
ELSEVIER

Theoretical Computer Science 266 (2001) 407–440

Theoretical Computer Science

www.elsevier.com/locate/tcs

Proof-term synthesis on dependent-type systems via explicit substitutions

César Muñoz [1]

Institute for Computer Applications in Science and Engineering, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681-2199, USA

Received August 1998; revised January 2000; accepted April 2000
Communicated by P.-L. Curien

Abstract

Typed λ-terms are used as a compact and linear representation of proofs in intuitionistic logic. This is possible since the Curry–Howard isomorphism relates proof-trees with typed λ-terms. The proofs-as-terms principle can be used to verify the validity of a proof by type checking the λ-term extracted from the complete proof-tree. In this paper we present a proof synthesis method for dependent-type systems where typed open terms are built incrementally at the same time as proofs are done. This way, every construction step, not just the last one, may be type checked. The method is based on a suitable calculus where substitutions as well as meta-variables are first-class objects.

Keywords: Proof synthesis; Higher-order unification; Explicit substitutions; Lambda calculus

1. Introduction

Thanks to the proofs-as-terms paradigm, a method of *proof* synthesis consists in finding a *term* of a given type. Since the set of λ-terms is enumerable, a complete method of proof synthesis in a framework where type checking is decidable consists in enumerating and type checking all the terms. Of course, this method is impractical for implementations. A smart enumeration of terms must take typing information and properties of the λ-calculus into account. In [38], Zaionc presents an algorithm for

[1] This research was supported by INRIA – Rocquencourt while the author was a research assistant at the INRIA Institute, by National Science Foundation grant CCR-9712383 while he was an international fellow at SRI International, and by the National Aeronautics and Space Administration under NASA Contract NAS1-97046 while he was in residence at ICASE.

E-mail address: munoz@icase.edu (C. Muñoz).

0304-3975/01/$ - see front matter
PII: S0304-3975(00)00196-1

proof construction in the propositional intuitionistic and classical logics via the simply typed λ-calculus, and Dowek shows in [12, 13] a complete term enumeration algorithm for the type systems of the Barendregt's cube.

Although the Curry–Howard isomorphism relates proofs with terms, proof construction and term synthesis do not necessarily go in the same direction. A natural deduction proof, for example, is driven by a bottom-up procedure, while term synthesis procedures go in a top-down manner. For instance, to prove a proposition B by *modus-ponens*, we assume $A \to B$ and A as hypotheses, and then we continue recursively trying to prove these two propositions. Eventually, we will get the axioms and the proof is finished. In contrast, to synthesize a term of type B, we start with the axioms to set up the variables, and then go down to the conclusion where the final term has the form $(M\ N)$ with M a term of type $A \to B$ and N a term of type A.

These two different construction mechanisms, bottom-up proof construction and top-down term synthesis, coexist in some theorem provers based on the proof-as-term paradigm. For example, in the proof assistant system Coq [3] proofs under construction, also called *incomplete proofs*, are represented as proof-trees. When the proof is done, a λ-term, that is, a *complete proof-term*, is synthesized. The soundness of the system relies on the type checker, which is a very small piece of code. However, if something goes wrong with the proof-tree construction, for example because a procedure manipulating a proof-tree is bugged, the problem is detected when the type checking of the *complete* proof-term takes place. That means, at the very last step of the proof-term synthesis.

A uniform representation of complete and incomplete proofs allows to identify the proof construction and term synthesis mechanisms. Furthermore, if such a representation supports an effective type-checking procedure, type inconsistencies can be detected during the whole process of the proof-term construction. In [28], Magnusson proposes an extension to the λ-calculus with place-holders and explicit substitutions to represent incomplete proofs. Her ideas were implemented in the theorem prover Alf [2], but a complete meta-theoretical study of the system and its properties is missing.

A term with place-holders is called an *open term*. Since several place-holders can appear in an open term, it is convenient to name them. In the λ-calculus with de Bruijn indices, named place-holders are just variables of the free-algebra of terms. In order to distinguish place-holders from variables of the λ-calculus, the former are called *meta-variables*. As a convention in this paper, meta-variables are written with the last uppercase letters of the alphabet: $X, Y, \ldots$.

The open term $\lambda x{:}A.Y$, can be seen as a proof-term of $A \to B$ provided that there exists a term of type B in the right context to replace Y. By using this replacement mechanism, also called *instantiation*, an incomplete proof becomes a complete one. In contrast to substitution of variables in the λ-calculus, instantiation of meta-variables is a first-order substitution that does not care about capture of variables. In the previous example the *instantiation* of Y with x results in the term $\lambda x{:}A.x$, while the *substitution* of x for Y in $\lambda x{:}A.Y$ results in $\lambda z{:}A.x$. Notice that unless A and B represent the same type, the resulting terms in both cases may be ill-typed.

As pointed out in [28, 15], open terms in the λ-calculus reveal new challenges. Assume, for example, that an open term is involved in a β-redex. The β-rule can create substitutions applied to meta-variables that cannot be effective while the meta-variables are not instantiated. In this case, a notation for suspended substitutions should be provided. Since the $\lambda\sigma$-calculus of explicit substitutions was introduced in [1], several other variants of explicit substitutions calculi have been proposed; among others [1, 36, 26, 23, 6, 27, 11, 24, 30, 18, 32]. The study of explicit substitution calculi showed up to be more complicated than that of the λ-calculus. For some of the explicit substitution calculi, questions about confluence, normalization and type checking are still open.

In [31, 33], we propose a variant of $\lambda\sigma$, called $\lambda\Pi_{\mathscr{L}}$, for dependent-type theories like λP [20] and the Calculus of Constructions [8, 9]. The $\lambda\Pi_{\mathscr{L}}$-calculus is confluent and weakly normalizing on well-typed expressions. The $\lambda\Pi_{\mathscr{L}}$-system does not enjoy confluence on the full set of open expressions, that is, $\lambda\Pi_{\mathscr{L}}$ is no longer confluent when meta-variables on the sort of substitutions are considered, and it does not preserve strong normalization, that is, arbitrary reductions on well-typed expressions may not terminate. However, we claim in this paper that the $\lambda\Pi_{\mathscr{L}}$-calculus is suitable as a framework to represent incomplete proof-terms in a constructive logic.

In this paper, we describe a proof-term synthesis method for λP and the Calculus of Construction via the $\lambda\Pi_{\mathscr{L}}$-calculus. The method uses the incomplete proof-term paradigm proposed in [33]. It is strongly inspired by that proposed by Dowek in [12, 13] for the Cube of Type Systems. In contrast to Dowek's method, our method combines both the bottom-up approach for proof construction, and the top-down synthesis of terms. In other words, proof-terms are synthesized at the same time that proofs are constructed. Since type checking is decidable in $\lambda\Pi_{\mathscr{L}}$, the soundness of the proof construction can be guaranteed step by step. From a practical point of view, implementation errors in procedures manipulating incomplete-proofs are detected by the type checker at any moment during the proof-construction process. The type checker of $\lambda\Pi_{\mathscr{L}}$ is still simple. In fact, we have implemented it, in the object-oriented functional language OCaml, in about 50 lines. We have also implemented a higher-order unification algorithm for ground expressions. The soundness of the whole implementation relies in the small piece of code corresponding to the type checker.

The rest of this section gives an overview to the dependent-type systems in which we are interested, the λP-calculus and the Calculus of Constructions, and to the $\lambda\sigma$-calculus of explicit substitutions. For a more comprehensive explanation on both subjects, we refer to [20, 9, 1]. In Section 2, we present the $\lambda\Pi_{\mathscr{L}}$-calculus and its dependent-type systems. In Section 3, we describe our method of proof synthesis. The soundness and completeness of the method are proved in Section 4. The last section presents related work and summarizes this work.

1.1. Dependent-type systems

The Dependent Type theory, namely λP [20], is a conservative extension of the simply-typed λ-calculus. It allows a finer stratification of terms by generalizing the

function space type. In fact, in λP, the type of a function $\lambda x{:}A.M$ is $\Pi x{:}A.B$ where B (the type of M) may depend on x. Hence, the type $A \to B$ of the simply typed λ-calculus is just a notation in λP for the product $\Pi x{:}A.B$ where x does not appear free in B.

From a logical point of view, the λP-calculus allows representation of proofs in the first-order intuitionistic logic using universal quantification. Via the types-as-proofs principle, a term of type $\Pi x{:}A.B$ is a proof-term of the proposition $\forall x{:}A.B$.

Terms in λP can be variables: $x, y, \ldots$, applications: $(M\ N)$, abstractions: $\lambda x{:}A.M$, products: $\Pi x{:}A.B$, or one of the sorts: $Type, Kind$.[2] Notice that terms and types belong to the same syntactical category. In this paper, for readability, we use the uppercase letters $A, B, \ldots$ to denote type terms, that is, terms of type (kind) $Type$ or $Kind$, and $M, N, \ldots$ to denote object terms, that is, terms of type A where A is a type term.

The type $\Pi x{:}A.B$ is a term, as well as the object $\lambda x{:}A.M$. However, terms are stratified in several levels according to a type discipline. For instance, given an appropriate context of variable declarations, $\lambda x{:}A..M : \Pi x{:}A..B$, $\Pi x{:}A..B : Type$, and $Type : Kind$. The term $Kind$ cannot be typed in any context, but it is necessary since a circular typing as $Type : Type$ leads to the Girard's paradox [19].

Typing judgments in λP have the form

$$\Gamma \vdash M : A$$

where Γ is a *context* of variable declarations, that is, a set of type assignments for free variables. We use the Greek letters Γ, Δ to range over contexts. Since types may be ill-typed, typing judgments for contexts are also necessary. The notation

$$\vdash \Gamma$$

captures that types in Γ are well-typed. The λP-type system is given in Fig. 1.

The Calculus of Constructions [8, 9] extends the λP-calculus with polymorphism and constructions of types. It is obtained by replacing rules (Prod) and (Abs) as shown in Fig. 2.

In a higher-order logic, as λP or the Calculus of Constructions, it may happen that two types syntactically different are the same module β-conversion. Rule (Conv) uses the equivalence relation $\equiv_\beta$ which is defined as the reflexive and transitive closure of the relation induced by the β-rule:

$(\lambda x{:}A.M\ N) \longrightarrow M[N/x]$. We recall that $M[N/x]$ is just a notation for the atomic substitution of the free occurrences of x in M by N, with renaming of bound variables in M when necessary.

1.2. Explicit substitutions

The $\lambda\sigma$-calculus [1] is a first-order rewrite system with two sorts of expressions: *terms* and *substitutions*. Well-formed expressions in the $\lambda\sigma$-calculus are defined by the

[2] The names $Type$ and $Kind$ are not standard, other couples of names used in the literature are: $(Set, Type)$, $(Prop, Type)$ and $(*, \Box)$.

$$\frac{}{\vdash \{\}}\ (\text{Empty}) \qquad \frac{\begin{array}{c}\Gamma \vdash A : s \\ s \in \{Kind, Type\} \\ x \text{ is a fresh variable}\end{array}}{\vdash \Gamma \cup \{x : A\}}\ (\text{Var-Decl})$$

$$\frac{\vdash \Gamma}{\Gamma \vdash Type : Kind}\ (\text{Type}) \qquad \frac{\begin{array}{c}\vdash \Gamma \\ (x : A) \in \Gamma\end{array}}{\Gamma \vdash x : A}\ (\text{Var})$$

$$\frac{\begin{array}{c}\Gamma \vdash A : Type \\ x \text{ is a fresh variable} \\ \Gamma \cup \{x : A\} \vdash B : s \\ s \in \{Kind, Type\}\end{array}}{\Gamma \vdash \Pi x{:}A.B : s}\ (\text{Prod}) \qquad \frac{\begin{array}{c}\Gamma \vdash A : Type \\ x \text{ is a fresh variable} \\ \Gamma \cup \{x : A\} \vdash M : B \\ \Gamma \cup \{x : A\} \vdash B : s \\ s \in \{Kind, Type\}\end{array}}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}\ (\text{Abs})$$

$$\frac{\begin{array}{c}\Gamma \vdash M : \Pi x{:}A.B \\ \Gamma \vdash N : A\end{array}}{\Gamma \vdash (M\ N) : A[N/x]}\ (\text{Appl}) \qquad \frac{\begin{array}{c}\Gamma \vdash M : A \\ \Gamma \vdash B : s \\ s \in \{Kind, Type\} \\ A \equiv_\beta B\end{array}}{\Gamma \vdash M : B}\ (\text{Conv})$$

Fig. 1. The λP-type system.

$$\frac{\begin{array}{c}x \text{ is a fresh variable} \\ \Gamma \cup \{x : A\} \vdash B : s \\ s \in \{Kind, Type\}\end{array}}{\Gamma \vdash \Pi x{:}A.B : s}\ (\text{Prod}) \qquad \frac{\begin{array}{c}x \text{ is a fresh variable} \\ \Gamma \cup \{x : A\} \vdash M : B \\ \Gamma \cup \{x : A\} \vdash B : s \\ s \in \{Kind, Type\}\end{array}}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}\ (\text{Abs})$$

Fig. 2. Rules (Prod) and (Abs) of the Calculus of Constructions.

following grammar.

Terms $M, N ::= \underline{1} | (MN) | \lambda M | M[S]$

Substitutions $S, T ::= id | \uparrow | M \cdot S | S \circ T$

The $\lambda\sigma$-calculus is presented in Fig. 3.

In $\lambda\sigma$, free and bound variables are represented by de Bruijn indices. They are encoded by means of the constant $\underline{1}$ and the substitution $\uparrow$. We write $\uparrow^n$ as a shorthand

$$
\begin{array}{lcll}
(\lambda M\ N) & \longrightarrow & M[N \cdot id] & \text{(Beta)} \\
(M\ N)[S] & \longrightarrow & (M[S]\ N[S]) & \text{(Application)} \\
(\lambda M)[S] & \longrightarrow & \lambda M[\underline{1} \cdot (S \circ \uparrow)] & \text{(Lambda)} \\
M[S][T] & \longrightarrow & M[S \circ T] & \text{(Clos)} \\
\underline{1}[M \cdot S] & \longrightarrow & M & \text{(VarCons)} \\
M[id] & \longrightarrow & M & \text{(Id)} \\
(S_1 \circ S_2) \circ T & \longrightarrow & S_1 \circ (S_2 \circ T) & \text{(Ass)} \\
(M \cdot S) \circ T & \longrightarrow & M[T] \cdot (S \circ T) & \text{(Map)} \\
id \circ S & \longrightarrow & S & \text{(Idl)} \\
S \circ id & \longrightarrow & S & \text{(Idr)} \\
\uparrow \circ (M \cdot S) & \longrightarrow & S & \text{(ShiftCons)} \\
\underline{1} \cdot \uparrow & \longrightarrow & id & \text{(VarShift)} \\
\underline{1}[S] \cdot (\uparrow \circ S) & \longrightarrow & S & \text{(SCons)}
\end{array}
$$

Fig. 3. The $\lambda\sigma$-calculus [1].

for $\overbrace{\uparrow \circ \cdots \circ \uparrow}^{n\text{-times}}$. We overload the notation $\underline{i}$ to represent the $\lambda\sigma$-term corresponding to the index i, i.e.,

$$
\underline{i} = \begin{cases} \underline{1} & \text{if } i = 1, \\ \underline{1}[\uparrow^n] & \text{if } i = n+1. \end{cases}
$$

An explicit substitution denotes a mapping from indices to terms. Thus, id maps each index i to the term $\underline{i}$, $\uparrow$ maps each index i to the term $\underline{i+1}$, $S \circ T$ 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.

2. A framework to represent incomplete proof-terms

The important elements of our framework are: explicit substitutions, open terms, and dependent types. A simply-typed version of $\lambda\sigma$ on open terms has been studied in [15]. In [31, 33], we propose the $\lambda\Pi_{\mathscr{L}}$-calculus which is a dependent-typed version of a variant of $\lambda\sigma$. The $\lambda\Pi_{\mathscr{L}}$-calculus is confluent and weakly normalizing on well-typed terms.

As usual in explicit substitution calculi, expressions of $\lambda\Pi_{\mathscr{L}}$ are structured in terms and substitutions. The $\lambda\Pi_{\mathscr{L}}$-calculus admits meta-variables only on the sort of terms.

$$
\begin{array}{lcll}
(\lambda_A.M\ N) & \longrightarrow & M[N \cdot_A \uparrow^0] & \text{(Beta)} \\
(\lambda_A.M)[S] & \longrightarrow & \lambda_{A[S]}.M[\underline{1} \cdot_A (S \circ \uparrow^1)] & \text{(Lambda)} \\
(\Pi_A.B)[S] & \longrightarrow & \Pi_{A[S]}.B[\underline{1} \cdot_A (S \circ \uparrow^1)] & \text{(Pi)} \\
(M\ N)[S] & \longrightarrow & (M[S]\ N[S]) & \text{(Application)} \\
M[S][T] & \longrightarrow & M[S \circ T] & \text{(Clos)} \\
\underline{1}[M \cdot_A S] & \longrightarrow & M & \text{(VarCons)} \\
M[\uparrow^0] & \longrightarrow & M & \text{(Id)} \\
(M \cdot_A S) \circ T & \longrightarrow & M[T] \cdot_A (S \circ T) & \text{(Map)} \\
\uparrow^0 \circ S & \longrightarrow & S & \text{(IdS)} \\
\uparrow^{n+1} \circ (M \cdot_A S) & \longrightarrow & \uparrow^n \circ S & \text{(ShiftCons)} \\
\uparrow^{n+1} \circ \uparrow^m & \longrightarrow & \uparrow^n \circ \uparrow^{m+1} & \text{(ShiftShift)} \\
\underline{1} \cdot_A \uparrow^1 & \longrightarrow & \uparrow^0 & \text{(Shift0)} \\
\underline{1}[\uparrow^n] \cdot_A \uparrow^{n+1} & \longrightarrow & \uparrow^n & \text{(ShiftS)} \\
Type[S] & \longrightarrow & Type & \text{(Type)}
\end{array}
$$

Fig. 4. The $\lambda\Pi_{\mathscr{L}}$-rewrite system.

The set of well-formed expressions in $\lambda\Pi_{\mathscr{L}}$ is defined by the following grammar:

$$
\begin{array}{llll}
\textbf{Natural numbers} & n & ::= & 0 \mid n+1 \\
\textbf{Meta-variables} & \chi & ::= & X \mid Y \mid \ldots \\
\textbf{Sorts} & s & ::= & Kind \mid Type \\
\textbf{Terms} & A,B,M,N & ::= & \underline{1} \mid s \mid \Pi_A.B \mid \lambda_A.M \mid \\
 & & & (MN) \mid M[S] \mid \chi \\
\textbf{Substitutions} & S,T & ::= & \uparrow^n \mid M \cdot_A S \mid S \circ T
\end{array}
$$

The equivalence relation $\equiv_{\lambda\Pi_{\mathscr{L}}}$ is defined as the symmetric and transitive closure of the relation induced by the rewrite system in Fig. 4. As usual, we denote by $\overset{\lambda\Pi_{\mathscr{L}}^*}{\longrightarrow}$ the reflexive and transitive closure of $\lambda\Pi_{\mathscr{L}}$.

The system $\Pi_{\mathscr{L}}$ is obtained by dropping rule (Beta) from $\lambda\Pi_{\mathscr{L}}$. As shown by Zantema (personal communication), the $\Pi_{\mathscr{L}}$-calculus is strongly normalizing.

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

Proof. See [33]. The proof uses the semantic labeling technique [39]. □

The set of normal-forms of an expression x (term or substitution) is denoted by $(x)\downarrow_{\Pi_{\mathscr{L}}}$.

An expression in $\lambda\Pi_{\mathscr{L}}$ is *ground* if it does not contain meta-variables. A ground expression is also *pure* if it is a $\Pi_{\mathscr{L}}$-normal form.

The $\lambda\Pi_{\mathscr{L}}$-calculus, just as $\lambda\sigma$, uses the composition operation to achieve confluence on terms with meta-variables. Rules (Idr) and (Ass) of $\lambda\sigma$ are not necessary in $\lambda\Pi_{\mathscr{L}}$.

We adopt the notation $\underline{i}$ as a shorthand for $\underline{1}[\uparrow^n]$ when $i = n+1$. In contrast to $\lambda\sigma$, $\uparrow^n$ is not a shorthand but an explicit substitution in $\lambda\Pi_{\mathscr{L}}$. Indeed, $\uparrow^0$ replaces id and $\uparrow^1$ replaces $\uparrow$. In general, $\uparrow^n$ denotes the mapping of each index i to the term $\underline{i+n}$. Using $\uparrow^n$, the non-left-linear rule (SCons) of $\lambda\sigma$, which is responsible of confluence and typing problems [11, 5, 33], can be dropped of the $\lambda\Pi_{\mathscr{L}}$-calculus. Notice that we do not assume any meta-theoretical property on natural numbers. They are constructed with 0 and $n+1$. Arithmetic calculations on indices are embedded in the rewrite system.

A *context* in $\lambda\Pi_{\mathscr{L}}$ is a list of types. The empty context is written ε. A context with head A and rest Γ is written $A.\ \Gamma$. In that case, A is the type of the index 1, the head of Γ (if Γ is not empty) is the type of the index 2, and so forth. In a dependent-type theory with de Bruijn indices, the order in which variables are declared in a context is important. In fact, in the context $A.\ \Gamma$, the indices in A are relative to Γ.

The type of a substitution is a context. This choice seems natural since substitutions denote mapping from indices to terms, and contexts are list of types. In fact, if the type of a substitution S is the context $A.\ \Delta$, the type of the term mapped by the substitution S on the index 1 is A, and so forth for the rest of indices.

2.1. Meta-variables

As we have said, meta-variables are first-class objects in $\lambda\Pi_{\mathscr{L}}$. Just as variables, they have to be declared in order to keep track of possible dependences 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. The pair (Γ, A) is unique (modulo $\equiv_{\lambda\Pi_{\mathscr{L}}}$) for each meta-variable. This requirement is enforced by the type system.

A list of meta-variable declarations is called a *signature*. We use the Greek letter Σ to range over signatures. The empty signature is written ε. A signature with head $X:_{\Gamma} A$ and rest Σ is written $X:_{\Gamma} A.\ \Sigma$. We overload the notation $\Sigma_1.\ \Sigma_2$ to write the concatenation of the signatures Σ_1 and Σ_2.

The order of the meta-variable declarations is important. In a signature $X_1 :_{\Gamma_1} A_1 \ldots .\ X_n :_{\Gamma_n} A_n$, the type A_i and the context Γ_i, $0 < i \leqslant n$, may depend only on meta-variables X_j, $i < j \leqslant n$. The indices in A_i are relative to the context Γ_i.

The main operation on meta-variables is *instantiation*. The instantiation of a meta-variable X with a term M in an expression y (term or substitution) replaces all the occurrences of X in y by M.

Definition 2 (*Instantiation*). The instantiation of a meta-variable X with a term M in an expression y, denoted $y\{X/M\}$, is defined by induction over the structure of y as follows.

- $s\{X/M\} = s$, if $s \in \{Kind, Type\}$.
- $\underline{1}\{X/M\} = \underline{1}$.
- $X\{X/M\} = M$.
- $Y\{X/M\} = Y$, if $Y \neq X$.

- $(\Pi_A.B)\{X/M\} = \Pi_{A\{X/M\}}.B\{X/M\}$.
- $(\lambda_A.N)\{X/M\} = \lambda_{A\{X/M\}}.N\{X/M\}$.
- $(N_1\ N_2)\{X/M\} = (N_1\{X/M\}\ N_2\{X/M\})$.
- $(N[S])\{X/M\} = N\{X/M\}[S\{X/M\}]$.
- $\uparrow^n\{X/M\} = \uparrow^n$.
- $(N \cdot_A S)\{X/M\} = N\{X/M\} \cdot_{A\{X/M\}} S\{X/M\}$.
- $(S \circ T)\{X/M\} = S\{X/M\} \circ T\{X/M\}$.

Application of instantiations extends to context and signatures, that is, $\Gamma\{X/M\}$ and $\Sigma\{X/M\}$, in the obvious way. In the case of signatures, the application $\Sigma\{X/M\}$ also removes the declaration of X in Σ, if it exits.

In contrast to substitution of variables, instantiation of meta-variables allows capture of variables. Moreover, instantiations are not first-class objects, i.e., the application of an instantiation is atomic and external to the $\lambda\Pi_{\mathscr{L}}$-calculus.

2.2. Type annotations

Type annotations in substitutions are introduced with rules (Beta), (Lambda), and (Pi), and then propagated with rule (Map). They can also be eliminated with rules (VarCons), (ShiftCons), and (Shift0). Notice that the type annotation that is propagated by rule (Map):

$$(M \cdot_A S) \circ T \to M[T] \cdot_A (S \circ T)$$

is A, not $A[T]$. Type annotations in substitutions act as remainder of types when substitutions are distributed under abstractions and products. As shown in [33], they are necessary to preserve typing in $\lambda\Pi_{\mathscr{L}}$-reductions.

2.3. η-conversion

In this paper we consider a calculus without η-conversion. Although, extensional versions of explicit substitution calculi have been studied for ground terms [24], work is necessary to understand the interaction of the η-rule with explicit substitutions, dependent types, and meta-variables.

2.4. Dependent types

In $\lambda\Pi_{\mathscr{L}}$, we consider typing assertions having one of the following forms:

$$\vdash \Sigma; \Gamma$$

to capture that the context Γ is valid in the signature Σ,

$$\Sigma; \Gamma \vdash M : A$$

to capture that the term M has type A (the type M has the kind A) in $\Sigma; \Gamma$, and

$$\Sigma; \Gamma \vdash S \triangleright \Delta$$

$$\frac{}{\vdash \varepsilon;\varepsilon}\ (\text{Empty}) \qquad \frac{\Sigma;\Gamma \vdash A : s \quad s \in \{Kind, Type\}}{\vdash \Sigma; A.\ \Gamma}\ (\text{Var-Decl})$$

$$\frac{\vdash \Sigma;\Gamma \quad X \text{ is a fresh meta-variable}}{\vdash X:_{\Gamma}\ Kind.\ \Sigma}\ (\text{Meta-Var-Decl}_1)$$

$$\frac{\Sigma;\Gamma \vdash A : s \quad s \in \{Kind, Type\} \quad X \text{is a fresh meta-variable}}{\vdash X:_{\Gamma}\ A.\ \Sigma}\ (\text{Meta-Var-Decl}_2)$$

Fig. 5. Valid signatures and contexts.

to capture that the substitution S has the type Δ in $\Sigma;\Gamma$. The scoping rules for variables and meta-variables are as follows. Contexts Γ, Δ, and expressions M, A, S may depend on any meta-variable declared in their respective signature Σ. Indices in M, A and S are relative to their respective context Γ.

Typing rules for signatures, contexts, and expressions are all mutually dependent. Valid signatures and contexts are defined by the typing rules in Fig. 5. Valid expressions in the λP-type system are defined by the typing rules in Fig. 6. In the case of the Calculus of Constructions, rules (Prod), (Abs), and (Cons) are modified as indicated in Fig. 7. Finally, conversion rules, are defined in Fig. 8.

In the following, we use $\vdash \Sigma$, $\vdash \Gamma$, $\Gamma \vdash M : A$, and $\Gamma \vdash S \triangleright \Delta$ as shorthands for $\vdash \Sigma;\varepsilon$, $\vdash \varepsilon;\Gamma$, $\varepsilon;\Gamma \vdash M : A$, and $\varepsilon;\Gamma \vdash S \triangleright \Delta$, respectively.

In this paper, unless otherwise stated, a judgment like $\Sigma;\Gamma \vdash M : A$ refers to the setting of $\lambda\Pi_{\mathscr{D}}$ in the Calculus of Constructions. However, the main properties of $\lambda\Pi_{\mathscr{D}}$ hold in both the Calculus of Constructions and the λP-type system. We prove in [31, 33] that $\lambda\Pi_{\mathscr{D}}$ satisfies, among others, the following properties (for the sake of simplicity we show the properties only for typed terms, but they hold in the same way for typed substitutions):

Proposition 3 (Sort soundness). *If $\Sigma;\Gamma \vdash M : A$, then either $A = Kind$, or $\Sigma;\Gamma \vdash A : s$, where $s \in \{Kind, Type\}$.*

Proposition 4 (Type uniqueness). *If $\Sigma;\Gamma \vdash M : A$ and $\Sigma;\Gamma \vdash M : B$, then $A \equiv_{\lambda\Pi_{\mathscr{D}}} B$.*

Proposition 5 (Subject reduction). *If $M \xrightarrow{\lambda\Pi_{\mathscr{D}}^*} N$ and $\Sigma;\Gamma \vdash M : A$, then $\Sigma;\Gamma \vdash N : A$.*

Proposition 6 (Soundness). *If $\Sigma;\Gamma \vdash M : A$, $\Sigma;\Gamma \vdash N : B$ and $M \equiv_{\lambda\Pi_{\mathscr{D}}} N$, then there exists a path of well-typed reductions between A and B.*

Proposition 7 (Weak normalization). *If $\Sigma;\Gamma \vdash M : A$, then M is weakly normalizing; therefore, M has at least one $\lambda\Pi_{\mathscr{D}}$-normal form.*

$$\frac{\vdash \Sigma;\Gamma}{\Sigma;\Gamma \vdash \mathit{Type} : \mathit{Kind}}\ (\text{Type}) \qquad \frac{\vdash \Sigma;A.\ \Gamma}{\Sigma;A.\ \Gamma \vdash \underline{1} : A[\uparrow^1]}\ (\text{Var})$$

$$\frac{\begin{array}{c}\Sigma;\Gamma \vdash A : \mathit{Type}\\ \Sigma;A.\ \Gamma \vdash B : s\\ s \in \{\mathit{Kind}, \mathit{Type}\}\end{array}}{\Sigma;\Gamma \vdash \Pi_A.B : s}\ (\text{Prod}) \qquad \frac{\begin{array}{c}\Sigma;\Gamma \vdash A : \mathit{Type}\\ \Sigma;A.\ \Gamma \vdash M : B\\ \Sigma;\Gamma \vdash \Pi_A.B : s\\ s \in \{\mathit{Kind}, \mathit{Type}\}\end{array}}{\Sigma;\Gamma \vdash \lambda_A.M : \Pi_A.B}\ (\text{Abs})$$

$$\frac{\begin{array}{c}\Sigma;\Gamma \vdash M : \Pi_A.B\\ \Sigma;\Gamma \vdash N : A\end{array}}{\Sigma;\Gamma \vdash (M\ N) : B[N \cdot_A \uparrow^0]}\ (\text{Appl}) \qquad \frac{\begin{array}{c}\Sigma;\Gamma \vdash S \triangleright \Delta\\ \Sigma;\Delta \vdash M : A\\ \Sigma;\Delta \vdash A : s\\ s \in \{\mathit{Kind}, \mathit{Type}\}\end{array}}{\Sigma;\Gamma \vdash M[S] : A[S]}\ (\text{Clos})$$

$$\frac{\begin{array}{c}\Sigma;\Gamma \vdash S \triangleright \Delta\\ \Sigma;\Delta \vdash A : \mathit{Kind}\end{array}}{\Sigma;\Gamma \vdash A[S] : \mathit{Kind}}\ (\text{Clos-Kind}) \qquad \frac{\begin{array}{c}\vdash \Sigma;\Gamma\\ X :_\Delta A \in \Sigma\\ \Delta \equiv_{\lambda\Pi_{\mathscr{L}}} \Gamma\end{array}}{\Sigma;\Gamma \vdash X : A}\ (\text{Meta-Var})$$

$$\frac{\vdash \Sigma;\Gamma}{\Sigma;\Gamma \vdash \uparrow^0 \triangleright \Gamma}\ (\text{Id}) \qquad \frac{\begin{array}{c}\vdash \Sigma;A.\ \Gamma\\ \Sigma;\Gamma \vdash \uparrow^n \triangleright \Delta\end{array}}{\Sigma;A.\ \Gamma \vdash \uparrow^{n+1} \triangleright \Delta}\ (\text{Shift})$$

$$\frac{\begin{array}{c}\Sigma;\Gamma \vdash S \triangleright \Delta_1\\ \Sigma;\Delta_1 \vdash T \triangleright \Delta_2\end{array}}{\Sigma;\Gamma \vdash T \circ S \triangleright \Delta_2}\ (\text{Comp}) \qquad \frac{\begin{array}{c}\Sigma;\Gamma \vdash M : A[S]\\ \Sigma;\Gamma \vdash S \triangleright \Delta\\ \Sigma;\Delta \vdash A : \mathit{Type}\end{array}}{\Sigma;\Gamma \vdash M \cdot_A S \triangleright A.\ \Delta}\ (\text{Cons})$$

Fig. 6. Valid expressions (λP-type system).

Proposition 8 (Church–Rosser). *If $M_1 \equiv_{\lambda\Pi_{\mathscr{L}}} M_2$, $\Sigma;\Gamma \vdash M_1 : A$, and $\Sigma;\Gamma \vdash M_2 : A$, then M_1 and M_2 are $\lambda\Pi_{\mathscr{L}}$-joinable, i.e., there exists M such that $M_1 \xrightarrow{\lambda\Pi_{\mathscr{L}}^*} M$ and $M_2 \xrightarrow{\lambda\Pi_{\mathscr{L}}^*} M$.*

Corollary 9 (Normal forms). *The $\lambda\Pi_{\mathscr{L}}$-normal form of a well-typed $\lambda\Pi_{\mathscr{L}}$-term always exists, and it is unique. If M is a well-typed term, we denote by $(M)\downarrow_{\lambda\Pi_{\mathscr{L}}}$ its $\lambda\Pi_{\mathscr{L}}$-normal form.*

The following proposition states the conditions that guarantee the soundness of instantiation of meta-variables in $\lambda\Pi_{\mathscr{L}}$.

$$\frac{\Sigma; A.\ \Gamma \vdash B : s \quad s \in \{Kind, Type\}}{\Sigma; \Gamma \vdash \Pi_A.B : s}\ (\text{Prod}) \qquad \frac{\Sigma; A.\ \Gamma \vdash M : B \quad \Sigma; A.\ \Gamma \vdash B : s \quad s \in \{Kind, Type\}}{\Sigma; \Gamma \vdash \lambda_A.M : \Pi_A.B}\ (\text{Abs})$$

$$\frac{\Sigma; \Gamma \vdash M : A[S] \quad \Sigma; \Gamma \vdash S \rhd \Delta \quad \Sigma; \Delta \vdash A : s \quad s \in \{Kind, Type\}}{\Sigma; \Gamma \vdash M \cdot_A S \rhd A.\ \Delta}\ (\text{Cons})$$

Fig. 7. Rules (Prod), (Abs), and (Cons) of the Calculus of Constructions.

$$\frac{\Sigma; \Gamma \vdash M : A \quad \Sigma; \Gamma \vdash B : s \quad s \in \{Kind, Type\} \quad A \equiv_{\lambda\Pi_{\mathscr{L}}} B}{\Sigma; \Gamma \vdash M : B}\ (\text{Conv}) \qquad \frac{\Sigma; \Gamma \vdash S \rhd \Delta_1 \quad \vdash \Sigma; \Delta_2 \quad \Delta_1 \equiv_{\lambda\Pi_{\mathscr{L}}} \Delta_2}{\Sigma; \Gamma \vdash S \rhd \Delta_2}\ (\text{Conv-Subs})$$

Fig. 8. Conversions.

Proposition 10 (Instantiation lemma). *Let M be a term such that $\Sigma_1; \Gamma \vdash M : A$, and Σ a signature having the form $\Sigma_2.\ X :_{\Gamma} A.\ \Sigma_1$,*
(1) *if $\vdash \Sigma; \Delta$, then $\vdash \Sigma_2\{X/M\}.\ \Sigma_1; \Delta\{X/M\}$,*
(2) *if $\Sigma; \Delta \vdash N : B$, then*
$\Sigma\{X/M\}; \Delta\{X/M\} \vdash N\{X/M\} : B\{X/M\}$, and
(3) *if $\Sigma; \Delta_1 \vdash S \rhd \Delta_2$, then $\Sigma\{X/M\}; \Delta_1\{X/M\} \vdash S\{X/M\} \rhd \Delta_2\{X/M\}$.*

Finally, the next property justifies the use of $\lambda\Pi_{\mathscr{L}}$ to build proof-terms in a constructive logic based on a dependent-type system. It states that when the signature is empty, $\lambda\Pi_{\mathscr{L}}$ types as many terms as the λ-calculus does.

Proposition 11 (Conservative extension). *Let M, A be pure terms in $\lambda\Pi_{\mathscr{L}}$, and Γ be a context containing only pure terms. Then, $\Gamma \vdash M : A$ in $\lambda\Pi_{\mathscr{L}}$ if and only if $\Gamma \vdash M : A$ in the respective dependent-typed version of the λ-calculus* (*modulo de Bruijn indices translation*).

3. A proof synthesis method in $\lambda\Pi_{\mathscr{L}}$

We introduce the basic ideas of our technique with an example. For readability, when discussing examples we use named variables and not de Bruijn indices. Nevertheless, we recall that our formalism uses a de Bruijn nameless notation of variables.

Assume a context with the variable declarations

$$
\begin{aligned}
&bool : Type,\\
&nat : Type,\\
&f : nat \to nat \to bool,\\
&g : (nat \to bool) \to nat,\\
&not : bool \to bool,\\
&eq : bool \to bool \to Type,\\
&h : \Pi p{:}\ (nat \to bool) \to bool.\Pi x : nat \to bool.(eq\,(p\,x)\,(not\,(p\,(f\,(g\,x))))).
\end{aligned}
$$

We address the problem of finding terms X and Y such that $X : (eq\ Y\,Y)$ and $Y : bool$. This problem happens to be a paraphrasing of a formulation given in [14] of the famous Cantor's theorem that there is not surjection from a set (in this case nat) to its power set (formed by the elements of type $nat \to bool$). It can be solved, for example using Dowek's method, by enumerating at the same time the terms Y of type $bool$ and the terms of type $(eq\ Y\,Y)$.

However, by combining proof construction and term synthesis we can do better. Instead of looking directly for Y, we could claim to know it, and try to find a term of type $(eq\ Y\,Y)$. Then, we use the typing information available for eq to guide the proof-term synthesis.

In our framework, we assume two meta-variable declarations $Y : bool$ and $X : (eq\ Y\,Y)$. Notice that the meta-variable Y appears in the type of X. In fact, in contrast to the simply-typed λ-calculus, in a dependent-typed calculus meta-variables may appear in types and in contexts. Typing rules for open terms should take into account these considerations.

A *solution* to X and Y is a couple of ground terms M, A such that when X is instantiated with M and Y with A, it holds $M : (eq\ A\,A)$ and $A : bool$.

By looking at the context of variables, we notice that a possible instantiation for X should use the variable h. Since we do not know the right arguments p and x to apply h, we declare new meta-variables $X_p : (nat \to bool) \to bool$ and $X_x : nat \to bool$, and proceed to instantiate X with $(h\,X_p\,X_x)$.

At this stage of the development, we have the following situation. Three meta-variables to solve: $Y : bool$, $X_p : (nat \to bool) \to bool$, and $X_x : nat \to bool$, and the incomplete proof-term $(h\,X_p\,X_x)$ of type $(eq\ Y\ Y)$. However, there is something wrong. The type given by the type system to the term $(h\,X_p\,X_x)$ is $(eq\,(X_p\,X_x)\,(not\,(X_p\,(f\,(g\,X_x)))))$, which is not convertible to $(eq\ Y\ Y)$. In fact, we should have been more careful with the instantiation of X with $(h\ X_p\ X_x)$. Since two syntactically different types can become equal via instantiation of meta-variables and β-reduction, we can instantiate a meta-variable with a term of different type, but we have to keep track of a set of disagreement types. In our case, if we want to instantiate X with $(h\,X_p\,X_x)$, we have to add the constraint $(eq\,(X_p\,X_x)\,(not\,(X_p\,(f\,(g\,X_x)))))\simeq(eq\ Y\ Y)$ to the disagreement set.

Thus, the goal is not to find *any* ground instantiation for the meta-variables, but one that reduces the disagreement set to a set of trivial constraints of the form $M \simeq M$, where M is a ground term.

If the original proposition holds, eventually we will instantiate all the meta-variables in such a way that the disagreement set is also solved. A possible solution to our example is

$$X_x = \lambda y : nat.(not\,(f\,y\,y)),$$
$$X_p = \lambda x : nat \to bool.(x\,(g\lambda y : nat.(not\,(f\,y\,y)))),$$
$$Y = (not\,(f(g\,\lambda y : nat.(not\,(f\,y\,y))))), \text{ and}$$
$$X = (h\,\lambda x : nat \to bool.(x\,(g\,\lambda y : nat.(not\,(f\,y\,y))))\,\lambda y : nat.(not\,(f\,y\,y))).$$

That solution was found by our prototype in 209 rounds (including back-tracking steps). Each round corresponds to the instantiation of one meta-variable or the simplification of the disagreement set. This number contrasts with the 1024 rounds that it took our algorithm to find the same solution by first enumerating all the terms of type *bool*.

The method to solve a set of meta-variables and a disagreement set can be summarized as:

(1) Take a meta-variable X to solve. Because eventually all the meta-variables have to be solved, any of them can be chosen. However, as we will explain later, some typing properties guide the choice of an appropriate meta-variable to solve.
(2) By using the type information, propose a term M, probably containing new meta-variables, to instantiate X.
(3) Declare the new meta-variables appearing in M and add to the disagreement set the typing constraints necessary to guarantee the soundness of the instantiation.
(4) Simplify the disagreement set. If a typing constraint is unsatisfiable, backtrack to step 2. Restore the disagreement set to that point.
(5) Stop if all the meta-variables are solved and the disagreement set contains only trivial equations. Otherwise, call recursively the procedure.

Our method improves Dowek's method in three ways:

- Proof construction and term synthesis are combined in a single method. Therefore, proof assistant systems based on the proofs-as-terms paradigm can use our framework to represent uniformly proof under construction and proof-terms.
- The first-order setting of the $\lambda\Pi_{\mathscr{D}}$-calculus eliminates most of the technical problems related to the higher-order aspects of the λ-calculus.
- In Dowek's method, variables, and not meta-variables, are used to represent placeholders. Since, these variables should range over all the set of well-typed terms, the type system where the proof synthesis method is described allows variable declarations where the original type system does not. That type system introduces some technical nuisances [12, 13]. In our framework this is not necessary. Meta-variables and variables have different declaration rules. In particular, meta-variables can be typed in sorts where variables cannot (see rules (Meta-Var-Decl$_1$), (Meta-Var-Decl$_1$), and (Var-Decl)).

3.1. The $\lambda\Pi_{\mathscr{L}}$-calculus with constraints

As we have seen in the informal description of the method, instantiation of meta-variables may need the resolution of a disagreement set. Indeed, the disagreement set is maintained in an extended kind of signatures called *constrained signatures*.

Definition 12 (*Constrained signatures*). A *constraint* $M \simeq_\Gamma N$ relates two terms M, N, and a context Γ. A *constrained signature* is a list containing meta-variable declarations and constraint declarations. Formally, they are defined by the following grammar:

Constrained signatures $\Xi ::= \varepsilon \mid X :_\Gamma A.\Xi \mid M \simeq_\Gamma N.\Xi$

Notice that constraints are declared together with meta-variables. This way, the type system may enforce that a constraint uses only meta-variables that have already been declared in a signature.

Definition 13 (*Equivalence modulo constraints*). Let Ξ be a constrained signature; we define the relation $\equiv_\Xi$ as the smallest equivalence relation compatible with structure such that

(1) if $M \equiv_{\lambda\Pi_{\mathscr{L}}} N$, then $M \equiv_\Xi N$, and
(2) if $M \simeq_\Gamma N \in \Xi$, then $M \equiv_\Xi N$.

We extend the $\lambda\Pi_{\mathscr{L}}$-calculus to deal with constraints.

Definition 14 ($\lambda\Pi_{\mathscr{L}}$ *with constraints*). The type system $\lambda\Pi_{\mathscr{L}}$ with constraints is defined as $\lambda\Pi_{\mathscr{L}}$ in Section 2, where we denote typing judgments by $\vdash\!\sim \Xi$, $\vdash\!\sim \Xi; \Gamma$ and $\Xi; \Gamma \vdash\!\sim M : A$, we add the rule

$$\frac{\Xi; \Gamma \vdash\!\sim M_1 : A \qquad \Xi; \Gamma \vdash\!\sim M_2 : A}{\vdash\!\sim M_1 \simeq_\Gamma M_2.\Xi}\ (\text{Constraint})$$

and we replace rules (Conv), (Conv-Subs), and (Meta-Var) by

$$\frac{\Xi; \Gamma \vdash\!\sim M : A \qquad \Xi; \Gamma \vdash\!\sim B : s \qquad s \in \{Kind, Type\} \qquad A \equiv_\Xi B}{\Xi; \Gamma \vdash\!\sim M : B}(\text{Conv})$$

$$\frac{\Sigma; \Gamma \vdash\!\sim S \lhd \Delta \qquad \vdash\!\sim \Sigma; \Delta' \qquad \Delta \equiv_\Xi \Delta'}{\Sigma; \Gamma \vdash\!\sim S \rhd \Delta'}(\text{Conv-Subs})$$

$$\frac{\vdash \Sigma; \Gamma \qquad X :_\Delta A \in \Sigma \qquad \Delta \equiv_\Xi \Gamma}{\Sigma; \Gamma \vdash X : A}(\text{Meta-Var})$$

As expected, a constrained signature Ξ is said to be *valid* if it holds $\vdash\!\sim \Xi$.

The $\lambda\Pi_{\mathscr{L}}$-calculus with constraints does not satisfy most of the typing properties of $\lambda\Pi_{\mathscr{L}}$ given in Section 2. In particular, it is not normalizing (not even weakly). For instance, the non-terminating term $(\lambda x{:}A.(x\ x)\quad \lambda x{:}A.(x\ x))$ can be typed in a constrained signature containing $A \simeq_\Gamma A \to A$ for some context Γ.

However, we can prove the following properties.

Lemma 15. *Let Ξ be a valid constrained signature and Σ be the signature where we have removed all the constraints of Ξ,*

(1) (a) *if* $\vdash \Sigma;\Gamma$, *then* $\mid\!\sim \Xi;\Gamma$,
(b) *if* $\Sigma;\Gamma \vdash M : A$, *then* $\Xi;\Gamma \mid\!\sim M : A$, *and*
(c) *if* $\Sigma;\Gamma \vdash S \triangleright \Delta$, *then* $\Xi;\Gamma \mid\!\sim S \triangleright \Delta$; *and*

(2) *if Ξ does not contain constraints, i.e., $\Sigma = \Xi$, then*
(a) *if* $\mid\!\sim \Xi;\Gamma$, *then* $\vdash \Sigma;\Gamma$,
(b) *if* $\Xi;\Gamma \mid\!\sim M : A$, *then* $\Sigma;\Gamma \vdash M : A$, *and*
(c) *if* $\Xi;\Gamma \mid\!\sim S \triangleright \Delta$, *then* $\Sigma;\Gamma \vdash S \triangleright \Delta$.

Proof. By simultaneous induction on the typing derivations. □

According to Lemma 15, if Ξ' is a prefix of a signature Ξ, and it does not contain constraints, the set of expressions that are typable in Ξ' satisfies the properties given in Section 2; in particular, these expressions have a $\lambda\Pi_{\mathscr{L}}$-normal form (Corollary 9). We exploit this fact to simplify constrained signatures. Indeed, we define the $\lambda\Pi_{\mathscr{L}}$-normal form of a constrained signature, with respect to the largest prefix which does not contain a constraint. We will see later that constrained signatures in $\lambda\Pi_{\mathscr{L}}$-normal form allow us to prune the search space of solutions to meta-variables.

Definition 16 (*Normal form of a constrained signature*). Let Ξ be a valid constrained signature, the *$\lambda\Pi_{\mathscr{L}}$-normal form* of Ξ, denoted by $(\Xi)\downarrow_{\lambda\Pi_{\mathscr{L}}}$, is defined by structural induction on Ξ.

(1) $(\varepsilon)\downarrow_{\lambda\Pi_{\mathscr{L}}} = \varepsilon$,
(2) Ξ has the form $X{:}_\Gamma A.\ \Xi'$ or $M \simeq_\Gamma N.\ \Xi'$
- if Ξ' contains constraints,

$$(X{:}_\Gamma A.\,\Xi')\downarrow_{\lambda\Pi_{\mathscr{L}}} = X{:}_\Gamma A.(\Xi')\downarrow_{\lambda\Pi_{\mathscr{L}}}$$

$$(M \simeq_\Gamma N.\,\Xi')\downarrow_{\lambda\Pi_{\mathscr{L}}} = M \simeq_\Gamma N.(\Xi')\downarrow_{\lambda\Pi_{\mathscr{L}}},$$

- if Ξ' does not contain constraints,

$$(X{:}_\Gamma A.\,\Xi')\downarrow_{\lambda\Pi_{\mathscr{L}}} = X{:}_{(\Gamma)\downarrow_{\lambda\Pi_{\mathscr{L}}}} (A)\downarrow_{\lambda\Pi_{\mathscr{L}}}.(\Xi')\downarrow_{\lambda\Pi_{\mathscr{L}}}$$

$$(M \simeq_\Gamma N.\,\Xi')\downarrow_{\lambda\Pi_{\mathscr{L}}} = (\Xi')\downarrow_{\lambda\Pi_{\mathscr{L}}},\ \text{if}\ (M)\downarrow_{\lambda\Pi_{\mathscr{L}}} = (N)\downarrow_{\lambda\Pi_{\mathscr{L}}}$$

$$(M \simeq_\Gamma N.\,\Xi')\downarrow_{\lambda\Pi_{\mathscr{L}}} = (M)\downarrow_{\lambda\Pi_{\mathscr{L}}} \simeq_{(\Gamma)\downarrow_{\lambda\Pi_{\mathscr{L}}}} (N)\downarrow_{\lambda\Pi_{\mathscr{L}}}.(\Xi')\downarrow_{\lambda\Pi_{\mathscr{L}}},\ \text{otherwise.}$$

We show below that the $\lambda\Pi_{\mathscr{L}}$-normal form of a constrained signature preserves typing.

Lemma 17. *Let Ξ be a valid constrained signature,*
(1) $\vdash\sim \Xi;\Gamma$ *if and only if* $\vdash\sim(\Xi)\downarrow_{\lambda\Pi_{\mathscr{L}}};\Gamma$,
(2) $\Xi;\Gamma\vdash M:A$ *if and only if* $(\Xi)\downarrow_{\lambda\Pi_{\mathscr{L}}};\Gamma\vdash\sim M:A$, *and*
(3) $\Xi;\Gamma\vdash S\triangleright\Delta$ *if and only if* $(\Xi)\downarrow_{\lambda\Pi_{\mathscr{L}}};\Gamma\vdash\sim S\triangleright\Delta$.

Proof. By simultaneous induction on the typing derivations. □

3.2. The problem

A constrained signature can be seen as a list of goals to be solved. Informally speaking, to solve a signature means to find ground instantiations for all the meta-variables in a way that all the constraints are reduced to trivial equations.

Definition 18 (*Parallel instantiation*). A *parallel instantiation* of a constrained signature Ξ is a function Ψ_Ξ from meta-variables of Ξ to terms. As usual, the function Ψ_Ξ is extended to be applied to arbitrary expressions. When Ξ can be inferred from the context, we simply write Ψ.

Definition 19 (*Solution*). Let Ξ be a valid constrained signature, we say that a parallel instantiation Ψ is a *solution* to Ξ if and only if
(1) for any constraint $M\simeq_\Gamma N\in\Xi$, we have $\Psi(\Gamma)\vdash\Psi(M):A$, $\Psi(\Gamma)\vdash\Psi(N):A$ and $\Psi(M)\equiv_{\lambda\Pi_{\mathscr{L}}}\Psi(N)$, and
(2) for any meta-variable declaration $X:_\Gamma A\in\Xi$, we have $\Psi(\Gamma)\vdash\Psi(X):\Psi(A)$.

In this case we say that Ξ is a *solvable* signature. Furthermore, if for all meta-variables X in Ξ, $\Psi(X)$ is a $\lambda\Pi_{\mathscr{L}}$-normal form, we say that Ψ is a *normal solution* to Ξ.

Notice that according to the previous definition, if Ψ is a solution to a constrained signature Ξ, for all meta-variables X in Ξ, $\Psi(X)$ is a ground term. If Ψ is also a normal solution, then $\Psi(X)$ is pure.

Definition 20 (*Equivalent solutions*). Let Ψ_1,Ψ_2 be solutions to a valid constrained signature Ξ. They are said to be *equivalent*, denoted $\Psi_1\equiv_{\lambda\Pi_{\mathscr{L}}}\Psi_2$, if and only if for all X in Ξ, $\Psi_1(X)\equiv_{\lambda\Pi_{\mathscr{L}}}\Psi_2(X)$.

To know whether or not a valid constrained signature is solvable is undecidable in the general case. In particular, it requires to decide the existence of solutions for constraints having the form $(X\ M_1\ \ldots\ M_i)\simeq_\Gamma(Y\ N_1\ \ldots\ N_j)$, where X and Y are meta-variables, and to solve the inhabitation problem in a dependent-type system. Those problems are known to be undecidable [29, 4].

Some kinds of signatures can be trivially discharged.

Remark 21. If a valid constrained signature Ξ is solvable, then there exists a *normal solution* to Ξ.

Definition 22 (*Failure signature*). Let Ξ be the $\lambda\Pi_{\mathscr{L}}$-normal form of a valid constrained signature; we say that Ξ is a *failure signature* if it contains a constraint relating two ground terms in $\lambda\Pi_{\mathscr{L}}$-normal form which are not identical.

Remark 23. Failure signatures are not solvable.

The Cantor's theorem example can be described in our formalism as follows. Let $\Gamma =$

$$h : \Pi p{:}(nat \to bool) \to bool.\, \Pi x{:}nat \to bool.\, (eq\ (p\ x)\ (not\ (p\ (f\ (g\ x))))).$$

$$eq : bool \to bool \to Type.\ not : bool \to bool.$$

$$g : (nat \to bool) \to nat.\ f : nat \to nat \to bool.\ bool : Type.\ nat : Type,$$

and $\Xi = X{:}_{\Gamma}\ (eq\ Y\ Y).\ Y{:}_{\Gamma}\ bool$, the following parallel instantiation Ψ is a solution to Ξ:

$$\Psi(Y) = (not\ (f\ (g\ \lambda y : nat.\, (not\ (f\ y\ y)))))$$

$$\Psi(X) = (h\ \lambda x{:}nat \to bool.\, (x\ (g\ \lambda y{:}nat.\, (not\ (f\ y\ y))))\ \lambda y{:}nat.\, (not\ (f\ y\ y))).$$

In the process of finding that solution, we have first solved the constrained signature $\Xi' =$

$$X \simeq_{\Gamma} (h\ X_p\ X_x).\ (eq\ (X_p\ X_x)\ (not\ (X_p\ (f\ (g\ X_x))))) \simeq_{\Gamma} (eq\ Y\ Y).$$

$$X_x{:}_{\Gamma}\ nat \to bool.\ X_p{:}_{\Gamma}\ (nat \to bool) \to bool.\ X{:}_{\Gamma}\ (eq\ Y\ Y).\ Y{:}_{\Gamma}\ bool,$$

which has the solution

$$\Psi'(X_p) = \lambda x : nat \to bool.\, (x(g\ \lambda y : nat.(not(f\ y\ y))))$$

$$\Psi'(X_x) = \lambda y : nat.(not(f\ y\ y))$$

$$\Psi'(Z) = \Psi(Z),\ \text{otherwise.}$$

It can be verified that, for example, $\Psi'((eq\ (X_p\ X_x)\ (not\ (X_p\ (f\ (g\ X_x))))) \equiv_{\lambda\Pi_{\mathscr{L}}} \Psi'(eq\ Y\ Y)$.

In the rest of this section, we describe a method to find a solution to a constrained signature via refinement steps. In the example above, Ξ' is a refinement of Ξ, and thus, a solution to Ξ can be deduced from a solution to Ξ'.

3.3. The construction steps: elementary graftings

We want to solve a constrained signature via successive instantiation of meta-variables. Each one of these instantiations is called an *elementary grafting*.[3]

[3] In Dowek's method, they are called *elementary substitutions*.

Definition 24 (*Grafting*). A *grafting* is an instantiation of a meta-variable, with possibly new declarations of meta-variables and constraints. Let X be a meta-variable, M be a term, and Ξ' be a constrained signature, the grafting of X with M in Ξ' is denoted by $\{X/_{\Xi'}M\}$.

Valid graftings (in Ξ) are defined by the following typing rule;

$$\frac{\begin{array}{c}\vdash\sim \Xi \\ \Xi = \Xi_2\,.\,X :_{\Gamma} A\,.\,\Xi_1 \\ \Xi'\,.\,\Xi_1; \Gamma \vdash\sim M : A \\ \vdash\sim \Xi_2.\Xi'\,.\,\Xi_1\end{array}}{\Xi \vdash\sim \{X/_{\Xi'}M\}} \quad \text{(Grafting)}$$

In the previous definition, Ξ' contains *only* the additional meta-variables and constraints that are necessary to type M. However, $\Xi_2.\ \Xi'.\ \Xi_1$ is a conservative extension of Ξ, i.e., all the expressions that are typable in Ξ, are typable in $\Xi_2.\ \Xi'.\ \Xi_1$, too. In particular, it holds $\vdash\sim \Xi'.\ \Xi_1$.

The grafting $\{X/_{\Xi'}M\}$ can be applied to an expression or a context in the same way as the instantiation $\{X/M\}$. However, only valid grafting can be applied to constrained signatures. Let Ξ be a valid constrained signature, the application of the grafting $\{X/_{\Xi'}M\}$ to Ξ, instantiates the meta-variable X with M in Ξ, and installs Ξ' in the right place of Ξ.

Definition 25 (*Application of grafting*). Let $\Xi{=}\Xi_2.\ X{:}_{\Gamma}\ A.\ \Xi_1$ such that $\Xi \vdash\sim \{X/_{\Xi'}M\}$,

$$\Xi\{X/_{\Xi'}M\} = \Xi_2\{X/M\}\,.\,\Xi'\,.\,\Xi_1.$$

The application of a valid grafting preserves typing.

Lemma 26. *Let Ξ be a valid constrained signature such that $\Xi \vdash\sim \{X/_{\Xi'}M\}$,*
(1) *if* $\vdash\sim \Xi; \Gamma$, *then* $\vdash\sim \Xi\{X/_{\Xi'}M\}; \Gamma\{X/M\}$,
(2) *if* $\Xi; \Gamma \vdash\sim M : A$, *then* $\Xi\{X/_{\Xi'}M\}; \Gamma\{X/M\} \vdash\sim M\{X/M\} : A\{X/M\}$, *and*
(3) *if* $\Xi; \Gamma \vdash\sim S \triangleright \Delta$, *then* $\Xi\{X/_{\Xi'}M\}; \Gamma\{X/M\} \vdash\sim S\{X/M\} \triangleright \Delta\{X/M\}$.

Proof. By induction on the typing derivations. The proof uses Proposition 10. □

The reduction to $\lambda\Pi_{\mathscr{S}}$-normal form of a constrained signature preserves its valid graftings.

Lemma 27. *Let Ξ be a valid constrained signature, $\Xi \vdash\sim \{X/_{\Xi'}M\}$ if and only if $(\Xi)\downarrow_{\lambda\Pi_{\mathscr{S}}} \vdash\sim \{X/_{\Xi'}M\}$.*

Proof. We show that $\Xi \vdash\sim \{X/_{\Xi'}M\}$ implies $(\Xi)\downarrow_{\lambda\Pi_{\mathscr{S}}} \vdash\sim \{X/_{\Xi'}M\}$. The other direction is similar. By Lemma 17, $(\Xi)\downarrow_{\lambda\Pi_{\mathscr{S}}}$ is a valid constrained signature. By Definition 16, Ξ and $(\Xi)\downarrow_{\lambda\Pi_{\mathscr{S}}}$ declare exactly the same meta-variables. By hypothesis, meta-variables

declared in Ξ' are not in Ξ. Since Ξ has the form $\Xi'_2.\ X:_\Gamma A'.\ \Xi'_1$, $(\Xi)\downarrow_{\lambda\Pi_{\mathscr{L}}}$ has the form $\Xi_2.\ X:_\Gamma A.\ \Xi_1$, where

(1) $\Xi'.\ \Xi'_1;\Gamma \mathrel{|\!\sim} M : A'$, and
(2) $\Xi_1 = (\Xi'_1)\downarrow_{\lambda\Pi_{\mathscr{L}}}$, $A = (A')\downarrow_{\lambda\Pi_{\mathscr{L}}}$.

From (1) and (3), $\Xi'.\ \Xi'_1;\Gamma \mathrel{|\!\sim} M : A$. Therefore, by Lemma 17 and (3), $\Xi'.\ \Xi_1;\Gamma \mathrel{|\!\sim} M : A$. □

For our Cantor's theorem example we verify below that

$$\Xi \mathrel{|\!\sim} \{|X/_{\Xi'}(h\,X_p\,X_x)|\},$$

where $\Xi = X:_\Gamma (eq\ Y\ Y).\ Y:_\Gamma bool$, and $\Xi' =$
$(eq\ (X_p\ X_x)\ (not\ (X_p\ (f\ (g\ X_x)))))) \simeq_\Gamma (eq\ Y\ Y).$
$X_x:_\Gamma nat \to bool.\ X_p:_\Gamma (nat \to bool) \to bool.$
In fact, Ξ' contains meta-variables which are not already declared in Ξ (thus, Ξ' can be safely installed in Ξ), X is declared in Ξ, and

$$\Xi'.\ Y:_\Gamma\ bool\ \mathrel{|\!\sim} (h\ X_p\ X_x) : (eq\ Y\ Y).$$

Then, by Definition 24,

$$\Xi \mathrel{|\!\sim} \{|X/_{\Xi'}(h\ X_p\ X_x)|\}.$$

Given a constrained signature, the choice of the next meta-variable to solve is crucial. Since properties like confluence and normalization are available for any typable expression in a prefix of a constrained signature without constraints, meta-variables in those prefixes are very appropriate to solve in the first place. The next property states that such variables exist.

Lemma 28. *Let Ξ be the $\lambda\Pi_{\mathscr{L}}$-normal form of a valid constrained signature such that $\Xi \neq \varepsilon$ and Ξ is not a failure signature. Then, Ξ has the form $\Xi_2.\ X:_\Gamma A.\ \Xi_1$, where*
(1) *Ξ_1 does not contain constraints, and*
(2) $\vdash X:_\Gamma A.\ \Xi_1$.

Proof. The constrained signature Ξ is not empty, then it has at least one element. Assume that the first element is a constraint $M \simeq_\Gamma N$. By hypothesis and Lemma 17, $\mathrel{|\!\sim} \Xi$. Hence, it holds that $\mathrel{|\!\sim} M \simeq_\Gamma N$. By inversion of rule (Constraint), $\Gamma \mathrel{|\!\sim} M : B$ and $\Gamma \mathrel{|\!\sim} N : B$. Since M, N, B are well-typed without meta-variables, they are ground, and by Lemma 15, it holds that $\Gamma \vdash M : B$ and $\Gamma \vdash N : B$. Since Ξ is a signature in $\lambda\Pi_{\mathscr{L}}$-normal form, M and N are not identical. But this is not possible because Ξ is not a failure context. Therefore, the first element of Ξ is not a constraint, and thus, Ξ has the form $\Xi_2.\ X:_\Gamma A.\ \Xi_1$, where Ξ_1 does not contain constraints. By the typing rules, we have $\mathrel{|\!\sim} X:_\Gamma A.\ \Xi_1$, and thus, by Lemma 15, $\vdash X:_\Gamma A.\ \Xi_1$. □

The type of a meta-variable gives enough information to guess a valid grafting. Assume, for example, that a meta-variable X has a type A. If $A = Kind$, then by inversion of the type rule (Type), X may be instantiated with $Type$. But also, by inversion of rule (Prod), X may be instantiated with the term $\Pi x{:}Z.Y$ where Z is a new type meta-variable and Y is a new meta-variable of type A (notice that Y should be declared in a context where the variable declaration $x : Z$ exists). This case also applies if $A = Type$.

If A is a product, i.e., $A = \Pi x : A_1.A_2$, by inversion of rule (Abs), we can instantiate X with the term $\lambda x{:}A_1.Y$ where Y is a new meta-variable of type A (declared in a context where the variable declaration $x : A_1$ exists).

In any case, and by inversion of rule (Appl), it is always possible to instantiate X with the term $(Y\,Z)$, where Y is a meta-variable of type $\Pi x{:}Y_B.Y_A$, Z is a meta-variable of type Y_B, Y_B is a type meta-variable, Y_A is a meta-variable with the same type as A (declared in a context where the variable declaration $x : Y_B$ exists), and the constraint $A \simeq Y_A[Z \cdot_{Y_B} \uparrow^0]$ is added to the constrained signature. However, since we are interested in solutions modulo $\equiv_{\lambda\Pi_{\mathscr{L}}}$, any normal instantiation of Y has the form $(n\,M_1 \ldots M_i)$ where n is a variable. Using this remark, we simplify the current case by using the variables of the context where the meta-variable X has been declared. Assume a variable declaration $n : \Pi x_1{:}A_1. \ldots \Pi x_j{:}A_j.B_1$. The meta-variable X can be instantiated with the term $(n\,X_1 \ldots X_i)$ of type B_2, where $i \leqslant j$, $X_1, \ldots, X_i$ are new meta-variables of the right type (according to the type of n), and the constraint $A \simeq B_2$ is added to the constrained signature. We call this case *imitation*, because it is very similar to the *imitation* rule of higher-order unification algorithms [22].

The imitation case, as it has been described before, is not complete. In a polymorphic type system, as the Calculus of Constructions, if the type of a term M is $\Pi x{:}A.B$, where B is not a product, the type of $(M\,N)$ may still be a product. That is, the number of arguments of M is not bounded by the number of products in its type. Take for example the context $O : nat.\ nat : Type.\ P : \Pi x{:}Type.x$. In this context, $(P\,nat) : nat$, $(P\,(nat \to nat)\,O) : nat$, $(P\,(nat \to nat \to nat)\,O\,O) : nat, \ldots$. In fact, for any natural number $i > 0$, there exist $M_1, \ldots, M_i$ such that $(P\,M_1 \ldots M_i) : nat$.

The fact that the number of arguments of a term is not fixed by its type is called *splitting* [21]. Splitting raises some technical problems in higher-order unification algorithms and so, in proof-synthesis methods [13].

Given the valid judgment $\Sigma; \Gamma \vdash M : \Pi x_1{:}A_1. \ldots \Pi x_i{:}A_i.B$, where B is not a product, for any $j > 0$, there exists a term N having the form $(M\,X_1 \ldots X_j)$ such that it is well-typed in a constrained signature extending Σ. The term N is called an *imitation of M of grade j*. Furthermore, if $j > i$, $(j - i)$ is the *splitting grade* of N. Otherwise, the splitting grade of N is 0. We describe a method to build imitations of arbitrary splitting grade.

Definition 29 (*Imitation with splitting*). Let Σ be a signature, without constraints, in $\lambda\Pi_{\mathscr{L}}$-normal form, M be a term such that $\Sigma; \Gamma \vdash M : A$, and $\Sigma; \Gamma \vdash A : s$ where $s \in \{Kind, Type\}$. For $i \geqslant 0$, the set of imitations of M of grade i, denoted $[\Sigma; \Gamma \vdash$

$M:A]^i$, is a set of judgments in $\lambda\Pi_{\mathscr{L}}$, with constraints, defined by induction on i as follows.

- If $i=0$, then $\{\Sigma;\Gamma \mid\sim M:A\}$.
- If $i>0$, then for all $\Xi;\Gamma \mid\sim N:B$ in $[\Sigma;\Gamma \vdash M:A]^{i-1}$, we consider the union of the following set of judgments.[4]
  - If B has the form $\Pi_{A_1}.A_2$, then

$$\{\Xi'.\Xi;\Gamma \mid\sim N' : B' \mid \Xi' = M :_\Gamma A_1,$$
$$X \text{ is a fresh meta-variable},$$
$$N' = (N\,X),$$
$$B' \in (A_2[X\cdot_{A_1}\uparrow^0])\downarrow_{\Pi_{\mathscr{L}}}\}$$

  - Otherwise – this is the case of splitting,

$$\{\Xi'.\Xi;\Gamma \mid\sim N' : B' \mid \Xi' = B' \simeq_\Gamma \Pi_{Y1}.Y_2.\ X :_\Gamma Y_1.\ Y_2 :_{Y_1.\Gamma} s_2.\ Y_1 :_\Gamma s_1,$$
$$X, Y_1, Y_2 \text{ are fresh meta-variables},$$
$$s_1 \in \{Kind, Type\},$$
$$s_2 = s,$$
$$N' = (N\,X),$$
$$B' \in (Y_2[X\cdot_{Y_1}\uparrow^0])\downarrow_{\Pi_{\mathscr{L}}}\}$$

We verify that judgments in the set $[\Sigma;\Gamma \vdash M:A]^i$ are valid.

Lemma 30. *Let Σ be a signature in $\lambda\Pi_{\mathscr{L}}$-normal form, M be a term such that $\Sigma;\Gamma \vdash M:A$ and $\Sigma;\Gamma \vdash A:s$ where $s\in\{Kind, Type\}$. For $i\geqslant 0$, the elements of $[\Sigma;\Gamma \vdash M:A]^i$ are valid judgments.*

Proof. By induction on i. The base case holds by Lemma 15. At the induction step we use rules (Appl), (Conv), and the fact that the reduction to $\Pi_{\mathscr{L}}$-normal form preserves the type. □

We formally define the *elementary graftings*.

Definition 31 (*Elementary graftings*). Let Ξ be the $\lambda\Pi_{\mathscr{L}}$-normal form of a valid constrained signature such that $\Xi\neq\varepsilon$ and Ξ is not a failure signature. We choose a meta-variable X in Ξ, i.e., $\Xi=\Xi_2.\ X:_\Gamma A.\ \Xi_1$, such that $\vdash X:_\Gamma A.\ \Xi_1$. Such a meta-variable exists by Lemma 28. We define the following graftings by case analysis on A (the cases are not disjoint):

(1) $A=Kind$. We consider the grafting $\{X/_\varepsilon Type\}$.

[4] We recall that $\Pi_{\mathscr{L}}$ is strongly normalizing (Lemma 1).

(2) $A \in \{Kind, Type\}$. For any $s \in \{Kind, Type\}$, we consider the grafting $\{X/_{\Xi'} \Pi_Z.Y\}$, where Z, Y are fresh meta-variables, and $\Xi' = Y:_{Z.\ \Gamma}\ A.\ Z:_{\Gamma}\ s$.

(3) $A = \Pi_{A_1}.A_2$. We consider the grafting $\{X/_{\Xi'} \lambda_{A_1}.Y\}$, where Y is a fresh meta-variable and $\Xi' = Y:_{A_1.\ \Gamma}\ A_2$.

(4) $\Xi_1; \Gamma \vdash A : s_1$, $s_1 \in \{Kind, Type\}$. For all variables n in the context Γ, i.e., $1 \leqslant n \leqslant |\Gamma|$, such that $\Xi_1; \Gamma \vdash \underline{n} : B$ (B is a $\lambda\Pi_{\mathscr{L}}$-normal form), we consider all the graftings

$$\{X \setminus_{A \simeq \Gamma A'.\Xi'} M\}$$

where $\Xi'.\ \Xi_1; \Gamma \mid\sim M : A'$ is in $[\Xi_1; \Gamma \vdash \underline{n} : B]^i$, for $i \geqslant 0$.

All the graftings considered above form the set of *elementary graftings* of the meta-variable X in Ξ.

Due to the splitting rule, the set of elementary graftings of one meta-variable is potentially infinite. Some of the elementary graftings lead to failure signatures. An early detection of failure signatures allows the pruning of the research space of valid graftings. This is why we use constrained signatures in $\lambda\Pi_{\mathscr{L}}$-normal form.

We verify that the elementary graftings are valid graftings.

Theorem 32 (Elementary graftings). *Let Ξ be the $\lambda\Pi_{\mathscr{L}}$-normal form of a valid constrained signature such that $\Xi \neq \varepsilon$ and Ξ is not a failure signature. If X is a meta-variable in Ξ such that it is well-typed without constraints, then the elementary graftings of X are valid graftings in Ξ.*

Proof. By Lemma 28, Ξ has the form $\Xi_2.\ X:_{\Gamma}\ A.\ \Xi_1$. First, we verify that

$$\mid\sim \Xi_1; \Gamma, \tag{1}$$

$$A = Kind \text{ or } \Xi_1; \Gamma \mid\sim A : s, \quad s \in \{Kind, Type\}. \tag{2}$$

Then, we reason by case analysis on A. We consider all the elementary graftings of X.

- $A = Kind$. By using Eq. (1) with rule (Type), we get $\Xi_1; \Gamma \mid\sim Type : Kind$. Therefore, $\Xi \mid\sim \{X/_{\varepsilon} Type\}$.
- $A \in \{Kind, Type\}$. For any $s' \in \{Kind, Type\}$, we consider the grafting $\{X/_{\Xi'} \Pi_Z.Y\}$, where Y, Z are fresh meta-variables, and $\Xi' = Y:_{Z.\ \Gamma}\ A.\ Z:_{\Gamma}\ s'$.
  We consider two cases according to the form of s'.
  - $s' = Kind$. We have the derivation
  $$\frac{\text{Eq. (1)}}{\mid\sim Z :_{\Gamma}\ Kind.\Xi_1}(\text{Meta-Var-Decl}_1)$$
  - $s' = Type$. We have the derivation
  $$\frac{\dfrac{\text{Eq. (1)}}{\Xi_1; \Gamma \mid\sim Type : Kind}(\text{Type})}{\mid\sim Z :_{\Gamma}\ Type.\Xi_1}(\text{Meta-Var-Decl}_2)$$

In both cases,

$$\vdash\sim Z :_{\Gamma} s'.\Xi_1. \tag{3}$$

The derivation continues as follows:

$$\dfrac{\dfrac{\text{Eq. (3)}}{Z :_{\Gamma} s'.\Xi_1; \Gamma \vdash\sim Z : s'}\,(\text{Meta-Var})}{\vdash\sim Z :_{\Gamma} s'.\Xi_1; Z.\Gamma}\,(\text{Var-Decl})$$

Now, we consider two cases according to the form of A.

- $A = Kind$. We have the derivation

$$\dfrac{\vdash\sim Z :_{\Gamma} s'.\Xi_1; Z.\Gamma}{\vdash\sim Y :_{Z.\Gamma} Kind.Z :_{\Gamma} s'.\Xi_1}\,(\text{Meta-Var -Decl}_1)$$

- $A = Type$. We have the derivation

$$\dfrac{\dfrac{\vdash\sim Z :_{\Gamma} s'.\Xi_1; Z.\Gamma}{Z :_{\Gamma} s'.\Xi_1; \Gamma \vdash\sim Type : Kind}\,(\text{Type})}{\vdash\sim Y :_{Z.\Gamma} Type.Z :_{\Gamma} s'.\Xi_1}\,(\text{Meta-Var -Decl}_2)$$

In both cases,

$$\vdash\sim Y :_{Z.\Gamma} A.\ Z :_{\Gamma} s'.\Xi_1. \tag{4}$$

But also

$$\dfrac{\dfrac{\text{Eq. (4)}}{Y :_{Z.\Gamma} A.\ Z :_{\Gamma} s'.\Xi_1; Z.\Gamma \vdash\sim Y : A}\,(\text{Meta-Var})}{Y :_{Z.\Gamma} A.\ Z :_{\Gamma} s'.\Xi_1; \Gamma \vdash\sim \Pi_Z Y : A}\,(\text{Prod})$$

Therefore, $\Xi \vdash\sim \{X/_{\Xi'} \Pi_Z.Y\}$.

- $A = \Pi_{A_1}.A_2$. We consider the grafting $\{X/_{\Xi'} \lambda_{A_1}.Y\}$, where Y is a fresh meta-variable, and $\Xi' = Y:_{A_1.\ \Gamma} A_2$. As in the previous case we have the derivation

$$\dfrac{\text{Eq. (2)}\quad \dfrac{\cdots}{\Xi'.\Xi_1; A_1.\Gamma \vdash\sim Y : A_2}}{\Xi'.\Xi_1 \vdash\sim \lambda_{A_1}.Y : \Pi_{A_1}.A_2}\,(\text{Abs})$$

Therefore, $\Xi \vdash\sim \{X/_{\Xi'} \lambda_{A_1}.Y\}$.

- For $1 \leqslant n \leqslant |\Gamma|$ such that $\Xi_1; \Gamma \vdash \underline{n} : B$ (B is a $\lambda\Pi_{\mathscr{D}}$-normal form), we consider all the graftings

$$\{X \setminus_{A \simeq \Gamma A'.\Xi'} M\}$$

where $\Xi'.\ \Xi_1; \Gamma \vdash\sim M : A'$ is in $[\Sigma; \Gamma \vdash \underline{n} : B]^i$, $i \geqslant 0$. By Lemma 30,

$$\Xi'.\ \Xi_1; \Gamma \vdash\sim M : A', \tag{5}$$

$$\Xi'.\ \Xi_1 \vdash\sim A' : s. \tag{6}$$

We also have

$$\dfrac{\text{Eq. (5)} \quad \dfrac{\Xi'.\Xi_1\Gamma \ \ \sim A : s \ \ \Xi'.\Xi_1;\Gamma \sim A' : s}{\sim A \simeq_\Gamma A'.\Xi'.\Xi_1}\,(\text{Constraint})}{A \simeq_\Gamma A'.\Xi'.\Xi_1;\Gamma \sim M : A.}\,(\text{Conv})$$

Therefore, $\Xi \sim \{X/_{A \simeq_\Gamma A'.\ \Xi'} M\}$. □

3.4. Splitting in λP

In a calculus without polymorphism, as λP, splitting is not possible. Thus, in that case the number of applications of a variable is fixed by its type. In our version of λP with meta-variables and explicit substitutions, splitting is still possible since we allow meta-variables of types and kinds. However, some simplifications are still possible.

A term having the form $(X[S]\,M_1\,M_i)$ or $(X\,M_1\,M_i)$, $i \geqslant 0$, where X is a meta-variable is said to be *flexible*. A term having the form *Type*, *Kind*, or $(\underline{n}\,M_1\,M_i)$, $i \geqslant 0$ is said to be *rigid*.

In the λP-type version of $\lambda\Pi_{\mathscr{D}}$, consider a term M such that $\Xi;\Gamma \sim M : \Pi_{A_1} \ldots \Pi_{A_i} B$. If B is a $\lambda\Pi_{\mathscr{D}}$-normal form and it is not a product, it is either flexible or rigid. If B is flexible, the number of applications of n depends on the actual parameters of M. If B is rigid, the number of applications of M cannot be greater than i. In that case, we could consider imitations of M only of grade $j \leqslant i$. Since their splitting grade is 0, the set of such imitations is finite (module renaming of fresh meta-variables).

3.5. Putting everything together: the method

Given a constrained signature Ξ, we solve each meta-variable by exploring the set of its elementary graftings. We can organize the search of elementary graftings as follows.

Definition 33 (*Search tree*). Let Ξ be a valid constrained signature; we build a *search tree* of Ξ, where nodes are labeled by constrained signatures in $\lambda\Pi_{\mathscr{D}}$-normal form and edges by elementary graftings, in the following way:

- The root is labeled by $(\Xi)\downarrow_{\lambda\Pi_{\mathscr{D}}}$.
- Nodes labeled by the empty signature or by failure signatures are leaves.
- If a node is labeled by a signature Ξ which is not empty or a failure signature, we choose a meta-variable X in Ξ such that it is well-typed in a signature without constraints and for each elementary grafting $\{X/_{\Xi'} M\}$ of X, we grow an edge labeled by this elementary signature to a new node labeled by $(\Xi\{X/_{\Xi'} M\})\downarrow_{\lambda\Pi_{\mathscr{D}}}$.

We claim that if there exists a node labeled by the empty signature in a search tree of Ξ, then Ξ is solvable, and a solution can be found by composing sequentially all the elementary graftings along a path in the search tree containing the node labeled by the empty signature. Conversely, if there exists a solution to a constrained signature Ξ, it can be found, modulo $\equiv_{\lambda\Pi_{\mathscr{D}}}$, in a search tree of Ξ. These two properties, *soundness and completeness*, are proved in Section 4.

A semi-algorithm to solve a valid constrained signature is to enumerate the nodes of a search tree to find a leaf labeled by the empty signature. Notice that the enumeration must deal with infinite paths in the tree, but also with infinite branching because the set of elementary graftings of a meta-variable is potentially infinite.

Example 34 (*Revisited Cantor's theorem example*). Let Γ be the context

$$h : \Pi p{:}(nat \to bool) \to bool.\, \Pi x{:}nat \to bool.\, (eq\ (p\ x)\ (not\ (p\ (f\ (g\ x))))).$$

$$eq : bool \to bool \to Type.\ not : bool \to bool.$$

$$g : (nat \to bool) \to nat.\ f : nat \to nat \to bool.\ bool : Type.\ nat : Type,$$

and $\Xi = X{:}_{\Gamma}\ (eq\ Y\ Y).\ Y{:}_{\Gamma}\ bool$. We find below a solution to Ξ via $\lambda\Pi_{\mathscr{D}}$.

A search tree is built from the root Ξ (notice that it is a $\lambda\Pi_{\mathscr{D}}$-normal form). Since Ξ does not contain constraints, we can take any meta-variable of Ξ to solve. Let us choose the meta-variable X. The type of X is neither a product nor a sort. Therefore, the only elementary graftings that are possible for this meta-variable are those generated by the imitation step. We instantiate X with an imitation of grade 2 of the variable h (no splitting takes place),

$$\begin{aligned}
&[\Xi; \Gamma \vdash h :\ \Pi p : (nat \to bool) \to bool.\Pi x : nat \to bool.\\
&\qquad (eq\ (p\ x)\ (not\ (p\ (f\ (g\ x)))))]^2 =\\
&\{\Xi'.\Xi; \Gamma \mid\sim (h\ X_p\ X_x) : (eq\ (X_p\ X_x)\ (not\ (X_p\ (f\ (g\ X_x)))))|\\
&\ X_x, X_p\ \textit{are fresh meta}\text{-variables},\\
&\ \Xi' = X_x :\ {}_{\Gamma} nat \to bool.\ X_p :\ {}_{\Gamma}(nat \to bool) \to bool\}
\end{aligned}$$

We label an edge with the elementary grafting,

$$\{X/_{\Xi_1}(h\ X_p\ X_x)\},$$

where $\Xi_1 =$

$$(eq\ (X_p\ X_x)\ (not\ (X_p\ (f\ (g\ X_x))))) \simeq_{\Gamma} (eq\ Y\ Y).$$

$$X_x{:}_{\Gamma}\ nat \to bool.\ X_p{:}_{\Gamma}\ (nat \to bool) \to bool.$$

This edge points to the constrained signature:

$$(eq\ (X_p\ X_x)\ (not\ (X_p\ (f\ (g\ X_x))))) \simeq_{\Gamma} (eq\ Y\ Y).$$

$$X_x{:}_{\Gamma}\ nat \to bool.\ X_p{:}_{\Gamma}\ (nat \to bool) \to bool.$$

$$Y{:}_{\Gamma}\ bool.$$

Notice that the meta-variable X is no longer in the signature. Instead, there are new meta-variables X_x and X_p. At this stage, any meta-variable can be chosen. We solve the meta-variable X_x of type $nat \to bool$. An elementary grafting of this meta-variable is

$$\{X_x/_{\Xi_2} \lambda y : nat.Z\},$$

where $\Xi_2 = Z:_{y:nat.\ \Gamma}\ bool$. We label a new edge with this elementary grafting. It points to the constrained signature:

$$(eq\ (X_p\ \lambda y{:}nat.Z)\ (not\ (X_p\ (f\ (g\ \lambda y{:}nat.Z))))) \simeq_\Gamma (eq\ Y\ Y).$$

$$Z:_{y:nat.\,\Gamma}\ bool.\ X_p:_{\Gamma}\ (nat \to bool) \to bool.$$

$$Y:_{\Gamma}\ bool.$$

Eventually, after some iterations an empty signature is obtained. A solution is found by composing all the elementary graftings along the path of the search tree leading to the empty signature.

4. Soundness and completeness

4.1. Soundness

We claim that if $\Xi_1 \overset{\theta_1}{\longmapsto} \Xi_2 \overset{\theta_2}{\longmapsto} \cdots \overset{\theta_{n-1}}{\longmapsto} \Xi_n$ is a path of the search tree of a valid constrained signature Ξ, such that $\Xi_1 = (\Xi)\downarrow_{\lambda\Pi_{\mathscr{L}}}$ and $\Xi_n = \varepsilon$, the sequential composition of the graftings $\theta_1, \ldots, \theta_{n-1}$ results in a solution to Ξ.

The proof of this statement goes as follows. First, we describe the lists of grafting that are valid with respect to a valid constrained signature. These lists are called *sequential graftings*. Next, we characterize the sequential graftings that lead to an empty signature. They are called *derivations*. The key points of the proof are:

(1) The sequential composition of the graftings in a derivation of Ξ is a solution to Ξ.
(2) A path from the root of a search tree of Ξ leading to an empty signature is a derivation of Ξ.

The soundness theorem is a consequence of (1) and (2).

Definition 35 (*Sequential grafting*). A list $\psi = \langle \theta_1, \ldots, \theta_i \rangle$, $i \geqslant 0$, of graftings is a *sequential grafting* of a valid constrained signature Ξ if and only if

- ψ is the empty list, i.e., $i = 0$, or
- $\Xi \mathrel{|\!\sim} \theta_1$ and $\langle \theta_2, \ldots, \theta_i \rangle$ is a sequential grafting of $\Xi\theta_1$.

The *application of* ψ to Ξ, is defined as $\Xi\psi = ((\Xi\theta_1)\ldots)\theta_i$. We overload this notation to apply sequential graftings to expressions and contexts.

Definition 36 (*Derivation*). A sequential grafting ψ of a valid constrained signature Ξ is called a *derivation* of Ξ if and only if $(\Xi\psi)\downarrow_{\lambda\Pi_{\mathscr{L}}} = \varepsilon$.

Remark 37. Failure signatures do not have derivations.

Definition 38 (*Sequential composition*). Let ψ be a sequential grafting of a valid constrained signature Ξ. The *sequential composition* of ψ, denoted by $\tilde{\psi}$, is the parallel instantiation defined for all X in Ξ as $\tilde{\psi}(X) = X\psi$.

The next propositions are proved at the end of this section. They are the key proving the soundness theorem.

Proposition 39. *If ψ is a derivation of a valid constrained signature Ξ, then $\tilde{\psi}$ – the sequential composition of ψ – is a solution to Ξ.*

Proposition 40. *Let $\Xi_1 \overset{\theta_1}{\longmapsto} \Xi_2 \overset{\theta_2}{\longmapsto} \cdots \overset{\theta_{n-1}}{\longmapsto} \Xi_n$, $n \geqslant 0$, be a path of a search tree of a valid constrained signature Ξ such that $\Xi_1 = (\Xi)\downarrow_{\lambda\Pi_{\mathscr{D}}}$, then the list of graftings $\psi = \langle \theta_1, \ldots, \theta_{n-1} \rangle$ is a sequential grafting of Ξ, and for $0 < i \leqslant n$, $\Xi_i = (\Xi\psi)\downarrow_{\lambda\Pi_{\mathscr{D}}}$.*

Theorem 41 (Soundness). *Let $(\Xi)\downarrow_{\lambda\Pi_{\mathscr{D}}} \overset{\psi}{\longmapsto} \varepsilon$ be a path of a search tree of a valid constrained signature Ξ, the sequential composition of ψ is a solution to Ξ.*

Proof. By Proposition 40, ψ is a sequential grafting of Ξ, and $\varepsilon = (\Xi\psi)\downarrow_{\lambda\Pi_{\mathscr{D}}}$. Therefore, by Definition 36, ψ is a derivation of Ξ. Finally, by Proposition 39, the sequential composition of ψ, i.e., $\tilde{\psi}$, is a solution to Ξ. □

The rest of this section is dedicated to the proof of Proposition 39 and Proposition 40.

First, we prove that sequential graftings preserve typing.

Lemma 42. *Let ψ be a sequential grafting of a valid constrained signature Ξ,*
(1) *if $\vdash_\sim \Xi; \Gamma$, then $\vdash_\sim \Xi\psi; \Gamma\psi$,*
(2) *if $\Xi; \Gamma \vdash_\sim M : A$, then $\Xi\psi; \Gamma\psi \vdash_\sim M\psi : A\psi$, and*
(3) *if $\Xi; \Gamma \vdash_\sim S \triangleright \Delta$, then $\Xi\psi; \Gamma\psi \vdash_\sim S\psi \triangleright \Delta\psi$.*

Proof. We reason by induction on the length of the list ψ and Lemma 26. □

Proposition 39 states that if ψ is a derivation of a valid constrained signature Ξ, then $\tilde{\psi}$ is a solution to Ξ.

Proof of Proposition 39. Since Ξ is a valid constrained signature, for any constraint $M_1 \simeq_\Gamma M_2$ and meta-variable declaration $X :_\Delta A$ in Ξ,

$$\Xi; \Gamma \vdash_\sim M_1 : B, \tag{7}$$

$$\Xi; \Gamma \vdash_\sim M_2 : B, \tag{8}$$

$$\Xi; \Delta \vdash_\sim X : A. \tag{9}$$

Because ψ is a sequential grafting of Ξ, and by Lemma 42,

$$\Xi\psi; \Gamma\psi \vdash\sim M_1\psi : B\psi\,, \tag{10}$$

$$\Xi\psi; \Gamma\psi \vdash\sim M_2\psi : B\psi\,, \tag{11}$$

$$\Xi\psi; \Delta\psi \vdash\sim X\psi : A\psi \tag{12}$$

By Lemma 17,

$$(\Xi\psi)\downarrow_{\lambda\Pi_{\mathscr{D}}}; \Gamma\psi \vdash\sim M_1\psi : B\psi\,, \tag{13}$$

$$(\Xi\psi)\downarrow_{\lambda\Pi_{\mathscr{D}}}; \Gamma\psi \vdash\sim M_2\psi : B\psi\,, \tag{14}$$

$$(\Xi\psi)\downarrow_{\lambda\Pi_{\mathscr{D}}}; \Delta\psi \vdash\sim X\psi : B\psi \tag{15}$$

By Definition 38, $\Gamma\psi = \tilde{\psi}(\Gamma)$, $\Delta\psi = \tilde{\psi}(\Delta)$, $M_1\psi = \tilde{\psi}(M_1)$, and $M_2\psi = \tilde{\psi}(M_2)$. Since ψ is a derivation of Ξ, $(\Xi\psi)\downarrow_{\lambda\Pi_{\mathscr{D}}} = \varepsilon$. Thus, $M_1 \simeq_\Gamma M_2$ is not in $(\Xi\psi)\downarrow_{\lambda\Pi_{\mathscr{D}}}$. Hence, $(M_1\psi)\downarrow_{\lambda\Pi_{\mathscr{D}}}$ and $(M_2\psi)\downarrow_{\lambda\Pi_{\mathscr{D}}}$ are identical ground terms (otherwise the constraint could not be discharged). Therefore, $\tilde{\psi}$ is a solution to Ξ. □

Lemma 43. *For all valid constrained signature Ξ, ψ is a sequential grafting of Ξ if and only if ψ is a sequential grafting of $(\Xi)\downarrow_{\lambda\Pi_{\mathscr{D}}}$.*

Proof. By induction on the length of ψ. If ψ is the empty list, then the conclusion is trivial by Definition 35. Otherwise, we use the induction hypothesis, and Lemma 27. □

Lemma 44. *For all valid constrained signature Ξ, if ψ is a sequential grafting of $(\Xi)\downarrow_{\lambda\Pi_{\mathscr{D}}}$, then $((\Xi)\downarrow_{\lambda\Pi_{\mathscr{D}}}\psi)\downarrow_{\lambda\Pi_{\mathscr{D}}} = (\Xi\psi)\downarrow_{\lambda\Pi_{\mathscr{D}}}$.*

Proof. By induction on the length of ψ. The base case is trivial. At the induction step we use equational reasoning on $\lambda\Pi_{\mathscr{D}}$. □

Proposition 40 states that for all $n \geqslant 0$, if $\Xi_1 \overset{\theta_1}{\longmapsto} \Xi_2 \overset{\theta_2}{\longmapsto} \cdots \overset{\theta_{n-1}}{\longmapsto} \Xi_n$ is a path of a search tree of a valid constrained signature Ξ such that $\Xi_1 = (\Xi)\downarrow_{\lambda\Pi_{\mathscr{D}}}$, the list of graftings $\psi = \langle\theta_1, \ldots, \theta_{n-1}\rangle$ is a sequential grafting of Ξ, and for $0 < i \leqslant n$, $\Xi_i = (\Xi\psi)\downarrow_{\lambda\Pi_{\mathscr{D}}}$.

Proof of Proposition 40. By induction on n. The base case is trivial. Assume that $n > 0$ and take $\psi' = \langle\theta_2, \ldots, \theta_i\rangle$. By construction, θ_1 is an elementary grafting of a metavariable in Ξ_1. Thus, by Theorem 32, θ_1 is a valid grafting of Ξ_1 and $\Xi_2 = (\Xi_1\theta_1)\downarrow_{\lambda\Pi_{\mathscr{D}}}$ is well-defined. By induction hypothesis, ψ' is a sequential grafting of $\Xi_1\theta_1$, and $\Xi_i = (\Xi_1\langle\theta_1\psi'\rangle)\downarrow_{\lambda\Pi_{\mathscr{D}}} = (\Xi_1\psi)\downarrow_{\lambda\Pi_{\mathscr{D}}}$. By Definition 35, ψ is a sequential grafting of $\Xi_1 = (\Xi)\downarrow_{\lambda\Pi_{\mathscr{D}}}$. Therefore, by Lemma 43, ψ is a sequential grafting of Ξ, and by Lemma 44, $\Xi_i = (\Xi\psi)\downarrow_{\lambda\Pi_{\mathscr{D}}}$. □

4.2. Completeness

The completeness property states that if there is a solution Ψ to a constraint signature Ξ, there exists a derivation ψ of Ξ, such that $\tilde{\psi} \equiv_{\lambda\Pi_{\mathscr{L}}} \Psi$. This claim is proved by induction on the size of Ψ.

Definition 45 (*Size of a pure term*). The size of a pure term is defined by induction over the structure of terms as follows.

- $|s| = 1$, if $s \in \{Kind, Type\}$.
- $|\underline{n}| = 1$.
- $|(M\ N)| = |M| + |N| + 1$.
- $|\lambda_A.M| = |A| + |M| + 1$.
- $|\Pi_A.B| = |A| + |B| + 1$.

Definition 46 (*Size of a parallel instantiation*). Let Ψ be a parallel instantiation of a constrained signature Ξ, the size of Ψ, denoted by $|\Psi|$, is the sum of the sizes of $\Psi(X)$ for all X in Ξ.

Lemma 47. *Let Ξ be a valid constrained signature in $\lambda\Pi_{\mathscr{L}}$-normal form, if Ψ is a normal solution of Ξ, then there exists a search tree of Ξ with a derivation ψ, such that $\tilde{\psi} \equiv_{\lambda\Pi_{\mathscr{L}}} \Psi$.*

Proof. By induction on the size of Ψ_Ξ.[5] Since Ψ_Ξ is a solution to Ξ, Ξ is not a failure signature. If $\Xi = \varepsilon$, the empty list is a derivation of Ξ. Otherwise, take the first meta-variable declared in Ξ, namely $X :_\Gamma A$. This meta-variable exists by Lemma 28. Notice that A and Γ do not depend on any other meta-variable or constraint. We reason by case analysis on $M = \Psi_\Xi(X)$. Since Ξ is a constrained signature in $\lambda\Pi_{\mathscr{L}}$-normal form and Ψ is a normal solution, M, A, Γ are ground $\lambda\Pi_{\mathscr{L}}$-normal forms.

- $M = Type$. In this case, $A = Kind$. Consider the elementary grafting of X, $\theta = \{X/_\varepsilon Type\}$. Let $\Xi_1 = (\Xi\theta)\downarrow_{\lambda\Pi_{\mathscr{L}}}$, Ξ_1 is well-defined by Lemma 26 and Theorem 32. We check that $\Psi'_{\Xi_1}(X) = \Psi_\Xi(X)$, $X \in \Xi_1$, is a normal solution of Ξ_1, and that $|\Psi'_{\Xi_1}| < |\Psi_\Xi|$.
- $M = \Pi_{A_1}.A_2$. In this case, $A \in \{Kind, Type\}$ and $\Gamma \vdash A_1 : s$, $s \in \{Kind, Type\}$. Consider the elementary grafting of X, $\theta = \{X/_{\Xi'} \Pi_Z.Y\}$, where Z, Y are fresh meta-variables, and $\Xi' = Y :_{Z.\,\Gamma} A.\ Z :_\Gamma s$. Let $\Xi_1 = (\Xi\theta)\downarrow_{\lambda\Pi_{\mathscr{L}}}$. We check that

$$\Psi'_{\Xi_1}(W) = \begin{cases} A_1 & \text{if } W = Z, \\ A_2 & \text{if } W = Y, \\ \Psi_\Xi(W) & \text{otherwise} \end{cases}$$

  is a normal solution of Ξ_1, and that $|\Psi'_{\Xi_1}| < |\Psi_\Xi|$.

[5] In this proof, the index Ξ of Ψ is relevant.

- $M=\lambda_{A_1}.N$. In this case, $A\in\Pi_{A_1}.A_2$ and $\Gamma\vdash N:A_2$. Consider the elementary grafting of X, $\theta=\{X/_{\Xi'}\lambda_{A_1}.Y\}$, where Y is a fresh meta-variable, and $\Xi'=Y:_{A_1.\ \Gamma}A_2$. Let $\Xi_1=(\Xi\theta)\downarrow_{\lambda\Pi_{\mathscr{S}}}$. We check that

$$\Psi'_{\Xi_1}(W)=\begin{cases}A_2 & \text{if } W=Y,\\ \Psi_\Xi(W) & \text{otherwise}\end{cases}$$

  is a normal solution of Ξ_1, and that $|\Psi'_{\Xi_1}|<|\Psi_\Xi|$.
- $M=(\underline{n}\ M_1\ \ldots\ M_i)$. In this case, $\Gamma\vdash\underline{n}:B$, B (in $\lambda\Pi_{\mathscr{S}}$-normal form) is a product, $\Gamma\vdash A:s$, and $s\in\{Kind,Type\}$. Consider the elementary grafting of X, $\theta=\{X/_{A\simeq_\Gamma A'.\ \Xi'}(\underline{n}\,X_1\ \ldots\ X_i)\}$ where $\Xi';\Gamma\vdash(\underline{n}\ X_1\ \ldots\ X_i):A'$ is in $[\Gamma\vdash\underline{n}:B]^i$. Let $\Xi_1=(\Xi\theta)\downarrow_{\lambda\Pi_{\mathscr{S}}}$. We check that

$$\Psi'_{\Xi_1}(W)=\begin{cases}M_j & \text{if } W=X_j,\ 0<j\leqslant i,\\ \Psi_\Xi(W) & \text{otherwise}\end{cases}$$

  is a normal solution of Ξ_1, and that $|\Psi'_{\Xi_1}|<|\Psi_\Xi|$.

In all the cases $|\Psi'_{\Xi_1}|<|\Psi_\Xi|$, then by induction hypothesis, there exists a search tree of Ξ_1 with a derivation ψ_1, such that $\widetilde{\psi_1}\equiv_{\lambda\Pi_{\mathscr{S}}}\Psi'_{\Xi_1}$. Then, $\psi=\langle\theta,\psi_1\rangle$ is a derivation of Ξ. Since $\Psi_\Xi(X)=\Psi'_{\Xi_1}(X\theta)$, for all $X\in\Xi$, $\Psi_\Xi(X)\equiv_{\lambda\Pi_{\mathscr{S}}}X\langle\theta,\psi_1\rangle=X\psi$. Therefore, $\tilde{\psi}\equiv_{\lambda\Pi_{\mathscr{S}}}\Psi_\Xi$. □

Theorem 48 (Completeness). *Let Ξ be a valid constrained signature, if Ψ is a solution of Ξ, then there exists a search tree of Ξ with a derivation ψ, such that $\tilde{\psi}\equiv_{\lambda\Pi_{\mathscr{S}}}\Psi$.*

Proof. If Ψ is a solution of Ξ, by Lemma 17 and Definition 19, Ψ is a solution of $(\Xi)\downarrow_{\lambda\Pi_{\mathscr{S}}}$ too. By Remark 21, the parallel instantiation $\Psi'(X)=(\Psi(X))\downarrow_{\lambda\Pi_{\mathscr{S}}}$, $X\in\Xi$, is a normal solution of $(\Xi)\downarrow_{\lambda\Pi_{\mathscr{S}}}$. Hence, by Lemma 47, there exists a search tree of $(\Xi)\downarrow_{\lambda\Pi_{\mathscr{S}}}$ with a derivation ψ, such that $\tilde{\psi}\equiv_{\lambda\Pi_{\mathscr{S}}}\Psi'$. Therefore, $\tilde{\psi}\equiv_{\lambda\Pi_{\mathscr{S}}}\Psi$. By Definition 33, a search tree of Ξ is a search tree of $(\Xi)\downarrow_{\lambda\Pi_{\mathscr{S}}}$. □

5. Related work and summary

Automatic proof synthesis is at the basis of proof assistant systems. A complete method for search of proof-trees based on resolution and unification was formulated by Robinson [37] for the first-order logic, and by Huet [21] for the higher-order logic. In type systems, higher-order unification (HOU) algorithms are known for the simply-typed λ-calculus [22] and for the λP-calculus of dependent types [17,35].

For the cube-type systems, Dowek [12, 13] reformulates the unification procedure and generalizes it as a method of term enumeration. Recently, Cornes [10] proposed an extension of Dowek's method to the Calculus of Constructions with Inductive Types.

Dowek et al. [15] propose a first-order presentation of Huet's HOU algorithm based on explicit substitutions and typed meta-variables. That algorithm is generalized to solve

higher-order equational unification by Kirchner and Ringeissen [25], and restricted to the case of higher-order patterns by Dowek, Hardin, Kirchner, and Pfenning in [16]. The algorithm for pattern unification via explicit substitutions has been extended (without proof) to dependent types, and implemented in the Twelf system [34].

On the other hand, Briaud [7] shows how HOU can be considered as a typed narrowing in the $\lambda\upsilon$-calculus of explicit substitutions. Magnusson [28] presents a unification algorithm in Martin-Löf's type theory with explicit substitutions. That algorithm solves first-order unification problems, but leaves unsolved the flexible-flexible constraints.

Our main contribution is the presentation of Dowek's method of proof synthesis in a suitable theory with explicit substitutions and meta-variables. This way, proof-terms can be built incrementally as the proofs are done, and each construction step is guaranteed by the type system.

Just as in [12, 13], the method presented here is sound and complete. Thus, it can be seen as a semi-algorithm for ground higher-order unification in λP and the Calculus of Constructions. Although the implementation issues are out of the scope of this paper, a preliminary version of our method has been implemented in OCaml, and it is electronically available by contacting the author.

The underlying calculus of the method proposed here is $\lambda\Pi_{\mathscr{L}}$. We believe that the same ideas can be applied to other formalisms satisfying at least the same typing properties as $\lambda\Pi_{\mathscr{L}}$, that is, confluence, weak-normalization, subject reduction, and instantiation lemma. Nevertheless, we remark that the $\lambda\Pi_{\mathscr{L}}$-calculus has some features that are useful for our proof-synthesis method and they seem to be in unification issues:

- It is a finite first-order rewriting system. In particular, some properties as soundness and completeness of the method are much simpler to prove.
- It uses general composition of substitutions and simultaneous substitutions. In [33], we discuss efficiency improvements to the method based on these features.
- Since substitutions distribute under abstractions and products, normal forms have a simple characterization. For example, the normal form of a type has the form $\Pi_{A_1} \ldots \Pi_{A_i}.A$ where A is not a product.

Finally, notice that $\lambda\Pi_{\mathscr{L}}$ does not handle the η-rule. Extensional versions of explicit substitution calculi have been studied for ground terms [24]. However, work is necessary to understand the interaction with dependent types and meta-variables.

Acknowledgements

Many persons have contributed to this work with useful remarks and suggestions, in particular Gilles Dowek, Delia Kesner, Nikolaj Bjørner, and the anonymous referees. The author is very grateful to them.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, J.-J. Lévy, Explicit substitution, J. Funct. Programming 1 (4) (1991) 375–416.

[2] T. Altenkirch, V. Gaspes, B. Nordström, B. von Sydow, A User's Guide to ALF, Chalmers University of Technology, Sweden, May 1994.
[3] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, B. Werner, The Coq Proof Assistant Reference Manual – Version V6.1, Tech. Rep. 0203, INRIA, August 1997.
[4] M. Bezem, J. Springintveld, A simple proof of the undecidability of inhabitation in λP, J. Funct. Programming 6 (5) (1996) 757–761.
[5] R. Bloo, Preservation of termination for explicit substitution, Ph.D. Thesis, Eindhoven University of Technology, 1997.
[6] R. Bloo, K.H. Rose, Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection, Proc. CSN-95: Computer Science in the Netherlands, November 1995.
[7] D. Briaud, Higher order unification as a typed narrowing, Tech. Rep., CRIN Report 96-R-112, 1996.
[8] T. Coquand, Une Théorie de Constructions, Thèse de doctorat, Université Paris 7, 1985.
[9] T. Coquand, G. Huet, The Calculus of Constructions, Inform. and Comput. 76 (1988) 96–120.
[10] C. Cornes, Conception d'un langage de haut niveau de représentation de preuves: récurrence par filtrage de motifs, unification en présence de types inductif primitifs. synthèse de lemmes d'inversion, Thèse de doctorat, Université Paris 7, 1997.
[11] P.-L. Curien, T. Hardin, J.-J. Lévy, Confluence properties of weak and strong calculi of explicit substitutions, J. ACM 43 (2) (1996) 362–397.
[12] G. Dowek, Démonstration automatique dans le calcul des constructions, Thèse de doctorat, Université Paris 7, 1991.
[13] G. Dowek, A complete proof synthesis method for type systems of the cube, J. Logic Comput. 3 (3) (1993) 287–315.
[14] G. Dowek, Automated theorem proving in first-order logic modulo: On the difference between type theory and set theory, First-order Theorem Proving '98, 1998, pp. 1–21.
[15] G. Dowek, T. Hardin, C. Kirchner, Higher-order unification via explicit substitutions (extended abstract), Proc. 10th Annual IEEE Symp. on Logic in Computer Science, San Diego, California, IEEE Computer Society Press, Silverspring, MD, 26–29 June 1995, pp. 366–374.
[16] G. Dowek, T. Hardin, C. Kirchner, F. Pfenning, Unification via explicit substitutions: the case of higher-order patterns, in: M. Maher (Ed.), Proc. Joint Internat. Conf. Symp. on Logic Programming, Bonn, Germany, MIT Press, Cambridge, MA, September 1996, pp. 259–273.
[17] C. Elliott, Higher-order unification with dependent types, in: N. Dershowitz (Ed.), Proc. Internat. Conf. on Rewriting Techniques and Applications (RTA-89), Lecture Notes in Computer Science, vol. 355, Chapel Hill, North Carolina, Springer, Berlin, April 1989, pp. 121–136.
[18] M.C.F. Ferreira, D. Kesner, L. Puel, Lambda-calculi with explicit substitutions and composition which preserve beta-strong normalization, in: M. Hanus, M. Rodríguez-Artalejo (Eds.), Algebraic and Logic Programming, 5th Internat. Conf., ALP'96, Lecture Notes in Computer Science, vol. 1139, Aachen, Germany, Springer, Berlin, 25–27 September 1996, pp. 284–298.
[19] J.-Y. Girard, Interprétation Fonctionelle et Élimination des Compures de l'Arithmétic d'Ordre Supérieur, Thèse de doctorat, Université Paris 7, 1972.
[20] R. Harper, F. Honsell, G. Plotkin, A framework for defining logics, J. Assoc. Comput. Mach. 40 (1) (1993) 143–184.
[21] G. Huet, Constrained Resolution A Complete Method for Higher Order Logic, Ph.D. Thesis, Case Western Reserve University, 1972.
[22] G. Huet, A unification algorithm for typed lambda calculus, Theoret. Comput. Sci. 1 (1) (1975) 27–57.
[23] F. Kamareddine, A. Ríos, A Lambda-Calculus à la De Bruijn with Explicit Substitutions, Lecture Notes in Computer Science, vol. 982, Springer, Berlin, 1995, pp. 45–62.
[24] D. Kesner, Confluence properties of extensional and non-extensional λ-calculi with explicit substitutions (extended abstract), in: H. Ganzinger (Ed.), Proc. 7th Internat. Conf. on Rewriting Techniques and Applications (RTA-96), Lecture Notes in Computer Science, vol. 1103, New Brunswick, New Jersey, Springer, Berlin, 1996, pp. 184–199.
[25] C. Kirchner, C. Ringeissen, Higher order equational unification via explicit substitutions, Proc. Internat. Conf. PLILP/ALP/HOA'97, Lecture Notes in Computer Science, vol. 1298, Southampton, England, Springer, Berlin, September 1997.

[26] P. Lescanne, From $\lambda\sigma$ to $\lambda\upsilon$ a journey through calculi of explicit substitutions, Proc. 21st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, January 1994, pp. 60–69.
[27] P. Lescanne, J. Rouyer-Degli, Explicit substitutions with de Bruijn's levels, in: J. Hsiang (Ed.), Proc. Internat. Conf. on Rewriting Techniques and Applications (RTA-95), Lecture Notes in Computer Science, vol. 914, Chapel Hill, North Carolina, Springer, Berlin, 1995, pp. 294–308.
[28] L. Magnusson, The Implementation of ALF-a proof editor based on Martin-Löf's monomorphic type theory with explicit substitution, Ph.D. Thesis, Chalmers University of Technology and Göteborg University, January 1995.
[29] D. Miller, Unification under a mixed prefix, J. Symbolic Comput. 14 (4) (1992) 321–358.
[30] C. Muñoz, Confluence and preservation of strong normalisation in an explicit substitutions calculus (extended abstract), Proc. 11th Annual IEEE Symp. on Logic in Computer Science, New Brunswick, New Jersey, IEEE Computer Society Press, Silverspring, MD, July 1996, pp. 440–447.
[31] C. Muñoz, Dependent types with explicit substitutions: A meta-theoretical development, Types for Proofs and Programs, Proc. Internat. Workshop TYPES'96, Lecture Notes in Computer Science, vol. 1512, Aussois, France, Springer, Berlin, 1997, pp. 294–316.
[32] C. Muñoz, A left-linear variant of $\lambda\sigma$, Proc. Internat. Conf. PLILP/ALP/HOA'97, Lecture Notes in Computer Science, vol. 1298, Southampton, England, Springer, Berlin, September 1997, pp. 224–234.
[33] C. Muñoz, Un calcul de substitutions pour la représentation de preuves partielles en théorie de types, Thèse de doctorat, Université Paris 7, 1997. English version available as INRIA research report RR-3309.
[34] F. Pfenning, C. Schürmann, Twelf user's guide, 1.2. Edition, Tech. Rep. CMU-CS-1998-173, Carnegie Mellon University, September 1998.
[35] D. Pym, A unification algorithm for the $\lambda\Pi$-calculus, Internat. J. Found. Comput. Sci. 3 (3) (1992) 333–378.
[36] A. Ríos, Contributions à l'étude de λ-calculs avec des substitutions explicites, Thèse de doctorat, Université Paris 7, 1993.
[37] J.A. Robinson, A machine-oriented logic based on the resolution principle, J. ACM 12 (1) (1965) 23–41.
[38] M. Zaionc, Mechanical procedure for proof construction via closed terms in typed λ calculus, J. Automat. Reason. 4 (2) (1988) 173–190.
[39] H. Zantema, Termination of term rewriting by semantic labelling, Fund. Inform. 24 (1995) 89–105.