Dependency Pairs Termination in Dependent Type Theory Modulo Rewriting

Frédéric Blanqui[1,2], Guillaume Genestier[2,3], and Olivier Hermant[3]

[1] INRIA
[2] LSV, ENS Paris-Saclay, CNRS, Université Paris-Saclay
[3] MINES ParisTech, PSL University

Abstract. Dependency pairs are a key concept at the core of modern automated termination provers for first-order term rewrite systems. In this paper, we introduce an extension of this technique for a large class of dependently-typed higher-order rewrite systems. This improves previous results by Wahlstedt on one hand and the first author on the other hand to strong normalization and non-orthogonal rewrite systems. This new criterion has been implemented in the type-checker Dedukti.

Keywords: Termination · Higher-Order Rewriting · Dependent Types · Dependency Pairs · Size-Change Principle

1 Introduction

Termination, that is, the absence of infinite computations, is an important problem in software verification, as well as in logic. For instance, in logic, termination is often used to prove cut elimination and consistency. In automated theorem provers and proof assistants, it is often used (together with confluence) to check decidability of equational theories and type-checking algorithms.

This paper introduces a new termination criterion for a large class of programs whose operational semantics can be described by higher-order rewrite rules [48] typable in the $\lambda\Pi$-calculus modulo rewriting ($\lambda\Pi/\mathcal{R}$ for short). $\lambda\Pi/\mathcal{R}$ is a system of dependent types where types are identified modulo the β-reduction of λ-calculus and a set $\mathcal{R}$ of rewrite rules given by the user to define not only functions but also types. It extends Barendregt's Pure Type System (PTS) λP [5], the logical framework LF [21], or Martin-Löf's type theory [36]. It can encode any functional PTS like system F or the calculus of constructions [14,3].

Dependent types, introduced by de Bruijn in Automath [11], subsume generalized algebraic data types (GADT) [55] used in some functional programming languages. They are now at the core of many proof assistants and programming languages: Coq, Twelf, Agda, Lean, Idris, ...

Our criterion has been implemented in Dedukti [15], a type-checker for $\lambda\Pi/\mathcal{R}$ that we will use in our examples. The code is available in [16] and could be easily adapted to a subset of other languages like Agda. As far as we know, this tool is the first one to automatically check termination in $\lambda\Pi/\mathcal{R}$, which includes both higher-order rewriting and dependent types.

An important concept in the termination of first-order term rewrite systems is the one of dependency pair. It generalizes the notion of recursive call in first-order functional programs to rewriting. Namely, the dependency pairs of a rewrite rule $f(l_1,\ldots,l_p)\to r$ are the pairs $(f(l_1,\ldots,l_p),g(m_1,\ldots,m_q))$ such that $g(m_1,\ldots,m_q)$ is a subterm of r and g is a function symbol defined by some rewrite rules. Dependency pairs have been introduced by Arts and Giesl [2] and have evolved into a general framework for termination [17]. It is now at the heart of many state-of-the-art automated termination provers for first-order rewrite systems and Haskell, Java or C programs.

Dependency pairs have been extended to different settings for higher-order rewriting: Combinatory Reduction Systems [28,30] and Higher-order Rewrite Systems [37], with two different approaches: dynamic dependency pairs include variable applications, which generates additional dependency pairs [31], while static dependency pairs exclude them by slightly restricting the class of systems that can be considered [46]. Here, we use the static approach.

In [54], Wahlstedt considered a system slightly less general than $\lambda\Pi/\mathcal{R}$ for which he proved the weak normalization property, that is, the existence of a finite reduction to normal form, when $\mathcal{R}$ uses matching on constructors only, like in the languages OCaml or Haskell. In this case, $\mathcal{R}$ is orthogonal: rules are left-linear (no variable occurs twice in a left-hand side) and have no critical pairs (no two rule left-hand side instances overlap). Interestingly, Wahlstedt's proof proceeds in two modular steps. First, he proves that typable terms have a normal form if there is no infinite sequences of function calls. Second, he proves that there is no infinite sequences of function calls if $\mathcal{R}$ satisfies Lee, Jones and Ben-Amram's size-change termination criterion (SCT) [34].

In this paper, we extend Wahlstedt's results in two directions. First, we prove a stronger normalization property: the absence of infinite reductions. Second, we assume that $\mathcal{R}$ is locally confluent, a much weaker condition than orthogonality. Hence, rules can be non-left-linear and have joinable critical pairs.

In [8], the first author developed a termination criterion for a calculus slightly more general than $\lambda\Pi/\mathcal{R}$, based on the notion of computability closure, assuming also that type-level rules are orthogonal. The computability closure of a term $f(l_1,\ldots,l_p)$ is a set of terms that terminate whenever $l_1,\ldots,l_p$ terminate. It is defined inductively thanks to deduction rules preserving this property, using a precedence and a fixed well-founded ordering for dealing with function calls. Termination can then be enforced by requiring each rule right-hand side to belong to the computability closure of its corresponding left-hand side.

We extend this work as well by replacing that fixed ordering by the dependency pair relation. In [8], there must be a decrease in every function call. Using dependency pairs allows one to have non-strict decreases. Then, following Wahlstedt, SCT can be used to enforce the absence of infinite chains of dependency pairs. But other criteria have been developed for this purpose. So, our results could allow the use of other techniques operating on dependency pairs to prove termination in $\lambda\Pi/\mathcal{R}$.

Outline The main result is Theorem 1, stating that the combination of type-preserving rewrite rules and β-reduction is strongly normalizing on typable terms if the signature is finite and the rewrite rules are locally confluent, Plain Function Passing (Def. 6) and Size-Change Terminating (Def. 7).

The proof involves three steps. First, in Section 2, we recall the terms and types of the $\lambda\Pi$-calculus modulo rewriting. Then, in Section 3, we introduce a model of this calculus based on an adaptation of Girard's reducibility candidates and prove that if every symbol of the signature is in the interpretation of its type, then every typable term is strongly normalizing. Second, in Section 4, we introduce our notion of dependency pair and prove that the termination of dependency pairs implies that every symbol of the signature is in the interpretation of its type. Finally, in Section 5, we show how to use size-change termination to obtain termination of the dependency pair relation. In Section 6, we summarize our results and give some hints on possible extensions.

For lack of space, some proofs are given in an annex at the end of the paper.

2 Terms and types

The set $\mathbb{T}$ of terms of $\lambda\Pi/\mathcal{R}$ is the same as those of Barendregt's λP [5]:

$$t \in \mathbb{T} = s \in \mathbb{S} \mid x \in \mathbb{V} \mid f \in \mathbb{F} \mid (x:t)t \mid tt \mid \lambda x:t.t$$

where $\mathbb{S} = \{\star, \Box\}$ is the set of sorts[4], $\mathbb{V}$ is an infinite set of variables and $\mathbb{F}$ is a set of function symbols, so that $\mathbb{S}$, $\mathbb{V}$ and $\mathbb{F}$ are pairwise disjoint. $\star$ is the type of type constructors and $\Box$ the type of their types. The dependent product $(x:A)B$ generalizes the arrow type $A \Rightarrow B$ of simply-typed λ-calculus: it is the type of functions taking an argument x of type A and returning a term whose type B may depend on x, like $\lambda x:A.t$. Let $\boldsymbol{t}$ denote a possibly empty sequence of terms $t_1, \ldots, t_n$ of length $|\boldsymbol{t}| = n$, and $\mathrm{FV}(t)$ be the set of free variables of t.

Given a term T, let its product arity $\alpha(T)$ be the integer $n \in \mathbb{N}$ such that $T = (x_1 : T_1) \ldots (x_n : T_n)U$ and U is not a product.

A typing environment Γ is a (possibly empty) sequence $x_1 : T_1, \ldots, x_n : T_n$ of type declarations for distinct variables, written $\boldsymbol{x} : \boldsymbol{T}$ for short. Given an environment $\Gamma = \boldsymbol{x} : \boldsymbol{T}$ and a term U, let $(\Gamma)U = (\boldsymbol{x} : \boldsymbol{T})U$.

For each $f \in \mathbb{F}$, we assume given a term Θ_f for its type and a sort s_f for its sort. Let Γ_f be the environment such that $\Theta_f = (\Gamma_f)U$ and $|\Gamma_f| = \alpha(\Theta_f)$.

Finally, we assume given a set $\mathcal{R}$ of rules $l \to r$ such that $\mathrm{FV}(r) \subseteq \mathrm{FV}(l)$ and l is of the form $f\boldsymbol{l}$. A symbol f is said to be defined if there is a rule of the form $f\boldsymbol{l} \to r$. In this paper, we are interested in the termination of

$$\to \,=\, \to_\beta \cup \to_\mathcal{R}$$

where $\to_\beta$ is the β-reduction of λ-calculus and $\to_\mathcal{R}$ is the smallest relation containing $\mathcal{R}$ and closed by substitution and context. Note that we consider rewriting with syntactic matching only. Following [9], it should however be possible

[4] Sorts refer here to the notion of sort in Pure Type Systems, not to be confused with the one used in some first-order settings. For instance, usual data types, like $\mathbb{N}$ or $\mathbb{B}$ are represented in $\lambda\Pi/\mathcal{R}$ as functions of type $\star$.

to extend the present results to rewriting with matching modulo $\beta\eta$ or some equational theory. Let SN be the set of terminating terms and, given a term t, let $\rightarrow(t) = \{u \in \mathbb{T} \mid t \rightarrow u\}$ be the set of immediate reducts of t.

The application of a substitution σ to a term t is written $t\sigma$. B_x^a is the term obtained by substituting a for x in B. Given a substitution σ, let $\mathrm{dom}(\sigma) = \{x | x\sigma \neq x\}$, $\mathrm{FV}(\sigma) = \bigcup_{x\in\mathrm{dom}(\sigma)} \mathrm{FV}(x\sigma)$ and $[x \mapsto a, \sigma]$ be the substitution $\{(x,a)\} \cup \{(y,b) \in \sigma \mid y \neq x\}$. Given another substitution σ', let $\sigma \rightarrow \sigma'$ if there is x such that $x\sigma \rightarrow x\sigma'$ and, for all $y \neq x$, $y\sigma = y\sigma'$.

The typing rules of $\lambda\Pi/\mathcal{R}$, in Figure 1, add to those of λP the rule (fun). Moreover, (conv) uses $\downarrow$ instead of $\downarrow_\beta$, where $\downarrow = \rightarrow^* {}^*\!\!\leftarrow$ is the joinability relation and $\rightarrow^*$ the reflexive and transitive closure of $\rightarrow$. We say that t has type T in Γ if $\Gamma \vdash t : T$ is derivable. A substitution σ is well-typed from Δ to Γ, written $\Gamma \vdash \sigma : \Delta$, if, for all $(x : T) \in \Delta$, $\Gamma \vdash x\sigma : T\sigma$ holds. The word "type" is used to denote a term occurring at the right-hand side of a colon in a typing judgment.

Typing induces a hierarchy on terms [7, Lemma 47]. At the top, there is the sort $\Box$ that is not typable. Then, comes the class $\mathbb{K}$ of kinds, whose type is $\Box$: $K = \star \mid (x : t)K$. Then, comes the class of predicates, whose types are kinds. Finally, at the bottom lie (proof) objects whose types are predicates.

Fig. 1. Typing rules of $\lambda\Pi/\mathcal{R}$

$$\text{(ax)}\quad \frac{}{\vdash \star : \Box} \qquad\qquad \text{(abs)}\quad \frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (x : A)B : s}{\Gamma \vdash \lambda x : A.b : (x : A)B}$$

$$\text{(var)}\quad \frac{\Gamma \vdash A : s \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : A \vdash x : A} \qquad\qquad \text{(app)}\quad \frac{\Gamma \vdash t : (x : A)B \quad \Gamma \vdash a : A}{\Gamma \vdash ta : B_x^a}$$

$$\text{(weak)}\quad \frac{\Gamma \vdash A : s \quad \Gamma \vdash b : B \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : A \vdash b : B} \qquad\qquad \text{(conv)}\quad \frac{\Gamma \vdash a : A \quad A \downarrow B \quad \Gamma \vdash B : s}{\Gamma \vdash a : B}$$

$$\text{(prod)}\quad \frac{\Gamma \vdash A : \star \quad \Gamma, x : A \vdash B : s}{\Gamma \vdash (x : A)B : s} \qquad\qquad \text{(fun)}\quad \frac{\vdash \Theta_f : s_f}{\vdash f : \Theta_f}$$

Example 1 *This extensive example, in the new* DEDUKTI *syntax, combines type-level rules (*`El` *decodes datatype codes into actual types), dependent types (*$\mathbb{L}$ *is the polymorphic type of lists parameterized with their length), higher-order variables (*`fil` *is a function filtering elements out of a list along a boolean function* `f`*), and critical pairs (*`fil` *can match a list defined by concatenation).*

This example can be handled neither by [54] nor by [8] because the system is not orthogonal and has no strict decrease in every recursive call. It can however be handled by our new termination criterion and its implementation [16].

```
symbol Set:TYPE      symbol arrow:Set⇒Set⇒Set

symbol El:Set⇒TYPE
```

```
  rule El (arrow a b) → El a ⇒ El b

symbol 𝔹:TYPE     symbol true:𝔹   symbol false:𝔹
symbol ℕ:TYPE     symbol 0:ℕ      symbol s:ℕ⇒ℕ

symbol infix +:ℕ⇒ℕ⇒ℕ
  rule 0 + q → q
  rule (s p) + q → s (p + q)

symbol 𝕃:Set⇒ℕ⇒TYPE
  symbol nil:∀a,𝕃 a 0
  symbol cons:∀a,El a ⇒ ∀p,𝕃 a p ⇒ 𝕃 a (s p)

symbol app:∀a p,𝕃 a p ⇒ ∀q,𝕃 a q ⇒ 𝕃 a (p+q)
  rule app a _ (nil _) q m → m
  rule app a _ (cons _ x p l) q m
       → cons a x (p+q) (app a p l q m)

symbol len_fil:∀a,(El a ⇒ 𝔹) ⇒ ∀p,𝕃 a p ⇒ ℕ
symbol len_fil_aux:𝔹 ⇒ ∀a,(El a ⇒ 𝔹) ⇒ ∀p,𝕃 a p ⇒ ℕ
  rule len_fil a f _ (nil _) → 0
  rule len_fil a f _ (cons _ x p l)
       → len_fil_aux (f x) a f p l
  rule len_fil a f _ (app _ p l q m)
       → (len_fil a f p l) + (len_fil a f q m)
  rule len_fil_aux true  a f p l → s (len_fil a f p l)
  rule len_fil_aux false a f p l → len_fil a f p l

symbol fil:∀a f p l,𝕃 a (len_fil a f p l)
symbol fil_aux:∀b a f,El a ⇒ ∀p l,𝕃 a (len_fil_aux b a f p l)
  rule fil a f _ (nil _) → nil a
  rule fil a f _ (cons _ x p l)  → fil_aux (f x) a f x p l
  rule fil a f _ (app _ p l q m)
       → app a (len_fil a f p l) (fil a f p l)
                (len_fil a f q m) (fil a f q m)
  rule fil_aux false a f x p l → fil a f p l
  rule fil_aux true  a f x p l
       → cons a x (len_fil a f p l) (fil a f p l)
```

Assumptions: We assume that $\rightarrow$ preserves typing (for all Γ, A, t and u, if $\Gamma \vdash t : A$ and $t \rightarrow u$, then $\Gamma \vdash u : A$) and is locally confluent ($\leftarrow\rightarrow \subseteq \downarrow$).

Local confluence implies that every $t \in \mathrm{SN}$ has a unique normal form $t\downarrow$ [39].

Preservation of typing by reduction is undecidable in in general [44]. See [4,8,44] for sufficient conditions.

3 Interpretation of types as reducibility candidates

We aim at proving the termination of the union of two relations, $\rightarrow_\beta$ and $\rightarrow_{\mathcal{R}}$, on the set of well-typed terms (which depends on $\mathcal{R}$). As it is well known,

termination is not a modular property in general [52]. As a $\rightarrow_\beta$ step can generate an $\rightarrow_\mathcal{R}$ step, and vice versa, we cannot expect to be able to prove the termination of $\rightarrow_\beta \cup \rightarrow_\mathcal{R}$ from the termination of $\rightarrow_\beta$ and the termination of $\rightarrow_\mathcal{R}$.

Instead, we prove the termination of $\rightarrow_\beta$ and $\rightarrow_\mathcal{R}$ together. Following Girard [19], we build a model of our calculus by interpreting types into sets of terminating terms. To this end, we need to find an interpretation $[\![\ ]\!]$ having the following properties:

- Because types are identified modulo conversion, we need $[\![\ ]\!]$ to be invariant by reduction: if T is typable and $T \rightarrow T'$, then we must have $[\![T]\!] = [\![T']\!]$.
- In order to handle β-reduction, we need a product type $(x : A)B$ to be interpreted as usual by the set of terms t such that, for all a in the interpretation of A, ta is in the interpretation of B^a_x, that is, we must have $[\![(x : A)B]\!] = \Pi a \in [\![A]\!].\ [\![B^a_x]\!]$ where $\Pi a \in P.\ Q(a) = \{t \mid \forall a \in P, ta \in Q(a)\}$.

A natural idea is thus to take:

$$[\![T]\!] = \text{if } T \in \text{SN and } T\!\downarrow = (x : A)B \text{ then } \Pi a \in [\![A]\!].\ [\![B^a_x]\!] \text{ else SN.}$$

But we do not know how to justify the validity of this definition directly when T is a predicate (if T is a kind, we can proceed by induction on its product arity). Indeed, for $[\![(x : A)B]\!]$ to be well-defined, one needs $[\![A]\!]$ and, for all $a \in [\![A]\!]$, $[\![B^a_x]\!]$ to be already well-defined. A well-known solution, used for encoding partial functions in type theory [12,10], is to simultaneously define the function and its domain (the interpretation of sorts), that is, to start with the partial function defined nowhere and extend its domain step by step.

So, let $\mathbb{I} = \mathcal{F}_p(\mathbb{T}, \mathcal{P}(\mathbb{T}))$ be the set of partial functions from $\mathbb{T}$ to $\mathcal{P}(\mathbb{T})$, and $G : \mathbb{I} \Rightarrow \mathbb{I}$ be the function such that $G(I) = \{(T, F(I)(T)) \mid T \in D(I)\}$ where:

- $D(I) = \left\{ T \in \text{SN} \,\middle|\, \begin{array}{l} \text{if } T \rightarrow^* (x : A)B \text{ then } A \in \text{dom}(I) \\ \text{and } \forall a \in I(A), B^a_x \in \text{dom}(I) \end{array} \right\}$,
- $F(I)(T) = \text{if } T \in \text{SN and } T\!\downarrow = (x : A)B \text{ then } \Pi a \in I(A).\ I(B^a_x) \text{ else SN.}$

Then, we can prove that G has a least fixpoint by using Abian and Brown's fixpoint theorem saying that, on a non-empty strictly inductive poset (*i.e.* every non-empty totally ordered subset has a least upper bound) like $(\mathcal{F}_p(X, Y), \subseteq)$, every monotone function has a least fixpoint [1]. Note that we cannot use Tarski's fixpoint theorem [47] here because $(\mathcal{F}_p(X, Y), \subseteq)$ is not a complete lattice. The proof that G is monotone is given in the annex.

We can then define the interpretation of all types as follows:

Definition 1 (Interpretation of types) *Let $\mathcal{I}$ be the least fixpoint of the function G above and $\mathcal{D} = \text{dom}(\mathcal{I})$. We define the interpretation of a term T wrt to a substitution σ, written $[\![T]\!]_\sigma$ (and simply $[\![T]\!]$ if σ is the identity), as follows:*

- $[\![s]\!]_\sigma = \mathcal{D}$ *if* $s \in \mathbb{S}$,
- $[\![(x : A)K]\!]_\sigma = \Pi a \in [\![A]\!]_\sigma.\ [\![K]\!]_{[x \mapsto a, \sigma]}$ *if* $K \in \mathbb{K}$ *and* $x \notin \text{dom}(\sigma)$,
- $[\![T]\!]_\sigma = \mathcal{I}(T\sigma)$ *if* $T \notin \mathbb{K} \cup \{\Box\}$ *and* $T\sigma \in \mathcal{D}$,
- $[\![T]\!]_\sigma = \text{SN}$ *otherwise.*

A substitution σ *is adequate wrt an environment* Γ*, written* $\sigma \models \Gamma$*, if, for all* $(x : A) \in \Gamma$, $x\sigma \in [\![A]\!]_\sigma$*. A typing map* Θ *is adequate if, for all* $f \in \mathbb{F}$, $f \in [\![\Theta_f]\!]$ *whenever* $\vdash \Theta_f : s_f$ *and* $\Theta_f \in [\![s_f]\!]$*.*

Let $\mathbb{C}$ *be the set of terms* $f\boldsymbol{t}$ *such that* $\vdash \Theta_f : s_f$, $\Theta_f \in [\![s_f]\!]$, $|\boldsymbol{t}| = \alpha(\Theta_f)$ *and, if* $\Gamma_f = \boldsymbol{x} : \boldsymbol{A}$ *and* $\sigma = [\boldsymbol{x} \mapsto \boldsymbol{t}]$*, then* $\sigma \models \Gamma_f$*.*

We can then prove that, for all terms T, $[\![T]\!]$ satisfies Girard's conditions of reducibility candidates [19], called computability predicates here, adapted to rewriting by including in neutral terms every term of the form $f\boldsymbol{t}$ when f is applied to enough arguments wrt $\mathcal{R}$:

Definition 2 (Computability predicates) *A term is neutral if it is of the form* $x\boldsymbol{v}$, $(\lambda x : A.t)u\boldsymbol{v}$ *or* $f\boldsymbol{v}$ *with, for every rule* $f\boldsymbol{l} \to r \in \mathcal{R}$, $|\boldsymbol{l}| \leq |\boldsymbol{v}|$*.*

Let $\mathbb{P}$ *be the set of all the sets of terms* S *(computability predicates) such that (a)* $S \subseteq \mathrm{SN}$*, (b)* $\to(S) \subseteq S$*, and (c)* $t \in S$ *if* t *is neutral and* $\to(t) \subseteq S$*.*

Note that neutral terms are closed by application: if t is neutral, then tu is neutral. Moreover, if t is neutral, then $\to(tu) = \to(t)u \cup t \to(u)$.

One can easily check that SN is a computability predicate.

Note also that a computability predicate is never empty: it contains every neutral term in normal form. In particular, it contains every variable.

We then get the following results (the proofs are given in Annex A):

Lemma 1 *(a) For all terms* T *and substitutions* σ, $[\![T]\!]_\sigma \in \mathbb{P}$*.*
(b) If T *is typable,* $T\sigma \in \mathcal{D}$ *and* $T \to T'$*, then* $[\![T]\!]_\sigma = [\![T']\!]_\sigma$*.*
(c) If T *is typable,* $T\sigma \in \mathcal{D}$ *and* $\sigma \to \sigma'$*, then* $[\![T]\!]_\sigma = [\![T]\!]_{\sigma'}$*.*
(d) If $(x : A)B$ *is typable and* $(x : A\sigma)B\sigma \in \mathcal{D}$*,*
then $[\![(x : A)B]\!]_\sigma = \Pi a \in [\![A]\!]_\sigma.\ [\![B]\!]_{[x \mapsto a, \sigma]}$*.*
(e) If $\Delta \vdash U : s$, $\Gamma \vdash \gamma : \Delta$ *and* $U\gamma\sigma \in \mathcal{D}$*, then* $[\![U\gamma]\!]_\sigma = [\![U]\!]_{\gamma\sigma}$*.*
(f) Given $P \in \mathbb{P}$ *and, for all* $a \in P$, $Q(a) \in \mathbb{P}$ *such that* $Q(a') \subseteq Q(a)$ *if* $a \to a'$*.*
Then, $\lambda x : A.b \in \Pi a \in P.\ Q(a)$ *if* $A \in \mathrm{SN}$ *and, for all* $a \in P$, $b^a_x \in Q(a)$*.*

We can finally prove that our model is adequate, that is, every term of type T belongs to $[\![T]\!]$, if the typing map Θ is adequate. This reduces the termination of well-typed terms to the computability of function symbols.

Lemma 2 *If* Θ *is adequate,* $\Gamma \vdash t : T$ *and* $\sigma \models \Gamma$*, then* $t\sigma \in [\![T]\!]_\sigma$*.*

Proof. First note that, if $\Gamma \vdash t : T$, then either $T = \Box$ or $\Gamma \vdash T : s$ [7, Lemma 28]. Moreover, if $\Gamma \vdash a : A$, $A \downarrow B$ and $\Gamma \vdash B : s$ (the premises of the (conv) rule), then $\Gamma \vdash A : s$ [7, Lemma 42]. Hence, the relation $\vdash$ is unchanged if one adds the premise $\Gamma \vdash A : s$ in (conv), giving the rule (conv'). Similarly, we add the premise $\Gamma \vdash (x : A)B : s$ in (app), giving the rule (app'). We now prove the lemma by induction on $\Gamma \vdash t : T$ using (app') and (conv'):

(ax) It is immediate that $\star \in [\![\Box]\!]_\sigma = \mathcal{D}$.
(var) By assumption on σ.
(weak) If $\sigma \models \Gamma, x : A$, then $\sigma \models \Gamma$. So, the result follows by induction hypothesis.

(prod) Is $((x:A)B)\sigma$ in $[\![s]\!]_\sigma = \mathcal{D}$? Wlog we can assume $x \notin \mathrm{dom}(\sigma) \cup \mathrm{FV}(\sigma)$. So, $((x:A)B)\sigma = (x:A\sigma)B\sigma$. By induction hypothesis, $A\sigma \in [\![\star]\!]_\sigma = \mathcal{D}$. Let now $a \in \mathcal{I}(A\sigma)$ and $\sigma' = [x \mapsto a, \sigma]$. Note that $\mathcal{I}(A\sigma) = [\![A]\!]_\sigma$. So, $\sigma' \models \Gamma, x:A$ and, by induction hypothesis, $B\sigma' \in [\![s]\!]_\sigma = \mathcal{D}$. Since $x \notin \mathrm{dom}(\sigma) \cup \mathrm{FV}(\sigma)$, we have $B\sigma' = (B\sigma)^a_x$. Therefore, $((x:A)B)\sigma \in [\![s]\!]_\sigma$.

(abs) Is $(\lambda x:A.b)\sigma$ in $[\![(x:A)B]\!]_\sigma$? Wlog we can assume that $x \notin \mathrm{dom}(\sigma) \cup \mathrm{FV}(\sigma)$. So, $(\lambda x:A.b)\sigma = \lambda x:A\sigma.b\sigma$. By Lemma 1d, $[\![(x:A)B]\!]_\sigma = \Pi a \in [\![A]\!]_\sigma.\ [\![B]\!]_{[x\mapsto a,\sigma]}$. By Lemma 1c, $[\![B]\!]_{[x\mapsto a,\sigma]}$ is an $[\![A]\!]_\sigma$-indexed family of computability predicates such that $[\![B]\!]_{[x\mapsto a',\sigma]} = [\![B]\!]_{[x\mapsto a,\sigma]}$ whenever $a \to a'$. Hence, by Lemma 1f, $\lambda x:A\sigma.b\sigma \in [\![(x:A)B]\!]_\sigma$ if $A\sigma \in \mathrm{SN}$ and, for all $a \in [\![A]\!]_\sigma$, $(b\sigma)^a_x \in [\![B]\!]_{\sigma'}$ where $\sigma' = [x \mapsto a, \sigma]$. By induction hypothesis, $((x:A)B)\sigma \in [\![s]\!]_\sigma = \mathcal{D}$. Since $x \notin \mathrm{dom}(\sigma) \cup \mathrm{FV}(\sigma)$, $((x:A)B)\sigma = (x:A\sigma)B\sigma$ and $(b\sigma)^a_x = b\sigma'$. Since $\mathcal{D} \subseteq \mathrm{SN}$, we have $A\sigma \in \mathrm{SN}$. Moreover, since $\sigma' \models \Gamma, x:A$, we have $b\sigma' \in [\![B]\!]_{\sigma'}$ by induction hypothesis.

(app') Is $(ta)\sigma = (t\sigma)(a\sigma)$ in $[\![B^a_x]\!]_\sigma$? By induction hypothesis, $t\sigma \in [\![(x:A)B]\!]_\sigma$, $a\sigma \in [\![A]\!]_\sigma$ and $((x:A)B)\sigma \in [\![s]\!] = \mathcal{D}$. By Lemma 1d, $[\![(x:A)B]\!]_\sigma = \Pi\alpha \in [\![A]\!]_\sigma.\ [\![B]\!]_{[x\mapsto\alpha,\sigma]}$. Hence, $(t\sigma)(a\sigma) \in [\![B]\!]_{\sigma'}$ where $\sigma' = [x \mapsto a\sigma, \sigma]$. Wlog we can assume $x \notin \mathrm{dom}(\sigma) \cup \mathrm{FV}(\sigma)$. So, $\sigma' = [x \mapsto a]\sigma$. Hence, by Lemma 1e, $[\![B]\!]_{\sigma'} = [\![B^a_x]\!]_\sigma$.

(conv') By induction hypothesis, $a\sigma \in [\![A]\!]_\sigma$, $A\sigma \in [\![s]\!]_\sigma = \mathcal{D}$ and $B\sigma \in [\![s]\!]_\sigma = \mathcal{D}$. By Lemma 1b, $[\![A]\!]_\sigma = [\![B]\!]_\sigma$. So, $a\sigma \in [\![B]\!]_\sigma$.

(fun) By induction hypothesis, $\Theta_f \in [\![s_f]\!]_\sigma = \mathcal{D}$. Therefore, $f \in [\![\Theta_f]\!]_\sigma = [\![\Theta_f]\!]$ since Θ is adequate. □

By taking the identity for σ, we get that $\to$ terminates on terms typable in $\lambda\Pi/\mathcal{R}$ if $\to$ is locally confluent and preserves typing, and Θ is adequate.

4 Dependency pairs theorem

Now, we prove that the computability of function symbols (*i.e.* the adequacy of Θ) can be reduced to the absence of infinite sequences of dependency pairs, as shown by Arts and Giesl for first-order rewriting [2].

Definition 3 (Dependency pairs) *Let $f\boldsymbol{l} > g\boldsymbol{m}$ iff there is a rule $f\boldsymbol{l} \to r \in \mathcal{R}$, g is defined and $g\boldsymbol{m}$ is a subterm of r such that $\boldsymbol{m}$ are all the arguments to which g is applied. The relation $>$ is the set of dependency pairs.*

Let $\tilde{>} = \to^*_{\mathrm{arg}} >_s$ be the relation on the set $\mathbb{C}$ (see Def. 1), where $f\boldsymbol{t} \to_{\mathrm{arg}} f\boldsymbol{u}$ iff $\boldsymbol{t} \to_{\mathrm{prod}} \boldsymbol{u}$ (reduction in one argument), and $>_s$ is the closure by substitution and left-application of $>$: $ft_1 \dots t_p \,\tilde{>}\, gu_1 \dots u_q$ iff there are a dependency pair $fl_1 \dots l_i > gm_1 \dots m_j$ with $i \le p$ and $j \le q$ and a substitution σ such that, for all $k \le i$, $t_k \to^* l_k\sigma$ and, for all $k \le j$, $m_k\sigma = u_k$.

In our setting, we have to close $>_s$ by left-application because function symbols are curryfied. When a function symbol f is not fully applied wrt $\alpha(\Theta_f)$, the missing arguments must be considered as potentially being anything. Indeed, the following rewrite system:

```
        app x y → x y              f x y → app (f x) y
```

whose dependency pairs are $fxy > app(fx)y$ and $fxy > fx$, does not terminate, but there is no way to construct an infinite sequence of dependency pairs without adding an argument to the right-hand side of the second dependency pair.

Example 2 *The rules of Example 1 have the following dependency pairs:*

```
A:                El (arrow a b) > El a
B:                El (arrow a b) > El b
C:                     (s p) + q > p + q
D:    app a _ (cons _ x p l) q m > p + q
E:    app a _ (cons _ x p l) q m > app a p l q m
F:len_fil a f _ (cons _ x p l)   > len_fil_aux (f x) a f p l
G:len_fil a f _ (app _ p l q m) >
                        (len_fil a f p l) + (len_fil a f q m)
H:len_fil a f _ (app _ p l q m) > len_fil a f p l
I:len_fil a f _ (app _ p l q m) > len_fil a f q m
J:    len_fil_aux true  a f p l > len_fil a f p l
K:    len_fil_aux false a f p l > len_fil a f p l
L:    fil a f _ (cons _ x p l)   > fil_aux (f x) a f x p l
M:    fil a f _ (app _ p l q m) >
                        app a (len_fil a f p l) (fil a f p l)
                              (len_fil a f q m) (fil a f q m)
N:    fil a f _ (app _ p l q m) > len_fil a f p l
O:    fil a f _ (app _ p l q m) > fil a f p l
P:    fil a f _ (app _ p l q m) > len_fil a f q m
Q:    fil a f _ (app _ p l q m) > fil a f q m
R:      fil_aux true  a f x p l > len_fil a f p l
S:      fil_aux true  a f x p l > fil a f p l
T:      fil_aux false a f x p l > fil a f p l
```

In [2], a sequence of dependency pairs interleaved with $\rightarrow_{\mathrm{arg}}$ steps is called a chain. Arts and Giesl proved that, in a first-order term algebra, $\rightarrow_{\mathcal{R}}$ terminates if and only if there are no infinite chains, that is, if and only if $\tilde{>}$ terminates. Now, one can easily check that, in a first-order term algebra, $\tilde{>}$ terminates if and only if, for all f and $\boldsymbol{t}$, $f\boldsymbol{t}$ terminates wrt $\tilde{>}$ whenever $\boldsymbol{t}$ terminates wrt $\rightarrow$. In our framework, this last condition is similar to saying that Θ is adequate.

We now introduce the class of systems to which we will extend Arts and Giesl's theorem.

Definition 4 (Well-structured system) *Let $\succeq$ be the smallest quasi-order on $\mathbb{F}$ such that $f \succeq g$ if g occurs in Θ_f or if there is a rule $f\boldsymbol{l} \rightarrow r \in \mathcal{R}$ with g (defined or undefined) occurring in r. Then, let $\succ = \succeq \setminus \preceq$ be the strict part of $\succeq$. A rewrite system $\mathcal{R}$ is well-structured system if:*

(a) $\succ$ is well-founded;
(b) for every rule $f\boldsymbol{l} \rightarrow r$, $|\boldsymbol{l}| \leq \alpha(\Theta_f)$;
(c) for every dependency pair $f\boldsymbol{l} > g\boldsymbol{m}$, $|\boldsymbol{m}| \leq \alpha(\Theta_g)$;

Fig. 2. Restricted type systems $\vdash_{f\boldsymbol{l}}$ and $\vdash_{\prec f}$

$$\text{(ax)}\quad \frac{}{\vdash_{f\boldsymbol{l}} \star : \square} \qquad \text{(weak)}\quad \frac{\Gamma \vdash_{\prec f} A : s \quad \Gamma \vdash_{f\boldsymbol{l}} b : B \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : A \vdash_{f\boldsymbol{l}} b : B}$$

$$\text{(var)}\quad \frac{\Gamma \vdash_{\prec f} A : s \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : A \vdash_{f\boldsymbol{l}} x : A} \qquad \text{(prod)}\quad \frac{\Gamma \vdash_{f\boldsymbol{l}} A : \star \quad \Gamma, x : A \vdash_{f\boldsymbol{l}} B : s}{\Gamma \vdash_{f\boldsymbol{l}} (x : A)B : s}$$

$$\text{(abs)}\quad \frac{\Gamma, x : A \vdash_{f\boldsymbol{l}} b : B \quad \Gamma \vdash_{\prec f} (x : A)B : s}{\Gamma \vdash_{f\boldsymbol{l}} \lambda x : A.b : (x : A)B}$$

$$\text{(conv')}\quad \frac{\Gamma \vdash_{f\boldsymbol{l}} a : A \quad A \downarrow B \quad \Gamma \vdash_{\prec f} B : s \quad \Gamma \vdash_{\prec f} A : s}{\Gamma \vdash_{f\boldsymbol{l}} a : B}$$

$$\text{(app')}\quad \frac{\Gamma \vdash_{f\boldsymbol{l}} t : (x : A)B \quad \Gamma \vdash_{f\boldsymbol{l}} a : A \quad \Gamma \vdash_{\prec f} (x : A)B : s}{\Gamma \vdash_{f\boldsymbol{l}} ta : B_x^a}$$

$$\text{(dp)}\quad \frac{\vdash_{\prec f} \Theta_g : s_g \quad \Gamma \vdash_{f\boldsymbol{l}} \gamma : \Sigma}{\Gamma \vdash_{f\boldsymbol{l}} g\boldsymbol{y}\gamma : V\gamma}\, (\Theta_g = (\Sigma)V, \Sigma = \boldsymbol{y} : \boldsymbol{U}, g\boldsymbol{y}\gamma < f\boldsymbol{l})$$

$$\text{(const)}\quad \frac{\vdash_{\prec f} \Theta_g : s_g}{\vdash_{f\boldsymbol{l}} g : \Theta_g} \qquad (g \text{ undefined})$$

and $\vdash_{\prec f}$ is defined by the same rules as $\vdash$, except (fun) replaced by:

$$(\mathrm{fun}_{\prec f})\quad \frac{\vdash_{\prec f} \Theta_g : s_g \quad g \prec f}{\vdash_{\prec f} g : \Theta_g}$$

(d) for every $f\boldsymbol{l} \to r$, *there is an environment* $\Delta_{f\boldsymbol{l}\to r}$ *such that, if* $\Theta_f = (\boldsymbol{x} : \boldsymbol{T})U$ *and* $\pi = [\boldsymbol{x} \mapsto \boldsymbol{l}]$, *then* $\Delta_{f\boldsymbol{l}\to r} \vdash_{f\boldsymbol{l}} r : U\pi$, *where* $\vdash_{f\boldsymbol{l}}$ *is defined in Fig. 2.*

One can check that the system of Example 1 is well-structured (the proof is given in Annex B).

Note that condition (a) is always satisfied when $\mathbb{F}$ is finite. The condition (b) ensures that a term of the form $f\boldsymbol{t}$ is neutral whenever $|\boldsymbol{t}| = \alpha(\Theta_f)$. Condition (c) ensures that $>$ is included in $\tilde{>}$.

Condition (d) not only states a typability property but also a stratification property. The relations $\vdash_{f\boldsymbol{l}}$ and $\vdash_{\prec f}$ are restrictions of the typing relation. Terms typable in $\vdash_{\prec f}$ contain only symbols strictly smaller than f. So, in a well-structured system, one does not need f to type Θ_f. The relation $\vdash_{f\boldsymbol{l}}$ corresponds to the notion of computability closure in [8], with the ordering on function calls replaced by the dependency pair relation. Note that the environment $\Delta_{f\boldsymbol{l}\to r}$ can be infered when one restricts left hand-sides to be patterns *à la* Miller [38].

Finally, we need matching to be compatible with computability, that is, we need that, if t is computable and $t = l\sigma$, then σ is computable, a condition called accessibility in [8]:

Definition 5 (Accessible system) *A well-structured system* $\mathcal{R}$ *is accessible if, for all rules* $f\boldsymbol{l} \to r \in \mathcal{R}$ *with* $\Theta_f = (\Gamma)U$ *and* $\Gamma = \boldsymbol{x} : \boldsymbol{T}$, *and substitution* σ, $\sigma \models \Delta_{f\boldsymbol{l}\to r}$ *whenever* $\vdash \Theta_f : s_f$, $\Theta_f \in [\![s_f]\!]$ *and* $\pi\sigma \models \Gamma$, *where* $\pi = [\boldsymbol{x} \mapsto \boldsymbol{l}]$.

This property is not always satisfied because subterm does preserve computability. Indeed, if C is an undefined type constant, then $[\![C]\!] = \mathrm{SN}$. However, $[\![C \Rightarrow C]\!] \neq \mathrm{SN}$ since $\omega = \lambda x : C.xx \in \mathrm{SN}$ and $\omega\omega \notin \mathrm{SN}$. Hence, if $\Theta_f = (x : T)C$ and $ft \in [\![C]\!]$, then we may not have $t \in [\![T]\!]$ if T is a product.

In the next sub-section, we give a simple condition for accessibility to hold.

We can now state the main lemma:

Lemma 3 *Θ is adequate if $\mathcal{R}$ is accessible and well-structured and $\tilde{>}$ terminates.*

Proof. Since $\mathcal{R}$ is well-structured, $\succ$ is well-founded by condition (a). We prove that, for all $f \in \mathbb{F}$, $f \in [\![\Theta_f]\!]$, by induction on $\succ$. So, let $f \in \mathbb{F}$ with $\Theta_f = (\Gamma_f)U$ and $\Gamma_f = x_1 : T_1, \ldots, x_n : T_n$. By induction hypothesis, we have that, for all $g \prec f$, $g \in [\![\Theta_g]\!]$.

Since $\rightarrow_{\mathrm{arg}}$ and $\tilde{>}$ terminate on $\mathbb{C}$ and $\rightarrow_{\mathrm{arg}} \tilde{>} \subseteq \tilde{>}$, we have that $\rightarrow_{\mathrm{arg}} \cup \tilde{>}$ terminates. We now prove that, for all $f\boldsymbol{t} \in \mathbb{C}$, we have $f\boldsymbol{t} \in [\![U]\!]_\theta$ where $\theta = [\boldsymbol{x} \mapsto \boldsymbol{t}]$, by induction on $\rightarrow_{\mathrm{arg}} \cup \tilde{>}$. By condition (b), $f\boldsymbol{t}$ is neutral. Hence, by definition of computability, it suffices to prove that, for all $u \in \rightarrow(f\boldsymbol{t})$, $u \in [\![U]\!]_\theta$. There are 2 cases:

- $u = f\boldsymbol{v}$ with $\boldsymbol{t} \rightarrow_{\mathrm{prod}} \boldsymbol{v}$. Then, we can conclude by induction hypothesis.
- There are $fl_1 \ldots l_k \rightarrow r \in \mathcal{R}$ and σ such that $u = (r\sigma)t_{k+1} \ldots t_n$ and, for all $i \in \{1, \ldots, k\}$, $t_i = l_i\sigma$. Since $f\boldsymbol{t} \in \mathbb{C}$, we have $\pi\sigma \models \Gamma_f$. Since $\mathcal{R}$ is accessible, we get that $\sigma \models \Delta_{f\boldsymbol{l} \rightarrow r}$. By condition (d), we have $\Delta_{f\boldsymbol{l} \rightarrow r} \vdash_{f\boldsymbol{l}} r : V\pi$ where $V = (x_{k+1} : T_{k+1}) \ldots (x_n : T_n)U$.
  Now, we prove that, for all Γ, t and T, if $\Gamma \vdash_{f\boldsymbol{l}} t : T$ ($\Gamma \vdash_{\prec f} t : T$ resp.) and $\sigma \models \Gamma$, then $t\sigma \in [\![T]\!]_\sigma$, by induction on the structure of the derivation of $\Gamma \vdash_{f\boldsymbol{l}} t : T$ ($\Gamma \vdash_{\prec f} t : T$ resp.), as in the proof of Lemma 2 except for (fun) replaced by (fun$_{\prec f}$) in one case, and (const) and (dp) in the other case.
  - (fun$_{\prec f}$) We have $g \in [\![\Theta_g]\!]$ by the induction hypothesis on g.
  - (const) Since g is undefined, it is neutral and normal. Therefore, it belongs to every computability predicate and in particular to $[\![\Theta_g]\!]_\sigma$.
  - (dp) By induction hypothesis, $y_i\gamma\sigma \in [\![U_i\gamma]\!]_\sigma$. By Lemma 1e, $[\![U_i\gamma]\!]_\sigma = [\![U_i]\!]_{\gamma\sigma}$. So, $\gamma\sigma \models \Sigma$ and $g\boldsymbol{y}\gamma\sigma \in \mathbb{C}$. Now, by condition (c), $g\boldsymbol{y}\gamma\sigma \tilde{<} f\boldsymbol{l}\sigma$ since $g\boldsymbol{y}\gamma < f\boldsymbol{l}$. Therefore, by induction hypothesis, $g\boldsymbol{y}\gamma\sigma \in [\![V\gamma]\!]_\sigma$.

  So, $r\sigma \in [\![V\pi]\!]_\sigma$ and, by Lemma 1d, $u \in [\![U]\!]_{[x_n \mapsto t_n, .., x_{k+1} \mapsto t_{k+1}, \pi\sigma]} = [\![U]\!]_\theta$. □

Any well-founded quasi-order $\succeq$ on $\mathbb{F}$ such that $f\boldsymbol{l} > g\boldsymbol{m}$ implies $f \succeq g$ could have been used. The quasi-order of Def. 4, defined syntactically, avoids the user the burden to provide one and is sufficient in every practical case met by the authors. However it is possible to construct ad-hoc systems which require a quasi-order richer than the one presented here.

4.1 Checking accessibility

We provide here a simple criterion for checking accessibility.

We have seen that, if C is an undefined constant, then $[\![C]\!] = \mathrm{SN}$ and, if $\Theta_f = (x : T)C$ and $ft \in [\![C]\!]$, then we do not have $t \in [\![T]\!]$ if T is a product. So,

to ensure accessibility, a simple condition is to require higher-order variables to be direct subterms of the left-hand side, a condition called plain-function-passing (PFP) in [33]. It is however possible to capture more functional subterms by using a more complex interpretation of types [8].

Definition 6 (PFP systems) *A well-structured system $\mathcal{R}$ is plain-function-passing (PFP) if, for all rules $f\boldsymbol{l} \to r \in \mathcal{R}$ with $\Theta_f = (\Gamma)U$ and $\Gamma = \boldsymbol{x} : \boldsymbol{T}$, $\boldsymbol{l} \notin \mathbb{K} \cup \{\Box\}$ and, for all $(x : T) \in \Delta_{f\boldsymbol{l}\to r}$, there is i such that $x = l_i$ and $T = T_i\pi$, where $\pi = [\boldsymbol{x} \mapsto \boldsymbol{l}]$, or else $x \in \mathrm{FV}(l_i)$ and $T = D\boldsymbol{t}$ with D undefined and $|\boldsymbol{t}| = \alpha(D)$.*

Lemma 4 *PFP systems are accessible.*

Proof. Let $f\boldsymbol{l} \to r$ be a PFP rule with $\Theta_f = (\Gamma)U$, $\Gamma = \boldsymbol{x} : \boldsymbol{T}$, $\pi = [\boldsymbol{x} \mapsto \boldsymbol{l}]$. Following Definition 5, assume that $\vdash \Theta_f : s_f$, $\Theta_f \in \mathcal{D}$ and $\pi\sigma \models \Gamma$. We have to prove that, for all $(x : T) \in \Delta_{f\boldsymbol{l}\to r}$, $x\sigma \in [\![T]\!]_\sigma$.

- If $x = l_i$ and $T = T_i\pi$. Then, $x\sigma = l_i\sigma \in [\![T_i]\!]_{\pi\sigma}$. Since $\vdash \Theta_f : s_f$, $T_i \notin \mathbb{K}\cup\{\Box\}$. Since $\Theta_f \in \mathcal{D}$ and $\pi\sigma \models \Gamma$, we have $T_i\pi\sigma \in \mathcal{D}$. So, $[\![T_i]\!]_{\pi\sigma} = \mathcal{I}(T_i\pi\sigma)$. Since $T_i \notin \mathbb{K} \cup \{\Box\}$ and $\boldsymbol{l} \notin \mathbb{K} \cup \{\Box\}$, $T_i\pi \notin \mathbb{K} \cup \{\Box\}$. Since $T_i\pi\sigma \in \mathcal{D}$, $[\![T_i\pi]\!]_\sigma = \mathcal{I}(T_i\pi\sigma)$. Thus, $x\sigma \in [\![T]\!]_\sigma$.
- If $x \in \mathrm{FV}(l_i)$ and T is of the form $C\boldsymbol{t}$ with $|\boldsymbol{t}| = \alpha(C)$. Then, $[\![T]\!]_\sigma = \mathrm{SN}$ and $x\sigma \in \mathrm{SN}$ since $l_i\sigma \in [\![T_i]\!]_\sigma \subseteq \mathrm{SN}$. ◻

Example 3 *The system of Example 1 is PFP. On the other hand, the following example is not PFP since* `f`:$\mathbb{N} \Rightarrow \mathbb{O}$ *is not argument of* `ordrec`*. It is however accessible if one takes an interpretation of* $\mathbb{O}$ *more complex than* SN *[8].*

```
symbol 𝕆:TYPE
  symbol zero:𝕆  symbol suc:𝕆⇒𝕆  symbol lim:(ℕ⇒𝕆)⇒𝕆

symbol ordrec:A⇒(𝕆⇒A⇒A)⇒((ℕ⇒𝕆)⇒(ℕ⇒A)⇒A)⇒𝕆⇒A
  rule ordrec u v w zero    → u
  rule ordrec u v w (suc x) → v x (ordrec u v w x)
  rule ordrec u v w (lim f) → w f (λn,ordrec u v w (f n))
```

5 Size-change termination

To complete our termination proof, we need to prove that there are no infinite chains of dependency pairs. In first-order rewriting, many techniques have been developed for that purpose. To cite just a few, see for instance [22,18,50]. We think that many of them can probably be extended to $\lambda\Pi/\mathcal{R}$, either because the structure of terms in which they are expressed can be abstracted away, or because they can be extended to deal also with variable applications, λ-abstractions and β-reductions. See for instance [46,30].

As an example, following Wahlstedt [54], we are going to use Lee, Jones and Ben-Amram's size-change termination criterion (SCT) [34]. It consists in

following arguments along function calls and check that, in every potential loop, one of them decreases. First introduced for first-order programming languages, it has then been extended to untyped λ-calculus [27], a subset of OCaml [45], Martin-Löf's type theory [54] and System F [35].

We first recall Hyvernat and Raffalli's matrix-based presentation of SCT [26]:

Definition 7 (Size-change termination) *Let $\rhd$ be the smallest transitive relation such that $ft_1 \dots t_n \rhd t_i$ when $f \in \mathbb{F}$. The call graph $\mathcal{G}(\mathcal{R})$ associated to $\mathcal{R}$ is the directed labeled graph on the defined symbols of $\mathbb{F}$ such that there is an edge between f and g iff there is a dependency pair $fl_1 \dots l_p > gm_1 \dots m_q$. This edge is labeled with the matrix $(a_{i,j})_{i \leq \alpha(\Theta_f), j \leq \alpha(\Theta_g)}$ where:*

- *if $l_i \rhd m_j$, then $a_{i,j} = -1$;*
- *if $l_i = m_j$, then $a_{i,j} = 0$;*
- *otherwise $a_{i,j} = \infty$ (in particular if $i > p$ or $j > q$).*

$\mathcal{R}$ is size-change terminating (SCT) if, in the transitive closure of $\mathcal{G}(\mathcal{R})$ (using the min-plus semi-ring to multiply the matrices labeling the edges), all idempotent matrices labeling a loop have some -1 on the diagonal.

Since in our definition of $\tilde{>}$ we allow the addition of arbitrary arguments at the right, we cannot compare those arguments with any other. That is the reason why we add lines or columns of ∞ symbols in matrices associated to dependency pairs containing partially applied symbols.

Lemma 5 *$\tilde{>}$ terminates if $\mathbb{F}$ is finite and $\mathcal{R}$ is SCT.*

Proof. Suppose that there is an infinite sequence $\chi = f_1 \boldsymbol{t}_1 \tilde{>} f_2 \boldsymbol{t}_2 \tilde{>} \dots$ Then, there is an infinite path in the call graph going through nodes labeled by $f_1, f_2, \dots$ Since $\mathbb{F}$ is finite, there is a symbol g occurring infinitely often in this path. So, there is an infinite sequence $g\boldsymbol{u}_1, g\boldsymbol{u}_2, \dots$ extracted from χ. Hence, for every $i, j \in \mathbb{N}^*$, there is a matrix in the transitive closure of the graph which labels the loops of g corresponding to the relation between $\boldsymbol{u}_i$ and $\boldsymbol{u}_{i+j}$. By Ramsey's theorem [42], there is an infinite sequence (ϕ_i) and a matrix M such that M corresponds to all the transitions $g\boldsymbol{u}_{\phi_i}, g\boldsymbol{u}_{\phi_j}$ with $i \neq j$. M is idempotent, indeed $g\boldsymbol{u}_{\phi_i}, g\boldsymbol{u}_{\phi_{i+2}}$ is labeled by M^2 by definition of the transitive closure and by M due to Ramsey's theorem, so $M = M^2$. Since, by hypothesis, $\mathcal{R}$ satisfies SCT, there is j such that $M_{j,j}$ is -1. So, for all i, $u^{(j)}_{\phi_i} (\to^* \rhd)^+ u^{(j)}_{\phi_{i+1}}$. Since $(\rhd \to) \subseteq (\to \rhd)$ and $\to_{\text{arg}}$ is well-founded on $\mathbb{C}$, the existence of an infinite sequence contradicts the fact that $\rhd$ is well-founded. $\square$

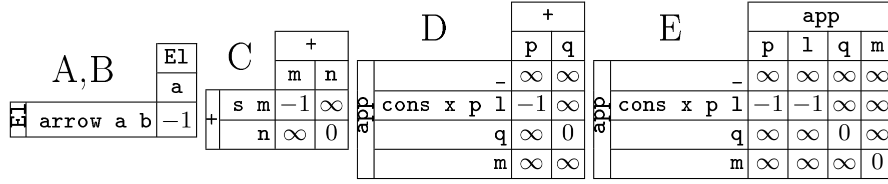
By combining all the previous results, we get:

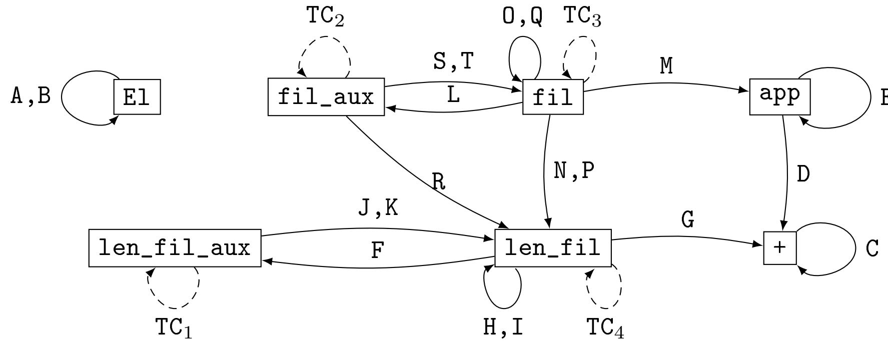
Theorem 1 *The relation $\to = \to_\beta \cup \to_\mathcal{R}$ terminates on terms typable in $\lambda\Pi/\mathcal{R}$ if $\to$ is locally confluent and preserves typing, $\mathbb{F}$ is finite and $\mathcal{R}$ is well-structured, plain-function passing and size-change terminating.*

Example 4 *To illustrate the interest of our criterion, we show that Example 1 satisfies the above conditions.*

One can easily check that every square matrix of the call graph below has some -1 on its diagonal. The letter associated to each matrix corresponds to the dependency pair in Example 2, except for `TC`*'s which comes from the computation of the transitive closure. The argument* `a` *is omitted everywhere. Dotted edges are the loops of the transitive closure.*

A representative subset of the matrices is displayed here, the full list of matrices can be found in Annex B





where $\mathtt{F} = \begin{pmatrix} \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & -1 & -1 \end{pmatrix}$, $\mathtt{J} = \mathtt{K} = \begin{pmatrix} \infty & \infty & \infty \\ 0 & \infty & \infty \\ \infty & 0 & \infty \\ \infty & \infty & 0 \end{pmatrix}$, $\mathtt{TC}_1 = \mathtt{J} \times \mathtt{F} = \begin{pmatrix} \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & -1 & -1 \end{pmatrix} \ldots$

6 Conclusion and future work

We proved a general modularity result extending Arts and Giesl's theorem that a rewrite relation terminates if there are no infinite sequences of dependency pairs [2] from first-order rewriting to dependently-typed higher-order rewriting. Then, following [54], we showed how to use Lee, Jones and Ben-Amram's size-change termination criterion to prove the absence of such infinite sequences [34].

This extends Wahlstedt's work [54] from weak to strong normalization, and from orthogonal to locally confluent rewrite systems. This extends the first author's work [8] from orthogonal to locally confluent systems, and from systems having a decreasing argument in each recursive call to systems with non-increasing arguments in recursive calls. Finally, this also extends previous works on static dependency pairs [46,29] from simply-typed λ-calculus to dependent types modulo rewriting.

In [43], Roux uses also dependency pairs for the termination of simply-typed higher-order rewrite systems, as well as a restricted form of dependent types where a type constant C is annotated by a pattern l representing the set of terms matching l. This extends to patterns the notion of indexed or sized types [24]. Then, for proving the absence of infinite chains, he uses simple projections [23], which can be seen as a particular case of SCT where strictly decreasing arguments are fixed (SCT can handle permutations in arguments).

We showed how to prove termination from local confluence. However, to decide local confluence, one often uses termination. Fortunately, there are a few confluence criteria not based on termination. The most famous one is (weak) orthogonality, that is, when the system is left-linear and has no critical pairs (or only trivial ones) [40], as it is the case in functional programming languages. A more general one is when critical pairs are "development-closed" [41].

We implemented our criterion in a tool called SizeChangeTool [16]. As far as we know, there are no other implementation of a termination checker for $\lambda\Pi/\mathcal{R}$. However, if we restrict ourselves to simple types, we can try to compare our tool with higher-order termination checkers, like Wanda [32] and SOL [20], on the benchmark of the the category "higher-order rewriting union beta" of the Termination Problems Data Base [53]. Even if SizeChangeTool only implements the technique presented in this paper, there are problems solved by it and not by one of the other provers [5], demonstrating that these provers would benefit from implementing this technique.

This work can be extended in several directions.

First, our termination criterion is limited to PFP rules, that is, to rules where higher-order variables are direct subterms of the left-hand side. To have higher-order variables in deeper subterms, we need to define a more complex interpretation of types, following [8].

Second, to handle function calls with deep higher-order variables, we also need to use an ordering more complex than the subterm ordering when computing the matrices labeling the SCT call graph (the ordering needed for Example 3 is called the "structural ordering" in [13,9]). Relations other than subterm has already been considered but in a first-order setting only [51].

Another interesting extension would be to handle function calls with locally size-increasing arguments like in the following example:

```
rule f x → g (s x)          rule g (s (s x)) → f x
```

where the number of `s`'s strictly decreases between two calls to `f` although the first rule makes the number of `s`'s increase. Hyvernat enriched the original SCT to handle such a situation [25].

[5] During the Termination Competition 2018 [49], Kop13/kop12thesis_ex7.23 was proved terminating by SizeChangeTool but not by Sol because there are two mutually recursive functions with non-strict decrease in one of the calls, and Hamana_Kikuchi_18/h17 was proved terminating by SizeChangeTool and not by Wanda.

References

1. Abian, S., Brown, A.B.: A theorem on partially ordered sets with applications to fixed point theorems. Canadian Journal of Mathematics **13**, 78–83 (1961)
2. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theoretical Computer Science **236**, 133–178 (2000)
3. Assaf, A.: A framework for defining computational higher-order logics. Ph.D. thesis, École Polytechnique, France (2015)
4. Barbanera, F., Fernández, M., Geuvers, H.: Modularity of strong normalization in the algebraic-λ-cube. Journal of Functional Programming **7**(6), 613–660 (1997)
5. Barendregt, H.: Lambda calculi with types. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.) Handbook of logic in computer science. Volume 2. Background: computational structures, pp. 117–309. Oxford University Press (1992)
6. Blanqui, F.: Type theory and rewriting (2001), English translation of [7]
7. Blanqui, F.: Théorie des types et récriture. Ph.D. thesis, Université Paris-Sud, France (2001), English translation in [6]
8. Blanqui, F.: Definitions by rewriting in the calculus of constructions. Mathematical Structures in Computer Science **15**(1), 37–92 (2005)
9. Blanqui, F.: Termination of rewrite relations on λ-terms based on Girard's notion of reducibility. Theoretical Computer Science **611**, 50–86 (2016)
10. Bove, A., Capretta, V.: Modelling general recursion in type theory. Mathematical Structures in Computer Science **15**(4), 671–708 (2005)
11. de Bruijn, N.G.: The mathematical language AUTOMATH, its usage, and some of its extensions. In: Proceedings of the 1968 Symposium on Automatic Demonstration. Lecture Notes in Mathematics, vol. 125 (1970)
12. Constable, R.: Partial functions in constructive formal theories. In: 6th Biannual GI Symposium on Theoretical Computer Science. Lecture Notes in Computer Science, vol. 145 (1983)
13. Coquand, T.: Pattern matching with dependent types. In: Proceedings of the International Workshop on Types for Proofs and Programs, 1992
14. Cousineau, D., Dowek, G.: 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)
15. Dedukti. `https://deducteam.github.io/` (2018)
16. Genestier, G.: SizeChangeTool. `https://github.com/Deducteam/SizeChangeTool` (2018)
17. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The dependency pair framework: combining techniques for automated termination proofs. In: Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Lecture Notes in Computer Science 3452 (2004)
18. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. Journal of Automated Reasoning **37**(3), 155–203 (2006)
19. Girard, J.Y., Lafont, Y., Taylor, P.: Proofs and types. Cambridge University Press (1988)
20. Hamana, M., et al.: SOL. `http://project-coco.uibk.ac.at/2018/papers/sol.pdf` (2018)
21. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM **40**(1), 143–184 (1993)
22. Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. Information and Computation **199**(1-2), 172–199 (2005)

23. Hirokawa, N., Middeldorp, A.: Tyrolean Termination Tool: techniques and features. Information and Computation **205**(4), 474–511 (2007)
24. Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: Proceedings of the 23th ACM Symposium on Principles of Programming Languages(1996)
25. Hyvernat, P.: The size-change termination principle for constructor based languages. Logical Methods in Computer Science **10**(1), 1–30 (2014)
26. Hyvernat, P., Raffalli, C.: Improvements on the "size change termination principle" in a functional language. In: 11th International Workshop on Termination(2010)
27. Jones, N.D., Bohr, N.: Termination analysis of the untyped lambda-calculus. In: Proceedings of the 15th International Conference on Rewriting Techniques and Applications, Lecture Notes in Computer Science 3091 (2004)
28. Klop, J.W., van Oostrom, V., van Raamsdonk, F.: Combinatory reduction systems: introduction and survey. Theoretical Computer Science **121**, 279–308 (1993)
29. Kop, C.: Higher order dependency pairs for algebraic functional systems. In: Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, Leibniz International Proceedings in Informatics 10 (2011)
30. Kop, C.: Higher order termination. Ph.D. thesis, VU University Amsterdam (2012)
31. Kop, C., Raamsdonk, F.V.: Dynamic dependency pairs for algebraic functional systems. Logical Methods in Computer Science **8**(2), 1–51 (2012)
32. Kop, C., et al.: Wanda, `http://wandahot.sourceforge.net/`
33. Kusakari, K., Sakai, M.: Enhancing dependency pair method using strong computability in simply-typed term rewriting systems. Applicable Algebra in Engineering Communication and Computing **18**(5), 407–431 (2007)
34. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: Proceedings of the 28th ACM Symposium on Principles of Programming Languages(2001)
35. Lepigre, R., Raffalli, C.: Practical subtyping for System F with sized (co-)induction (2017)
36. Martin-Löf, P.: Intuitionistic type theory. Bibliopolis, Napoli, Italy (1984)
37. Mayr, R., Nipkow, T.: Higher-order rewrite systems and their confluence. Theoretical Computer Science **192**(2), 3–29 (1998)
38. Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. Journal of Logic and Computation **1**(4), 497–536 (1991)
39. Newman, M.: On theories with a combinatorial definition of "equivalence". Annals of Mathematics **43**(2), 223–243 (1942)
40. van Oostrom, V.: Confluence for abstract and higher-order rewriting. Ph.D. thesis, Vrije Universiteit Amsterdam, NL (1994)
41. van Oostrom, V.: Developing developments. Theoretical Computer Science **175**(1), 159–181 (1997)
42. Ramsey, F.P.: On a problem of formal logic. Proceedings of the London Mathematical Society **s2-30**(1), 264–286 (1930)
43. Roux, C.: Refinement Types as Higher-Order Dependency Pairs. In: Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, Leibniz International Proceedings in Informatics 10 (2011)
44. Saillard, R.: Type checking in the Lambda-Pi-calculus modulo: theory and practice. Ph.D. thesis, Mines ParisTech, France (2015)
45. Sereni, D., Jones, N.D.: Termination analysis of higher-order functional programs. In: Proceedings of the 3rd Asian Symposium on Programming Languages and Systems, Lecture Notes in Computer Science 3780 (2005)

46. Suzuki, S., Kusakari, K., Blanqui, F.: Argument filterings and usable rules in higher-order rewrite systems. IPSJ Transactions on Programming **4**(2), 1–12 (2011)
47. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics **5**, 285–309 (1955)
48. TeReSe: Term rewriting systems, Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press (2003)
49. `http://termination-portal.org/wiki/Termination_Competition` (2018)
50. Thiemann, R.: The DP framework for proving termination of term rewriting. Ph.D. thesis, RWTH Aachen University, Germany (2007), technical Report AIB-2007-17
51. Thiemann, R., Giesl, J.: The size-change principle and dependency pairs for termination of term rewriting. Applicable Algebra in Engineering Communication and Computing **16**(4), 229–270 (2005)
52. Toyama, Y.: Counterexamples to termination for the direct sum of term rewriting systems. Information Processing Letters **25**(3), 141–143 (1987)
53. Termination problem data base (tpdb) (2018), `http://cl2-informatik.uibk.ac.at/mercurial.cgi/TPDB`
54. Wahlstedt, D.: Dependent type theory with first-order parameterized data types and well-founded recursion. Ph.D. thesis, Chalmers University of Technology, Sweden (2007)
55. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: Proceedings of the 30th ACM Symposium on Principles of Programming Languages(2003)

A Proofs of lemmas on the interpretation

A.1 Definition of the interpretation

Lemma 6 *For all sets A and B, the set $(\mathcal{F}_p(A,B),\subseteq)$ of functional relations on $A\times B$ ordered by inclusion is a non-empty strictly inductive poset (i.e. every non-empty totally ordered subset has a least upper bound) with $\emptyset$ as smallest element.*

Proof. $\mathcal{F}_p(A,B)$ is not empty since it contains $\emptyset$, and $\emptyset$ is smaller than every element. Let $C=\{R_k\}$ be a non-empty totally ordered subset of $\mathcal{F}_p(A,B)$ and $R=\bigcup_k R_k$. We now prove that R is functional, hence that $R=\mathrm{lub}(C)$. Let $(a,b),(a,b')\in R$. By definition, there are k and k' such that $(a,b)\in R_k$ and $(a,b')\in R_{k'}$. Since C is totally ordered, either $R_k\subseteq R_{k'}$ or $R_{k'}\subseteq R_k$. In the first case, $(a,b)\in R_{k'}$ and $b=b'$ since $R_{k'}$ is functional. In the second case, $(a,b')\in R_k$ and $b=b'$ since R_k is a functional relation. □

Lemma 7 *G is monotone wrt inclusion.*

Proof. We first prove that D is monotone. Let $I\subseteq J$ and $T\in D(I)$. We have to show that $T\in D(J)$. To this end, we have to prove (1) $T\in\mathrm{SN}$ and (2) if $T\to^*(x:A)B$ then $A\in\mathrm{dom}(J)$ and, for all $a\in J(A)$, $B_x^a\in\mathrm{dom}(J)$:

1. Since $T\in D(I)$, we have $T\in\mathrm{SN}$.
2. Since $T\in D(I)$ and $T\to^*(x:A)B$, we have $A\in\mathrm{dom}(I)$ and, for all $a\in I(A)$, $B_x^a\in\mathrm{dom}(I)$. Since $I\subseteq J$, we have $\mathrm{dom}(I)\subseteq\mathrm{dom}(J)$ and $J(A)=I(A)$ since I and J are functional relations. Therefore, $A\in\mathrm{dom}(J)$ and, for all $a\in I(A)$, $B_x^a\in\mathrm{dom}(J)$.

We now prove that G is monotone. Let $I\subseteq J$ and $T\in D(I)$. We have to show that $(T,F(I)(T))\in G(J)$. First, $T\in D(J)$ since D is monotone. Second, we have to demonstrate that $F(I)(T)=F(J)(T)$.

If $T{\downarrow}=(x:A)B$, then $F(I)(T)=\Pi a\in I(A).\ I(B_x^a)$ and $F(J)(T)=\Pi a\in J(A).\ J(B_x^a)$. Since $T\in D(I)$, we have $A\in\mathrm{dom}(I)$ and, for all $a\in I(A)$, $B_x^a\in\mathrm{dom}(I)$. Since $\mathrm{dom}(I)\subseteq\mathrm{dom}(J)$, we have $J(A)=I(A)$ and, for all $a\in I(A)$, $J(B_x^a)=I(B_x^a)$. Therefore, $F(I)(T)=F(J)(T)$.

Now, if $T{\downarrow}$ is not a product, then $F(I)(T)=F(J)(T)=\mathrm{SN}$. □

A.2 Computability predicates

Lemma 8 *$\mathcal{D}$ is a computability predicate.*

Proof. Note that $\mathcal{D}=D(\mathcal{I})$.

1. $\mathcal{D}\subseteq\mathrm{SN}$ by definition of D.
2. Let $T\in\mathcal{D}$ and T' such that $T\to T'$. We have $T'\in\mathrm{SN}$ since $T\in\mathrm{SN}$. Assume now that $T'\to^*(x:A)B$. Then, $T\to^*(x:A)B$, $A\in\mathcal{D}$ and, for all $a\in\mathcal{I}(A)$, $B_x^a\in\mathcal{D}$. Therefore, $T'\in\mathcal{D}$.

3. Let T be a neutral term such that $\rightarrow(T) \subseteq \mathcal{D}$. Since $\mathcal{D} \subseteq \mathrm{SN}$, $T \in \mathrm{SN}$. Assume now that $T \rightarrow^* (x : A)B$. Since T is neutral, there is $U \in \rightarrow(T)$ such that $U \rightarrow^* (x : A)B$. Therefore, $A \in \mathcal{D}$ and, for all $a \in \mathcal{I}(A)$, $B_x^a \in \mathcal{D}$. $\square$

Lemma 9 *If $P \in \mathbb{P}$ and, for all $a \in P$, $Q(a) \in \mathbb{P}$, then $\Pi a \in P.\ Q(a) \in \mathbb{P}$.*

Proof. Let $R = \Pi a \in P.\ Q(a)$.

1. Let $t \in R$. We have to prove that $t \in \mathrm{SN}$. Let $x \in \mathbb{V}$. Since $P \in \mathbb{P}$, $x \in P$. So, $tx \in Q(x)$. Since $Q(x) \in \mathbb{P}$, $Q(x) \subseteq \mathrm{SN}$. Therefore, $tx \in \mathrm{SN}$, and $t \in \mathrm{SN}$.
2. Let $t \in R$ and t' such that $t \rightarrow t'$. We have to prove that $t' \in R$. Let $a \in P$. We have to prove that $t'a \in Q(a)$. By definition, $ta \in Q(a)$ and $ta \rightarrow t'a$. Since $Q(a) \in \mathbb{P}$, $t'a \in Q(a)$.
3. Let t be a neutral term such that $\rightarrow(t) \subseteq R$. We have to prove that $t \in R$. Hence, we take $a \in P$ and prove that $ta \in Q(a)$. Since $P \in \mathbb{P}$, we have $a \in \mathrm{SN}$ and $\rightarrow^*(a) \subseteq P$. We now prove that, for all $b \in \rightarrow^*(a)$, $tb \in Q(a)$, by induction on $\rightarrow$. Since t is neutral, tb is neutral too and it suffices to prove that $\rightarrow(tb) \subseteq Q(a)$. Since t is neutral, $\rightarrow(tb) = \rightarrow(t)b \cup t\rightarrow(b)$. By induction hypothesis, $t\rightarrow(b) \subseteq Q(a)$. By assumption, $\rightarrow(t) \subseteq R$. So, $\rightarrow(t)a \subseteq Q(a)$. Since $Q(a) \in \mathbb{P}$, $\rightarrow(t)b \subseteq Q(a)$ too. Therefore, $ta \in Q(a)$ and $t \in R$. $\square$

Lemma 10 *For all $T \in \mathcal{D}$, $\mathcal{I}(T)$ is a computability predicate.*

Proof. Since $\mathcal{F}_p(\mathbb{T}, \mathbb{P})$ is a strictly inductive poset, it suffices to prove that $\mathcal{F}_p(\mathbb{T}, \mathbb{P})$ is closed by G. Assume that $J \in \mathcal{F}_p(\mathbb{T}, \mathbb{P})$. We have to prove that $G(J) \in \mathcal{F}_p(\mathbb{T}, \mathbb{P})$, that is, for all $T \in D(J)$, $F(J)(T) \in \mathbb{P}$. There are two cases:

- If $T \in \mathrm{SN}$ and $T{\downarrow} = (x : A)B$, then $F(J)(T) = \Pi a \in J(A).\ J(B_x^a)$. By assumption, $J(A) \in \mathbb{P}$ and, for $a \in J(A)$, $J(B_x^a) \in \mathbb{P}$. Hence, by Lemma 9, $F(J)(T) \in \mathbb{P}$.
- Otherwise, $F(J)(T) = \mathrm{SN} \in \mathbb{P}$. $\square$

Lemma 1a *For all terms T and substitutions σ, $[\![T]\!]_\sigma \in \mathbb{P}$.*

Proof. By induction on T. If $T = s$, then $[\![T]\!]_\sigma = \mathcal{D} \in \mathbb{P}$ by Lemma 8. If $T = (x : A)K \in \mathbb{K}$, then $[\![T]\!]_\sigma = \Pi a \in [\![A]\!]_\sigma.\ [\![K]\!]_{[x \mapsto a, \sigma]}$. By induction hypothesis, $[\![A]\!]_\sigma \in \mathbb{P}$ and, for all $a \in [\![A]\!]_\sigma$, $[\![K]\!]_{[x \mapsto a, \sigma]} \in \mathbb{P}$. Hence, by Lemma 9, $[\![T]\!]_\sigma \in \mathbb{P}$. If $T \notin \mathbb{K} \cup \{\square\}$ and $T\sigma \in \mathcal{D}$, then $[\![T]\!]_\sigma = \mathcal{I}(T\sigma) \in \mathbb{P}$ by Lemma 10. Otherwise, $[\![T]\!]_\sigma = \mathrm{SN} \in \mathbb{P}$. $\square$

A.3 Invariance by reduction

We now prove that the interpretation is invariant by reduction.

Lemma 11 *If $T \in \mathcal{D}$ and $T \rightarrow T'$, then $\mathcal{I}(T) = \mathcal{I}(T')$.*

Proof. First note that $T' \in \mathcal{D}$ since $\mathcal{D} \in \mathbb{P}$. Hence, $\mathcal{I}(T')$ is well defined. Now, we have $T \in \mathrm{SN}$ since $\mathcal{D} \subseteq \mathrm{SN}$. So, $T' \in \mathrm{SN}$ and, by local confluence and Newman's lemma, $T{\downarrow} = T'{\downarrow}$. If $T{\downarrow} = (x : A)B$ then $\mathcal{I}(T) = \Pi a \in \mathcal{I}(A).\ \mathcal{I}(B_x^a) = \mathcal{I}(T')$. Otherwise, $\mathcal{I}(T) = \mathrm{SN} = \mathcal{I}(T')$. $\square$

Lemma 1b *If T is typable, $T\sigma \in \mathcal{D}$ and $T \to T'$, then $[\![T]\!]_\sigma = [\![T']\!]_\sigma$.*

Proof. By assumption, there are Γ and U such that $\Gamma \vdash T : U$. Since $\to$ preserves typing, we also have $\Gamma \vdash T' : U$. So, $T \neq \Box$, and $T' \neq \Box$. Moreover, $T \in \mathbb{K}$ iff $T' \in \mathbb{K}$ since $\Gamma \vdash T : \Box$ iff $T \in \mathbb{K}$ and T is typable. In addition, we have $T'\sigma \in \mathcal{D}$ since $T\sigma \in \mathcal{D}$ and $\mathcal{D} \in \mathbb{P}$.

We now prove the result, with $T \to^= T'$ instead of $T \to T'$, by induction on T. If $T \notin \mathbb{K}$, then $T' \notin \mathbb{K}$ and, since $T\sigma, T'\sigma \in \mathcal{D}$, $[\![T]\!]_\sigma = \mathcal{I}(T\sigma) = \mathcal{I}(T'\sigma) = [\![T']\!]_\sigma$ by Lemma 11. If $T = \star$, then $[\![T]\!]_\sigma = \mathcal{D} = [\![T']\!]_\sigma$. Otherwise, $T = (x : A)K$ and $T' = (x : A')K'$ with $A \to^= A'$ and $K \to^= K'$. By inversion, we have $\Gamma \vdash A : \star$, $\Gamma \vdash A' : \star$, $\Gamma, x : A \vdash K : \Box$ and $\Gamma, x : A' \vdash K' : \Box$. So, by induction hypothesis, $[\![A]\!]_\sigma = [\![A']\!]_\sigma$ and, for all $a \in [\![A]\!]_\sigma$, $[\![K]\!]_{\sigma'} = [\![K']\!]_{\sigma'}$, where $\sigma' = [x \mapsto a, \sigma]$. Therefore, $[\![T]\!]_\sigma = [\![T']\!]_\sigma$. □

Lemma 1c *If T is typable, $T\sigma \in \mathcal{D}$ and $\sigma \to \sigma'$, then $[\![T]\!]_\sigma = [\![T]\!]_{\sigma'}$.*

Proof. By induction on T.

- If $T \in \mathbb{S}$, then $[\![T]\!]_\sigma = \mathcal{D} = [\![T]\!]_{\sigma'}$.
- If $T = (x : A)K$ and $K \in \mathbb{K}$, then $[\![T]\!]_\sigma = \Pi a \in [\![A]\!]_\sigma.\ [\![K]\!]_{[x \mapsto a, \sigma]}$ and $[\![T]\!]_{\sigma'} = \Pi a \in [\![A]\!]_{\sigma'}.\ [\![K]\!]_{[x \mapsto a, \sigma']}$. By induction hypothesis, $[\![A]\!]_\sigma = [\![A]\!]_{\sigma'}$ and, for all $a \in [\![A]\!]_\sigma$, $[\![K]\!]_{[x \mapsto a, \sigma]} = [\![K]\!]_{[x \mapsto a, \sigma']}$. Therefore, $[\![T]\!]_\sigma = [\![T]\!]_{\sigma'}$.
- If $T\sigma \in \mathcal{D}$, then $[\![T]\!]_\sigma = \mathcal{I}(T\sigma)$ and $[\![T]\!]_{\sigma'} = \mathcal{I}(T\sigma')$. Since $T\sigma \to^* T\sigma'$, by Lemma 1b, $\mathcal{I}(T\sigma) = \mathcal{I}(T\sigma')$.
- Otherwise, $[\![T]\!]_\sigma = \mathrm{SN} = [\![T]\!]_{\sigma'}$. □

A.4 Adequacy of the interpretation

Lemma 1d *If $(x : A)B$ is typable, $((x : A)B)\sigma \in \mathcal{D}$ and $x \notin \mathrm{dom}(\sigma) \cup \mathrm{FV}(\sigma)$, then $[\![(x : A)B]\!]_\sigma = \Pi a \in [\![A]\!]_\sigma.\ [\![B]\!]_{[x \mapsto a, \sigma]}$.*

Proof. If B is a kind, this is immediate. Otherwise, since $((x : A)B)\sigma \in \mathcal{D}$, $[\![(x : A)B]\!]_\sigma = \mathcal{I}(((x : A)B)\sigma)$. Since $x \notin \mathrm{dom}(\sigma) \cup \mathrm{FV}(\sigma)$, we have $((x : A)B)\sigma = (x : A\sigma)B\sigma$. Since $(x : A\sigma)B\sigma \in \mathcal{D}$ and $\mathcal{D} \subseteq \mathrm{SN}$, we have $[\![(x : A)B]\!]_\sigma = \Pi a \in \mathcal{I}(A\sigma{\downarrow}).\ \mathcal{I}((B\sigma{\downarrow})^a_x)$.

Since $(x : A)B$ is typable, A is of type $\star$ and $A \notin \mathbb{K} \cup \{\Box\}$. Hence, $[\![A]\!]_\sigma = \mathcal{I}(A\sigma)$ and, by Lemma 11, $\mathcal{I}(A\sigma) = \mathcal{I}(A\sigma{\downarrow})$.

Since $(x : A)B$ is typable and not a kind, B is of type $\star$ and $B \notin \mathbb{K} \cup \{\Box\}$. Hence, $[\![B]\!]_{[x \mapsto a, \sigma]} = \mathcal{I}(B[x \mapsto a, \sigma])$. Since $x \notin \mathrm{dom}(\sigma) \cup \mathrm{FV}(\sigma)$, $B[x \mapsto a, \sigma] = (B\sigma)^a_x$. Hence, $[\![B]\!]_{[x \mapsto a, \sigma]} = \mathcal{I}((B\sigma)^a_x)$ and, by Lemma 11, $\mathcal{I}((B\sigma)^a_x) = \mathcal{I}((B\sigma{\downarrow})^a_x)$.

Therefore, $[\![(x : A)B]\!]_\sigma = \Pi a \in [\![A]\!]_\sigma.\ [\![B]\!]_{[x \mapsto a, \sigma]}$. □

Note that, by iterating this lemma, we get that $v \in [\![(\boldsymbol{x} : \boldsymbol{T})U]\!]$ iff, for all $\boldsymbol{t}$ such that $[\boldsymbol{x} \mapsto \boldsymbol{t}] \models \boldsymbol{x} : \boldsymbol{T}$, $v\boldsymbol{t} \in [\![U]\!]_{[\boldsymbol{x} \mapsto \boldsymbol{t}]}$.

Lemma 1e *If $\Delta \vdash U : s$, $\Gamma \vdash \gamma : \Delta$ and $U\gamma\sigma \in \mathcal{D}$, then $[\![U\gamma]\!]_\sigma = [\![U]\!]_{\gamma\sigma}$.*

Proof. We proceed by induction on U. Since $\Delta \vdash U : s$ and $\Gamma \vdash \gamma : \Delta$, we have $\Gamma \vdash U\gamma : s$.

- If $s = \star$, then $U, U\gamma \notin \mathbb{K} \cup \{\square\}$ and $[\![U\gamma]\!]_\sigma = \mathcal{I}(U\gamma\sigma) = [\![U]\!]_{\gamma\sigma}$ since $U\gamma\sigma \in \mathcal{D}$.
- Otherwise, $s = \square$ and $U \in \mathbb{K}$.
  - If $U = \star$, then $[\![U\gamma]\!]_\sigma = \mathcal{D} = [\![U]\!]_{\gamma\sigma}$.
  - Otherwise, $U = (x : A)K$ and, by Lemma 1d, $[\![U\gamma]\!]_\sigma = \Pi a \in [\![A\gamma]\!]_\sigma.\ [\![K\gamma]\!]_{[x\mapsto a,\sigma]}$ and $[\![U]\!]_{\gamma\sigma} = \Pi a \in [\![A]\!]_{\gamma\sigma}.\ [\![K]\!]_{[x\mapsto a,\gamma\sigma]}$. By induction hypothesis, $[\![A\gamma]\!]_\sigma = [\![A]\!]_{\gamma\sigma}$ and, for all $a \in [\![A\gamma]\!]_\sigma$, $[\![K\gamma]\!]_{[x\mapsto a,\sigma]} = [\![K]\!]_{\gamma[x\mapsto a,\sigma]}$. Wlog we can assume $x \notin \mathrm{dom}(\gamma) \cup \mathrm{FV}(\gamma)$. So, $[\![K]\!]_{\gamma[x\mapsto a,\sigma]} = [\![K]\!]_{[x\mapsto a,\gamma\sigma]}$. □

Lemma 1f *Let P be a computability predicate and Q a P-indexed family of computability predicates such that $Q(a') \subseteq Q(a)$ whenever $a \to a'$. Then, $\lambda x : A.b \in \Pi a \in P.\ Q(a)$ whenever $A \in \mathrm{SN}$ and, for all $a \in P$, $b_x^a \in Q(a)$.*

Proof. Let $a_0 \in P$. Since $P \in \mathbb{P}$, we have $a_0 \in \mathrm{SN}$ and $x \in P$. Since $Q(x) \in \mathbb{P}$ and $b = b_x^x \in Q(x)$, we have $b \in \mathrm{SN}$. Let $a \in \to^* (a_0)$. We can prove that $(\lambda x : A.b)a \in Q(a_0)$ by induction on (A, b, a) ordered by $(\to, \to, \to)_{\mathrm{prod}}$. Since $Q(a_0) \in \mathbb{P}$ and $(\lambda x : A.b)a$ is neutral, it suffices to prove that $\to((\lambda x : A.b)a) \subseteq Q(a_0)$. If the reduction takes place in A, b or a, we can conclude by induction hypothesis. Otherwise, $(\lambda x : A.b)a \to b_x^a \in Q(a)$ by assumption. Since $a_0 \to^* a$ and $Q(a') \subseteq Q(a)$ whenever $a \to a'$, we have $b_x^a \in Q(a_0)$. □

B Termination proof of Example 1

We first check that this system is well-structured. For each rule $f\boldsymbol{l} \to r$, we take the environment $\Delta_{f\boldsymbol{l}\to r}$ made of all the variables of r with the following types: `a:Set, b:Set, p:`$\mathbb{N}$`, q:`$\mathbb{N}$`, x:El a, l:`$\mathbb{L}$` a p, m:`$\mathbb{L}$` a q, f:El a`$\Rightarrow \mathbb{B}$.

The precedence infered for this example is the smallest containing:

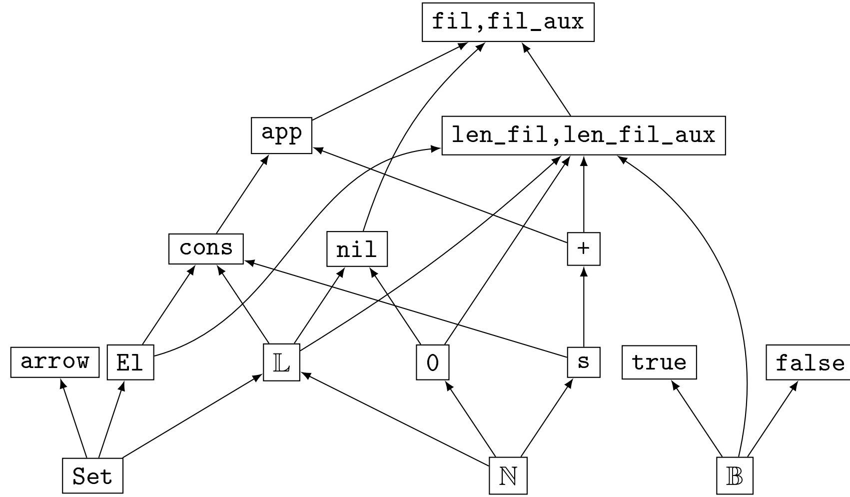
- comparisons linked to the typing of symbols:

`Set` $\preceq$ `arrow`	`Set`,$\mathbb{L}$,`0` $\preceq$ `nil`
`Set` $\preceq$ `El`	`Set`,`El`,$\mathbb{N}$,$\mathbb{L}$,`s` $\preceq$ `cons`
$\mathbb{B} \preceq$ `true`	`Set`,$\mathbb{N}$,$\mathbb{L}$,`+` $\preceq$ `app`
$\mathbb{B} \preceq$ `false`	`Set`,`El`,$\mathbb{B}$,$\mathbb{N}$,$\mathbb{L}$ $\preceq$ `len_fil`
$\mathbb{N} \preceq$ `0`	$\mathbb{B}$,`Set`,`El`,$\mathbb{N}$,$\mathbb{L}$ $\preceq$ `len_fil_aux`
$\mathbb{N} \preceq$ `s`	`Set`,`El`,$\mathbb{B}$,$\mathbb{N}$,$\mathbb{L}$,`len_fil` $\preceq$ `fil`
$\mathbb{N} \preceq$ `+`	$\mathbb{B}$,`Set`,`El`,$\mathbb{N}$,$\mathbb{L}$,`len_fil_aux` $\preceq$ `fil_aux`
`Set`,$\mathbb{N} \preceq \mathbb{L}$	

- and comparisons related to calls:

`s` $\preceq$ `+`	`s`,`len_fil` $\preceq$ `len_fil_aux`
`cons`,`+` $\preceq$ `app`	`nil`,`fil_aux`,`app`,`len_fil` $\preceq$ `fil`
`0`,`len_fil_aux`,`+` $\preceq$ `len_fil`	`fil`,`cons`,`len_fil` $\preceq$ `fil_aux`

This precedence can be sum up in the following diagram, where symbols in the same box are equivalent:



The matrices in Example 4 are:

$$\mathtt{A,B}{=}(\,{-1}\,),\ \mathtt{C}{=}\begin{pmatrix}-1&\infty\\\infty&0\end{pmatrix},\ \mathtt{D}{=}\begin{pmatrix}\infty&\infty\\-1&\infty\\\infty&0\\\infty&\infty\end{pmatrix},\ \mathtt{E}{=}\begin{pmatrix}\infty&\infty&\infty&\infty\\-1&-1&\infty&\infty\\\infty&\infty&0&\infty\\\infty&\infty&\infty&0\end{pmatrix},\ \mathtt{F}{=}\begin{pmatrix}\infty&0&\infty&\infty\\\infty&\infty&\infty&\infty\\\infty&\infty&-1&-1\end{pmatrix},$$

$$\mathtt{J}{=}\mathtt{K}{=}\begin{pmatrix}\infty&\infty&\infty\\0&\infty&\infty\\\infty&0&\infty\\\infty&\infty&0\end{pmatrix},\ \mathtt{G}{=}\begin{pmatrix}\infty&\infty\\\infty&\infty\\\infty&\infty\end{pmatrix},\ \mathtt{H}{=}\mathtt{I}{=}\mathtt{N}{=}\mathtt{O}{=}\mathtt{P}{=}\mathtt{Q}{=}\begin{pmatrix}0&\infty&\infty\\\infty&\infty&\infty\\\infty&-1&-1\end{pmatrix},\ \mathtt{L}{=}\begin{pmatrix}\infty&0&\infty&\infty&\infty\\\infty&\infty&\infty&\infty&\infty\\\infty&\infty&-1&-1&-1\end{pmatrix},$$

$$\mathtt{M}{=}\begin{pmatrix}\infty&\infty&\infty&\infty\\\infty&\infty&\infty&\infty\\\infty&\infty&\infty&\infty\end{pmatrix},\ \mathtt{R}{=}\mathtt{S}{=}\mathtt{T}{=}\begin{pmatrix}\infty&\infty&\infty\\0&\infty&\infty\\\infty&\infty&\infty\\\infty&0&\infty\\\infty&\infty&0\end{pmatrix}.$$

Which leads to the matrices labeling a loop in the transitive closure:

$$\mathtt{TC}_1{=}\mathtt{J}{\times}\mathtt{F}{=}\begin{pmatrix}\infty&\infty&\infty&\infty\\\infty&0&\infty&\infty\\\infty&\infty&\infty&\infty\\\infty&\infty&-1&-1\end{pmatrix},\ \mathtt{TC}_2{=}\mathtt{S}{\times}\mathtt{L}{=}\begin{pmatrix}\infty&\infty&\infty&\infty&\infty\\\infty&0&\infty&\infty&\infty\\\infty&\infty&\infty&\infty&\infty\\\infty&\infty&\infty&\infty&\infty\\\infty&\infty&-1&-1&-1\end{pmatrix},$$

$$\mathtt{TC}_3{=}\mathtt{L}{\times}\mathtt{S}{=}\mathtt{TC}_4{=}\mathtt{F}{\times}\mathtt{J}{=}\begin{pmatrix}0&\infty&\infty\\\infty&\infty&\infty\\\infty&-1&-1\end{pmatrix}{=}\mathtt{O}{=}\mathtt{H}.$$

It would be useless to compute matrices labeling edges which are not in a strongly connected component of the call-graph (like $\mathtt{S}{\times}\mathtt{R}$), but it is necessary to compute all the products which could label a loop, especially to verify that all loop-labeling matrices are idempotent, which is indeed the case here.