A type system with implicit types

Lionel Vaux

Last change: 3rd November 2004

Foreword

This paper is a mainly a translation of my "Mémoire de Maîtrise" ([Vau03], in french). It describes results obtained during a training period that took place in Chalmers' Computing Science Department (Göteborg University, Sweden), during summer 2003, under the supervision of Thierry Coquand. Obviously, most of the original ideas in this work are his, and I'd like to thank him here most warmly, for welcoming me and introducing me to type theory.

I'd also like to thank Pierre Hyvernat and more generally PhD students from both departments for the discussions we had (and for the cakes). Many thanks to Catarina Coquand too, for her warm welcome.

Even though we first met months after the training period, I'd like to thank Alexandre Miquel, whose Implicit Calculus of Constructions is the very basis of this work, because his PhD thesis is an example to follow both in scientific content and in style.

Last, I owe many thanks to Andreas Abel for his careful reading and meaningful remarks.

Introduction

Since the 70's and the Curry-Howard isomorphism, the use of type systems in the proofs-as-program paradigm has provided numerous formalisms, in which proofs are identified with λ-terms.

For instance, the Calculus of Constructions (CC) is the type-theoretical basis of formal proof assistants such as Coq. However the richness of CC has a drawback which is common to nearly all higher order type system: in contrast to the conciseness of type systems such as that of ML or system F, the syntax of CC, and in general that of all pure type systems, makes heavy use of redundant typing information inside terms.

Let's illustrate that point with polymorphic lists. In OCaml (ML):

```
[] : 'a list
```

and in Coq (CC):

```
nil : (A:Set)(list A)
```

Hence in CC there is no such thing as a polymorphic empty list: each list type `(list A)` has its own empty list `(nil A)`, whereas in ML, `[]` is truly in every list type, *e.g.* `(int list)`.

In existing implementations, this problem has already been addressed using technical solutions: some constructor arguments can be marked as implicit and those implicit arguments are computed by the program when needed, instead of given by the user. However, type theorists shall keep in mind that these computed arguments still occur internally in proof terms: they are necessary for the type system.

Alexandre Miquel's Implicit Calculus of Constructions (ICC) is an attempt to solve this problem by the use of implicit dependent types in a type system very close to CC (including a universe hierarchy): see [Miq98] and [Miq01] for a complete description of this system and its properties. Let's just recall here that types can be formed using the usual dependent product (denoted $\Pi x{:}A.B$) but also using an implicit dependent product (denoted $\cap x{:}A.B$) which bears some similarities to intersection of types, like *e.g.* in Curry-style system F. The main typing rules involving implicit products are:

$$\frac{\Gamma, x : A \vdash M : B \quad x \notin FV(M)}{\Gamma \vdash M : \cap x{:}A.B}\ (\textsc{Gen})$$

and

$$\frac{\Gamma \vdash M : \cap x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash M : B(N/x)}\ (\textsc{Inst})$$

These rules clearly allow to omit some parts of terms that are relevant for typing only. This is not an implementation trick: a term M in type $\cap x{:}A.B$ really is in every type $B(N/x)$ with N in type A.

ICC has the following noteworthy properties:

- there is no unicity of typing;
- typing is undecidable;

- ICC is coherent: there is no closed term with type $\cap A{:}\mathsf{Set}.A$;
- typed terms are strongly normalizing.

This paper presents an implicit calculus strongly inspired by ICC but simpler in several aspects. It has been conceived in order to allow better understanding of implicit dependent product, and to be easily implementable (although there should only exist partial implementations, because this calculus is highly suspected to be undecidable too).

Called IC in the following, this calculus introduces implicit dependent types in an type system close to Martin-Löf's logical framework, and simpler than pure type systems. Types and terms form two distinct syntactical classes: types are not typable themselves, hence there is no sort nor universe hierarchy. However, dependent types and implicit dependent types are still present, so that not all syntactically correct types are valid in any context: we need rules to handle the well-formedness of types and contexts. Moreover, subtyping is directly handled by the type system. Hence, as the reader may have expected it, IC typing rules are quite numerous (see figure 1) but most properties of the calculus become easier to prove, because it can be given simpler models than ICC. Despite these simplifications, the interest of implicit constructions still holds, as exposed in 1.3.

Plan In section 1, we present IC syntax and type system. In sections 2 and 3 we study generic models of IC and their properties, and give sufficient conditions for an interpretation of the calculus into a model to be correct. In section 4 we define a saturation condition that is preserved by the basic operations on models and in section 5, we use saturation to prove coherence of IC. In section 6, we briefly describe a partial implementation of a type checker for IC, and in section 7, we mention ongoing and future works.

1 The calculus

1.1 Syntax

We define terms, types and contexts according to the following grammars:

$$\begin{array}{rl} \text{Terms:} & M, N ::= x \mid \lambda x M \mid M\ N \\ \text{Types:} & A, B ::= \mathsf{Set} \mid \mathsf{El}\ M \mid \Pi x{:}A.B \mid \cap x{:}A.B \\ \text{Contexts:} & \Gamma ::= \diamond \mid \Gamma, x : A \end{array}$$

Terms are actually pure (possibly open) λ-terms. El converts a term into a type; $\mathsf{El}\ M$ is well-formed if M has type Set (see typing rules, in section 1.2), *i.e.* terms in Set are codes for types. $\Pi x{:}A.B$ is the usual dependent product; in the following, it is called explicit product, by contrast with $\cap x{:}A.B$, the implicit product, which uses the same contruction, but obeys different typing rules. Equality on terms or types is α-convertibility.

Remark There is actually no notion like *sort* in the type system: types are syntactically distinct from terms (hence the El construction), and are not typable themselves.

Definition 1 *Write $M(N/x)$ for capture avoiding substitution of N for x in term M. Write $\rightarrow_\beta$, $\rightarrow_\eta$ and $\rightarrow_{\beta\eta}$ respectively for β-, η- and $\beta\eta$-reduction and $\equiv_\beta$, $\equiv_\eta$ and $\equiv_{\beta\eta}$ respectively for β-, η- and $\beta\eta$-conversion. Those are easily extended to types and contexts.*

1.2 Typing

We simultaneously define judgements of form $\Gamma \vdash$, $\Gamma \vdash A$, $\Gamma \vdash M : A$ and $\Gamma \vdash A \leq B$ by the rules given in figure 1. We may use the generic notation $\Gamma \vdash J$ for any judgement with $J ::= \epsilon \mid A \mid M : A \mid A \leq B$.

1.2.1 Typing terms in an implicit product

The main two typing rules for implicit product are (Gen) and (Inst) which are similar to (Abs) and (App) for explicit product. We can interpret both as defining implicit product as type intersection: a term is in an intersection iff it is in every type of the intersection. It is important to note that in (Gen), the variable which is bound by the product cannot be free in the typed term.

1.2.2 Implicit product and subtyping

Again, we can interpret implicit product as an intersection. We may read (S-Imp) and (Imp-S) as follows:

- a type is a sub-type of an intersection if it is a sub-type of every type of the intersection:
- an intersection is a sub-type of a type if one type of the intersection is such.

1.2.3 Implicit product and strengthening rules

Rules of form:

$$\frac{\Gamma, x : A \vdash J \quad x \notin FV(J)}{\Gamma \vdash J}$$

are called strengthening rules and their appearing as typing rules is quite peculiar. Indeed they are generally admissible in the usual Calculus of Constructions and its variants, but not here. For instance, denote $\bot = \cap A{:}\mathsf{Set}.A$ the implicit contradiction. Using strengthening rules, we can infer that $\cap x{:}\bot.\bot = \bot$ (extensionally: $\Gamma \vdash M : \bot \iff \Gamma \vdash M : \cap x{:}\bot.\bot$), which is quite surprising about an operator we would tend to identify with an intersection; indeed, we will see in section 5 that $\bot$ has no proof in the empty context.

With stengthening rules, we can actually prove a more general result:

$$(\Gamma \vdash M : B \iff \Gamma \vdash M : \cap x{:}A.B) \text{ when } x \notin FV(B)$$

even when there is no N such that $\Gamma \vdash N : A$. ($\Rightarrow$) is easily proved without strengthening rules, but for ($\Leftarrow$) we need to introduce a variable of type A.

Contexts:

$$\frac{}{\diamond \vdash}\ (\text{WF-E}) \qquad \frac{\Gamma \vdash A \quad x \notin DV(\Gamma)}{\Gamma, x : A \vdash}\ (\text{WF-C})$$

Types:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{Set}}\ (\textsc{Set}) \qquad \frac{\Gamma \vdash M : \mathsf{Set}}{\Gamma \vdash \mathsf{El}\ M}\ (\textsc{El})$$

$$\frac{\Gamma, x : A \vdash B}{\Gamma \vdash \Pi x{:}A.B}\ (\textsc{Exp}) \qquad \frac{\Gamma, x : A \vdash B}{\Gamma \vdash \cap x{:}A.B}\ (\textsc{Imp})$$

Terms:

$$\frac{\Gamma \vdash \quad (x : A) \in \Gamma}{\Gamma \vdash x : A}\ (\textsc{Var})$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x M : \Pi x{:}A.B}\ (\textsc{Abs}) \qquad \frac{\Gamma \vdash M : \Pi x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B(N/x)}\ (\textsc{App})$$

$$\frac{\Gamma, x : A \vdash M : B \quad x \notin FV(M)}{\Gamma \vdash M : \cap x{:}A.B}\ (\textsc{Gen}) \qquad \frac{\Gamma \vdash M : \cap x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash M : B(N/x)}\ (\textsc{Inst})$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A \leq B}{\Gamma \vdash M : B}\ (\textsc{Cum})$$

Sub-typing:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{Set} \leq \mathsf{Set}}\ (\text{S-}\textsc{Set})$$

$$\frac{\Gamma \vdash M : \mathsf{Set} \quad \Gamma \vdash N : \mathsf{Set} \quad M \equiv_{\beta\eta} N}{\Gamma \vdash \mathsf{El}\ M \leq \mathsf{El}\ N}\ (\text{S-}\textsc{El})$$

$$\frac{\Gamma \vdash A_2 \leq A_1 \quad \Gamma, x : A_2 \vdash B_1 \leq B_2 \quad \Gamma, x : A_1 \vdash B_1}{\Gamma \vdash \Pi x{:}A_1.B_1 \leq \Pi x{:}A_2.B_2}\ (\text{S-}\textsc{Exp})$$

$$\frac{\Gamma, x : A \vdash C \leq B \quad x \notin FV(C)}{\Gamma \vdash C \leq \cap x{:}A.B}\ (\text{S-}\textsc{Imp})$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B(M/x) \leq C \quad \Gamma, x : A \vdash B}{\Gamma \vdash \cap x{:}A.B \leq C}\ (\textsc{Imp}\text{-S})$$

Strengthening:

$$\frac{\Gamma, x : A \vdash B \quad x \notin FV(B)}{\Gamma \vdash B}\ (\textsc{Str-Type})$$

$$\frac{\Gamma, x : A \vdash M : B \quad x \notin FV(M) \cup FV(B)}{\Gamma \vdash M : B}\ (\textsc{Str-Term})$$

$$\frac{\Gamma, x : A \vdash B \leq C \quad x \notin FV(B) \cup FV(C)}{\Gamma \vdash B \leq C}\ (\textsc{Str-Sub})$$

Figure 1: Typing rules

1.3 Examples

The simplest example of a polymorphic function is identity. In CC, identity needs a domain argument. In a *Coq*-like idiom, we may write:

$$id : \Pi A{:}\mathsf{Set}.(A \to A) := \lambda A{:}\mathsf{Set}\lambda x{:}Ax$$

In IC, we may write:

$$id : \cap A{:}\mathsf{Set}.(\mathsf{El}\ A \to \mathsf{El}\ A) := \lambda xx$$

Not only definition is shorter, usage is also more concise because we don't need to give a domain as an argument.

Now we can precise our remarks about polymorphic lists and the parallel with ML-like type systems. In CC, we usually have:

$$list : \mathsf{Set} \to \mathsf{Set}$$

$$nil : \Pi A{:}\mathsf{Set}.(list\ A)$$

$$cons : \Pi A{:}\mathsf{Set}.(A \to list\ A \to list\ A)$$

whereas in IC we can obtain:

$$list : \mathsf{Set} \to \mathsf{Set}$$

$$nil : \cap A{:}\mathsf{Set}.(list\ (\mathsf{El}\ A))$$

$$cons : \cap A{:}\mathsf{Set}.((\mathsf{El}\ A) \to list\ (\mathsf{El}\ A) \to list\ (\mathsf{El}\ A))$$

and then we are in a rather ML-like approach. Indeed, in IC, *nil* has type $list\ (\mathsf{El}\ A)$ for all A in Set. Furthermore, we don't have to give the basic type of considered lists as an argument of *cons*.

However, we have to be careful in our attempts to generalize the use of implicit product. Indeed, we need to chose those explicit products that can become implicit. The criterium for this choice follows from the structure of rule (Gen): one argument can be "forgotten" if it has no computational use *i.e.* it does not appear in the proof. For instance, one can write a fonction:

$$head : \cap A{:}\mathsf{Set}.\Pi l{:}list\ (\mathsf{El}\ A).((not\ (eq\ (list\ (\mathsf{El}\ A))\ l\ nil)) \to \mathsf{El}\ A)$$

but cannot achieve:

$$head' : \cap A{:}\mathsf{Set}.\Pi l{:}list\ (\mathsf{El}\ A).\cap p{:}not\ (eq\ (list\ (\mathsf{El}\ A))\ l\ nil).\mathsf{El}\ A$$

Indeed, we need a proof of $not\ (eq\ (list\ (\mathsf{El}\ A))\ l\ nil)$ to get the first element of l. More precisely, we can prove that for such a term $head'$, the aplication $(head'\ nil)$ is typable, which is exactly what we wanted to avoid.

One last point: implicit product is "finer" than explicit product, in the sense that it is more discriminant. Indeed an explicit product has at least as many proofs as the corresponding implicit product: if $\Gamma \vdash M : \cap x{:}A.B$, $x \notin FV(M)$, then $\Gamma \vdash \lambda xM : \Pi x{:}A.B$.

In particular, implicit contradiction $\bot = \cap A{:}\mathsf{Set}.(\mathsf{El}\ A)$ implies explicit contradiction $\bot' = \Pi A{:}\mathsf{Set}.(\mathsf{El}\ A)$.

1.4 Srong normalization

We can prove strong normalization of typed terms without much effort by relating IC with simply typed λ-calculus.

Lemma 1 *If M is a typable term in IC, then M is a simply typed λ-term.*

Proof. Define $(\cdot)^\diamond$ as a forgetfull transformation from IC types to simple types:

$$\begin{array}{rcl} \mathsf{Set}^\diamond & = & \alpha \\ (\mathsf{El}\ M)^\diamond & = & \alpha \\ (\cap x{:}A.B)^\diamond & = & B^\diamond \\ (\Pi x{:}A.B)^\diamond & = & A^\diamond \to B^\diamond \end{array}$$

and extend it to contexts:

$$(x_1 : A_1, \ldots, x_n : A_n)^\diamond = x_1 : {A_1}^\diamond, \ldots, x_n : {A_n}^\diamond$$

Obviously $(A(M/x))^\diamond = A^\diamond$. Then one easily proves by induction on IC typing judgements that the following properties hold:

- if $\Gamma \vdash A \leq B$ then $A^\diamond = B^\diamond$;
- if $\Gamma \vdash M : A$ then M has type $A^\diamond$ in context $\Gamma^\diamond$ in usual simply typed λ-calculus.

□

1.5 Well-formedness of derivable judgements

Along with well-formedness of typing judgements, the following lemma is useful for inductions on derivable judgements by ensuring that some sub-derivations occur.

Lemma 2 (Well-formedness) *Let $\Gamma = x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$ be a context, A and B types and M a term. Write $\Gamma[i] = x_1 : A_1, \ldots, x_i : A_i$. The following properties hold.*

(i) if $\Gamma \vdash$ is derivable then for all i s.t. $1 \leq i \leq n$, $FV(A_i) \subset DV(\Gamma[i-1])$ and $\Gamma[i-1] \vdash A_i$ occurs in the derivation;

(ii) if $\Gamma \vdash J$ is derivable then $FV(J) \subset DV(\Gamma)$ and $\Gamma \vdash$ occurs in the derivation.

Proof. Both parts are proved simultaneously by induction on the struture of derivations. □

This is the only lemma about the type system itself that is used in the rest of the paper. In the following, we introduce models of the calculus and results about these models; then we deduce properties of the calculus through a correctness theorem.

2 λ-models

The notion of λ-model is the abstract structure which the interpretation of terms relies on. Once given the axiomatization of an interpretation of terms, we can define models of the calculus up to such an interpretation, thus achieve generality. Then we will be able to obtain properties of the calculus by studying particular instances of such λ-models.

2.1 Notations and definition

Denote **Var** the set of variables and Λ that of terms. Given a set $\mathfrak{D}$, denote $\mathbf{Val}(\mathfrak{D})$ the set of functions $\mathbf{Var} \longrightarrow \mathfrak{D}$ (environments).

Definition 2 *A λ-model is a triple $\mathfrak{M} = (\mathfrak{D}, \circ, [\![.]\!]_.)$ s.t. $\circ \ : \ (\mathfrak{D} \times \mathfrak{D}) \longrightarrow \mathfrak{D}$ and $[\![.]\!]_. \ : \ \Lambda \times \mathbf{Val}(\mathfrak{D}) \longrightarrow \mathfrak{D}$ obey the following axioms :*

$$\begin{array}{lr} [\![x]\!]_\rho = \rho(x) & (\delta) \\ [\![M\ N]\!]_\rho = [\![M]\!]_\rho \circ [\![N]\!]_\rho & (App) \\ [\![\lambda x M]\!]_\rho \circ a = [\![M]\!]_{\rho[x \mapsto a]} & (\beta) \end{array}$$

$\circ$ is the model's application and $[\![.]\!]_.$ is the interpretation of terms according to environments.

This is the very least we can demand from a "useable" interpretation of terms, but in the following, we will need additional properties.

2.2 Semi-extensionality

Definition 3 *A semi-extensional λ-model is a λ-model obeying the additional axiom :*

$$(\forall a \in \mathfrak{D},\ [\![M]\!]_{\rho[x\mapsto a]} = [\![N]\!]_{\rho'[x\mapsto a]}) \Longrightarrow [\![\lambda x M]\!]_\rho = [\![\lambda x N]\!]_{\rho'}. \quad (\xi)$$

In a semi-extensional λ-model, the following properties hold.

Lemma 3 (Irrelevance) $(\forall x \in FV(M),\ \rho(x) = \rho'(x)) \Longrightarrow [\![M]\!]_\rho = [\![M]\!]_{\rho'}$.

Proof. By induction on M using (ξ) for abstraction. □

Lemma 4 (Substitution) $[\![M(N/x)]\!]_\rho = [\![M]\!]_{\rho[x\mapsto [\![N]\!]_\rho]}$.

Proof. By induction on M. We give details in the case $\lambda x M$: modulo α-conversion in $\lambda x M$, $x \notin FV(N)$ and $x \neq y$, and then

$$\begin{array}{rcll} \forall a \in \mathfrak{D},\ [\![M(N/y)]\!]_{\rho[x\mapsto a]} & = & [\![M]\!]_{\rho[x\mapsto a][y\mapsto [\![N]\!]_{\rho[x\mapsto a]}]} & \text{(induction hypothesis)} \\ & = & [\![M]\!]_{\rho[x\mapsto a][y\mapsto [\![N]\!]_\rho]} & (x \notin FV(N)) \\ & = & [\![M]\!]_{\rho[y\mapsto [\![N]\!]_\rho][x\mapsto a]} & (x \neq y) \end{array}$$

hence $[\![(\lambda x M)(N/y)]\!]_\rho = [\![\lambda x M]\!]_{\rho[y\mapsto [\![N]\!]_\rho]}$ by (ξ). □

Lemma 5 (β-equivalence) *If $M \equiv_\beta N$ then $[\![M]\!]_\rho = [\![N]\!]_\rho$.*

Proof. It is sufficient to prove the result when $M \to_\beta N$. By induction on M : M is not a variable and we use lemma 4 for application and (ξ) for abstraction, along with induction hypothesis on sub-terms. □

2.3 Extensionality

Definition 4 *An extensional λ-model is a λ-model with the additional axiom :*

$$(\forall a \in \mathfrak{D},\ b \circ a = c \circ a) \Longrightarrow b = c \qquad (Ext)$$

Lemma 6 (Weakening (Ext)) *An extensional λ-model is semi-extensional.*

Proof. (ξ) is straightforward from (β) and (Ext). □

In an extensional λ-model, the following lemma holds.

Lemma 7 ($\beta\eta$-equivalence) *If $M \equiv_{\beta\eta} N$ then $[\![M]\!]_\rho = [\![N]\!]_\rho$.*

Proof. It is sufficient to prove the result when $M \to_\beta N$ or $M \to_\eta N$. The first case holds by lemma 5. The second one is straightforward using (Ext) and lemma 4. □

3 Per-models

Remark The following is an attempt to generalize the method used in [CPT]. The main outcome of our work in this part and the next one is a notion of saturation that does not rely on any particular λ-model.

In the following, we define a structure on λ-models to model types : a per-model interprets a type in an environment as a partial equivalence relation on a λ-model. The interpretation of types is henceforth parametrized by the interpretation of terms.[1]

3.1 Pers

A per (*partial equivalence relation*) $\mathcal{A}$ on set $\mathfrak{D}$ is a binary relation which is symmetric and transitive. $\mathbf{Per}(\mathfrak{D})$ is the set of all pers on $\mathfrak{D}$. We may write $u = v : \mathcal{A}$ for $(u, v) \in \mathcal{A}$ and $u : \mathcal{A}$ for $u = u : \mathcal{A}$. When $u : \mathcal{A}$, $\overline{u}_\mathcal{A}$ is the equivalence class of u in $\mathcal{A}$, *i.e.* the set of all $v \in \mathfrak{D}$ s.t. $u = v : \mathcal{A}$. Write $\mathfrak{D}/\mathcal{A}$ for the set of all equivalence classes in $\mathcal{A}$. $\mathbf{Fam}(\mathcal{A})$ is the set of all functions $\mathcal{F} : \mathfrak{D}/\mathcal{A} \to \mathbf{Per}(\mathfrak{D})$. If $\mathcal{F} \in \mathbf{Fam}(\mathcal{A})$ and $u : \mathcal{A}$, we may write $\mathcal{F}(u)$ instead of $\mathcal{F}(\overline{u}_\mathcal{A})$.

Assume we have a binary operation $\circ$ on $\mathfrak{D}$. If $\mathcal{A} \in \mathbf{Per}(\mathfrak{D})$ and $\mathcal{F} \in \mathbf{Fam}(\mathcal{A})$, define $\overline{\Pi}(\mathcal{A}, \mathcal{F})$ and $\overline{\sqcap}(\mathcal{A}, \mathcal{F}) \in \mathbf{Per}(\mathfrak{D})$ by :

$$u_1 = u_2 : \overline{\Pi}(\mathcal{A}, \mathcal{F}) \text{ iff } \forall v_1 = v_2 : \mathcal{A},\ u_1 \circ v_1 = u_2 \circ v_2 : \mathcal{F}(v_1)$$

and

$$u_1 = u_2 : \overline{\sqcap}(\mathcal{A}, \mathcal{F}) \text{ iff } \forall v : \mathcal{A},\ u_1 = u_2 : \mathcal{F}(v).$$

Remark We can identify any subset $\mathcal{A}$ of $\mathfrak{D}$ with a per on $\mathfrak{D}$ by :

$$u_1 = u_2 : \mathcal{A} \iff (u_1 \in \mathcal{A} \text{ and } u_1 = u_2)$$

i.e. $\overline{u}_\mathcal{A} = \{u\}$ if $u \in \mathcal{A}$ and $\overline{u}_\mathcal{A} = \emptyset$ if $u \notin \mathcal{A}$.

[1] Other structures can possibly be used this way : in particular, it is possible to give a model in which types are interpreted by subsets of the λ-model — actually, this is a particular instance of per-model in which all pers are sets, *i.e.* pers whose classes are singletons.

3.2 Intensional equality on types

In the following, we assume $\mathfrak{M} = (\mathfrak{D}, \circ, [\![.]\!]_.)$ is a λ-model, $* \in \mathbf{Per}(\mathfrak{D})$ and $\mathcal{E} \in \mathbf{Fam}(*)$. Denote $\mathcal{T}_\Lambda$ the set of types.

We inductively define a binary relation on couples (A, ρ), with $A \in \mathcal{T}_\Lambda$ and $\rho \in \mathbf{Val}(\mathfrak{D})$, and a partial function $(A, \rho) \mapsto \lceil\!\lceil A \rfloor\!\rfloor_\rho \in \mathbf{Per}(\mathfrak{D})$ by :

- $[\mathsf{Set}]_{\rho_1} = [\mathsf{Set}]_{\rho_2}$ and $\lceil\!\lceil \mathsf{Set} \rfloor\!\rfloor_{\rho_1} = *$;
- $[\mathsf{El}\ M_1]_{\rho_1} = [\mathsf{El}\ M_2]_{\rho_2}$ iff $[\![M_1]\!]_{\rho_1} = [\![M_2]\!]_{\rho_2} : *$, and then $\lceil\!\lceil \mathsf{El}\ M_1 \rfloor\!\rfloor_{\rho_1} = \mathcal{E}([\![M_1]\!]_{\rho_1})$;
- $[\mathrm{R}x_1{:}A_1.B_1]_{\rho_1} = [\mathrm{R}x_2{:}A_2.B_2]_{\rho_2}$ iff
  - $[A_1]_{\rho_1} = [A_2]_{\rho_2}$,
  - $a_1 = a_2 : \lceil\!\lceil A_1 \rfloor\!\rfloor_{\rho_1} \Longrightarrow [B_1]_{\rho_1[x_1 \mapsto a_1]} = [B_2]_{\rho_2[x_2 \mapsto a_2]}$,
  - and $\mathcal{F}_i \ :\ a \mapsto \lceil\!\lceil B_i \rfloor\!\rfloor_{\rho_i[x_i \mapsto a]} \in \mathbf{Fam}(\lceil\!\lceil A_i \rfloor\!\rfloor_{\rho_i})$,

  and then $\lceil\!\lceil \mathrm{R}x_1{:}A_1.B_1 \rfloor\!\rfloor_{\rho_1} = \overline{\mathrm{R}}(\lceil\!\lceil A_1 \rfloor\!\rfloor_{\rho_1}, \mathcal{F}_1)$ (where $\mathrm{R} = \Pi$ or $\cap$).

We may write $[A]_\rho$ for $[A]_\rho = [A]_\rho$. $\lceil\!\lceil A \rfloor\!\rfloor_\rho$ is defined whenever $[A]_\rho$ holds.

Lemma 8 *Intensional equality is a per on $\mathcal{T}_\Lambda \times \mathbf{Val}(\mathfrak{D})$.*

Proof. Easy proof by induction. □

Lemma 9 *If $[A_1]_{\rho_1} = [A_2]_{\rho_2}$ then $\lceil\!\lceil A_1 \rfloor\!\rfloor_{\rho_1} = \lceil\!\lceil A_2 \rfloor\!\rfloor_{\rho_2}$.*

Proof. Easy induction on the proof of $[A_1]_{\rho_1} = [A_2]_{\rho_2}$. □

Thus we may omit the last condition when proving $[\mathrm{R}x_1{:}A_1.B_1]_{\rho_1} = [\mathrm{R}x_2{:}A_2.B_2]_{\rho_2}$ as stated in the following lemma.

Corollary 10 *If $[A_1]_{\rho_1} = [A_2]_{\rho_2}$ and $a_1 = a_2 : \lceil\!\lceil A_1 \rfloor\!\rfloor_{\rho_1} \Longrightarrow [B_1]_{\rho_1[x_1 \mapsto a_1]} = [B_2]_{\rho_2[x_2 \mapsto a_2]}$, then $[\mathrm{R}x_1{:}A_1.B_1]_{\rho_1} = [\mathrm{R}x_2{:}A_2.B_2]_{\rho_2}$.*

If $\mathfrak{M}$ is semi-extensional, the following lemmas hold.

Lemma 11 (Irrelevance) *Assume $\forall x_1 \in FV(A_1),\ \rho_1(x_1) = \rho'_1(x_1)$ and $\forall x_2 \in FV(A_2),\ \rho_2(x_2) = \rho'_2(x_2)$. Then $[A_1]_{\rho_1} = [A_2]_{\rho_2} \iff [A_1]_{\rho'_1} = [A_2]_{\rho'_2}$.*

Proof. We prove $[A_1]_{\rho_1} = [A_2]_{\rho_2} \Longrightarrow [A_1]_{\rho'_1} = [A_2]_{\rho'_2}$ by induction on the proof of $[A_1]_{\rho_1} = [A_2]_{\rho_2}$, using lemma 3 in the case $[\mathsf{El}\ M_1]_{\rho_1} = [\mathsf{El}\ M_2]_{\rho_2}$.
□

Lemma 12 (Substitution) *$[B_1(M_1/x_1)]_{\rho_1} = [B_2(M_2/x_2)]_{\rho_2}$ iff $[B_1]_{\rho_1[x_1 \mapsto [\![M_1]\!]_{\rho_1}]} = [B_2]_{\rho_2[x_2 \mapsto [\![M_2]\!]_{\rho_2}]}$, and then $[B_1(M_1/x_1)]_{\rho_1} = [B_1]_{\rho_1[x_1 \mapsto [\![M_1]\!]_{\rho_1}]}$.*

Proof. We prove $[B_1(M_1/x_1)]_{\rho_1} = [B_2(M_2/x_2)]_{\rho_2} \Longrightarrow ([B_1]_{\rho_1[x_1 \mapsto [\![M_1]\!]_{\rho_1}]} = [B_2]_{\rho_2[x_2 \mapsto [\![M_2]\!]_{\rho_2}]}$ and $[B_1(M_1/x_1)]_{\rho_1} = [B_1]_{\rho_1[x_1 \mapsto [\![M_1]\!]_{\rho_1}]})$ by induction on the proof of $[B_1(M_1/x_1)]_{\rho_1} = [B_2(M_2/x_2)]_{\rho_2}$ using lemma 4 in the case $[\mathsf{El}\ N_1(M_1/x_1)]_{\rho_1} = [\mathsf{El}\ N_2(M_2/x_2)]_{\rho_2}$.
We prove $[B_1]_{\rho_1[x_1 \mapsto [\![M_1]\!]_{\rho_1}]} = [B_2]_{\rho_2[x_2 \mapsto [\![M_2]\!]_{\rho_2}]} \Longrightarrow ([B_1(M_1/x_1)]_{\rho_1} = [B_2(M_2/x_2)]_{\rho_2}$ and $[B_1(M_1/x_1)]_{\rho_1} = [B_1]_{\rho_1[x_1 \mapsto [\![M_1]\!]_{\rho_1}]})$ by induction on the proof of $[B_1]_{\rho_1[x_1 \mapsto [\![M_1]\!]_{\rho_1}]} = [B_2]_{\rho_2[x_2 \mapsto [\![M_2]\!]_{\rho_2}]}$, using lemma 4 in the case $[\mathsf{El}\ N_1]_{\rho_1[x_1 \mapsto M_1]} = [\mathsf{El}\ N_2]_{\rho_2[x_2 \mapsto M_2]}$.
□

3.3 Validity

3.3.1 Interpretation of contexts

Given Γ context, define $\llbracket\Gamma\rrbracket \subset \mathbf{Val}(\mathfrak{D}) \times \mathbf{Val}(\mathfrak{D})$.

$$\frac{}{\rho_1 = \rho_2 : \llbracket \diamond \rrbracket}\ (\text{E-I})$$

$$\frac{\rho_1 = \rho_2 : \llbracket\Gamma\rrbracket \quad [A]_{\rho_1} = [A]_{\rho_2} \quad \rho_1(x) = \rho_2(x) : \llbracket A\rrbracket_{\rho_1}}{\rho_1 = \rho_2 : \llbracket\Gamma, x : A\rrbracket}\ (\text{C-I})$$

3.3.2 Valid judgements

We simultaneously define judgements of form $\Gamma \Vdash J$ by :

$$\frac{}{\diamond \Vdash}\ (\text{E-V}) \qquad \frac{\Gamma \Vdash A \quad x \notin DV(\Gamma)}{\Gamma, x : A \Vdash}\ (\text{C-V})$$

$$\frac{\Gamma \Vdash \quad \forall \rho_1 = \rho_2 : \llbracket\Gamma\rrbracket,\ [A_1]_{\rho_1} = [A_2]_{\rho_2}}{\Gamma \Vdash A_1 = A_2}\ (\text{Type-V})$$

$$\frac{\Gamma \Vdash A \quad \forall \rho_1 = \rho_2 : \llbracket\Gamma\rrbracket,\ \llbracket M_1\rrbracket_{\rho_1} = \llbracket M_2\rrbracket_{\rho_2} : \llbracket A\rrbracket_{\rho_1}}{\Gamma \Vdash M_1 = M_2 : A}\ (\text{Term-V})$$

$$\frac{\Gamma \Vdash A_1 \quad \Gamma \Vdash A_2 \quad \forall \rho : \llbracket\Gamma\rrbracket,\ \llbracket A_1\rrbracket_\rho \subset \llbracket A_2\rrbracket_\rho}{\Gamma \Vdash A_1 \leq A_2}\ (\text{Sub-V})$$

where $\Gamma \Vdash A$ stands for $\Gamma \Vdash A = A$.

3.4 Correctness

3.4.1 Useful properties

Lemma 13 *If* $\Gamma, x : A \Vdash$ *then* $\Gamma \Vdash$.

Proof. Inversion of (C-V) and (Type-V). □

Lemma 14 *(i) If* $\Gamma \Vdash$ *then* $(\rho_1, \rho_2 \mapsto \rho_1 = \rho_2 : \llbracket\Gamma\rrbracket) \in \mathbf{Per}(\mathbf{Val}(\mathfrak{D}))$;

(ii) $(A_1, A_2 \mapsto \Gamma \Vdash A_1 = A_2) \in \mathbf{Per}(\mathcal{T}_\Lambda)$;

(iii) $(M_1, M_2 \mapsto \Gamma \Vdash M_1 = M_2 : A) \in \mathbf{Per}(\Lambda)$.

Proof. (i) is proved by induction on Γ. (ii) is straightforward by lemma 8. (iii) is easy since the interpretation of a type is a per. □

Lemma 15 *If* $\Gamma \Vdash$, $\rho_1 = \rho_2 : \llbracket\Gamma\rrbracket$ *and* $y \notin DV(\Gamma)$ *then* $\forall a_1, a_2 \in \mathfrak{D}$, $\rho_1[y \mapsto a_1] = \rho_2[y \mapsto a_2] : \llbracket\Gamma\rrbracket$.

Proof. Induction on Γ. □

Lemma 16 *If* $\Gamma_1 \subset \Gamma_2$ *and* $\Gamma_2 \Vdash$ *then*

- $\llbracket\Gamma_2\rrbracket \subset \llbracket\Gamma_1\rrbracket$
- $\Gamma_1 \Vdash J \Longrightarrow \Gamma_2 \Vdash J$.

Proof. Easy. First prove the result with $\Gamma_2 = \Gamma_1, x : A$. □

Lemma 17 *Assume* $\mathfrak{M}$ *is extensional. If* $\Gamma \Vdash M_1 : A$ *and* $M_1 \equiv_{\beta\eta} M_2$ *then* $\Gamma \Vdash M_1 = M_2 : A$.

Proof. Straightforward using lemma 7. □

3.4.2 Correctness theorem

The following theorem states that an extensional λ-model founds a correct model of the calculus as soon as every interpretation of any type in any environment is non-empty. This condition is strongly related with strengthening rules and remarks we did in 1.2.3. Indeed the fact that

$$(\Gamma \vdash M : B \iff \Gamma \vdash M : \cap x{:}A.B) \text{ when } x \notin FV(B)$$

even when there is no N such that $\Gamma \vdash N : A$ suggests that $[\![A]\!]_\rho$ has to be non-empty for every relevant ρ, because the interpretation of an implicit product is an intersection of pers.

Theorem 18 *Assume $\mathfrak{M}$ is extensional. Assume for all $\rho \in \mathbf{Val}(\mathfrak{D})$ and all $A \in \mathcal{T}_\Lambda$, $[A]_\rho$ implies $[\![A]\!]_\rho \neq \emptyset$. Then the following implications (written as inferences) hold :*

$$\frac{}{\diamond \Vdash} \text{(M-WF-E)} \qquad \frac{\Gamma \Vdash A \quad x \notin DV(\Gamma)}{\Gamma, x : A \Vdash} \text{(M-WF-C)} \qquad \frac{\Gamma \Vdash}{\Gamma \Vdash \mathsf{Set}} \text{(M-Set)}$$

$$\frac{\Gamma \Vdash M = N : \mathsf{Set}}{\Gamma \Vdash \mathsf{El}\ M = \mathsf{El}\ N} \text{(M-El)} \qquad \frac{\Gamma \Vdash A_1 = A_2 \quad \Gamma, x : A_1 \Vdash B_1 = B_2}{\Gamma \Vdash \Pi x{:}A_1.B_1 = \Pi x{:}A_2.B_2} \text{(M-Exp)}$$

$$\frac{\Gamma \Vdash A_1 = A_2 \quad \Gamma, x : A_1 \Vdash B_1 = B_2}{\Gamma \Vdash \cap x{:}A_1.B_1 = \cap x{:}A_2.B_2} \text{(M-Imp)} \qquad \frac{\Gamma \Vdash \quad (x : A) \in \Gamma}{\Gamma \Vdash x : A} \text{(M-Var)}$$

$$\frac{\Gamma, x : A \Vdash M_1 = M_2 : B}{\Gamma \Vdash \lambda x M_1 = \lambda x M_2 : \Pi x{:}A.B} \text{(M-Abs)}$$

$$\frac{\Gamma \Vdash M_1 = M_2 : \Pi x{:}A.B \quad \Gamma \Vdash N_1 = N_2 : A}{\Gamma \Vdash M_1\ N_1 = M_2\ N_2 : B(N_1/x)} \text{(M-App)}$$

$$\frac{\Gamma, x : A \Vdash M_1 = M_2 : B \quad x \notin FV(M_1) \cup FV(M_2)}{\Gamma \Vdash M_1 = M_2 : \cap x{:}A.B} \text{(M-Gen)}$$

$$\frac{\Gamma \Vdash M_1 = M_2 : \cap x{:}A.B \quad \Gamma \Vdash N : A}{\Gamma \Vdash M_1 = M_2 : B(N/x)} \text{(M-Inst)}$$

$$\frac{\Gamma \Vdash M_1 = M_2 : A \quad \Gamma \Vdash A \leq B}{\Gamma \Vdash M_1 = M_2 : B} \text{(M-Cum)} \qquad \frac{\Gamma \Vdash}{\Gamma \Vdash \mathsf{Set} \leq \mathsf{Set}} \text{(M-S-Set)}$$

$$\frac{\Gamma \Vdash M_1 : \mathsf{Set} \quad \Gamma \Vdash M_2 : \mathsf{Set} \quad M_1 \equiv_{\beta\eta} M_2}{\Gamma \Vdash \mathsf{El}\ M_1 \leq \mathsf{El}\ M_2} \text{(M-S-El)}$$

$$\frac{\Gamma \Vdash A_2 \leq A_1 \quad \Gamma, x : A_2 \Vdash B_1 \leq B_2 \quad \Gamma \Vdash \Pi x{:}A_1.B_1}{\Gamma \Vdash \Pi x{:}A_1.B_1 \leq \Pi x{:}A_2.B_2} \text{(M-S-Exp)}$$

$$\frac{\Gamma, x : A \Vdash C \leq B \quad x \notin FV(C)}{\Gamma \Vdash C \leq \cap x{:}A.B} \text{(M-S-Imp)}$$

$$\frac{\Gamma \Vdash M : A \quad \Gamma \Vdash B(M/x) \leq C \quad \Gamma \Vdash \cap x{:}A.B}{\Gamma \Vdash \cap x{:}A.B \leq C} \text{(M-Imp-S)}$$

$$\frac{\Gamma, x : A \Vdash B_1 = B_2 \quad x \notin FV(B_1) \cup FV(B_2)}{\Gamma \Vdash B_1 = B_2} \text{(M-Str-Type)}$$

$$\frac{\Gamma, x : A \Vdash M_1 = M_2 : B \quad x \notin FV(M_1) \cup FV(M_2) \cup FV(B)}{\Gamma \Vdash M_1 = M_2 : B} \text{(M-Str-Term)}$$

$$\frac{\Gamma, x : A \Vdash B_1 \leq B_2 \quad x \notin FV(B_1) \cup FV(B_2)}{\Gamma \Vdash B_1 \leq B_2} \text{(M-Str-Sub)}$$

Proof. Easy but tiring. Each implication is proved by induction. The order of proofs matters : in particular you must prove (M-Str-Type) before (M-S-Imp). Again, it is important to notice that the implications (M-Str-...) are true thanks to the non-emptiness of the interpretations of types. □

Corollary 19 *If $\mathfrak{M}$ is extensional and for all (A, ρ), $[A]_\rho \Longrightarrow \llbracket A \rrbracket_\rho \neq \emptyset$ then $\Gamma \vdash J \Longrightarrow \Gamma \Vdash J$.*

Hence we have determined sufficient conditions for our model to be correct. In the following section, we present a method to establish those conditions.

4 Saturation

The idea behind the notion of saturation is to find properties on pers, that are preserved by the combinators used in products ($\overline{\Pi}$ and $\overline{\cap}$). Thus, if such a property holds for $*$ and $\mathcal{E}(a)$, forall $a : *$, then it holds for every $\llbracket A \rrbracket_\rho$.

The main interest of our saturation property is that it is still at an abstract level : it does not rely on one particular λ-model.

Assume $\mathfrak{M} = (\mathfrak{D}, \circ, \llbracket . \rrbracket_.)$ is a λ-model.

Definition 5 $(\mathbf{N}, \mathbf{S})$ *is a stable pair on $\mathfrak{M}$ iff :*

- $\mathbf{N} \neq \emptyset$ *;*
- $\mathbf{N} \subset \mathbf{S} \in \mathbf{Per}(\mathfrak{D})$ *;*
- $\forall n_1 = n_2 : \mathbf{N}$*,* $\forall s_1 = s_2 : \mathbf{S}$*,* $n_1 \circ s_1 = n_2 \circ s_2 : \mathbf{N}$ *;*
- $(\forall n_1 = n_2 : \mathbf{N},\ a_1 \circ n_1 = a_2 \circ n_2 : \mathbf{S}) \Longrightarrow a_1 = a_2 : \mathbf{S}$.

Notice that the last condition could be considered as a somehow *relative extensionality* of $\mathbf{S}$ on $\mathbf{N}$.

In the following, we assume we have a stable pair $(\mathbf{N}, \mathbf{S})$ on $\mathfrak{D}$.

Definition 6 $\mathcal{A} \in \mathbf{Per}(\mathfrak{D})$ *is saturated iff* $\mathbf{N} \subset \mathcal{A} \subset \mathbf{S}$.

Lemma 20 *Assume $*$ is saturated and $\forall a : *$, $\mathcal{E}(a)$ is saturated. Then $[A]_\rho \Longrightarrow \llbracket A \rrbracket_\rho$ is saturated.*

Proof. By induction on the proof of $[A]_\rho$. Set and $\mathsf{El}\ M$ are trivial cases by assumption.
If $[\mathrm{R}x{:}A.B]_\rho$, then $[A]_\rho$ and $\forall a_1 = a_2 : \llbracket A \rrbracket_\rho$, $[B]_{\rho[x \mapsto a_1]} = [B]_{\rho[x \mapsto a_2]}$. Thus $\llbracket A \rrbracket_\rho$ is saturated and $\forall a : \llbracket A \rrbracket_\rho$, $\llbracket B \rrbracket_{\rho[x \mapsto a]}$ is saturated.
$\forall a : \llbracket A \rrbracket_\rho$, $\mathbf{N} \subset \llbracket B \rrbracket_{\rho[x \mapsto a]}$ thus $\mathbf{N} \subset \llbracket \cap x{:}A.B \rrbracket_\rho$. Let $c_1 = c_2 : \llbracket \cap x{:}A.B \rrbracket_\rho$. $\llbracket A \rrbracket_\rho$ is saturated hence non-empty. Let $a : \llbracket A \rrbracket_\rho$, $c_1 = c_2 : \llbracket B \rrbracket_{\rho[x \mapsto a]} \subset \mathbf{S}$. It follows that $\llbracket \cap x{:}A.B \rrbracket_\rho$ is saturated.
Let $n_1 = n_2 : \mathbf{N}$. $\forall a_1 = a_2 : \llbracket A \rrbracket_\rho$, $a_1 = a_2 : \mathbf{S}$. Hence $n_1 \circ a_1 = n_2 \circ a_2 : \mathbf{N} \subset \llbracket B \rrbracket_{\rho[x \mapsto a_1]}$. Thus $\mathbf{N} \subset \llbracket \Pi x{:}A.B \rrbracket_\rho$. Let $c_1 = c_2 : \llbracket \Pi x{:}A.B \rrbracket_\rho$. $\forall n_1 = n_2 : \mathbf{N}$, $\mathbf{N} \subset \llbracket A \rrbracket_\rho$ implies $c_1 \circ n_1 = c_2 \circ n_2 : \llbracket B \rrbracket_{\rho[x \mapsto n_1]} \subset \mathbf{S}$. Hence $c_1 = c_2 : \mathbf{S}$. It follows that $\llbracket \Pi x{:}A.B \rrbracket_\rho$ is saturated. □

Corollary 21 *Assume $\mathfrak{D}$ is an extensional λ-model with a stable pair $(\mathbf{N}, \mathbf{S})$ such that $*$ is saturated and $\forall a : *$, $\mathcal{E}(a)$ is saturated. Then theorem 19 holds.*

Proof. $[A]_\rho \Longrightarrow \llbracket A \rrbracket_\rho$*saturated* $\Longrightarrow \llbracket A \rrbracket_\rho \supset \mathbf{N} \neq \emptyset$. □

In the following section, we use saturation on a particular model to prove consistency.

5 Consistency

We here prove consistency of the type system using saturation[2]. For this purpose, it is sufficient to give a correct model in which the interpretation of $\perp$ as a per contains only interpretations of terms with free variables. This will forbid the existence of a closed term with type $\perp$ in the empty context.

5.1 Per-model on Λ'

Denote $\Lambda' = \Lambda/\equiv_{\beta\eta}$ and $\mathfrak{M}_{\Lambda'}$ the λ-model such that :

- $\mathfrak{D} = \Lambda'$;
- $M \circ N = M\ N$;
- $[\![M]\!]_\rho = M\ \rho$.

Remark Here we identify any term M with its class modulo $\equiv_{\beta\eta}$. The well-definedness of $\circ$ and $[\![.]\!]_.$ relies on the following lemma.

Lemma 22 *If $M_1 \equiv_{\beta\eta} M_2$, $N_1 \equiv_{\beta\eta} N_2$ and $\rho_1 \equiv_{\beta\eta} \rho_2$ then $M_1\ N_1 \equiv_{\beta\eta} M_2 N_2$ and $M_1\ \rho_1 \equiv_{\beta\eta} M_2\ \rho_2$.*

Hence equality on $\mathfrak{M}_{\Lambda'}$ is $\equiv_{\beta\eta}$ on terms.

Lemma 23 *$\mathfrak{M}_{\Lambda'}$ is an extensional λ-model*

Proof. Assume $(\forall P \in \Lambda,\ M\ P \equiv_{\beta\eta} N\ P)$; then let $x \notin FV(M) \cup FV(N)$, $M\ x \equiv_{\beta\eta} N\ x$ hence $M \leftarrow_\eta \lambda x M\ x \equiv_{\beta\eta} \lambda x N\ x \rightarrow_\eta N$. □

We now define a per-model, using $\mathfrak{M}_{\Lambda'}$ as its founding λ-model.

Definition 7 *A term ν is neutral iff it is of form $(x\ N_1\ \ldots\ N_p)$.*

In the following, we write $\mathbf{N}$ for the set of all neutral terms modulo $\equiv_{\beta\eta}$. More precisely :

Definition 8 $M = N : \mathbf{N} \iff (M \equiv_{\beta\eta} \nu,\ N \equiv_{\beta\eta} \nu$ *and ν is neutral*$)$.

Lemma 24 $(\mathbf{N}, \Lambda')$ *is a stable pair.*

Proof. The first three conditions are straightforward. Moreover

$$(\forall \nu \in \mathbf{N},\ (M\ \nu) \equiv_{\beta\eta} (N\ \nu)) \Longrightarrow M \equiv_{\beta\eta} N$$

because variables are neutral terms and the argument used in the proof of lemma 23 applies, hence the last condition. □

Define $* = \mathbf{N}$ and $\forall a : *,\ \mathcal{E}(a) = \mathbf{N}$. Then lemmas 20 and 21 hold.

[2]Saturation techniques are usually involved in strong normalization proofs but our definition of models prevents such a use here. Indeed correctness holds for per-models founded upon extensional λ-models: two $\beta\eta$-equivalent terms share the same interpretation. Hence we could only prove weak normalization using saturation.

5.2 Application

Define the identity environment $\rho_0 \in \mathbf{Val}_{\Lambda'} \;:\; x \mapsto x$.

Lemma 25 *If* $\Gamma \Vdash$ *then* $\rho_0 : \llbracket \Gamma \rrbracket$.

Proof. Induction on Γ. □

Lemma 26 $\bot$ *has no proof in the empty context.*

Proof. Recall that $\bot = \cap A{:}\mathsf{Set}.\mathsf{El}\ A$, so that $\llbracket \bot \rrbracket_{\rho_0} = \mathbf{N}$. Suppose there exists M s.t. $\vdash M : \bot$. Then by correctness, $\Vdash M : \bot$ and in particular $\llbracket M \rrbracket_{\rho_0} : \llbracket \bot \rrbracket_{\rho_0}$ *i.e.* $M : \mathbf{N}$. Hence M has free variables, whereas lemma 2 implies $FV(M) \subset \emptyset$. This is a contradiction. □

6 Implementation

IC was derived from ICC for which type checking is known to be undecidable, and type checking for IC is highly expected to be undecidable too, even though this has not been proved yet.

However, a partial implementation of a type checker for IC has been developped during the training period. It is roughly based upon the following informal algorithm:

- let context Γ, term M and type A be given as input for type checking,
  - if A is an explicit product and M an abstraction, invert rule (Abs);
  - if A is an implicit product, invert rule (Gen);
  - else try to build a "canonical" type B for M in Γ such that $\Gamma \vdash M : B$ and check wether $\Gamma \vdash B \leq A$;
- let context Γ and types A and B be given as input for subtyping ($\Gamma \vdash A \leq B$?) ,
  - if nor A nor B is an implicit product, or if B is an implicit product, invert rule (S-Set) or (S-El) or (S-Exp) or (S-Imp);
  - else try to invert (Imp-S) and create a meta-variable for the term involved in the rule.

The idea is that inverting (S-El) produces $\beta\eta$-equations on terms with meta-variables that can hopefully be solved so as to determinate every involved meta-variable.

Hence the output of the algorithm can be of several kinds:

- yes ($\Gamma \vdash M : A$);
- no ($\Gamma \not\vdash M : A$);
- unsolved: a set of $\beta\eta$-equations on terms with meta-variables which, if solved, ensure derivability of the typing judgement;
- failure: could not produce a canonical type (typically when trying to infer a type for term $(\theta\ N)$ where θ is an unsolved meta-variable).

```
A     : *
x     : A
y     : A
eq    : A->A->*
f     : A->A->A
com   : {x:A}{y:A} (eq (f x y) (f y x))
cong  : {x:A}{y:A}{z:A}{t:A} (eq x y) -> (eq z t) -> (eq (f x z) (f y t))
ref   : {x:A} (eq x x)
sym   : {x:A}{y:A} (eq x y) -> (eq y x)
assoc : {x:A}{y:A}{z:A} (eq (f x (f y z)) (f (f x y) z))
lem   : {x:A}{y:A}{z:A} (eq x y) -> (eq (f x z) (f y z)) = [h](cong h ref)
```

Figure 2: Typed equality

```
A  : *
B  : *
id : ({x:A} B) -> B = [x]x
```

Figure 3: Ilustrates 1.2.3

This implementation, if still not satisfactory, is however powerfull enough to give a definitive answer on some interesting inputs like those in figures 2 to 5. Note the conciseness of proof terms.

The syntax is basically as follows: `*` denotes Set, `(x:A)B` denotes $\Pi x{:}A.B$ and `{x:A}B` denotes $\cap x{:}A.B$; as terms can never be mistaken with types, we can omit the El construction and write `M` for $\mathsf{El}\ M$. A line of the form `x:A` checks well-formedness of type A in the current context Γ, then adds $x : A$ to Γ. A line of the form `x:A=M` checks wether $\Gamma \vdash M : A$ then adds $x : A$ to Γ.

7 Further work

7.1 Subject reduction

Subject reduction in IC *i.e.* $(\Gamma \vdash M : A \wedge M \rightarrow_{\beta\eta} N) \Longrightarrow \Gamma \vdash N : A$, hasn't been formally proved yet. In particular, it is necessary to prove an inversion lemma of the following form:

Conjecture 27 *If* $\Gamma \vdash M : \cap x{:}A.B$ *with* $x \notin DV(\Gamma)$, *or* $\Gamma \vdash \lambda x M : \Pi x{:}A.B$ *then* $\Gamma, x : A \vdash M : B$. *If* $\Gamma \vdash \cap x{:}A.B$ *or* $\Gamma \vdash \Pi x{:}A.B$ *then* $\Gamma, x : A \vdash B$.

It couldn't be proved directly in ICC because of the presence of strengthening rules, and a typing rule for η-reduction:

$$\frac{\Gamma \vdash \lambda x(M\ x) : A \quad x \notin FV(A)}{\Gamma \vdash M : A}\ (\textsc{Ext})$$

which is not exactly reproduced here; notice however that rule (S-El) bears some similarities to (Ext).

```
perp     : *
not      : * -> *
notE     : {A:*} (not A) -> A -> perp
notI     : {A:*} (A->perp) -> (not A)
list     : (A:*) *
nil      : {A:*} (list A)
eq       : {A:*} A -> A -> *
head     : {A:*} (x:(list A)) {p:(not (eq x nil))} (list A)
unwanted : {A:*} (list A) = (head nil)
```

Figure 4: Illustrates 1.3

```
eq    : {A:*} A -> A -> *
trans : {A:*}{x:A}{y:A}{z:A} (eq x z) -> (eq y z) -> (eq x y)
id    : {A:*} (A->A) -> (A->A) -> *
k     : {A:*} (A->A) -> (A->A) -> (A->A)
axk   : {A:*}{x:A->A}{y:A->A} (id (k x y) x)
N     : *
z     : N
B     : *
true  : B
f     : {A:*} (A->A) -> N
axf   : {A:*}{x:A->A}{y:A->A} (id x y) -> (eq (f x) (f y))
rem1  : (eq (f (k ([x]x) ([x]z))) (f ([x]x)))
      = (axf axk)
rem2  : (eq (f (k ([x]x) ([x]true))) (f ([x]x)))
      = (axf axk)
lem   : (eq (f (k ([x]x) ([x]z))) (f (k ([x]x) ([x]true))))
      = (trans rem1 rem2)
```

Figure 5: Non-conservativity (IC is more flexible than CC)

7.2 IC as a logical framework

A (yet unsolved) problem arise when attempting to get a result similar to those of [HHP87] and [NL98]. These papers state an adequation between terms, formulas and proofs of some logical systems such as natural deduction for first order logic on one hand and canonical terms in well chosen types in a type system designed for that purpose (LF in [HHP87]) on the other hand.

Canonical terms are normal terms in which every variable is fully applied (*i.e.* every variable x occurs in a subterm of the form $(x\ N_1\ \dots\ N_{n(x)})$ where $n(x)$ is the arity of x). Canonicity can be defined in IC, but it is not as simple as in LF because bound variables have no type annotation and there is no unicity of typing.

However, adequation relies heavily on inversion lemma, which hasn't been proved yet for IC. Hence, here again, inversion lemma is an obligatory step to further work.

7.3 Universal calculus

One interesting study would be that of a calculus with universal constructions, to be related with implicit constructions. This would be an intermediate between a calculus with explicit dependent product only and IC.

The idea is to allow new terms:

$$M ::= x \mid (M\ M) \mid \lambda x M \mid \nu x M \mid (M\ |M)$$

and replace rules (Gen) and (Inst) with:

$$\frac{\Gamma, x : A \vdash M : B \quad x \notin FV(|M|)}{\Gamma \vdash \nu x M : \cap x{:}A.B}\ (\text{Gen'}) \frac{\Gamma \vdash M : \cap x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash (M\langle N\rangle) : B(N/x)}\ (\text{Inst'})$$

νM and $M\langle N\rangle$ are annotations used for typing only. $|M|$ is the pure λ-term obtained by erasing annotations. Typechecking in this system becomes decidable and we can define a translation from universal to implicit constructions by a forget map.

This calculus would be particularly interesting in an implementation in which one could write $(M?)$ instead of $(M\langle N\rangle)$ when N is clear from the context. The main advantage is that once a term with type annotations is typed in the calculus with universal constructions, we can apply the forget map to erase subterms that are relevant for typing only and obtain a pure lambda-term with the same structure and that has the same type in IC.

Moreover, syntactical properties such as the inversion lemma are easy to establish in this calculus. Hence this could be a solution to create a logical framework close to IC.

Part of this study has been carried out in [Vau04].

References

[CPT] Thierry Coquand, Randy Pollack, and Makoto Takeyama. A logical framework with dependently typed records. `http://www.cs.chalmers.se/~coquand/MODULE/main.html`.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings 2nd Annual IEEE Symp. on Logic in Computer Science, LICS'87, Ithaca, NY, USA, 22–25 June 1987*, pages 194–204. IEEE Computer Society Press, New York, 1987.

[Miq98] Alexandre Miquel. *Arguments implicites dans le Calcul des Constructions*. Rapport de stage de DEA, Université Paris 7, 1998. `http://pauillac.inria.fr/~miquel/index.html`.

[Miq01] Alexandre Miquel. *Le Calcul des Constructions Implicite : syntaxe et sémantique*. PhD thesis, Université Paris 7, dec. 2001. `http://pauillac.inria.fr/~miquel/index.html`.

[NL98] George C. Necula and Peter Lee. Efficient representation and validation of proofs. In *Logic in Computer Science*, pages 93–104, 1998.

[Vau03] Lionel Vaux. Un calcul des constructions implicite: étude d'une variante du calcul d'alexandre miquel. `http://perso.ens-lyon.fr/lionel.vaux/`, 2003.

[Vau04] Lionel Vaux. A note on an implicit calculus with annotated terms: introducing universal dependent types. `http://perso.ens-lyon.fr/lionel.vaux/`, 2004.