THE SIMPLY-TYPED PURE PATTERN TYPE SYSTEM ENSURES STRONG NORMALIZATION

Benjamin Wack
LORIA & Université Henri Poincaré, Nancy, France
Benjamin.Wack@loria.fr

Abstract Pure Pattern Type Systems (P^2TS) combine in a unified setting the capabilities of rewriting and **λ-calculus.** Their type systems, adapted from Barendregt's **λ-cube,** are especially interesting from a logical point of view. Strong normalization, an essential property for logical soundness, had only been conjectured so far: in this paper, we give a positive answer for the simply-typed system.

The proof is based on a translation of terms and types from P^2TS into the **λ-calculus.** First, we deal with untyped terms, ensuring that reductions are faithfully mimicked in the **λ-calculus.** For this, we rely on an original encoding of the pattern matching capability of P^2TS into the **λ-calculus.**

Then we show how to translate types: the expressive power of System $F\omega$ is needed in order to fully reproduce the original typing judgments of P^2TS. We prove that the encoding is correct with respect to reductions and typing, and we conclude with the strong normalization of simply-typed P^2TS terms.

1 Introduction

The **λ-calculus** and term rewriting provide two fundamental computational paradigms that had a deep influence on the development of programming and specification languages, and on proof environments. The idea that having computational power at hand makes deduction significantly easier and safer is widely acknowledged (Dowek et al., 2003; Werner, 1994). Many frameworks have been designed with a view to integrate these two formalisms: either by enriching first-order rewriting with higher-order capabilities (Klop et al., 1993) or by adding algebraic features to the **λ-calculus** (*case* expressions with dependent types (Coquand, 1992), a typed pattern calculus (Kesner et al., 1996) and calculi of algebraic constructions (Blanqui, 2001)).

The *rewriting calculus,* or **ρ-calculus,** by unifying the **λ-calculus** and the rewriting, makes all the basic ingredients of rewriting explicit objects, in particular the notions of *rule application* and *result.* A rewrite rule becomes a first-class object which can be created and manipulated in the calculus, whereas in works like (Blanqui, 2001), the rewriting remains a bit external to the calculus.

In (Cirstea et al., 2001), a collection of type systems for the **ρ-calculus** was presented, extending Barendregt's **λ-cube** to a **ρ-cube.** Later, these type systems have been studied deeper for the similar formalism of P^2TS (Barthe et al., 2003). Yet, the rewriting calculus has also been assigned some type systems that do *not* prevent infinite reductions (Cirstea et al., 2004). Thus, strong normalization did remain an open problem for P^2TS. In this paper, we give a first positive answer to this problem. Since consistency is related to termination, this result makes P^2TS a good candidate for a proof-term language integrating deduction and computation at the same level.

The main contributions of this paper are:

- a more recent version of P^2TS, enhanced with a signature for the types of constants and some corrections on the product rules;
- a concise encoding of pattern matching in the λ-calculus, which has other potential applications for the encoding of term rewriting systems;
- a translation of the simply-typed system of P^2TS into System $\mathsf{F}\omega$ emphasizing some particular typing mechanisms of P^2TS;
- a proof of strong normalization for simply-typed P^2TS terms.

This paper is organized as follows. In Section 2, we recall the syntax and the small-step semantics of P^2TS. In Section 3, we give an untyped version of the translation, showing how pattern matching is encoded. In Sections 4 and 5, we present the type systems of P^2TS and System $\mathsf{F}\omega$. In Sections 6 and 7, we give the fully typed translation and we outline a proof of correctness for three important elements of the typed translation: variables, constants and delayed matching constraints. In Section 8, we state the key lemmas used in the full strong normalization proof.

We assume the reader is reasonably familiar with the notations and results of typed λ-calculi (Barendregt, 1992), of the ρ-calculus (Cirstea et al., 2004) and of P^2TS (Barthe et al., 2003).

Conventions and notations Generally, the reader can assume that every capital letter denotes an object belonging to P^2TS, and every small letter denotes an object belonging to the λ-calculus (except for constants and their arity). For instance, in P^2TS: *X, Y, Z* are variables; *A, B, C* are terms; *P, Q* are patterns; a, f, g are constants; Φ, Ψ are types; Ξ is an atomic type. In System $\mathsf{F}\omega$: w, x, y, z are variables; t, u are terms; β, γ are type variables; σ, τ are types; k is a kind. Moreover, we will use the notations: α, α_i for an arity; θ for a substitution; Γ, Δ for contexts (mainly in P^2TS); Σ for a signature.

Syntactic equivalence of terms will be denoted by $\equiv$. If a substitution θ has domain $X_1 \dots X_n$ and $\forall i,\ X_i\theta \equiv A_i$, we will also write it $[X_1 := A_1 \dots X_n := A_n]$. We assume that the signature Σ of constants that can be used in P^2TS is finite, which is legitimate since a given (finite) term only uses a finite number of constants. Therefore, we will number the constants $f_1, \dots, f_S$, where S is the cardinal of Σ. To denote a tuple of terms $B_k \dots B_n$, we will use the vector notation $\overrightarrow{B}_{(k..n)}$, or simply $\overrightarrow{B}$ when k and n are obvious from the context. This notation will be used in combination with operators according to their default associativity: for instance, in System $\mathsf{F}\omega$, $A\overrightarrow{B} \triangleq AB_1 \dots B_n$ and $\lambda\overrightarrow{x}.A \triangleq \lambda x_1 \dots \lambda x_n.A$. To avoid confusion between symbols, we will use bold $\boldsymbol{\lambda}$ and $\boldsymbol{\Pi}$ for P^2TS and roman λ and Π for System $\mathsf{F}\omega$.

2 P^2TS: dynamic semantics

In this section, we recall the syntax of P^2TS and their evaluation rules. The syntax of P^2TS extends that of the typed λ-calculus with structures and patterns (Barthe et al., 2003). Several choices can be made for the set of patterns P: in this paper, *we only consider algebraic patterns,* whose shape is defined below. The main reason for this restriction is that patterns containing symbols such as λ require higher-order matching, which seems difficult to encode in a typed λ-calculus.

$$\begin{array}{lll}
\textit{Signature} & \Sigma ::= \emptyset \mid \Sigma, f : A & \qquad \textit{Context} \quad \Gamma ::= \emptyset \mid \Gamma, X : A \\
\textit{Pattern} & P ::= X \mid f \bullet \overrightarrow{P} & \\
\textit{Term} & A ::= f \mid X \mid \lambda(P : \Delta).A \mid \Pi(P : \Delta).A \mid [P \ll_\Delta A]A \mid A \bullet A \mid A ; A &
\end{array}$$

A term with shape $\lambda(P : \Delta).A$ is an *abstraction* with pattern P, body A and context Δ. The term $[P \ll_\Delta B]A$ is a *delayed matching constraint* with pattern P, body A, argument B and context Δ. A term $\Pi(P : \Delta).A$ is a *dependent product,* and will be used as a type; finally, $(A; B)$ is a *structure* and $A \bullet B$ is an *application.* The application of a constant symbol, say f, to a term A will be denoted by $f \bullet A$ too; it follows that the usual algebraic notation of a term is currified, *e.g.* $f(A_1, \ldots, A_n) \triangleq f \bullet A_1 \bullet \cdots \bullet A_n \triangleq f \bullet \overrightarrow{A}$.

DEFINITION 1 (FREE VARIABLES $\mathcal{FV}$ OF A TERM)

$$\begin{array}{lcllcl}
\mathcal{FV}(A;B) = \mathcal{FV}(A \bullet B) & \triangleq & \mathcal{FV}(A) \cup \mathcal{FV}(B) & \mathcal{FV}(X) & \triangleq & \{X\} \\
\mathcal{FV}(\lambda(P:\Delta).A) & \triangleq & \mathcal{FV}(\Pi(P:\Delta).A) & \triangleq & (\mathcal{FV}(A) \cup \mathcal{FV}(\Delta)) \setminus \mathcal{D}om(\Delta) & \\
\mathcal{FV}([P \ll_\Delta B]A) & \triangleq & (\mathcal{FV}(A) \cup \mathcal{FV}(\Delta)) \setminus \mathcal{D}om(\Delta) & & & \\
\mathcal{FV}(\Gamma, X : A) & \triangleq & \mathcal{FV}(\Gamma) \cup \mathcal{FV}(A) & \mathcal{FV}(f) & \triangleq & \emptyset
\end{array}$$

In this paper, extending Church's notation, the context Δ in $\lambda(P : \Delta).B$ (resp. $[P \ll_\Delta B]A$ or $\Pi(P : \Delta).B$) contains the type declarations of the free variables appearing in the pattern P, *i.e.* $\mathcal{D}om(\Delta) = \mathcal{FV}(P)$. These variables are bound in the abstraction. The context Δ will be omitted when we consider untyped terms. As usual, we work modulo α-*conversion* and we use Barendregt's *"hygiene-convention"* (Barendregt, 1992), *i.e.* free and bound variables have different names.

For the purpose of this paper, we consider only syntactic pattern matching; a syntactic matching equation $P \ll A$ has either no solution or a unique solution noted $\theta_{(P \ll A)}$. In fact, it seems difficult to encode more elaborated matching theories: for instance, associative matching can generate an arbitrary high number of distinct solutions. Thus, to give a faithful account of all matching solutions in the λ-calculus, one would probably need a fixed point.

$$\begin{array}{lrcl}
(\rho) & (\lambda(P:\Delta).A) \bullet B & \to_\rho & [P \ll_\Delta B]A \\
(\sigma) & [P \ll_\Delta B]A & \to_\sigma & A\theta_{(P \ll B)} \\
(\delta) & (A;B) \bullet C & \to_\delta & A \bullet C ; B \bullet C
\end{array}$$

Figure 1. **Top-level rules of P^2TS**

The top-level rules are presented in Fig. 1. By the (ρ) rule, the application of a term $\lambda(P : \Delta).A$ to a term B reduces to the delayed matching constraint $[P \ll_\Delta B]A$; the application of the (σ) rule consists in solving the matching equation $P \ll B$ and applying the obtained substitution (if it exists) to the the term A. If no solution exists, the (σ) rule is not fired and the term $[P \ll_{\Delta} B]A$ is not reduced. As usual, $\mapsto_{\rho\sigma\delta}$ denotes the congruent closure of $\rightarrow_{\rho} \cup \rightarrow_{\sigma} \cup \rightarrow_{\delta}$, and $\mapsto\!\!\!\rightarrow_{\rho\sigma\delta}$ (resp. $=_{\rho\sigma\delta}$) is defined as the reflexive and transitive (resp. reflexive, symmetric and transitive) closure of $\mapsto_{\rho\sigma\delta}$.

3 Untyped encoding

In this section we translate the untyped P^2TS with algebraic patterns. The process of syntactic pattern matching consists in discriminating whether the argument begins with the expected constant, and recursively use pattern matching on subterms. It is this (quite simple) algorithm that we encode in the **λ-calculus.** We use the following notations: S is the number of symbols appearing in the signature. The i**th** symbol of Σ is denoted by f_i.

To build the encoding of pattern matching, we need three conditions:

1 each constant f_i has a "maximal" arity α_i, in the sense that f_i is never applied to more than α_i arguments;
2 in every matching equation $f_i \bullet \overrightarrow{P}_{(1..p)} \ll f_j \bullet \overrightarrow{B}_{(1..q)}$, we have $\alpha_i - p = \alpha_j - q$;
3 each term $(A;\, B)$ has a maximal arity α.

In particular, when $i = j$, the second condition reduces to $p = q$, which is an essential condition for resolving this matching equation.

In this section, we assume these properties. In Section 4, we will see that typing enforces the three conditions. They remain true in some untyped situations too: for instance, if we were to encode a Term Rewriting System, the arity of the constants would be given, and partial application of a constant would be forbidden, ensuring that in every matching equation $\alpha_i - p = \alpha_j - q = 0$.

The translation is given in Fig. 2, by a recursive function $\llbracket \cdot \rrbracket$ mapping P^2TS terms to **λ-terms.** We use a fresh variable $x_{\perp}$; if a closed term is needed, we add an abstraction "$\lambda x_{\perp}$" once the whole P^2TS term is translated.

$$
\begin{aligned}
\llbracket X \rrbracket &\triangleq X \\
\llbracket f_i \rrbracket &\triangleq \lambda \overrightarrow{x}_{(1..\alpha_i)}.\, (\lambda \overrightarrow{z}_{(1..S)}.(z_i\ \overrightarrow{x}_{(1..\alpha_i)})) \\
\llbracket A; B \rrbracket &\triangleq \lambda \overrightarrow{x}_{(1..\alpha)}.\, \Big(\lambda z.\big(z(\llbracket A \rrbracket \overrightarrow{x}_{(1..\alpha)})(\llbracket B \rrbracket \overrightarrow{x}_{(1..\alpha)})\big)\Big) \\
\llbracket \lambda X.A \rrbracket &\triangleq \lambda X.\llbracket A \rrbracket \\
\llbracket \lambda (f_i \bullet \overrightarrow{P}_{(1..p)}).A \rrbracket &\triangleq \lambda y.(y \overrightarrow{x\perp}_{(p+1..\alpha_i)}\ \overrightarrow{x\perp}_{(1..i-1)} \llbracket \lambda \overrightarrow{P}_{(1..p)}.\lambda \overrightarrow{x}'_{(p+1..\alpha_i)}.A \rrbracket\ \overrightarrow{x\perp}_{(i+1..S)}) \\
\llbracket A \bullet B \rrbracket &\triangleq \llbracket A \rrbracket \llbracket B \rrbracket \\
\llbracket [P \ll B]A \rrbracket &\triangleq \text{the term obtained by head-}\beta\text{-reducing } \llbracket (\lambda P.A) \bullet B \rrbracket
\end{aligned}
$$

Figure 2. Untyped term translation

Let us briefly explain this translation:

- In $\llbracket f_i \rrbracket$, the variables $x_1 \ldots x_{\alpha_i}$ will be instantiated by the arguments $\overrightarrow{B}$ of f_i (which explains why we had to bound the arity of f_i). Then, among the variables $z_1 \ldots z_S$, the one corresponding to the head constant of P is selected.
- $\llbracket A; B \rrbracket$ is translated into the usual pair encoding of the **λ-calculus,** and the abstractions $\lambda \overrightarrow{x}$ distribute the arguments to both elements of the pair.

- In $[\![\lambda X.A]\!]$, the abstraction over a single variable is straightforwardly translated into a λ-**abstraction**.
- In $[\![\lambda(f_i \bullet \overrightarrow{P}_{(1..p)}).A]\!]$, the variable y will be instantiated by the argument of this function (for instance $[\![f_j \bullet \overrightarrow{B}]\!]$). If necessary, the $\alpha_i - p$ first occurrences of the variable $x_\perp$ instantiate the remaining variables $x_{q+1} \dots x_{\alpha_j}$ which can appear in $[\![f_j]\!]$: this is where we use the condition $\alpha_i - p = \alpha_j - q$. Then, if $f_i = f_j$, the $z_1 \dots z_S$ select $[\![\lambda \overrightarrow{P}_{(1..p)}.\lambda \overrightarrow{x'}_{(p+1..\alpha_i)}.A]\!]$ and the encoding of pattern matching can then go on (pointwise) with the sub-patterns $P_1 \dots P_p$ and the subterms $B_1 \dots B_p$; if matching fails, $x_\perp$ is selected, witnessing the failure. The fresh variables $x'_{p+1} \dots x'_{\alpha_i}$ will be instantiated by $x_\perp$'s, but they do not appear in $[\![A]\!]$. If a variable X has multiple occurrences in the pattern, by α-**conversion**, only one of the subpatterns P_i will get the "original" variable, and the other X's are renamed to fresh variables not occurring in $[\![A]\!]$ (so matching failures due to non-linearity are not detected by the encoding).
- $[\![A \bullet B]\!]$ is translated into standard λ-**calculus** application.
- $[\![[P \ll B]A]\!]$ is $[\![(\lambda P.A) \bullet B]\!]$ where y has been instantiated by $[\![B]\!]$.

LEMMA 1 (CLOSURE BY SUBSTITUTION)
For any P^2TS terms A and $B_1, \dots, B_n$, for any variables $X_1, \dots X_n$,

$$[\![A[X_1 := B_1 \dots X_n := B_n]]\!] = [\![A]\!][X_1 := [\![B_1]\!] \dots X_n := [\![B_n]\!]]$$

THEOREM 1 (FAITHFUL REDUCTIONS)
For any terms A and B, if $A \mapsto_{\rho\delta} B$, then $[\![A]\!] \mapsto\!\!\!\!\to_\beta [\![B]\!]$ in at least one step.

EXAMPLE 1 (TRANSLATION OF A SUCCESSFUL DELAYED MATCHING)

$$\begin{array}{rcl} (\lambda Y.\underline{(\lambda(f \bullet X).X) \bullet Y}) \bullet (f \bullet a) & \mapsto_\rho & \underline{(\lambda Y.[f \bullet X \ll Y]X) \bullet (f \bullet a)} \\ & \mapsto_\rho & \underline{[Y \ll f \bullet a]}[f \bullet X \ll Y]X \\ & \mapsto_\delta & \underline{[f \bullet X \ll f \bullet a]}X \\ & \mapsto_\delta & a \end{array}$$

The inner delayed matching constraint is essential here because it has to "wait "for the instantiation of Y before performing matching. For the translation, we consider $\Sigma = \{a_1, f_2\}$ with $\alpha_1 = 0$ and $\alpha_2 = 1$. The reductions are shown on Fig. 3. The selected λ-***abstraction*** *and its argument are underlined.*

4 The typed P^2TS: static semantics

This section presents a version of the type systems of P^2TS with some minor adaptations. The inference rules are given in Fig. 4. For a detailed explanation of these rules, the reader can refer to (Barthe et al., 2003); here, we will only discuss some differences with regard to previous type systems for the ρ-**calculus** and P^2TS:

- In (Cirstea and Kirchner, 2000), a first strongly normalizing type system for the ρ-**calculus** was introduced; however, the proof of normalization is mainly based on a heavy restriction over the types of constants.

$$\begin{array}{ll}
 & \overbrace{(\lambda Y.\Big(\overbrace{(\underline{\lambda y}.(y x_\perp(\lambda X.X)))}^{[\![\lambda(f\bullet X).X]\!]}\underline{Y}\Big))}^{[\![\lambda Y.(\lambda(f\bullet X).X)\bullet Y]\!]}\Big(\overbrace{(\lambda x_1.\lambda z_1\lambda z_2.(z_2 x_1))}^{[\![f]\!]}\overbrace{(\lambda u_1\lambda u_2.u_1)}^{[\![a]\!]}\Big)\\
\mapsto_\beta & (\lambda Y.(Y x_\perp(\lambda X.X)))\Big((\underline{\lambda x_1}.\lambda z_1\lambda z_2.(z_2 x_1))\underline{(\lambda u_1\lambda u_2.u_1)}\Big)\\
\mapsto_\beta & (\underline{\lambda Y}.(Y x_\perp(\lambda X.X)))\underline{(\lambda z_1\lambda z_2.(z_2(\lambda u_1\lambda u_2.u_1)))}\\
\mapsto_\beta & (\underline{\lambda z_1}\lambda z_2.(z_2(\lambda u_1\lambda u_2.u_1)))\underline{x_\perp}(\lambda X.X)\\
\mapsto_\beta & (\underline{\lambda z_2}.(z_2(\lambda u_1\lambda u_2.u_1)))\underline{(\lambda X.X)}\\
\mapsto_\beta & (\underline{\lambda X}.X)\underline{(\lambda u_1\lambda u_2.u_1)}\\
\mapsto_\beta & (\lambda u_1\lambda u_2.u_1)\\
= & [\![a]\!]
\end{array}$$

Figure 3. **Translation of a successful delayed matching**

- In (Cirstea et al., 2004), we studied a more permissive type system, still enforcing subject reduction, but allowing to typecheck some terms with infinite reductions. Therefore, this type system was not fit for using the ρ-**calculus** as a proof-term language.
- The type systems of (Cirstea et al., 2001; Barthe et al., 2003) were designed in order to provide a strongly normalizing calculus where there was no restriction on the type of the constants (apart those imposed by the type system). Until now, strong normalization was an open problem for these systems. Here, we show this property for a slight variation of (Barthe et al., 2003). We have introduced a signature Σ which prevents the type of a constant to depend on free variables.

  In rules (MSORT) and (PROD), the first premise avoids a collapse of the P^2TS-cube. If we had just taken $\Psi_0 : s_1$, with $\vdash_\rho f : \Pi(\beta : *).\beta$, the pattern $f\bullet\gamma$ would have sort $*$ but could be used to instantiate the type variable γ, enabling polymorphism in the simply-typed system.

  In the rule (*Var*), we use $\Gamma_\Box \triangleq \{X : \Phi \in \Gamma \mid \Sigma, \Gamma \vdash_\rho \Phi : \Box\}$ to avoid free *term* variables occuring in the type of a variable. It is mainly because we want to keep the system "simply-typed", in the sense that matching constraints occurring in types do not yield types depending on terms. For the type systems allowing terms depending on types, this restriction will have to be relaxed.

  Finally, the rule (STRUCT) can seem quite restrictive, since case-dependent expressions such as $\lambda(0 : nat).0\ ; \lambda(s\bullet X : nat).X$ are forbidden. However, it is non-trivial to weaken this rule. For example, if we had typed $\lambda(0 : nat).0\ ;\ \lambda(s\bullet X : nat).X$ with $\Pi(N : nat).nat$, we could have built a typed term with infinite reductions as in (Cirstea et al., 2004).

The notion of arity we have assumed in the untyped encoding can be properly defined here using types: if f_i has type Φ_i, then α_i is defined as $\alpha(\Phi_i)$:

$$\begin{array}{rcl}
\alpha(\Xi) & \triangleq & 0\\
\alpha(\Pi P.\Psi) & \triangleq & 1 + \alpha(\Psi)\\
\alpha([P \ll B]\Psi) & \triangleq & \alpha(\Psi)
\end{array}$$

$$\frac{}{\emptyset \vdash_\rho * : \Box}\ (\textsc{Axiom}) \qquad \frac{\Sigma,\Gamma \vdash_\rho A : \Phi \qquad \Sigma,\Gamma \vdash_\rho B : \Phi}{\Sigma,\Gamma \vdash_\rho A; B : \Phi}\ (\textsc{Struct})$$

$$\frac{\Sigma,\Gamma_\Box \vdash_\rho \Phi : s \qquad X \notin \mathcal{D}om(\Gamma)}{\Sigma,\Gamma, X{:}\Phi \vdash_\rho X : \Phi}\ (\textsc{Var}) \qquad \frac{\Sigma \vdash_\rho \Phi : s \qquad f \notin \mathcal{D}om(\Sigma)}{\Sigma, f : \Phi \vdash_\rho f : \Phi}\ (\textsc{Const})$$

$$\frac{\Sigma,\Gamma \vdash_\rho A : \Phi \qquad \Sigma,\Gamma \vdash_\rho \Psi : s \qquad X \notin \mathcal{D}om(\Gamma)}{\Sigma,\Gamma, X{:}\Psi \vdash_\rho A : \Phi}\ (\textsc{Weak}\Gamma)$$

$$\frac{\Sigma \vdash_\rho A : \Phi \qquad \Sigma \vdash_\rho \Psi : s \qquad f \notin \mathcal{D}om(\Sigma)}{\Sigma, f{:}\Psi \vdash_\rho A : \Phi}\ (\textsc{Weak}\Sigma)$$

$$\frac{\Sigma,\Gamma \vdash_\rho A : \Psi \qquad \Sigma,\Gamma \vdash_\rho \Phi : s \qquad \Phi =_{\rho\delta} \Psi}{\Sigma,\Gamma \vdash_\rho A : \Phi}\ (\textsc{Conv})$$

$$\frac{\Sigma,\Gamma,\Delta \vdash_\rho A : \Phi \qquad \Sigma,\Gamma \vdash_\rho \mathbf{\Pi}(P:\Delta).\Phi : s}{\Sigma,\Gamma \vdash_\rho \boldsymbol{\lambda}(P:\Delta).A : \mathbf{\Pi}(P:\Delta).\Phi}\ (\textsc{Abs})$$

$$\frac{\Sigma,\Gamma \vdash_\rho A : \mathbf{\Pi}(P:\Delta).\Phi \qquad \Sigma,\Gamma \vdash_\rho [P \ll_\Delta B]\Phi : s}{\Sigma,\Gamma \vdash_\rho A \bullet B : [P \ll_\Delta B]\Phi}\ (\textsc{Appl})$$

$$\frac{\Sigma,\Gamma,\Delta \vdash_\rho A : \Phi \qquad \Sigma,\Gamma \vdash_\rho [P \ll_\Delta B]\Phi : s}{\Sigma,\Gamma \vdash_\rho [P \ll_\Delta B]A : [P \ll_\Delta B]\Phi}\ (\textsc{Match})$$

$$\frac{\forall (X{:}\Psi) \in \Delta,\ \Sigma,\Gamma,\Delta \vdash_\rho \Psi : s_1 \qquad \Sigma,\Gamma,\Delta \vdash_\rho P : \Psi_0 \qquad \Sigma,\Gamma,\Delta \vdash_\rho \Phi : s_2}{\Sigma,\Gamma \vdash_\rho \mathbf{\Pi}(P:\Delta).\Phi : s_2}\ (\textsc{Prod})$$

$$\frac{\begin{array}{c}\forall (X{:}\Psi) \in \Delta,\ \Sigma,\Gamma,\Delta \vdash_\rho \Psi : s_1 \\ \Sigma,\Gamma,\Delta \vdash_\rho P : \Psi_0 \qquad \Sigma,\Gamma \vdash_\rho B : \Psi_0 \qquad \Sigma,\Gamma,\Delta \vdash_\rho \Phi : s_2\end{array}}{\Sigma,\Gamma \vdash_\rho [P \ll_\Delta B]\Phi : s_2}\ (\textsc{MSort})$$

In the simply-typed system, $(s_1, s_2) = (*, *)$.

Figure 4. **The typing rules of P^2TS**

One is easily convinced that a term $f_i \bullet \overrightarrow{A}$ where $\overrightarrow{A}$ contains more than α_i elements can not be correctly typed. Similarly, in a term $(A; B) \bullet \overrightarrow{C}$, A and B have a common type Φ so $\overrightarrow{C}$ can not contain more than $\alpha(\Phi)$ elements.

The second condition on arities is enforced too: in a given matching equation $f_i \bullet \overrightarrow{P}_{(1..p)} \ll f_j \bullet \overrightarrow{B}_{(1..q)}$, typing enforces that $f_i \bullet \overrightarrow{P}_{(1..p)}$ and $f_j \bullet \overrightarrow{B}_{(1..q)}$ have the same type, which immediately imposes $\alpha_i - p = \alpha_j - q$.

Some properties of these calculi, proved in (Barthe et al., 2003), are:

LEMMA 2 (SUBSTITUTION) *If* $\Gamma, X : \Phi, \Delta \vdash_\rho A : \Psi$ *and* $\Gamma \vdash_\rho B : \Phi$, *then* $\Gamma, \Delta[X := B] \vdash_\rho A[X := B] : \Psi[X := B]$.

THEOREM 2 (SUBJECT REDUCTION)
If $\Gamma \vdash_\rho A : \Phi$ *and* $A \mapsto_{\rho\delta} A'$, *then* $\Gamma \vdash_\rho A' : \Phi$.

LEMMA 3 (UNIQUENESS OF TYPES UP TO SECOND ORDER)
If $(s_1, s_2) \in \{(*, *), (\Box, *)\}$, *if* $\Gamma \vdash_\rho A : \Phi_1$ *and* $\Gamma \vdash_\rho A : \Phi_2$, *then* $\Phi_1 =_{\rho\delta} \Phi_2$.

In this paper, we only treat the case of the simply typed calculus, corresponding to $(s_1, s_2) = \{(*, *)\}$. In particular, this implies uniqueness of types.

As a conclusion to this section, let us briefly explain why usual reducibility techniques seem to fail for this typed calculus. Roughly speaking, the interpretation of a type $\Pi(P : \Delta).\Phi$ should be a function space whose domain is defined not only as the interpretation of the type of P but also as terms matching with P and whose suitable subterms belong to the interpretations of the types appearing in Δ. Quickly, this imbrication of interpretations leads to circularities in the definition of interpretations. Thus, it seems really tricky to obtain a proper definition of the reducibility candidates.

5 The System $\mathsf{F}\omega$

In this section, we shortly recall the type system $\mathsf{F}\omega$, first introduced and studied in (Girard, 1972). The formalism and its properties have been generalized to the Calculus of Constructions (Coquand and Huet, 1988), and later on to Pure Type Systems. Here, we follow the generic presentation of (Barendregt, 1992). The inference rules are given in Fig. 5. Here, the possible product rules are $\{(*,*),(\Box,*),(\Box,\Box)\}$.

$$\frac{}{\emptyset \vdash_{\mathsf{F}\omega} * : \Box}\;(\textsc{axiom}) \qquad \frac{\Gamma, x:\sigma \vdash_{\mathsf{F}\omega} t:\tau \qquad \Gamma \vdash_{\mathsf{F}\omega} \Pi(x:\sigma).\tau : s}{\Gamma \vdash_{\mathsf{F}\omega} \lambda(x:\sigma).t : \Pi(x:\sigma).\tau}\;(\textsc{abs})$$

$$\frac{\Gamma \vdash_{\mathsf{F}\omega} \sigma : s \qquad x \notin \mathcal{D}om(\Gamma)}{\Gamma, x{:}\sigma \vdash_{\mathsf{F}\omega} x:\sigma}\;(\textsc{var}) \qquad \frac{\Gamma \vdash_{\mathsf{F}\omega} t : \Pi(x:\sigma).\tau \qquad \Gamma \vdash_{\mathsf{F}\omega} u:\sigma}{\Gamma \vdash_{\mathsf{F}\omega} t\,u : \tau[x:=u]}\;(\textsc{appl})$$

$$\frac{\Gamma \vdash_{\mathsf{F}\omega} t:\sigma \qquad \Gamma \vdash_{\mathsf{F}\omega} \tau : s \qquad x \notin \mathcal{D}om(\Gamma)}{\Gamma, x{:}\tau \vdash_{\mathsf{F}\omega} t:\sigma}\;(\textsc{weak})$$

$$\frac{\Gamma \vdash_{\mathsf{F}\omega} t:\tau \qquad \Gamma \vdash_{\mathsf{F}\omega} \sigma : s \qquad \sigma =_\beta \tau}{\Gamma \vdash_{\mathsf{F}\omega} t:\sigma}\;(\textsc{conv})$$

$$\frac{\Gamma \vdash_{\mathsf{F}\omega} \sigma : s_1 \qquad \Gamma, x:\sigma \vdash_{\mathsf{F}\omega} \tau : s_2 \qquad (s_1,s_2) \in \{(*,*),(\Box,*),(\Box,\Box)\}}{\Gamma \vdash_{\mathsf{F}\omega} \Pi(x:\sigma).\tau : s_2}\;(\textsc{prod})$$

Figure 5. The typing rules of $\mathsf{F}\omega$

In all the remaining, for a type $\Pi(x:\sigma).\tau$, we will use the usual type arrow abbreviation $\sigma \to \tau$ whenever $x \notin \mathcal{FV}(\tau)$, *i.e.* for terms depending on terms (product rule $(*, *)$) and for types depending on types (product rule $(\Box,\Box)$).

Some well-known properties of this calculus are (Girard, 1972; Barendregt, 1992):

LEMMA 4 (SUBSTITUTION) *If* $\Gamma, x:\sigma, \Delta \vdash_{\mathsf{F}\omega} t:\tau$ *and* $\Gamma \vdash_{\mathsf{F}\omega} u:\sigma$, *then* $\Gamma, \Delta[x:=u] \vdash_{\mathsf{F}\omega} t[x:=u] : \tau[x:=u]$.

THEOREM 3 (SUBJECT REDUCTION)
If $\Gamma \vdash_{\mathsf{F}\omega} t:\sigma$ *and* $t \mapstochar\twoheadrightarrow_\beta t'$, *then* $\Gamma \vdash_{\mathsf{F}\omega} t' : \sigma$.

LEMMA 5 (UNIQUENESS OF TYPES)
If $\Gamma \vdash_{\mathsf{F}\omega} t:\sigma_1$ *and* $\Gamma \vdash_{\mathsf{F}\omega} t:\sigma_2$, *then* $\sigma_1 =_\beta \sigma_2$.

THEOREM 4 (STRONG NORMALIZATION)
If $\Gamma \vdash_{\mathsf{F}\omega} t:\sigma$, *then* t *is strongly normalizing.*

6 The typed translation algorithm

Here, instead of translating a term to a term, we translate a typed term into a (typable) term. For simplicity of presentation, we still write $\llbracket A \rrbracket$ but, as one can see on

Fig. 6, the translation of a term A is generally based on the fact that A is typable. Supposing we are given a type derivation for a judgment $\Sigma, \Gamma \vdash_\rho A : \Phi$, we recursively build a term $\llbracket A \rrbracket$ typable in $\llbracket \Gamma \rrbracket$. There is no translation for Σ since, as we will see in Section 7, the context $x_\perp : \perp$. is sufficient to type $\llbracket f \rrbracket$ for any constant $f \in \Sigma$.

For the rest of the paper, we adopt the following abbreviations, for any types $\sigma, \sigma_1, \ldots, \sigma_n, \tau$ in $\mathsf{F}\omega$. The third definition is a special case of the second one with $\alpha = 0$:

$$\begin{aligned}
[\sigma]^S \to \tau \quad &\triangleq \underbrace{\sigma \to \ldots \to \sigma}_{S} \to \tau \\
\{\sigma_1, \ldots, \sigma_\alpha\} \quad &\triangleq \Pi(\beta : *).([\sigma_1 \to \ldots \sigma_\alpha \to \beta]^S \to \beta) \\
\{\emptyset\} \quad &\triangleq \Pi(\beta : *).([\beta]^S \to \beta)
\end{aligned}$$

For each variable X appearing in a P^2TS term, we add in the corresponding λ-term a type variable β_X which appears in the type of $\llbracket X \rrbracket$. This variable β_X is common to every occurrence of X in the term, and if X is bound, we bind β_X at the same point as X in the translation. If X is free, then in the translation of the context, the type variable β_X appears just before X. The need for β_X is explained in Section 7.

First we define the translation of types (*i.e.* terms such that $\Gamma \vdash_\rho \Phi : *$) by four mutually dependent definitions:

$\llbracket \Phi \rrbracket^X_{\vec{\gamma}}$ translates the type Φ, supposing it is the type of the variable X depending on the list of type variables $\vec{\gamma}$. The free type variable β_X (univocally corresponding to X) appears in this translation.

$$\begin{aligned}
\llbracket \Xi \rrbracket^X_{\vec{\gamma}} &\triangleq \beta_X \vec{\gamma} \quad \textbf{(where } \Xi \textbf{ is atomic)} \\
\llbracket \Pi(P : \Delta).\Psi \rrbracket^X_{\vec{\gamma}} &\triangleq \prod_{(Y:\Phi_Y)\in\Delta} \overrightarrow{\beta_Y : \mathbb{K}(\Phi_Y)_\emptyset}.\,(\ulcorner P \urcorner_\Delta \to \llbracket \Psi \rrbracket^X_{\vec{\gamma} \cup \overrightarrow{\beta_Y}}) \\
\llbracket [P \ll_\Delta B]\Psi \rrbracket^X_{\vec{\gamma}} &\triangleq \prod_{(Y:\Phi_Y)\in\Delta} \overrightarrow{\beta'_Y : \mathbb{K}(\Phi_Y)_\emptyset}.\,\left((\sigma \to \ulcorner P \urcorner_\Delta) \to \llbracket \Psi \rrbracket^X_{\vec{\gamma} \cup \overrightarrow{\beta'_Y}}\right) \\
&\quad \textbf{where } \llbracket \Gamma \rrbracket \vdash_{\mathsf{F}\omega} \llbracket B \rrbracket : \sigma
\end{aligned}$$

$\llbracket \Phi \rrbracket^f_{\vec{\tau}}$ translates the type Φ, supposing it is the type of the constant f depending on the list of types $\vec{\tau}$.

$$\begin{aligned}
\llbracket \Xi \rrbracket^f_{\vec{\tau}} &\triangleq \{\vec{\tau}\} \qquad \textbf{(where } \Xi \textbf{ is atomic)} \\
\llbracket \Pi(P : \Delta)\Psi \rrbracket^f_{\vec{\tau}} &\triangleq \prod_{(Y:\Phi_Y)\in\Delta} \overrightarrow{\beta_Y : \mathbb{K}(\Phi_Y)_\emptyset}.\,(\ulcorner P \urcorner_\Delta \to \llbracket \Psi \rrbracket^f_{\vec{\tau} \cup \ulcorner P \urcorner_\Delta}) \\
\llbracket [P \ll_\Delta B]\Psi \rrbracket^f_{\vec{\tau}} &\triangleq \prod_{(Y:\Phi_Y)\in\Delta} \overrightarrow{\beta'_Y : \mathbb{K}(\Phi_Y)_\emptyset}.\,\left((\sigma \to \ulcorner P \urcorner_\Delta) \to \llbracket \Psi \rrbracket^f_{\vec{\tau}}\right) \\
&\quad \textbf{where } \llbracket \Gamma \rrbracket \vdash_{\mathsf{F}\omega} \llbracket B \rrbracket : \sigma
\end{aligned}$$

The only free variable appearing in $\llbracket \Phi \rrbracket^X_\emptyset$ is β_X, and the arguments $\vec{\gamma}$ of β_X are the bound variables whose scope extends to this subterm of the type. Similarly, in $\llbracket \Phi \rrbracket^f_\emptyset$, no variable is free, and all the bound variables whose scope extends to the subterm $\{\vec{\tau}\}$ are represented in this subterm.

$\ulcorner P \urcorner_\Delta$ flattens a pattern P with $\mathcal{FV}(P) \subseteq \mathcal{D}om(\Delta)$. Since patterns appear in the P^2TS types, the translation at the type level must be accurate.

$$\begin{aligned}
\ulcorner f_i \bullet \vec{P} \urcorner_\Delta &\triangleq \llbracket \Pi(P'_{p+1} : \Delta_{p+1}) \ldots \Pi(P'_{\alpha_i} : \Delta_{\alpha_i}) \,.\, \Xi \rrbracket^{f}_{\xrightarrow[\ulcorner P \urcorner_\Delta]{}} \\
&\qquad \text{where } \Sigma \vdash_\rho f_i : \Pi(P'_1 : \Delta_1) \ldots \Pi(P'_{\alpha_i} : \Delta_{\alpha_i}) \,.\, \Xi \\
\ulcorner X \urcorner_\Delta &\triangleq \llbracket \Phi \rrbracket^X_\emptyset \qquad \text{if } X : \Phi \in \Delta
\end{aligned}$$

$\mathbb{K}(\Phi)_{\vec{k}}$ computes the kind of β_X if X has type Φ.

$$\begin{aligned}
\mathbb{K}(\Xi)_{\vec{k}} &\triangleq \vec{k} \to * \\
\mathbb{K}(\Pi(P : \Delta).\Psi)_{\vec{k}} &\triangleq \mathbb{K}(\Psi)_{\vec{k} \cup \bigcup_{(Y:\Phi_Y) \in \Delta} \mathbb{K}(\Phi_Y)_\emptyset} \\
\mathbb{K}([P \ll_\Delta B]\Psi)_{\vec{k}} &\triangleq \mathbb{K}(\Psi)_{\vec{k} \cup \bigcup_{(Y:\Phi_Y) \in \Delta} \mathbb{K}(\Phi_Y)_\emptyset}
\end{aligned}$$

We can extend this translation to contexts, the base case being given by $x_\perp$:

$$\begin{aligned}
\llbracket \emptyset \rrbracket &\triangleq x_\perp : \perp \\
\llbracket \Gamma, X : \Phi \rrbracket &\triangleq \llbracket \Gamma \rrbracket,\ \beta_X : \mathbb{K}(\Phi)_\emptyset,\ X : \llbracket \Phi \rrbracket^X_\emptyset \qquad (\text{if } \Gamma \vdash_\rho \Phi : *) \\
\llbracket \Gamma, X : * \rrbracket &\triangleq \llbracket \Gamma \rrbracket, X : *
\end{aligned}$$

Finally, we can translate typed terms. The translation is given in two distinct parts: in Fig. 6, we give all the cases that are simply adapted from the untyped case. In Fig. 7, we deal with the trickiest situations: matching constraints and conversion in the types. These last cases are further explained in Section 7.

7 Rationale of the typed translation

In this section we treat three key constructs of the typed translation:

1 the type of a translated constant (accounting for the use of System F);

2 the type of a variable (requiring types depending on types);

3 the translation of matching constraints appearing in the P^2TS types.

Typing the translation of a constant

First, let us study how constants and their translation affect typing. In order to get a typed translation, in the previous section, we have added to the untyped term $\llbracket f_i \rrbracket$ some type abstractions. The type abstractions $\lambda(\beta_Y : \mathbb{K}(\Phi_Y)_\emptyset)$ are needed for correctly typing the variables, as we will see in the next subsection. Here, we are interested in the type abstraction $\lambda(\beta : *)$ appearing in $\llbracket f_i \rrbracket$.

To explain the modifications we made, let us start from the untyped translation. We suppose $\vdash f_i : \Pi P_1 \ldots \Pi P_{\alpha_i} \,.\, \Xi$ where Ξ is an atomic type, and we assume that each P_n is translated to a certain type σ_n. Then we have:

$$x_\perp : \beta \vdash \llbracket f_i \rrbracket : \sigma_1 \to \ldots \to \sigma_\alpha \to [\sigma_1 \to \ldots \to \sigma_\alpha \to \beta]^S \to \beta$$

What remains unclear is the meaning of β. The type of a translated abstraction is:

$$\vdash \llbracket \lambda(f_i \bullet \vec{X}_{(1..p)}).A \rrbracket : \left(\sigma_{p+1} \to \ldots \to \sigma_\alpha \to [\sigma_1 \to \ldots \to \sigma_\alpha \to \tau]^S \to \gamma\right) \to \gamma$$

$$
\begin{aligned}
\llbracket X \rrbracket &\triangleq X \\
\llbracket f_i \rrbracket &\triangleq \lambda\overrightarrow{\beta_Y}_1.\lambda x_1 \ldots \lambda\overrightarrow{\beta_Y}_{\alpha_i}.\lambda x_{\alpha_i}.\lambda(\beta:*)\,(\lambda\overrightarrow{z}_{(1..S)}.(z_i\,\overrightarrow{x}_{(1..\alpha_i)})) \\
&\quad \text{where } \Sigma \vdash_\rho f_i : \mathbf{\Pi}(P_1:\Delta_1)\ldots\mathbf{\Pi}(P_{\alpha_i}:\Delta_{\alpha_i})\,.\,\Xi \\
&\quad \text{and } \overrightarrow{\beta_Y}_n \text{ correspond to } \overrightarrow{(Y:\Phi_Y)} \in \Delta_n, \text{ with } \overrightarrow{\beta_Y : \mathbb{K}(\Phi_Y)_\emptyset}. \\
\llbracket A;B \rrbracket &\triangleq \overrightarrow{\lambda\overrightarrow{\beta_Y}}.\lambda x_{(1..\alpha)}.\Big(\lambda z.\big(z(\llbracket A\rrbracket\overrightarrow{\overrightarrow{\beta_Y}}x_{(1..\alpha)})(\llbracket B\rrbracket\overrightarrow{\overrightarrow{\beta_Y}}x_{(1..\alpha)})\big)\Big) \\
&\quad \text{where } \Sigma,\Gamma \vdash_\rho A;B : \mathbf{\Pi}(P_1:\Delta_1)\ldots\mathbf{\Pi}(P_{\alpha_i}:\Delta_{\alpha_i})\,.\,\Xi \\
&\quad \text{and } \overrightarrow{\beta_Y}_n \text{ are the type variables corresponding to } \mathcal{FV}(P_n). \\
\llbracket \boldsymbol{\lambda}(X:\Phi).A \rrbracket &\triangleq \lambda(\beta_X:\mathbb{K}(\Phi)_\emptyset).\lambda(X:\llbracket\Phi\rrbracket_\emptyset^X).\llbracket A\rrbracket \\
&\quad \text{where } \Sigma,\Gamma \vdash_\rho \boldsymbol{\lambda}(X:\Phi).A : \mathbf{\Pi}(X:\Phi).\Psi \\
\llbracket \boldsymbol{\lambda}(f_i\bullet\overrightarrow{P}_{(1..p)}:\Delta).A \rrbracket &\triangleq \lambda_{X\in\Delta}(\overrightarrow{\beta_X:\mathbb{K}(\Phi_X)}).\lambda y.(y\overrightarrow{\bot}\,\overrightarrow{(x_\bot\ \ulcorner P'\urcorner_\Delta[\beta_Y:=\bot])}_{(p+1..\alpha_i)} \\
&\qquad \tau\ \overrightarrow{(x_\bot\tau_0)}\ \llbracket\boldsymbol{\lambda}\overrightarrow{P{:}\Delta}_{(1..p)}.\boldsymbol{\lambda}\overrightarrow{x'{:}\ \ulcorner P'\urcorner_\Delta}_{(p+1..\alpha_i)}.A\rrbracket\ \overrightarrow{(x_\bot\tau_0)}) \\
&\quad \text{where } \Sigma,\Gamma \vdash_\rho \boldsymbol{\lambda}(f_i\bullet\overrightarrow{P}_{(1..p)}:\Delta).A : \mathbf{\Pi}(f_i\bullet\overrightarrow{P}_{(1..p)}:\Delta).\Psi \\
&\quad \text{and } \Sigma \vdash_\rho f_i : \mathbf{\Pi}\overrightarrow{(P':\Delta)}_{(1..\alpha_i)}).\Xi \quad \text{and } \llbracket\Gamma\rrbracket \vdash_{\mathsf{F}\omega} \llbracket A\rrbracket : \tau \\
&\quad \text{and } \llbracket\Gamma\rrbracket \vdash_{\mathsf{F}\omega} \llbracket\boldsymbol{\lambda}\overrightarrow{P}_{(1..p)}.\boldsymbol{\lambda}\overrightarrow{x'}_{(p+1..\alpha_i)}.A\rrbracket : \tau_0 \\
\llbracket A\bullet B \rrbracket &\triangleq \llbracket A\rrbracket\,\overrightarrow{\tau_X}\,\llbracket B\rrbracket \qquad \text{if } \llbracket\Gamma\rrbracket \vdash_{\mathsf{F}\omega} \llbracket B\rrbracket : \ulcorner P\urcorner_\Delta\,[\overrightarrow{\beta_X:=\tau_X}]_{X\in\Delta} \\
&\quad \text{where } \Sigma,\Gamma \vdash_\rho A\bullet B : [P \ll_\Delta B]\Psi \\
\llbracket [P\ll_\Delta B]A \rrbracket &\triangleq \text{the term obtained by head-}\beta\text{-reducing } \llbracket(\boldsymbol{\lambda}(P:\Delta).A)\bullet B\rrbracket
\end{aligned}
$$

Figure 6. **Typed term translation without matching constraints**

Therefore, the $\boldsymbol{\lambda}$-term $\llbracket\boldsymbol{\lambda}(f_i\bullet\overrightarrow{X}_{(1..p)}).A\rrbracket\,(\llbracket f_i\rrbracket\llbracket B_1\rrbracket\ldots\llbracket B_p\rrbracket)$ has a valid type only if $[\sigma_1\to\ldots\to\sigma_\alpha\to\tau]^S\to\gamma=[\sigma_1\to\ldots\to\sigma_\alpha\to\beta]^S\to\beta$, *i.e.* $\tau=\beta=\gamma$ and $\vdash\llbracket B_1\rrbracket{:}\sigma_1\ \ldots\vdash\llbracket B_p\rrbracket{:}\sigma_p$. The types β and γ should be replaced by the return type τ of the function which is applied to $\llbracket f_i\bullet\overrightarrow{B}\rrbracket$. Since one can not guess what function will be applied to a given term, we introduce the polymorphism of Girard's System F in the target language. The resulting modification can be seen on Fig. 6: in $\llbracket f_i\rrbracket$, we abstract over the type variable β, which is instantiated with τ by $\llbracket\boldsymbol{\lambda}(f_i\bullet\overrightarrow{P}).A\rrbracket$.

Thanks to polymorphism, the variable $x_\bot$ can get type $\Pi(\iota:*).\iota$, which is usually noted $\bot$. Then, if we need an arbitrary term with type σ, we use $x_\bot\sigma$. This means that all the $\boldsymbol{\lambda}$-terms we build are typable in a context containing $x_\bot{:}\bot$; again, we can add an abstraction "$\lambda(x_\bot{:}\bot)$" to get a closed term.

The types $\llbracket\Phi\rrbracket_\emptyset^f$ have been built to fit with the new translation of constants: a translated constant $\llbracket f_i\rrbracket$ with arity α_i takes α_i arguments with types $\sigma_1\ldots\sigma_{\alpha_i}$ and returns a term with type $\{\sigma_1,\ldots\sigma_{\alpha_i}\}$. The types $\ulcorner P\urcorner_\Delta$ extend this notion to nested patterns: for instance $\llbracket f\bullet(g\bullet x_1)\bullet x_2\rrbracket$ will have type $\{\{\sigma_1\},\sigma_2\}$. This flattening process keeps the shape of the pattern but forgets the constants used.

Typing a variable

In this subsection, we explain why we need a new type variable β_X for each variable X appearing in a $\boldsymbol{P^2TS}$ term (including bound variables appearing in a type).

- *(Constraint postponement):* if $\nexists \overrightarrow{\tau_X}, \sigma =_\beta \ulcorner P \urcorner_\Delta \, [\overrightarrow{\beta_X := \tau_X}]_{X \in \mathcal{D}om(\Delta)}$

  $\llbracket A \bullet B \rrbracket \triangleq \lambda_{(Y \in \Delta)} \overrightarrow{\beta'_Y : \mathbb{K}(\Phi_Y)}. \lambda\left(w : \sigma \to \ulcorner P_{[Y:=Y']} \urcorner_\Delta\right) . \left(\llbracket A \rrbracket \, \overrightarrow{\beta'_Y} \, (w \llbracket B \rrbracket)\right)$

  where $\Sigma, \Gamma \vdash_\rho A \bullet B : [P \ll_\Delta B]\Psi$

  and $\llbracket \Gamma \rrbracket \vdash_{F\omega} \llbracket B \rrbracket : \sigma$

- *(Constraint resolution)* : For a postponement variable $w : \sigma \to \ulcorner P \urcorner_\Delta$ appearing in a term $\llbracket A \rrbracket$, whenever a subsequent instantiation θ of some free type variables (in σ) enforces:

  $$\exists \overrightarrow{\tau_X}, \, \sigma\theta \; =_\beta \; \ulcorner P \urcorner_\Delta \, [\overrightarrow{\beta_X := \tau_X}]_{X \in \mathcal{D}om(\Delta)}$$

  If $\Sigma, \Gamma \vdash_\rho A : \Phi$, replace $\llbracket A \rrbracket$ with $solve(\llbracket A \rrbracket, \Phi)$ defined as follows:

  $$solve(t, \Pi P.\Psi) \triangleq \lambda \overrightarrow{\beta'_X}. \left(\llbracket \lambda P_{[X:=X']}. \rrbracket \; solve(t \overrightarrow{\beta'_X} \llbracket P_{[X:=X']} \rrbracket, \Psi)\right)$$

  (where $\llbracket \lambda P. \rrbracket$ denotes the same abstractions as in Fig. 6 when encoding $\llbracket \lambda P.A \rrbracket$)

  $$solve(t, [P \ll_\Delta B]\Psi) \triangleq \lambda_{Y \in \mathcal{D}om(\Delta)} \overrightarrow{\beta''_Y}. \lambda w'. \, solve(t \overrightarrow{\beta''_Y} w', \Psi)$$

  if $P \ll B$ is not the matching constraint associated to w.

  $$solve(t, [P \ll_\Delta B]\Psi) \triangleq t \, \overrightarrow{\tau_X} \, (\lambda(x : \sigma\theta).x)$$

  if $P \ll B$ is the matching constraint associated to w.

Figure 7. **Typed term translation for delayed matching constraints**

Consider the following examples:

$$\begin{array}{rl} (\Xi : *), (X : \Pi(Y : \Xi).\Xi) \vdash_\rho X & : \Pi(Y : \Xi).\Xi \\ (\Xi : *) \vdash_\rho \lambda(Y : \Xi).Y & : \Pi(Y : \Xi).\Xi \\ (\Xi : *), (f : \Pi(Y : \Xi).\Xi) \vdash_\rho f & : \Pi(Y : \Xi).\Xi \end{array}$$

Both terms $\lambda Y.Y$ and f can instantiate X since they have the same type. However, the typed translation gives (in the context $\Gamma = (x_\perp : \perp)$):

$$\begin{array}{ll} \Gamma \vdash_{F\omega} \lambda(\beta_Y : *).\lambda(Y : \beta_Y).Y & : \Pi(\beta_Y : *).\beta_Y \to \beta_Y \\ \Gamma \vdash_{F\omega} \llbracket f \rrbracket & : \Pi(\beta_Y : *).\beta_Y \to \{\beta_Y\} \end{array}$$

The type variable β_X which appears in $\vdash_{F\omega} X : \Pi(\beta_Y : *).\beta_Y \to \beta_X \beta_Y$ allows us to treat both cases: an abstraction $\lambda X.A$ is translated into $\lambda \beta_X.\lambda X.\llbracket A \rrbracket$, so we can give the expected type to X if we instantiate β_X with the correct term:

$\lambda Y.Y$	$\beta_X := \lambda(\beta : *).\beta$	$\beta_X \beta_Y \mapsto_\beta \beta_Y$
f	$\beta_X := \lambda(\beta : *).\{\beta\}$	$\beta_X \beta_Y \mapsto_\beta \{\beta_Y\}$

The need for types depending on types appears here: β_X must be able to build a new type where some type variables, like β_Y, may appear whereas they are bound in the type of X. The function $\mathbb{K}(\cdot)$ computes a suitable kind for β_X according to the kinds of the arguments β_Y of β_X. Here, we have $\vdash_{F\omega} \beta_X : * \to *$.

Translating matching constraints appearing in P^2TS types

The part of the typed translation shown in Fig. 6 mainly consists in correctly combining information obtained by the translation of smaller terms. However, for application (and matching constraints), the argument of a function must transmit it some type information. In P^2TS, this process is initiated by the matching constraints appearing in types, and carried on by the conversion rule.

In System $\mathsf{F}\omega$, we can not encode pattern matching in the types, so matching constraints must be treated at the meta-level, *i.e.* during the translation. Let us study the two kinds of matching constraints appearing in the types:

$[P \ll_\Delta B]\Psi$ **with** $\exists\theta,\ B =_{\rho\sigma\delta} P\theta$: By successive application of Lemmas 2, 1 and 4, we can prove that the same equality holds for the types in System $\mathsf{F}\omega$: if $[\![B]\!]$ has type σ, then $\exists\overrightarrow{\tau_X},\ \sigma =_\beta \ulcorner P \lrcorner_\Delta\ [\overrightarrow{\beta_X := \tau_X}]$ where $\overrightarrow{X} = \mathcal{FV}(P)$. The proof of Theorem 5 is constructive: it gives an algorithm for computing the $\overrightarrow{\tau_X}$.

$[P \ll_\Delta B]\Psi$ **with** $\forall\theta,\ B \neq_{\rho\sigma\delta} P\theta$: In this case, a new postponement variable w is created with type $\sigma \rightarrow \ulcorner P \lrcorner_\Delta$, where σ is the type of $[\![B]\!]$. The type of w appears in $[\![[P \ll B]\Psi]\!]_\emptyset^X$, accounting for the delayed matching constraint in the type. The term $w[\![B]\!]$ is used so that the **λ-term** is well-typed, since $[\![\lambda(P : \Delta).A]\!]$ expects a term of type $\ulcorner P \lrcorner_\Delta$. Suppose some subsequent applications instantiate some free variables in B (replacing it with a term $B\theta_0$) such that $\exists\theta, B\theta_0 =_{\rho\sigma\delta} P\theta$. Then, we should instantiate the free type variables $\overrightarrow{\beta_X}$ of $\ulcorner P \lrcorner_\Delta$ with suitable types $\overrightarrow{\tau_X}$ and instantiate w with the identity since we had translated B into $w[\![B]\!]$.

From a typing point of view, it is sound: because of the substitutions θ_0 and θ, the type of w is now $\sigma[\![\theta_0]\!] \rightarrow \ulcorner P \lrcorner_\Delta\ [\overrightarrow{\beta_X := \tau_X}]$ and the equality $B\theta_0 =_{\rho\sigma\delta} P\theta$ ensures that $\sigma[\![\theta_0]\!] =_\beta \ulcorner P \lrcorner_\Delta\ [\overrightarrow{\beta_X := \tau_X}]$, which means w has a suitable type for identity. The subtle point is that w can be located quite deep in the term we are considering: this is why we use the function $solve(\cdot,\cdot)$ given in Fig. 7, which performs a kind of **η-expansion** to instantiate w.

8 Strong normalization

In this section, we give the properties of our typed encoding.

PROPOSITION 1 (FAITHFUL REDUCTIONS) *Lemma 1 and Theorem 1 are still valid with the typed translation: each* **ρσδ-reduction** *can be mimicked by at least one* **β-reduction** *(and the postponement variables* w *only prevent unsuccessful matchings).*

LEMMA 6 (WELL-KINDEDNESS)
$$\forall\Sigma,\forall\Gamma,\forall\Phi,\quad \Sigma,\Gamma \vdash_\rho \Phi : * \ \Rightarrow\ [\![\Gamma]\!], \beta_X : \mathbb{K}(\Phi)_\emptyset \vdash_{\mathsf{F}\omega} [\![\Phi]\!]_\emptyset^X : *$$

THEOREM 5 (WELL-TYPED TRANSLATION) $\forall\Sigma,\Gamma, A, \Phi,$ *if* $\Sigma,\Gamma \vdash_\rho A{:}\Phi$ *then, for afresh variable* Z, $\exists\tau_A,\quad [\![\Gamma]\!] \vdash_{\mathsf{F}\omega} [\![A]\!] : [\![\Phi]\!]_\emptyset^Z[\beta_Z := \tau_A]$

THEOREM 6 (STRONG NORMALIZATION OF TYPABLE P^2TS TERMS)
$\forall\Sigma,\Gamma, A, \Phi,$ *if* $\Sigma,\Gamma \vdash_\rho A : \Phi$ *then A is strongly normalizing.*

Proof: A P^2TS**-typable** term A is translated into an $\mathsf{F}\omega$**-typable** term which has no infinite reduction, so by Proposition 1, A is strongly normalizing. □

9 Conclusion and perspectives

We have proved strong normalization of the simply-typed P^2TS by translating it into System $\mathsf{F}\omega$. First, we have shown how to encode untyped syntactic pattern matching in the λ-calculus. Introducing types in the translation then proved an interesting challenge. One difficulty comes from the pattern matching occuring in the P^2TS types, which calls for accurate adjustments in the translation. Another remarkable point is that the typing mechanisms of P^2TS can be expressed only with the expressive power of System $F\omega$, which is rather surprising since we only deal with the simply-typed P^2TS. This fact leads us to think that, with the same product rules, the expressive power of P^2TS is greater than the one of the λ-calculus.

An interesting development of this work would be to adapt the proof for the other type systems of P^2TS. In the long term, we expect to use P^2TS as the base language for a powerful proof assistant combining the logical soundness of the λ-calculus and the computational power of the rewriting. This proof of strong normalization is a main stepstone for this research direction, since logical soundness is deeply related to strong normalization.

Acknowledgements Thanks to H. Cirstea, C. Kirchner and L. Liquori for the constant support and interest they put in this work; P. Blackburn for some useful insights about the typed λ-calculus; S. Salvati for many fruitful informal discussions about System F; F. Blanqui, G. Dowek and anonymous referees for their valuable comments.

Long version A detailed version of this article containing proofs and type derivations can be found at `http://www.loria.fr/~wack/papers/rhoSN.ps.gz`.

References

Barendregt, H. P. (1992). Lambda calculi with types. In Abramsky, S., Gabbay, D., and Maibaum, T., editors, *Handbook of Logic in Computer Science.* Clarendon Press.

Barthe, G., Cirstea, H., Kirchner, C., and Liquori, L. (2003). Pure Patterns Type Systems. In *POPL 2003, New Orleans, USA.* ACM.

Blanqui, F. (2001). Definitions by rewriting in the calculus of constructions. In *LICS,* pages 9–18.

Cirstea, H. and Kirchner, C. (2000). The typed rewriting calculus. In *Third International Workshop on Rewriting Logic and Application,* Kanazawa (Japan).

Cirstea, H., Kirchner, C., and Liquori, L. (2001). The Rho Cube. In Honsell, F., editor, *FOSSACS,* volume 2030 of *LNCS,* pages 166–180, Genova, Italy.

Cirstea, H., Liquori, L., and Wack, B. (2004). Rewriting calculus with fixpoints: Untyped and first-order systems. In *TYPES'03,* LNCS, Torino. To be published.

Coquand, T. (1992). Pattern matching with dependent types. In *Informal proceedings workshop on types for proofs and programs,* pages 71–84. Båstad, Suède.

Coquand, T. and Huet, G. (1988). The calculus of constructions. *Information and Computation,* 76:95–120.

Dowek, G., Hardin, T., and Kirchner, C. (2003). Theorem proving modulo, revised version. Rapport de Recherche 4861, INRIA.

Girard, J.-Y. (1972). *Interprétation fonctionnelle et élimination des coupures de l'arithmetique d'ordre supérieur.* PhD thesis, Université Paris VII.

Kesner, D., Puel, L., and Tannen, V. (1996). A typed pattern calculus. *Information and Computation,* 124(1):32–61.

Klop, J., van Oostrom, V., and van Raamsdonk, F. (1993). Combinatory reduction systems: introduction and survey. *TCS,* 121:279–308.

Werner, B. (1994). *Une Théorie des Constructions Inductives.* PhD thesis, Université Paris VII.