A classical version of system F_ω

Stéphane Lengrand[1,2] and Alexandre Miquel[1]
[1] PPS & Université Paris 7
175 rue du Chevaleret, 75013 Paris, France
[2] School of Computer Science, University of St Andrews
North Haugh, St Andrews, Fife, KY16 9SX, Scotland

Abstract

We present a version of system F_ω in which the layer of type constructors is essentially the same as in the traditional presentation of F_ω [Gir72], whereas provability of types is classical. The proof-term calculus accounting for the classical reasoning is a variant of Barbanera and Berardi's symmetric λ-calculus [BB96].

We prove that the whole calculus is strongly normalising. For the layer of type constructors, we use Tait and Girard's reducibility method combined with orthogonality techniques. For the (classical) layer of terms, we use Barbanera and Berardi's method based on a symmetric notion of reducibility candidate which does not seem to be captured by orthogonality.

1 Introduction

Approaches to a Curry-Howard correspondence for classical logic seem to converge towards the idea of programs equipped with some notion of control [Par92, BB96, Urb00, Sel99, CH00]. The general notion of reduction/computation is non-confluent but there are possible ways to restrict reductions and thus recover confluence. Two such canonical ways are related to CBV and CBN, with associated semantics given by CPS-translations, which correspond to the usual encodings of classical logic into intuitionistic logic known as "not-not"-translations.

It is then tempting to try and build, on such a correspondence for classical logic, powerful type theories, such as those developed in intuitionistic logic (Pure Type Systems, Martin-Löf type theories).

The latter however exploit the fact that predicates are pure functions, which, when fully applied, give rise to formulae with logical meanings. The Curry-Howard correspondence in intuitionistic logic can then describe these pure functions as the inhabitants of implicative types in a higher type layer (often called the layer of kinds).

On the other hand, inhabitants of implicative types in classical logic can be much wilder than pure functions (owing to the aforementioned notion of control), so it is not clear what meaning could be given to those simili-predicates, built from classical inhabitants of implicative types, and whose reductions may not even be confluent.

However this is for the layer of types, which various type theories cleanly separate from the layer of terms. This is what system F_ω [Gir72] does as a completely intuitionitic system.

This paper shows that it is perfectly safe to have cohabiting layers with different logics, provided that the layer of types is free from any dependency on terms, i.e. that the system has no dependent types.

System F_ω is thus the most powerful system of Barendregt's Cube without such dependencies [Bar91]. We present here a version of it called $F_\omega^{\mathcal{C}}$ that is classical in the following sense:

The layer of type constructors is purely functional, i.e. intuitionistic: it is in fact the lambda-calculus extended with constants for logical connectives. Then, for those objects of the layer that are formulae, we have a notion of provability with proof derivations and proof-terms, which is here classical instead of intuitionistic.

Obviously, weaker systems than F_ω (System F, simply-typed λ-calculus) also cleanly separate the layer of types from that of terms, but the former is trivial as no computation happens there.

System F_ω features notions of computation in both layers, both strongly normalising. But in contrast to the traditional intuitionistic version, our classical version $F_\omega^{\mathcal{C}}$ has two *different* notions of computation (one intuitionistic and confluent, the other one classical and non-confluent), so two different proof techniques are used. This system is thus particularly adequate to illustrate and compare these two techniques.

The former, used to prove strong normalisation of the intuitionistic layer of types, adapts Tait and Girard's reducibility method, rephrased with the terminology and concepts of orthogonality, which provides a high level of abstraction and potential for modularity.

The latter, used to prove strong normalisation of the classical layer of terms, adapts Barbanera and Berardi's method based on a symmetric notion of reducibility candidate [BB96]. To our knowledge, the method had not been pushed to such a typing system as that of F_ω, but it works without any surprise. Difficulties would come with dependent types (the only feature of Barendregt's Cube missing here), precisely because they would pollute the layer of types with non-confluence and unclear semantics.

Finally, the main purpose of presenting these two proof techniques in the same paper is to express them whilst pointing out similarities, and to examine whether or not the concepts of the symmetric candidates method can be captured by the concept of orthogonality. Such an open problem is raised in the conclusion.

Section 2 introduces $F_\omega^{\mathcal{C}}$, section 3 establishes strong normalisation of the layer of types, section 4 establishes strong normalisation of the layer of terms, and section 5 establishes some logical properties of F_ω such as consistency.

2 Syntax, Reduction and Typing of $F_\omega^{\mathcal{C}}$

2.1 Syntax

$F_\omega^{\mathcal{C}}$ distinguishes four syntactic categories: *kinds*, *type constructors* (or *constructors* for short), *terms* and *programs*:

Kinds $\quad K, K' ::= \star \mid K \to K'$

Constructors $\quad A, B, C, \ldots ::= \alpha \mid \alpha^\perp \mid \lambda\alpha\!:\!K\,.\,B \mid B\ A$
$\quad\mid A \wedge B \mid A \vee B$
$\quad\mid \forall\alpha\!:\!K\,.\,B \mid \exists\alpha\!:\!K\,.\,B$

Terms $\quad t, u, v, \ldots ::= x \mid \mu x^A.p$
$\quad\mid \langle t, u\rangle \mid \lambda x^A y^B.c$
$\quad\mid \lambda\alpha\!:\!K\,.\,t \mid \langle A, t\rangle$

Programs $\quad p ::= \{t \mid u\}$

Kinds, that are exactly the same as in system F_ω [Gir72, BG01], are a system of simple types for type constructors. (We use the word 'kind' to distinguish kinds from the types which appear at the level of type constructors.) The basic kind $\star$ is the kind of *types*, that is, the kind of all type constructors that represent types of terms—or *propositions* through the Curry-Howard correspondence.

Type constructors are basically simply-typed λ-terms with two binary operators $A \wedge B$ (conjunction), $A \vee B$ (disjunction) and two extra binders $\forall\alpha : K . A$ and $\exists\alpha : K . A$ to represent universal and existential quantification. (There is no primitive implication in the system.)

As in linear logic, *negation* $\alpha \mapsto \alpha^\perp$ is only primitive on variables, but the extension as an involution $A \mapsto A^\perp$ on all type constructors is defined via de Morgan laws:

$$\begin{array}{rlrl}
(\alpha)^\perp &= \alpha^\perp & (\alpha^\perp)^\perp &= \alpha \\
(A \wedge B)^\perp &= A^\perp \vee B^\perp & (A \vee B)^\perp &= A^\perp \wedge B^\perp \\
(\forall\alpha : K . B)^\perp &= \exists\alpha : K . B^\perp & (\exists\alpha : K . B)^\perp &= \forall\alpha : K . B^\perp \\
(\lambda\alpha : K . B)^\perp &= \lambda\alpha : K . B^\perp & (B\ A)^\perp &= B^\perp\ A
\end{array}$$

Notice how negation propagates through λ-abstraction and application. In our calculus, the notation $A^\perp$ is not only meaningful for types (that is, constructors of kind $\star$), but it is defined for all type constructors.

With negation extended to all type constructors we can define implication $A \Rightarrow B$ as $(A^\perp) \vee B$.

However, one must take care in the way constructor variables are bound. In what follows, we assume that the constructions $\forall\alpha : K . B$, $\exists\alpha : K . B$ and $\lambda\alpha : K . B$ bind all free occurrences of the variable α in B, including those which correspond to a subterm of the form $\alpha^\perp$. (In other words, the syntactic construction $\alpha^\perp$ is not a variable.) For instance, the type constructor

$$\neg \quad = \quad \lambda\alpha : \star . \alpha^\perp$$

is closed; this is the type constructor which represents negation as a function of kind $\star \rightarrow \star$. The computation rules of negation are incorporated in the calculus by extending the definition of the (external) operation of substitution written $B\{\alpha\backslash A\}$ to the case where B is a negated variable, setting:

$$\begin{array}{rcll}
\alpha^\perp\{\alpha\backslash A\} &=& A^\perp & \\
\beta^\perp\{\alpha\backslash A\} &=& \beta^\perp & \qquad\qquad (\text{if } \beta \neq \alpha)
\end{array}$$

(The notation $B\{\alpha\backslash A\}$ is defined as usual for the other constructions.)

The (proof-)terms of our calculus are basically the terms of Barbanera and Berardi's symmetric λ-calculus, with the difference that connectives are treated multiplicatively. In particular, disjunction is treated as a negative connective whose proofs are built using a double binder written $\lambda x^A y^B . p$. On the other hand, proofs of conjunction are introduced as usual, using the pairing construct written $\langle t, u \rangle$.

Finally, programs are built by making two terms t and u interact using a construction written $\{t \mid u\}$, where each term can be understood as the evaluation context of the other term. We assume that this construction is symmetric, that is, that $\{t \mid u\}$ and $\{u \mid t\}$ denote the same program. Henceforth, terms and programs are considered up to this equality together with α-conversion.

2.2 Reduction and Typing for Types

The reduction relation on the layer of type constructors is β-reduction, which is defined as usual as the contextual closure of the relation

$$(\lambda\alpha\,{:}\,K\,.\,B)A \quad \longrightarrow_\beta \quad B\{\alpha\backslash A\}\,.$$

However, the extension of the definition of substitution to negated variables mechanically enhances β-reduction in such a way that we get de Morgan equalities for free:

$$\begin{array}{rclcrcl} \neg(A\wedge B) & =_\beta & \neg A\vee\neg B & \qquad & \neg(A\vee B) & =_\beta & \neg A\wedge\neg B \\ \neg(\forall\alpha\,{:}\,K\,.\,B) & =_\beta & \exists\alpha\,{:}\,K\,.\,\neg B & & \neg(\exists\alpha\,{:}\,K\,.\,B) & =_\beta & \forall\alpha\,{:}\,K\,.\,\neg B \end{array}$$

(Here, $\neg$ denotes the type constructor $\lambda\alpha\,{:}\,\star\,.\,\alpha^\perp$.)

Proposition 1 — *The (enhanced) β-reduction on type constructors is confluent.*

Proof: This is proved by introducing the corresponding notion of parallel reduction, following Tait and Martin-Löf [Bar84]. $\square$

Typing contexts for variables of type constructors, that we call *signatures*, are (unordered) lists of declarations of the form $(\alpha : K)$:

Signatures $\qquad \Sigma \ ::= \ \alpha_1 : K_1, \ldots, \alpha_n : K_n$

The inference rules of the typing judgment $\Sigma \vdash A : K$ ('In the signature Σ, A is a constructor of kind K') are given in Fig. 1.

$$\frac{}{\Sigma\vdash\alpha : K}\,(\alpha : K)\in\Sigma \qquad \frac{}{\Sigma\vdash\alpha^\perp : K}\,(\alpha : K)\in\Sigma$$

$$\frac{\Sigma,\alpha : K\vdash B : K'}{\Sigma\vdash\lambda\alpha\,{:}\,K\,.\,B : K\to K'} \qquad \frac{\Sigma\vdash B : K\to K' \qquad \Sigma\vdash A : K}{\Sigma\vdash B\;A : K'}$$

$$\frac{\Sigma\vdash A : \star \qquad \Sigma\vdash B : \star}{\Sigma\vdash A\wedge B : \star} \qquad \frac{\Sigma\vdash A : \star \qquad \Sigma\vdash B : \star}{\Sigma\vdash A\vee B : \star}$$

$$\frac{\Sigma,\alpha : K\vdash B : \star}{\Sigma\vdash\forall\alpha\,{:}\,K\,.\,B : \star} \qquad \frac{\Sigma,\alpha : K\vdash B : \star}{\Sigma\vdash\exists\alpha\,{:}\,K\,.\,B : \star}$$

Figure 1: Typing rules for type constructors

The type system for type constructors enjoys good properties such as:

Proposition 2 (Subject-reduction) — *If $\Sigma\vdash A : K$ and if $A \longrightarrow_\beta A'$, then $\Sigma\vdash A' : K$.*

2.3 Reduction and Typing for Terms and Programs

At the level of (proof-)terms and programs, reduction is defined by the rules of Fig. 2.

$$\begin{array}{rll}
\{\mu x^A.p \mid t\} & \longrightarrow_{\mu} & p\{x\backslash t\} \\
\{\langle t_1, t_2\rangle \mid \lambda {x_1}^A {x_2}^B.p\} & \longrightarrow_{\wedge\vee_l} & \{t_1 \mid \mu {x_1}^A.\{t_2 \mid \mu {x_2}^B.p\}\} \\
\text{or} & \longrightarrow_{\wedge\vee_r} & \{t_2 \mid \mu {x_2}^B.\{t_1 \mid \mu {x_1}^A.p\}\} \\
\{\lambda\alpha : K . t \mid \langle A, u\rangle\} & \longrightarrow_{\forall\exists} & \{t\{\alpha\backslash A\} \mid u\}
\end{array}$$

Figure 2: Reduction rules on terms and programs

As in Barbanera and Berardi's symmetric λ-calculus [BB96] or in Curien and Herbelin's $\lambda\mu\tilde{\mu}$-calculus [CH00], the critical pair

$$\begin{tikzcd}
 & \{\mu x^A.p \mid \mu y^{A'}.q\} \arrow[dl] \arrow[dr] & \\
p\{x\backslash \mu y^{A'}.q\} & & q\{y\backslash \mu x^A.p\}
\end{tikzcd}$$

cannot be closed, so that reduction is not confluent at this level.

Typing contexts for variables of terms, that we simply call *contexts*, are lists of declarations of the form $(x : A)$:

Contexts $\qquad \Gamma \ ::= \ x_1 : A_1, \ldots, x_n : A_n$

Since types A that appear in a context may depend on constructor variables, each context Γ only makes sense in a given signature Σ. In what follows, we say that a context Γ is *well-formed* in a signature Σ and write $\mathsf{wf}_\Sigma(\Gamma)$ if for all declarations $(x : A) \in \Gamma$, the judgment $\Sigma \vdash A : \star$ is derivable.

From this, we define two judgments, namely:

$\Gamma \vdash_\Sigma t : A$	'In the signature Σ and context Γ, the term t has type A'
$\Gamma \vdash_\Sigma p \diamond$	'In the signature Σ and context Γ, the program p is well-formed'

Both judgments are defined by mutual induction from the rules given in Fig. 3.

$$\frac{\mathsf{wf}_\Sigma(\Gamma)}{\Gamma \vdash_\Sigma x : A}\ (x : A) \in \Gamma \qquad \frac{\Gamma, x : A \vdash_\Sigma p \diamond}{\Gamma \vdash_\Sigma \mu x^A.p : A^\perp}$$

$$\frac{\Gamma \vdash_\Sigma t : A \qquad \Gamma \vdash_\Sigma u : B}{\Gamma \vdash_\Sigma \langle t, u\rangle : A \wedge B} \qquad \frac{\Gamma, x : A, y : B \vdash_\Sigma p \diamond}{\Gamma \vdash_\Sigma \lambda x^A y^B.p : A^\perp \vee B^\perp}$$

$$\frac{\Gamma \vdash_{\Sigma, \alpha : K} t : B}{\Gamma \vdash_\Sigma \lambda\alpha : K . t : \forall\alpha : K . B} \qquad \frac{\Sigma \vdash A : K \qquad \Gamma \vdash_\Sigma u : B\{\alpha\backslash A\}}{\Gamma \vdash_\Sigma \langle A, u\rangle : \exists\alpha : K . B}$$

$$\frac{\Gamma \vdash_\Sigma t : A \qquad \Sigma \vdash A' : \star}{\Gamma \vdash_\Sigma t : A'}\ A =_\beta A' \qquad \frac{\Gamma \vdash_\Sigma t : A \qquad \Gamma \vdash_\Sigma u : A^\perp}{\Gamma \vdash_\Sigma \{t \mid u\} \diamond}$$

Figure 3: Typing rules for terms and programs

Again, the type system for proof-terms enjoys the subject-reduction property, despite the non-deterministic nature of reduction:

Proposition 3 (Subject-reduction)

1. *If* $\Gamma \vdash_\Sigma t : A$ *and* $t \longrightarrow_{\mu,\wedge\vee_l,\wedge\vee_r,\forall\exists} t'$*, then* $\Gamma \vdash_\Sigma t' : A$*.*
2. *If* $\Gamma \vdash_\Sigma p \diamond$ *and* $p \longrightarrow_{\mu,\wedge\vee_l,\wedge\vee_r,\forall\exists} p'$*, then* $\Gamma \vdash_\Sigma p' \diamond$*.*

Proof: By mutual induction on the judgments $\Gamma \vdash_\Sigma t : A$ and $\Gamma \vdash_\Sigma p \diamond$. □

Example 1 Here is a proof of the Law of excluded middle:

$$\dfrac{\dfrac{\dfrac{x\colon \alpha^\perp, y\colon \alpha \vdash_{\alpha:\,\star} x : \alpha^\perp \qquad x\colon \alpha^\perp, y\colon \alpha \vdash_{\alpha:\,\star} y : \alpha}{x\colon \alpha^\perp, y\colon \alpha \vdash_{\alpha:\,\star} \{x \mid y\} \diamond}}{\vdash_{\alpha:\,\star} \lambda x^{\alpha^\perp} y^\alpha.\{x \mid y\} : \alpha \vee (\alpha^\perp)}}{\vdash \lambda\alpha:\star.\,\lambda x^{\alpha^\perp} y^\alpha.\{x \mid y\} : \forall\alpha:\star.\,\alpha \vee (\alpha^\perp)}$$

Example 2 Here is Lafont's example of non-confluence. Suppose $\Gamma \vdash_{\alpha:\,\star} p_1 \diamond$ and $\Gamma \vdash_{\alpha:\,\star} p_2 \diamond$. With $x \notin \mathcal{FV}(p_1)$ and $y \notin \mathcal{FV}(p_2)$, by weakening (admissible in $F_\omega^{\mathcal{C}}$) we get

$$\dfrac{\dfrac{\Gamma, x\colon \alpha \vdash_{\alpha:\,\star} p_1 \diamond}{\Gamma \vdash_{\alpha:\,\star} \mu x^\alpha.p_1 : \alpha^\perp} \qquad \dfrac{\Gamma, y\colon \alpha^\perp \vdash_{\alpha:\,\star} p_2 \diamond}{\Gamma \vdash_{\alpha:\,\star} \mu y^{\alpha^\perp}.p_2 : \alpha}}{\Gamma \vdash_{\alpha:\,\star} \{\mu x^\alpha.p_1 \mid \mu y^{\alpha^\perp}.p_2\} \diamond}$$

But $\{\mu x^\alpha.p_1 \mid \mu y^{\alpha^\perp}.p_2\} \longrightarrow_\mu^* p_1$ or $\{\mu x^\alpha.p_1 \mid \mu y^{\alpha^\perp}.p_2\} \longrightarrow_\mu^* p_2$. And unless the system is proof-irrelevant, p_1 and p_2 can be completely different.

Note that, in constrast to Barbanera and Berardi's symmetric λ-calculus, our design choices for the typing rules are such that, by constraining terms and programs to be linear, we get exactly the multiplicative fragment of linear logic [Gir87].

3 Strong normalisation of type constructors

We now prove that all well-typed constructors are strongly normalisable. For that, let us write SN_C the set of all strongly normalisable type constructors.

We call a *stack* (of type constructors) any finite sequence $S = (A_1, \ldots, A_n)$ of type constructors. Given a type constructor B and a stack $S = (A_1, \ldots, A_n)$, we define the application BS by setting $BS = BA_1 \cdots A_n$.

We say that a stack $S = (A_1, \ldots, A_n)$ is strongly normalisable when all its elements $A_1, \ldots, A_n$ are strongly normalisable. The set of all strongly normalisable stacks is written SN_C^*. In general, applying a strongly normalisable constructor $B \in \mathsf{SN}_\mathsf{C}$ to a strongly normalisable stack $S \in \mathsf{SN}_\mathsf{C}^*$ does not yield a strongly normalisable constructor BS. In the case where $BS \in \mathsf{SN}_\mathsf{C}$, we thus say that B and S are *orthogonal*, and write $B \perp S$.

Given a subset $X \subset \mathsf{SN}_\mathsf{C}$, we write $X^\perp$ the subset of SN_C^* called the *orthogonal of* X and defined by

$$X^\perp \quad = \quad \{S \in \mathsf{SN}_\mathsf{C}^* \mid B \perp S \text{ for all } B \in X\}\,.$$

(The orthogonal $Y^\perp \subset \mathsf{SN}_\mathsf{C}$ of a subset $Y \subset \mathsf{SN}_\mathsf{C}^*$ is defined dually.) As for any negation defined by orthogonality, the operation $X \mapsto X^\perp$ fulfils the following properties on SN_C (as well as on SN_C^*):

1. $X \subset Y$ entails $Y^\perp \subset X^\perp$ (contravariance)
2. $X \subset X^{\perp\perp}$ (closure)

3. $X^{\perp\perp\perp} = X^{\perp}$ (tri-orthogonal)

Definition 1 (Reducibility candidate) — We call a *reducibility candidate* any subset $X \subset \mathsf{SN_C}$ such that $X = X^{\perp\perp}$.

Notice that reducibility candidates are precisely the subsets $X \subset \mathsf{SN_C}$ of the form $X = Y^{\perp}$ for some subset $Y \subset \mathsf{SN}^*_\mathsf{C}$. In particular, $\mathsf{SN_C}$ is a reducibility candidate, since $\mathsf{SN_C} = \{()\}^{\perp}$ (writing () the empty stack).

Reducibility candidates enjoy the following properties:

Proposition 4 — *For all reducibility candidates X:*

1. *$X \subset \mathsf{SN_C}$;*
2. *X contains all variables α and negated variables $\alpha^{\perp}$;*
3. *X is closed under β-reduction, that is:*
   if $B \in X$ and $B \longrightarrow_\beta B'$, then $B' \in X$;
4. *X is closed under head β-expansion, that is:*
   if $B\{\alpha\backslash A\} \in X$ and $A \in \mathsf{SN_C}$, then $(\lambda\alpha : K \,.\, B)A \in X$.

Proof: Item 1 holds by definition. Item 2 holds since αS (resp. $\alpha^{\perp}S$) is strongly normalisable as soon as the stack S is strongly normalisable. Item 3 holds since strongly normalisable type constructors are closed under β-reduction. □

Finally, item 4 is a consequence of the following lemma:

Lemma 5 — *If the type constructors A and $B\{\alpha\backslash A\}A_1 \cdots A_n$ are strongly normalisable, then so is $(\lambda\alpha : K \,.\, B)AA_1 \cdots A_n$.*

Given two subsets $X, Y \subset \mathsf{SN_C}$, we write $X \to Y = \{B \mid \forall A{\in}X \;\; (BA){\in}Y\}$.

Lemma 6 — *If both X and Y are reducibility candidates, then so is $X \to Y$.*

Proof: First remark that for all subsets $X \subset \mathsf{SN_C}$ and $Z \subset \mathsf{SN}^*_\mathsf{C}$ one has:

$$X \to Z^{\perp} \quad = \quad (X :: Z)^{\perp} \quad = \quad \{A :: S \mid A{\in}X \text{ and } S{\in}Z\}^{\perp}$$

(where A, S denotes the consing operation on stacks). Then, if Y is a reducibility candidate, we get $X \to Y = X \to Y^{\perp\perp} = (X :: Y^{\perp})^{\perp}$. □

From this, we interpret each kind K as a reducibility candidate written $[K]$ and recursively defined by $[\star] = \mathsf{SN_C}$ and $[K \to K'] = [K] \to [K']$.

Lemma 7 — *If the typing judgment $\alpha_1 : K_1, \ldots, \alpha_n : K_n \vdash B : K$ is derivable, then for all $A_1 \in [K_1]$, …, $A_n \in [K_n]$ one has*

$$B\{\alpha_1, \ldots, \alpha_n \backslash A_1, \ldots, A_n\} \in [K]$$

(where $B\{\alpha_1, \ldots, \alpha_n \backslash A_1, \ldots, A_n\}$ denotes the parallel substitution of the type constructors $A_1, \ldots, A_n$ to the variables $\alpha_1, \ldots, \alpha_n$ in the type constructor B).

Proof: By induction on the derivation of $\alpha_1 : K_1, \ldots, \alpha_n : K_n \vdash B : K$. □

From this we get:

Theorem 8 — *It $\Sigma \vdash B : K$, then B is strongly normalisable.*

Proof: Apply lemma 7 with $A_1 = \alpha_1$, …, $A_n = \alpha_n$ (identity substitution), using item 2 of Prop. 4. □

4 Strong normalisation of terms

This proof is adapted from [BB96, Pol04, DGLL05].

Let us consider terms and programs without their type annotations, a.k.a. Curry-style terms and programs, whose syntax is the following:

Curry-style terms	$t, u, v, \ldots$	$::=$	$x \mid \mu x.p \mid \langle t, u\rangle \mid \lambda xy.p \mid \lambda\_.t \mid \langle\_, t\rangle$
Curry-style programs	p	$::=$	$\{t \mid u\}$

The corresponding reduction rules, that are shown in Fig. 4, define the set SN of Curry-style terms and Curry-style programs.

$$
\begin{array}{lcl}
\{\mu x.p \mid t\} & \longrightarrow_{\mu} & p\{x\backslash t\} \\
\{\langle t_1, t_2\rangle \mid \lambda x_1 x_2.p\} & \longrightarrow & \{t_1 \mid \mu x_1.\{t_2 \mid \mu x_2.p\}\} \\
 & \text{or} & \{t_2 \mid \mu x_2.\{t_1 \mid \mu x_1.p\}\} \\
\{\lambda\_.t \mid \langle\_, u\rangle\} & \longrightarrow & \{t \mid u\}
\end{array}
$$

Figure 4: Reductions without types

Definition 2 — The type-erasure operation from terms (resp. programs) to Curry-style terms (resp. Curry-style programs) is recursively defined by:

$$
\begin{array}{ll}
\|x\| & = x \\
\|\langle t, u\rangle\| & = \langle \|t\|, \|u\|\rangle \\
\|\lambda x^A y^B.p\| & = \lambda xy.\|p\| \\
\|\mu x^A.p\| & = \mu x.\|p\| \\
\|\lambda\alpha : K\,.\,t\| & = \lambda\_.\|t\| \\
\|\langle A, t\rangle\| & = \langle\_, \|t\|\rangle \\
\|\{t \mid u\}\| & = \{\|t\| \mid \|u\|\}
\end{array}
$$

Lemma 9 — *Provided all types in a term t are strongly normalising (for β), if $\|t\| \in \mathsf{SN}$ then $t \in \mathsf{SN}$.*

Proof: Let $\mathcal{M}(t)$ be the multiset of all the types and kinds appearing in t, equipped with the multiset order based on the terminating β-reduction on types.
Every reduction from t decrease the pair $(\|t\|, \mathcal{M}(t))$ in lexicographic order. $\square$

Definition 3 (Orthogonality)
- We say that that a Curry-style term t is *orthogonal* to a Curry-style term u, written $t \perp u$, if $\{t \mid u\} \in \mathsf{SN}$.
- We say that that a set $\mathcal{U}$ of Curry-style terms is *orthogonal* to a set $\mathcal{V}$ of Curry-style terms, written $\mathcal{U} \perp \mathcal{V}$, if $\forall t \in \mathcal{U}, \forall u \in \mathcal{V}, t \perp u$.

Remark 10 — If $t\{x\backslash v\} \perp u\{x\backslash v\}$, then $t \perp u$ and $\mu x.\{t \mid u\} \in \mathsf{SN}$.

Definition 4 — A set $\mathcal{U}$ of Curry-style terms is *simple* if it is non-empty and it contains no Curry-style term of the form $\mu x.p$.

Definition 5 — A pair $(\mathcal{U}, \mathcal{V})$ of sets of Curry-style terms is *saturated* if:

- $\mathsf{Var} \subseteq \mathcal{U}$ and $\mathsf{Var} \subseteq \mathcal{V}$
- $\{\mu x.\{t \mid u\} \mid \forall v \in \mathcal{V}, t\{x\backslash v\} \perp u\{x\backslash v\}\} \subseteq \mathcal{U}$
and $\{\mu x.\{t \mid u\} \mid \forall v \in \mathcal{U}, t\{x\backslash v\} \perp u\{x\backslash v\}\} \subseteq \mathcal{V}$.

Definition 6 — Whenever $\mathcal{U}$ is simple, we define the following function
$\Phi_{\mathcal{U}}(\mathcal{V}) = \mathcal{U} \cup \mathsf{Var} \cup \{\mu x.\{t \mid u\} \mid \forall v \in \mathcal{V}, t\{x\backslash v\} \perp u\{x\backslash v\}\}$.

Remark 11 — For all simple $\mathcal{U}$, $\Phi_{\mathcal{U}}$ is anti-monotone. Hence, for any simple $\mathcal{U}$ and $\mathcal{V}$, $\Phi_{\mathcal{U}} \circ \Phi_{\mathcal{V}}$ is monotone, so it admits a fixed point $\mathcal{U}' \supseteq \mathcal{U}$.

Theorem 12 — *Assume that $\mathcal{U}$ and $\mathcal{V}$ are simple with $\mathcal{U} \perp \mathcal{V}$.*
There exist $\mathcal{U}'$ and $\mathcal{V}'$ such that $\mathcal{U} \subseteq \mathcal{U}'$ and $\mathcal{V} \subseteq \mathcal{V}'$, $\mathcal{U}' \perp \mathcal{V}'$ and $(\mathcal{U}', \mathcal{V}')$ is saturated.

Proof: Let $\mathcal{U}'$ be a fixed point of $\Phi_{\mathcal{U}} \circ \Phi_{\mathcal{V}}$, and let $\mathcal{V}' = \Phi_{\mathcal{V}}(\mathcal{U}')$. We have

$$\begin{aligned}
\mathcal{U}' &= \Phi_{\mathcal{U}}(\mathcal{V}') = \mathcal{U} \cup \mathsf{Var} \cup \{\mu x.\{t \mid u\} \mid \forall v \in \mathcal{V}', t\{x\backslash v\} \perp u\{x\backslash v\}\} \\
\mathcal{V}' &= \Phi_{\mathcal{V}}(\mathcal{U}') = \mathcal{V} \cup \mathsf{Var} \cup \{\mu x.\{t \mid u\} \mid \forall v \in \mathcal{U}', t\{x\backslash v\} \perp u\{x\backslash v\}\}
\end{aligned}$$

It is clearly saturated. We now prove that $\mathcal{U}' \perp \mathcal{V}'$.

Since $\mathcal{U} \perp \mathcal{V}$ and $\mathcal{U}$ and $\mathcal{V}$ are non-empty, we have $\mathcal{U} \subseteq \mathsf{SN}$ and $\mathcal{V} \subseteq \mathsf{SN}$. We also have $\mathsf{Var} \subseteq \mathsf{SN}$. Finally, by Remark 10, we conclude $\mathcal{U}' \subseteq \mathsf{SN}$ and $\mathcal{V}' \subseteq \mathsf{SN}$.

Now assume $u \in \mathcal{U}'$ and $v \in \mathcal{V}'$. We show $u \perp v$ by lexicographical induction on the length of the longest derivation starting from $u \in \mathsf{SN}$ and that of the longest derivation starting from $v \in \mathsf{SN}$.

If $u \in \mathcal{U}$ and $v \in \mathcal{V}$ then $u \perp v$ because $\mathcal{U} \perp \mathcal{V}$. If not, we prove $u \perp v$ by showing that whenever $\{u \mid v\} \longrightarrow p$, then $p \in \mathsf{SN}$.

- If $\{u \mid v\} \longrightarrow \{u' \mid v\}$ or $\{u \mid v\} \longrightarrow \{u \mid v'\}$, the induction hypothesis applies.
- The only other case is $u = \mu x.p$ (resp. $v = \mu x.p$) and $\{u \mid v\} \longrightarrow p\{x\backslash v\}$ (resp. $\{u \mid v\} \longrightarrow p\{x\backslash u\}$). But since $u \in \mathcal{U}'$ and $v \in \mathcal{V}'$, we know that $p\{x\backslash v\} \in \mathsf{SN}$ (resp. $p\{x\backslash u\} \in \mathsf{SN}$).

□

Definition 7 — Now we interpret kinds:

$$\begin{aligned}
[\![\star]\!] &= \{(\mathcal{U}, \mathcal{V}) \mid \mathcal{U} \perp \mathcal{V} \text{ and } (\mathcal{U}, \mathcal{V}) \text{ is saturated}\} \\
[\![K \to K']\!] &= [\![K]\!] \to [\![K']\!]
\end{aligned}$$

Given, a pair $p \in [\![\star]\!]$, we write p^+ (resp. p^-) its first (resp. second) component. We also define the function $\mathsf{swap}_K : [\![K]\!] \to [\![K]\!]$ by induction on K:

$$\begin{aligned}
\mathsf{swap}_\star(\mathcal{U}, \mathcal{V}) &= (\mathcal{V}, \mathcal{U}) \\
\mathsf{swap}_{K \to K'} f &= \mathsf{swap}_{K'} \circ f
\end{aligned}$$

Let $\mathsf{swap} : (\bigcup_K [\![K]\!]) \to (\bigcup_K [\![K]\!])$ be the disjoint union of all the swap_K.

Definition 8 — Let $\mathcal{U}$ and $\mathcal{V}$ be sets of Curry-style terms. We set the following definitions:

$$\begin{aligned}
\langle \mathcal{U}, \mathcal{V} \rangle &= \{\langle u, v \rangle \mid u \in \mathcal{U}, v \in \mathcal{V}\} \\
\lambda \mathcal{U}\mathcal{V}.\diamond &= \{\lambda xy.p \mid \forall u \in \mathcal{U}\ \forall v \in \mathcal{V}\ p\{x, y\backslash u, v\} \in \mathsf{SN}\} \\
\lambda\_.\mathcal{U} &= \{\lambda\_.u \mid u \in \mathcal{U}\} \\
\langle \_, \mathcal{U} \rangle &= \{\langle \_, u \rangle \mid u \in \mathcal{U}\}
\end{aligned}$$

Note that those sets are always simple.

Definition 9 — We say that a mapping $\rho : \mathsf{Var}^T \to \bigcup_K [\![K]\!]$ is *compatible* with Σ if $\forall (\alpha : K) \in \Sigma, \rho(\alpha) \in [\![K]\!]$.

Definition 10 — For each A such that $\Sigma \vdash A : K$ for some K, and for each ρ compatible with Σ, we define $[\![A]\!]_\rho \in [\![K]\!]$ as follows:

$$
\begin{array}{lcl}
[\![\alpha]\!]_\rho & = & \rho(\alpha) \\
[\![\alpha^\perp]\!]_\rho & = & \mathsf{swap}(\rho(\alpha)) \\
[\![A \wedge B]\!]_\rho & = & \text{any saturated } (\mathcal{U},\mathcal{V}) \text{ such that} \\
 & & \langle [\![A]\!]^+_\rho, [\![B]\!]^+_\rho \rangle \subseteq \mathcal{U} \\
 & & \lambda [\![A]\!]^+_\rho [\![B]\!]^+_\rho . \diamond \subseteq \mathcal{V} \\
 & & \mathcal{U} \perp \mathcal{V} \\
[\![A \vee B]\!]_\rho & = & \text{any saturated } (\mathcal{U},\mathcal{V}) \text{ such that} \\
 & & \lambda [\![A]\!]^-_\rho [\![B]\!]^-_\rho . \diamond \subseteq \mathcal{U} \\
 & & \langle [\![A]\!]^-_\rho, [\![B]\!]^-_\rho \rangle \subseteq \mathcal{V} \\
 & & \mathcal{U} \perp \mathcal{V} \\
[\![\forall \alpha : K' . A]\!]_\rho & = & \text{any saturated } (\mathcal{U},\mathcal{V}) \text{ such that} \\
 & & \lambda \_ . \bigcap_{h \in [\![K']\!]} [\![A]\!]^+_{\rho,\alpha\mapsto h} \subseteq \mathcal{U} \\
 & & \langle \_, \bigcup_{h \in [\![K']\!]} [\![A]\!]^-_{\rho,\alpha\mapsto h} \rangle \subseteq \mathcal{V} \\
 & & \mathcal{U} \perp \mathcal{V} \\
[\![\exists \alpha : K' . A]\!]_\rho & = & \text{any saturated } (\mathcal{U},\mathcal{V}) \text{ such that} \\
 & & \langle \_, \bigcup_{h \in [\![K']\!]} [\![A]\!]^+_{\rho,\alpha\mapsto h} \rangle \subseteq \mathcal{U} \\
 & & \lambda \_ . \bigcap_{h \in [\![K']\!]} [\![A]\!]^-_{\rho,\alpha\mapsto h} \subseteq \mathcal{V} \\
 & & \mathcal{U} \perp \mathcal{V} \\
[\![\lambda \alpha : K' . A]\!]_\rho & = & h \in [\![K']\!] \mapsto [\![A]\!]_{\rho,\alpha\mapsto h} \\
[\![A\ B]\!]_\rho & = & ([\![A]\!]_\rho)([\![B]\!]_\rho)
\end{array}
$$

The soundness of the definition inductively relies on the fact that $[\![A]\!]_\rho \in [\![K]\!]$ and ρ keeps being compatible with Σ. The existence of the saturated extensions in the case of $A \wedge B$, $A \vee B$, $\forall \alpha : K' . A$ and $\exists \alpha : K' . A$ is given by Theorem 12.

Remark 13

- Notice that $[\![A^\perp]\!]_\rho = \mathsf{swap}[\![A]\!]_\rho$.
- $[\![A]\!]_{\rho,\alpha\mapsto[\![B]\!]_\rho} = [\![A\{\alpha \backslash B\}]\!]_\rho$
- If $A \longrightarrow_\beta B$ then $[\![A]\!]_\rho = [\![B]\!]_\rho$.
- If $\Sigma \vdash A : \star$, then $[\![A]\!]_\rho$ is saturated, with $[\![A]\!]^+_\rho \subseteq \mathsf{SN}$ and $[\![A]\!]^-_\rho \subseteq \mathsf{SN}$.

Theorem 14 — *If $x_1 : A_1, \ldots, x_n : A_n \vdash_\Sigma t : A$ then for all ρ compatible with Σ, and for all $t_1 \in [\![A_1]\!]_\rho$, $\ldots$, $t_n \in [\![A_n]\!]_\rho$ we have:*

$$\|t\|\{x_1, \ldots, x_n \backslash t_1, \ldots, t_n\} \in [\![A]\!]^+_\rho$$

Proof: By induction on the typing tree. □

Corollary 15 — *If $x_1 : A_1, \ldots, x_n : A_n \vdash_\Sigma t : A$ then $t \in SN$.*

Proof: We first prove that we can find a ρ compatible with Σ (for $\alpha : \star$, take $\rho(\alpha)$ to be any saturated extension of $(\mathsf{Var}, \mathsf{Var})$). Then we can apply Theorem 14 and conclude by Lemma 9. □

5 Logical Properties

5.1 Consistency

The consistency of $F_\omega^{\mathcal{C}}$ follows from corollary 15 using a very simple combinatorial argument. Let us first notice that all untyped programs that are in normal form are of one of the following thirteen forms:

Variable-Variable	$\{x \mid y\}$
Variable-Pair	$\{x \mid \lambda x^A y^B.p\}$
Variable-Lambda	$\{x \mid \langle t, u\rangle\}$
Variable-∀Lambda	$\{x \mid \lambda\alpha : K \,.\, t\}$
Variable-∃Witness	$\{x \mid \langle A, t\rangle\}$
Pair-Pair	$\{\langle t_1, u_1\rangle \mid \langle t_2, u_2\rangle\}$
Lambda-Lambda	$\{\lambda {x_1}^{A_1} {y_1}^{B_1}.p_1 \mid \lambda {x_2}^{A_2} {y_2}^{B_2}.p_2\}$
∀Lambda-∀Lambda	$\{\lambda\alpha_1 : K \,.\, t_1 \mid \lambda\alpha_2 : K \,.\, t_2\}$
∃Witness-∃Witness	$\{\langle A_1, t_1\rangle \mid \langle A_2, t_2\rangle\}$
Lambda-∀Lambda	$\{\lambda {x_1}^{A_1} {y_1}^{B_1}.p_1 \mid \lambda\alpha : K \,.\, t_2\}$
Pair-∀Lambda	$\{\langle t_1, u_1\rangle \mid \lambda\alpha : K \,.\, t_2\}$
Lambda-∃Witness	$\{\lambda {x_1}^{A_1} {y_1}^{B_1}.p_1 \mid \langle A_2, t_2\rangle\}$
Pair-∃Witness	$\{\langle t_1, u_1\rangle \mid \langle A_2, t_2\rangle\}$

However, if we restrict to well-typed programs, the last eight forms are ruled out for obvious typing reasons[1], hence:

Fact 16 — *There is no closed well-typed program in normal form.*

Combining this with corollary 15, we get

Proposition 17 — *There is no closed well-typed program.*

From which we deduce that the formalism is logically consistent.

5.2 Translating $F_\omega + \mathrm{DNE}$ into $F_\omega^{\mathcal{C}}$

The definition of implication $A \Rightarrow B$ as $(A^\perp) \vee B$ naturally suggests a translation from system F_ω to system $F_\omega^{\mathcal{C}}$. We annotate sequents in F_ω using $\vdash^{F_\omega}$.

The translation proceeds as follows: each kind of F_ω is translated as itself, and each type constructor A of F_ω is translated as a type constructor A^* of $F_\omega^{\mathcal{C}}$ by the equations

$$\begin{array}{rcl} \alpha^* & = & \alpha \\ (\forall\alpha : K \,.\, A)^* & = & \forall\alpha : K \,.\, A^* \\ (A \Rightarrow B)^* & = & A^{*\perp} \vee B^* \\ (\lambda\alpha : K \,.\, B)^* & = & \lambda\alpha : K \,.\, B^* \\ (B\ A)^* & = & B^*\ A^* \end{array}$$

We then easily check that

Proposition 18 — *If $\Sigma \vdash^{F_\omega} A : K$, then $\Sigma \vdash A^* : K$.*

Proposition 19 — *If $A \longrightarrow_{F_\omega} B$, then $A^* \longrightarrow_{F_\omega^{\mathcal{C}}} B^*$.*

[1]In each of these eight forms, both members introduce a main connective or quantifier which is not the dual of the one introduced on the other side, which contradicts the typing rule of programs.

We now translate proof-terms, adapting Prawitz's translation of natural deduction into sequent calculus:

$$\begin{array}{rcl}
x^* & = & x \\
(\lambda x\colon A.t)^* & = & \lambda x^A y^B.t^*_y \\
(\lambda\alpha:K\,.\,t)^* & = & \lambda\alpha:K\,.\,\mu y^B.t^*_y \\
v^*_t & = & \{v^* \mid t\} \\
(t_1\ t_2)^*_t & = & {t_1}^*_{\langle \mu y^B.{t_2}^*_y,t\rangle} \\
(t_1\ A)^*_t & = & {t_1}^*_{\langle A^*,t\rangle}
\end{array}$$

where B is the type of t in the case of abstractions, and the type of t_2 in applications.

The translation preserves typing (using Proposition 19):

Proposition 20 — *If* $\Gamma \vdash^{F_\omega}_{\Sigma} v : A$, *then* $\Gamma^* \vdash_\Sigma v^* : A$.
If $\Gamma \vdash^{F_\omega}_{\Sigma} u : A$ *and* $\Gamma^*, y\colon B^\perp \vdash_\Sigma t : A^{*\perp}$, *then* $\Gamma^*, y\colon B^\perp \vdash_\Sigma u^*_t \diamond$.

We can then simulate β-reduction of F_ω:

Proposition 21 — *If* $v \longrightarrow_{F_\omega} v'$, *then* $v^* \longrightarrow^+_{F^{\mathcal{C}}_\omega} v'^*$.
If $u \longrightarrow_{F_\omega} u'$, *then* $u^*_t \longrightarrow^+_{F^{\mathcal{C}}_\omega} u'^*_t$.

Since $F^{\mathcal{C}}_\omega$ is classical, we have a proof of the axiom of double negation elimination $\mathrm{DNE} = \forall\alpha:\star.\,((\alpha \Rightarrow \bot) \Rightarrow \bot) \Rightarrow \alpha$ (where $\bot = \forall\alpha:\star.\,\alpha$ and $\top = \exists\alpha:\star.\,\alpha$). Indeed, $\mathrm{DNE}^* = \forall\alpha:\star.\,((\alpha^\perp \vee \bot) \wedge \top) \vee \alpha$ and

$$\vdash \lambda\alpha:\star.\,\lambda x^B y^{\alpha^\perp}.\{x \mid \langle \lambda x'^{\alpha} y'^{\top}.\{x' \mid y\}, \langle \alpha^\perp, y\rangle\rangle\} : \mathrm{DNE}^*$$

where $B = (\alpha \wedge \top) \vee \bot$. We call this term **c**. Hence, provable propositions of system $F_\omega + \mathrm{DNE}$ become provable propositions of system $F^{\mathcal{C}}_\omega$:

Proposition 22 — *For all derivable judgments of the form*

$$z : \forall\alpha\ (\neg\neg\alpha \Rightarrow \alpha),\ \Gamma\ \vdash^{F_\omega}_{\Sigma}\ t\ :\ A$$

we have

$$\Gamma^* \vdash_\Sigma \mu y^{A^*}.t^*_y\{z\backslash \boldsymbol{c}\} : A^*$$

Through the translation $A \mapsto A^*$, system $F^{\mathcal{C}}_\omega$ appears as an extension of system $F_\omega + \mathrm{DNE}$. We conjecture that this extension is conservative.

6 Conclusion

It is important to understand why, in section 4 we did not simply use sets of terms equal to their bi-orthogonal closure as reducibility candidates for proof-terms, as we did in section 3 for type constructors.

The reason is that in general, the pair $(\mathcal{U}^{\perp\perp}, \mathcal{U}^{\perp})$ formed by a set of terms $\mathcal{U}^{\perp\perp}$ closed under bi-orthogonal (a 'type') and its orthogonal $\mathcal{U}^{\perp}$ does not seem to be saturated in the sense of Def. 5. Technically, there is no equivalent for proof terms of Prop. 4 (item 4), due to the presence of a μ-μ critical pair which makes the proof difficult or impossible to adapt in the nondeterministic case. This lack of saturation is the motivation of the fixpoint construction above to interpret types.

However, the question whether pairs of the form $(\mathcal{U}^{\perp\perp}, \mathcal{U}^{\perp})$ are saturated or not is still open, as far as we known. If the answer is positive, then the fixpoint construction is unnecessary. Otherwise, it would be interesting to investigate whether the choice of a more sophisticated relation of orthogonality could solve the problem and replace the fixpoint construction.

References

[Bar84] H. P. Barendregt. *The Lambda-Calculus, its syntax and semantics.* Studies in Logic and the Foundation of Mathematics. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1984. Second edition.

[Bar91] H. P. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[BB96] F. Barbanera and S. Berardi. A symmetric lambda-calculus for classical program extraction. *Information and Computation*, 125(2):103–117, 1996.

[BG01] H. Barendregt and H. Geuvers. Proof-assistants using dependent type systems. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1149–1238. Elsevier and MIT Press, 2001.

[CH00] P.-L. Curien and H. Herbelin. The duality of computation. In *Proc. of the 5^{th} ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'00)*, pages 233–243. ACM Press, 2000.

[DGLL05] D. J. Dougherty, S. Ghilezan, P. Lescanne, and S. Likavec. Strong normalization of the dual classical sequent calculus. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 12th Int. Conf. (LPAR'05)*, volume 3835 of *LNCS*, pages 169–183. Springer, dec 2005.

[Gir72] J.-Y. Girard. Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur. These D'État, Université Paris VII, 1972.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[Par92] M. Parigot. $\lambda\mu$-calculus: An algorithmique interpretation of classical natural deduction. In *Proc. of the Int. Conf. on Logic Programming and Automated Reasoning (LPAR).* LNCS1, 1992.

[Pol04] E. Polonovski. Strong normalization of lambda-mu-mu/tilde-calculus with explicit substitutions. In I. Walukiewicz, editor, *Proc. of the 7th Int. Conf. on Foundations of Software Science and Computation Structures (FOSSACS'04)*, volume 2987 of *LNCS*, pages 423–437. Springer, March 2004.

[Sel99] P. Selinger. Control categories and duality: on the categorical semantics of the $\lambda\mu$-calculus. *Mathematical Structures in Computer Science*, 1999.

[Urb00] C. Urban. *Classical Logic and Computation.* PhD thesis, University of Cambridge, 2000.