Type safety of rewrite rules in dependent types

Frédéric Blanqui

▶ **To cite this version:**

Frédéric Blanqui. Type safety of rewrite rules in dependent types. FSCD 2020 - 5th International Conference on Formal Structures for Computation and Deduction, Jun 2020, Paris, France. pp.14, 10.4230/LIPIcs.FSCD.2020.13 . hal-02981528

HAL Id: hal-02981528
https://inria.hal.science/hal-02981528v1

Submitted on 28 Oct 2020

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.

Type safety of rewrite rules in dependent types

Frédéric Blanqui
Université Paris-Saclay, ENS Paris-Saclay, CNRS, Inria
Laboratoire Spécification et Vérification, 94235, Cachan, France

Abstract

The expressiveness of dependent type theory can be extended by identifying types modulo some additional computation rules. But, for preserving the decidability of type-checking or the logical consistency of the system, one must make sure that those user-defined rewriting rules preserve typing. In this paper, we give a new method to check that property using Knuth-Bendix completion.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Equational logic and rewriting; Theory of computation → Logic and verification

Keywords and phrases subject-reduction, rewriting, dependent types

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.11

Supplement Material `https://github.com/wujuihsuan2016/lambdapi/tree/sr`

Acknowledgements The author thanks Jui-Hsuan Wu for his prototype implementation and his comments on a preliminary version of this work.

1 Introduction

The $\lambda\Pi$-calculus, or LF [12], is an extension of the simply-typed λ-calculus with dependent types, that is, types that can depend on values like, for instance, the type Vn of vectors of dimension n. And two dependent types like Vn and Vp are identified as soon as n and p are two expressions having the same value (modulo the evaluation rule of λ-calculus, β-reduction).

In the $\lambda\Pi$-calculus modulo rewriting, function and type symbols can be defined not only by using β-reduction but also by using rewriting rules [19]. Hence, types are identified modulo β-reduction and some user-defined rewriting rules. This calculus has been implemented in a tool called Dedukti [9].

Adding rewriting rules adds a lot of expressivity for encoding logical or type systems. For instance, although the $\lambda\Pi$-calculus has no native polymorphism, one can easily encode higher-order logic or the calculus of constructions by using just a few symbols and rules [8]. As a consequence, various tools have been developed for translating actual terms and proofs from various systems (Coq, OpenTheory, Matita, Focalize, ...) to Dedukti, and back, opening the way to some interoperability between those systems [1]. The Agda system recently started to experiment with rewriting too [7].

To preserve the decidability of type-checking and the logical consistency, it is however essential that the rules added by the user preserve typing, that is, if an expression e has some type T and a rewriting rule transforms e into a new expression e', then e' should have type T too. This property is also very important in programming languages, to avoid some errors (a program declared to return a string should not return an integer).

When working in the simply-typed λ-calculus, it is not too difficult to ensure this property: it suffices to check that, for every rewriting rule $l \hookrightarrow r$, the right-hand side (RHS) r has the same type as the left-hand side (LHS) l, which is decidable.

The situation is however much more complicated when working with dependent types modulo user-defined rewriting rules. As type-checking requires one to decide the equivalence


© Inria;
licensed under Creative Commons License CC-BY

of two expressions, it is undecidable in general to say whether a rewriting rule preserves typing, even β-reduction alone [17].

Note also that, in the $\lambda\Pi$-calculus modulo rewriting, the set of well-typed terms is not fixed but depends on the rewriting rules themselves (it grows when one adds rewriting rules).

Finally, the technique used in the simply-typed case (checking that both the LHS and the RHS have the same type) is not satisfactory in the case of dependent types, as it often forces rule left-hand sides to be non-linear [6], making other important properties (namely confluence) more difficult to establish and the implementation of rewriting less efficient (if it does not use sharing).

▶ **Example 1.** As already mentioned, Dedukti is often used to encode logical systems and proofs coming from interactive or automated theorem provers. For instance, one wants to be able to encode the simply-typed λ-calculus in Dedukti. Using the new Dedukti syntax[1], this can be done as follows (rule variables must be prefixed by `$` to distinguish them from function symbols with the same name):

```
constant symbol T: TYPE // Dedukti type for representing simple types
constant symbol arr: T → T → T // arrow simple type constructor

injective symbol τ: T → TYPE// interprets T elements as Dedukti types
rule τ (arr $x $y) ↪ τ $x → τ $y // (Curry-Howard isomorphism)

// representation of simply-typed λ-terms
symbol lam: Π a b, (τ a → τ b) → τ (arr a b)
symbol app: Π a b, τ (arr a b) → (τ a → τ b)

rule app $a $b (lam $a' $b' $f) $x ↪ $f $x // β-reduction
```

Proving that the above rule preserves typing is not trivial as it is equivalent to proving that β-reduction has the subject-reduction property in the simply-typed λ-calculus. And, indeed, the previous version of Dedukti was unable to prove it.

The LHS is typable if f is of type $\tau(\mathtt{arr}\, a'\, b')$, $\tau(\mathtt{arr}\, a'\, b') \simeq \tau(\mathtt{arr}\, a\, b)$, and x is of type τa. Then, in this case, the LHS is of type τb.

Here, one could be tempted to replace a' by a, and b' by b, so that these conditions are satisfied but this would make the rewriting rule non left-linear and the proof of its confluence problematic [13].

Fortunately, this is not necessary. Indeed, we can prove that the RHS is typable and has the same type as the LHS by using the fact that $\tau(\mathtt{arr}\, a'\, b') \simeq \tau(\mathtt{arr}\, a\, b)$ when the LHS is typable. Indeed, in this case, and thanks to the rule defining τ, f is of type $\tau a \to \tau b$. Therefore, the RHS has type τb as well.

In this paper, we present a new method for doing this kind of reasoning automatically. By using Knuth-Bendix completion [15, 18], the equations holding when a LHS is typable are turned into a convergent (*i.e.* confluent and terminating) set of rewriting rules, so that the type-checking algorithm of Dedukti itself can be used to check the type of a RHS modulo these equations.

Outline. The paper is organized as follows. In Section 2, we recall the definition of the $\lambda\Pi$-calculus modulo rewriting. In Section 3, we recall what it means for a rewriting rule to

[1] `https://github.com/Deducteam/lambdapi`

preserve typing. In Section 4, we describe a new algorithm for checking that a rewriting rule preserves typing and provide general conditions for ensuring its termination. Finally, in Section 5, we compare this new approach with previous ones and conclude.

2 λΠ-calculus modulo rewriting

Following Barendregt's book on typed λ-calculus [4], the λΠ-calculus is a Pure Type System (PTS) on the set of sorts $\mathcal{S} = \{\star, \Box\}$:[2]

▶ **Definition 2** (λΠ-term algebra). *A λΠ-term algebra is defined by:*

- *a set $\mathcal{F}$ of function symbols,*
- *an infinite set $\mathcal{V}$ of variables,*

such that $\mathcal{V}$, $\mathcal{F}$ and $\mathcal{S}$ are pairwise disjoint.

The set $\mathcal{T}(\mathcal{F},\mathcal{V})$ of λΠ-terms is then inductively defined as follows:

$$t,u := s \in \mathcal{S} \mid x \in \mathcal{V} \mid f \in \mathcal{F} \mid \lambda x:t,u \mid tu \mid \Pi x:t,u$$

where $\lambda x:t,u$ is called an abstraction, tu an application, and $\Pi x:t,u$ a (dependent) product (simply written $t \to u$ if x does not occur in u). As usual, terms are identified modulo renaming of bound variables (x is bound in $\lambda x:t,u$ and $\Pi x:t,u$). We denote by $\mathrm{FV}(t)$ the free variables of t. A term is said to be closed if it has no free variables.

A substitution is a finite map from $\mathcal{V}$ to $\mathcal{T}(\mathcal{F},\mathcal{V})$. It is written as a finite set of pairs. For instance, $\{(x,a)\}$ is the substitution mapping x to a.

Given a substitution σ and a term t, we denote by $t\sigma$ the capture-avoiding replacement of every free occurrence of x in t by its image in σ.

▶ **Definition 3** (λΠ-calculus). *A λΠ-calculus on $\mathcal{T}(\mathcal{F},\mathcal{V})$ is given by:*

- *a function $\Theta : \mathcal{F} \to \mathcal{T}(\mathcal{F},\mathcal{V})$ mapping every function symbol f to a term Θ_f called its type (we will often write $f : A$ instead of $\Theta_f = A$),*
- *a function $\Sigma : \mathcal{F} \to \mathcal{S}$ mapping every function symbol f to a sort Σ_f,*
- *a set $\mathcal{R}$ of rewriting rules $(l,r) \in \mathcal{T}^2$, written $l \hookrightarrow r$, such that $\mathrm{FV}(r) \subseteq \mathrm{FV}(l)$.*

We then denote by $\simeq$ the smallest equivalence relation containing $\hookrightarrow = \hookrightarrow_{\mathcal{R}} \cup \hookrightarrow_\beta$ where $\hookrightarrow_{\mathcal{R}}$ is the smallest relation stable by context and substitution containing $\mathcal{R}$, and $\hookrightarrow_\beta$ is the usual β-reduction relation.

▶ **Example 4.** For representing natural numbers, we can use the function symbols $N : \star$ of sort $\Box$, and the function symbols $0 : N$ and $s : N \to N$ of sort $\star$. Addition can be represented by $+ : N \to N \to N$ of sort $\star$ together with the following set of rules:

$$\begin{array}{rcl} 0+y & \hookrightarrow & y \\ x+0 & \hookrightarrow & x \\ x+(sy) & \hookrightarrow & s(x+y) \\ (sx)+y & \hookrightarrow & s(x+y) \\ (x+y)+z & \hookrightarrow & x+(y+z) \end{array}$$

Note that Dedukti allows overlapping LHS and matching on defined symbols like in this example. (It also allows higher-order pattern matching like in Combinatory Reduction Systems (CRS) [14] but we do not consider this feature in the current paper.)

[2] PTS sorts should not be confused with the notion of sort used in first-order logic. The meaning of these sorts will be explained after the definition of typing (Definition 5). Roughly speaking, $\star$ is the type of objects and proofs, and $\Box$ is the type of set families and predicates.

Figure 1 Typing rules of the $\lambda\Pi$-calculus modulo rewriting

$$\text{(ax)}\quad \vdash \star : \Box$$

$$\text{(fun)}\quad \frac{\vdash \Theta_f : \Sigma_f}{\vdash f : \Theta_f}$$

$$\text{(var)}\quad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \qquad (x \notin \Gamma)$$

$$\text{(weak)}\quad \frac{\Gamma \vdash t : T \quad \Gamma \vdash A : s}{\Gamma, x : A \vdash t : T} \qquad (x \notin \Gamma)$$

$$\text{(prod)}\quad \frac{\Gamma \vdash A : \star \quad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A, B : s}$$

$$\text{(app)}\quad \frac{\Gamma \vdash t : \Pi x : A, B \quad \Gamma \vdash a : A}{\Gamma \vdash ta : B\{(x, a)\}}$$

$$\text{(abs)}\quad \frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash \Pi x : A, B : s}{\Gamma \vdash \lambda x : A, b : \Pi x : A, B}$$

$$\text{(conv)}\quad \frac{\Gamma \vdash t : T \quad \Gamma \vdash U : s}{\Gamma \vdash t : U} \qquad (T \simeq U)$$

Throughout the paper, we assume a given $\lambda\Pi$-calculus $\Lambda = (\mathcal{F}, \mathcal{V}, \Theta, \Sigma, \mathcal{R})$.

▶ **Definition 5** (Well-typed terms). *A typing environment is a possibly empty ordered sequence of pairs $(x_1, A_1), \ldots, (x_n, A_n)$, written $x_1 : A_1, \ldots, x_n : A_n$, where the x_i's are distinct variables and the A_i's are terms.*

A term t has type A in a typing environment Γ if the judgment $\Gamma \vdash t : A$ is derivable from the rules of Figure 1. An environment Γ is valid if some term is typable in it.

A substitution σ is a well-typed substitution from an environment Γ to an environment Γ', written $\Gamma' \vdash \sigma : \Gamma$, if, for all $x : A \in \Gamma$, we have $\Gamma' \vdash x\sigma : A\sigma$.

Note that well-typed substitutions preserve typing: if $\Gamma \vdash t : T$ and $\Gamma' \vdash \sigma : \Gamma$, then $\Gamma' \vdash t\sigma : T\sigma$ [5].

A type-checking algorithm for the $\lambda\Pi$-calculus modulo (user-defined) rewriting rules is implemented in the Dedukti tool [9].

We first recall a number of basic properties that hold whatever $\mathcal{R}$ is and can be easily proved by induction on $\vdash$ [5]:

▶ **Lemma 6.** **(a)** *If t is typable, then every subterm of t is typable.*
(b) *$\Box$ is not typable.*
(c) *If $\Gamma \vdash t : T$ then either $T = \Box$ or $\Gamma \vdash T : s$ for some sort s.*
(d) *If $\Gamma \vdash t : \Box$ then t is a kind, that is, of the form $\Pi x_1 : T_1, \ldots, \Pi x_n : T_n, \star$.*
(e) *If $\Gamma \vdash t : T$, $\Gamma \subseteq \Gamma'$ and Γ' is valid, then $\Gamma' \vdash t : T$.*

Throughout the paper, we assume that, for all f, $\vdash \Theta_f : \Sigma_f$. Indeed, if $\vdash \Theta_f : \Sigma_f$ does not hold, then no well-typed term can contain f. (This assumption is implicit in the presentations of LF using signatures [12].)

More importantly, we will assume that $\hookrightarrow$ is confluent on the set $\mathcal{T}(\mathcal{F}, \mathcal{V})$ of untyped terms, that is, for all terms $t, u, v \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, if $t \hookrightarrow^* u$ and $t \hookrightarrow^* v$, then there exists a term $w \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ such that $u \hookrightarrow^* w$ and $v \hookrightarrow^* w$, where $\hookrightarrow^*$ is the reflexive and transitive closure of $\hookrightarrow$.

This condition is required for ensuring that conversion behaves well with respect to products (if $\Pi x : A, B \simeq \Pi x : A', B'$ then $A \simeq A'$ and $B \simeq B'$), which in particular implies subject-reduction for $\hookrightarrow_\beta$.

This last assumption may look strong, all the more so since confluence is undecidable. However this property is satisfied by many systems in practice. For instance, $\hookrightarrow$ is confluent if the left-hand sides of $\mathcal{R}$ are algebraic (Definition 8), linear and do not overlap with each other [21]. This is in particular the case of the rewriting systems corresponding to the function definitions allowed in functional programming languages such as Haskell, Agda, OCaml or Coq. But confluence can be relaxed in some cases: when there are no type-level rewriting rules [3] or when the right-hand sides of type-level rewriting rules are not products [6].

When $\hookrightarrow$ is confluent, the typing relation satisfies additional properties. For instance, the set of typable terms can be divided into three disjoint classes:

- the terms of type $\Box$, called kinds, of the form $\Pi x_1 : A_1, \ldots, \Pi x_n : A_n, \star$;
- the terms whose type is a kind, called predicates;
- the terms whose type is a predicate, called objects.

3 Subject-reduction

A relation $\rhd$ preserves typing (subject-reduction property) if, for all environments Γ and all terms t, u and A, if $\Gamma \vdash t : A$ and $t \rhd u$, then $\Gamma \vdash u : A$.

One can easily check that $\hookrightarrow_\beta$ preserves typing when $\hookrightarrow$ is confluent [5]. Our aim is therefore to check that $\hookrightarrow_\mathcal{R}$ preserves typing too. To this end, it is enough to check that every rule $l \hookrightarrow r \in \mathcal{R}$ preserves typing, that is, for all environments Γ, substitutions σ and terms A, if $\Gamma \vdash l\sigma : A$, then $\Gamma \vdash r\sigma : A$.

A first idea is to require that:

(*) there exist Δ and B such that $\Delta \vdash l : B$ and $\Delta \vdash r : B$.

But this condition is not sufficient in general as shown by the following example:

▶ **Example 7.** Consider the rule $f(xy) \hookrightarrow y$ with $f : B \to B$. In the environment $\Delta = x : B \to B, y : B$, we have $\Delta \vdash l : B$ and $\Delta \vdash r : B$. However, in the environment $\Gamma = x : A \to B, y : A$, we have $\Gamma \vdash l : B$ and $\Gamma \vdash r : A$.

The condition (*) is sufficient if the rule left-hand side is a non-variable simply-typed first-order term [3], a notion that we slightly generalize as follows:

▶ **Definition 8** (Pattern). *We assume that the set of variables is split in two disjoint sets, the algebraic variables and the non-algebraic ones, and that there is an injection $\widehat{\ }$ from algebraic variables to non-algebraic variables.*

A term is algebraic if it is an algebraic variable or of the form $ft_1 \ldots t_n$ with each t_i algebraic and f a function symbol whose type is of the form $\Pi x_1 : A_1, \ldots, x_n : A_n, B$.

A term is an object-level algebraic term if it is algebraic and all its function symbols are of sort $\star$.

A pattern is an algebraic term of the form $ft_1 \ldots t_n$ where each t_i is an object-level algebraic term.

The distinction between algebraic and non-algebraic variables is purely technical: for generating equations (Definition 10), we need to associate a type $\widehat{x}$ to every variable x, and we need those variables $\widehat{x}$ to be distinct from one another and distinct from the variables

used in rules. To do so, we split the set of variables into two disjoint sets. The ones used in rules are called algebraic, and the others are called non-algebraic. Finally, we ask the function $\widehat{\ }$ to be an injection from the set of algebraic variables to the set of non-algebraic variables.

In the rest of the paper, we also assume that rule left-hand sides are patterns. Hence, every rule is of the form $f\,l_1 \dots l_n \hookrightarrow r$, and we say that a symbol $f \in \mathcal{F}$ is defined if there is in $\mathcal{R}$ a rule of the form $f\,l_1 \dots l_n \hookrightarrow r$.

However, the condition (*) is not satisfactory in the context of dependent types. Indeed, when function symbols have dependent types, it often happens that a term is typable only if it is non-linear. And, with non-left-linear rewriting rules, $\hookrightarrow$ is generally not confluent on untyped terms [13], while there exist many confluence criteria for left-linear rewriting systems [21].

Throughout the paper, we will use the following simple but paradigmatic example to illustrate how our new algorithm works:

▶ **Example 9.** Consider the following rule to define the *tail* function on vectors:

$$tail\ n\ (cons\ x\ p\ v) \hookrightarrow v$$

where $tail : \Pi n : N, V(sn) \to Vn$, $V : N \to \star$, $nil : V0$, $cons : R \to \Pi n : N, Vn \to V(sn)$ and $R : \star$.

For the left-hand side to be typable, we need to take $p = n$, because $tail\ n$ expects an argument of type $V(sn)$, but $cons\ x\ p\ v$ is of type $V(sp)$.

Yet, the rule with $p \neq n$ preserves typing. Indeed, assume that there is an environment Γ, a substitution σ and a term A such that $\Gamma \vdash tail\ n\sigma\ (cons\ x\sigma\ p\sigma\ v\sigma) : A$. By inversion of typing rules, we get $V(n\sigma) \simeq A$, $\Gamma \vdash A : s$ for some sort s, $V(sp\sigma) \simeq V(sn\sigma)$ and $\Gamma \vdash v\sigma : Vp\sigma$. Assume now that V and s are undefined, that is, there is no rule of $\mathcal{R}$ of the form $Vt \hookrightarrow u$ or $st \hookrightarrow u$. Then, by confluence, $p\sigma \simeq n\sigma$. Therefore, $Vp\sigma \simeq A$ and $\Gamma \vdash v\sigma : A$.

Hence, that a rewriting rule $l \hookrightarrow r$ preserves typing does not mean that its left-hand side l must be typable [6]. Actually, if no instance of l is typable, then $l \hookrightarrow r$ trivially preserves typing (since it can never be applied)! The point is therefore to check that any typable instance of $l \hookrightarrow r$ preserves typing.

4 A new subject-reduction criterion

The new criterion that we propose for checking that $l \hookrightarrow r$ preserves typing proceeds in two steps. First, we generate conversion constraints that are satisfied by every typable instance of l (Figure 2). Then, we try to check that r has the same type as l in the type system where the conversion relation is extended with the equational theory generated by the conversion constraints inferred in the first step. For type-checking in this extended type theory to be decidable and implementable using Dedukti itself, we use Knuth-Bendix completion [15] to replace the set of conversion constraints by an equivalent but convergent (*i.e.* terminating and confluent) set of rewriting rules.

4.1 Inference of typability constraints

We first define an algorithm for inferring typability constraints and then prove its correctness and completeness.

▶ **Definition 10** (Typability constraints). *For every algebraic term t, we assume given a valid environment $\Delta_t = \widehat{y_1} : \star, y_1 : \widehat{y_1}, \ldots, \widehat{y_k} : \star, y_k : \widehat{y_k}$ where $y_1, \ldots, y_k$ are the free variables of t.*

Let $\uparrow$ be the partial function defined in Figure 2. It takes as input a term t and returns a pair $(A, \mathcal{E})$, written $A[\mathcal{E}]$, where A is a term and $\mathcal{E}$ is a set of equations, an equation being a pair of terms (l, r) usually written $l = r$.

A substitution σ satisfies a set $\mathcal{E}$ of equations, written $\sigma \models \mathcal{E}$, if for all equations $a = b \in \mathcal{E}$, $a\sigma \simeq b\sigma$.

Figure 2 Typability constraints

$$\frac{}{y \uparrow \widehat{y}[\emptyset]}$$

$$\frac{f : \Pi x_1 : T_1, \ldots, \Pi x_n : T_n, U \quad t_1 \uparrow A_1[\mathcal{E}_1] \quad t_n \uparrow A_n[\mathcal{E}_n]}{f t_1 \ldots t_n \uparrow U\sigma[\mathcal{E}_1 \cup \ldots \cup \mathcal{E}_n \cup \{A_1 = T_1\sigma, \ldots, A_n = T_n\sigma\}]}$$
$$\text{where } \sigma = \{(x_1, t_1), \ldots, (x_n, t_n)\}$$

▶ **Example 11.** In our running example $\mathit{tail}\ n\ (\mathit{cons}\ x\ p\ v) \hookrightarrow v$, we have $\mathit{cons}\ x\ p\ v \uparrow V(sp)[\mathcal{E}_1]$ with $\mathcal{E}_1 = \{\widehat{x} = T, \widehat{p} = N, \widehat{v} = Vp\}$, and $\mathit{tail}\ n\ (\mathit{cons}\ x\ p\ v) \uparrow Vn[\mathcal{E}_2]$ with $\mathcal{E}_2 = \mathcal{E}_1 \cup \{\widehat{n} = N, V(sp) = V(sn)\}$.

▶ **Lemma 12.** *If $\Gamma \vdash \Pi x_1 : T_1, \ldots, \Pi x_n : T_n, U : s$ then, for all i, $\Gamma^{i-1} \vdash T_i : \star$ and $\Gamma^n \vdash U : s$, where $\Gamma^i = \Gamma, x_1 : T_1, \ldots, x_i : T_i$.*

Proof. Since $\hookrightarrow$ is confluent and left-hand sides are patterns, $s \simeq s'$ iff $s = s'$. The result follows then by inversion of typing rules and weakening. ◀

In particular, because $\vdash \Theta_f : \Sigma_f$ for all f, we have:

▶ **Corollary 13.** *For all function symbols $f : \Pi x_1 : T_1, \ldots, \Pi x_n : T_n, U$ and integer i, we have $\Gamma_f^{i-1} \vdash T_i : \star$ and $\Gamma_f^n \vdash U : \Sigma_f$ where $\Gamma_f^i = x_1 : T_1, \ldots, x_i : T_i$.*

▶ **Lemma 14.** *For all environments Γ, terms $t, x_1, T_1, \ldots, x_n, T_n, U, T$ and substitutions σ for $x_1, \ldots, x_n$, if $\Gamma \vdash t : \Pi x_1 : T_1, \ldots, \Pi x_n : T_n, U$ and $\Gamma \vdash t x_1\sigma \ldots x_n\sigma : T$, then $U\sigma \simeq T$ and $\Gamma \vdash \sigma : \Delta^n$ where $\Delta^n = x_1 : T_1, \ldots, x_n : T_n$.*

Proof. Let $\sigma_i = \{(x_1, t_1), \ldots, (x_{i-1}, t_{i-1})\}$. We proceed by induction on n.

- Case $n = 0$. By equivalence of types.
- Case $n > 0$. By inversion of typing rules and weakening, $\Gamma, \Delta^{n-1} \vdash T_n : \star$, $\Gamma \vdash t x_1\sigma_{n-1} \ldots x_{n-1}\sigma_{n-1} : \Pi x_n : A, B$, $\Gamma \vdash x_n\sigma : A$ and $B\{(x_n, x_n\sigma)\} \simeq T$. By induction hypothesis, $\Gamma \vdash \sigma_{n-1} : \Delta^{n-1}$ and $(x_n : T_n\sigma_{n-1})U\sigma_{n-1} \simeq \Pi x_n : A, B$. By substitution, $\Gamma \vdash T_n\sigma_{n-1} : \star$. By confluence, $T_n\sigma_{n-1} \simeq A$ and $U\sigma_{n-1} \simeq B$. Therefore, by conversion, $\Gamma \vdash x_n\sigma : T_n\sigma$ and $\Gamma \vdash \sigma : \Delta^n$. Now, x_n can always be chosen so that $U\sigma = U\sigma_{n-1}\{(x_n, x_n\sigma)\}$. Therefore, $U\sigma \simeq B\{(x_n, x_n\sigma)\} \simeq T$.

◀

▶ **Lemma 15.** *(Correctness) For all algebraic terms t, terms T and sets of equations $\mathcal{E}$, if $t \uparrow T[\mathcal{E}]$ then, for all valid environments Γ, substitutions $\widehat{\theta}$ such that $\Gamma \vdash \widehat{\theta} : \Delta_t$ and $\widehat{\theta} \models \mathcal{E}$, we have $\Gamma \vdash t\widehat{\theta} : T\widehat{\theta}$.*

- *(Completeness) For all environments Γ, patterns t, substitutions θ and terms A, if $\Gamma \vdash t\theta : A$, then there are a term T, a set of equations $\mathcal{E}$ and a substitution $\widehat{\theta}$ extending θ such that $t \uparrow T[\mathcal{E}]$, $\widehat{\theta} \models \mathcal{E}$, $\Gamma \vdash \widehat{\theta} : \Delta_t$ and $A \simeq T\widehat{\theta}$.*

Proof.
- (Correctness) By induction on t.
  - Case $t = y$. Then, $T = \widehat{y}$ and $\mathcal{E} = \emptyset$. By assumption, we have $\Gamma \vdash y\theta : \widehat{y}\theta$. Therefore, $\Gamma \vdash t\theta : T\theta$.
  - Case $t = ft_1 \dots t_n$ with $f : \Pi x_1 : T_1, \dots, x_n : T_n, U$, $t_1 \uparrow A_1[\mathcal{E}_1]$, $\dots$, $t_n \uparrow A_n[\mathcal{E}_n]$. Then, $T = U\sigma$ and $\mathcal{E} = \mathcal{E}_1 \cup \dots \cup \mathcal{E}_n \cup \{A_1 = T_1\sigma, \dots, A_n = T_n\sigma\}$ where $\sigma = \{(x_1, t_1), \dots, (x_n, t_n)\}$.
    By Lemma 12, we have $\Gamma_f^{i-1} \vdash T_i : \star$.
    By induction hypothesis, for all i, we have $\Gamma \vdash t_i\widehat{\theta} : A_i\widehat{\theta}$ and $A_i\widehat{\theta} \simeq T_i\sigma\widehat{\theta}$.
    We now prove that, for all i, $\Gamma \vdash T_i\sigma\widehat{\theta} : \star$ and $\Gamma \vdash x_i\sigma\widehat{\theta} : T_i\sigma\widehat{\theta}$, hence that $\Gamma \vdash \sigma : \Gamma_f^i$, by induction on i.
    * Case $i = 1$. Since $\vdash T_1 : \star$, T_1 is closed and $T_1\sigma\widehat{\theta} = T_1$. Therefore, by weakening, $\Gamma \vdash T_1\sigma\widehat{\theta} : \star$ and, by conversion, $\Gamma \vdash x_1\sigma\widehat{\theta} : T_1\sigma\widehat{\theta}$.
    * Case $i > 1$. By induction hypothesis, $\Gamma \vdash \sigma\widehat{\theta} : \Gamma_f^{i-1}$. Since $\Gamma_f^{i-1} \vdash T_i : \star$, by substitution, we get $\Gamma \vdash T_i\sigma\widehat{\theta} : \star$. Therefore, by conversion, $\Gamma \vdash x_i\sigma\widehat{\theta} : T_i\sigma\widehat{\theta}$.

    Hence, $\Gamma \vdash \sigma\widehat{\theta} : \Gamma_f^n$. Now, since $\Gamma_f^n \vdash fx_1 \dots x_n : U$, by substitution, we get $\Gamma \vdash t : U\sigma\widehat{\theta}$.
- (Completeness) We first prove completeness for object-level algebraic terms t such that $\Gamma \vdash A : \star$, by induction on t.
  - Case $t = y$. We take $T = \widehat{y}$, $\mathcal{E} = \emptyset$ and $\widehat{\theta} = \theta \cup \{(\widehat{y}, A)\}$. We have $t \uparrow T[\mathcal{E}]$, $\widehat{\theta} \models \mathcal{E}$ and $A \simeq T\widehat{\theta}$. Now, $\Gamma \vdash y\widehat{\theta} : \widehat{y}\widehat{\theta}$ and $\Gamma \vdash \widehat{y}\widehat{\theta} : \star$. Therefore, $\Gamma \vdash \widehat{\theta} : \Delta_t$.
  - Case $t = ft_1 \dots t_n$ with $f : \Pi x_1 : T_1, \dots, x_n : T_n, U$. By Lemma 12, for all i, we have $\Gamma_f \vdash x_i : T_i$ and $\Gamma_f \vdash T_i : \star$, where $\Gamma_f = x_1 : T_1, \dots, x_n : T_n$. By Lemma 14 because $\vdash \Theta_f : \Sigma_f$ for all f, we have $A \simeq U\sigma\theta$ and $\Gamma \vdash \sigma\theta : \Gamma_f$. Hence, by substitution, for all i, we have $\Gamma \vdash t_i\theta : T_i\sigma\theta$ and $\Gamma \vdash T_i\sigma\theta : \star$. Therefore, by induction hypothesis, there are A_i, $\mathcal{E}_i$ and $\widehat{\theta}_i$ extending θ such that $t_i \uparrow A_i[\mathcal{E}_i]$, $\widehat{\theta}_i \models \mathcal{E}_i$, $\Gamma \vdash \widehat{\theta}_i : \Delta_{t_i}$ and $T_i\sigma\theta \simeq A_i\widehat{\theta}_i$. Then, let $T = U\sigma$, $\mathcal{E} = \mathcal{E}_1 \cup \dots \cup \mathcal{E}_n \cup \{(A_1, T_1\sigma), \dots, (A_n, T_n\sigma)\}$, $y\widehat{\theta} = y\theta$ if $y \in \mathrm{FV}(t)$, and $\widehat{y}\widehat{\theta} = \widehat{y}\widehat{\theta}_i$ where i is the smallest integer such that $y \in \mathrm{FV}(t_i)$. Then, we have $t \uparrow T[\mathcal{E}]$ and $A \simeq U\sigma\theta = T\widehat{\theta}$.
    If $y \in \mathrm{FV}(t_i) \cap \mathrm{FV}(t_j)$, then $y\widehat{\theta}_i = y\theta = y\widehat{\theta}_j$ since $\widehat{\theta}_i$ and $\widehat{\theta}_j$ are both extensions of θ. Now, if $\Gamma \vdash y\widehat{\theta}_i : \widehat{y}\widehat{\theta}_i$ and $\Gamma \vdash y\widehat{\theta}_j : \widehat{y}\widehat{\theta}_j$ then, by equivalence of types, $\widehat{y}\widehat{\theta}_i \simeq \widehat{y}\widehat{\theta}_j$. Therefore, $\widehat{\theta} \models \mathcal{E}$ and $\Gamma \vdash \widehat{\theta} : \Delta_t$.

  Let now t be a pattern. By definition, t is of the form $ft_1 \dots t_n$ with $f : \Pi x_1 : T_1, \dots, x_n : T_n, U$ and each t_i an object-level algebraic term. As we have seen above, for all i, we have $\Gamma \vdash t_i\theta : T_i\sigma\theta$ and $\Gamma \vdash T_i\sigma\theta : \star$. Therefore, by completeness for object level algebraic terms, there are A_i, $\mathcal{E}_i$ and $\widehat{\theta}_i$ extending θ such that $t_i \uparrow A_i[\mathcal{E}_i]$, $\widehat{\theta}_i \models \mathcal{E}_i$, $\Gamma \vdash \widehat{\theta}_i : \Delta_{t_i}$ and $T_i\sigma\theta \simeq A_i\widehat{\theta}_i$. We can now conclude like in the previous case.

◀

▶ **Example 16.** In our running example $tail\ n\ (cons\ x\ p\ v) \hookrightarrow v$, we have seen that $cons\ x\ p\ v \uparrow V(sp)[\mathcal{E}_1]$ with $\mathcal{E}_1 = \{\widehat{x} = T, \widehat{p} = N, \widehat{v} = Vp\}$, and $tail\ n\ (cons\ x\ p\ v) \uparrow Vn[\mathcal{E}_2]$ with $\mathcal{E}_2 = \mathcal{E}_1 \cup \{\widehat{n} = N, V(sp) = V(sn)\}$. This means that, if σ is a substitution and $(tail\ n\ (cons\ x\ p\ v))\sigma$ is typable, then $\sigma \models \mathcal{E}_2$. In particular, $V(sp\sigma) \simeq V(sn\sigma)$.

4.2 Type-checking modulo typability constraints

For checking that the right-hand side of a rewriting rule $l \hookrightarrow r$ has the same type as the left-hand side modulo the typability constraints $\mathcal{E}$ of the left hand-side, we introduce a new

$\lambda\Pi$-calculus as follows:

▶ **Definition 17.** *Given a pattern l and a set of equations $\mathcal{E}$ such that $\mathcal{R}$ contains no variable of $\{x \mid x \in \mathrm{FV}(l)\} \cup \{\widehat{x} \mid x \in \mathrm{FV}(l)\}$*[3]*, we define a new $\lambda\Pi$-calculus $\Lambda_{l,\mathcal{E}} = (\mathcal{F}', \mathcal{V}', \Theta', \Sigma', \mathcal{R}')$ where:*

- $\mathcal{F}' = \mathcal{F} \cup \{x \mid x \in \mathrm{FV}(l)\} \cup \{\widehat{x} \mid x \in \mathrm{FV}(l)\}$
- $\mathcal{V}' = \mathcal{V} - (\{x \mid x \in \mathrm{FV}(l)\} \cup \{\widehat{x} \mid x \in \mathrm{FV}(l)\})$
- $\Theta' = \Theta \cup \{(x, \widehat{x}) \mid x \in \mathrm{FV}(l)\} \cup \{(\widehat{x}, \star) \mid x \in \mathrm{FV}(l)\}$
- $\Sigma' = \Sigma \cup \{(x, \star) \mid x \in \mathrm{FV}(l)\} \cup \{(\widehat{x}, \Box) \mid x \in \mathrm{FV}(l)\}$
- $\mathcal{R}' = \mathcal{R} \cup \mathcal{E} \cup \mathcal{E}^{-1}$, *where $l = r \in \mathcal{E}^{-1}$ iff $r = l \in \mathcal{E}$.*

We denote by $\simeq_{l,\mathcal{E}}$ the conversion relation of $\Lambda_{l,\mathcal{E}}$, and by $\vdash_{l,\mathcal{E}}$ its typing relation.

$\Lambda_{l,\mathcal{E}}$ is similar to Λ except that the symbols of $\{x \mid x \in \mathrm{FV}(l)\} \cup \{\widehat{x} \mid x \in \mathrm{FV}(l)\}$ are not variables but function symbols, and that the set of rewriting rules is extended by $\mathcal{E} \cup \mathcal{E}^{-1}$ which, in $\Lambda_{l,\mathcal{E}}$, is a set of closed rewriting rules (rules and equations are synonyms: they both are pairs of terms).

▶ **Lemma 18.** *For all patterns l, sets of equations $\mathcal{E}$ and substitutions σ in Λ, and for all terms t, u in $\Lambda_{l,\mathcal{E}}$, if $\sigma \models \mathcal{E}$ and $t \simeq_{l,\mathcal{E}} u$, then $t\sigma \simeq u\sigma$.*[4]

Proof. Immediate as each application of an equation $(a, b) \in \mathcal{E} \cup \mathcal{E}^{-1}$ can be replaced by a conversion $a \simeq b$. ◀

▶ **Theorem 19.** *For all patterns l, sets of equations $\mathcal{E}$, and terms T, r in Λ, if $l \uparrow T[\mathcal{E}]$ and $\vdash_{l,\mathcal{E}} r : T$, then $l \hookrightarrow r$ preserves typing in Λ.*

Proof. Let Δ be an environment, σ be a substitution and A be a term of Λ such that $\Delta \vdash l\sigma : A$. By Lemma 15 (completeness), there are a term T', a set of equations $\mathcal{E}'$ and a substitution $\widehat{\sigma}$ extending σ such that $t \uparrow T'[\mathcal{E}']$, $\widehat{\sigma} \models \mathcal{E}'$, $\Delta \vdash \widehat{\sigma} : \Delta_t$ and $A \simeq T'\widehat{\sigma}$. Since $\uparrow$ is a function, we have $T' = T$ and $\mathcal{E}' = \mathcal{E}$.

We now prove that, if $\Gamma \vdash_{l,\mathcal{E}} t : T$, then $\Delta, \Gamma\widehat{\sigma} \vdash t\widehat{\sigma} : T\widehat{\sigma}$, by induction on $\vdash_{l,\mathcal{E}}$ (note that $\widehat{\sigma}$ replaces function symbols by terms).

(fun) $\dfrac{\vdash_{l,\mathcal{E}} \Theta'_f : \Sigma'_f}{\vdash_{l,\mathcal{E}} f : \Theta'_f}$. By induction hypothesis, we have $\Delta \vdash \Theta'_f\widehat{\sigma} : \Sigma'_f\widehat{\sigma} = \Sigma'_f$.

- Case $f \in \mathcal{F}$. Then, $f\widehat{\sigma} = f$, $\Theta'_f\widehat{\sigma} = \Theta'_f = \Theta_f$ and $\Sigma'_f = \Sigma_f$. By inverting typing rules, we get $\vdash \Theta_f : \Sigma_f$. Therefore, by (fun) and (weak), $\Delta \vdash f : \Theta_f$, that is, $\Delta \vdash f\widehat{\sigma} : \Theta_f\widehat{\sigma}$.
- Case $f = x \in \mathrm{FV}(l)$. Then, $f\widehat{\sigma} = x\sigma$ and $\Theta'_f\widehat{\sigma} = \widehat{x}\widehat{\sigma}$. Therefore, $\Delta \vdash f\widehat{\sigma} : \Theta_f\widehat{\sigma}$ since $\Delta \vdash x\widehat{\sigma} : \widehat{x}\widehat{\sigma}$.
- Case $f = \widehat{x}$ with $x \in \mathrm{FV}(l)$. Then, $f\widehat{\sigma} = \widehat{x}\widehat{\sigma}$ and $\Theta'_f\widehat{\sigma} = \star$. Therefore, $\Delta \vdash f\widehat{\sigma} : \Theta_f\widehat{\sigma}$ since $\Delta \vdash \widehat{x}\widehat{\sigma} : \star$.

(conv) $\dfrac{\Gamma \vdash_{l,\mathcal{E}} t : T \quad T \simeq_{l,\mathcal{E}} U \quad \Gamma \vdash_{l,\mathcal{E}} U : s}{\Gamma \vdash_{l,\mathcal{E}} t : U}$. By induction hypothesis, $\Delta, \Gamma\widehat{\sigma} \vdash t\widehat{\sigma} : T\widehat{\sigma}$ and $\Delta, \Gamma\widehat{\sigma} \vdash U\widehat{\sigma} : s$. By Lemma 18, $T\widehat{\sigma} \simeq U\widehat{\sigma}$ since $\widehat{\sigma} \models \mathcal{E}$. Hence, by (conv), $\Delta, \Gamma\widehat{\sigma} \vdash t\widehat{\sigma} : U\widehat{\sigma}$.

- The other cases follow easily by induction hypothesis.

Hence, we have $\Delta \vdash r\widehat{\sigma} : T\widehat{\sigma}$. Since $\mathrm{FV}(r) \subseteq \mathrm{FV}(l)$, we have $r\widehat{\sigma} = r\sigma$. Since $\Delta \vdash l\sigma : A$, by Lemma 6, either $\Delta \vdash A : s$ for some sort s, or $A = \Box$ and $l\sigma$ is of the form $\Pi x_1 : A_1, \ldots, \Pi x_k : A_k, \star$. Since l is a pattern, l is of the form $f l_1 \ldots l_n$. Therefore, $\Delta \vdash A : s$ and, by (conv), $\Delta \vdash r\sigma : A$. ◀

[3] This can always be done by renaming variables.
[4] Note that, here, we extend the notion of substitution by taking maps on $\mathcal{V} \cup \mathcal{F}$.

▶ **Example 20.** We have seen that $cons\ x\ p\ v \uparrow V(sp)[\mathcal{E}_1]$ with $\mathcal{E}_1 = \{\widehat{x} = T, \widehat{p} = N, \widehat{v} = Vp\}$, and $tail\ n\ (cons\ x\ p\ v) \uparrow Vn[\mathcal{E}_2]$ with $\mathcal{E}_2 = \mathcal{E}_1 \cup \{\widehat{n} = N, V(sp) = V(sn)\}$. After the previous theorem, the rewriting rule defining *tail* preserves typing if we can prove that $\vdash_{l,\mathcal{E}_2} v : Vn$ where, in $\Lambda_{l,\mathcal{E}_2}$, v is a function symbol of type $\widehat{v}$ and sort $\star$, and types are identified modulo $\simeq$ and the equations of $\mathcal{E}_2$. But this is not possible since $v : Vp$ and $Vp \not\simeq_{l,\mathcal{E}_2} Vn$. Yet, if $\sigma \models V(sp) = V(sn)$ and V and s are undefined then, by confluence, $\sigma \models p = n$ and thus $\sigma \models Vp = Vn$. We therefore need to simplify the set of equations before type-checking the right-hand side.

4.3 Simplification of typability constraints

In this section, we show that Theorem 19 can be generalized by using any valid simplification relation, and give an example of such a relation.

▶ **Definition 21** (Valid simplification relation)**.** *A relation $\rightsquigarrow$ on sets of equations is valid if, for all sets of equations $\mathcal{D}, \mathcal{D}'$ and substitutions σ, if $\sigma \models \mathcal{D}$ and $\mathcal{D} \rightsquigarrow \mathcal{D}'$, then $\sigma \models \mathcal{D}'$.*

Theorem 19 can be easily generalized as follows:

▶ **Theorem 22** (Preservation of typing)**.** *For all patterns l, sets of equations $\mathcal{D}, \mathcal{E}$, and terms T, r in Λ, if $l \uparrow T[\mathcal{D}]$, $\mathcal{D} \rightsquigarrow^* \mathcal{E}$ and $\vdash_{l,\mathcal{E}} r : T$, then $l \hookrightarrow r$ preserves typing in Λ.*

We have seen in the previous example that, thanks to confluence, $\sigma \models p = n$ whenever $\sigma \models sp = sn$ and s is undefined. But this last condition is a particular case of a more general property:

▶ **Definition 23** (I-injectivity)**.** *Given $f : \Pi x_1 : T_1, \ldots, \Pi x_n : T_n, U$ and a set $I \subseteq \{1, \ldots, n\}$, we say that f is I-injective when, for all $t_1, u_1, \ldots, t_n, u_n$, if $ft_1 \ldots t_n \simeq fu_1 \ldots u_n$ and, for all $i \notin I$, $t_i \simeq u_i$, then, for all $i \in I$, $t_i \simeq u_i$.*

For instance, f is $\{1, \ldots, n\}$-injective if f is undefined. The new version of Dedukti allows users to declare if a function symbol is I-injective (like the function τ in Example 1), and a procedure for checking I-injectivity of function symbols defined by rewriting rules has been developed and implemented in Dedukti [22]. For instance, the function symbol τ of Example 1, which is defined by the rule $\tau(\mathrm{arr}\, x\, y) \hookrightarrow \tau x \to \tau y$, can be proved to be $\{1\}$-injective.

Clearly, I-injectivity can be used to define a valid simplification relation. In fact, one can easily check that the following simplification rules are valid too:

▶ **Lemma 24.** *The relation defined in Figure 3 is a valid simplification relation.*

Proof. We only detail the first rule which says that, if some substitution σ validates some equation $t = u$, that is, if $t\sigma \simeq u\sigma$, then σ validates any equation $t' = u'$ where t' and u' are reducts of t and u respectively. Indeed, since t' is a reduct of t, $t' \simeq t$. Similarly, $u' \simeq u$. Therefore, by stability of conversion by substitution and transitivity, $t'\sigma \simeq u'\sigma$. ◀

Figure 3 Some valid simplification rules on typability constraints

$$
\begin{array}{rcl}
\mathcal{D} \uplus \{t = u\} & \rightsquigarrow & \mathcal{D} \cup \{t' = u'\} \text{ if } t \hookrightarrow^* t' \text{ and } u \hookrightarrow^* u' \\
\mathcal{D} \uplus \{\Pi x : t_1, t_2 = \Pi x : u_1, u_2\} & \rightsquigarrow & \mathcal{D} \cup \{t_1 = u_1, t_2 = u_2\} \text{ if } x \text{ is fresh} \\
\mathcal{D} \uplus \{ft_1 \ldots t_n = fu_1 \ldots u_n\} & \rightsquigarrow & \mathcal{D} \cup \{t_i = u_i \mid i \in I\} \\
 & & \text{if } f \text{ is } I\text{-injective and } \forall i \notin I, t_i \simeq_{l,\mathcal{D}} u_i
\end{array}
$$

▶ **Example 25.** We can now handle our running example. We have $cons\ x\ p\ v \uparrow V(sp)[\mathcal{E}_1]$ with $\mathcal{E}_1 = \{\widehat{x} = T, \widehat{p} = N, \widehat{v} = Vp\}$, $l = tail\ n\ (cons\ x\ p\ v) \uparrow Vn[\mathcal{E}_2]$ with $\mathcal{E}_2 = \mathcal{E}_1 \cup \{\widehat{n} = N, V(sp) = V(sn)\}$, and $\mathcal{E}_2 \rightsquigarrow^* \mathcal{E}_2' = \mathcal{E}_1 \cup \{\widehat{n} = N, p = n\}$ since V and s are $\{1\}$-injective. Therefore, $\vdash_{l,\mathcal{E}_2'} v : Vn$ and $l \hookrightarrow v$ preserves typing.

The above simplification relation works for the rewriting rule defining *tail* but may not be sufficient in more general situations:

▶ **Example 26.** Let $\mathcal{D}$ be the set of equations $\{fct = ga, fcu = gb, a = b\}$ and assume that f is $\{2\}$-injective. Then the equation $t = u$ holds as well, but $\mathcal{D}$ cannot be simplified by the above rules because it contains no equation of the form $fct = fcu$.

We leave for future work the development of more general simplification relations.

4.4 Decidability conditions

We now discuss the decidability of type-checking in $\Lambda_{l,\mathcal{E}}$ and of the simplification relation based on injectivity, assuming that $\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R}$ is terminating and confluent so that type-checking is decidable in Λ. In both cases, we have to decide $\simeq_{l,\mathcal{E}}$, the reflexive, symmetric and transitive closure of $\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R} \cup \hookrightarrow_\mathcal{E} \cup \hookrightarrow_{\mathcal{E}^{-1}}$, where $\mathcal{E}$ is a set of closed equations.

As it is well known, an equational theory is decidable if there exists a convergent (*i.e.* terminating and confluent) rewriting system having the same equational theory: to decide whether two terms are equivalent, it suffices to check that their normal forms are identical.

In [15], Knuth and Bendix introduced a procedure to compute a convergent rewriting system included in some termination ordering, when equations are algebraic. Interestingly, this procedure always terminates when equations are closed, if one takes a termination ordering that is total on closed terms like the lexicographic path ordering $>_{lpo}$ wrt any total order $>$ on function symbols (for more details, see for instance [2]).

For the sake of self-contentness, we recall in Figure 4 a rule-based definition of closed completion. These rules operate on a pair $(\mathcal{E}, \mathcal{D})$ made of a set of equations $\mathcal{E}$ and a set of rules $\mathcal{D}$. Starting from $(\mathcal{E}, \emptyset)$, completion consists in applying these rules as long as possible. This process necessarily ends on $(\emptyset, \mathcal{D})$ where $\mathcal{D}$ is terminating (because $\mathcal{D} \subseteq >_{lpo}$) and confluent (because it has no critical pairs).

Figure 4 Rules for closed completion

$$
\begin{array}{rcl}
(\mathcal{E} \uplus \{l = r\}, \mathcal{D}) & \rightsquigarrow & (\mathcal{E}, \{l \hookrightarrow r\} \cup \mathcal{D}) \text{ if } l > r \\
(\mathcal{E} \uplus \{l = r\}, \mathcal{D}) & \rightsquigarrow & (\mathcal{E}, \{r \hookrightarrow l\} \cup \mathcal{D}) \text{ if } l < r \\
(\mathcal{E} \uplus \{t = t\}, \mathcal{D}) & \rightsquigarrow & (\mathcal{E}, \mathcal{D}) \\
(\mathcal{E}, \{l[g] \hookrightarrow r, g \hookrightarrow d\} \uplus \mathcal{D}) & \rightsquigarrow & (\mathcal{E} \cup \{l[d] = r\}, \{g \hookrightarrow d\} \cup \mathcal{D}) \\
(\mathcal{E}, \{l \hookrightarrow r[g], g \hookrightarrow d\} \uplus \mathcal{D}) & \rightsquigarrow & (\mathcal{E}, \{l \hookrightarrow r[d], g \hookrightarrow d\} \cup \mathcal{D})
\end{array}
$$

We leave for future work the extension of this procedure to the case of non-algebraic, and possibly higher-order, equations.

If we apply this procedure to the set $\mathcal{E}$ of equations (assuming that they are algebraic), we get that $\simeq_{l,\mathcal{E}}$ is the reflexive, symmetric and transitive closure of $\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R} \cup \hookrightarrow_\mathcal{D}$, where $\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R}$ and $\hookrightarrow_\mathcal{D}$ are both terminating and confluent. However, termination is not modular in general, even when combining two systems having no symbols in common [20].

There exists many results on the modularity of confluence and termination of first-order rewriting systems when these systems have no symbols in common, or just undefined symbols

(see for instance [11] for some survey). But, here, we have higher-order rewriting rules that may share defined symbols.

So, instead, we may try to apply general modularity results on abstract relations [10]. In particular, for all terminating relations P and Q, $P \cup Q$ terminates if P steps can be postponed, that is, if $PQ \subseteq QP^*$. In our case, we may try to postpone the $\mathcal{D}$ steps:

▶ **Lemma 27.** *For all sets of higher-order rewriting rules $\mathcal{R}$ and $\mathcal{D}$, we have that $\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R} \cup \hookrightarrow_\mathcal{D}$ terminates if:*
(a) *$\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R}$ and $\hookrightarrow_\mathcal{D}$ terminate,*
(b) *$\mathcal{R}$ is left-linear,*
(c) *$\mathcal{D}$ is closed,*
(d) *no right-hand side of $\mathcal{D}$ is $\beta\mathcal{R}$-reducible or headed by an abstraction,*
(e) *no right-hand side of $\mathcal{D}$ unifies with a non-variable subterm of a left-hand side of $\mathcal{R}$.*

Proof. As usual, we define positions in a term as words on $\{1, 2\}$: $\mathrm{Pos}(s) = \mathrm{Pos}(x) = \mathrm{Pos}(f) = \{\varepsilon\}$, the empty word representing the root position, and $\mathrm{Pos}(tu) = \mathrm{Pos}(\lambda x : t, u) = \mathrm{Pos}(\Pi x : t, u) = 1 \cdot \mathrm{Pos}(t) \cup 2 \cdot \mathrm{Pos}(u)$.

Assume that $t \hookrightarrow_\mathcal{D} u$ at position p and $u \hookrightarrow_{\beta\mathcal{R}} v$ at position q. If p and q are disjoint, then these reductions can be trivially permuted: $t \hookrightarrow_{\beta\mathcal{R}} \hookrightarrow_\mathcal{D} v$. The case $p \leq q$ (p prefix of q) is not possible since $\mathcal{D}$ is closed (c) and no right-hand side of $\mathcal{D}$ is $\beta\mathcal{R}$-reducible (d). So, we are left with the case $q < p$:

- Case $u \hookrightarrow_\beta v$. The case $p = q1$ is not possible since no right-hand side of $\mathcal{D}$ is headed by an abstraction (d). So, $t|_q$ is of the form $(\lambda x : A, b)a$ and the $\mathcal{D}$ step is in A, b or a. Therefore, $t \hookrightarrow_\beta \hookrightarrow^*_\mathcal{D} v$.
- Case $u \hookrightarrow_\mathcal{R} v$, that is, when $u|_q = l\sigma$ where l is a left-hand side of a rule of $\mathcal{R}$. The case $p = qs$ where s is a non-variable position of l is not possible because no non-variable subterm of a left-hand side of $\mathcal{R}$ unifies with a right-hand side of $\mathcal{D}$ (e). Therefore, since l is left-linear (b), $t|_q$ is of the form $l\theta$ for some substitution θ, and the $\mathcal{D}$ step occurs in some $x\theta$. Hence, $t \hookrightarrow_\mathcal{R} \hookrightarrow^*_\mathcal{D} v$.

◀

▶ **Example 28.** As we have already seen, the typability conditions of $l = \mathit{tail}\ n\ (\mathit{cons}\ x\ p\ v)$ is the set of equations $\mathcal{E} = \{\widehat{x} = T, \widehat{p} = N, \widehat{v} = Vp, \widehat{n} = N, V(sp) = V(sn)\}$. By taking $\widehat{x} > \widehat{v} > \widehat{p} > \widehat{n} > V > T > N > s > p > n$ as total order on function symbols, the Knuth-Bendix completion procedure yields with $>_{lpo}$ the rewriting system $\mathcal{D} = \{\widehat{x} \hookrightarrow T, \widehat{p} \hookrightarrow N, \widehat{v} \hookrightarrow Vp, \widehat{n} \hookrightarrow N, V(sp) \hookrightarrow V(sn)\}$. After Lemma 27, $\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R} \cup \hookrightarrow_\mathcal{D}$ is convergent if $\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R}$ is convergent, $\mathcal{R}$ is left-linear and V and s are undefined. This works as well if, instead of $\mathcal{E}$, we use its simplification $\mathcal{E}' = \{\widehat{x} = T, \widehat{p} = N, \widehat{v} = Vp, \widehat{n} = N, p = n\}$. In this case, we get the rewriting system $\mathcal{D} = \{\widehat{x} \hookrightarrow T, \widehat{p} \hookrightarrow N, \widehat{v} \hookrightarrow Vn, \widehat{n} \hookrightarrow N, p \hookrightarrow n\}$.

▶ **Example 29.** Finally, let's come back to the rewriting rule $\mathrm{app}\, a\, b\, (\mathrm{lam}\, a'\, b'\, f)\, x \hookrightarrow f\, x$ of Example 1 encoding the β-reduction of simply-type λ-calculus. As already mentioned, the previous version of Dedukti was unable to prove that this rule preserves typing. Thanks to our new algorithm, the new version of Dedukti[5] can now do it.

The computability constraints of the LHS are $\widehat{f} = \tau a' \to \tau b'$, $\tau a' \to \tau b' = \tau(\mathrm{arr}\, a\, b)$ and $\widehat{x} = \tau a$. Preservation of typing cannot be proved without simplifying this set of equations to $\widehat{f} = \tau a' \to \tau b'$, $\tau a' = \tau a$, $\tau b' = \tau b$ and $\widehat{x} = \tau a$.

[5] `https://github.com/Deducteam/lambdapi`

Then, any total order on function symbols allows to prove preservation of typing. For instance, by taking $\widehat{f} > \to > a' > a > b' > b$, we get the rewriting rules $\widehat{f} \hookrightarrow \tau a \to \tau b$, $\tau a' \hookrightarrow \tau a$, $\tau b' \hookrightarrow \tau b$ and $\widehat{x} \hookrightarrow \tau a$, so that one can easily check that, modulo these rewriting rules, $f\,x$ has type τb. Therefore, $\mathrm{app}\,a\,b\,(\mathrm{lam}\,a'\,b'\,f)\,x \hookrightarrow f\,x$ preserves typing.

Note that the result does not depend on the total order taken on function symbols. For instance, if one takes $\widehat{f} > \to > a > a' > b' > b$ (flipping the order of a and a'), we get the rewriting rules $\widehat{f} \hookrightarrow \tau a' \to \tau b$, $\tau a \hookrightarrow \tau a'$, $\tau b' \hookrightarrow \tau b$ and $\widehat{x} \hookrightarrow \tau a'$. In this case, $f\,x$ has type τb as well. Flipping the order of b and b' would work as well.

5 Related works and conclusion

The problem of type safety of rewriting rules in dependent type theory modulo rewriting has been first studied for simply-typed function symbols by Barbanera, Fernández and Geuvers in [3]. In [6], the author extended these results to polymorphically and dependently typed function symbols, and showed that rule left-hand sides do not need to be typable for rewriting to preserve typing. This was later studied in more details and implemented in Dedukti by Saillard [17]. In this approach, one first extracts a substitution ρ (called a pre-solution in Saillard's work) from the typability constraints of the left-hand side l and check that, if l is of type A, then the right-hand side r is of type $A\rho$ (in the same system). For instance, from the simplified set of constraints $\mathcal{E}' = \{\widehat{x} = T, \widehat{p} = N, \widehat{v} = Vp, \widehat{n} = N, p = n\}$ of our running example, one can extract the substitution $\rho = \{(n, p)\}$ and check that v has type $(Vn)\rho = Vp$. However, it is not said how to compute useful pre-solutions (note that we can always take the identity as pre-solution). In practice, the pre-solution is often given by the user thanks to annotations in rules. A similar mechanism called inaccessible or "dot" patterns exists in Agda too [16].

An inconvenience of this approach is that, in some cases, no useful pre-solution can be extracted. For instance, if, in the previous example, we take the original set of constraints $\mathcal{E} = \{\widehat{x} = T, \widehat{p} = N, \widehat{v} = Vp, \widehat{n} = N, V(sp) = V(sn)\}$ instead of its simplified version $\mathcal{E}'$, then we cannot extract any useful pre-solution.

In this paper, we proposed a more general approach where we check that the right-hand side has the same type as the left-hand side modulo the equational theory generated by the typability constraints of the left-hand side seen as closed equations (Theorem 19). A prototype implementation is available on:
`https://github.com/wujuihsuan2016/lambdapi/tree/sr`.

To ensure the decidability of type-checking in this extended system, we propose to replace these equations by an equivalent but convergent rewriting system using Knuth-Bendix completion [15] (which always terminates on closed equations), and provide conditions for preserving the termination and confluence of the system when adding these new rules (Lemma 27). This approach has also the advantage that Dedukti itself can be used to check the type safety of user-defined Dedukti rules.

We also showed that, for the algorithm to work, the typability constraints sometimes need to be simplified first, using the fact that some function symbols are injective (Theorem 22). It would be interesting to be able to detect or check injectivity automatically (see [22] for preliminary results on this topic), and also to find a simplification procedure more general than the one of Figure 3.

References

1. A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Dedukti: a logical framework based on the λΠ-calculus modulo theory, 2019. Draft.
2. F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.
3. F. Barbanera, M. Fernández, and H. Geuvers. Modularity of strong normalization in the algebraic-λ-cube. *Journal of Functional Programming*, 7(6):613–660, 1997.
4. H. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of logic in computer science. Volume 2. Background: computational structures*, pages 117–309. Oxford University Press, 1992.
5. F. Blanqui. *Théorie des types et récriture*. PhD thesis, Université Paris-Sud, France, 2001.
6. F. Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005.
7. J. Cockx and A. Abel. Sprinkles of Extensionality for Your Vanilla Type Theory (abstract). Presented at TYPES'2016.
8. D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-Pi-calculus modulo. In *Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science 4583, 2007.
9. Dedukti. `https://deducteam.github.io/`, 2018.
10. H. Doornbos and B. von Karger. On the union of well-founded relations. *Logic Journal of the Interest Group in Pure and applied Logic*, 6(2):195–201, 1998.
11. B. Gramlich. Modularity in term rewriting revisited. *Theoretical Computer Science*, 464:3–19, 2012.
12. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.
13. J. W. Klop. *Combinatory reduction systems*. PhD thesis, Utrecht Universiteit, NL, 1980. Published as Mathematical Center Tract 129.
14. J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.
15. D. Knuth and P. Bendix. Simple word problems in universal algebra. In *Computational problems in abstract algebra, Proceedings of a Conference held at Oxford in 1967*, pages 263–297. Pergamon Press, 1970. Reproduced in [18].
16. U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, Sweden, 2007.
17. R. Saillard. *Type checking in the Lambda-Pi-calculus modulo: theory and practice*. PhD thesis, Mines ParisTech, France, 2015.
18. J. H. Siekmann and G. Wrightson, editors. *Automation of reasoning. 2: classical papers on computational logic 1967-1970*. Symbolic computation. Springer, 1983.
19. TeReSe. *Term rewriting systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.
20. Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25(3):141–143, 1987.
21. V. van Oostrom. *Confluence for abstract and higher-order rewriting*. PhD thesis, Vrije Universiteit Amsterdam, NL, 1994.
22. J.-H. Wu. Checking the type safety of rewrite rules in the λπ-calculus modulo rewriting. `https://hal.inria.fr/hal-02288720`, 2019. Internship report.