École Normale Supérieure

Département d'Informatique

Rapport de Stage de M1

Checking the type safety of rewrite rules in $\lambda\Pi$-calculus modulo rewriting

Wu Jui-Hsuan

supervised by

Frédéric Blanqui[12] & Valentin Blot[12]

[1]INRIA

[2]ENS Paris-Saclay

11 March 2019 – 31 July 2019

1 Introduction

Lambdapi is a new proof assistant based on the $\lambda\Pi$-calculus modulo theory [5], which extends the simply typed lambda calculus with dependent types and an equivalence relation on types generated by user-defined rewrite rules. The expressiveness of rewriting allows to formalize proofs that cannot be done in other proof assistants. However, as Lambdapi is based on the propositions-as-types interpretation, one should verify that the system satisfies the subject reduction property before starting to construct proofs within it. During my internship, I studied a new algorithm proposed by F. Blanqui for checking type preservation of rewrite rules and implemented it in Lambdapi. Besides, in order to improve the unification (modulo rewriting) procedure, I designed an algorithm for checking injectivity of function symbols.

2 $\lambda\Pi$-calculus modulo

2.1 $\lambda\Pi$-calculus

The $\lambda\Pi$-calculus is an extension of the simply typed lambda calculus ($\lambda \rightarrow$) with dependent types. In this calculus, we introduce a particular type $Type$ and "types" in $\lambda \rightarrow$ are just terms of type $Type$ in the $\lambda\Pi$-calculus. Some terms have the type $A \rightarrow Type$, where A is of type $Type$, and can be applied to a term t of type A to build a type depending on t. Besides, a type $Kind$ is introduced as the type of terms $Type$, $A \rightarrow Type$, etc. Finally, the usual arrow $A \rightarrow B$ is extended to the dependent product $\Pi x : A.B$ to handle the possible dependency of the type B on the type A.

We write $\mathcal{S}$ for the set of sorts $\{Type, Kind\}$ and we assume given a set $\mathcal{X} = \{x, y, z, \cdots\}$ of variables and a set $\mathcal{F}$ of function symbols.

Then we can define the set $\mathcal{T}$ of terms of the $\lambda\Pi$-calculus.

Definition 2.1. (Term) The set of terms in the $\lambda\Pi$-calculus is given by:

$$t := x \mid s \mid f \mid tt \mid \lambda x : t.t \mid \Pi x : t.t, \text{ where } x \in \mathcal{X}, s \in \mathcal{S} \text{ and } f \in \mathcal{F}$$

We then assume that each function symbol f is equipped with a type $\tau(f)$ and a sort $s(f)$. If $\tau(f) = \Pi x_1 : T_1.\cdots.\Pi x_n : T_n.U$ where U is not a product, then f is said of arity n. In fact, function symbols can be regarded as variables declared in a global context ("signature") of the system.

We now present the typing rules of the $\lambda\Pi$-calculus:

Well-formedness of the empty context

$$\frac{}{[\,] \text{ well-formed}}$$

Declaration of a variable

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \text{ well-formed}} \; s \in \mathcal{S}$$

$Type$ is of type $Kind$

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash Type : Kind}$$

Variable

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash x : A} \; x : A \in \Gamma$$

Product

$$\frac{\Gamma \vdash A : Type \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A.B : s} \; s \in \mathcal{S}$$

Abstraction

$$\frac{\Gamma \vdash A : Type \qquad \Gamma, x : A \vdash B : s \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : \Pi x : A.B} \; s \in \mathcal{S}$$

Application

$$\frac{\Gamma \vdash t : \Pi x : A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : (u/x)B}$$

Function symbols

$$\frac{\vdash \tau(f) : s(f)}{\vdash f : \tau(f)}$$

Conversion

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash t : B} \; s \in \mathcal{S}, A \equiv_\beta B$$

Fig. 1. Typing rules of the λΠ-calculus.

We have the uniqueness (modulo $\equiv_\beta$) of types in λΠ-calculus.

Proposition 2.1. For all Γ, t, A, B, if $\Gamma \vdash t : A$ and $\Gamma \vdash t : B$, then $A \equiv_\beta B$.

2.2 λΠ-calculus modulo rewriting

The λΠ-calculus modulo rewriting is an extension of the λΠ-calculus with a set R of rewrite rules. Let $\equiv$ be the minimal congruence that contains R and β-conversion. The typing rules of the λΠ-calculus modulo R can be obtained from those of the λΠ-calculus by replacing $\equiv_\beta$ with $\equiv$ in the conversion rule.

Notation 2.1. In the following, we denote the smallest congruence containing a relation S by $\equiv_S$.

2.3 Lambdapi

2.3.1 Metavariables

In Lambdapi, we extend the λΠ-calculus modulo by introducing metavariables which are used to represent yet unknown term. We denote by $\mathcal{M}$ the set of metavariables.

$$t \coloneqq \cdots \mid M \text{ where } M \in \mathcal{M}$$

We now define the types of metavariables and the algebraic fragment of Lambdapi.

Each metavariable M is equipped with a type t_M, with some contraints explained below.

The type of a yet unknown object-level term is also yet unknown, but the type of this type should be a sort. We distinguish thus object-level metavariables and type-level metavariables:

Definition 2.2.

- A type-level metavariable is a metavariable M s.t. $t_M \in \mathcal{S}$.

- An object-level metavariable is a metavariable M s.t. t_M is a type-level metavariable.
- A metavariable is either a type-level metavariable or an object-level metavariable.
- We denote by M_0 the set of object-level metavariables and by $Meta(t)$ the set of object-level metavariables occurring in t.

The algebraic fragment of the terms of Lambdapi corresponds to the terms of first-order rewriting:

Definition 2.3. Let A_0 be the smallest set of terms such that $M_0 \subseteq A_0$, and $fa_1 \cdots a_n \in A_0$ for all $f \in \mathcal{F}, a_1, \cdots, a_n \in A_0$. The set $\mathcal{A}$ of algebraic terms is defined as $A_0 - \mathcal{M}_0$

We now define *substitutions*.

Definition 2.4. (substitutions) A substitution σ is a map assigning to each metavariable M a term. Its application to a term is defined as follows:

- $x\sigma = x$ for $x \in \mathcal{X}$
- $f\sigma = f$ for $f \in \mathcal{F}$
- $(tu)\sigma = (t\sigma)(u\sigma)$
- $(\lambda x : A.t)\sigma = \lambda x : A\sigma.t\sigma$
- $(\Pi x : A.t)\sigma = \Pi x : A\sigma.t\sigma$
- $M\sigma = \sigma(M)$
- We define the domain $dom(\sigma)$ of a substitution σ as the set $\{M \in \mathcal{M} \mid M\sigma \neq M\}$.

2.3.2 Type inference and type checking

The type inference relation of Lambdapi is the same as that of the $\lambda\Pi$-calculus modulo rewriting. Note that, if a term contains a metavariable, then it is not typable.

Since metavariables are used to represent unknown terms in a term t, one might want to infer the constraints on terms replacing the metavariables of t that need to be satisfied to make the term t typable.

We now present the constraint-generating type inference and type checking algorithm for algebraic terms, consisting of the following judgement forms.

$$\Gamma \vdash t \uparrow A[E] \qquad \text{type checking}$$
$$\Gamma \vdash t \downarrow A[E] \qquad \text{type inference}$$

Note that we add a special field E which contains a set of constraints.

Function applications

$$\frac{\tau(f) = \Pi x_1 : A_1. \cdots .\Pi x_n : A_n.U \qquad \forall i \in \{1, \cdots, n\}, \vdash t_i \uparrow A_i[E_i]}{\vdash ft_1 \cdots t_n \downarrow (t_1/x_1, \cdots, t_n/x_n)U[\bigcup_{1 \leq i \leq n} E_i]}$$

Object-level metavariables

$$\frac{}{\vdash M \uparrow T[\{(t_M, T)\}]}$$

Type checking of function applications

$$\frac{\vdash f t_1 \cdots t_n \downarrow A[E]}{\vdash f t_1 \cdots t_n \uparrow B[E \cup \{(A, B)\}]}$$

Fig. 2. (Constraint-generating) type inference and type checking algorithm for algebraic terms.

Proposition 2.2.

1. If t is an algebraic term, then there exists at most one (E, A) such that $\vdash t \downarrow A[E]$.
2. If t is an algebraic term or an object-level metavariable, then for all term A, there exists at most one E such that $\vdash t \uparrow A[E]$.

Remark 2.1. The proposition above guarantees the existence of two algorithms $InferType$ and $CheckType$ that correspond to the type inference and the type checking respectively.

Definition 2.5. We say that a substitution σ satisfies a set E of equations, written $\sigma \models E$, if for all $(a, b) \in E, a\sigma \equiv b\sigma$.

Definition 2.6. We say that a substitution σ' is a type-level extension of another substitution σ if $M\sigma' = M\sigma$ for all $M \in dom(\sigma)$ and $dom(\sigma') - dom(\sigma) \subseteq \{t_M \mid M \in M_0 \cap dom(\sigma)\}$.

Theorem 2.1. Suppose that $\vdash t \uparrow T[E]$ (or $\vdash t \downarrow T[E]$). Let σ be a substitution. Then we have:

1. $(\sigma \models E$ and $\forall M \in Meta(t), \Gamma \vdash M\sigma : t_M\sigma) \Rightarrow \Gamma \vdash t\sigma : T\sigma$.
2. $\Gamma \vdash t\sigma : V \Rightarrow$ there exists a type-level extension σ' of σ s.t. $V \equiv T\sigma', \sigma' \models E$ and $\forall M \in Meta(t), \Gamma \vdash M\sigma' : t_M\sigma')$.

Proof. By induction on the derivation of $\vdash t \downarrow T[E]$ (or $\vdash t \uparrow T[E]$)

- (Function applications)
  1. Suppose that $\sigma \models \bigcup_{1 \le i \le n} E_i$ and $\forall M \in Meta(f t_1 \cdots t_n), \Gamma \vdash M\sigma : t_M\sigma$. We have, $\forall 1 \le i \le n$, $\sigma \models E_i$ and $\forall M \in Meta(t_i), \Gamma \vdash M\sigma : t_M\sigma$. By I.H., $\forall i \in \{1, \cdots, n\}, \Gamma \vdash t_i\sigma : A_i\sigma = A_i$ (the A_i's and U contain no metavariable since function symbols are declared in the global context). Hence, $\Gamma \vdash (f t_1 \cdots t_n)\sigma = f(t_1\sigma) \cdots (t_n\sigma) : (t_1\sigma/x_1, \cdots, t_n\sigma/x_n)U = ((t_1/x_1, \cdots, t_n/x_n)U)\sigma$.
  2. Suppose that $\Gamma \vdash f(t_1\sigma) \cdots (t_n\sigma) : V$. We have $\forall i, \Gamma \vdash t_i\sigma : A_i$. Thus by I.H., we have, for all i, there exists a type-level extension σ'_i of σ s.t. $\sigma'_i \models E_i$ and $\forall M \in Meta(t_i), \Gamma \vdash M\sigma' : t_M\sigma'$. Let σ' be the substitution defined by $dom(\sigma') = \bigcup_{1 \le i \le n} dom(\sigma'_i)$ and $M\sigma' = M\sigma'_i$. Note that, if an object-level metavariable M occurs in the domains of σ'_i and σ'_j, we have, by I.H., $\Gamma \vdash M\sigma'_i : t_M\sigma'_i$ and $\Gamma \vdash M\sigma'_j : t_M\sigma'_j$. σ'_i (resp. σ'_j) is a type-level extension of σ, we have thus $M\sigma'_i = M\sigma = M\sigma'_j$. Hence, $t_M\sigma'_i \equiv t_M\sigma'_j$ by uniqueness of types. We choose thus the value of $t_M\sigma'$ to be either $t_M\sigma'_i$ or $t_M\sigma'_j$. By doing so, $\sigma' \models \bigcup_{1 \le i \le n} E_i$ and $\forall M \in Meta(f t_1 \cdots t_n), \Gamma \vdash M\sigma' : t_M\sigma'$. To finish, we have $V = (t_1\sigma/x_1, \cdots, t_n\sigma/x_n)U = ((t_1/x_1, \cdots, t_n/x_n)U)\sigma = ((t_1/x_1, \cdots, t_n/x_n)U)\sigma'$.
- (Object-level metavariables)
  1. Suppose that $\sigma \models \{(t_M, T)\}$, i.e., $t_M\sigma \equiv T\sigma$ and $\Gamma \vdash M\sigma : t_M\sigma$. By the conversion rule, we have thus $\Gamma \vdash M\sigma : T\sigma$.
  2. Suppose that $\Gamma \vdash M\sigma : V$. By defining σ' as $N\sigma' = M\sigma$ if $N \neq t_M$ and $t_M\sigma' = V$, we have the required properties.
- (Type checking of function applications) Trivial.

□

2.3.3 Rewrite rules

We only consider algebraic rewrite rules in the algorithm proposed in the next section.

Definition 2.7. (Algebraic rewrite rules) An algebraic rewrite rule is a pair (l, r) of terms, written as $l \to r$, such that:

1. l is algebraic;
2. r is either an object-level metavariable or an algebraic term;
3. $Meta(r) \subseteq Meta(l)$;

Now we can define the rewrite relation induced by the rewrite system.

Definition 2.8. (Rewrites and rewrite steps)

- Let $l \to r$ be an algebraic rewrite rule and σ be a substitution. Then $l\sigma \to r\sigma$ is called a rewrite and $l\sigma$ is called a redex.
- Let C be a context and $l\sigma \to r\sigma$ be a rewrite. Then $C[l\sigma] \to C[r\sigma]$ is a rewrite step. This defines the rewrite relation generated by the rewrite system.

3 Subject reduction in $\lambda\Pi$-calculus modulo

We first give the notion of product compatibility.

Definition 3.1. (Product compatibility) We say the product compatibility property is satisfied if $\Pi x : A.B \equiv \Pi x : A'.B'$ implies $A \equiv A'$ and $B \equiv B'$.

Then we give a sufficient condition for the product compatibility.

Proposition 3.1. (Product compatibility from confluence) If $\to$ is confluent, then the product compatibility property holds.

Now we introduce the type preservation of rewrite rules, one of the main points we investigate in this work.

Definition 3.2. (Type preservation of rewrite rules) A rule $l \to r$ is type-preserving if for any well-formed context Γ, any substitution σ and any term T, $\Gamma \vdash l\sigma : T$ implies $\Gamma \vdash r\sigma : T$.

In [4], F. Barbanera, M. Fernández, and H. Geuvers give the following theorem:

Theorem 3.1. (Subject reduction for $\to_\beta$) The subject reduction for $\to_\beta$, expressed below, is a consequence of the product compatibility.
$SR_{\to_\beta} : \forall \Gamma, t_1, t_2, T, (\Gamma \vdash t_1 : T \text{ and } t_1 \to_\beta t_2) \Rightarrow \Gamma \vdash t_2 : T$.

Theorem 3.2. (Subject reduction for $\to_R$) The subject reduction for $\to_R$, expressed below, is equivalent to the type preservation of all the rewrite rules in R.
$SR_{\to_R} : \forall \Gamma, t_1, t_2, T, (\Gamma \vdash t_1 : T \text{ and } t_1 \to_R t_2) \Rightarrow \Gamma \vdash t_2 : T$.

This result remains valid in Lambdapi. Since we assume always the confluence in Lambdapi, the subject reduction for $\to_R$ is equivalent to the type preservation of the rewrite system.

3.1 Type preservation of rewrite rules

In this section, we present an algorithm for checking type preservation of algebraic rewrite rules.

We first give an example to show the idea of the algorithm briefly.

Example 3.1. Consider the rule $l = tail\ n\ (cons\ x\ p\ v) \to v = r$ with $N : Type, s : N \Rightarrow N, A : Type, V : N \Rightarrow Type, cons : \Pi x : A.\Pi n : N.V\ n \Rightarrow V\ (s\ n)$, and $tail : \Pi n : N.V\ (s\ n) \Rightarrow V\ n$.

The idea is to retrieve some information about σ from the typability of $l\sigma$ and then use this information to prove the typability of the $r\sigma$. For example, to make $tail\ (n\sigma)\ (cons\ (x\sigma)(p\sigma)(v\sigma))$ typable, we should ask the type of $n\sigma$ to be (convertible to) N.

We first infer the type of the LHS:
$\vdash tail\ n\ (cons\ x\ p\ v) \downarrow V\ n[\{(t_n, N), (t_p, N), (t_v, V\ p), (V\ (s\ p), V\ (s\ n))\}]$. We now check if the RHS has the same type $V\ n$: $\vdash v \uparrow V\ n[\{(t_v, V\ n)\}]$ (recall: t_M represents the type of the object-level metavariable M).
By using the equation $(t_v, V\ p)$, we can tranform the equation $(t_v, V\ n)$ to solve into the equation $(V\ p, V\ n)$. Since V and s are constants, it is possible to get the constraint (p, n) from the constraint $(V\ (s\ p), V\ (s\ n))$, and the equation $(V\ p, V\ n)$ can be thus solved by the constraint (p, n). This observation also shows that a refinement of constraints should be done before dealing with the equations to solve.

As the type inference and type checking algorithm is "constraint-generating", our main goal is to prove that the set E of constraints generated by the LHS of a rule is "stronger" than the one generated by the RHS (E_{to_solve}). More precisely, we want to prove that every substitution σ satisfying E satisfies E_{to_solve} as well. The problem is hard and might be undecidable. Here, we propose a partial solution using a completion procedure that gives a rewrite system R' equivalent to E', obtained from E by replacing each metavariable M with a fresh symbol c_M.

Algorithm 1: CheckSR

input : an algebraic rule $l \to r$
output: a boolean value b

1 $res \leftarrow true$
2 $(U, E) \leftarrow InferType(l)$
3 $E_{to\_solve} \leftarrow CheckType(U, r)$
4 $E' \leftarrow E\rho$
`/* ρ replaces every metavariable M with a fresh (constant) symbol c_M */`
5 $(E'_{fo}, E'_{not\_fo}) \leftarrow FO(E')$
`/* split the equations into one first-order part and the rest */`
6 $R' \leftarrow Completion(E'_{fo})$
7 $E'_{to\_solve} \leftarrow E_{to\_solve}\rho$
8 **for** $(t, u) \in E'_{to\_solve}$ **do**
9 $\quad res \leftarrow res \wedge (t \downarrow_{R \cup R' \cup \beta} u \vee Eq_{constr}(E'_{not\_fo}, R', t, u))$
10 **end**
11 return res

Here, $InferType(l)$ returns (U, E) if and only if $\vdash l \downarrow U[E]$ and $CheckType(U, r)$ returns E_{to_solve} if and only if $\vdash r \uparrow U[E_{to\_solve}]$.

We now describe the completion procedure employed in the form of a rewrite system.

$$S \cup \{g \to d, l[g] \to r\} \hookrightarrow \begin{cases} S \cup \{g \to d, l[d] \to r\}) & \text{if } l[d] > r \\ S \cup \{g \to d, r \to l[d]\}) & \text{if } l[d] < r \\ S \cup \{g \to d\} & \text{if } l[d] = r \end{cases}$$

where $>$ is a total reduction order on terms

Note that only closed first-order term rewrite systems are considered here.

Lemma 3.1. The system above is terminating.

By considering the multiset ordering $>_{mul}$, the sum of the sizes of all the terms (LHS's and RHS's) strictly decreases after applying a rewrite step.

Proposition 3.2. Let S be a normal term in the system above. Then S has no critical pair.

Proof. First note that any critical pair between two "closed" rules $g \to d$ and $l \to r$ is of the form $(d, c[r])$ or $(c[d], r)$ where c is a context. In both cases, one of the LHS's is a subterm of the other. Hence, if S has a critical pair, then S is not in normal form, which leads to a contradiction. □

Theorem 3.3. Let S be a rewrite system included in the reduction order $>$. Then the normal form $nf(S)$ of S is a confluent and terminating rewrite system. Moreover, $\equiv_{nf(S)} = \equiv_S$.

Proof. By Proposition 3.2, the normal form $nf(S)$ of S is locally confluent. It is not difficult to prove that it is also included in $>$, which implies its termination. By Newman's lemma, $nf(S)$ is confluent. To prove the equivalence of the systems, it suffices to prove that $S \hookrightarrow S' \Rightarrow \leftrightarrow_S^* = \leftrightarrow_{S'}^*$. □

The completion procedure takes a set E of equations as input, orients it following the reduction order $>$:

$$S = \{max_>(a, b), min_>(a, b) \mid (a, b) \in E, a \neq b\}$$

and returns the normal form of S.

A problem of modularity arises when we consider the union of the user-defined system, the *beta*-conversion, and the one generated by completion. To guarantee the conversion is decidable in the union , we want the latter to be confluent and terminating. In general, there is no guarantee that this property is verified. It might be interesting to restrict to some special cases. For instance, if the user-defined system is left-linear, confluent and there is no critical pair between the two systems, then the union is confluent. There are some works about the modularity of termination in first-order term rewriting, but it is still unknown whether these results can be generalized to higher-order cases.

Since the union might not be complete, we propose a simple function that strengthens the power of the algorithm using higher-order constraints.

Algorithm 2: Eq_{constr}

input : a set E of constraints, a rewrite system R', two terms t and u
output: a boolean value b

```
b ← false
for (v, w) ∈ E do
    b ← b ∨ (t ↓_{R∪R'∪β} v ∧ u ↓_{R∪R'} w) ∨ (t ↓_{R∪R'∪β} w ∧ u ↓_{R∪R'} v)
end
return b
```

We now give the main theorem of this section:

Theorem 3.4. If $CheckSR(l \to r)$ returns true, then the rule $l \to r$ is type-preserving.

Proof. Suppose that $\Gamma \vdash l\sigma : T$. We have $\vdash l \downarrow U[E]$ and by Theorem 2.1, there exists a type-level extension σ' of σ s.t. $T \equiv U\sigma'$, for all $(A, B) \in E$, $A\sigma' \equiv B\sigma'$, and $\forall M \in Meta(l), \Gamma \vdash M\sigma' : t_M\sigma'$. Note that, for all $(t, u) \in E_{to\_solve}, t\rho \equiv_{R\cup E\rho} u\rho$ (*) since $\equiv_{R'} = \equiv_{E'_{fo}}$ and $E'_{not\_fo} \subseteq E\rho$. Let ρ^{-1} be the inverse of ρ.

The essential point of the proof is the following: for all term a containing no metavariable, $(a\rho^{-1}\sigma') \downarrow = ((a \downarrow)\rho^{-1}\sigma') \downarrow$.
We proceed by structural induction on a. The only non-trivial case is the base case $a = c_M$ where M is a metavariable. $(c_M\rho^{-1}\sigma') \downarrow = (M\sigma') \downarrow = ((c_M \downarrow)\rho^{-1}\sigma') \downarrow$ since $c_M \downarrow = c_M$.

From the property above, we have: $a \equiv b \Rightarrow a\rho^{-1}\sigma' \equiv b\rho^{-1}\sigma'$.
Moreover, $E\rho\rho^{-1}\sigma' = E\sigma' \subseteq \equiv$ and thus for all $(a, b), a \equiv_{R\cup E\rho} b \Rightarrow a\rho^{-1}\sigma' \equiv b\rho^{-1}\sigma'$. Hence, by (*), for all $(t, u) \in E'_{to\_solve}, t\sigma' \equiv u\sigma'$ and $\forall M \in Meta(r), \Gamma \vdash M\sigma' : t_M\sigma'$ since $Meta(r) \subseteq Meta(l)$. Hence, $\Gamma \vdash r\sigma = r\sigma' : U\sigma' \equiv T$. The rule $l \to r$ is thus type-preserving. □

The theorem above guarantees the correctness of the algorithm but in practice, we want to retrieve more information from the constraints E in order to generate a "better" rewrite system R' that allows us to solve more equations in E'_{to_solve}. First note that we only consider the substitutions satisfying E, i.e., σ s.t. $\forall (a, b) \in E, a\sigma \equiv b\sigma$. Imagine that we now have a unification modulo rewriting algorithm that generates a most general unifier ρ when applied to E. We have, for all $x \in dom(\rho), x\rho \equiv x\sigma$ since ρ, which gives a common property of all these substitutions satisfying E. Here, unification is used as refinement of constraints and a possible approach is presented in the next section.

4 Unification modulo rewriting and injectivity of function symbols

In this section, we study the unification modulo rewriting procedure and the injectivity of function symbols. Our study here focuses on first-order term rewrite systems. We first describe a naive unification modulo procedure and give an example to show its limit, which justifies the study of injectivity of symbols. Next, we give an algorithm for checking the (partial) injectivity of symbols.

In the following, we assume given a complete (i.e. terminating and confluent) first-order term rewrite system R. Let $\to$ be the rewrite relation induced by R and $\equiv$ the reflexive, symmetric, and transitive closure of $\to$.

4.1 Unification modulo rewriting

The unification algorithm presented later takes a unification problem as an input.

Definition 4.1. (Unification problems) A unification problem is a record with four fields: *subst*, *to_solve*, *unsolved* and *recheck*, where

- *subst* is a substitution obtained previously,
- *to_solve* is a set of equations to solve,
- *unsolved* is a set of equations that cannot be solved, and
- *recheck* is a boolean value that indicates if the unification procedure has to recheck the unsolved equations when there is no equation in *to_solve*.

We denote by $eq(P)$ the set of all the equations in P, that is, the union of $P.to\_solve$ and $P.unsolved$.

Definition 4.2. Two sets of equations P and Q are said to be equivalent if for all substitution σ, we have the equivalence: $\forall (t, u) \in P, t\sigma \equiv u\sigma \Leftrightarrow \forall (v, w) \in Q, v\sigma \equiv w\sigma$

Algorithm 3: Unification

input : a unification problem P
output: a substitution and a set of unsolved equations

if $P.to\_solve = [\,]$ **then**
  if $P.recheck = false$ **then**
    return $(P.subst, P.unsolved)$
  else
    $P.to\_solve \leftarrow P.unsolved$
    $P.unsolved \leftarrow [\,]$
    $P.recheck \leftarrow false$
    return $Unification(P)$
else
  $(t, u) \leftarrow P.to\_solve.pop()$
  $Unificaion\_aux((t, u), P)$

We now present an algorithm for unification modulo rewriting:

Algorithm 4: Unification_aux

input : an equation (t, u) on terms and a unification problem P
output: a substitution and a set of unsolved equations

$t \leftarrow t\downarrow$
$u \leftarrow u\downarrow$
if $t = u$ **then**
  return $Unification(P)$
else if $t = gt_1 \cdots t_n$ *and* $u = hu_1 \cdots u_m$ *with* g, h *two symbols* **then**
  if g *and* h *are distinct constants* **then**
    raise $Unsolvable$
  else if $g = h$ *and* g *is constant* **then**
    for $i = 1$ *to* n **do**
      $P.to\_solve.insert((t_i, u_i))$
    end
    return $Unification(P)$
  else
    $P.unsolved.insert((t, u))$
    return $Unification(P)$
else if $(t, u) = (x, s)$ *or* (s, x) *with* $x \in V - Var(s)$ **then**
  $P \leftarrow P[x \leftarrow s]$ `/* replace all occurrences of` x `in` P `with` s `*/`
  $P.subst.insert((x, s))$
  $P.recheck \leftarrow true$
  return $Unification(P)$
else
  $P.unsolved.insert((t, u))$
  return $Unification(P)$

Proposition 4.1.

- If the call $Unification\_aux((t, u), P)$ raises the exception $Unsolvable$, then t and u are not unifiable.
- If in the call $Unification\_aux((t, u), P)$ we make the call $Unification(P')$, then $\{(t, u)\} \cup eq(P)$ and $eq(P')$ are equivalent.
- If in the call $Unification(P)$ we make the call $Unification(P')$ then $eq(P)$ and $eq(P')$ are equivalent.

- If in the call $Unification(P)$ we make the call $Unification\_aux((t,u),P')$, then $eq(P)$ and $\{(t,u)\} \cup eq(P')$ are equivalent.
- If the call $Unification(P)$ returns (ρ, Q) without any other call, then $eq(P)$ and the union of ρ (a substitution $\{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\}$ is identified with the set $\{(x_1,t_1), \cdots (x_n,t_n)\}$ and $eq(Q)$ are equivalent.

Theorem 4.1.

- Suppose that $Unification(P)$ raises an error. Then $eq(P)$ is not unifiable.
- Suppose that $Unification(P)$ returns (ρ, Q). Let σ be a substitution. Then $eq(P)$ is equivalent to the union of ρ and $eq(Q)$.

In a syntactic unification, unifying two terms of the form $ft_1 \cdots t_n$ and $fu_1 \cdots u_n$ is equivalent to unifying the pairs $(t_1,u_1), \cdots, (t_n,u_n)$. However, this is no longer true in our setting with congruence. This observation gives rise to the study of the injectivity of symbols, given in detail in the next section.

The study of injectivity can be useful for checking the subject reduction property. For example, we can encode in Lambdapi some logical systems using a type T for representing propositions and a function symbol $\epsilon : T \Rightarrow Type$ for interpreting the Curry-Howard correspondence. The injectivity of ϵ is sometimes needed when checking the subject reduction for certain rules.

4.2 Injectivity of function symbols

Assume that the symbols of R are defined in a certain order, i.e., there exists an order $>_{symb}$ on the set S of symbols such that $f >_{symb} g$ iff f is defined after g (the rules defining g do not contain any symbol f s.t. $f >_{symb} g$).

Alternatively, we can reformulate this by saying the dependency graph between symbols is a DAG:

Definition 4.3. We say that f depends on g if there exists a rule $fl \to r$ such that g appears in l or r. The dependency graph between symbols is a directed graph (V,E) defined by $V = S$ and $E = \{(f,g) \mid f \text{ depends on } g\}$.

Moreover, we assume that each symbol has a fixed arity.

Consider the following example:

Example 4.1. Let o and s be two constant symbols.

The following rules define the symbol id:
$R_1 : id\ o \to o$
$R_2 : id\ (s\ x) \to s\ (id\ x)$

Since the rewrite system is assumed to be terminating and confluent, the relation $> := (\triangleright \cup \to)^+$ is well-defined and well-founded. The relation $>_{id}$ defined by $(t,u) >_{id} (t',u') \Leftrightarrow (id\ t \geq id\ t'$ and $id\ u > id\ u')$ or $(id\ t > id\ t'$ and $id\ u \geq id\ u')$ is also well-founded.

We now prove $id\ t \equiv id\ u \Rightarrow t \equiv u$ by induction on (t,u) using the relation $>_{id}$.

- If $id\ t$ and $id\ u$ are in normal form, then $id\ t = id\ u$ by confluence and thus $t = u$.
- If $id\ t$ is in normal form and there exists v s.t. $id\ u \to v$. We distinguish two cases:
  1. u is in normal form: we have $id\ u \to_{R_1} v$ or $id\ u \to_{R_2} v$.
     If $id\ u \to_{R_1} v$, then $u = v = o$. We have thus $id\ t = id\ t\downarrow = id\ u\downarrow = o$, which is impossible.
     If $id\ u \to_{R_2} v$, then there exists w such that $u = s\ w$ and $v = s\ (id\ w)$. Since s is a constant, the normal form of v is also headed by s, which leads to a contradiction since $v\downarrow = id\ u\downarrow = id\ t\downarrow = id\ t$.

2. there exists u' such that $u \to u'$. We have $(t,u) >_{id} (t,u')$ and $id\ u' \equiv id\ u \equiv id\ t$. Thus by I.H., we have $u' \equiv t$, which gives $u \equiv t$.

- If $id\ u$ is in normal form and there exists w s.t. $id\ t \to\ w$, then we proceed as in the previous case.
- If $id\ t$ and $id\ u$ are not in normal form, then we distinguish two cases:
  1. If one of the terms t and u is not in normal form, then we can conclude by simply applying the I.H..
  2. If t and u are in normal form, then there exist $i,j \in \{1,2\}$ and two terms v and w s.t. $id\ t \to_{R_i} v$ and $id\ u \to_{R_j} w$.
     If $(i,j) = (1,1)$, then it is clear that $t = u = o$.
     If $(i,j) = (1,2)$ (resp. $(2,1)$), then it is contradictory since v is headed by the constant o (resp. s) while w is headed by the constant s (resp. o).
     If $(i,j) = (2,2)$, then there exist t' and u' such that $t = s\ t', v = s\ (id\ t'), u = s\ u'$ and $w = s\ (id\ u')$. We have $v \equiv id\ t \equiv id\ u \equiv w$, which implies $id\ t' \equiv id\ u'$ since s is constant. We have $(t,u) >_{id} (t',u')$ since $id\ t \to s\ (id\ t') \triangleright id\ t$ and $id\ u > id\ u'$ for the same reason. Thus, by I.H., $t' \equiv u'$, which gives $t \equiv u$.

Intuitively, we want to design an algorithm that checks the injectivity of a given symbol "by induction" following the structure of the proof above. The second point in the proof by induction above gives rise to a function ($CheckSingleRule$) that checks if the structure of each rule verifies certain properties while the fourth point leads to a function ($CheckTwoRules$) that compares the structure of each rule with another (and itself).

Definition 4.4. Let n be the arity of f. We say that f is I-injective (modulo) if

$$(ft_1 \cdots t_n \equiv fu_1 \cdots u_n \wedge \forall i \in I, t_i \equiv u_i) \Rightarrow \forall i \notin I, t_i \equiv u_i.$$

In the following, we propose an algorithm for checking I-injectivity.

Algorithm 5: CheckInjectivity

input : f, I
output: a boolean value b

```
res ← true
for fl → r ∈ R do
    res ← res ∧ CheckSingleRule(f, I, fl → r)
end
for fl → r ∈ R do
    for fg → d ∈ R do
        res ← res ∧ CheckTwoRules(f, I, fl → r, fg → d)
    end
end
return res
```

Note that, if $fl \to r$ is the same as $fg \to d$, the metavariables in the latter are renamed before calling to $CheckTwoRules$.

The call $CheckSingleRule(f, I, fl \to r)$ corresponds to the case where exactly one of the terms ft and fu is in normal form, and the other can be reduced at the top-level using the rule $fl \to r$. The idea is quite-straight forward: we attempt to use the conditions $ft \equiv fu$ and $t_i \equiv u_i (i \in I)$ to solve the equations $t_i \equiv u_i (i \notin I)$. The most delicate part is that we proceed in an abstract level: we consider rules and metavariables instead of actual terms. Thus, it is appropriate to adapt the approach "Unification as refinement of constraints" taken in $checkSR$ to this case, by using a

hypothetical unification procedure, instead of the one presented previously.

Algorithm 6: CheckSingleRule

input : f, I and a rule $fl \to r$
output: a boolean value b

if $\exists i \in I, l_i = r$ **then**
  return *true*
else if r *is of the form* gl' *with* $g \neq f$ **then**
  if $NoErasing\_rec(g)$ **then**
    return *true*
  else
    return $false$
else
  $P \leftarrow \{(fx_1 \cdots x_n, r)\} \cup \{(x_i, l_i) \mid i \in I\}$ `/* ` x_i `are fresh variables */`
  $Q \leftarrow \{(x_i, l_i) \mid i \notin I\}$
  return $InferFromConstraints(f, I, P, Q)$

As observed in Example 4.1, if we can prove that the normal form of a term cannot be headed by f after applying the rule $fl \to r$, then the case where exactly one of the terms ft and fu is in normal form and the other can be reduced at the top-level using the rule $fl \to r$ can be eliminated. The following function $NoErasing\_rec$ gives one sufficient condition for having this property.

Proposition 4.2. If $NoErasing\_rec(f)$ returns *true*, then the normal form of a term headed by f is headed by f or a symbol defined previously.

Proof. By induction on $>_{symb}$. Trivial. □

Algorithm 7: NoErasing_rec

input : a symbol f
output: a boolean value b

if f *has erasing rules, i.e., rules of the form* $fl \to x$ **then**
  return $false$
else
  $res \leftarrow true$
  for $fl \to gl' \in$ *Rules with* $f \neq g$ **do**
    $res \leftarrow res \wedge NoErasing\_rec(g)$
  end
  return res

The call $CheckTwoRules(f, I, fl \to r, fg \to d)$ corresponds to the case where ft (resp. fu) can be reduced at the top-level using the rule $fl \to r$ (resp. $fg \to d$). The idea is almost the same as that of $CheckSingleRule$.

Algorithm 8: CheckTwoRules

input : f, I, two rules $fl \to r$ and $fg \to d$
output: a boolean value b

$P \leftarrow \{(r, d)\} \cup \{(l_i, g_i) \mid i \in I\}$
$Q \leftarrow \{(l_i, g_i) \mid i \notin I\}$
return $InferFromConstraints(f, I, P, Q)$

Algorithm 9: InferFromConstraints

input : f, I, two sets of constraints P and Q
output: a boolean value b

try:
  $P \leftarrow \{subst = \{\}, unsolved = \{\}, to\_solve = P, recheck = false\}$
  $(\rho, constr) \leftarrow Hypo\_Unification(f, I, P)$
  $Q' \leftarrow Q\rho[x \leftarrow c_x]$ /* c_x are fresh variables */
  $res \leftarrow true$
  for $(t, u) \in Q'$ **do**
    $res \leftarrow res \wedge (t \equiv_{R \cup constr[x \leftarrow c_x]} u)$
  end
  return res
catch *Unsolvable* **:**
  return $true$
end

Intuitively, $InferFromConstraints(f, I, P, Q)$ attempts to solve the constraints Q by using the constraints generated from $Hypo\_Unification(f, I, P)$.

We now describe a first-order hypothetical unification (modulo rewriting) algorithm.

Algorithm 10: Hypo_Unification

input : f, I and a unification problem P
output: a substitution and a set of unsolved equations

if $P.to\_solve = [\,]$ **then**
  if $P.recheck = false$ **then**
    return $unmarked(P.subst, P.unsolved)$
      /* erase all the marks # */
  else
    $P.to\_solve \leftarrow P.unsolved$
    $P.unsolved \leftarrow [\,]$
    $P.recheck \leftarrow false$
    return $Hypo\_Unification(f, I, P)$
else
  $(t, u) \leftarrow P.to\_solve.pop()$
  $Hypo\_Unificaion\_aux(f, I, (t, u), P)$

Algorithm 11: Hypo_Unification_aux

input : f, I, an equation (t, u) on terms and a unification problem P
output: a substitution and a set of unsolved equations

```
1  t ← t↓
2  u ← u↓
3  if t = u then
4  |  return Hypo_Unification(f, I, P)
5  else if t = g t_1 ··· t_n and u = h u_1 ··· u_m with g, h two symbols then
6  |  if g = f^# or h = f^# then
7  |  |  P.unsolved.insert((t, u))
8  |  |  return Hypo_Unification(f, I, P)
9  |  else if g ≠ h then
10 |  |  if g and h are constants then
11 |  |  |  raise Unsolvable
12 |  |  else
13 |  |  |  P.unsolved.insert((t, u))
14 |  |  |  return Hypo_Unification(f, I, P)
15 |  else
16 |  |  J ← {i | t_i ≡ u_i}
17 |  |  if (g = f and I ⊆ J) or CheckInjectivity(g, J) then
18 |  |  |  for i ∉ J do
19 |  |  |  |  P.to_solve.insert((t_i, u_i))
20 |  |  |  end
21 |  |  |  return Hypo_Unification(f, I, P)
22 |  |  else
23 |  |  |  P.unsolved.insert((t, u))
24 |  |  |  return Hypo_Unification(f, I, P)
25 else if (t, u) = (x, s) or (s, x) with x ∈ V − Var(s) then
26 |  P ← P[x ← s_f#]          /* replace all occurrences of x in P with s_f# */
27 |  P.subst.insert((x, s))
28 |  P.recheck ← true
29 |  return Hypo_Unification(f, I, P)
30 else
31 |  P.unsolved.insert((t, u))
32 |  return Hypo_Unification(f, I, P)
```

Here, $f^{\#}$ is a special symbol introduced to guarantee that the induction hypothesis in our proof by induction is applied "correctly". It is considered exactly the same as f except when unifying a term headed by $f^{\#}$ with another term. When we consider a f which is not marked with $\#$, we use f^o to avoid the confusion. We denote by $s_{f\#}$ the term obtained from s by replacing all occurrences of f with $f^{\#}$. If we admit that $CheckInjectivity(f, I) = true \Rightarrow f$ is I-injective, then $Hypo\_Unification$ implements the first-order unification (modulo rewriting) under the hypothesis that f is I-injective (on some set of terms).

The following theorem gives a sufficient condition for I-injectivity.

Theorem 4.2. Let f be a symbol and $I \subseteq \{1, \cdots, arity(f)\}$. If $CheckInjectivity(f, I)$ returns *true*, then f is I-injective.

Proof. We first define a relation S on the set $E = \{(f, I) \mid CheckInjectivity(f, I)$ terminates$\}$ such that $(f, I)\ S\ (g, J)$ if the call $CheckInjectivity(g, J)$ appears in the recursion tree of the call $CheckInjectivity(f, I)$. S is well-founded since for all $(f, I) \in E$, $CheckInjectivity(f, I)$ terminates.

We prove $\boxed{H_0(f, I) : CheckInjectivity(f, I) \text{ returns true} \Rightarrow f \text{ is } I\text{-injective}}$ by induction on E using

the well-founded relation S.

Now, fix f and I and suppose that $H_0(g, J)$ holds for all $(f, I)\ S\ (g, J)$.

Since the rewrite system R is assumed to be terminating and confluent, the relation $>:= (\triangleright \cup \rightarrow)^+$ is well-founded. Let $>_{prod\_f}$ be the well-founded relation defined by $(t, u) >_{prod\_f} (t', u') \Leftrightarrow (ft > ft'$ and $fu \geq fu')$ or $(ft \geq ft'$ and $fu > fu')$. Note that t, u, t', and u' are n-tuples of terms, where $n = arity(f)$. In the following, we write $t \rightarrow t'$ if there exists i s.t. $t_i \rightarrow t'_i$ and $t_j = t'_j\ \forall j \neq i$.

Suppose that $CheckInjectivity(f, I)$ returns true.
We now prove $\boxed{H_1((t, u)) : (ft \equiv fu \wedge \forall i \in I, t_i \equiv u_i) \Rightarrow \forall i \notin I,\ t_i \equiv u_i}$ by induction on (t, u) using $>_{prod\_f}$.

- Suppose that ft and fu are in normal form. Then $ft = ft\downarrow = fu\downarrow = fu$ and $\forall i, t_i = u_i$.
- Suppose that ft is in normal form and that there exists v s.t. $fu \rightarrow v$. We distinguish two cases:

  1. u_i are all in normal form. There exist thus a rule $fl \rightarrow r$ and a substitution σ such that $fu = fl\sigma$ and $v = r\sigma$. Since $CheckSingleRule(f, I, fl \rightarrow r)$ returns *true*, we can distinguish three cases:
     - there exists $i \in I$ s.t. $l_i = r$. In this case $v = r\sigma = l_i\sigma = u_i \equiv t_i$, but we also have $v \equiv fu \equiv ft$, which leads to a contradiction since ft and t_i are both in normal form.
     - r is of the form gl' with $g \neq f$ and $NoErasing\_rec(g)$ returns true. This case is impossible since $v = r\sigma = gl'\sigma$ and by Proposition 1.2., $v\downarrow$ cannot be headed by f.
     - $InferFromConstraints(f, I, P, Q)$ returns true where $P = \{(fx_1 \cdots x_n, r)\} \cup \{(x_i, l_i) \mid i \in I\}$ and $Q = \{(x_i, l_i) \mid i \notin I\}$. Let σ' be the substitution that extends σ with $x_i\sigma' = t_i$.
       Consider the call to $Hypo\_Unification$ in $InferFromConstraints(f, I, P, Q)$ and let $Hypo\_Unification(f, I, P_0), \cdots, Hypo\_Unification(f, I, P_k)$ be the sequence of the recursive calls to $Hypo\_Unification$ in this call (in particular, $eq(P_0) = P$).
       We now prove $\boxed{H_2(i) : \forall (a, b) \in eq(P_i), a\sigma' \equiv b\sigma'}$ and
       $\boxed{H_3(i) : \forall (a, b) \in eq(P_i), \forall f^o t' \trianglelefteq a, \forall f^o u' \trianglelefteq b, (t, u) >_{prod\_f} (t'\sigma'\downarrow, u'\sigma'\downarrow)}$ by induction on i (f^o denotes the unmarked symbol f).
       $H_2(0)$ holds since $r\sigma' = r\sigma = v \equiv fu \equiv ft = (fx_1 \cdots x_n)\sigma'$ and for all $i \in I$, $l_i\sigma' = l_i\sigma = u_i \equiv t_i = x_i\sigma'$.
       Let $(a, b) \in eq(P_0) = P$. If $(a, b) = (fx_1 \cdots x_n, r)$, then for all $f^o t' \trianglelefteq a, f^o u' \trianglelefteq b$, $f^o t'\sigma' \trianglelefteq (fx_1 \cdots x_n)\sigma' = ft$ and $f^o u'\sigma' \trianglelefteq r\sigma' = v \leftarrow fu$ which proves $(t, u) >_{prod\_f} (t'\sigma'\downarrow, u'\sigma'\downarrow)$ since $fc \geq f(c\downarrow)$ for all c. Thus $H_3(0)$ holds.
       Now suppose that $H_2(i)$ and $H_3(i)$ hold and prove that $H_2(i+1)$ and $H_3(i+1)$ hold. Note that we only need to check these properties for the equations inserted into P_i. The only non-trivial inductive steps are (consider the equation treated in the call to *Hypo_Unification_aux* between the calls $Hypo\_Unification(f, I, P_i)$ and $Hypo\_Unification(f, I, P_{i+1})$):
       (a) $(f^o \underbrace{t'_1 \cdots t'_n}_{t'}, f^o \underbrace{u'_1 \cdots u'_n}_{u'})$ and $I \subseteq J$ where $J = \{j \mid t'_j \equiv u'_j\}$ (case 1 in the line 17):
       (H_2): By I.H. (H_3), $(t, u) >_{prod\_f} (t'\sigma'\downarrow, u'\sigma'\downarrow)$. By I.H. ($H_2$), $f^o(t'\sigma'\downarrow) \equiv f^o t'\sigma' \equiv f^o u'\sigma' \equiv f^o(u'\sigma'\downarrow)$ and thus by I.H. (H_1), $t'_j\sigma' \equiv t'_j\sigma'\downarrow \equiv u'_j\sigma'\downarrow \equiv u'_j\sigma'\ \forall j \notin I \subseteq J$.
       (H_3): Trivial.
       (b) $(gt'_1 \cdots t'_m, gu'_1 \cdots u'_m)$ such that $g \neq f$ and that $CheckInjectivity(g, J)$ returns true where $J = \{j \mid t'_j \equiv u'_j\}$ (case 2 in the line 17):
       (H_2): By I.H. (H_2), $g(t'_1\sigma') \cdots (t'_m\sigma') \equiv g(u'_1\sigma') \cdots (u'_m\sigma')$ and by I.H. (H_0), g is J-injective Thus $t'_j\sigma' \equiv u'_j\sigma'\ \forall j \notin J$.
       (H_3): Trivial.

(c) (x, s) where $x \in V$ (line 25):
(H_2): By I.H. (H_2), $x\sigma' \equiv s\sigma'$ (we have even $s\sigma' \downarrow = x\sigma'$ since there exists i s.t. $x\sigma' \triangleleft t_i$ (or u_i) which is in normal form). For all $(a, b) \in eq(P_{i+1})$, there exists $(c, d) \in eq(P_i)$ such that $a = c[x \leftarrow s]$ and $b = d[x \leftarrow s]$. We have thus $a\sigma' = c[x \leftarrow s]\sigma' = c\sigma'[x\sigma' \leftarrow s\sigma'] \equiv c\sigma'$ and $b\sigma' \equiv d\sigma'$. Note that the notation $c\sigma'[x\sigma' \leftarrow s\sigma']$ here is not standard and it simply denotes the substitution of all subterms $x\sigma'$ corresponding to an occurrence of x in c with $s\sigma'$.
(H_3): Let $(a, b) \in eq(P_{i+1})$ and $f^o t'$ (resp. $f^o u'$) be a subterm of a (resp. b). Since the substitution $[x \leftarrow s_{f\#}]$ does not introduce any occurrence of f^o (unmarked f) in P_{i+1}, there exist $(c, d) \in eq(P_i)$, $f^o t'' \trianglelefteq c$ and $f^o u'' \trianglelefteq d$ such that $f^o t' = f^o t''[x \leftarrow s]$ and $f^o u' = f^o u''[x \leftarrow s]$. We have thus $t'\sigma' \downarrow = t''\sigma'[x\sigma' \leftarrow s\sigma'] \downarrow = t''\sigma' \downarrow$ and $u'\sigma' \downarrow = u''\sigma' \downarrow$ since $x\sigma' \equiv s\sigma'$ by I.H. (H_2).
By I.H. (H_3), we have $(t, u) >_{prod\_f} (t''\sigma' \downarrow, u''\sigma' \downarrow) = (t'\sigma' \downarrow, u'\sigma' \downarrow)$.

It is clear that $\forall (x, s) \in P_i.subst, x\sigma' \equiv s\sigma'$. By (H_2), we have:

- If $Hypo\_Unification(f, I, P_0)$ raises the exception $Unsolvable$, then there does not exist a substitution ρ s.t. $\forall (a, b) \in P_0, a\rho \equiv b\rho$, which is impossible, since σ' verifies this property.
- If $Hypo\_Unification(f, I, P_0)$ returns $(\rho, constr)$, then $\forall (x, x\rho) \in \rho, x\sigma' \equiv x\rho\sigma'$ and $\forall (a, b) \in constr, a\sigma' = b\sigma' (*)$. By induction, we have further $a\sigma' \equiv a\rho\sigma'$ for all term a. In this case, since $InferFromConstrs(f, I, P, Q)$ returns true, we have, for all $(c, d) \in Q$, $c\rho[x \leftarrow c_x] \equiv_{R \cup constr[x \leftarrow c_x]} d\rho[x \leftarrow c_x]$. Thus, by replacing c_x with $x\sigma'$, $c\rho\sigma' \equiv_{R \cup constr\sigma'} d\rho\sigma'$ and $c\rho\sigma' \equiv d\rho\sigma'$ by $(*)$. Hence, $c\sigma' \equiv d\sigma'$ for all $(c, d) \in Q$.

2. there exists $v = fu'$ with $u \to u'$. We have $(t, u) >_{prod\_f} (t, u')$ and $\forall i \in I\ u'_i \equiv u_i \equiv t_i$. By I.H., $\forall i \notin I\ t_i \equiv u'_i \equiv u_i$.

- there exist v and w such that $ft \to v$ and $fu \to w$. We assume that t_i and u_i are all in normal form (if not, then the assertion can be proved by simply applying the I.H. (H_1)). There exist thus two rules $fl \to r$, $fg \to d$ and a substitution σ s.t. $ft = fl\sigma, v = r\sigma, fu = fg\sigma$ and $w = d\sigma$ (we can assume $Var(l) \cap Var(g) = \emptyset$ by renaming). By hypothesis, $CheckTwoRules(f, I, fl \to r, fg \to d)$ returns true, which means that $InferFromConstraints(f, I, P, Q)$ returns true where $P = \{(r, d\} \cup \{(l_i, g_i) \mid i \in I\}$ and $Q = \{(l_i, g_i) \mid i \notin I\}$.

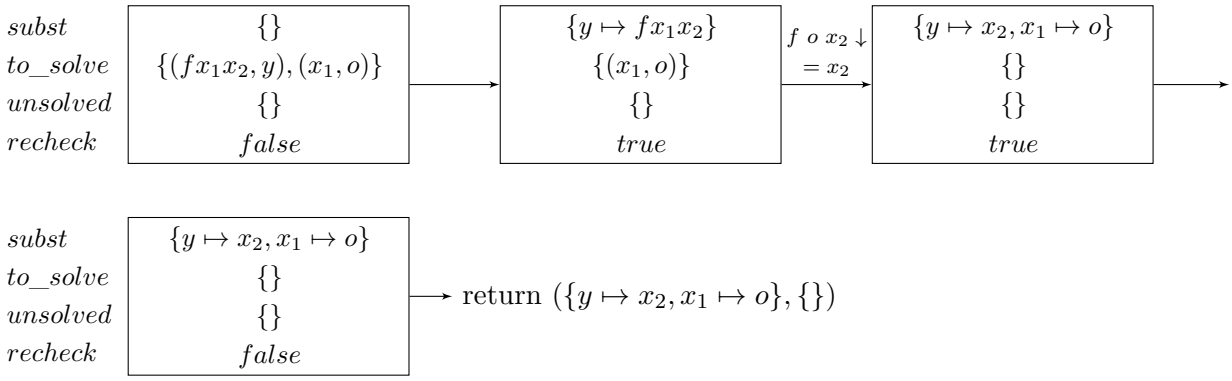
  We can proceed as in the first point in the previous case by replacing σ' with σ, P and Q with their counterparts here.

□

Example 4.2. Let s and o be two constant symbols.
$R_1 : f\ o\ y \to y$
$R_2 : f\ (s\ x)\ y \to s\ (f\ x\ y)$

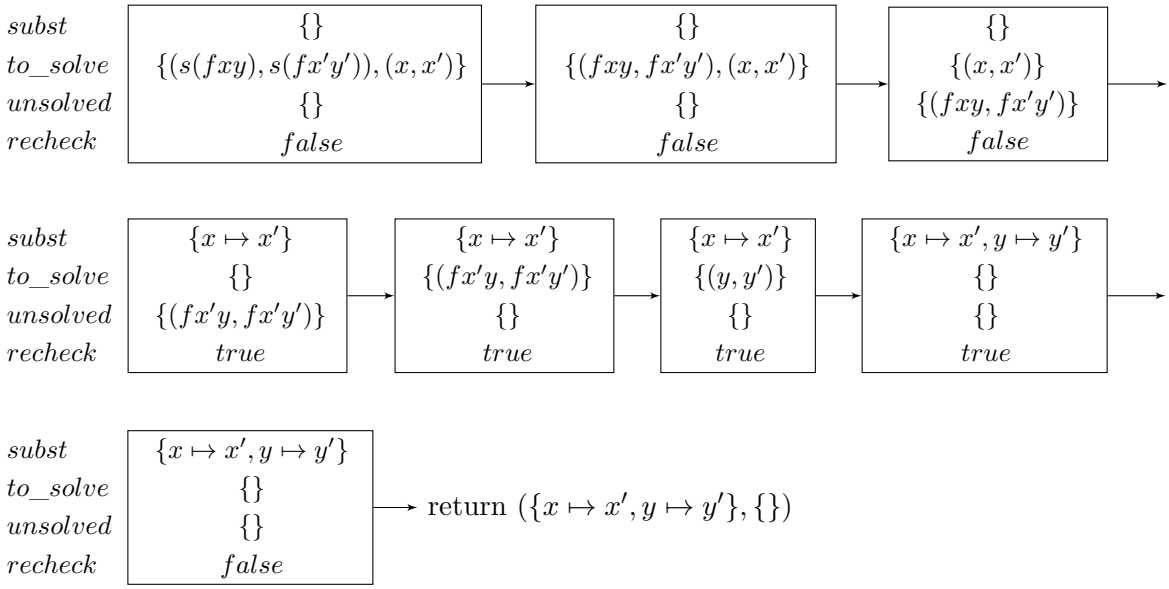
We can prove that f is $\{1\}$-injective by applying the algorithm above:

- In the call $CheckSingleRule(f, \{1\}, R_1)$, the branch chosen is that of the line 8. Thus we first make a call to $Hypo\_Unification$ to unify the equations (fx_1x_2, y) and (x_1, o). The sequence of unification problems considered in the recursive calls to $Hypo\_Unification$ is presented below:



We now apply the substitution obtained to the equation (x_2, y) to solve and the equation obtained is an equality. Hence, $CheckSingleRule(f, \{1\}, R_1)$ returns true.

- In the call $CheckSingleRule(f, \{1\}, R_2)$, the branch chosen is that of the line 3. Since s is constant, $NoErasing\_rec(s)$ returns true. Hence, $CheckSingleRule(f, \{1\}, R_2)$ returns true.
- In the call $CheckTwoRules(f, \{1\}, R_1, R_1)$, we make the call $InferFromContraints(f, \{1\}, P, Q)$ with $P = \{(y, y'), (o, o)\}$ and $Q = \{(y, y')\}$. The call $Hypo\_Unification(f, \{1\}, P)$ returns $(\{y \mapsto y'\}, \{\})$ and the application of the substitution to Q produces an equality. Hence, $CheckTwoRules(f, \{1\}, R_1, R_1)$ returns true.
- In the call $CheckTwoRules(f, \{1\}, R_2, R_2)$, we make the call $InferFromConstraints(f, \{1\}, P, Q)$ with $P = \{(s\ (f\ x\ y), s\ (f\ x'\ y')), (s\ x, s\ x')\}$ and $Q = \{(y, y')\}$. The call $Hypo\_Unification(f, \{1\}, P)$ returns $(\{x \mapsto x', y \mapsto y'\}, \{\})$.



The application of the substitution obtained to Q produces an equality.
Hence $CheckTwoRules(f, \{1\}, R_2, R_2)$ returns true.

- In the call $CheckTwoRules(f, \{1\}, R_1, R_2)$, we make the call $InferFromConstraints(f, \{1\}, P, Q)$ with $P = \{(y, s\ (f\ x\ y')), (o, s\ x)\}$ and $Q = \{(y, y')\}$. Within this call, the call $Hypo\_Unification(P)$ raises the error $Unsolvable$ since s and o are both constant. Thus $CheckTwoRules(f, \{1\}, R_1, R_2)$ returns true.

Thus, $CheckInjectivity(f, \{1\})$ returns true.

However, the algorithm does not allow us to prove that f is $\{2\}$-injective. In fact, in the call $CheckTwoRules(f, \{2\}, R_1, R_2)$, we make the call $InferFromConstraints(f, \{2\}, P, Q)$ with $P = \{(y, y'), (y, s\ \ (f\ \ x\ \ y'))\}$ and $Q = \{(o, s\ \ x)\}$. The call $Hypo\_Unification(P, Q)$ returns $(\{y \mapsto y'\}, \{(y', s\ \ (f\ \ xy'))\})$, but we do not have $o \equiv_{R \cup (c_y, s(fc_xc_{y'}))}\ sc_x$

Remark 4.1. The algorithm might not terminate in some cases: when we call $CheckInjectivity(f, I)$, there might be a call $CheckInjectivity(f, J)$ and when we call $CheckInjectivity(f, J)$, there might be a call $CheckInjectivity(f, I)$ with $I \not\subseteq J$ and $J \not\subseteq I$. A possible solution is to pass as argument a list of pairs (f, J) that should be avoided when making a recursive call to $CheckInjectivity$. For example, when calling $CheckInjectivity(f, J)$, we pass an extra argument (f, I) to make sure that the call $CheckInjectivity(f, I)$ will not be made in this call.

This approach can be extended to the case where there does not exist an order on symbols that is compatible with their dependency, however, it does not allow us to deal with the following example:

Example 4.3. Let s be a constant symbol.
$R_1 : f\ x \to g\ x$
$R_2 : g\ (s\ x) \to s(f\ x)$

In this example, to prove the injectivity of f, we need to have the injectivity of g, and vice versa. What we need here is therefore a proof by mutual induction for the injectivity of f and g.

This observation leads to the generalization obtained from the original algorithm by replacing (f, I) with a list (f_k, I_k) of injective conditions to check.

5 Related work

The subject reduction is a well-known property for the simply typed λ-calculus, but it becomes a difficult problem in presence of dependent types and rewriting. In [4], Barbanera, Fernández, and Geuvers proved that for the algebraic-λ-cube, which extends Barendregt's λ-cube with algebraic rewriting, the product compatibility (PC) implies the subject reduction property for β-reduction. In [3], Blanqui worked on the case of the calculus of algebraic constructions, an extension of the calculus of constructions with object-level and type-level rewrite rules. In [8], Saillard proved the equvalence between the subject reduction for β-reduction and the product compatibility property in the $\lambda\Pi$-calculus modulo rewriting.

The injectivity of function symbols is a new topic but some works on program inversion using term rewriting systems have been done in the last few decades. For example, in [7], Nishida, Sakai, and Sakabe proposed a partial-invesion compiler of constructor term rewriting systems and in [2], Almendros-Jiménez and Vidal worked on systems expressing functional input-output relations.

6 Conclusion and future work

In this report, we propose an algorithm for checking the well-typedness of rewrite rules. The completion procedure allows to retrieve more information from constraints and makes it more possible to solve the equations that need to be satisfied to guarantee that the RHS has the same type as the LHS. However, since higher-order completion is difficult in general, our approach here considers the union of the user-defined rewrite system and the rewrite system obtained by (first-order) completion. Thus, a problem of modularity arises. Until now, we cannot guarantee that the union obtained is confluent and terminating, which makes the conversion decidable, and this is definitely to be improved in the future.

I also propose an algorithm for checking the injectivity of symbols. The use of marked symbols is the most delicate part of this algorithm but this might be improved in the future by choosing a special well-founded relation instead of the one chosen here. Moreover, as mentioned in the last paragraph of the previous section, there exists a generalization of the algorithm that allows to prove several injectivity conditions at the same time. However, it is still not yet known whether there exists an efficient way of determining the injectivity conditions that need to be proved simultaneously from a

single condition. If there exists such a way, then the algorithm proposed can be significantly improved and generalized.

Both algorithms have been implemented in Lambdapi and the code is now available on https://github.com/wujuihsuan2016/lambdapi/tree/sr/src.

References

[1] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, R. Saillard. Dedukti: a logical framework based on the λΠ-calculus modulo theory, 2016. Draft.

[2] J. M. Almendros-Jiménez and G. Vidal. Automatic partial inversion of inductively sequential functions. In Z. Horváth, et al., editors, *Implementations and Applications of Functional Languages*, volume 4449 of *LNCS*, pages 253-270. Springer 2007.

[3] F. Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(1):37-92, 2005.

[4] F. Barbanera, M. Fernández, H. Geuvers. Modularity of strong normalization in the algebraic-λ-cube. *Journal of Functional Porgramming*, 7(6):613-660, 1997.

[5] D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *TLCA 2007*, volume 4583 of *LNCS*, pages 102-117. Springer, 2007.

[6] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1:253-281, 1991.

[7] N., Nishida, M. Sakai and T. Sakabe. Partial inversion of constructor term rewriting systems. In J. Giesl, editor, *Term Rewriting and Applications*, volume 3467 of *LNCS*, pages 264-278, 2005.

[8] R. Saillard. Type checking in the lambda-Pi-calculus modulo: theory and practice. PhD thesis, Mines ParisTech, France, 2015.