dans la sémantique du λ-calcul.

$$\frac{!\Gamma \vdash A}{!\Gamma \vdash !A}\,(\vdash !) \qquad \frac{\Gamma, A^{\alpha} \vdash B}{\Gamma, !A^{[\alpha]} \vdash B}\,(!\vdash) \qquad \frac{\Gamma \vdash B}{\Gamma, !A^{[]} \vdash B}\,(!w)$$
$$\frac{\Gamma, !A^{\overline{\alpha}}, !A^{\overline{\alpha}'} \vdash B}{\Gamma, !A^{\overline{\alpha}+\overline{\alpha}'} \vdash B}\,(!c)$$

On a déjà donné la sémantique $[\![!]\!]$ de la règle de *promotion* $(\vdash !)$ au début de cette section. Il est crucial ici de forcer le contexte de la prémisse à être uniquement constitué de formules *réplicables*, c'est-à-dire de la forme $!B$. En prenant l'analogie des ressources : on transforme ici une ressource unique en un argument susceptible d'être dupliqué ou effacé ; ce à partir de quoi on le construit est donc soumis à la même exigence. La règle de *déréliction* $(! \vdash)$ correspond à l'utilisation d'une instance d'argument. Les deux dernières sont dites structurelles, car elles n'introduisent pas de connecteurs. L'affaiblissement $(!w \vdash)$ permet d'introduire une ressource sans l'utiliser : avec la déréliction, on peut donc traduire $\dfrac{}{\Gamma, A \vdash A}\,(\text{Hyp})$ en conservant la sémantique :

$$\cfrac{\cfrac{\cfrac{}{A \vdash A}\,(\text{Id})}{!A \vdash A}\,(!\vdash)}{!\Gamma, !A \vdash A}\,(!w). \tag{1}$$

La contraction $(!c \vdash)$ sépare en deux « paquets » les utilisations possibles d'une ressource, comme par exemple dans la traduction de l'application :

$$\cfrac{\cfrac{!\Gamma \vdash !A \multimap B \qquad \cfrac{\cfrac{!\Gamma \vdash A}{!\Gamma \vdash !A}\,(\vdash !) \qquad \cfrac{}{B \vdash B}\,(\text{Id})}{!\Gamma, !A \multimap B \vdash B}\,(\multimap\vdash)}{!\Gamma, !\Gamma \vdash B}\,(\text{Cut})}{!\Gamma \vdash B}\,(!c). \tag{2}$$

Quid de la conjonction dans ce cadre ? L'union disjointe $A + B$ des ensembles reste appropriée pour interpréter les couples : une relation s de A dans $B_1 + B_2$ est univoquement définie par ses *projections* $\pi_i s = \{(\alpha, \beta);\ (\alpha, (i, \beta)) \in s\}$ pour $i \in \{1, 2\}$. Une caractéristique habituelle de la conjonction est cependant d'être adjointe à l'implication : $(A \wedge B) \Rightarrow C \cong A \Rightarrow (B \Rightarrow C)$. Cette propriété échoue pour $+$ avec l'implication linéaire : on a plutôt $(A + B) \multimap C \cong (A \multimap C) + (B \multimap C)$. C'est en fait le produit cartésien des ensembles qui convient.

Remarque 3.11. *C'est exactement la même chose qu'en algèbre linéaire : on peut bien définir l'espace des couples, mais une application bilinéaire* $A \multimap (B \multimap C)$ ne dépend pas linéairement *du couple de ses arguments ; on doit plutôt la voir comme une application linéaire* $(A \otimes B) \multimap C$*, où* $A \otimes B$ *désigne le* produit tensoriel. *Notez d'ailleurs qu'une base de l'espace des couples s'obtient en faisant l'union disjointe de bases de* A *et* B*, tandis qu'une base du tenseur s'obtient comme un produit de bases.*

On obtient donc *deux conjonctions* en logique linéaire : une additive & (appelée *avec*) pour l'espace des couples, interprétée par $+$; et une multiplicative $\otimes$, adjointe à l'implication linéaire, interprétée par $\times$. Les règles du & sont les mêmes que celles qu'on avait données pour $\wedge$:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}\,(\vdash\&) \quad \text{et} \quad \frac{\Gamma, A_i \vdash B}{\Gamma, A_1 \,\&\, A_2 \vdash B}\,(\&_i\vdash)\ .$$

Celles du tenseur diffèrent par la gestion des contextes :

$$\frac{\Gamma \vdash A^{\alpha} \qquad \Delta \vdash B^{\beta}}{\Gamma, \Delta \vdash A \otimes B^{(\alpha,\beta)}}\,(\vdash\otimes) \quad \text{et} \quad \frac{\Gamma, A^{\alpha}, B^{\beta} \vdash C}{\Gamma, A \otimes B^{(\alpha,\beta)} \vdash C}\,(\otimes\vdash)\ .$$

Notez que, si on pouvait librement affaiblir ou contracter les hypothèses, les règles de & et celles de $\otimes$ définiraient le même connecteur.

Définition 3.12. *Les formules de la logique linéaire sont :*

$$A, B, C, \ldots ::= X \mid A \multimap B \mid A \otimes B \mid A \,\&\, B \mid !A$$

et ses preuves sont celles construites avec les règles (Id), (Cut), $(\vdash \multimap)$, $(\multimap \vdash)$, $(\vdash \&)$, $(\& \vdash)$, $(\vdash \otimes)$, $(\otimes \vdash)$, $(\vdash !)$, $(! \vdash)$, $(!w)$ *et* $(!c \vdash)$.

On peut alors traduire la logique minimale dans la logique linéaire, *en préservant la sémantique et la prouvabilité.*

Définition 3.13. *À toute dérivation de typage en* λ*-calcul, de conclusion* $\Gamma \vdash A$*, on associe une preuve de la logique linéaire de conclusion* $!\Gamma \vdash A$*, de la manière suivante :*

- *la règle* (Var) *est traduite comme en (1) ;*
- *la règle* (λ) *est traduite par la règle* $(\vdash \multimap)$ *;*
- *la règle* (App) *est traduite comme en (2) ;*
- *la règle* (Cpl) *est traduite par* $(\vdash \&)$ *;*
- *la règle* (π_i) *est traduite par*

$$\dfrac{!\Gamma \vdash A_1 \,\&\, A_2 \qquad \dfrac{\dfrac{}{A_i \vdash A_i}\,(Id)}{A_1 \,\&\, A_2 \vdash A_i}\,(\&_i\vdash)}{!\Gamma \vdash A_i}\,(Cut)$$

Propriété 3.14. *La sémantique d'un* λ*-terme typé est celle de la preuve associée en logique linéaire. Par ailleurs, si* A *et les formules de* Γ *sont des formules de la logique minimale, alors* $\Gamma \vdash A$ *est prouvable en logique minimale si et seulement si* $!\Gamma \vdash A$ *est prouvable en logique linéaire.*

La logique linéaire peut donc être vue statiquement comme un système plus fin que la logique minimale. En fait, elle décompose également la dynamique du λ-calcul donnée par la β-réduction. Pour le voir, il faut décrire le procédé de calcul sur les preuves linéaires. Comme chez Gentzen, il s'agit d'éliminer les coupures.

On ne donne pas ici toutes les règles de réduction, mais seulement quelques exemples emblématiques en table 5. L'idée générale est qu'une coupure entre

$$\dfrac{\dfrac{}{A \vdash A}\,(\mathrm{Id}) \qquad \overset{\pi}{\nabla}\;\; \Gamma, A \vdash B}{\Gamma, A \vdash B}\,(\mathrm{Cut}) \quad \rightsquigarrow \quad \overset{\pi}{\nabla}\;\; \Gamma, A \vdash B$$

$$\dfrac{\dfrac{\overset{\pi_1}{\nabla}\;\; \Gamma_1 \vdash A_1 \qquad \overset{\pi_2}{\nabla}\;\; \Gamma_2 \vdash A_2}{\Gamma_1, \Gamma_2 \vdash A_1 \otimes A_2}\,(\vdash\otimes) \qquad \dfrac{\overset{\rho}{\nabla}\;\; \Delta, A_1, A_2 \vdash B}{\Delta, A_1 \otimes A_2 \vdash B}\,(\otimes\vdash)}{\Gamma_1, \Gamma_2, \Delta \vdash B}\,(\mathrm{Cut}) \quad \rightsquigarrow \quad \dfrac{\overset{\pi_1}{\nabla}\;\; \Gamma_1 \vdash A_1 \qquad \dfrac{\overset{\pi_2}{\nabla}\;\; \Gamma_2 \vdash A_2 \qquad \overset{\rho}{\nabla}\;\; \Delta, A_1, A_2 \vdash B}{\Gamma_2, \Delta, A_1 \vdash B}\,(\mathrm{Cut})}{\Gamma_1, \Gamma_2, \Delta \vdash B}\,(\mathrm{Cut})$$

$$\dfrac{\overset{\pi}{\nabla}\;\; \Gamma \vdash A \qquad \dfrac{\overset{\rho}{\nabla}\;\; \Delta, A, B, C \vdash D}{\Delta, A, B \otimes C \vdash D}\,(\otimes\vdash)}{\Gamma, \Delta, B \otimes C \vdash D}\,(\mathrm{Cut}) \quad \rightsquigarrow \quad \dfrac{\overset{\pi}{\nabla}\;\; \Gamma \vdash A \qquad \overset{\rho}{\nabla}\;\; \Delta, A, B, C \vdash D}{\Gamma, \Delta, B, C \vdash D}\,(\mathrm{Cut}) \quad \dfrac{}{\Gamma, \Delta, B \otimes C \vdash D}\,(\otimes\vdash)$$

$$\dfrac{\dfrac{\overset{\pi}{\nabla}\;\; !\Gamma \vdash A}{!\Gamma \vdash !A}\,(\vdash !) \qquad \dfrac{\overset{\rho}{\nabla}\;\; \Delta \vdash B}{\Delta, !A \vdash B}\,(!w)}{!\Gamma, \Delta \vdash B}\,(\mathrm{Cut}) \quad \rightsquigarrow \quad \dfrac{\overset{\rho}{\nabla}\;\; \Delta \vdash B}{!\Gamma, \Delta \vdash B}\,(!w)$$

$$\dfrac{\dfrac{\overset{\pi}{\nabla}\;\; !\Gamma \vdash A}{!\Gamma \vdash !A}\,(\vdash !) \qquad \dfrac{\overset{\rho}{\nabla}\;\; \Delta, A \vdash B}{\Delta, !A \vdash B}\,(!\vdash)}{!\Gamma, \Delta \vdash B}\,(\mathrm{Cut}) \quad \rightsquigarrow \quad \dfrac{\overset{\pi}{\nabla}\;\; !\Gamma \vdash A \qquad \overset{\rho}{\nabla}\;\; \Delta, A \vdash B}{!\Gamma, \Delta \vdash B}\,(\mathrm{Cut})$$

$$\dfrac{\dfrac{\overset{\pi}{\nabla}\;\; !\Gamma \vdash A}{!\Gamma \vdash !A}\,(\vdash !) \qquad \dfrac{\overset{\rho}{\nabla}\;\; \Delta, !A, !A \vdash B}{\Delta, !A \vdash B}\,(!c)}{!\Gamma, \Delta \vdash B}\,(\mathrm{Cut}) \quad \rightsquigarrow \quad \dfrac{\dfrac{\overset{\pi}{\nabla}\;\; !\Gamma \vdash A}{!\Gamma \vdash !A}\,(\vdash !) \qquad \dfrac{\dfrac{\overset{\pi}{\nabla}\;\; !\Gamma \vdash A}{!\Gamma \vdash !A}\,(\vdash !) \qquad \overset{\rho}{\nabla}\;\; \Delta, !A, !A \vdash B}{!\Gamma, \Delta, !A \vdash B}\,(\mathrm{Cut})}{!\Gamma, !\Gamma, \Delta \vdash B}\,(\mathrm{Cut}) \quad \dfrac{}{!\Gamma, \Delta \vdash B}\,(!c)$$

TABLE 5 – Élimination des coupures en logique linéaire intuitionniste (exemples).

une règle gauche et une règle droite portant sur la formule coupée s'élimine en créant des coupures sur les sous formules. Dans le cas d'une coupure sur l'identité, on se contente de simplifier. Dans le cas où la formule coupée n'est pas introduite par une règle, on fait remonter la coupure dans le contexte. Le point intéressant concerne les coupures sur la promotion, pour lesquelles la réduction reflète l'intuition : l'affaiblissement efface l'argument ; la contraction le duplique ; la déréliction le « lit ».

Théorème 3.15. *Toute preuve de la logique linéaire admet une unique forme normale par élimination des coupures. De plus, l'élimination des coupures simule la β-réduction : la forme normale de la traduction d'un λ-terme est la traduction de sa forme β-normale.* [10]

On n'a en fait raconté ici que le début de l'histoire. D'abord, on aurait pu remarquer que l'union disjointe des ensembles permet non seulement de représenter des couples de relations, mais aussi les relations définies par cas sur l'union disjointe. On introduit ainsi un connecteur de disjonction $\oplus$ qui fonctionne comme une somme directe (on peut montrer qu'il n'y a rien de tel pour les multirelations). Comme pour la conjonction, il existe une deuxième version de la disjonction, notée $\parr$: celle-ci est associée à l'implication linéaire, d'une manière similaire à l'équivalence entre $A \Rightarrow B$ et $\neg A \vee B$, *qui n'est valable qu'en logique classique.* Pour la faire apparaître, il faut donc introduire une forme linéaire de la négation classique $A^\perp$ telle que $A \multimap B = A^\perp \parr B$.

Et c'est d'ailleurs le principal succès de la logique linéaire que d'autoriser l'introduction d'une négation *involutive* ($A^{\perp\perp} = A$) tout en conservant la sémantique dénotationnelle et les propriétés syntaxiques qu'on vient de survoler. En effet, le système présenté ici n'est que la version intuitionniste de la logique linéaire : en calcul des séquents, le passage de la logique minimale à la logique classique ne se fait pas en ajoutant une règle pour le raisonnement par l'absurde, mais en autorisant les séquents à comporter un nombre arbitraire de conclusions, ce qui établie une symétrie entre hypothèses et conclusions. Or on peut très naturellement reprendre les constructions précédentes en considérant cette fois des séquents symétriques, là où les multirelations forçaient à distinguer le type de retour des fonctions.

On conserve même la propriété 3.14 en l'état. Avec une autre traduction pour l'implication, on peut par ailleurs obtenir une caractérisation de la prouvabilité en logique classique : on y reviendra en section 5.2. La section suivante propose une autre piste, basée sur une interprétation calculatoire du raisonnement par l'absurde.

10. En fait il faut quotienter par une équivalence structurelle sur les preuves de la logique linéaire : l'ordre dans lequel on enchaîne les contractions et les affaiblissements dans le contexte n'est pas pertinent.

4 De vraies démonstrations et de vrais programmes

Pour l'instant, on n'a considéré que le calcul propositionnel, de surcroît privé de la formule $\bot$ qui représente l'absurde. On peut légitimement se demander le rapport qu'il y a entre ce système minimaliste et les véritables démonstrations mathématiques. C'est l'étude fine du système minimal qui nous permet de répondre à cette inquiétude, car on dispose maintenant des outils nécessaires pour combler les deux lacunes sus-nommées. Dans cette partie, on va donc voir comment sortir du cadre purement propositionnel pour s'attaquer à de vrais énoncés et démonstrations. On procèdera par étapes :

- l'introduction du *polymorphisme*, quantification sur les types, permet d'écrire des formules qui caractérisent des structures de données ;
- l'introduction de quantification du premier ordre, et donc le passage au calcul des prédicats, permet d'écrire des types suffisamment expressifs pour spécifier le comportement calculatoire des termes ;
- l'introduction d'*opérateurs de contrôle* dans le calcul permet de donner un sens calculatoire aux modes de raisonnements de la logique classique comme le raisonnement par l'absurde.

Ces différentes extensions, qui sont en fait assez indépendantes, consistent donc à enrichir la correspondance de Curry-Howard en jouant sur deux tableaux : l'expressivité du langage logique (pour la quantification) et l'expressivité du modèle de calcul (pour la logique classique). On verra dans la section 4.3 comment ces extensions se combinent dans un même système où peuvent donc s'étudier de vraies démonstrations et de vrais programmes.

4.1 Données et quantification

Comme déjà suggéré dans la remarque 2.8, le λ-calcul pur est Turing-puissant, on peut y représenter toutes les fonctions calculables. On ne va donc pas avoir besoin d'étendre le langage pour y représenter des données, mais on peut se poser la question du statut logique des données. Existe-t-il une formule logique qui définit ce qu'est un entier de Church ? Comme on va le voir ici, c'est l'introduction des quantificateurs dans le langage logique qui va nous permettre de répondre positivement.

Définition 4.1. *On se donne un ensemble de variables du premier ordre, notées dans la suite par les lettres $x, y, z \ldots$ et un ensemble de variables du second ordre, notées $X, Y, Z \ldots$, chacune munie d'une arité fixée. Les formules de l'arithmétique du second ordre sont définies par la grammaire suivante :*

$$\begin{array}{ll} \textit{individus :} & t, u, \ldots ::= x \mid 0 \mid S(t) \mid t + u \mid t \times u \\ \textit{énoncés :} & A, B, \ldots ::= X(t_1, \ldots, t_k) \mid \bot \mid A \Rightarrow B \mid \forall x\, A \mid \forall X\, A \end{array}$$

$$\frac{\Gamma \vdash M : A \qquad x \notin \Gamma}{\Gamma \vdash M : \forall x\, A}\ \langle\forall 1\rangle \qquad \frac{\Gamma \vdash M : \forall x\, A}{\Gamma \vdash M : A[t/x]}\ [\forall 1]$$

$$\frac{\Gamma \vdash M : A \qquad X \notin \Gamma}{\Gamma \vdash M : \forall X\, A}\ \langle\forall 2\rangle \qquad \frac{\Gamma \vdash M : \forall X\, A}{\Gamma \vdash M : A[B[t_1, \ldots, t_k]/X(t_1, \ldots, t_k)]}\ [\forall 2]$$

Dans la règle $[\forall 2]$, X est un variable propositionnelle d'arité k et B est un énoncé à k paramètres ; $A[B[t_1, \ldots, t_k]/X(t_1, \ldots, t_k)]$ est l'énoncé A dans lequel chaque occurrence de X, appliquée à une famille $t_1, \ldots, t_k$, est remplacée par l'énoncé B avec $t_1, \ldots, t_k$ comme valeur des paramètres.

TABLE 6 – Quantification dans le typage

Les termes d'individus sont des expressions qui représentent des éléments d'une structure que l'on décrit. Comme notre langage est celui de l'arithmétique (le terme $S(t)$ représente le *successeur* de t, c'est-à-dire $t + 1$, comme dans le formalisme de l'arithmétique de Peano), cette structure contiendra en particulier des entiers.

Les énoncés atomiques $X(t_1, \ldots, t_n)$ sont à présent paramétrés par des individus, ils représentent des prédicats et relations sur la structure considérée. On n'introduit que les quantificateurs universels dans un souci de minimalité, comme pour les autres connecteurs : $\exists x\, A$ est représenté par $\neg\forall x\, \neg A$.

Définition 4.2. *Les règles de typage des termes du λ-calcul par les énoncés de l'arithmétique du second ordre sont celles du λ-calcul simplement typé augmentées des règles de quantification de la table 6.*

À première vue, il semble que ces règles ne font pas grand chose pour donner une interprétation calculatoire des quantificateurs : elles permettent de les introduire et de les éliminer mais elles n'affectent pas les termes. On va voir qu'il n'en est rien.

4.1.1 Second ordre : le polymorphisme

On a vu dans la remarque 2.8 qu'il est possible de coder un entier n par un itérateur $\underline{n} := \lambda f\, \lambda x\, (f)^n\, x$ qui prend en arguments une fonction f et une entrée x et applique n fois f à x. Un terme de la forme $\underline{n}$ est appelé *entier de Church*. Il est facile de vérifier que pour n'importe quel type A, le terme $\underline{n}$ a le type attendu $(A \Rightarrow A) \Rightarrow (A \Rightarrow A)$: il prend en argument une fonction de A dans A et l'itère n fois. Le problème de ce type est qu'il n'est pas canonique puisqu'il dépend de A : on ne peut pas typer un entier de Church sans savoir à quelle fonction il s'appliquera, de plus donner un type à un entier force le type auquel on l'appliquera, ainsi on ne peut pas écrire une fonction qui utiliserait son argument entier dans deux contextes qui demanderaient des types différents.

La quantification du second ordre apporte la solution à ce problème par le *polymorphisme*, qui permet de rendre compte dans les types du fait que le

terme $\underline{n}$ est typé $(A \Rightarrow A) \Rightarrow (A \Rightarrow A)$ pour tous les choix possibles du type A. En effet, grâce à la règle $\langle \forall 2 \rangle$ on a maintenant le typage

$$\vdash \underline{n} : \forall X \big((X \Rightarrow X) \Rightarrow (X \Rightarrow X) \big)$$

Ceci fait, on peut construire les primitives du calcul avec les entiers en définissant les éléments nécessaires dans le monde quantifié :

$$\begin{aligned}
\mathbb{N} &\coloneqq \forall X \big((X \Rightarrow X) \Rightarrow (X \Rightarrow X) \big) && \text{type des entiers} \\
0 &\coloneqq \lambda f\, \lambda x\, x && \text{constante zéro} \\
\boldsymbol{S}N &\coloneqq \lambda f\, \lambda x\, (f)\, ((N)\, f)\, x && \text{fonction successeur} \\
\boldsymbol{R}(M,N,P) &\coloneqq ((M)\, N)\, P && \text{itérateur}
\end{aligned}$$

On vérifie sans peine que ces définitions sont bien typées :

$$\frac{}{\Gamma \vdash 0 : \mathbb{N}} \qquad \frac{\Gamma \vdash N : \mathbb{N}}{\Gamma \vdash \boldsymbol{S}N : \mathbb{N}} \qquad \frac{\Gamma \vdash M : \mathbb{N} \qquad \Gamma \vdash N : A \qquad \Gamma \vdash P : A \Rightarrow A}{\Gamma \vdash \boldsymbol{R}(M,N,P) : A}$$

de plus on a les règles de calcul suivantes :

$$\boldsymbol{R}(0,N,P) \to_\beta P \qquad \boldsymbol{R}(\boldsymbol{S}M,N,P) \to_\beta (N)\, \boldsymbol{R}(M,N,P)$$

qui signifient que $\boldsymbol{R}(M,N,P)$ est un récurseur qui applique M fois N à P. Mieux encore, le type $\mathbb{N}$ identifie exactement [11] les entiers de Church :

Théorème 4.3. *Les entiers de Church sont les seules formes normales de type* $\mathbb{N}$.

Exercice 4.4. *Montrez que le terme* $\underline{add}$ *de l'exercice 2.2 a le type* $\mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}$.

Selon les mêmes principes, on pourrait étendre le λ-calcul avec d'autres types : booléens, listes, arbres étiquetés, etc. On obtient ainsi un genre de langage de programmation simple mais assez expressif [12] qui est fonctionnel (puisque les fonctions peuvent être passées en argument à d'autres fonctions), pur (il n'y a pas de notion d'effet de bord, une expression a toujours la même valeur), avec garantie de terminaison pour tous les programmes.

11. Il y a ici un petit mensonge : il y a deux termes en forme normale qui représentent l'entier 1, à savoir $\lambda f\, \lambda x\, (f)\, x$ et $\lambda f\, f$. On peut éviter ce problème (au demeurant bénin) en posant à la place $\mathbb{N} = \forall X (X \Rightarrow (X \Rightarrow X) \Rightarrow X)$ et en adaptant la définition de 0, $\boldsymbol{S}$ et $\boldsymbol{R}$.

12. Avec les primitives utilisées il n'est pas très difficile de montrer que toute fonction récursive primitive peut être représentée par un terme de type $\mathbb{N} \Rightarrow \mathbb{N}$. En revanche, on n'a pas toutes les fonctions récursives puisque le langage, par construction, ne permet de définir que des fonctions totales. Pour cela il faudrait ajouter un combinateur de point fixe, et on obtiendrait le langage PCF [25] qui est un standard en sémantique des langages de programmation.

4.1.2 Premier ordre : la spécification

L'utilisation du polymorphisme permet de définir les entiers directement avec les termes du λ-calcul. L'ajout du langage de l'arithmétique et de la quantification du premier ordre permettent de les utiliser dans les énoncés qui servent à typer. Pour commencer, on peut raffiner le type des entiers, en donnant un type différent à chacun :

$$\mathbb{N}(t) \coloneqq \forall X \big(\forall x(X(x) \Rightarrow X(S(x))) \Rightarrow X(0) \Rightarrow X(t)\big)$$

Cet énoncé rappelle une forme générique du schéma de récurrence : pour tout énoncé X à un paramètre, à partir d'une preuve de l'étape de récurrence $\forall x(X(x) \Rightarrow X(S(x)))$ et d'une preuve du cas de base $X(0)$, on peut déduire une preuve de $X(t)$ pour le terme t concerné.

On interprètera ici l'énoncé $\mathbb{N}(t)$ comme « t est un entier » et l'utilisation de la logique du second ordre nous permet de poser qu'un entier est, par définition, un objet pour lequel le schéma de récurrence est applicable.

Ainsi la quantification n'est pas si anodine qu'il n'y paraissait initialement : le simple fait de montrer qu'un terme désigne un entier impose d'avoir une preuve qui est en fait un entier de Church qui implémente cet entier.

Exercice 4.5. *Montrez que $\boldsymbol{S}$, c'est-à-dire $\lambda n\, \lambda f\, \lambda x\, (f)\, ((n)\, f)\, x$, est typable par l'énoncé $\forall x(\mathbb{N}(x) \Rightarrow \mathbb{N}(S(x)))$.*

On pourrait chercher à montrer qu'un terme pour l'addition des entiers de Church, par exemple $\lambda m\, \lambda n\, \lambda f\, \lambda x\, ((n)\, f)\, ((m)\, f)\, x$, est typable par un énoncé comme $\forall x \forall y(\mathbb{N}(x) \Rightarrow \mathbb{N}(y) \Rightarrow \mathbb{N}(x + y))$. Avec les ingrédients dont nous disposons, ce n'est pas le cas : rien ne permet de faire le lien entre le successeur S et la somme parce qu'il n'y a pas d'axiomes, même pas d'égalité.

Grâce à la quantification du second ordre, il n'est pas nécessaire d'ajouter l'égalité au langage. On a recours à l'égalité dite de Leibniz, qui exprime que t et u sont égaux s'ils satisfont les mêmes propriétés :

$$(t = u) \coloneqq \forall X(X(t) \Rightarrow X(u)).$$

Exercice 4.6. *Construisez un terme de type $\forall x \forall y(\mathbb{N}(x) \Rightarrow \mathbb{N}(y) \Rightarrow \mathbb{N}(x+y))$ en supposant donnés un terme A_0 de type $\forall x(x = x + 0)$ et un terme A_S de type $\forall x \forall y(S(x + y) = x + S(y))$.*

On peut construire un terme correspondant à cette question en partant du terme pour l'addition des entiers de Church et en y insérant judicieusement une applications de A_0 et une de A_S. L'essentiel de la difficulté est de bien utiliser les règles d'introduction et d'élimination des quantificateurs.

Pour compléter cette implémentation du type de l'addition, il reste à préciser qui sont ces termes A_0 et A_S. Leurs types sont des axiomes qu'il est impossible de démontrer, puisque rien dans nos règles de déduction ne

parle de l'addition. On ajoute donc A_0 et A_S comme des constantes dans le langage de calcul et on ajoute des règles de typage pour leur donner le type voulu.

Afin de préserver l'interprétation calculatoire des démonstrations, il faut ensuite définir comment la β-réduction se comporte avec ces constantes. On peut vérifier que la cohérence du système de types est préservée si on pose les règles $(A_0)\, M \to_\beta M$ et $(A_S)\, M \to_\beta M$: ces règles de réduction préservent le type (on démontre par induction sur M que si $(A_0)\, M$ est typable d'un type A, alors M est typable du même type A) et la propriété de normalisation forte du théorème 2.13 est préservée.

De la même manière, on peut ensuite ajouter à notre système les autres axiomes de l'arithmétique, en trouvant pour chacun la règle de β-réduction appropriée. Le théorème de normalisation forte montre alors que toute démonstration se réduit à une démonstration sans coupure, c'est-à-dire que l'on retrouve une formulation du *Hauptsatz* évoqué précédemment. Néanmoins, on est toujours ici dans le système de la logique minimale. On va voir maintenant comment étendre notre interprétation à toute la logique classique.

4.2 Contrôle et logique classique

Dans l'interprétation BHK, le type $\bot$ doit être vide puisque l'absurde n'est pas démontrable. Pour sortir de cette impasse apparente, soyons plus subtils : on dit en fait qu'un programme de type $\bot$ ne peut pas renvoyer de résultat, sinon il témoignerait de l'absurde. En programmation, il y a des instructions très utiles qui ne répondent jamais, ce sont les instructions de *contrôle*, et en particulier les mécanismes de levée d'exceptions. Ainsi la fonction suivante, en syntaxe Java, est valable même s'il n'y a aucun moyen de fabriquer une valeur de type T :

```
T my_function (int x) {
    throw new Exception();
}
```

Dans un langage plus fonctionnel comme Caml, on a bien raise : exn -> 'a, c'est-à-dire que le type de retour d'une expression qui lève une exception peut être n'importe lequel. Ainsi, il apparaît crucial de faire la distinction entre les *valeurs* d'un type A et les *calculs* censés répondre au type A, dont le comportement ne se limite pas aux valeurs qu'ils répondent.

4.2.1 Le λ-calcul avec contrôle

Cette remarque sur les opérateurs de contrôle, due à Griffin [16], est la clé de l'interprétation calculatoire de la logique classique, mais sa mise en œuvre a un prix : quand on introduit le contrôle, il faut faire apparaître explicitement l'environnement d'exécution du programme, les continuations, et tout ce qu'il faut pour manipuler explicitement le flot de contrôle. Il est

donc nécessaire de préciser le modèle de calcul. Plusieurs choix sont possibles en partant du λ-calcul, ici on utilisera la machine de Krivine [20].

Définition 4.7. *Les termes du λc-calcul sont définis par la grammaire suivante :*

$$\begin{array}{ll} \textit{termes} & M, N, \ldots ::= x \mid \lambda x\, M \mid (M)\, N \mid \boldsymbol{cc} \mid \boldsymbol{k}_\pi \\ \textit{piles} & \pi ::= \varepsilon \mid M \cdot \pi \end{array}$$

On note Λ l'ensemble des termes clos et Π l'ensemble des piles de termes clos. Un exécutable *est le couple d'un terme clos M et d'une pile close π ; on le notera $M \star \pi$.*

L'exécution est la relation binaire transitive $\succ$ sur les exécutables engendrée par les règles suivantes :

$$\begin{array}{llll} \text{push :} & (M)\, N \star \pi \succ M \star N \cdot \pi & \text{save :} & \boldsymbol{cc} \star M \cdot \pi \succ M \star \boldsymbol{k}_\pi \cdot \pi \\ \text{pop :} & \lambda x\, M \star N \cdot \pi \succ M[N/x] \star \pi & \text{restore :} & \boldsymbol{k}_\pi \star M \cdot \pi' \succ M \star \pi \end{array}$$

Une pile est donc simplement une suite finie de termes clos, l'opération $M \cdot \pi$ représente l'ajout d'un terme M en tête d'une pile π. Un exécutable est la donnée d'un terme et d'une pile, et l'exécution précise comment le programme manipule sa pile.

Pour se rendre compte qu'on n'a pas fondamentalement changé de cadre, il faut se figurer un exécutable $M \star N_1 \cdot \cdots \cdot N_k \cdot \varepsilon$ comme une version du terme $(((M)\, N_1) \cdots)\, N_k$ où l'on a explicitement distingué le terme M de son contexte. Ainsi la paire de règles *push/pop* est une décomposition de la stratégie de réduction appelée *réduction de tête faible*, qui consiste à ne pas appliquer la β-réduction n'importe où dans un terme, mais uniquement à gauche d'une application et pas dans une abstraction.

Dans ce modèle, l'évaluation d'un terme M consiste à le réduire (selon la stratégie considérée) jusqu'à ce qu'il soit sous la forme d'une abstraction. La suite du calcul, c'est-à-dire la *continuation*, consistera à appliquer cette abstraction à des arguments pour poursuivre le calcul. Une continuation correpond donc formellement à une pile de termes et la paire de règle *save/restore* définit les primitives de contrôle :

- la règle *save* définit que $\boldsymbol{cc}$ exécute son argument M en lui passant en argument la continuation courante sous forme du terme $\boldsymbol{k}_\pi$, à la manière du *call-with-current-continuation* de Scheme ;
- la règle *restore*, quant à elle, signifie qu'une continuation $\boldsymbol{k}_\pi$, lorsqu'on cherche à l'appliquer à un argument M, oublie le contexte courant (c'est-à-dire la pile π') pour rétablir la pile d'arguments sauvegardée π.

Remarque 4.8. *Il faut ici souligner qu'il est nécessaire de choisir un ordre d'évaluation précis. La β-réduction générale n'est plus pertinente en présence de contrôle parce que le résultat du calcul d'un terme pourra être significativement différent selon de la stratégie employée. Considérons un terme*

$T \coloneqq (\boldsymbol{cc})\, \lambda k\, ((k)\, A)\, (k)\, B$ où A et B sont des termes clos. En évaluant à gauche de l'application $((k)\, A)\, (k)\, B$, on réduit T en A, mais en évaluant à droite on obtiendrait le terme B, ainsi le comportement de T est mal défini si la stratégie de réduction n'est pas précisée.

4.2.2 Typage en présence de contrôle

Dans le λ-calcul pur, on exprime la correction du système de typage par un argument de normalisation : si un terme M est typable, alors

- d'une part tous ses réduits sont typables avec le même type, c'est la préservation du type par réduction, en anglais *subject reduction* ;
- d'autre part toutes les suites de réductions de M finissent par atteindre un même terme irréductible, c'est la *normalisation forte*.

Dans un cadre où l'exécution d'un terme n'a de sens que dans un contexte particulier, ces notions ne sont pas applicables. En particulier, un terme de type $\bot$ ne rend pas la main, mais il doit quand même faire quelque chose de raisonnable étant donnée l'exécution en cours. La notion de correction dépend donc de ce que l'on attend globalement du comportement d'un exécutable. Ainsi, le fait qu'un terme M se comporte correctement pour un type A (on dira que M *réalise* A) dépend donc à présent d'un paramètre qui définit ce qu'est une exécution correcte.

Définition 4.9. *On appelle* observation *un ensemble $\perp\!\!\!\perp$ d'exécutables stable par anti-réduction, c'est-à-dire tel que pour tous exécutables e et e' tels que $e \succ e'$, si $e' \in \perp\!\!\!\perp$ alors $e \in \perp\!\!\!\perp$. On en déduit l'orthogonalité entre termes et piles :*

- *un terme M est orthogonal à une pile π, noté $M \perp\!\!\!\perp \pi$, si $M \star \pi \in \perp\!\!\!\perp$,*
- *l'orthogonal d'un ensemble P de piles est $P^{\perp\!\!\!\perp} \coloneqq \{M \mid \forall \pi \in P, M \perp\!\!\!\perp \pi\}$,*
- *l'orthogonal d'un ensemble de termes est défini de façon similaire.*

On appellera valeur de fausseté *un ensemble de piles et* valeur de vérité *un ensemble de termes qui est l'orthogonal d'un ensemble de piles.*

La terminaison de l'exécution est une observation possible, mais elle est loin d'être la plus intéressante, de plus on gagne en expressivité en permettant de varier les observations. On se rapproche ainsi de la notion de *test*, familière dans les calculs de processus [6], où il est bien connu que plusieurs formes de test sont pertinentes. Une pile est donc un test que l'on fait passer à un terme ; une valeur de fausseté est un ensemble de tests et une valeur de vérité est un ensemble de termes définis en fonction d'une batterie de tests.

Cette notion de valeur de vérité indique ce que l'on va faire maintenant : on interprètera les formules par des valeurs de vérité, qui maintenant ne sont plus des valeurs booléennes mais des ensembles de termes définis par une contrainte sémantique issue de $\perp\!\!\!\perp$, et on dira qu'un terme *réalise* un type A s'il appartient à la valeur de vérité de A.

Définition 4.10. *Soit $\perp\!\!\!\perp$ une observation. Une valuation est le choix, pour chaque variable propositionnelle X, d'une valeur de fausseté $[X]$. Étant donnée une valuation, on définit la valeur de fausseté d'une formule par induction :*

$$[A \Rightarrow B] \coloneqq \left\{ M \cdot \pi \ \mid M \in [A]^{\perp\!\!\!\perp}, \pi \in [B] \ \right\} \qquad [\bot] \coloneqq \Pi$$

On définit la valeur de vérité par $[\![A]\!] = [A]^{\perp\!\!\!\perp}$. On dit qu'un terme clos M réalise *un type A si $M \in [\![A]\!]$ et on note alors $M \Vdash A$.*

Notons qu'à cause de la contrainte de clôture de $\perp\!\!\!\perp$ par anti-réduction et de la règle *push*, la définition de $[A \Rightarrow B]$ revient essentiellement à dire que $M \Vdash A \Rightarrow B$ si pour tout $N \Vdash A$ on a $(M)\, N \Vdash B$.

Pour le cas de $\bot$, on dit exactement qu'un terme M se comporte bien pour le type $\bot$ s'il se comporte bien quel que soit le contexte dans lequel on l'exécute, ce qui revient à dire qu'il n'utilise pas son contexte. Ainsi, pour chaque exécutable $M \star \pi \in \perp\!\!\!\perp$, le terme $(\boldsymbol{k}_\pi)\, M$ est dans l'ensemble $[\![\bot]\!]$ puisque pour toute pile π' on a

$$(\boldsymbol{k}_\pi)\, M \star \pi' \succ_{push} \boldsymbol{k}_\pi \star M \cdot \pi' \succ_{restore} M \star \pi \ \in \perp\!\!\!\perp.$$

La correction du typage se formule maintenant par le théorème d'adéquation suivant, dont la démonstration découle directement des définitions. Cette notion de correction permet de justifier l'introduction de nouvelles règles de typage : on l'utilise en particulier pour typer la constante $\boldsymbol{cc}$.

Théorème 4.11. *On suppose choisies une observation $\perp\!\!\!\perp$ et une valuation $[\cdot]$. Si un jugement de typage $x_1 : A_1, \ldots, x_k : A_k \vdash M : B$ est dérivable, alors pour tous termes clos $N_1 \Vdash A_1, \ldots, N_k \Vdash A_k$ on a $M[N_1/x_1, \ldots, N_k/x_k] \Vdash B$.*

Théorème 4.12. *Pour toute observation $\perp\!\!\!\perp$ et tous types A et B, on a*

$$\boldsymbol{cc} \Vdash ((A \Rightarrow B) \Rightarrow A) \Rightarrow A.$$

Démonstration. Il s'agit de montrer que pour toute pile $\pi \in [((A \Rightarrow B) \Rightarrow A) \Rightarrow A]$ on a $\boldsymbol{cc} \perp\!\!\!\perp \pi$. Par définition, on a $\pi = M \cdot \pi_1$ avec $M \Vdash (A \Rightarrow B) \Rightarrow A$ et $\pi_1 \in [A]$. Par la règle *save* on a $\boldsymbol{cc} \star \pi \succ M \star \boldsymbol{k}_{\pi_1} \cdot \pi_1$.

Montrons que $\boldsymbol{k}_{\pi_1}$ réalise $A \Rightarrow B$. Pour cela, considérons donc une pile de $[A \Rightarrow B]$, nécessairement de la forme $N \cdot \pi_2$ avec $N \Vdash A$ et $\pi_2 \in [B]$. Par la règle *restore*, on a $\boldsymbol{k}_{\pi_1} \star N \cdot \pi_2 \succ N \star \pi_1$, or on a $\pi_1 \in [A]$, donc $N \perp\!\!\!\perp \pi_1$, et par anti-réduction on a $\boldsymbol{k}_{\pi_1} \perp\!\!\!\perp N \cdot \pi_2$. Ainsi on a bien $\boldsymbol{k}_{\pi_1} \Vdash A \Rightarrow B$.

On a alors $\boldsymbol{k}_{\pi_1} \cdot \pi_1 \in [(A \Rightarrow B) \Rightarrow A]$ d'où $M \perp\!\!\!\perp \boldsymbol{k}_{\pi_1} \cdot \pi_1$ par définition des valeurs de vérité, et par stabilité de $\perp\!\!\!\perp$ par anti-réduction on a $\boldsymbol{cc} \perp\!\!\!\perp \pi$. □

On peut donc, en préservant l'adéquation, étendre le système de typage en posant $\boldsymbol{cc} : ((A \Rightarrow B) \Rightarrow A) \Rightarrow A$. On ajoute ainsi la *loi de Peirce* déjà évoquée dans les exemples 2.2 et 2.14 comme un axiome, ce qui nous fait revenir à la logique classique. Notre système déductif a maintenant la puissance et l'expressivité de l'arithmétique classique du second ordre.

Au moyen de cette primitive de contrôle, on peut implémenter des structures de contrôle plus raffinées : traitement d'exceptions, *backtracking*, des formes de coroutines, etc. Chaque fois, il s'agira du pendant calculatoire d'un style de raisonnement non intuitionniste, pas forcément aussi puissants que la loi de Peirce : tiers-exclu, principe de Markov, etc.

4.3 Modèles de réalisabilité

Nous disposons donc à présent d'une logique capable d'exprimer des énoncés mathématiques et de les démontrer avec la puissance de la logique classique du second ordre. On peut donc y étudier des théories (arithmétique, théorie des ensembles...), mais a-t-on une notion de modèle pertinente ? La réponse est oui, mais il faut revenir sur le théorème d'adéquation du typage avec contrôle pour l'étendre aux quantificateurs.

Définition 4.13. *On suppose fixés un modèle $\mathcal{M}$ des axiomes de Peano (par exemple les entiers naturels...) et une observation $\perp\!\!\!\perp$. Une valuation consiste en*

- *un point $[x] \in \mathcal{M}$ pour chaque variable du premier ordre x,*
- *une fonction $[X] : \mathcal{M}^k \to \mathfrak{P}(\Pi)$ pour chaque variable du second ordre X d'arité k.*

On en déduit l'interprétation $[t]$ d'un terme t dans $\mathcal{M}$ de la façon standard. La valeur de fausseté des formules quantifiées est définie inductivement par

$$[X(t_1,\dots,t_k)] \coloneqq [X]([t_1],\dots,[t_k]) \quad [A \Rightarrow B] \coloneqq \{\ M \cdot \pi\ ; M \in [A]^{\perp\!\!\!\perp}, \pi \in [\pi]\}$$
$$[\forall x\, A] \coloneqq \bigcup_{[x]\in\mathcal{M}}\ [A] \qquad [\forall X\, A] \coloneqq \bigcup_{[X]\in\mathcal{M}^k\to\mathfrak{P}(P)}\ [A] \qquad [\bot] \coloneqq \Pi$$

Cette définition étend donc celle donnée précédemment. On peut y remarquer que les formules $\bot$ et $\forall X\, X$ ont la même valeur de fausseté (et donc de vérité) : c'est cohérent avec le fait que de $\bot$ on peut déduire tout ce que l'on veut et cela permet même de se passer de la constante $\bot$.

Définition 4.14. *Une* quasi-preuve *est un terme clos du λc calcul ne contenant aucun sous-terme de la forme $\boldsymbol{k}_\pi$. Un énoncé A est considéré comme* vrai *dans une valuation donnée s'il existe une quasi-preuve dans $[\![A]\!]$.*

C'est cette notion qui nous donne des modèles au sens traditionnel : une fois choisie une valuation, l'ensemble des formules vraies est une théorie complète, d'où on peut tirer un modèle.

Propriété 4.15. *Tout énoncé équationnel $\forall x_1 \dots \forall x_k (t = u)$ qui est vrai dans $\mathcal{M}$ est réalisé par $I := \lambda x\, x$, indépendamment du choix de $\perp\!\!\!\perp$ et de la valuation.*

Cette propriété n'implique pas que les énoncés vrais en réalisabilité sont les mêmes que ceux qui sont vrais dans $\mathcal{M}$: on ne parle ici que des équations, sans connecteurs logiques. Par exemple, on peut montrer que l'énoncé $\forall x\, \mathbb{N}(x)$ est faux en général. En effet, un terme M réalisant ce type, par définition de l'interprétation des quantificateurs, réaliserait le type $\mathbb{N}(t)$ pour tout terme t, en particulier il réaliserait le type de 0 et celui de 1. Il devrait donc se comporter, du point de vue de l'observation $\perp\!\!\!\perp$, à la fois comme l'entier 0 et comme l'entier 1, ce qui est impossible sauf pour des choix très particuliers de $\perp\!\!\!\perp$. Ceci signifie que la théorie des formules vraies dans une observation donnée décrit un modèle qui contient en général autre chose que des entiers. Ce qu'il contient précisément dépend du choix de l'observation $\perp\!\!\!\perp$.

Dans l'étude de la *réalisabilité classique*, on cherche à identifier quelles « instructions » il faut ajouter au calcul et quelles conditions associées il faut poser sur $\perp\!\!\!\perp$ pour réaliser certains axiomes, et ainsi, en étudiant le comportement des programmes obtenus, comprendre la *spécification* associée à certains théorèmes mathématiques classiques. Citons, à titre d'exemples :

- l'axiome du choix correspond à un mécanisme capable de produire des témoins d'existence à la demande, en usant éventuellement d'une forme de mémoire et de *backtracking* pour préserver sa cohérence ;
- en théorie des ensembles, la construction de *forcing* (qui permet de construire des modèles réfutant l'hypothèse du continu) peut être vue comme une transformation de programme par adjonction d'une mémoire globale où peuvent avoir lieu des effets.

On ne développera pas ici ces exemples, beaucoup trop techniques et exploratoires pour la portée de ce cours, mais le lecteur intéressé pourra y voir de plus près dans la littérature [19, 23].

5 Analyse des programmes par la linéarité

D'un côté, on a donc une analyse fine des démonstrations intutionnistes et, en conséquence, de la sémantique du λ-calcul. D'un autre côté, on a une interprétation calculatoire riche des démonstrations classiques. On va voir maintenant comment faire le lien entre ces deux directions.

5.1 Interprétation calculatoire des preuves linéaires

Notre histoire a commencé en remarquant que les démonstrations en déduction naturelle minimale ont exactement la même structure que les termes du λ-calcul simplement typé. L'analyse de la sémantique (relationnelle) nous

$$\frac{}{u{=}v \vdash u : X^\perp, v : X}\ (\text{Hyp}) \qquad \frac{P \vdash \Gamma, x_1 \cdots x_n : A \qquad Q \vdash x_1 \cdots x_n : A^\perp, \Delta}{(\boldsymbol{\nu} x_1 \cdots x_n)(P \mid Q) \vdash \Gamma, \Delta}\ (\text{Cut})$$

$$\frac{P \vdash \Gamma, x_1 \cdots x_n : A \qquad Q \vdash \Delta, y_1 \cdots y_n : B}{P \mid Q \vdash \Gamma, \Delta, x_1 \cdots x_n y_1 \cdots y_n : A \otimes B}\ (\vdash \otimes)$$

$$\frac{P \vdash \Gamma, x_1 \cdots x_n : A, y_1 \cdots y_n : B}{P \vdash \Gamma, x_1 \cdots x_n y_1 \cdots y_n : A \parr B}\ (\vdash \parr)$$

$$\frac{P \vdash \Gamma, x_1 \cdots x_n : A}{\bar{u}(x_1 \cdots x_n).P \vdash \Gamma, u : ?A}\ (\vdash ?) \qquad \frac{P \vdash ?\Gamma, x_1 \cdots x_n : A}{!u(x_1 \cdots x_n).P \vdash ?\Gamma, u : !A}\ (\vdash !)$$

$$\frac{P \vdash \Gamma, u : ?A, v : ?A}{P[w/u, v] \vdash \Gamma, w : ?A}\ (?c) \qquad \frac{P \vdash \Gamma}{P \vdash \Gamma, u : ?A}\ (?w)$$

TABLE 7 – Termes calculatoires pour la logique linéaire classique

a menés à introduire la linéarité, puis l'étude de la linéarité a révélé le système logique de plus « bas niveau » qu'est la logique linéaire. Voyons maintenant comment cette décomposition se traduit du côté calculatoire.

On pourrait commencer par attribuer des termes aux démonstrations de la logique linéaire intuitionniste, telle qu'elle est définie dans la la section 3.3. Cette approche fonctionne très bien : on obtient un λ-calcul enrichi d'informations sur la linéarité (calculatoire) de ses arguments.

Mais les choses les plus intéressantes arrivent quand on considère la logique linéaire complète, dans sa version symétrique. En effet, dans ce cas on peut déjà simplifier la structure de la logique en ne considérant que des séquents *monolatères* de la forme $\vdash A_1, \ldots, A_n$. On a alors uniquement les règles d'introduction des connecteurs à droite du symbole $\vdash$, et on reformule simplement la règle d'axiome pour qu'elle démontre $\vdash A^\perp, A$ et la règle de coupure pour qu'elle élimine une conclusion A face à une conclusion $A^\perp$. En contrepartie, il faut donc que chaque connecteur logique ait un dual (au sens des lois de De Morgan) ; pour la modalité ! on introduit alors la modalité ?, appelée *pourquoi-pas*, qui dénote le fait qu'une démonstration de $?A$ donne éventuellement plusieurs fois la conclusion $?A$.

Pour l'interprétation calculatoire, donnons tout de suite un langage :

Définition 5.1. *On se donne un ensemble de* noms*, désignés par les lettres u, v, x, y. Les termes du π-calcul fonctionnel sont définis par la grammaire*

$$P, Q ::= x{=}y \mid (P \mid Q) \mid (\boldsymbol{\nu} x_1 \cdots x_n)P \mid\ !u(x_1 \cdots x_n).P \mid\ \ \bar{u}(x_1 \cdots x_n).P$$

Dans les trois dernières constructions, les variables x_i sont liées, on peut donc les renommer librement tant qu'il n'y a pas de conflit avec les autres noms présents, comme le x dans $\lambda x\, M$. Ces termes sont attribués aux démonstrations de logique linéaire par les règles de la table 7.

Un nom représente intuitivement une occurence d'une formule atomique ou précédée d'une modalité. On a donc simplement représenté les démons-

trations par des termes et les occurences de formules par des suites de noms, en prenant déjà quelques libertés :

- La règle ($\vdash ⅋$) n'affecte pas le terme associé, ce qui est cohérent avec le fait que la virgule dans un séquent a le même sens qu'un ⅋.
- Les règles ($?w$) et ($?c$) permettent d'utiliser un nom un nombre quelconque de fois dans un terme ; c'est le pendant des affaiblissements et contractions qui sont implicites en λ-calcul.

Les connaisseurs reconnaîtront dans ce langage une variante sur le π-calcul, introduit par Milner [22] pour modéliser les systèmes concurrents, à peu près à l'époque où Girard introduisait la logique linéaire. L'exécution de ces termes suit effectivement le modèle du π-calcul :

- un terme $\bar{u}(x_1 \cdots x_n).P$ est interprété comme l'émission sur le nom u d'un signal contenant un n-uplet de noms $x_1 \cdots x_n$ spécifiques à P, suivi de l'exécution du terme P ;
- un terme $!u(x_1 \cdots x_n).P$ est interprété comme un récepteur qui, chaque fois qu'il reçoit un signal contenant des noms $y_1 \cdots y_n$ sur u, déclenche l'exécution d'une copie de P instanciée par ces noms.

La notation $(\boldsymbol{\nu} x_1 \cdots x_n)P$ siginifie que les noms $x_1 \cdots x_n$ sont privés dans P ; l'environnement de P ne peut pas les utiliser.

Définition 5.2. *Soit $\equiv$ l'équivalence sur les termes du π-calcul fonctionnel qui rend la construction $P \mid Q$ associative et commutative, qui identifie $x{=}y$ avec $y{=}x$ et qui contient les règles*

$$(\boldsymbol{\nu} x_1 \cdots x_n)P \mid Q \equiv (\boldsymbol{\nu} x_1 \cdots x_n)(P \mid Q) \qquad (x{=}y \mid P[x/z]) \equiv (x{=}y \mid P[y/z])$$
$$(\boldsymbol{\nu} x_1 \cdots x_n)(\boldsymbol{\nu} y_1 \cdots y_m)P \equiv (\boldsymbol{\nu} x_1 \cdots x_n y_1 \cdots y_m)P$$

en supposant que les x_i n'apparaissent pas dans Q. L'exécution est la relation $\succ$ définie à $\equiv$ près sur les termes par la règle

$$\bar{u}(x_1 \cdots x_n).P \mid !u(x_1 \cdots x_n).Q \succ (\boldsymbol{\nu} x_1 \cdots x_n)(P \mid Q) \mid !u(x_1 \cdots x_n).Q$$

On n'étudiera pas en détail ici le lien entre cette notion d'exécution et l'élimination des coupures dans les démonstrations, mais dans les grandes lignes, voici ce que l'on observe :

- Le fait que $\mid$ soit une loi de monoide commutatif rend compte de commutations admissibles entre les règles logiques.
- La règle de renommage incluse dans $\equiv$ garantit que les égaliseurs $x{=}y$ rendent les noms x et y équivalents. Logiquement, elle souligne la neutralité de la règle d'axiome pour l'élimination des coupures.
- La règle sur la portée des noms introduits par $(\boldsymbol{\nu} x_1 \cdots x_n)$ correspond à l'élimination des coupures entre connecteurs $\otimes$ et ⅋.
- La réduction $\succ$ rend compte des règles d'élimination des coupures pour les modalités, ce sont les règles les plus significatives dans le contenu calculatoire de démonstrations.

L'étude de ce calcul peut se faire par les outils de la théorie de la concurrence ; sans typage il a d'ailleurs une expressivité voisine de celle du π-calcul, une correspondance plus complète s'obtiendrait en y ajoutant une représentation des connecteurs $\oplus$ et &. Typé, il correspond précisement à la logique linéaire, qui est un raffinement de la logique intuitionniste, donc fondamentalement quelque chose de fonctionnel. Pousser plus précisément la correspondance avec les démonstrations nécessiterait l'introduction des *réseaux de démonstration* [13], un formalisme graphique pour les démonstrations de logique linéaire que nous avons passé sous silence ici...

5.2 Analyse linéaire de la logique classique

La logique linéaire apparaît comme une décomposition de la logique intuitionniste, en remarquant que dans une démonstration intuitionniste de $A \Rightarrow B$ on a exactement une démonstration de B mais que l'hypothèse A est utilisée un nombre quelconque de fois. On obtient ainsi la décomposition $(A \Rightarrow B) = (!A \multimap B)$ où la modalité ! rend compte de la multiplicité des utilisations de A. La formulation de l'implication intuitionniste $X \multimap Y$ comme une négation et une disjonction linéaires $X^\perp \parr Y$ fait apparaître l'implication intuitionniste comme $(A \Rightarrow B) = (?A^\perp \parr B)$, en utilisant la modalité ? introduite plus haut. Cette modalité ? nous permet de dépasser l'ambition initiale d'analyser la logique intuitionniste : elle nous permet de traduire en logique linéaire la logique *classique* et d'en donner ainsi une sémantique intéressante.

C'est une heureuse surprise, car on croyait (du moins jusqu'à la remarque de Griffin sur le typage des opérateurs de contrôle évoquée dans la section 4.2) que la procédure d'élimination des coupures de la logique classique devait nécessairement mener à des sémantiques dégénérées. La raison est donnée par ce que l'on appelle dans le jargon la *paire critique de Lafont* :

$$\dfrac{\dfrac{\pi_1}{\Gamma \vdash}}{\Gamma \vdash \Delta}\,(\mathrm{Wk}) \quad \rightsquigarrow \quad \dfrac{\dfrac{\dfrac{\pi_1}{\Gamma \vdash}\,(\mathrm{Wk})}{\Gamma \vdash A} \quad \dfrac{\dfrac{\pi_2}{\vdash \Delta}\,(\mathrm{Wk})}{A \vdash \Delta}}{\Gamma \vdash \Delta}\,(\mathrm{Cut}) \quad \rightsquigarrow \quad \dfrac{\dfrac{\pi_2}{\vdash \Delta}}{\Gamma \vdash \Delta}\,(\mathrm{Wk})$$

où (Wk) désigne la règle d'affaiblissement de la logique classique. En effet, comme π_1 ne fournit la conclusion A qu'au moyen d'un affaiblissement, eliminer la coupure contre π_2 qui utilise A en hypothèse consiste simplement à éliminer π_2 pour déduire sa conclusion Δ par affaiblissement. Mais comme la logique classique est symétrique, le même argument justifie que l'on peut aussi éliminer π_1 pour ne garder que π_2 ! Ainsi, dans une sémantique invariante par élimination des coupures, π_1 et π_2 doivent avoir la même interprétation. En d'autres termes, toutes les démonstrations d'un même énoncé doivent être identifiées : on a une interprétation dégénérée.

Les modalités de la logique linéaire permettent de régler ce problème en résolvant ce genre de conflit. L'idée est d'utiliser une variation dans la

décomposition de l'implication de manière à permettre les affaiblissements et contractions des deux côtés d'une implication.

Théorème 5.3. *On note $(\cdot)^T$ la traduction des formules et séquents de logique classique en logique linéaire telle que*

$$(A \Rightarrow B)^T \coloneqq !?A^T \multimap ?B^T$$
$$(A_1, \ldots, A_n \vdash B)^T \coloneqq \vdash ?!(A_1^T)^\perp, \ldots, ?!(A_n^T)^\perp, ?B^T$$

Alors une formule A est démontrable en logique classique si et seulement si sa traduction A^T est démontrable en logique linéaire.

Pour passer d'une démonstration linéaire à une démonstration classique, il suffit essentiellement d'oublier tout ce qui concerne les modalités : les règles linéaires n'étant que des versions contraintes des règles classiques, la correction est préservée. Pour l'implication réciproque, il suffit d'exhiber une traduction des règles de démonstrations classiques en enchaînements de règles linéaires, en introduisant correctement les modalités.

Ceci induit une traduction des termes du λ-calcul vers ceux du π-calcul fonctionnel, paramétrée par le nom a attribué à la conclusion :

$$(x)_a^T \coloneqq \bar{x}(b).b{=}a$$
$$(\lambda x\, M)_a^T \coloneqq \bar{a}(xb).M_b^T$$
$$((M)\, N)_a^T \coloneqq (\boldsymbol{\nu} f)\big(M_f^T \mid !f(xb).(!x(v).N_v^T \mid b{=}a)\big)$$

La vérification du bon typage de ces termes, en supposant que les λ-termes traduits sont bien typés, est laissée au lecteur.

L'étude du comportement calculatoire des processus ainsi obtenus, qui est un peu technique, révèle que leur exécution par $\succ$ correspond fidèlement à l'exécution $\succ$ des exécutables dans la machine de Krivine, la seule différence étant une étape de déréférencement lorsqu'une variable apparaît en tête. L'étude détaillée de l'exécution est laissée en exercice au lecteur, dans les grandes lignes les termes en $!f(xb)$ sont les chaînons de la pile, duplicables car le contrôle permet de copier les piles, et les termes en $!x(v)$ représentent les valeurs stockées sur la pile, également duplicables comme arguments de fonctions. Ainsi cette traduction s'avère être une interprétation par *passage de continuations*, ou traduction CPS, connue depuis bien longtemps dans le cadre pur du λ-calcul [24]. La jonction est ainsi faite avec la section précédente !

Si l'on cherche à implémenter le type de la loi de Peirce au travers de cette traduction, en cherchant les preuves possibles du type $(((A \Rightarrow B) \Rightarrow A) \Rightarrow A)^T$, on est amené directement à poser la définition suivante :

$$(\boldsymbol{cc})_a^T \coloneqq \bar{a}(cb).\ \bar{c}(f).!f(xr).(!x(k).\bar{k}(vs).\bar{v}(b').b'{=}b \mid r{=}b).$$

Ce terme est un peu effrayant au premier abord mais en étudiant calmement ses exécutions possibles on vérifie qu'il a bien le comportement du terme ***cc*** défini précédemment.

Cette traduction $(\cdot)^T$ correspond donc bien à la réduction de tête (d'où sa notation) des λ-termes en présence de contrôle. Une version duale, engendrée par la traduction $(A \Rightarrow B)^Q \coloneqq !A^Q \multimap ?!B^Q$, engendre un modèle d'exécution différent que l'on pourraît qualifier par analogie de « réduction de queue », qui est en fait une forme d'exécution en *appel par valeur*.

5.3 Vers la concurrence

Ainsi, le modèle de calcul de plus bas niveau qui se cache derrière les démonstrations de logique linéaire permet une décomposition du calcul fonctionnel avec contrôle d'une façon qui permet d'étendre la correspondance de Curry-Howard. En oubliant le contenu logique, ce modèle de calcul se rapproche des langages permettant d'étudier le calcul concurrent, et on peut y distinguer les aspects qui en sont typiques : interaction, synchronisation possiblement non-déterministe, partage de ressources, risque d'interblocage, etc. C'est le typage par la logique linéaire qui contraint le comportement à être essentiellement fonctionnel. Il est donc naturel de se demander s'il est possible, au moyen d'autres systèmes logiques, d'étendre la correspondance au-delà du monde parfait des fonctions pures.

Une source possible de non-déterminisme dans le calcul non-typé est l'ambiguïté présente dans la communication. Considérons par exemple un terme de la forme

$$T \coloneqq (\boldsymbol{\nu} a)\big(\bar{a}(u).u{=}u' \mid \bar{a}(v).v{=}v' \mid !a(x).P \mid \ !a(y).Q\big)$$

D'après les règles du calcul, on a $P \succ (\boldsymbol{\nu} x)(P \mid x{=}u') \mid (\boldsymbol{\nu} y)(Q \mid y{=}v')$ et $P \succ (\boldsymbol{\nu} x)(P \mid x{=}v') \mid (\boldsymbol{\nu} y)(Q \mid y{=}u')$ selon l'appariement entre les actions sur a. La logique élimine ce cas en interdisant purement et simplement la présence de plusieurs actions $!a()$ dans un même terme, seules les actions $\bar{a}()$ peuvent être multiples, grâce aux règles sur les modalités.

Le modèle relationnel revient ici nous suggérer une piste pour relâcher cette contrainte, car dans ce modèle il existe un morphisme de *cocontraction* de type $!A \otimes !A \multimap !A$ qui consiste simplement en la somme des multiensembles,[13] et qui est symétrique de la contraction $!A \multimap !A \otimes !A$. En termes de règles de démonstration, et d'interprétation calculatoire, on obtient la règle

$$\frac{P \vdash \Gamma, !A \qquad Q \vdash \Delta, !A}{P \mid Q \vdash \Gamma, \Delta, !A}$$

13. Il est possible d'écrire un morphisme de ce type avec les règles de base, mais uniquement en oubliant l'un des deux membres du tenseur au moyen d'un affaiblissement, ce qui n'est pas l'effet recherché.

qui permet de typer l'exemple précédent et d'interpréter le processus dans le modèle relationnel au même titre que les autres constructions du langage. De même, on peut introduire une *codéréliction* et un *coaffaiblissement* qui permettent de créer des actions linéaires voire inexistantes là où on n'avait encore que des actions répliquées $!a()$. On gagne ainsi en expressivité, en faisant un pas vers la concurrence tout en restant dans le monde typé. Cet élargissement peut être remonté dans le λ-calcul où il apparaît comme l'ajout d'opérateurs *différentiels* qui permettent d'internaliser l'interprétation des λ-termes comme des fonctions analytiques : c'est le sujet de la dernière partie du cours.

C'est à peu près le mieux que l'on sache faire à l'heure actuelle comme pendant logique du calcul concurrent : des structures algébriques existent pour rendre compte de phénomènes concurrents beaucoup moins restreints (les structures d'événements, les traces de Mazurkiewicz) mais on ne connaît pas de système logique permettant d'en capturer les propriétés avec la même finesse que la correspondance de Curry-Howard. Le sujet reste grand ouvert.

6 Envoi : à propos de sémantique quantitative

Le vocabulaire et les notations de la logique linéaire reflètent pour une large part ceux de l'algèbre. Au delà d'une simple analogie, soulignée dans la remarque 3.11, il y a de bonnes raisons à cela, notamment historiques.

On a déjà indiqué que la logique linéaire avait été inventée par Girard, en étudiant une sémantique dénotationnelle du λ-calcul dans les espaces cohérents : au lieu de simples ensembles, ce modèle est construit sur des graphes, et les relations doivent respecter certaines propriétés sur les arêtes. Ce modèle est lui même une simplification des domaines qualitatifs [11] qu'il avait précédemment introduits pour donner une sémantique dénotationnelle au système F, lequel étend le λ-calcul simplement typé à la quantification du second ordre telle que décrite en section 4.1.1. Et les domaines qualitatifs proviennent d'une relecture qualitative d'une idée précédemment développée par Girard : la sémantique quantitative [12], dont on va esquisser les idées.

Le principe du modèle relationnel vu plus haut était de capturer la sémantique d'un programme en considérant les utilisations possibles qu'il faisait de son argument pour produire un résultat. Un programme utilisant exactement une fois son argument étant dit linéaire, on peut généraliser et considérer comme n-linéaire un programme qui consulte systématiquement n fois son argument. Le modèle relationnel représente donc un programme quelconque comme une superposition de programmes multilinéaires.

En mathématiques, on connaît une situation similaire : les polynômes sont des sommes finies de monômes, archétypes de fonctions multilinéaires ; et plus généralement, une fonction analytique est définie par une série entière. L'idée de la sémantique quantitative est justement de considérer les λ-termes

comme des fonctions analytiques. Plus précisément, on peut interpréter un terme t comme une combinaison linéaire $\sum_{\alpha\in[\![t]\!]} t_\alpha \alpha$: pour $\alpha \in [\![t]\!]$, le scalaire t_α représente la contribution de l'atome α dans le comportement global de t. L'application est alors donnée par une série entière :

$$((s)\,t)_\beta = \sum_{(\overline{\alpha},\beta)\in s} s_{(\overline{\alpha},\beta)} t^{\overline{\alpha}} \tag{3}$$

où $t^{[\alpha_1,\dots,\alpha_n]} = t_{\alpha_1}\cdots t_{\alpha_n}$. Intuitivement : pour calculer la contribution de β dans $(s)\,t$, on fait la somme sur toutes les manières d'obtenir β avec une instance de s, et on pondère par les contributions des instances de t utilisées au cours d'un même calcul.

Bien sûr, on ne peut pas s'en tenir à la version naïve : il faut que la série converge. La solution de Girard utilisait une machinerie catégorique sophistiquée, qui l'a mené à la logique linéaire par simplifications successives. Suite à une série de travaux de Thomas Ehrhard au début des années 2000, on peut aujourd'hui présenter assez littéralement la sémantique quantitative dans un cadre d'algèbre linéaire standard (les preuves sont des matrices), avec une topologie moins standard (il en faut une, puisque ces matrices sont infinies).

La clef est de s'assurer que la somme converge *parce qu'elle est finie* : on introduit pour cela une structure au dessus du modèle relationnel, et on obtient des *espaces de finitude* [7]. Cette structure de finitude permet de munir les espaces vectoriels considérés d'une topologie assez différente de celles qu'on a l'habitude de voir : en effet, les ouverts sont engendrés par des sous espaces vectoriels et ces espaces ne sont en général pas métrisables.

On peut alors vérifier que les vecteurs de $A \multimap B$ sont exactement les matrices qui représentent des applications linéaires et continues de A dans B, pour cette topologie. De plus, les vecteurs de $!A \multimap B$ (et donc en particulier les interprétations de λ-termes typés) sont des généralisations de séries entières et les applications correspondantes sont analytiques : les λ-termes sont infiniment dérivables ! Et en relisant astucieusement cette propriété dans la syntaxe, on obtient le système de la logique linéaire différentielle dû à Thomas Ehrhard et Laurent Regnier. Mais ceci est une autre histoire...

7 Notice bibliographique

Les textes de référence sur le λ-calcul ne manquent pas : la référence classique est le Barendregt [2], mais le Krivine [18] à l'avantage d'être concis tout en abordant la question du typage.

Le David–Nour–Raffalli [5] est une bonne introduction à la théorie de la démonstration. Pour une référence plus complète, notamment sur les théorèmes de Gödel, voir le Cori–Lascar [8, 9]. Sur la correspondance de Curry-Howard plus précisément, le *Proofs and Types* [15] ou le premier tome du

Point aveugle [14] de Girard sont riches de remarques éclairantes, tandis que pour un cours exhaustif on pourra préférer le Sørensen–Urzyczyn [27].

Sur les modèles du λ-calcul, le Barendregt présente un point de vue assez classique, qu'on pourra compléter avec la somme d'Amadio et Curien sur les domaines [1]. Sur la logique linéaire et ses modèles, le *Point aveugle* fournit une bonne référence en français. Par ailleurs, la communauté française autour de la logique linéaire a entrepris la rédaction d'une sorte de manuel de référence de la logique linéaire et des sujets connexes, sous la forme d'un wiki : `http://llwiki.ens-lyon.fr/`.

Pour les autres points abordés dans ce chapitre (réalisabilité classique ; liens entre logique linéaire et calcul concurrent ; sémantique quantitative), il faut encore écumer les articles de recherche.

Références

[1] R. Amadio and P. L. Curien. Domains and lambda calculi. 1998.

[2] H. Barendregt. *The Lambda-calculus*. North Holland, 1984.

[3] G. Berry. Stable Models of Typed lambda-Calculi. In *International Colloquium on Automata, Languages and Programming*, pages 72–89, 1978.

[4] H. B. Curry. Functionality in Combinatory Logic. *Proceedings of The National Academy of Sciences*, 20 :584–590, 1934.

[5] R. David, K. Nour, and C. Raffalli. *Introduction à la logique : théorie de la démonstration : cours et exercices corrigés*. Sciences sup. Dunod, 2004.

[6] R. de Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34 :83–133, 1984.

[7] T. Ehrhard. Finiteness spaces. *Mathematical Structures in Computer Science*, 15(4) :615–646, 2005.

[8] R. C. et Daniel Lascar. *Logique mathématique 1 - Calcul propositionnel ; algèbre de Boole ; calcul des prédicats*. Dunod, 2003.

[9] R. C. et Daniel Lascar. *Logique mathématique 2 - Fonctions récursives ; théorème de Gödel ; Théorie des ensembles*. Dunod, 2003.

[10] J.-Y. Girard. The system f of variable types, fifteen years later. *Theor. Comput. Sci.*, 45(2) :159–192, 1986.

[11] J.-Y. Girard. The system F of variable types, fifteen years later. *Theor. Comput. Sci.*, 45(2) :159–192, 1986.

[12] J.-Y. Girard. Normal functors, power series and lambda-calculus. *Annals of Pure and Applied Logic*, 37(2) :129–177, 1988.

[13] J.-Y. Girard. Proof-nets : The parallel syntax for proof-theory. In P. Agliano and A. Ursini, editors, *Logic and Algebra*. M. Dekker, New York, 1996.

[14] J.-Y. Girard. *Le point aveugle : cours de logique. Tome 1. , Vers la perfection*. Visions des sciences. Hermann, Paris, 2006.

[15] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*. CUP, Cambridge, 1989.

[16] T. G. Griffin. A formulae-as-types notion of control. In *17th ACM Symposium on Principles of Programming Languages (POPL)*, pages 47–58, jan 1990.

[17] W. Howard. The formulae-as-types notion of construction. In H. Curry, J. Seldin, and R. Hindley, editors, *To H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[18] J.-L. Krivine. Lambda-calculus types and models. Feb. 2002.

[19] J.-L. Krivine. Dependent choice, 'quote' and the clock. *Theoretical Computer Science*, 308 :259–276, 2003.

[20] J.-L. Krivine. A call-by-name λ-calculus machine. *Higher-order and symbolic computation*, 20 :199–207, 2007.

[21] J. Lambek and P. J. Scott. *Higher Order Categorical Logic*. Cambridge University Press, 1986.

[22] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts i and ii). *Information and Computation*, 100 :1–77, 1992.

[23] A. Miquel. Forcing as a program transformation. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science*, pages 197–206. IEEE Computer Society, 2011.

[24] G. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2) :125–159, dec 1975.

[25] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5 :223–255, 1977.

[26] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab., 1971.

[27] M. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 2006.

Index des mots-clés