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

Jui-Hsuan Wu

To cite this version:

Jui-Hsuan Wu. Checking the type safety of rewrite rules in the \(\lambda\Pi\)-calculus modulo rewriting. Computer Science [cs]. 2019. hal-02288720

HAL Id: hal-02288720
https://inria.hal.science/hal-02288720
Submitted on 15 Sep 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License
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 λΠ-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 λΠ-calculus modulo

2.1 λΠ-calculus

The λΠ-calculus is an extension of the simply typed lambda calculus (λ→) with dependent types. In this calculus, we introduce a particular type $\text{Type}$ and “types” in $\lambda$→ are just terms of type $\text{Type}$ in the λΠ-calculus. Some terms have the type $A \rightarrow \text{Type}$, where $A$ is of type $\text{Type}$, and can be applied to a term $t$ of type $A$ to build a type depending on $t$. Besides, a type $\text{Kind}$ is introduced as the type of terms $\text{Type}$, $A \rightarrow \text{Type}$, 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 $\{\text{Type}, \text{Kind}\}$ 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 λΠ-calculus.

Definition 2.1. (Term) The set of terms in the λΠ-calculus is given by:

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

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 λΠ-calculus:

Well-formedness of the empty context

$$\begin{array}{c}
\text{[ ] well-formed}
\end{array}$$

Declaration of a variable

$$\begin{array}{c}
\Gamma \vdash A : s \\
\Gamma, x : A \text{ well-formed} \\
s \in \mathcal{S}
\end{array}$$

$\text{Type}$ is of type $\text{Kind}$

$$\begin{array}{c}
\Gamma \text{ well-formed} \\
\Gamma \vdash \text{Type} : \text{Kind}
\end{array}$$

Variable

$$\begin{array}{c}
\Gamma \text{ well-formed} \\
\Gamma \vdash x : A \\
x : A \in \Gamma
\end{array}$$
We have the uniqueness (modulo $\equiv_{\beta}$) of types in $\lambda\Pi$-calculus.

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

### 2.2 $\lambda\Pi$-calculus modulo rewriting

The $\lambda\Pi$-calculus modulo rewriting is an extension of the $\lambda\Pi$-calculus with a set $R$ of rewrite rules. Let $\equiv$ be the minimal congruence that contains $R$ and $\beta$-conversion. The typing rules of the $\lambda\Pi$-calculus modulo $R$ can be obtained from those of the $\lambda\Pi$-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 $\lambda\Pi$-calculus modulo by introducing metavariables which are used to represent yet unknown term. We denote by $\mathcal{M}$ the set of metavariables.

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

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. $t_M \in S$. 

• 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 $\text{Meta}(t)$ 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 $f \in F, a_1, \cdots, a_n \in A_0$. The set $A$ of algebraic terms is defined as $A_0 - M_0$.

We now define substitutions.

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

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

**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.

$$
\Gamma \vdash t \uparrow A[E] \quad \text{type checking}
$$

$$
\Gamma \vdash t \downarrow A[E] \quad \text{type inference}
$$

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

**Function applications**

$$
\tau(f) = \Pi x_1 : A_1, \cdots, \Pi x_n : A_n, U \quad \forall i \in \{1, \cdots, n\}, \vdash t_i \uparrow A_i[E_i]
$$

$$
\vdash ft_1 \cdots t_n \downarrow (t_1/x_1, \cdots, t_n/x_n)U[\bigcup_{1 \leq i \leq n} E_i]
$$

**Object-level metavariables**

$$
\vdash M \uparrow T[\{(t_M, T)\}]
$$
Type checking of function applications

\[ \vdash ft_1 \cdots t_n \downarrow A[E] \]
\[ \vdash ft_1 \cdots t_n \uparrow B[E \cup \{(A, B)\}] \]

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 \( \text{InferType} \) and \( \text{CheckType} \) that correspond to the type inference and the type checking respectively.

**Definition 2.5.** We say that a substitution \( \sigma \) 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 \( \sigma' \) is a type-level extension of another substitution \( \sigma \) if \( M\sigma' = M\sigma \) for all \( M \in \text{dom}(\sigma) \) and \( \text{dom}(\sigma') - \text{dom}(\sigma) \subseteq \{t_M | M \in M_0 \cap \text{dom}(\sigma)\} \).

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

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

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

- (Function applications)

  1. Suppose that \( \sigma \models \bigcup_{1 \leq i \leq n} E_i \) and \( \forall M \in \text{Meta}(ft_1 \cdots t_n), \Gamma \vdash M\sigma : t_M\sigma \). We have, \( \forall 1 \leq i \leq n, \sigma \models E_i \text{ and } \forall M \in \text{Meta}(t_i), \Gamma \vdash M\sigma : t_M\sigma. \) By I.H., \( \forall i \in \{1, \ldots, n\}, \Gamma \vdash t_i\sigma : A_i\sigma = A_i \) (the \( A_i\)'s and \( U \) contain no metavariable since function symbols are declared in the global context). Hence, \( \Gamma \vdash (ft_1 \cdots t_n)\sigma = f(t_1\sigma) \cdots (t_n\sigma) : (t_1\sigma/x_1, \ldots, t_n\sigma/x_n)U = ((t_1/x_1, \ldots, t_n/x_n)\sigma) \).

  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 \( \sigma'_i \) of \( \sigma \) s.t. \( \sigma'_i \models E_i \) and \( \forall M \in \text{Meta}(t_i), \Gamma \vdash M\sigma'_i : t_M\sigma'_i. \) Let \( \sigma' \) be the substitution defined by \( \text{dom}(\sigma') = \bigcup_{1 \leq i \leq n} \text{dom}(\sigma'_i) \) and \( M\sigma' = M\sigma'_i. \) Note that, if an object-level metavariable \( M \) occurs in the domains of \( \sigma'_i \) and \( \sigma'_j \), we have, by I.H., \( \Gamma \vdash M\sigma'_i : t_M\sigma'_i \) and \( \Gamma \vdash M\sigma'_j : t_M\sigma'_j. \) \( \sigma'_i \) (resp. \( \sigma'_j \)) is a type-level extension of \( \sigma \), we have thus \( M\sigma'_i = M\sigma'' \) or \( M\sigma'_j \). 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, \( \sigma' \models \bigcup_{1 \leq i \leq n} E_i \) and \( \forall M \in \text{Meta}(ft_1 \cdots t_n), \Gamma \vdash M\sigma' : t_M\sigma'. \) To finish, we have \( V = (t_1\sigma/x_1, \ldots, t_n\sigma/x_n)U = ((t_1/x_1, \ldots, t_n/x_n)\sigma') \).

- (Object-level metavariables)

  1. Suppose that \( \sigma \models \{(t_M, T)\} \), i.e., \( t_M\sigma = T\sigma \) 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 \( \sigma' \) 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. \( \square \)
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 \(l \rightarrow r\), such that:

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

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

**Definition 2.8.** (Rewrites and rewrite steps)

- Let \(l \rightarrow r\) be an algebraic rewrite rule and \(\sigma\) be a substitution. Then \(l\sigma \rightarrow r\sigma\) is called a rewrite and \(l\sigma\) is called a redex.
- Let \(C\) be a context and \(l \rightarrow r\) be a rewrite. Then \(C[l\sigma] \rightarrow C[r\sigma]\) is a rewrite step. This defines the rewrite relation \(\rightarrow_R\) generated by the rewrite system \(R\).

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 \(\rightarrow\) 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 \(l \rightarrow r\) is type-preserving if for any well-formed context \(\Gamma\), any substitution \(\sigma\) 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 \(\rightarrow_\beta\)) The subject reduction for \(\rightarrow_\beta\), expressed below, is a consequence of the product compatibility.

\[
\text{SR}_{\rightarrow_\beta} : \forall \Gamma, t_1, t_2, T, (\Gamma \vdash t_1 : T \text{ and } t_1 \rightarrow_\beta t_2) \Rightarrow \Gamma \vdash t_2 : T.
\]

**Theorem 3.2.** (Subject reduction for \(\rightarrow_R\)) The subject reduction for \(\rightarrow_R\), expressed below, is equivalent to the type preservation of all the rewrite rules in \(R\).

\[
\text{SR}_{\rightarrow_R} : \forall \Gamma, t_1, t_2, T, (\Gamma \vdash t_1 : T \text{ and } t_1 \rightarrow_R t_2) \Rightarrow \Gamma \vdash t_2 : T.
\]

This result remains valid in Lambdapi. Since we assume always the confluence in Lambdapi, the subject reduction for \(\rightarrow_R\) 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 \( l = \text{tail} \ n \ (\text{cons} \ x \ p \ v) \rightarrow v = r \) with \( N : \text{Type}, s : N \Rightarrow N, A : \text{Type}, V : N \Rightarrow \text{Type}, \text{cons} : \Pi x : A.\Pi n : N. V n \Rightarrow V (s \ n), \text{and tail} : \Pi n : N. V (s \ n) \Rightarrow V n \).

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

We first infer the type of the LHS:

\( \vdash \text{tail} \ n \ (\text{cons} \ x \ p \ v) \Downarrow V \ n \{\{(t_n, N), (t_p, N), (t_v, V p), (V \ s \ p), V \ s \ n)\}\}. \)

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 \((t_v, V n)\), 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 \( (E_{\text{to\_solve}}) \). More precisely, we want to prove that every substitution \( \sigma \) satisfying \( E \) satisfies \( E_{\text{to\_solve}} \) 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 \( l \rightarrow r \)
output: a boolean value \( b \)
1 res \leftarrow true
2 \((U, E) \leftarrow \text{InferType}(l)\)
3 \(E_{\text{to\_solve}} \leftarrow \text{CheckType}(U, r)\)
4 \(E' \leftarrow E_{\rho}\)
   /* \( \rho \) replaces every metavariable \( M \) with a fresh (constant) symbol \( c_M \) */
5 \((E_{\text{fo}}, E_{\text{not\_fo}}) \leftarrow \text{FO}(E')\)
   /* split the equations into one first-order part and the rest */
6 \(R' \leftarrow \text{Completion}(E_{\text{fo}})\)
7 \(E'_{\text{to\_solve}} \leftarrow E_{\text{to\_solve}} \rho\)
8 for \((t, u) \in E'_{\text{to\_solve}}\) do
9     \[ res \leftarrow res \land \ (t \downarrow_{R', R' \cup \beta} u \lor Eq_{\text{constr}}(E'_{\text{not\_fo}}, R', t, u)) \]
10 end
11 return res
```

Here, \( \text{InferType}(l) \) returns \((U, E)\) if and only if \( \vdash l \Downarrow U[E] \) and \( \text{CheckType}(U, r) \) returns \( E_{\text{to\_solve}} \) if and only if \( \vdash r \Uparrow U[E_{\text{to\_solve}}] \).

We now describe the completion procedure employed in the form of a rewrite system.
\[
S \cup \{g \rightarrow d, l[g] \rightarrow r\} \mapsto \begin{cases} 
S \cup \{g \rightarrow d, l[d] \rightarrow r\} & \text{if } l[d] > r \\
S \cup \{g \rightarrow d, r \rightarrow l[d]\} & \text{if } l[d] < r \\
S \cup \{g \rightarrow d\} & \text{if } l[d] = r
\end{cases}
\]

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 \( >_{\text{mul}} \), 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 \( g \rightarrow d \) and \( l \rightarrow r \) 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 \( \text{nf}(S) \) of \( S \) is a confluent and terminating rewrite system. Moreover, \( \equiv_{\text{nf}(S)} = \equiv_S \).

**Proof.** By Proposition 3.2, the normal form \( \text{nf}(S) \) 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, \( \text{nf}(S) \) is confluent. To prove the equivalence of the systems, it suffices to prove that \( S \mapsto S' \Rightarrow \leftrightarrow^*_{\text{nf}} = \leftrightarrow^*_{\text{S'}} \).

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**

\begin{itemize}
\item **input**: a set \( E \) of constraints, a rewrite system \( R' \), two terms \( t \) and \( u \)
\item **output**: a boolean value \( b \)
\item \( b \leftarrow \text{false} \)
\item for \( (v, w) \in E \) do
\item \( b \leftarrow b \lor (t \downarrow_{R_0R'^\cup\beta} v \land u \downarrow_{R_0R'} w) \lor (t \downarrow_{R_0R'^\cup\beta} w \land u \downarrow_{R_0R'} v) \)
\item end
\item return \( b \)
\end{itemize}

We now give the main theorem of this section:

**Theorem 3.4.** If \( \text{CheckSR}(l \rightarrow r) \) returns true, then the rule \( l \rightarrow r \) is type-preserving.
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., $\sigma$ 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 $\rho$ when applied to $E$. We have, for all $x \in dom(\rho), x\rho \equiv x\sigma$ since $\rho$, 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 $\rightarrow$ be the rewrite relation induced by $R$ and $\equiv$ the reflexive, symmetric, and transitive closure of $\rightarrow$.

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: $\text{subst}$, $\text{to\_solve}$, $\text{unsolved}$ and $\text{recheck}$, where

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

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

Definition 4.2. Two sets of equations $P$ and $Q$ are said to be equivalent if for all substitution $\sigma$, 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

1 if $P$:to_solve = [] then
2   if $P$:recheck = false then
3      return ($P$:subst, $P$:unsolved)
4   else
5      $P$:to_solve ← $P$:unsolved
6      $P$:unsolved ← []
7      $P$:recheck ← false
8      return Unification($P$)
9 else
10  $(t, u) ← P$:to_solve.pop()
11  Unification_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

1 $t ← t \downarrow$
2 $u ← u \downarrow$
3 if $t = u$ then
4   return Unification($P$)
5 else if $t = gt_1 \cdots t_n$ and $u = hu_1 \cdots u_m$ with $g, h$ two symbols then
6   if $g$ and $h$ are distinct constants then
7      raise Unsolvable
8   else if $g = h$ and $g$ is constant then
9      for $i = 1$ to $n$ do
10         $P$:to_solve.insert($(t_i, u_i))
11      end
12      return Unification($P$)
13 else
14   $P$:unsolved.insert($(t, u))
15   return Unification($P$)
16 else if $(t, u) = (x, s)$ or $(s, x)$ with $x ∈ V − Var(s)$ then
17   $P ← P[x ← s]$  /* replace all occurrences of $x$ in $P$ with $s$ */
18   $P$:subst.insert($(x, s))
19   $P$:recheck ← true
20   return Unification($P$)
21 else
22   $P$:unsolved.insert($(t, u))
23   return Unification($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 $\{ (t, u) \} ∪ eq(P)$ 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 $\text{Unification}(P)$ we make the call $\text{Unification\_aux}((t,u),P')$, then $eq(P)$ and
\[\{t,u\} \cup eq(P')\] are equivalent.

• If the call $\text{Unification}(P)$ returns $(\rho,Q)$ without any other call, then $eq(P)$ and the union of $\rho$
(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 $\text{Unification}(P)$ raises an error. Then $eq(P)$ is not unifiable.

• Suppose that $\text{Unification}(P)$ returns $(\rho,Q)$. Then $eq(P)$ is equivalent to the union of $\rho$ 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/Dedukti 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 $\epsilon$ 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 $>_\text{symb}$ on the
set $S$ of symbols such that $f >_\text{symb} g$ iff $f$ is defined after $g$ (the rules defining $g$ do not contain any
symbol $f$ s.t. $f >_\text{symb} g$).

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 $fl \rightarrow r$ 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 $\text{id}$:

\begin{align*}
R_1 : \text{id } o & \rightarrow o \\
R_2 : \text{id } (s \ x) & \rightarrow s (\text{id } x)
\end{align*}

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

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

• 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. $id u \rightarrow v$. We distinguish two cases:

  1. $u$ is in normal form: we have $id u \rightarrow R_1 v$ or $id u \rightarrow R_2 v$.
     - If $id u \rightarrow R_1 v$, then $u = v = o$. We have thus $id t = id t \downarrow = id u \downarrow = o$, which is impossible.
     - If $id u \rightarrow R_2 v$, 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 $v \downarrow = id u \downarrow = id t \downarrow = id t$. 

10
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. \( id t \to_R_v \) and \( id u \to_R_w \).

    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 \( id t \to s (id t') \to id t \) 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

\[
(ft_1 \cdots t_n \equiv fu_1 \cdots u_n \land \forall i \in I, t_i \equiv u_i) \Rightarrow \forall i \notin I, t_i \equiv u_i.
\]

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

**Algorithm 5:** CheckInjectivity

```plaintext
input : f, I
output: a boolean value b
res \leftarrow true
for fl \to r \in R do
    res \leftarrow res \land CheckSingleRule(f, I, fl \to r)
end
for fl \to r \in R do
    for fg \to d \in R do
        res \leftarrow res \land CheckTwoRules(f, I, fl \to r, fg \to 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 \( \text{CheckSingleRule}(f, I, fl \to r) \) 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 straightforward: 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
Algorithm 6: CheckSingleRule

input : $f, I$ and a rule $fl \rightarrow r$
output: a boolean value $b$
1 if $\exists i \in I, l_i = r$ then
2 return true
3 else if $r$ is of the form $gl'$ with $g \neq f$ then
4 if NoErasing_rec($g$) then
5 return true
6 else
7 return false
8 else
9 $P \leftarrow \{(fx_1 \cdots xn, r)\} \cup \{(x_i, l_i) \mid i \in I\}$ /* $x_i$ are fresh variables */
10 $Q \leftarrow \{(x_i, l_i) \mid i \notin I\}$
11 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 $fl \rightarrow r$, 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 $fl \rightarrow r$ 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 $b$
1 if $f$ has erasing rules, i.e., rules of the form $fl \rightarrow x$ then
2 return false
3 else
4 $res \leftarrow true$
5 for $fl \rightarrow gl' \in$ Rules with $f \neq g$ do
6 $res \leftarrow res \land$ NoErasing_rec($g$)
7 end
8 return $res$

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

Algorithm 8: CheckTwoRules

input : $f, I$, two rules $fl \rightarrow r$ and $fg \rightarrow d$
output: a boolean value $b$
1 $P \leftarrow \{(r, d)\} \cup \{(l_i, g_i) \mid i \in I\}$
2 $Q \leftarrow \{(l_i, g_i) \mid i \notin I\}$
3 return InferFromConstraints($f, I, P, Q$)
Algorithm 9: InferFromConstraints

**input**: \( f, I \), two sets of constraints \( P \) and \( Q \)

**output**: a boolean value \( b \)

```python
try:
    P ← \{subst = \{}, unsolved = \{}, to_solve = P, recheck = false\}
    (ρ, constr) ← Hypo_Unification(f, I, P)
    Q' ← Qρ[x ← cx]
    /* \( cx \) are fresh variables */
    res ← true
    for \((t, u) \in Q'\) do
        res ← res ∧ (t ≡ R\_constr[x←cx] u)
    return res
    catch Unsolvable:
        return true
end
```

Intuitively, \( \text{InferFromConstraints}(f, I, P, Q) \) attempts to solve the constraints \( Q \) by using the constraints generated from \( \text{Hypo}_\_\text{Unification}(f, I, P) \).

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

```python
if P.to_solve = [] then
    if P.recheck = false then
        return unmarked(P.subst, P.unsolved)
    /* erase all the marks */
    else
        P.to_solve ← P.unsolved
        P.unsolved ← []
        P.recheck ← false
        return Hypo_Unification(f, I, P)
    else
        (t, u) ← P.to_solve.pop()
        Hypo_Unification_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. \(t \leftarrow t \downarrow\)
2. \(u \leftarrow u \downarrow\)
3. if \(t = u\) then
   4. return Hypo_Unification\((f, I, P)\)
4. else if \(t = gt_1 \cdots t_n\) and \(u = hu_1 \cdots u_m\) with \(g, h\) two symbols then
5. if \(g = f^\#\) or \(h = f^\#\) then
   6. \(P\).unsolved.insert\((t, u)\)
   7. return Hypo_Unification\((f, I, P)\)
8. else if \(g \neq h\) then
   9. if \(g\) and \(h\) are constants then
      10. raise Unsolvable
      11. else
         12. \(P\).unsolved.insert\((t, u)\)
         13. return Hypo_Unification\((f, I, P)\)
   14. else
      15. \(J \leftarrow \{i \mid t_i \equiv u_i\}\)
      16. if \((g = f\) and \(I \subseteq J\)) or CheckInjectivity\((g, J)\) then
         17. for \(i \notin J\) do
            18. \(P\).to_solve.insert\((t_i, u_i)\)
         19. end
         20. return Hypo_Unification\((f, I, P)\)
      21. else
         22. \(P\).unsolved.insert\((t, u)\)
         23. return Hypo_Unification\((f, I, P)\)
      24. end
      25. else if \((t, u) = (x, s)\) or \((s, x)\) with \(x \in V - \text{Var}(s)\) then
         26. \(P \leftarrow P[x \leftarrow s_f^\#]\) /* replace all occurrences of \(x\) in \(P\) with \(s_f^\#\) */
         27. \(P\).subst.insert\((x, s)\)
         28. \(P\).recheck \leftarrow 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 \(\text{CheckInjectivity}(f, I) = true \Rightarrow f\) is \(I\)-injective, then \(\text{Hypo}_\text{Unification}\) 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 \(I \subseteq \{1, \cdots, \text{arity}(f)\}\). If \(\text{CheckInjectivity}(f, I)\) returns \(true\), then \(f\) is \(I\)-injective.

**Proof.** We first define a relation \(S\) on the set \(E = \{(f, I) \mid \text{CheckInjectivity}(f, I) \text{ terminates}\}\) such that \((f, I) S (g, J)\) if the call \(\text{CheckInjectivity}(g, J)\) appears in the recursion tree of the call \(\text{CheckInjectivity}(f, I)\). \(S\) is well-founded since for all \((f, I) \in E,\) \(\text{CheckInjectivity}(f, I)\) terminates. We prove \(\boxed{H_0(f, I) : \text{CheckInjectivity}(f, I) \text{ returns true } \Rightarrow f \text{ is } I\text{-injective}}\) by induction on \(E\) using
We now prove hypotheses that the well-founded relation

\[ S \]

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

Suppose that \( \text{CheckInjectivity}(f, I) \) returns true.

We now prove \( H_1((t, u)) : (ft \equiv fu \land \forall i \in I, t_i \equiv u_i) \Rightarrow \forall i \notin I, t_i \equiv u_i \) by induction on \((t, u)\) using \( \Rightarrow_{prod,f} \).

- Suppose that \( ft \) and \( fu \) are in normal form. Then \( ft = fu \downarrow = fu \) and \( \forall i, t_i = u_i \).
- Suppose that \( ft \) is in normal form and that there exists \( v \) s.t. \( fu \to v \). We distinguish two cases:
  1. \( u_i \) are all in normal form. There exist thus a rule \( fl \to r \) and a substitution \( \sigma \) such that \( fu = f\sigma \) and \( v = r\sigma \). Since \( \text{CheckSingleRule}(f, I, fl \to r) \) 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 \( \text{NoErasing_rec}(g) \) returns true. This case is impossible since \( v = \sigma = gl'\sigma \) and by Proposition 1.2., \( v \downarrow \) cannot be headed by \( f \).
     - \( \text{InferFromConstraints}(f, I, P, Q) \) returns true where \( P = \{(fx_1 \cdots xn) \} \cup \{(x_i, l_i) \mid i \in I \} \) and \( Q = \{(x_i, l_i) \mid i \notin I \} \). Let \( \tau' \) be the substitution that extends \( \sigma \) with \( x_i\tau' = t_i \).
   
   Consider the call to \( \text{Hypo-Unification} \) in \( \text{InferFromConstraints}(f, I, P, Q) \) and let \( \text{Hypo-Unification}(f, I, P_0), \ldots, \text{Hypo-Unification}(f, I, P_k) \) be the sequence of the recursive calls to \( \text{Hypo-Unification} \) in this call (in particular, \( eq(P_0) = P \)).

   We now prove \( H_2(i) : \forall(a, b) \in eq(P_1), a\tau' \equiv b\tau' \) and 

   \[ H_3(i) : \forall(a, b) \in eq(P_1), \forall f^o t' \triangleleft a, \forall f^o u' \triangleleft b, (t, u) \Rightarrow_{prod,f} (t\tau' \downarrow, u\tau' \downarrow) \]

   by induction on \( i \) (\( f^o \) denotes the unmarked symbol \( f \)).

   \( H_2(0) \) holds since \( ro' = r\sigma = v \equiv fu \equiv ft = (fx_1 \cdots xn)\sigma' \) and for all \( i \in I, l_i\sigma' = l_i = u_i \equiv t_i = x_i\tau' \).

   Let \((a, b) \in eq(P_0) = P \). If \((a, b) = (fx_1 \cdots xn, r)\), then for all \( f^o t' \triangleleft a, f^o u' \triangleleft b, f^o t' \sigma' \triangleleft (fx_1 \cdots xn)\sigma' = ft \) and \( f^o u' \sigma' \triangleleft r\sigma = v \iff fu \) which proves \((t, u) \Rightarrow_{prod,f} (t\tau' \downarrow, u\tau' \downarrow) \) since \( fe \geq f(c) \) 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 \( P_1 \). The only non-trivial inductive steps are (consider the equation treated in the call to \( \text{Hypo-Unification}_\text{aux} \) between the calls \( \text{Hypo-Unification}(f, I, P_1) \) and \( \text{Hypo-Unification}(f, I, P_{i+1}) \)):

(a) \( (f^o t'_1 \cdots t'_o, f^o u'_1 \cdots u'_o) \) and \( I \subseteq J \) where \( J = \{ j \mid t'_j \equiv u'_j \} \) (case 1 in the line 17):

   \( H_2) \): By I.H. \( (H_3), (t, u) \Rightarrow_{prod,f} (t\tau' \downarrow, u\tau' \downarrow) \). By I.H. \( (H_2), f^o(t\tau' \downarrow) \equiv f^o(u\tau' \downarrow) \equiv f^o(u\tau' \downarrow) \) and thus by I.H. \( (H_1), t'_j\tau' \equiv u'_j\tau' \downarrow \equiv u'_j\tau' \downarrow \forall j \notin I \subseteq J \).

   \( H_3) \): Trivial.

(b) \( (gt'_1 \cdots t'_m, gu'_1 \cdots u'_m) \) such that \( g \neq f \) and that \( \text{CheckInjectivity}(g, J) \) returns true where \( J = \{ j \mid t'_j \equiv u'_j \} \) (case 2 in the line 17):

   \( H_2) \): By I.H. \( (H_2), g(t'_j\sigma' \downarrow) \equiv g(u'_j\sigma' \downarrow) \) 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):

(H2): By I.H. (H2), \(x \sigma' = x \sigma\) (we have even \(s \sigma' \downarrow = x \sigma\) since there exists \(i\) s.t. \(x \sigma' < t_i\) (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 \(aa' = c[x \leftarrow s]a' = ca'[x \sigma' \leftarrow x \sigma'] \equiv c \sigma'\) and \(ba' \equiv d \sigma'.\) Note that the notation \(c \sigma'[x \sigma' \leftarrow x \sigma']\) here is not standard and it simply denotes the substitution of all subterms \(x \sigma'\) corresponding to an occurrence of \(x\) in \(c\) with \(x \sigma'\).

(H3): 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 \([x \leftarrow s_{f \sigma}]\) does not introduce any occurrence of \(f^o\) (unmarked \(f\)) in \(P_{i+1}\), there exist \((c, d) \in eq(P_i)\), \(f^o t'' \subseteq c\) and \(f^o u'' \subseteq d\) such that \(f^o t' = f^o t''[x \leftarrow s]\) and \(f^o u' = f^o u''[x \leftarrow s]\). We have thus \(t' \sigma' \downarrow = t'' \sigma'[x \sigma' \leftarrow x \sigma'] \downarrow = t'' \sigma' \downarrow\) and \(u' \sigma' \downarrow = u'' \sigma' \downarrow\) since \(x \sigma' \equiv x \sigma\) by I.H. (H2).

By I.H. (H3), we have \((t, u) \succ_{prod-f} (t'' \sigma', u'' \sigma') = (t' \sigma', u' \sigma')\).

It is clear that \(\forall (x, s) \in P, subst, x \sigma' \equiv x \sigma'\). By (H2), we have:

- If \(Hypo\_Unification(f, I, P_0)\) raises the exception \(Unsolvable\), then there does not exist a substitution \(\rho\) s.t. \(\forall (a, b) \in P_0, a \rho \equiv b \rho\), which is impossible, since \(x \sigma'\) verifies this property.

- If \(Hypo\_Unification(f, I, P_0)\) returns \((\rho, constr)\), then \(\forall (x, s) \rho \equiv x \rho \sigma'\) and \(\forall (a, b) \in constr, a \rho \equiv b \rho\) for all term \(a\). In this case, since \(Infer\_From\_Constraints(f, I, P, Q)\) returns true, we have, for all \((c, d) \in Q\), \(c p[x \leftarrow c_x] \equiv_{R, constr[x \leftarrow c_x]} dp[x \leftarrow c_x]\). Thus, by replacing \(c_x\) with \(x \sigma'\), \(c p \rho \equiv_{R, constr[x \leftarrow c_x]} d p \rho\) and \(c p a \rho \equiv_{R, constr[x \leftarrow c_x]} d p a \rho\) by (*). Hence, \(a \rho \equiv d a \rho\) for all \((c, d) \in Q\).

2. there exists \(v = f u'\) with \(u \rightarrow u'\). We have \((t, u) \succ_{prod-f} (t, u')\) and \(\forall i \in I u_i' = u_i \equiv t_i\). By I.H., \(\forall i \not\in I t_i = u_i' = u_i\).

- there exist \(v\) and \(w\) such that \(ft \rightarrow v\) and \(fu \rightarrow w\). 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. (H1)). There exist thus two rules \(fl \rightarrow r\), \(fg \rightarrow d\) and a substitution \(\sigma\) s.t. \(ft = fl \sigma, v = r \sigma, fu = f g \sigma\) and \(w = d \sigma\), (we can assume \(Var(l) \cap Var(g) = \emptyset\) by renaming). By hypothesis, \(Check\_Two\_Rules(f, I, fl \rightarrow r, fg \rightarrow d)\) returns true, which means that \(Infer\_From\_Constraints(f, I, P, Q)\) returns true where \(P = \{(r, d) \cup \{(l_i, g_i) \mid i \in I\}\}\) and \(Q = \{(l_i, g_i) \mid i \not\in I\}\).

We can proceed as in the first point in the previous case by replacing \(x \sigma'\) with \(x \sigma, P\) and \(Q\) with their counterparts here.

\[\square\]

**Example 4.2.** Let \(s\) and \(o\) be two constant symbols.

\[
R_1 : f \circ o \rightarrow y
R_2 : f(s \ x) y \rightarrow s(f \ x \ y)
\]

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

- In the call \(Check\_Single\_Rule(f, \{1\}, R_1)\), the branch chosen is that of the line 8. Thus we first make a call to \(Hypo\_Unification\) to unify the equations \((f \ x, x_2, y)\) and \((x_1, o)\). The sequence of unification problems considered in the recursive calls to \(Hypo\_Unification\) is presented below:
Thus, \( \text{CheckInjectivity}(f, \{1\}) \) returns true.

However, the algorithm does not allow us to prove that \( f \) is \( \{2\} \)-injective. In fact, in the call \( \text{CheckTwoRules}(f, \{2\}, R_1, R_2) \), we make the call \( \text{InferFromConstraints}(f, \{2\}, P, Q) \) with \( P = \{(y',y),(o,a)\} \) and \( Q = \{(y,y')\} \). The call \( \text{Hypo_\text{Unification}}(f, \{2\}, P) \) returns \( \{(y \mapsto y')\},\{\}\} \) and the application of the substitution to \( Q \) produces an equality. Hence, \( \text{CheckTwoRules}(f, \{2\}, R_1, R_2) \) returns true.

Thus, \( \text{CheckInjectivity}(f, \{1\}) \) returns true.
Remark 4.1. The algorithm might not terminate in some cases: when we call \texttt{CheckInjectivity}(f, I), there might be a call \texttt{CheckInjectivity}(f, J) and when we call \texttt{CheckInjectivity}(f, J), there might be a call \texttt{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 \texttt{CheckInjectivity}. For example, when calling \texttt{CheckInjectivity}(f, J), we pass an extra argument $(f, I)$ to make sure that the call \texttt{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.

\begin{align*}
R_1 : f \ x & \to g \ x \\
R_2 : g \ (s \ x) & \to s \ (f \ x)
\end{align*}

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 $\lambda$-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-$\lambda$-cube, which extends Barendregt’s $\lambda$-cube with algebraic rewriting, the product compatibility (PC) implies the subject reduction property for $\beta$-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 equivalence between the subject reduction for $\beta$-reduction and the product compatibility property in the $\lambda$I-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-inversion 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/wujuilhsuan2016/lambdapi/tree/sr/src.

References


