Absolute Explicit Unification

Nikolaj Bjørner[1*] and César Muñoz[2**]

[1] Kestrel Institute `nikolaj@kestrel.edu`
[2] Institute for Computer Applications in Science and Engineering (ICASE)
`munoz@icase.edu`

Abstract. This paper presents a system for explicit substitutions in Pure Type Systems (PTS). The system allows to solve type checking, type inhabitation, higher-order unification, and type inference for PTS using purely first-order machinery. A novel feature of our system is that it combines substitutions and variable declarations. This allows as a side-effect to type check let-bindings. Our treatment of meta-variables is also explicit, such that instantiations of meta-variables is internalized in the calculus. This produces a confluent λ-calculus with distinguished holes and explicit substitutions that is insensitive to α-conversion, and allows directly embedding the system into rewriting logic.

1 Introduction

Explicit substitutions provide a convenient framework for encoding higher-order typed λ-calculus using first-order machinery. In particular, this allows to integrate higher-order unification with first-order provers, rewriting logic, and to delay evaluation and resolve scoping when type checking dependent-typed terms. On the other hand, several problems related to type theory, such as type checking (with definitions), type inference, checking equality of well typed terms, proof-term refinement, and the inhabitation problem, can be solved using the same machinery, once it is properly developed. We therefore here combine explicit substitutions, variable declarations as explicit substitutions, and explicit instantiation of meta-variables using first-order rewrite rules. The combination is formulated for pure type systems, and applies therefore for arbitrary type systems as those of the λ-cube. A higher-order unification procedure for systems in the λ-cube is a particular payoff.

It is well-known that definitions, i.e., let-in expressions, are problematic in dependent-type systems [24, 4]. Two approaches have been used to extend the λ-calculus with definitions. Severi and Poll [24] consider definitions as terms and extend the reduction relation to unfold definitions during the typing process. Bloo *et al* [4] do not extend the syntax of terms (although they use a different notation for terms called *item notation*), but they consider definitions as part

* This research is supported by NASA under award No NAG 2-1227, and DARPA under Contract No. F30602-96-C-0282.

** This research was supported by the National Aeronautics and Space Administration under NASA Contract NAS1-97046.

of the contexts. Combining the two approaches, Bloo [3] proposes a calculus of explicit substitutions where substitutions are also part of the contexts. In Bloo's system, α-conversion remains as an implicit rule and non-explicit substitutions are still required to unfold definitions on types. The unfolding of definitions on types is inconvenient when they contain meta-variables. In this paper, we propose a system where context are special case of explicit substitutions, we can safely extend our calculus with definitions and meta-variables, and yet be completely insensitive to α-conversion.

Finally, our extended calculus of explicit substitutions, where instantiations are also explicit, realizes the spirit of delayed evaluation to also cover instantiation of meta-variables. Indeed, our calculus can be considered a λ-calculus of contexts[1] i.e., a λ-calculus with place-holders where the mechanism of filling in holes is explicit.

The rest of this paper is organized as follows. We continue with a short review of *pure type systems* and *explicit substitutions*. More detailed descriptions can be found in [2] and [1]. Section 2 introduces the system $\mathrm{PTS}_{\mathcal{L}}$, which is a system of pure types with explicit substitutions. Section 3 presents two novel aspects of $\mathrm{PTS}_{\mathcal{L}}$ with respect to other proposals: definitions and explicit instantiations. Section 4 summarizes a meta-theoretical investigation, and Section 5 applies the system to higher-order unification and related typing problems.

Pure Type Systems: *Pure Type Systems* [14, 2] is a formalism to describe a family of type systems which generalizes the cube of typed lambda calculi. A *Pure Type System*, PTS for short, is a typed λ-calculus given by a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, where $\mathcal{S}$ is a base set of *sorts*, $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ is a set of *axioms*, and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is a set of *rules*. A sort s_1 is a *top sort* if it does not exist a sort s_2 such that $(s_1, s_2) \in \mathcal{A}$.

The set of PTS *(pseudo-)terms* is formed by variables $x, y, \ldots$, applications: $(M\ N)$, abstractions: $\lambda x{:}A.M$, products: $\Pi x{:}A.B$, and sorts $s \in \mathcal{S}$. Abstractions and products are binding structures. As usual, in higher-order formal systems, terms are considered equivalents modulo α-conversion, i.e., renaming of bound variables. Notice that there is no syntactical distinction between terms denoting objects and terms denoting types. For sake of readability, we use the uppercase Roman letters $A, B, \ldots$ to range over terms denoting types. The notation $A \to B$ is used for $\Pi x{:}A.B$ when x does not appear free in B.

A PTS type judgment has the form $\Gamma \vdash M : A$ where Γ is a *typing context*, i.e., a list $x_1 : A_1, \ldots, x_n : A_n$ of type assignments for variables, where $n \geq 0$ and $x_i \neq x_j$ for $i \neq j$. We use the Greek letters Γ, Δ to range over typing contexts. A variable x is *fresh* (in Γ) if $x \neq x_i$ for $1 \leq i \leq n$. The typing rules of the PTS defined by $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ are shown in Figure 1. In rules (Start) and (Weak), we assume that x is fresh in Γ.

We here consider the extension of PTS with the η-equality, i.e., the relation $\equiv$ in rule (Conv) is the congruence relation induced by the rewrites β:

[1] In a typed calculus, the word *context* has two meanings: variable declaration and expression with a distinguished hole. When confusion may arise, we will write *typing context* in the former case and *expression context* in the latter one.

$$\frac{(s_1, s_2) \in \mathcal{A}}{\vdash s_1 : s_2} \text{(Sort)} \qquad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \text{(Start)}$$

$$\frac{\Gamma \vdash A : s \quad \Gamma \vdash M : B}{\Gamma, x : A \vdash M : B} \text{(Weak)} \qquad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \quad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash \Pi x{:}A.B : s_3} \text{(Prod)}$$

$$\frac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B} \text{(Abs)} \qquad \frac{\Gamma \vdash M : \Pi x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash (M\ N) : B[N/x]} \text{(Appl)} \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s \quad A \equiv B}{\Gamma \vdash M : B} \text{(Conv)}$$

Fig. 1. PTS typing rules

$(\lambda x{:}A.M\ N) \longrightarrow M[N/x]$ and η: $\lambda y{:}A.(M\ y) \longrightarrow M$ where y is not free in M. Recall that $M[N/x]$ denotes atomic substitution of the free occurrences of x in M by N, with renaming of bound variables in M when necessary.

A PTS is *functional* if (1) $(s, s_1), (s, s_2) \in \mathcal{A}$ implies $s_1 = s_2$ and (2) $(s, s', s_1), (s, s', s_2) \in \mathcal{R}$ implies $s_1 = s_2$. The *cube of type systems* (Barendregt's cube) [2] are the PTS such that $\mathcal{S} = \{\star, \Box\}$, $\mathcal{A} = \{(\star, \Box)\}$, $(\star, \star, \star) \in \mathcal{R}$, and for (s_1, s_2, s_3) in $\mathcal{R}$, it holds $s_2 = s_3$. Well-known type systems of Barendregt's cube: the simply-typed λ-calculus, system F, LF, and the calculus of constructions, are all functional. For example, the simply-typed λ-calculus has $\mathcal{R} = \{(\star, \star, \star)\}$.

Explicit substitutions: The $\lambda\sigma$-calculus [1] is a first-order rewrite system with two sorts of expressions: *terms* and *substitutions*. By using substitutions as first-class objects and de Bruijn indices notation for variables, the $\lambda\sigma$-calculus allows a first-order encoding of the λ-calculus. In consequence, technical nuisances due to higher-order aspects of the λ-calculus can be minimized or eliminated (e.g., α-conversion) in explicit substitution calculi.

The rewrite system of $\lambda\sigma$ includes a surjective-pairing rule (SCons): $\underline{1}[S] \cdot (\uparrow \circ S) \longrightarrow S$. Rule (SCons) is responsible of confluence and typing problems in $\lambda\sigma$ [9, 20]. These problems are overcame in a variant of $\lambda\sigma$, called $\lambda_{\mathcal{L}}$ [20]. The $\lambda_{\mathcal{L}}$-calculus has the same general features as $\lambda\sigma$, i.e., simple, finite, and first-order presentation, but it does not contain rule (SCons).

Expressions of the untyped version of $\lambda_{\mathcal{L}}$ are *terms*: $\underline{1}$, applications $(M\ N)$, abstractions λM, and closures $M[S]$; and *explicit substitutions*: $\uparrow^n$, $M \cdot S$, and $S \circ T$; where M, N range over terms, S, T range over substitutions, and n ranges over natural numbers constructed with 0 and $n + 1$. The $\lambda_{\mathcal{L}}$-calculus is given in Figure 2. Free and bound variables are represented by de Bruijn indices. They are encoded by means of the constant $\underline{1}$ and the substitution $\uparrow^n$. We overload the notation $\underline{n}$ to represent the $\lambda_{\mathcal{L}}$-term corresponding to the index n, i.e., $\underline{n+1} = \underline{1}[\uparrow^n]$. The occurrence of an index i in a term M is *free* when that occurrence is bound by j λ-abstractions and $j < i$. By convenience, we write *free indices* to mean free occurrences of indices in a given term.

An explicit substitution denotes a mapping from indices to terms. Thus, $\uparrow^n$ maps each index i to the term $\underline{i+n}$, $S \circ T$ is the composition of the mapping denoted by T with the mapping denoted by S (notice that the composition of substitution follows a reverse order with respect to the usual notation of function composition), and finally, $M \cdot S$ maps the index 1 to the term M, and recursively, the index $i + 1$ to the term mapped by the substitution S on the index i.

The $\lambda_{\mathcal{L}}$-calculus, just as $\lambda\sigma$, uses the composition operation to achieve confluence on the calculus of substitutions (the calculus without (Beta)). [2]

$$
\begin{array}{llll@{\quad}llll}
(\lambda M\ N) & \longrightarrow & M[N \cdot \uparrow^0] & \text{(Beta)} & (M \cdot S) \circ T & \longrightarrow & M[T] \cdot (S \circ T) & \text{(Map)} \\
(\lambda M)[S] & \longrightarrow & \lambda M[\underline{1} \cdot (S \circ \uparrow^1)] & \text{(Lambda)} & \uparrow^0 \circ S & \longrightarrow & S & \text{(IdS)} \\
(M\ N)[S] & \longrightarrow & (M[S]\ N[S]) & \text{(Application)} & \uparrow^{n+1} \circ (M \cdot S) & \longrightarrow & \uparrow^n \circ S & \text{(ShiftCons)} \\
M[S][T] & \longrightarrow & M[S \circ T] & \text{(Clos)} & \uparrow^{n+1} \circ \uparrow^m & \longrightarrow & \uparrow^n \circ \uparrow^{m+1} & \text{(ShiftShift)} \\
\underline{1}[M \cdot S] & \longrightarrow & M & \text{(VarCons)} & \underline{1} \cdot \uparrow^1 & \longrightarrow & \uparrow^0 & \text{(Shift0)} \\
M[\uparrow^0] & \longrightarrow & M & \text{(Id)} & \underline{1}[\uparrow^n] \cdot \uparrow^{n+1} & \longrightarrow & \uparrow^n & \text{(ShiftS)}
\end{array}
$$

Fig. 2. The $\lambda_{\mathcal{L}}$-rewrite system

2 Pure Type Systems with Explicit Substitutions

In this section, we present an explicit substitution λ-calculus for PTS, namely $\text{PTS}_{\mathcal{L}}$. The main features of $\text{PTS}_{\mathcal{L}}$ are: a first-order setting insensitive to α-conversion, typing contexts as explicit substitutions, and support for expression contexts. As previously pointed out, higher-order aspects of the λ-calculus, including α-conversion, may be handled in a first-order setting via explicit substitutions *and* de Bruijn indices. We use the $\lambda_{\mathcal{L}}$-calculus as the base calculus of $\text{PTS}_{\mathcal{L}}$.

The $\text{PTS}_{\mathcal{L}}$-system: As in the case of PTS, a $\text{PTS}_{\mathcal{L}}$ is defined by a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ of sorts, axioms, and rules. The grammar of well-formed $\text{PTS}_{\mathcal{L}}$ *(pseudo-)expressions* extends the one of $\lambda_{\mathcal{L}}$ with sorts s, meta-variables X, type annotated abstractions $\lambda_A.M$, products $\Pi_A.B$, and type annotated substitutions $M \cdot_A S$. In $\text{PTS}_{\mathcal{L}}$, meta-variables, as well as substitutions, are first-class objects. However, only meta-variables on the sort of terms are allowed. We assume a set $\mathcal{V}$ of meta-variables. This set will be precisely defined in Section 3.

An expression is *ground* if it does not contain meta-variables. A ground expression is also *pure* if it does not contain other explicit substitutions than those appearing in the terms denoting de Bruijn indices (i.e., in terms of the form $\underline{i}$).

We define the $\lambda\Pi_{\mathcal{L}}$-rewrite system as the extension to $\lambda_{\mathcal{L}}$ with products and sorts given by Figure 3. The system $\Pi_{\mathcal{L}}$ is obtained by dropping rule (Beta) from $\lambda\Pi_{\mathcal{L}}$.

$$
\begin{array}{lll@{\qquad}lll}
(\lambda_A.M\ N) & \longrightarrow & M[N \cdot_A \uparrow^0] & (M \cdot_A S) \circ T & \longrightarrow & M[T] \cdot_A (S \circ T) \\
(\lambda_A.M)[S] & \longrightarrow & \lambda_{A[S]}.M[\underline{1} \cdot_A (S \circ \uparrow^1)] & \uparrow^0 \circ S & \longrightarrow & S \\
(\Pi_A.B)[S] & \longrightarrow & \Pi_{A[S]}.B[\underline{1} \cdot_A (S \circ \uparrow^1)] & \uparrow^{n+1} \circ (M \cdot_A S) & \longrightarrow & \uparrow^n \circ S \\
(M\ N)[S] & \longrightarrow & (M[S]\ N[S]) & \uparrow^{n+1} \circ \uparrow^m & \longrightarrow & \uparrow^n \circ \uparrow^{m+1} \\
M[S][T] & \longrightarrow & M[S \circ T] & \underline{1} \cdot_A \uparrow^1 & \longrightarrow & \uparrow^0 \\
\underline{1}[M \cdot_A S] & \longrightarrow & M & \underline{1}[\uparrow^n] \cdot_A \uparrow^{n+1} & \longrightarrow & \uparrow^n \\
M[\uparrow^0] & \longrightarrow & M & s[S] & \longrightarrow & s
\end{array}
$$

Fig. 3. The $\lambda\Pi_{\mathcal{L}}$-rewrite system

Lemma 1. *The $\Pi_{\mathcal{L}}$-calculus is terminating.*

[2] The $\lambda\sigma$-calculus is not confluent on general open expressions [9]. However, it is confluent on semi-open expressions [23]. The $\lambda\sigma_{\Uparrow}$-calculus, a variant of $\lambda\sigma$, achieves confluence on general open expressions [9].

Proof. See [20]. The proof, due to Zantema, uses the semantic labeling technique [27]. $\square$

The set of $\Pi_{\mathcal{L}}$-normal forms of an expression x (term or substitution) is denoted by $(x)\downarrow_{\Pi_{\mathcal{L}}}$. The equivalence relation $\equiv_{\lambda\Pi_{\mathcal{L}}}$ (resp. $\equiv_{\Pi_{\mathcal{L}}}$) is defined as the congruence relation induced by the rewrite system $\lambda\Pi_{\mathcal{L}}$ (resp. $\Pi_{\mathcal{L}}$).

Typing contexts: Typing contexts in $\mathrm{PTS}_{\mathcal{L}}$ contain more information than in PTS. In order to combine declaration and definition of variables within a typing context, we associate a type *and a term* to the each variable declaration. Indeed, a *typing context* in $\mathrm{PTS}_{\mathcal{L}}$ is an explicit substitution having the form $M_1 \cdot_{A_1} \ldots \cdot M_n \cdot_{A_n} \uparrow^n$, where, for $0 < i \leq n$, M_i is either $\underline{i}$ or is equal to a term $N_i[\uparrow^i]$. When M_i is $\underline{i}$, we say that index i is *declared*; otherwise, we say that the index i is *defined*. Furthermore, M_i and A_i are called the *term definition* and *type declaration* of the index i, respectively. Note that not every explicit substitution denotes a typing context. We use the Greek letters Γ and Δ to range over explicit substitutions denoting typing contexts.

Free indices in term definitions and in type declarations obey different conventions. Let Γ be $M_1 \cdot_{A_1} \ldots \cdot M_n \cdot_{A_n} \uparrow^n$. Free indices in a term definition M_i are absolute, i.e., they refer to the whole context Γ. Cyclic *definitions* are avoided by construction. Notice that we require M_i to be equal to $N_i[\uparrow^i]$, for some term N_i. In that case, for all free indices j of M_i, it holds that $j > i$. On the other hand, free indices in a type declaration A_i are relative, i.e., they refer to the portion of the context where the index i is declared or defined. Therefore, by using this convention, cyclic *declarations* are impossible.

Although a different convention for the free indices in a typing context is still possible, we prefer the one sketched above since it allows an elegant encoding of contexts as explicit substitutions. The intention is to identify the evaluation of a term M in a context Γ as the $\Pi_{\mathcal{L}}$-reduction of $M[\Gamma]$. In particular, notice that a typing context without definitions has the form $\underline{1} \cdot_{A_1} \ldots \cdot \underline{n} \cdot_{A_n} \uparrow^n$. This substitution $\Pi_{\mathcal{L}}$-reduces to $\uparrow^0$. Therefore, as expected, the evaluation of a term M in a context Γ which does not contain definitions results in M.

The fact that indices are either *defined* or *declared* is rather a convenient way to explain typing contexts. The type system does not really distinguish both classes of indices.

Meta-variables and constraints: *Meta-variables* and *constraints* are used to deal with higher-order unification problems. Informally, meta-variables stand for instantiation variables and constraints are term equalities to be solved.

Meta-variables are first-class objects in $\mathrm{PTS}_{\mathcal{L}}$. Just as variables, they have to be declared in order to keep track of possible dependencies between terms and types. A *meta-variable declaration* has the form $X{:}_{\Gamma} A$, where Γ and A are, respectively, a context and a type assigned to the meta-variable X. Indices in A are relative to Γ.

A *constraint* $M \simeq_{\Gamma} N$ relates two terms M, N and a context Γ. Indices in M and N are relative to Γ. Similarly to meta-variables, constraints respect a typing discipline: terms M and N have the same type in Γ.

Definition 1 (Constrained signatures). *A* (constrained) signature *is a list containing meta-variable declarations and constraint declarations. An empty signature is denoted by ϵ. Furthermore, if Σ is a constrained signature, $X{:}_\Gamma A.\,\Sigma$ and $M \simeq_\Gamma N.\,\Sigma$ are well-formed constrained signatures.*

We overload the notation $\Sigma_1.\,\Sigma_2$ to write the concatenation of the signatures Σ_1 and Σ_2. A meta-variable X is *fresh* with respect to a signature Σ, denoted $X \notin \Sigma$, if there are not A and Γ such that $X{:}_\Gamma A$ is in Σ.

Equality: In order to deal with constraints, $\mathrm{PTS}_\mathcal{L}$ needs a finer notion of convertibility than that for PTS (Section 1).

Definition 2 (Equivalence modulo constraints). *Let Σ be a signature; we define the relation $\equiv_\Sigma$ as the smallest congruence such that (1) if $M \equiv_{\lambda\Pi_\mathcal{L}} N$, then $M \equiv_\Sigma N$, (2) if $M \simeq_\Gamma N \in \Sigma$, then $M[\Gamma] \equiv_\Sigma N[\Gamma]$, and (3) if $M \equiv_\Sigma N[\uparrow^1]$, then $\lambda_A.(M\ \underline{1}) \equiv_\Sigma N$.*

The last case of the definition above handles η-conversions. In this way, we avoid to consider an η-rule explicitly in the $\lambda\Pi_\mathcal{L}$-calculus.

We extend $\equiv_\Sigma$ to relate typing contexts as follows: (1) $\uparrow^n \equiv_\Sigma \uparrow^n$ and (2) $M \cdot_A \Gamma \equiv_\Sigma N \cdot_B \Delta$, if $M \equiv_\Sigma N$, $A[\Gamma] \equiv_\Sigma B[\Delta]$, and $\Gamma \equiv_\Sigma \Delta$.

$\mathrm{PTS}_\mathcal{L}$ typing rules: In PTS, typing rules for validity of typing contexts are implicit. However, in that case, structural rules (Start) and (Weak) are necessary to create an initial context and to add new variable declarations to it. The notation $\mid\!\sim \Sigma;\Gamma$ captures that Γ is valid in the valid signature Σ. We write $\Sigma;\Gamma \mid\!\sim M : A$ to state that the term M has type A in $\Sigma;\Gamma$. For explicit substitutions S we write $\Sigma;\Gamma \mid\!\sim S \triangleright \Delta$ to state that S has type Δ in $\Sigma;\Gamma$. The type system for $\mathrm{PTS}_\mathcal{L}$ is given in Figure 4. The judgments omits Σ, when it is ϵ, and Γ, when it is $\uparrow^0$. We reserve $\vdash$ for judgments where Σ does not contain constraints, otherwise we use $\mid\!\sim$. A signature Σ is *valid* if $\mid\!\sim \Sigma$ holds.

We use the following functions on contexts:

- $\Gamma \,\mathbf{o}\, \uparrow^i = \Delta$, where Δ is the normal form of $\Gamma \circ \uparrow^i$ with respect to the rewrite system composed by the rules (Map), (Ids), (ShiftCons), and (ShiftShift). We also have to consider the case when i is negative: if $\Gamma \equiv_{\Pi_\mathcal{L}} \Delta \,\mathbf{o}\, \uparrow^j$ and $j > i$, we consider $\Gamma \,\mathbf{o}\, \uparrow^{-i}$ equal to $\Delta \,\mathbf{o}\, \uparrow^{j-i}$.
- To add and remove elements from a context we use the shorthands $push(M, A, \Gamma) = M \cdot_A (\Gamma \,\mathbf{o}\, \uparrow^1)$, $top(M \cdot_A \Gamma) = A[\Gamma]$, and $pop(M \cdot_A \Gamma) = \Gamma \,\mathbf{o}\, \uparrow^{-1}$.
- The *order* of a substitution S indicates how many term definitions will be either consumed, if the number is negative, or produced, if the number is positive, when S is applied to a context Γ. Formally, $order(\uparrow^n) = -n$, $order(M \cdot_A S) = 1 + order(S)$, $order(S \circ T) = order(S) + order(T)$.
- Given a context Γ, the operation $\lfloor \Gamma \rfloor_0$ computes a new context where all the definitions in Γ have been transformed into declarations, as follows $\lfloor \uparrow^m \rfloor_n = \uparrow^n$, $\lfloor M \cdot_A \Gamma \rfloor_n = \underline{n+1} \cdot_A \lfloor \Gamma \rfloor_{n+1}$.

$$\frac{}{|\!\sim \epsilon; \uparrow^0}\;(\text{Empty}) \qquad \frac{\Sigma; \Gamma \,|\!\sim M_1 : A \quad \Sigma; \Gamma \,|\!\sim M_2 : A}{|\!\sim M_1 \simeq_\Gamma M_2.\ \Sigma; \Gamma}\;(\text{Constraint})$$

$$\frac{\Sigma; \Gamma \,|\!\sim A : s}{|\!\sim \Sigma; push(\underline{1}, A, \Gamma)}\;(\text{Var-Decl}) \qquad \frac{\Sigma; \Gamma \,|\!\sim M : A \quad \Sigma; \Gamma \,|\!\sim A : s}{|\!\sim \Sigma; push(M[\Gamma \circ \uparrow^1], A, \Gamma)}\;(\text{Let-Decl})$$

$$\frac{|\!\sim \Sigma; \Gamma \quad X \notin \Sigma}{|\!\sim X{:}_\Gamma\, s.\ \Sigma; \Gamma}\;(\text{Meta-Var-Decl}_1) \qquad \frac{\Sigma; \Gamma \,|\!\sim A : s \quad X \notin \Sigma}{|\!\sim X{:}_\Gamma\, A[\Gamma].\ \Sigma; \Gamma}\;(\text{Meta-Var-Decl}_2)$$

$$\frac{|\!\sim \Sigma; \Gamma \quad (s_1, s_2) \in \mathcal{A}}{\Sigma; \Gamma \,|\!\sim s_1 : s_2}\;(\text{Sort}) \qquad \frac{|\!\sim \Sigma; \Gamma}{\Sigma; \Gamma \,|\!\sim \underline{1} : top(\Gamma)}\;(\text{Var})$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \,|\!\sim A : s_1 \\ \Sigma; push(\underline{1}, A, \Gamma) \,|\!\sim B : s_2 \\ (s_1, s_2, s_3) \in \mathcal{R}\end{array}}{\Sigma; \Gamma \,|\!\sim \Pi_A.B : s_3}\;(\text{Prod}) \qquad \frac{\begin{array}{c}\Sigma; push(\underline{1}, A, \Gamma) \,|\!\sim M : B \\ \Sigma; \Gamma \,|\!\sim \Pi_A.B : s\end{array}}{\Sigma; \Gamma \,|\!\sim \lambda_A.M : \Pi_{A[\Gamma]}.B}\;(\text{Abs})$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \,|\!\sim M : \Pi_A.B \\ \Sigma; \Gamma \,|\!\sim N : A\end{array}}{\Sigma; \Gamma \,|\!\sim (M\ N) : B[N[\Gamma] \cdot_A \uparrow^0]}\;(\text{Appl})$$

$$\frac{\begin{array}{cc}|\!\sim \Sigma; \Gamma & X{:}_\Delta\, A \in \Sigma \\ \Sigma; \Delta \,|\!\sim A : s & \lfloor \Delta \rfloor_0 \equiv_\Sigma \lfloor \Gamma \rfloor_0\end{array}}{\Sigma; \Gamma \,|\!\sim X : A[\Gamma]}\;(\text{Meta-Var}) \qquad \frac{\begin{array}{cc}|\!\sim \Sigma; \Gamma & X{:}_\Delta\, s \in \Sigma \\ s \text{ is a top sort} & \lfloor \Delta \rfloor_0 \equiv_\Sigma \lfloor \Gamma \rfloor_0\end{array}}{\Sigma; \Gamma \,|\!\sim X : s}\;(\text{Meta-Var-Sort})$$

$$\frac{\begin{array}{cc}\Sigma; \Gamma \,|\!\sim S \rhd \Delta & \Sigma; \Delta \,|\!\sim M : A \\ \Sigma; \Delta \,|\!\sim A : s & n = order(S)\end{array}}{\Sigma; \Gamma \,|\!\sim M[S] : A[\uparrow^{-n}]}\;(\text{Clos}) \qquad \frac{\begin{array}{c}\Sigma; \Gamma \,|\!\sim S \rhd \Delta \\ \Sigma; \Delta \,|\!\sim M : s \quad s \text{ is a top sort}\end{array}}{\Sigma; \Gamma \,|\!\sim M[S] : s}\;(\text{Clos-Sort})$$

$$\frac{|\!\sim \Sigma; \Gamma}{\Sigma; \Gamma \,|\!\sim \uparrow^0 \rhd \Gamma}\;(\text{Id}) \qquad \frac{|\!\sim \Sigma; \Gamma \quad \Sigma; pop(\Gamma) \,|\!\sim \uparrow^n \rhd \Delta}{\Sigma; \Gamma \,|\!\sim \uparrow^{n+1} \rhd \Delta}\;(\text{Shift})$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \,|\!\sim S \rhd \Delta_1 \\ \Sigma; \Delta_1 \,|\!\sim T \rhd \Delta_2\end{array}}{\Sigma; \Gamma \,|\!\sim T \circ S \rhd \Delta_2}\;(\text{Comp}) \qquad \frac{\begin{array}{cc}\Sigma; \Gamma \,|\!\sim M : A[S] & \Sigma; \Gamma \,|\!\sim S \rhd \Delta \\ \Sigma; \Delta \,|\!\sim A : s & n = order(S)\end{array}}{\Sigma; \Gamma \,|\!\sim M \cdot_A S \rhd push(M[\Gamma \circ \uparrow^{n+1}], A, \Delta)}\;(\text{Cons})$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \,|\!\sim M : A \\ \Sigma; \Gamma \,|\!\sim B : s \\ A \equiv_\Sigma B[\Gamma] \equiv_{\lambda\Pi_\mathcal{L}} B\end{array}}{\Sigma; \Gamma \,|\!\sim M : B}\;(\text{Conv}) \qquad \frac{\begin{array}{c}\Sigma; \Gamma \,|\!\sim S \rhd \Delta_1 \\ |\!\sim \Sigma; \Delta_2 \\ \Delta_1 \equiv_\Sigma \Delta_2\end{array}}{\Sigma; \Gamma \,|\!\sim S \rhd \Delta_2}\;(\text{Conv-Subs})$$

Fig. 4. The $\lambda\Pi_\mathcal{L}$-type system

Relating $\text{PTS}_\mathcal{L}$ to PTS: Since $\text{PTS}_\mathcal{L}$ allows arbitrary constraints between terms, strong normalization, as well as other usual typing properties, can be easily violated in arbitrary $\text{PTS}_\mathcal{L}$. For instance, the term $(\lambda x{:}A.(x\ x)\ \lambda x{:}A.(x\ x))$ can be typed in any $\text{PTS}_\mathcal{L}$ containing the constraint $A \simeq_\Gamma A \to A$. However, as we will see below, $\text{PTS}_\mathcal{L}$ is a conservative extension of PTS. Furthermore, when only pure expressions are considered, i.e., signatures are empty (and then we use $\vdash$ rather than $|\!\sim$), $\text{PTS}_\mathcal{L}$ types as many terms as PTS (modulo $\Pi_\mathcal{L}$-reductions).

We say that a typing context is *pure* if it is the identity substitution or it has the form $\underline{1} \cdot_{A_1} \ldots \underline{n} \cdot_{A_n} \uparrow^n$ and A_i is pure for $1 \leq i \leq n$. Notice that pure contexts reduce to $\uparrow^0$ via application of rules (Shift0) and (Shift S).

Theorem 1 (Conservative extension). *Let M, A be pure terms and Γ be a pure context. Then, $\Gamma \vdash M : A$ in a PTS, if and only if $\Gamma \vdash M : A$ in $\text{PTS}_\mathcal{L}$ (we assume de Bruijn indices translation. For details of this translation see [9]).*

Proof. Both cases require typing properties for PTS and $\text{PTS}_{\mathcal{L}}$. We refer to [14] for the proofs of properties in PTS and to Section 4 for a summary of properties in $\text{PTS}_{\mathcal{L}}$. The case PTS $\Rightarrow$ $\text{PTS}_{\mathcal{L}}$, proceeds by induction on the typing derivation. Note that if $M \equiv_{\beta\eta} N$ then $M \equiv_{\Sigma} N$. For the case $\text{PTS}_{\mathcal{L}} \Rightarrow$ PTS, we first prove by induction on the typing derivation that for arbitrary terms N,B, and pure context Γ, $\Gamma \vdash N : B$ in $\text{PTS}_{\mathcal{L}}$ implies $\Gamma \vdash (N){\downarrow_{\Pi_{\mathcal{L}}}} : (B){\downarrow_{\Pi_{\mathcal{L}}}}$. □

3 Definitions and Explicit Instantiations

In this section, we address two novel aspects of $\lambda\Pi_{\mathcal{L}}$: the ability to encode let-in expressions and the support for expression contexts.

Definitions: Let-in expressions are a convenient way to support definitions in λ-calculus. In the simply-typed λ-calculus an expression as $\texttt{let}\ x := N : A\ \texttt{in}\ M$ can be encoded as $(\lambda x{:}A.M\ \ N)$. In this case, definition unfolding is performed by the β-reduction mechanism. The behavior of definitions in a dependent-type system cannot be straightforwardly encoded as a β-reduction. Consider, for example, the expression $\texttt{let}\ x := 0 : nat\ \texttt{in}\ (m\ (l\ x))$ in a context $\Gamma = m : (A\ 0) \to nat.\ 0 : nat.\ l : (\Pi n{:}nat.(A\ n)).\ A : nat \to \mathit{Type}.\ nat : \mathit{Type}.$[3] Although this term is unfolded into $(m\ (l\ 0))$ in the same way that the term $((\lambda x{:}nat.(m\ (l\ x)))\ 0)$ β-reduces to $(m\ (l\ 0))$, the term $((\lambda x{:}nat.(m\ (l\ x)))\ 0)$ cannot be typed in Γ. This is because the information that the variable x will be substituted by 0 in $(m\ (l\ x))$ is not taken into account by the application rule. Indeed, the type of $(l\ x)$ is $(A\ x)$, not $(A\ 0)$ as expected by m, and, therefore, the term $\lambda x{:}nat.(m\ (l\ x))$ is ill-typed in Γ. Solutions to the above problem require either to consider definitions as first-class terms (not just a macro expansion to a β-redex) [24] or to use a different notation and typing rules for applications [4]. In [3], Bloo proposes a calculus of explicit substitutions for PTS with contexts extended with term definitions.

As a side effect to combine definitions and declarations in contexts, let-in expressions can be encoded as explicit substitutions. In $\text{PTS}_{\mathcal{L}}$, as well as in Bloo's system, $\texttt{let}\ x := N : A\ \texttt{in}\ M$ is just a shorthand for the term $M[x := N : A]$ (or $M[N \cdot_A \uparrow^0]$ in our nameless notation). On the other hand, the typing rule for applications remains unmodified.

In contrast to Bloo's approach, we use a uniform notation for typing contexts and explicit substitutions. Furthermore, we internalize completely the substitution mechanism within the theory. In particular, we do not require any implicit substitution mechanism. Implicit substitutions are problematic when meta-variables are allowed. Notice that meta-variables, in a dependent-type theory, may also appear in typing contexts.

Explicit instantiations: In term rewriting, an *expression context* is an expression with distinguished holes. Filling a hole in a expression context with an

[3] For readability, we use named variables when discussing examples. Nevertheless, as we have said, $\text{PTS}_{\mathcal{L}}$ uses a de Bruijn nameless notation of variables.

arbitrary term is a first-order substitution that does not care of renaming variables. It is well-known that expression contexts in λ-calculus, and in general in higher-order rewrite systems, raise technical problems. For instance, contexts are not compatible with α and β-conversion. Calculus of contexts for simply-typed λ-calculus have been studied in for instance [5]. In previous approaches, either β-redexes cannot be reduced in context terms or new binding structures together with delicate type systems have been required to handle holes and explicit filling of holes.

Explicit substitutions overcome these difficulties in λ-calculus, and thus, they allow us to formulate $\mathrm{PTS}_{\mathcal{L}}$ as λ-calculus of contexts where meta-variables denote holes. To complete the framework, we must provide an explicit mechanism to perform instantiations.

Up till now, meta-variables were given in an abstract set $\mathcal{V}$. In order to remain in a first-order setting, we index meta-variables with positive natural numbers. Therefore, we write X_p for a meta-variable indexed by the positive natural number p. Expressions of a $\mathrm{PTS}_{\mathcal{L}}$-system with explicit instantiations include $M\{\theta\}$ and $S\{\theta\}$ as first-class terms and substitutions where θ is a set of instantiations. We use list of terms to represent a set of instantiations.

An instantiation θ is *well-formed* if it has the form $M_1 \cdot \ldots M_n \cdot m$, where $n \leq m$ and each term M_i is either X_i or a term which does not contain the meta-variable X_i. In the latter case, M_i is called the *instantiation term* of X_i in θ. The *grade* of θ, also written *grade*(θ), is given by $m - n$.

The structure of explicit instantiation is analogous to the structure of explicit substitutions. Indeed, an instantiation $M \cdot \theta$ denotes the replacement of meta-variables X_1 by M, X_2 by the head of θ and so forth. The lookup mechanism of meta-variables in an explicit instantiation is also analogous to the lookup of variables in an explicit substitution. The index p of the meta-variable X_p is consumed at the same time that the instantiation θ is traversed. The grade of θ helps to reconstruct the original index of the meta-variable in the cases of failed look-ups, i.e., the meta-variable X_p does not appear in θ.

In contrast to explicit substitutions, instantiations do not care about recalculation of free-indices in expressions. The calculus of explicit instantiations, called $\mathcal{X}_{\{\mathcal{L}\}}$, is depicted in Figure 5.

$$
\begin{array}{lcl}
s\{\theta\} & \longrightarrow & s \\
\underline{1}\{\theta\} & \longrightarrow & \underline{1} \\
(\Pi_A . B)\{\theta\} & \longrightarrow & \Pi_{A\{\theta\}} . B\{\theta\} \\
(\lambda_A . N)\{\theta\} & \longrightarrow & \lambda_{A\{\theta\}} . N\{\theta\} \\
(N_1\ N_2)\{\theta\} & \longrightarrow & (N_1\{\theta\}\ N_2\{\theta\}) \\
(N[S])\{\theta\} & \longrightarrow & N\{\theta\}[S\{\theta\}] \\
\uparrow^n\{\theta\} & \longrightarrow & \uparrow^n
\end{array}
\qquad
\begin{array}{lcl}
(N \cdot_A S)\{\theta\} & \longrightarrow & N\{\theta\} \cdot_{A\{\theta\}} S\{\theta\} \\
(S \circ T)\{\theta\} & \longrightarrow & S\{\theta\} \circ T\{\theta\} \\
X_p\{0\} & \longrightarrow & X_p \\
X_p\{n+1\} & \longrightarrow & X_{p+1}\{n\} \\
X_1\{M \cdot \theta\} & \longrightarrow & M \\
X_{p+1}\{M \cdot \theta\} & \longrightarrow & X_p\{\theta\}
\end{array}
$$

Fig. 5. The $\mathcal{X}_{\{\mathcal{L}\}}$-rewrite system

Lemma 2. *The $\mathcal{X}_{\{\mathcal{L}\}}$-rewrite system is confluent and terminating.*

Proof. To show that $\mathcal{X}_{\{\mathcal{L}\}}$ is terminating, we use a measure that penalizes top-level application of instantiations and the height of trees representing instantiations. Furthermore, $\mathcal{X}_{\{\mathcal{L}\}}$ is orthogonal, then it is confluent. $\square$

Typing rules for explicit instantiations are shown in Figure 6. We extend the relation $\equiv_\Sigma$ to consider the case of $\mathcal{X}_{\{\mathcal{L}\}}$-reductions as follows: if $M \xrightarrow{\mathcal{X}_{\{\mathcal{L}\}}} N$, then $M \equiv_\Sigma N$. We also have a new kind of typing judgment: $\Sigma; n \mathrel{|\!\sim} \theta \Rightarrow \Sigma'$. It has a double purpose. In first place, it enforces the invariant $n = \mathit{grade}(\theta)$. Secondly, it states that for all instantiation term M_i in θ, M_i is well-typed with respect to the meta-variable X_i. In that case, a constraint $X_i \simeq_\Gamma M_i$ is accumulated in a new signature Σ'.

$$\frac{\Sigma; 0 \mathrel{|\!\sim} \theta \Rightarrow \Sigma' \qquad \Sigma; \Gamma \mathrel{|\!\sim} M : A}{\Sigma'.\ \Sigma; \Gamma \mathrel{|\!\sim} M\{\theta\} : A}\ (\text{Inst-Term}) \qquad \frac{\Sigma; 0 \mathrel{|\!\sim} \theta \Rightarrow \Sigma' \qquad \Sigma; \Gamma \mathrel{|\!\sim} S \triangleright \Delta}{\Sigma'.\ \Sigma; \Gamma \mathrel{|\!\sim} S\{\theta\} \triangleright \Delta}\ (\text{Inst-Subs})$$

$$\frac{\Sigma; \mathit{grade}(\theta) \mathrel{|\!\sim} \theta \Rightarrow \Sigma' \qquad \Sigma; \Gamma \mathrel{|\!\sim} X_{\mathit{grade}(\theta)+p} : A}{\Sigma'.\ \Sigma; \Gamma \mathrel{|\!\sim} X_p\{\theta\} : A}\ (\text{Lookup}) \qquad \frac{\mathrel{|\!\sim} \Sigma}{\Sigma; n \mathrel{|\!\sim} n \Rightarrow \epsilon}\ (\text{Empty-Inst})$$

$$\frac{\Sigma; n+1 \mathrel{|\!\sim} \theta \Rightarrow \Sigma'}{\Sigma; n \mathrel{|\!\sim} X_{n+1} \cdot \theta \Rightarrow \Sigma'}\ (\text{List-Inst}_1) \qquad \frac{\Sigma; n+1 \mathrel{|\!\sim} \theta \Rightarrow \Sigma' \qquad \Sigma = \Sigma_1 .\ X_{n+1} :_\Gamma A.\ \Sigma_2 \qquad \Sigma_2; \Gamma \mathrel{|\!\sim} M : A}{\Sigma; n \mathrel{|\!\sim} M \cdot \theta \Rightarrow X_{n+1} \simeq_\Gamma M.\ \Sigma'}\ (\text{List-Inst}_2)$$

Fig. 6. Typing rules for explicit instantiations

We denote by $x\{\theta\}_\downarrow$, the $\mathcal{X}_{\{\mathcal{L}\}}$-normal form of an expression $x\{\theta\}$ where x is either a term or a substitution. We extend this notation to contexts and signatures as follows: $\epsilon\{\theta\}_\downarrow = \epsilon$, $(A.\Gamma)\{\theta\}_\downarrow = A\{\theta\}_\downarrow.\Gamma\{\theta\}_\downarrow$, $(X{:}_\Gamma A.\ \Sigma)\{\theta\}_\downarrow = X{:}_{\Gamma\{\theta\}_\downarrow} A\{\theta\}_\downarrow.\ \Sigma\{\theta\}_\downarrow$, and $(M \simeq_\Gamma N.\ \Sigma)\{\theta\}_\downarrow = M\{\theta\}_\downarrow \simeq_{\Gamma\{\theta\}_\downarrow} N\{\theta\}_\downarrow.\ \Sigma\{\theta\}_\downarrow$.

$\mathrm{PTS}_\mathcal{L}$ allows to represent open terms in PTS, i.e., terms with meta-variables. The *instantiation* mechanism of meta-variables fills the place-holders in open terms. This way, an open term in $\mathrm{PTS}_\mathcal{L}$ gradually becomes a pure term in PTS.

Lemma 3 (Instantiation). *Assume* $\Sigma; 0 \mathrel{|\!\sim} \theta \Rightarrow \Sigma'$. *(1) If* $\mathrel{|\!\sim} \Sigma; \Delta$, *then* $\mathrel{|\!\sim} \Sigma\{\theta\}_\downarrow; \Delta\{\theta\}_\downarrow$, *(2) if* $\Sigma; \Delta \mathrel{|\!\sim} N : B$, *then* $\Sigma\{\theta\}_\downarrow; \Delta\{\theta\}_\downarrow \mathrel{|\!\sim} N\{\theta\}_\downarrow : B\{\theta\}_\downarrow$, *and (3) if* $\Sigma; \Delta_1 \mathrel{|\!\sim} S \triangleright \Delta_2$, *then* $\Sigma\{\theta\}_\downarrow; \Delta_1\{\theta\}_\downarrow \mathrel{|\!\sim} S\{\theta\}_\downarrow \triangleright \Delta_2\{\theta\}_\downarrow$.

Proof. First, we prove by simultaneous induction on terms and substitutions that $\mathcal{X}_{\{\mathcal{L}\}}$-reductions preserve typing, i.e. if $\Sigma; \Gamma \mathrel{|\!\sim} M : A$ and $M \xrightarrow{\mathcal{X}_{\{\mathcal{L}\}}} N$, then $\Sigma; \Gamma \mathrel{|\!\sim} N : A$, and if $\Sigma; \Gamma \mathrel{|\!\sim} S \triangleright \Delta$ and $S \xrightarrow{\mathcal{X}_{\{\mathcal{L}\}}} T$, then $\Sigma; \Gamma \mathrel{|\!\sim} S \triangleright \Delta$. Hence, we can prove that (1) $\mathrel{|\!\sim} \Sigma'.\ \Sigma\{\theta\}_\downarrow; \Delta\{\theta\}_\downarrow$, (2) $\Sigma'.\ \Sigma\{\theta\}_\downarrow; \Delta\{\theta\}_\downarrow \mathrel{|\!\sim} N\{\theta\}_\downarrow : B\{\theta\}_\downarrow$, and (3) $\Sigma'.\ \Sigma\{\theta\}_\downarrow; \Delta_1\{\theta\}_\downarrow \mathrel{|\!\sim} S\{\theta\}_\downarrow \triangleright \Delta_2\{\theta\}_\downarrow$. Finally, we use a weakening property on signatures which states that constraints of the form $X_p \simeq_\Gamma M$ can be safely removed from a signature when the meta-variable X_p does not appear as a sub-term in the remaining signature. □

4 Meta-theory

We here summarize a meta-theoretical study for dependent-type versions for $\lambda\Pi_\mathcal{L}$. It extends [20] by considering explicit definitions and instantiations. Proofs are available at [21].

First of all, we prove some properties about contexts. The first explains the meaning of the typing rules for substitutions. The second states that terms in valid contexts always are in normal form with respect to term definitions. The third property states that the order of a valid contexts is 0. The last one states that contexts type themselves.

Lemma 4 (Properties about valid contexts). *(1) If $\Sigma;\Gamma \mathrel{|\!\!\sim} S \triangleright \Delta$, then $S \circ \Gamma \equiv_\Sigma \Delta\, \mathbf{o} \uparrow^{-order(S)}$. (2) If $\mathrel{|\!\!\sim} \Sigma;\Gamma$, then $\Gamma \circ \Gamma \equiv_{\Pi_\mathcal{L}} \Gamma$. (3) If $\mathrel{|\!\!\sim} \Sigma;\Gamma$, then $order(\Gamma) = 0$. (4) If $\mathrel{|\!\!\sim} \Sigma;\Gamma$, then $\Sigma;\Gamma \mathrel{|\!\!\sim} \Gamma \triangleright \Gamma$.*

Proof. All the proofs are by induction on the typing derivations[21]. □

We cannot go further with usual properties as *subject reduction*, *confluence*, or *normalization*. As we have mentioned before, when arbitrary constraints are allowed, these properties can be easily violated. To continue our meta-theoretical study we make some assumptions. First of all, we consider only functional $\mathrm{PTS}_\mathcal{L}$. Hence, we include all the type-systems of the Barendregt's cube. Furthermore, we consider signatures without constraints. Then we use $\vdash$ rather that $\mathrel{|\!\!\sim}$ to denote typing judgments.

Even with the assumptions above, the system $\lambda\Pi_\mathcal{L}$ is not confluent on pseudo-expressions, i.e., on expressions that are not well-typed. To handle this problem, we follow the development in [20]. The approach is originally due to Geuvers [13]. It exploits the fact that the rewrite-system $\lambda\Pi_\mathcal{L}$ without type annotations is confluent. We prove the following key lemma.

Lemma 5 (Geuvers' lemma). *Given a signature Σ without constraints, (1) if $\Pi_{A_1}.B_1 \equiv_\Sigma \Pi_{A_2}.B_2$, then $A_1 \equiv_\Sigma A_2$ and $B_1 \equiv_\Sigma B_2$, and (2) if $(c\ M_1 \ldots M_n) \equiv_\Sigma N$, then $N \xrightarrow{\lambda\Pi_\mathcal{L}} (c\ N_1 \ldots N_n)$ where $M_i \equiv_\Sigma N_i$.*

Proof. First, we prove that $\Pi_\mathcal{L}$ is confluent on expressions without type annotations on substitutions. Notice that η is not a rewrite-rule and that only meta-variables on the sort of terms are allowed. Then, we show that $\Pi_\mathcal{L}$commutes with the parallelization of (Beta) [20]. The conclusion follows from a positive use of the counter-example for the confluence of the system with type annotations [13, 20]. □

Geuvers' lemma is enough to prove the following properties. Most of them require simultaneous induction on terms and substitutions.

Sort soundness If $\Sigma;\Gamma \vdash M : A$, then either $A = s$, where s is a top sort, or $\Sigma;\Gamma \vdash A : s$, where $s \in \mathcal{S}$.

Type uniqueness For a functional $\mathrm{PTS}_\mathcal{L}$, if $\Sigma;\Gamma \vdash M : A$ and $\Sigma;\Gamma \vdash M : B$, then $A \equiv_\Sigma B$.

Subject reduction For a functional $\mathrm{PTS}_\mathcal{L}$, if $M \xrightarrow{\lambda\Pi_\mathcal{L}{}^*} N$ and $\Sigma;\Gamma \vdash M : A$, then $\Sigma;\Gamma \vdash N : A$.

Church-Rosser For a functional weakly normalizing $\mathrm{PTS}_\mathcal{L}$, if $M_1 \equiv_{\lambda\Pi_\mathcal{L}} M_2$, $\Sigma;\Gamma \vdash M_1 : A$, and $\Sigma;\Gamma \vdash M_2 : A$, then M_1 and M_2 are $\lambda\Pi_\mathcal{L}$-joinable, i.e., there exists M such that $M_1 \xrightarrow{\lambda\Pi_\mathcal{L}{}^*} M$ and $M_2 \xrightarrow{\lambda\Pi_\mathcal{L}{}^*} M$.

It is well-known that explicit substitutions calculi based on $\lambda\sigma$ do not satisfy the strong-normalization property, not even in the simply-typed theory [19]. In [20], it is shown that $\lambda\Pi_{\mathcal{L}}$ without constraints and without η-equality is weakly normalizing on the Calculus of Constructions.

5 Applications

The PTS$_{\mathcal{L}}$-type system cannot be applied bottom up because some of the rules require premises that are not present in the conclusion, and others require to check equality under $\equiv_{\Sigma}$. On the other hand, it is possible to divide type checking into three sub-tasks that can be treated separately: (1) type inference, (2) checking (ground) equality, and (3) unification. This section therefore develops the relevant machinery for accomplishing these tasks.

Type inference: We obtain a type inference system from $\equiv_{\Sigma}$ by converting the judgments $\Sigma; \Gamma \mathrel{|\!\!\sim} M : A$ and $\Sigma; \Gamma \mathrel{|\!\!\sim} S \triangleright \Delta$ into $\Sigma; \Gamma \mathrel{|\!\!\sim}_{\mathcal{M}} M \implies A,\ \Sigma'$, respectively, $\Sigma; \Gamma \mathrel{|\!\!\sim}_{\mathcal{M}} S \implies \Delta,\ \Sigma'$, where the type A, context Δ, and auxiliary constraints Σ' are produced by applying the type inference rules. For lack of space we will not give the full inference system here, but illustrate some of the more interesting cases: the rule to infer the type of an application. The full system appears in [21].

$$\frac{\begin{array}{c}\Sigma; \Gamma \mathrel{|\!\!\sim}_{\mathcal{M}} M \implies A_1,\ \Sigma_1 \quad \Sigma_1; \Gamma \mathrel{|\!\!\sim}_{\mathcal{M}} N \implies A_2,\ \Sigma_2 \\ \Sigma_2; \Gamma \mathrel{|\!\!\sim}_{\mathcal{M}} A_1 : \mathcal{S} \implies s_3,\ \Sigma_3 \quad \Sigma_3; \Gamma \mathrel{|\!\!\sim}_{\mathcal{M}} A_2 : \mathcal{S} \implies s_1,\ \Sigma_4 \\ H \notin \Sigma_4 \quad (s_1, s_2, s_3) \in \mathcal{R} \quad \Delta = push(1, A_2, \Gamma) \\ \Sigma_5 = A_1 \simeq_{\Gamma} \Pi_{A_2}.H.\ H :_{\Delta} s_2.\ \Sigma_4\end{array}}{\Sigma; \Gamma \mathrel{|\!\!\sim}_{\mathcal{M}} (M\ N) \implies H[N \cdot_{A_2} \uparrow^0],\ \Sigma_5}$$

These rules can be encoded directly with Braga's conditional rewrite rule extension to Maude [7].

Checking equality: The rewrite-system $\lambda\Pi_{\mathcal{L}}$ provides a way to check equality under β-reduction, but not η equality. One could add an η-rule as a conditional rewrite rule $\lambda_A.(M\ \underline{1}) \longrightarrow N$ if $M \equiv_{\lambda\Pi_{\mathcal{L}}} N[\uparrow^1]$, and check equality of terms by first applying reduction to normal form. A more economical approach is to interleave weak head normal form conversion with rules that decompose equalities into smaller equalities. By weak head normal form we here take the minimal reduction under $\lambda\Pi_{\mathcal{L}}$-system that allows to decompose an equality using the other rules in Fig. 7. The cases not mentioned there are assumed as failures.

Our rules are directly structural on the shape of terms. We should note that the $\lambda - N$ rule realizes η-equality, and that the $\lambda - \lambda$ rule does not check equality of types because we assume that type constraints on the equated terms are checked by other constraints. The alternative notion of *algorithmic equality* studied by Harper and Pfenning [15] recurses on the structure of types in the context of LF. It is so far unclear how this idea adapts to type systems with impredicative type polymorphism.

whnf	$M \simeq_{\Gamma} N$	$\Longrightarrow$	$\mathrm{whnf}(M) \simeq_{\Gamma} \mathrm{whnf}(N)$
$\lambda - \lambda$	$\lambda_A.M \simeq_{\Gamma} \lambda_B.N$	$\Longrightarrow$	$M \simeq_{push(\underline{1},A,\Gamma)} N$
$\Pi - \Pi$	$\Pi_{A_1}.B_1 \simeq_{\Gamma} \Pi_{A_2}.B_2$	$\Longrightarrow$	$A_1 \simeq_{\Gamma} A_2.\ B_1 \simeq_{push(\underline{1},A_1,\Gamma)} B_2$
$\lambda - N$	$\lambda_A.M \simeq_{\Gamma} N$	$\Longrightarrow$	$M \simeq_{push(\underline{1},A,\Gamma)} (N[\uparrow^1]\ \underline{1})$
Subst	$M \simeq_{\Gamma} N$	$\Longrightarrow$	$M[\Gamma] \simeq_{\Gamma} N[\Gamma]$
$\mathcal{S} - \mathcal{S}$	$s \simeq_{\Gamma} s$	$\Longrightarrow$	ϵ
App-App	$(\underline{n}\ M_1\ \ldots\ M_k) \simeq_{\Gamma} (\underline{n}\ N_1\ \ldots\ N_k)$	$\Longrightarrow$	$M_1 \simeq_{\Gamma} N_1. \ldots M_k \simeq_{\Gamma} N_k$

Fig. 7. Rules for checking term equality

Unification and Inhabitation: With the available machinery, the inhabitation and unification problems for pure type systems is now a relatively straightforward adaption from [20, 10], and for the case of higher-order pattern unification from [11]. The full paper [21] contains a detailed treatment, but here we highlight the main ideas. As usual in higher-order unification, we distinguish between the cases of rigid-rigid, flex-rigid, and flex-flex pairs.

Rigid-Rigid The rules from Fig. 7 can be directly adapted to decompose rigid-rigid pairs.

Flex-Rigid To solve flex-rigid pairs we need in general to generate terms with de Brujin indices applied to meta-variables. To do this in a way that preserves types we here introduce an auxiliary operation χ_i that produces an application to i fresh meta-variables and calculates its type.

$$\frac{\Sigma; \Gamma \mid\!\sim M \implies A,\ \Sigma_1}{\Sigma; \Gamma \mid\!\sim_{\chi^0} M \implies M : A,\ \Sigma_1} \qquad \frac{\begin{array}{c}\Sigma; \Gamma \mid\!\sim_{\chi^i} M \implies N : A,\ \Sigma_1 \quad \Sigma_1; \Gamma \mid\!\sim A : \mathcal{S} \implies s_2,\ \Sigma_2 \\ X_{i+1}, Y_1, Y_2 \notin \Sigma_2 \quad s_1 \in \mathcal{S} \quad (s_1, s_2) \in \mathcal{A} \quad \Delta = push(\underline{1}, Y_1, \Gamma) \\ X_{i+1} :_{\Gamma} Y_1.\ Y_2 :_{\Delta} s_2.\ Y_1 :_{\Gamma} s_1.\ \Sigma_2; \Gamma \mid\!\sim \{A \simeq \Pi_{Y_1}.Y_2\} \implies \Sigma_3\end{array}}{\Sigma; \Gamma \mid\!\sim_{\chi^{i+1}} M \implies (N\ X_{i+1}) : Y_2[X_{i+1} \cdot_{Y_1} \uparrow^0],\ \Sigma_3}$$

A useful property about these judgments is that if $\Sigma; \Gamma \mid\!\sim_{\chi^i} M$ fails (produces an unsatisfiable constraint), then $\Sigma; \Gamma \mid\!\sim_{\chi^j} M$ fails for every $j \geq i$. This restricts unbounded branching in the cases where we do not make direct use of impredicative type abstraction. For example, in LF one computes the necessary i by inspecting the number of Π nested in A.

To prepare elimination of a meta-variable in a flex-rigid pair by instantiation, we furthermore must order meta-variable declarations in signatures according to their dependencies. For this purpose we introduce an operation to permute signatures. Permutations respect type derivation in $\mathrm{PTS}_{\mathcal{L}}$.

As the final preparation to the flex-rigid rules, we replace occurrences of $(X[S]\ M)$ in Σ by $((\lambda_{A_1}.Y)[S]\ M)$, for a fresh meta-variable $Y :_{push(1,A_1,\Gamma)} A_2$, when $X :_{\Gamma} \Pi_{A_1}.A_2 \in \Sigma$. This produces flex-rigid pairs, that we can finally reduce as follows:

$$\Sigma \implies \Sigma_3\{N/X\}_{\downarrow} \ \text{ if } \begin{cases} (X[M_1 \cdot_{A_1} \ldots M_p \cdot_{A_p} \uparrow^n] \simeq_{\Gamma} (\underline{m}\ N_1\ \ldots\ N_q)) \in \Sigma, \quad X :_{\Delta} B \in \Sigma \\ h \in \{1, \ldots, p\} \cup (\textbf{if } m > n \textbf{ then } \{m - n + p\} \textbf{ else } \emptyset) \\ i \geq 0 \quad \Sigma; \Gamma \mid\!\sim_{\chi^i} \underline{h} \implies N : A,\ \Sigma_1 \\ \Sigma_1; \Gamma \mid\!\sim \{A \simeq B\} \implies \Sigma_2 \quad permute(\Sigma_2) \implies \Sigma_4.\ \Delta :_X B.\ \Sigma_3 \end{cases}$$

Flex-Flex When all rigid-rigid and flex-rigid pairs are solved, we are left with flex-flex pairs of the form $X[M_1 \cdot_{A_1} \ldots M_p \cdot_{A_p} \uparrow^n] \simeq_\Gamma Y[N_1 \cdot_{B_1} \ldots N_q \cdot_{B_q} \uparrow^m]$. For an arbitrary PTS these equations are solved by enumerating inhabitants for X and Y and checking equality using the other rules. The main tool for enumerating the inhabitants is χ_i as exemplified for the calculus of constructions in [20]. This is unlike the situation when one of the substitutions is a *pattern* substitution [11], or if the types of X and Y are *simple* [16].

6 Related Work and Conclusion

Dependent-type versions of $\lambda\Pi_{\mathcal{L}}$ have been studied in [20]. The definition of $\mathrm{PTS}_{\mathcal{L}}$ that we present here generalizes those versions in several ways. First of all, we consider $\beta\eta$-equality instead of just β-equality. Secondly, we also identify contexts with a particular kind of explicit substitutions. Thus, let-expressions can be easily encoded in $\mathrm{PTS}_{\mathcal{L}}$. Finally, we advocate for explicit instantiations as a way to obtain a calculus of contexts.

Pure type systems and explicit substitutions have been studied in [25, 3]. A polymorphic calculus with explicit substitutions has been proposed in [6]. In contrast to our work, in those approaches explicit substitutions are not first-class objects and atomic implicit substitutions are required by the type systems. In [18], Magnusson uses meta-variables and explicit substitutions to represent incomplete proof-terms in the Martin Löf Type Theory, but a complete meta-theoretical study of the system is missing. More recently, Strecker has developed in [26] a complete meta-theory for a variant of the Extended Calculus of Constructions [17] that supports meta-variables. Strecker's system provides an explicit notation for substitutions. However, it cannot be really considered as an explicit substitution *calculus*. Indeed, in Strecker's approach, the substitution mechanism is implemented by means of an external substitution operation, and the explicit notation is just used to suspend the application of external substitutions to meta-variables. That is, explicit substitutions are only allowed when they are attached to meta-variables. The $\mathrm{PTS}_{\mathcal{L}}$-system completely internalizes the substitution mechanism into the theory. This theoretical treatment of substitutions allows a finer and granular control on the application of substitutions. In particular, Strecker's definition of β corresponds to a strategy in $\lambda\Pi_{\mathcal{L}}$ where a (Beta)-rule if followed by $\Pi_{\mathcal{L}}$-normalization. Unification algorithms for dependent types have been studied previously in [22, 12], and for the case of explicit substitutions in [10]. In comparison to Agda [8] our calculus erases the distinction between standard term and term with an explicit substitution.

Acknowledgments The authors would like to thank C. Braga for his help with Maude and to the anonymous referees for useful remarks.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitution. *Journal of Functional Programming*, 1(4):375–416, 1991.

2. H. Barendregt. Lambda calculi with types. In Gabbai, Abramski, and Maibaum, editors, *Handbook of Logic in Comp. Sci.* Oxford Univ. Press, 1992.
3. R. Bloo. *Preservation of Termination for Explicit Substitution.* PhD thesis, Eindhoven University of Technology, 1997.
4. R. Bloo, F. Kamareddine, and R. Nederpelt. The Barendregt cube with definitions and generalised reduction. *Inf. and Computation*, 126(2):123–143, 1 May 1996.
5. M. Bognar and Roel de Vrijer. The context calculus λ-c. In *LFM'99: Proceedings of Workshop on Logical Frameworks and Meta-languages*, 1999.
6. E. Bonelli. The polymorphic λ-calculus with explicit substitutions. In *Proc. of 2nd Intl. Workshop on Explicit Substitutions*, 1999.
7. C. Braga, H. Haeusler, J. Meseguer, and P. Mosses. Maude action tool: Using reflection to map action semantics to rewriting logic. Submitted, 2000.
8. C. Coquand and T. Coquand. Structured Type Theory. In *Proc. of Workshop on Logical Frameworks and Meta-Languages*, September 1999.
9. P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, March 1996.
10. G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. In *Proc. of LICS*, pages 366–374, 1995.
11. G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In *Proc. of LICS*, 1996.
12. C. Elliott. Higher-order unification with dependent types. In N. Dershowitz, editor, *Proc. of RTA*, volume 355 of *LNCS*, pages 121–136. Spr.-Verlag, 1989.
13. H. Geuvers. The Church-Rosser property for $\beta\eta$-reduction in typed λ-calculi. In *Proc. of LICS*, pages 453–460, 1992.
14. H. Geuvers. *Logics and Type Systems.* PhD thesis, Kath. Univ. Nijmegen, 1993.
15. R. Harper and F. Pfenning. On Equivalence and Canonical Forms in the LF Type Theory. In *LFM'99: Proc. of Workshop on Logical Frameworks and Meta-languages*, 1999.
16. G. Huet. A unification algorithm for typed λ-calculus. *Th. C. S.*, 1(1):27–57, 1975.
17. Z. Luo. *An Extended Calculus of Constructions.* Phd thesis CST-65-90, Lab. for Foundations of C. S., Dept. of C. S., Univ. of Edinburgh, July 1990.
18. L. Magnusson. *The Implementation of ALF—A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution.* PhD thesis, Chalmers and Göteborg Univ., 1995.
19. P. A. Mellies. Typed λ-calculi with explicit substitutions may not terminate. *LNCS*, 902:328–338, 1995.
20. C. Muñoz. *Un calcul de substitutions pour la représentation de preuves partielles en théorie de types.* Thèse de doctorat, Université Paris VII, 1997.
21. C. Muñoz and N. Bjørner. Absolute explicit unification. Manuscript. Available from http://www.icase.edu/ munoz/, July 2000.
22. F. Pfenning. Unification and anti-unification in the calculus of constructions. In *Proc. of LICS*, pages 74–85, 1991.
23. A. Ríos. *Contributions à l'étude de λ-calculs avec des substitutions explicites.* Thèse de doctorat, Université Paris VII, 1993.
24. P. Severi and E. Poll. Pure type systems with defn's. *LNCS*, 813:316–330, 1994.
25. M.-O. Stehr and J. Meseguer. Pure type systems in rewriting logic. Proceedings of LFM'99: Workshop on Logical Frameworks and Meta-languages, sept 1999.
26. M. Strecker. *Construction and Deduction in Type Theories.* PhD thesis, Univ. Ulm, 1999.
27. H. Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995.