Dependency Pairs Termination in Dependent Type Theory Modulo Rewriting

Frédéric Blanqui
INRIA, France
LSV, ENS Paris-Saclay, CNRS, Université Paris-Saclay, France

Guillaume Genestier
LSV, ENS Paris-Saclay, CNRS, Université Paris-Saclay, France
MINES ParisTech, PSL University, Paris, France

Olivier Hermant
MINES ParisTech, PSL University, Paris, France

Abstract

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

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

Keywords and phrases termination, higher-order rewriting, dependent types, dependency pairs

Digital Object Identifier 10.4230/LIPIcs.FSCD.2019.9

Acknowledgements The authors thank the anonymous referees for their comments, which have improved the quality of this article.

1 Introduction

Termination, that is, the absence of infinite computations, is an important problem in software verification, as well as in logic. In logic, it 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 rewriting rules [33] 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 rewriting rules given by the user to define not only functions but also types. It extends Barendregt's Pure Type System (PTS) λP [3], the logical framework LF [16] and Martin-Löf's type theory. It can encode any functional PTS like System F or the Calculus of Constructions [10].

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

Our criterion has been implemented in Dedukti, a type-checker for $\lambda\Pi/\mathcal{R}$ that we will use in our examples. The code is available in [12] 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.

This criterion is based on dependency pairs, an important concept in the termination of first-order term rewriting systems. It generalizes the notion of recursive call in first-order functional programs to rewriting. Namely, the dependency pairs of a rewriting 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 rewriting rules. Dependency pairs have been introduced by Arts and Giesl [2] and have evolved into a general framework for termination [13]. It is now at the heart of many state-of-the-art automated termination provers for first-order rewriting systems and HASKELL, JAVA or C programs.

Dependency pairs have been extended to different simply-typed settings for higher-order rewriting: Combinatory Reduction Systems [23] and Higher-order Rewriting Systems [29], with two different approaches: dynamic dependency pairs include variable applications [24], while static dependency pairs exclude them by slightly restricting the class of systems that can be considered [25]. Here, we use the static approach.

In [37], Wahlstedt considered a system slightly less general than $\lambda\Pi/\mathcal{R}$ for which he provided conditions that imply the weak normalization, that is, the existence of a finite reduction to normal form. In his system, $\mathcal{R}$ 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). Wahlstedt's proof proceeds in two modular steps. First, he proves that typable terms have a normal form if there is no infinite sequence of function calls. Second, he proves that there is no infinite sequence of function calls if $\mathcal{R}$ satisfies Lee, Jones and Ben-Amram's size-change termination criterion (SCT) [26].

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: rules can be non-left-linear and have joinable critical pairs.

In [5], 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 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 [5], 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 sequence of dependency pairs. But other criteria have been developed for this purpose that could be adapted to $\lambda\Pi/\mathcal{R}$.

Outline

The main result is Theorem 11 stating that, for a large class of rewriting systems $\mathcal{R}$, the combination of β and $\mathcal{R}$ is strongly normalizing on terms typable in $\lambda\Pi/\mathcal{R}$ if, roughly speaking, there is no infinite sequence of dependency pairs.

The proof involves two steps. First, after recalling the terms and types of $\lambda\Pi/\mathcal{R}$ in Section 2, we introduce in Section 3 a model of this calculus based on Girard's reducibility candidates [15], and prove that every typable term is strongly normalizing if every symbol of the signature is in the interpretation of its type (Adequacy lemma). Second, in Section 4, we introduce our notion of dependency pair and prove that every symbol of the signature is in the interpretation of its type if there is no infinite sequence of dependency pairs.

In order to show the usefulness of this result, we give simple criteria for checking the conditions of the theorem. In Section 5, we show that *plain function passing* systems belong to the class of systems that we consider. And in Section 6, we show how to use size-change termination to obtain the termination of the dependency pair relation.

Finally, in Section 7 we compare our criterion with other criteria and tools and, in Section 8, we summarize our results and give some hints on possible extensions.

For lack of space, some proofs are given in an appendix 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 [3]:

$$t \in \mathbb{T} = s \in \mathbb{S} \mid x \in \mathbb{V} \mid f \in \mathbb{F} \mid \forall x : t, t \mid tt \mid \lambda x : t, t$$

where $\mathbb{S} = \{\text{TYPE}, \text{KIND}\}$ is the set of sorts[1], $\mathbb{V}$ 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.

Furthermore, we assume given a set $\mathcal{R}$ of rules $l \to r$ such that $\text{FV}(r) \subseteq \text{FV}(l)$ and l is of the form $f\vec{l}$. A symbol f is said to be defined if there is a rule of the form $f\vec{l} \to r$. 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: we consider rewriting with syntactic matching only. Following [6], it should however be possible 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 $\to(t) = \{u \in \mathbb{T} \mid t \to u\}$ be the set of immediate reducts of t.

A typing environment Γ is a (possibly empty) sequence $x_1 : T_1, \ldots, x_n : T_n$ of pairs of variables and terms, where the variables are distinct, written $\vec{x} : \vec{T}$ for short. Given an environment $\Gamma = \vec{x} : \vec{T}$ and a term U, let $\forall\Gamma, U$ be $\forall\vec{x} : \vec{T}, U$. The product arity $\text{ar}(T)$ of a term T is the integer $n \in \mathbb{N}$ such that $T = \forall x_1 : T_1, \ldots \forall x_n : T_n, U$ and U is not a product. Let $\vec{t}$ denote a possibly empty sequence of terms $t_1, \ldots, t_n$ of length $|\vec{t}| = n$, and $\text{FV}(t)$ be the set of free variables of t.

For each $f \in \mathbb{F}$, we assume given a term Θ_f and a sort s_f, and let Γ_f be the environment such that $\Theta_f = \forall\Gamma_f, U$ and $|\Gamma_f| = \text{ar}(\Theta_f)$.

The application of a substitution σ to a term t is written $t\sigma$. Given a substitution σ, let $\text{dom}(\sigma) = \{x | x\sigma \neq x\}$, $\text{FV}(\sigma) = \bigcup_{x \in \text{dom}(\sigma)} \text{FV}(x\sigma)$ and $[x \mapsto a, \sigma]$ ($[x \mapsto a]$ if σ is the identity) be the substitution $\{(x, a)\} \cup \{(y, b) \in \sigma \mid y \neq x\}$. Given another substitution σ', let $\sigma \to \sigma'$ if there is x such that $x\sigma \to x\sigma'$ 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) similar to (var). Moreover, (conv) uses $\downarrow$ instead of $\downarrow_\beta$, where $\downarrow \, = \, \to^* \, {}^*\!\leftarrow$ is the joinability relation and $\to^*$ the reflexive and transitive closure of $\to$. 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 (and we usually use capital letters for types). Hence, KIND is the type of TYPE, Θ_f is the type of f, and s_f is the type of Θ_f. Common data types like natural numbers $\mathbb{N}$ are usually declared in $\lambda\Pi$ as function symbols of type TYPE: $\Theta_\mathbb{N} = \text{TYPE}$ and $s_\mathbb{N} = \text{KIND}$.

[1] Sorts refer here to the notion of sort in Pure Type Systems, not the one used in some first-order settings.

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

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

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

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

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

The dependent product $\forall x : A, B$ 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. If B does not depend on x, we sometimes simply write $A \Rightarrow B$.

Typing induces a hierarchy on terms [4, Lemma 47]. At the top, there is the sort KIND that is not typable. Then, comes the class $\mathbb{K}$ of kinds, whose type is KIND: $K = \text{TYPE} \mid \forall x : t, K$ where $t \in \mathbb{T}$. Then, comes the class of predicates, whose types are kinds. Finally, at the bottom lie (proof) objects whose types are predicates.

▶ **Example 1** (Filter function on dependent lists)**.** To illustrate the kind of systems we consider, we give an extensive example in the new DEDUKTI syntax combining type-level rewriting rules (`El` converts datatype codes into DEDUKTI 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 matching on defined function symbols (`fil` can match a list defined by concatenation). Note that this example cannot be represented in COQ or AGDA because of the rules using matching on `app`. And its termination can be handled neither by [37] nor by [5] 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 [12]. For readability, we removed the `&` which are used to identify pattern variables in the rewriting rules.

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

symbol El: Set ⇒ TYPE         rule El (arrow a b) → El a ⇒ El b

symbol Bool: TYPE       symbol true: Bool       symbol false: Bool
symbol Nat: TYPE        symbol zero: Nat        symbol s: Nat ⇒ Nat

symbol plus: Nat ⇒ Nat ⇒Nat      set infix 1 "+" ≔ plus
  rule zero + q → q              rule (s p) + q → s (p + q)

symbol List: Set ⇒ Nat ⇒ TYPE
  symbol nil: ∀a, List a zero
  symbol cons:∀a, El a ⇒ ∀p, List a p ⇒ List a (s p)

symbol app: ∀a p, List a p ⇒ ∀q, List a q ⇒ List 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 ⇒ Bool) ⇒ ∀p, List a p ⇒ Nat
symbol len_fil_aux: Bool ⇒ ∀a, (El a ⇒ Bool) ⇒ ∀p, List a p ⇒ Nat
```

```
  rule len_fil a f _ (nil _)          → zero
  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, List a (len_fil a f p l)
symbol fil_aux: ∀b a f, El a ⇒ ∀p l, List 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. Throughout the paper, we assume that $\to$ is locally confluent ($\leftarrow\to \subseteq \downarrow$) and preserves typing (for all Γ, A, t and u, if $\Gamma \vdash t : A$ and $t \to u$, then $\Gamma \vdash u : A$).

Note that local confluence implies that every $t \in \mathrm{SN}$ has a unique normal form $t\downarrow$.

These assumptions are used in the interpretation of types (Definition 2) and the adequacy lemma (Lemma 5). Both properties are undecidable in general. For confluence, DEDUKTI can call confluence checkers that understand the HRS format of the confluence competition. For preservation of typing by reduction, it implements an heuristic [31].

3 Interpretation of types as reducibility candidates

We aim to prove the termination of the union of two relations, $\to_\beta$ and $\to_\mathcal{R}$, on the set of well-typed terms (which depends on $\mathcal{R}$ since $\downarrow$ includes $\to_\mathcal{R}$). As is well known, termination is not modular in general. As a β step can generate an $\mathcal{R}$ step, and vice versa, we cannot expect to prove the termination of $\to_\beta \cup \to_\mathcal{R}$ from the termination of $\to_\beta$ and $\to_\mathcal{R}$. The termination of $\lambda\Pi/\mathcal{R}$ cannot be reduced to the termination of the simply-typed λ-calculus either (as done for $\lambda\Pi$ alone in [16]) because of type-level rewriting rules like the ones defining `El` in Example 1. Indeed, type-level rules enable the encoding of functional PTS like Girard's System F, whose termination cannot be reduced to the termination of the simply-typed λ-calculus [10].

So, following Girard [15], to prove the termination of $\to_\beta \cup \to_\mathcal{R}$, 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 $T \to T'$, then we must have $[\![T]\!] = [\![T']\!]$.
- As usual, to handle β-reduction, we need a product type $\forall x : A, B$ to be interpreted by the set of terms t such that, for all a in the interpretation of A, ta is in the interpretation of $B[x \mapsto a]$, that is, we must have $[\![\forall x : A, B]\!] = \Pi a \in [\![A]\!].\, [\![B[x \mapsto a]]\!]$ where $\Pi a \in P.\, Q(a) = \{t \mid \forall a \in P, ta \in Q(a)\}$.

First, we define the interpretation of predicates (and TYPE) as the least fixpoint of a monotone function in a directed-complete (= chain-complete) partial order [28]. Second, we define the interpretation of kinds by induction on their size.

▶ **Definition 2** (Interpretation of types). *Let $\mathbb{I} = \mathcal{F}_p(\mathbb{T}, \mathcal{P}(\mathbb{T}))$ be the set of partial functions from $\mathbb{T}$ to the powerset of $\mathbb{T}$. It is directed-complete wrt inclusion, allowing us to define $\mathcal{I}$ as the least fixpoint of the monotone function $F : \mathbb{I} \to \mathbb{I}$ such that, if $I \in \mathbb{I}$, then:*

- *The domain of $F(I)$ is the set $D(I)$ of all the terminating terms T such that, if T reduces to some product term $\forall x : A, B$ (not necessarily in normal form), then $A \in \mathrm{dom}(I)$ and, for all $a \in I(A)$, $B[x \mapsto a] \in \mathrm{dom}(I)$.*
- *If $T \in D(I)$ and the normal form[2] of T is not a product, then $F(I)(T) = \mathrm{SN}$.*
- *If $T \in D(I)$ and $T{\downarrow} = \forall x : A, B$, then $F(I)(T) = \Pi a \in I(A).\, I(B[x \mapsto a])$.*

We now introduce $\mathcal{D} = D(\mathcal{I})$ and define the interpretation of a term T wrt to a substitution σ, $[\![T]\!]_\sigma$ (and simply $[\![T]\!]$ if σ is the identity), as follows:

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

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

Let $\mathbb{C}$ be the set of terms of the form $f\vec{t}$ such that $|\vec{t}| = \mathrm{ar}(\Theta_f)$, $\vdash \Theta_f : s_f$, $\Theta_f \in [\![s_f]\!]$ and, if $\Gamma_f = \vec{x} : \vec{A}$ and $\sigma = [\vec{x} \mapsto \vec{t}]$, then $\sigma \models \Gamma_f$. (Informally, $\mathbb{C}$ is the set of terms obtained by fully applying some function symbol to computable arguments.)

We can then prove that, for all terms T, $[\![T]\!]$ satisfies Girard's conditions of reducibility candidates, called computability predicates here, adapted to rewriting by including in neutral terms every term of the form $f\vec{t}$ when f is applied to enough arguments wrt $\mathcal{R}$ [5]:

▶ **Definition 3** (Computability predicates). *A term is neutral if it is of the form $(\lambda x : A, t)u\vec{v}$, $x\vec{v}$ or $f\vec{v}$ with, for every rule $f\vec{l} \to r \in \mathcal{R}$, $|\vec{l}| \le |\vec{v}|$.*

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 satisfy the following key property: if t is neutral then, for all u, tu is neutral and every reduct of tu is either of the form $t'u$ with t' a reduct of t, or of the form tu' with u' a reduct of 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 Appendix A):

▶ **Lemma 4.**

(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 $\forall x : A, B$ is typable and $\forall x : A\sigma, B\sigma \in \mathcal{D}$, then $[\![\forall x : A, B]\!]_\sigma = \Pi a \in [\![A]\!]_\sigma.\, [\![B]\!]_{[x \mapsto a, \sigma]}$.*
(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 $a \in P$, $b[x \mapsto a] \in Q(a)$.*

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

[2] Because we assume local confluence, every terminating term T has a unique normal form $T{\downarrow}$.

▶ **Lemma 5** (Adequacy). *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 $T = \mathrm{KIND}$ or $\Gamma \vdash T : s$ [4, Lemma 28]. Moreover, if $\Gamma \vdash a : A$, $A \downarrow B$ and $\Gamma \vdash B : s$ (the premises of the (conv) rule), then $\Gamma \vdash A : s$ [4, Lemma 42] (because $\to$ preserves typing). 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 $\Gamma \vdash \forall x : A, B : s$ 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 $\mathrm{TYPE} \in [\![\mathrm{KIND}]\!]_\sigma = \mathcal{D}$.

(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 $(\forall x : A, B)\sigma$ in $[\![s]\!]_\sigma = \mathcal{D}$? Wlog we can assume $x \notin \mathrm{dom}(\sigma) \cup \mathrm{FV}(\sigma)$. So, $(\forall x : A, B)\sigma = \forall x : A\sigma, B\sigma$. By induction hypothesis, $A\sigma \in [\![\mathrm{TYPE}]\!]_\sigma = \mathcal{D}$. 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 $B\sigma' = (B\sigma)[x \mapsto a]$. Therefore, $(\forall x : A, B)\sigma \in [\![s]\!]_\sigma$.

(abs) Is $(\lambda x : A, b)\sigma$ in $[\![\forall x : A, B]\!]_\sigma$? 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 4d, $[\![\forall x : A, B]\!]_\sigma = \Pi a \in [\![A]\!]_\sigma.\, [\![B]\!]_{[x\mapsto a,\sigma]}$. By Lemma 4c, $[\![B]\!]_{[x\mapsto a,\sigma]}$ 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 4f, $\lambda x : A\sigma, b\sigma \in [\![\forall x : A, B]\!]_\sigma$ if $A\sigma \in \mathrm{SN}$ and, for all $a \in [\![A]\!]_\sigma$, $(b\sigma)[x \mapsto a] \in [\![B]\!]_{\sigma'}$ where $\sigma' = [x \mapsto a, \sigma]$. By induction hypothesis, $(\forall x : A, B)\sigma \in [\![s]\!]_\sigma = \mathcal{D}$. Since $x \notin \mathrm{dom}(\sigma) \cup \mathrm{FV}(\sigma)$, $(\forall x : A, B)\sigma = \forall x : A\sigma, B\sigma$ and $(b\sigma)[x \mapsto a] = b\sigma'$. 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 $[\![B[x \mapsto a]]\!]_\sigma$? By induction hypothesis, $t\sigma \in [\![\forall x : A, B]\!]_\sigma$, $a\sigma \in [\![A]\!]_\sigma$ and $(\forall x : A, B)\sigma \in [\![s]\!] = \mathcal{D}$. By Lemma 4d, $[\![\forall x : A, B]\!]_\sigma = \Pi \alpha \in [\![A]\!]_\sigma.\, [\![B]\!]_{[x\mapsto\alpha,\sigma]}$. 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 4e, $[\![B]\!]_{\sigma'} = [\![B[x \mapsto a]]\!]_\sigma$.

(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 4b, $[\![A]\!]_\sigma = [\![B]\!]_\sigma$. 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. ◀

4 Dependency pairs theorem

Now, we prove that 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 6** (Dependency pairs). *Let $f\vec{l} > g\vec{m}$ iff there is a rule $f\vec{l} \to r \in \mathcal{R}$, g is defined and $g\vec{m}$ is a subterm of r such that $\vec{m}$ 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 $\mathbb{C}$ (Def. 2), where $f\vec{t} \to_{\mathrm{arg}} f\vec{u}$ iff $\vec{t} \to_{prod} \vec{u}$ (reduction in one argument), and $>_s$ is the closure by substitution and left-application of $>$: $ft_1 \ldots t_p \mathrel{\tilde{>}} gu_1 \ldots u_q$ iff there are a dependency pair $fl_1 \ldots l_i > gm_1 \ldots m_j$ with $i \leq p$ and $j \leq q$ and a substitution σ such that, for all $k \leq i$, $t_k \to^* l_k\sigma$ and, for all $k \leq j$, $m_k\sigma = u_k$.

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

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

whose dependency pairs are `f x y > app (f x) y` and `f x y > f x`, 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 7.** The rules of Example 1 have the following dependency pairs (the pairs whose left-hand side is headed by `fil` or `fil_aux` can be found in Appendix B):

```
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
```

In [2], a sequence of dependency pairs interleaved with $\to_{\text{arg}}$ steps is called a chain. Arts and Giesl proved that, in a first-order term algebra, $\to_{\mathcal{R}}$ terminates if and only if there are no infinite chains, that is, if and only if $\tilde{>}$ terminates. Moreover, in a first-order term algebra, $\tilde{>}$ terminates if and only if, for all f and $\vec{t}$, $f\vec{t}$ terminates wrt $\tilde{>}$ whenever $\vec{t}$ terminates wrt $\to$. 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 8** (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 $f\vec{l} \to r \in \mathcal{R}$ with g (defined or undefined) occurring in r. Then, let $\succ\ = \succeq \setminus \preceq$ be the strict part of $\succeq$. A rewriting system $\mathcal{R}$ is well-structured if:*

(a) *$\succ$ is well-founded;*
(b) *for every rule $f\vec{l} \to r$, $|\vec{l}| \le \operatorname{ar}(\Theta_f)$;*
(c) *for every dependency pair $f\vec{l} > g\vec{m}$, $|\vec{m}| \le \operatorname{ar}(\Theta_g)$;*
(d) *every rule $f\vec{l} \to r$ is equipped with an environment $\Delta_{f\vec{l}\to r}$ such that, if $\Theta_f = \forall\vec{x} : \vec{T}, U$ and $\pi = [\vec{x} \mapsto \vec{l}]$, then $\Delta_{f\vec{l}\to r} \vdash_{f\vec{l}} r : U\pi$, where $\vdash_{f\vec{l}}$ is the restriction of $\vdash$ defined in Fig. 2.*

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

The relation $\vdash_{f\vec{l}}$ corresponds to the notion of computability closure in [5], with the ordering on function calls replaced by the dependency pair relation. It is similar to $\vdash$ except that it uses the variant of (conv) and (app) used in the proof of the adequacy lemma; (fun) is split in the rules (const) for undefined symbols and (dp) for dependency pairs whose left-hand side is $f\vec{l}$; every type occurring in an object term or every type of a function symbol occurring in a term is required to be typable by using symbols smaller than f only.

The environment $\Delta_{f\vec{l}\to r}$ can be inferred by Dedukti when one restricts rule left hand-sides to some well-behaved class of terms like algebraic terms or Miller patterns (in λProlog).

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

Finally, we need matching to be compatible with computability, that is, if $f\vec{l} \to r \in \mathcal{R}$ and $\vec{l}\sigma$ are computable, then σ is computable, a condition called accessibility in [5]:

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

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

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

$$\text{(app')}\ \frac{\Gamma \vdash_{f\vec{l}} t : \forall x : A, B \quad \Gamma \vdash_{f\vec{l}} a : A \quad \Gamma \vdash_{\prec f} \forall x : A, B : s}{\Gamma \vdash_{f\vec{l}} ta : B[x \mapsto a]}$$

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

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

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

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

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

Figure 2 Restricted type systems $\vdash_{f\vec{l}}$ and $\vdash_{\prec f}$.

▶ **Definition 9** (Accessible system). *A well-structured system $\mathcal{R}$ is accessible if, for all substitutions σ and rules $f\vec{l} \to r$ with $\Theta_f = \forall \vec{x} : \vec{T}, U$ and $|\vec{x}| = |\vec{l}|$, we have $\sigma \models \Delta_{f\vec{l}\to r}$ whenever $\vdash \Theta_f : s_f$, $\Theta_f \in [\![s_f]\!]$ and $[\vec{x} \mapsto \vec{l}]\sigma \models \vec{x} : \vec{T}$.*

This property is not always satisfied because the subterm relation does not preserve computability in general. 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 c is an undefined function symbol of type $\Theta_c = (C \Rightarrow C) \Rightarrow C$, then $c\omega \in [\![C]\!]$ but $\omega \notin [\![C \Rightarrow C]\!]$.

We can now state the main lemma:

▶ **Lemma 10.** *Θ is adequate if $\tilde{>}$ terminates and $\mathcal{R}$ is well-structured and accessible.*

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

- $u = f\vec{v}$ with $\vec{t} \to_{prod} \vec{v}$. Then, we can conclude by the first induction hypothesis.
- There are $fl_1 \ldots l_k \to r \in \mathcal{R}$ 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 $f\vec{t} \in \mathbb{C}$, we have $\pi\sigma \models \Gamma_f$. Since $\mathcal{R}$ is accessible, we get that $\sigma \models \Delta_{f\vec{l}\to r}$. By condition (d), we have $\Delta_{f\vec{l}\to r} \vdash_{f\vec{l}} r : V\pi$ where $V = \forall x_{k+1} : T_{k+1}, \ldots \forall x_n : T_n, U$.
  Now, we prove that, for all Γ, t and T, if $\Gamma \vdash_{f\vec{l}} t : T$ ($\Gamma \vdash_{\prec f} t : T$ resp.) and $\sigma \models \Gamma$, then $t\sigma \in [\![T]\!]_\sigma$, by a third induction on the structure of the derivation of $\Gamma \vdash_{f\vec{l}} t : T$ ($\Gamma \vdash_{\prec f} t : T$ resp.), as in the proof of Lemma 5 except for (fun) replaced by ($\mathrm{fun}_{\prec f}$) in one case, and (const) and (dp) in the other case.
  ($\mathrm{fun}_{\prec f}$) We have $g \in [\![\Theta_g]\!]$ by the first 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 the third induction hypothesis, $y_i\gamma\sigma \in [\![U_i\gamma]\!]_\sigma$. By Lemma 4e, $[\![U_i\gamma]\!]_\sigma = [\![U_i]\!]_{\gamma\sigma}$. So, $\gamma\sigma \models \Sigma$ and $g\vec{y}\gamma\sigma \in \mathbb{C}$. Now, by condition (c), $g\vec{y}\gamma\sigma \tilde{<} f\vec{l}\sigma$ since $g\vec{y}\gamma < f\vec{l}$. Therefore, by the second induction hypothesis, $g\vec{y}\gamma\sigma \in [\![V\gamma]\!]_\sigma$.

So, $r\sigma \in [\![V\pi]\!]_\sigma$ and, by Lemma 4d, $u \in [\![U]\!]_{[x_n\mapsto t_n,..,x_{k+1}\mapsto t_{k+1},\pi\sigma]} = [\![U]\!]_\theta$. ◀

Note that the proof still works if one replaces the relation $\succeq$ of Definition 8 by any well-founded quasi-order such that $f \succeq g$ whenever $f\vec{l} > g\vec{m}$. The quasi-order of Definition 8, defined syntactically, relieves the user of the burden of providing 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.

By combining the previous lemma and the Adequacy lemma (the identity substitution is computable), we get the main result of the paper:

▶ **Theorem 11.** *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, $\mathcal{R}$ is well-structured and accessible, and $\tilde{>}$ terminates.*

For the sake of completeness, we are now going to give sufficient conditions for accessibility and termination of $\tilde{>}$ to hold, but one could imagine many other criteria.

5 Checking accessibility

In this section, we give a simple condition to ensure accessibility and some hints on how to modify the interpretation when this condition is not satisfied.

As seen with the definition of accessibility, the main problem is to deal with subterms of higher-order type. 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 [25], and satisfied by Example 1.

▶ **Definition 12** (PFP systems). *A well-structured $\mathcal{R}$ is PFP if, for all $f\vec{l} \to r \in \mathcal{R}$ with $\Theta_f = \forall\vec{x}:\vec{T}, U$ and $|\vec{x}| = |\vec{l}|$, $\vec{l} \notin \mathbb{K} \cup \{\text{KIND}\}$ and, for all $y : T \in \Delta_{f\vec{l}\to r}$, there is i such that $y = l_i$ and $T = T_i[\vec{x} \mapsto \vec{l}]$, or else $y \in \mathrm{FV}(l_i)$ and $T = D\vec{t}$ with D undefined and $|\vec{t}| = \mathrm{ar}(D)$.*

▶ **Lemma 13.** *PFP systems are accessible.*

Proof. Let $f\vec{l} \to r$ be a PFP rule with $\Theta_f = \forall\Gamma, U$, $\Gamma = \vec{x} : \vec{T}$, $\pi = [\vec{x} \mapsto \vec{l}]$. Following Definition 9, assume that $\vdash \Theta_f : s_f$, $\Theta_f \in \mathcal{D}$ and $\pi\sigma \models \Gamma$. We have to prove that, for all $(y : T) \in \Delta_{f\vec{l}\to r}$, $y\sigma \in [\![T]\!]_\sigma$.

- Suppose $y = l_i$ and $T = T_i\pi$. Then, $y\sigma = l_i\sigma \in [\![T_i]\!]_{\pi\sigma}$. Since $\vdash \Theta_f : s_f$, $T_i \notin \mathbb{K} \cup \{\text{KIND}\}$. 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 $T_i \notin \mathbb{K} \cup \{\text{KIND}\}$ and $\vec{l} \notin \mathbb{K} \cup \{\text{KIND}\}$, $T_i\pi \notin \mathbb{K} \cup \{\text{KIND}\}$. Since $T_i\pi\sigma \in \mathcal{D}$, $[\![T_i\pi]\!]_\sigma = \mathcal{I}(T_i\pi\sigma)$. Thus, $y\sigma \in [\![T]\!]_\sigma$.
- Suppose $y \in \mathrm{FV}(l_i)$ and T is of the form $C\vec{t}$ with $|\vec{t}| = \mathrm{ar}(C)$. Then, $[\![T]\!]_\sigma = \mathrm{SN}$ and $y\sigma \in \mathrm{SN}$ since $l_i\sigma \in [\![T_i]\!]_\sigma \subseteq \mathrm{SN}$. ◀

But many accessible systems are not PFP. They can be proved accessible by changing the interpretation of type constants (a complete development is left for future work).

▶ **Example 14** (Recursor on Brouwer ordinals).

```
symbol Ord: TYPE
 symbol zero: Ord   symbol suc: Ord⇒Ord    symbol lim: (Nat⇒Ord)⇒Ord

symbol ordrec: A⇒(Ord⇒A⇒A)⇒((Nat⇒Ord)⇒(Nat⇒A)⇒A)⇒Ord⇒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))
```

The above example is not PFP because `f:Nat⇒Ord` is not argument of `ordrec`. Yet, it is accessible if one takes for $[\![\texttt{Ord}]\!]$ the least fixpoint of the monotone function $F(S) = \{t \in \mathrm{SN} \mid \text{if } t \to^* \texttt{lim}\, f \text{ then } f \in [\![\texttt{Nat}]\!] \Rightarrow S, \text{ and if } t \to^* \texttt{suc}\, u \text{ then } u \in S\}$ [5].

Similarly, the following encoding of the simply-typed λ-calculus is not PFP but can be proved accessible by taking

$$[\![\texttt{T}\ c]\!] = \text{if } c\!\downarrow = \texttt{arrow}\, a\, b \text{ then } \{t \in \mathrm{SN} \mid \text{if } t \to^* \texttt{lam} f \text{ then } f \in [\![\texttt{T}\ a]\!] \Rightarrow [\![\texttt{T}\ b]\!]\} \text{ else } \mathrm{SN}$$

▶ **Example 15** (Simply-typed λ-calculus).

```
symbol Sort : TYPE                symbol arrow : Sort ⇒ Sort ⇒ Sort

symbol T : Sort ⇒ TYPE
  symbol lam : ∀ a b, (T a ⇒ T b) ⇒ T (arrow a b)
  symbol app : ∀ a b, T (arrow a b) ⇒ T a ⇒ T b
  rule app a b (lam _ _ f) x → f x
```

6 Size-change termination

In this section, we give a sufficient condition for $\tilde{>}$ to terminate. For first-order rewriting, many techniques have been developed for that purpose. To cite just a few, see for instance [17, 14]. 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.

As an example, following Wahlstedt [37], we are going to use Lee, Jones and Ben-Amram's size-change termination criterion (SCT) [26]. It consists in following arguments along function calls and checking that, in every potential loop, one of them decreases. First introduced for first-order functional languages, it has then been extended to many other settings: untyped λ-calculus [21], a subset of OCaml [32], Martin-Löf's type theory [37], System F [27].

We first recall Hyvernat and Raffalli's matrix-based presentation of SCT [20]:

▶ **Definition 16** (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 $(a_{i,j})_{i \le \mathrm{ar}(\Theta_f), j \le \mathrm{ar}(\Theta_g)}$ 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.

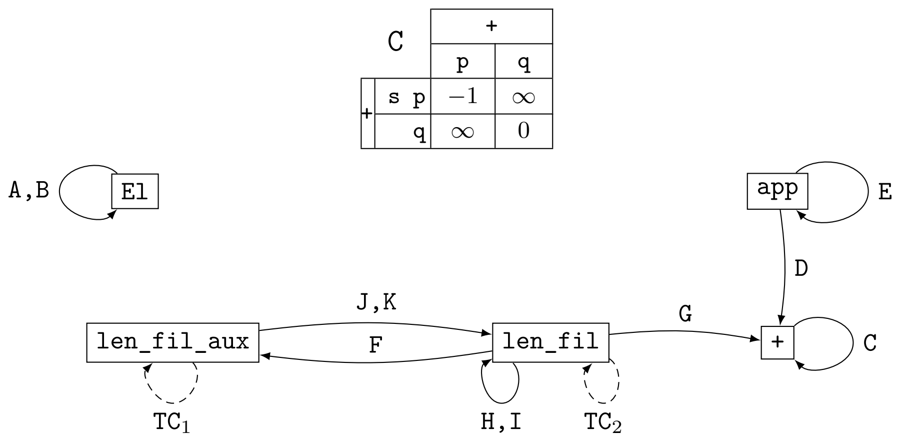

Figure 3 Matrix of dependency pair `C` and call graph of the dependency pairs of Example 7.

We add lines and columns of ∞'s in matrices associated to dependency pairs containing partially applied symbols (cases $i > p$ or $j > q$) because missing arguments cannot be compared with any other argument since they are arbitrary.

The matrix associated to the dependency pair `C: (s p) + q > p + q` and the call graph associated to the dependency pairs of Example 7 are depicted in Figure 3. The full list of matrices and the extensive call graph of Example 1 can be found in Appendix B.

▶ **Lemma 17.** *$\tilde{>}$ terminates if $\mathbb{F}$ is finite and $\mathcal{R}$ is SCT.*

Proof. Suppose that there is an infinite sequence $\chi = f_1\vec{t}_1 \tilde{>} f_2\vec{t}_2 \tilde{>} \dots$ 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 $g\vec{u}_1, g\vec{u}_2, \dots$ 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 $\vec{u}_i$ and $\vec{u}_{i+j}$. By Ramsey's theorem, there is an infinite sequence (ϕ_i) and a matrix M such that M corresponds to all the transitions $g\vec{u}_{\phi_i}, g\vec{u}_{\phi_j}$ with $i \neq j$. M is idempotent, indeed $g\vec{u}_{\phi_i}, g\vec{u}_{\phi_{i+2}}$ 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. ◀

By combining all the previous results, we get:

▶ **Theorem 18.** *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.*

The rewriting system of Example 1 verifies all these conditions (proof in the appendix).

7 Implementation and comparison with other criteria and tools

We implemented our criterion in a tool called SizeChangeTool [12]. As far as we know, there are no other termination checker for $\lambda\Pi/\mathcal{R}$.

If we restrict ourselves to simply-typed rewriting systems, then we can compare it with the termination checkers participating in the category "higher-order rewriting union beta" of the termination competition: Wanda uses dependency pairs, polynomial interpretations, HORPO and many transformation techniques [24]; SOL uses the General Schema [6] and other techniques. As these tools implement various techniques and SizeChangeTool only one, it is difficult to compete with them. Still, there are examples that are solved by SizeChangeTool and not by one of the other tools, demonstrating that these tools would benefit from implementing our new technique. For instance, the problem `Hamana_Kikuchi_18/h17` is proved terminating by SizeChangeTool but not by Wanda because of the rule:

```
rule map f (map g l) → map (comp f g) l
```

And the problem `Kop13/kop12thesis_ex7.23` is proved terminating by SizeChangeTool but not by Sol because of the rules:[3]

```
rule f h x (s y) → g (c x (h y)) y    rule g x y → f (λ_,s 0) x y
```

One could also imagine to translate a termination problem in $\lambda\Pi/\mathcal{R}$ into a simply-typed termination problem. Indeed, the termination of $\lambda\Pi$ alone (without rewriting) can be reduced to the termination of the simply-typed λ-calculus [16]. This has been extended to $\lambda\Pi/\mathcal{R}$ when there are no type-level rewrite rules like the ones defining `El` in Example 1 [22]. However, this translation does not preserve termination as shown by the Example 15 which is not terminating if all the types $\mathbb{T}x$ are mapped to the same type constant.

In [30], Roux also uses dependency pairs for the termination of simply-typed higher-order rewriting 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 [18]. Then, for proving the absence of infinite chains, he uses simple projections [17], which can be seen as a particular case of SCT where strictly decreasing arguments are fixed (SCT can also handle permutations in arguments).

Finally, if we restrict ourselves to orthogonal systems, it is also possible to compare our technique to the ones implemented in the proof assistants Coq and Agda. Coq essentially implements a higher-order version of primitive recursion. Agda on the other hand uses SCT.

Because Example 1 uses matching on defined symbols, it is not orthogonal and can be written neither in Coq nor in Agda. Agda recently added the possibility of adding rewrite rules but this feature is highly experimental and comes with no guaranty. In particular, Agda termination checker does not handle rewriting rules.

Coq cannot handle inductive-recursive definitions [11] nor function definitions with permuted arguments in function calls while it is no problem for Agda and us.

8 Conclusion and future work

We proved a general modularity result extending Arts and Giesl's theorem that a rewriting relation terminates if there are no infinite sequences of dependency pairs [2] from first-order rewriting to dependently-typed higher-order rewriting. Then, following [37], we showed how to use Lee, Jones and Ben-Amram's size-change termination criterion to prove the absence of such infinite sequences [26].

This extends Wahlstedt's work [37] from weak to strong normalization, and from orthogonal to locally confluent rewriting systems. This extends the first author's work [5] 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 [25] from simply-typed λ-calculus to dependent types modulo rewriting.

[3] We renamed the function symbols for the sake of readability.

To get this result, we assumed local confluence. However, one often uses termination to check (local) confluence. Fortunately, there are 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) [35], as it is the case in functional programming languages. A more general one is when critical pairs are "development-closed" [36].

This work can be extended in various directions.

First, our tool is currently 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 like in Example 14, we need to define a more complex interpretation of types, following [5].

Second, to handle recursive calls in such systems, 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 handling Example 14 is the "structural ordering" of Coq and Agda [9, 6]. Relations other than subterm have already been considered in SCT but in a first-order setting only [34].

But we may want to go further because the structural ordering is not enough to handle the following system which is not accepted by Agda:

▶ **Example 19** (Division). m/n computes $\lceil \frac{m}{n} \rceil$.

```
symbol minus: Nat⇒Nat⇒Nat              set infix 1 "-" ≔ minus
  rule 0 - n → 0      rule m - 0 → m      rule (s m) - (s n) → m - n
symbol div: Nat⇒Nat⇒Nat                set infix 1 "/" ≔ div
  rule 0 / (s n) → 0   rule (s m) / (s n) → s ((m - n) / (s n))
```

A solution to handle this system is to use arguments filterings (remove the second argument of -) or simple projections [17]. Another one is to extend the type system with size annotations as in Agda and compute the SCT matrices by comparing the size of terms instead of their structure [1, 7]. In our example, the size of `m - n` is smaller than or equal to the size of `m`. One can deduce this by using user annotations like in Agda, or by using heuristics [8].

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 SCT to handle such systems [19].

References

1 A. Abel. MiniAgda: integrating sized and dependent types. In *Proceedings of the Workshop on Partiality and Recursion in Interactive Theorem Provers*, Electronic Proceedings in Theoretical Computer Science 43, 2010. doi:10.4204/EPTCS.43.2.

2 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000. doi:10.1016/S0304-3975(99)00207-8.

3 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.

4 F. Blanqui. *Théorie des types et récriture*. PhD thesis, Université Paris-Sud, France, 2001. 144 pages. URL: http://tel.archives-ouvertes.fr/tel-00105522.

5 F. Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005. doi:10.1017/S0960129504004426.

6 F. Blanqui. Termination of rewrite relations on λ-terms based on Girard's notion of reducibility. *Theoretical Computer Science*, 611:50–86, 2016. doi:10.1016/j.tcs.2015.07.045.

7 F. Blanqui. Size-based termination of higher-order rewriting. *Journal of Functional Programming*, 28(e11), 2018. 75 pages. doi:10.1017/S0956796818000072.

8 W. N. Chin and S. C. Khoo. Calculating sized types. *Journal of Higher-Order and Symbolic Computation*, 14(2-3):261–300, 2001. doi:10.1023/A:1012996816178.

9 T. Coquand. Pattern matching with dependent types. In *Proceedings of the International Workshop on Types for Proofs and Programs*, 1992. URL: http://www.lfcs.inf.ed.ac.uk/research/types-bra/proc/proc92.ps.gz.

10 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. URL: 10.1007/978-3-540-73228-0_9.

11 P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549, 2000. URL: http://www.jstor.org/stable/2586554.

12 G. Genestier. SizeChangeTool, 2018. URL: https://github.com/Deducteam/SizeChangeTool.

13 J. Giesl, R. Thiemann, and P. Schneider-Kamp. 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. doi:10.1007/978-3-540-32275-7_21.

14 J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006. doi:10.1007/s10817-006-9057-7.

15 J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and types*. Cambridge University Press, 1988. URL: http://www.paultaylor.eu/stable/prot.pdf.

16 R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. doi:10.1145/138027.138060.

17 N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool: techniques and features. *Information and Computation*, 205(4):474–511, 2007. doi:10.1016/j.ic.2006.08.010.

18 J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23th ACM Symposium on Principles of Programming Languages*, 1996. doi:10.1145/237721.240882.

19 P. Hyvernat. The size-change termination principle for constructor based languages. *Logical Methods in Computer Science*, 10(1):1–30, 2014. doi:10.2168/LMCS-10(1:11)2014.

20 P. Hyvernat and C. Raffalli. Improvements on the "size change termination principle" in a functional language. In *11th International Workshop on Termination*, 2010. URL: https://lama.univ-savoie.fr/~raffalli/pdfs/wst.pdf.

21 N. D. Jones and N. Bohr. 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. doi:10.1007/978-3-540-25979-4_1.

22 J.-P. Jouannaud and J. Li. Termination of Dependently Typed Rewrite Rules. In *Proceedings of the 13th International Conference on Typed Lambda Calculi and Applications*, Leibniz International Proceedings in Informatics 38, 2015. doi:10.4230/LIPIcs.TLCA.2015.257.

23 J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993. doi:10.1016/0304-3975(93)90091-7.

24 C. Kop. *Higher order termination*. PhD thesis, VU University Amsterdam, 2012. URL: http://hdl.handle.net/1871/39346.

25 K. Kusakari and M. Sakai. 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. doi:10.1007/s00200-007-0046-9.

26 C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, 2001. doi:10.1145/360204.360210.

27 R. Lepigre and C. Raffalli. Practical subtyping for System F with sized (co-)induction, 2017. arXiv:1604.01990.

28 G. Markowsky. Chain-complete posets and directed sets with applications. *Algebra Universalis*, 6:53–68, 1976.

29 R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192(2):3–29, 1998. doi:10.1016/S0304-3975(97)00143-6.

30 C. Roux. 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. doi:10.4230/LIPIcs.RTA.2011.299.

31 R. Saillard. *Type checking in the Lambda-Pi-calculus modulo: theory and practice*. PhD thesis, Mines ParisTech, France, 2015. URL: https://pastel.archives-ouvertes.fr/tel-01299180.

32 D. Sereni and N. D. Jones. 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. doi:10.1007/11575467_19.

33 TeReSe. *Term rewriting systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

34 R. Thiemann and J. Giesl. The size-change principle and dependency pairs for termination of term rewriting. *Applicable Algebra in Engineering Communication and Computing*, 16(4):229–270, 2005. doi:10.1007/s00200-005-0179-7.

35 V. van Oostrom. *Confluence for abstract and higher-order rewriting*. PhD thesis, Vrije Universiteit Amsterdam, NL, 1994. URL: http://www.phil.uu.nl/~oostrom/publication/ps/phdthesis.ps.

36 V. van Oostrom. Developing developments. *Theoretical Computer Science*, 175(1):159–181, 1997. doi:10.1016/S0304-3975(96)00173-9.

37 D. Wahlstedt. *Dependent type theory with first-order parameterized data types and well-founded recursion*. PhD thesis, Chalmers University of Technology, Sweden, 2007. URL: http://www.cse.chalmers.se/alumni/davidw/wdt_phd_printed_version.pdf.

A Proofs of lemmas on the interpretation

A.1 Definition of the interpretation

▶ **Lemma 20.** *F 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 $a \in J(A)$, $B[x \mapsto a] \in \mathrm{dom}(J)$:

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 $a \in I(A)$, $B[x \mapsto a] \in \mathrm{dom}(I)$. 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 $a \in I(A)$, $B[x \mapsto a] \in \mathrm{dom}(J)$.

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

If $T{\downarrow} = (x:A)B$, then $F(I)(T) = \Pi a \in I(A).\, I(B[x \mapsto a])$ and $F(J)(T) = \Pi a \in J(A).\, J(B[x \mapsto a])$. Since $T \in D(I)$, we have $A \in \mathrm{dom}(I)$ and, for all $a \in I(A)$, $B[x \mapsto a] \in \mathrm{dom}(I)$. Since $\mathrm{dom}(I) \subseteq \mathrm{dom}(J)$, we have $J(A) = I(A)$ and, for all $a \in I(A)$, $J(B[x \mapsto a]) = I(B[x \mapsto a])$. 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 21.** *$\mathcal{D}$ 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 $a \in \mathcal{I}(A)$, $B[x \mapsto a] \in \mathcal{D}$. Therefore, $T' \in \mathcal{D}$.
3. Let T be a neutral term such that $\to(T) \subseteq \mathcal{D}$. Since $\mathcal{D} \subseteq \mathrm{SN}$, $T \in \mathrm{SN}$. Assume now that $T \to^* (x:A)B$. Since T is neutral, there is $U \in \to(T)$ such that $U \to^* (x:A)B$. Therefore, $A \in \mathcal{D}$ and, for all $a \in \mathcal{I}(A)$, $B[x \mapsto a] \in \mathcal{D}$. ◀

▶ **Lemma 22.** *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 $t \to t'$. 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 $ta \to t'a$. Since $Q(a) \in \mathbb{P}$, $t'a \in Q(a)$.
3. Let t be a neutral term such that $\to(t) \subseteq R$. 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 $\to^*(a) \subseteq P$. We now prove that, for all $b \in \to^*(a)$, $tb \in Q(a)$, by induction on $\to$. Since t is neutral, tb is neutral too and it suffices to prove that $\to(tb) \subseteq Q(a)$. Since t is neutral, $\to(tb) = \to(t)b \cup t \to(b)$. By induction hypothesis, $t \to(b) \subseteq Q(a)$. By assumption, $\to(t) \subseteq R$. So, $\to(t)a \subseteq Q(a)$. Since $Q(a) \in \mathbb{P}$, $\to(t)b \subseteq Q(a)$ too. Therefore, $ta \in Q(a)$ and $t \in R$. ◀

▶ **Lemma 23.** *For all $T \in \mathcal{D}$, $\mathcal{I}(T)$ is a computability predicate.*

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

- If $T{\downarrow} = (x:A)B$, then $F(I)(T) = \Pi a \in I(A).\, I(B[x \mapsto a])$. By assumption, $I(A) \in \mathbb{P}$ and, for $a \in I(A)$, $I(B[x \mapsto a]) \in \mathbb{P}$. Hence, by Lemma 22, $F(I)(T) \in \mathbb{P}$.
- Otherwise, $F(I)(T) = \mathrm{SN} \in \mathbb{P}$. ◀

▶ **Lemma 4a.** *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 21. 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 22, $[\![T]\!]_\sigma \in \mathbb{P}$. If $T \notin \mathbb{K} \cup \{\textsc{kind}\}$ and $T\sigma \in \mathcal{D}$, then $[\![T]\!]_\sigma = \mathcal{I}(T\sigma) \in \mathbb{P}$ by Lemma 23. Otherwise, $[\![T]\!]_\sigma = \mathrm{SN} \in \mathbb{P}$. ◀

A.3 Invariance by reduction

We now prove that the interpretation is invariant by reduction.

▶ **Lemma 24.** *If $T \in \mathcal{D}$ and $T \to T'$, 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 $\mathcal{I}(T) = \Pi a \in \mathcal{I}(A). \mathcal{I}(B[x \mapsto a]) = \mathcal{I}(T')$. Otherwise, $\mathcal{I}(T) = \mathrm{SN} = \mathcal{I}(T')$. ◀

▶ **Lemma 4b.** *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, $T \neq \mathrm{KIND}$, and $T' \neq \mathrm{KIND}$. Moreover, $T \in \mathbb{K}$ iff $T' \in \mathbb{K}$ since $\Gamma \vdash T : \mathrm{KIND}$ 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 24. If $T = \mathrm{TYPE}$, 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 $\Gamma \vdash A : \mathrm{TYPE}$, $\Gamma \vdash A' : \mathrm{TYPE}$, $\Gamma, x : A \vdash K : \mathrm{KIND}$ and $\Gamma, x : A' \vdash K' : \mathrm{KIND}$. 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 4c.** *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 4b, $\mathcal{I}(T\sigma) = \mathcal{I}(T\sigma')$.
- Otherwise, $[\![T]\!]_\sigma = \mathrm{SN} = [\![T]\!]_{\sigma'}$. ◀

A.4 Adequacy of the interpretation

▶ **Lemma 4d.** *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 $[\![(x : A)B]\!]_\sigma = \Pi a \in \mathcal{I}(A\sigma{\downarrow}). \mathcal{I}((B\sigma{\downarrow})[x \mapsto a])$.

Since $(x : A)B$ is typable, A is of type TYPE and $A \notin \mathbb{K} \cup \{\mathrm{KIND}\}$. Hence, $[\![A]\!]_\sigma = \mathcal{I}(A\sigma)$ and, by Lemma 24, $\mathcal{I}(A\sigma) = \mathcal{I}(A\sigma{\downarrow})$.

Since $(x : A)B$ is typable and not a kind, B is of type TYPE and $B \notin \mathbb{K} \cup \{\mathrm{KIND}\}$. Hence, $[\![B]\!]_{[x \mapsto a, \sigma]} = \mathcal{I}(B[x \mapsto a, \sigma])$. Since $x \notin \mathrm{dom}(\sigma) \cup \mathrm{FV}(\sigma)$, $B[x \mapsto a, \sigma] = (B\sigma)[x \mapsto a]$. Hence, $[\![B]\!]_{[x \mapsto a, \sigma]} = \mathcal{I}((B\sigma)[x \mapsto a])$ and, by Lemma 24, $\mathcal{I}((B\sigma)[x \mapsto a]) = \mathcal{I}((B\sigma{\downarrow})[x \mapsto a])$.

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

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

▶ **Lemma 4e.** *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 $s = \text{TYPE}$, then $U, U\gamma \notin \mathbb{K} \cup \{\text{KIND}\}$ and $[\![U\gamma]\!]_\sigma = \mathcal{I}(U\gamma\sigma) = [\![U]\!]_{\gamma\sigma}$ since $U\gamma\sigma \in \mathcal{D}$.
- Otherwise, $s = \text{KIND}$ and $U \in \mathbb{K}$.
  - If $U = \text{TYPE}$, then $[\![U\gamma]\!]_\sigma = \mathcal{D} = [\![U]\!]_{\gamma\sigma}$.
  - Otherwise, $U = (x : A)K$ and, by Lemma 4d, $[\![U\gamma]\!]_\sigma = \Pi a \in [\![A\gamma]\!]_\sigma. [\![K\gamma]\!]_{[x\mapsto a,\sigma]}$ 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 $x \notin \text{dom}(\gamma) \cup \text{FV}(\gamma)$. So, $[\![K]\!]_{\gamma[x\mapsto a,\sigma]} = [\![K]\!]_{[x\mapsto a,\gamma\sigma]}$. ◀

▶ **Lemma 4f.** *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 $A \in \text{SN}$ and, for all $a \in P$, $b[x \mapsto a] \in Q(a)$.*

Proof. Let $a_0 \in P$. Since $P \in \mathbb{P}$, we have $a_0 \in \text{SN}$ and $x \in P$. Since $Q(x) \in \mathbb{P}$ and $b = b[x \mapsto x] \in Q(x)$, we have $b \in \text{SN}$. 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 $(\to, \to, \to)_{\forall}$:,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, $(\lambda x : A, b)a \to b[x \mapsto a] \in Q(a)$ by assumption. Since $a_0 \to^* a$ and $Q(a') \subseteq Q(a)$ whenever $a \to a'$, we have $b[x \mapsto a] \in Q(a_0)$. ◀

B Termination proof of Example 1

Here is the comprehensive list of dependency pairs in the example:

```
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
```

The whole callgraph is depicted below. The letter associated to each matrix corresponds to the dependency pair presented above and in example 7, except for TC 's which comes from the computation of the transitive closure and labels dotted edges.

$$\begin{tikzcd}
\boxed{\texttt{El}} \arrow[loop left, "\texttt{A,B}"] & & \boxed{\texttt{fil\_aux}} \arrow[loop above, dashed, "\texttt{TC}_4"] \arrow[rr, bend left, "\texttt{S,T}"] \arrow[drr, "\texttt{R}"] & & \boxed{\texttt{fil}} \arrow[loop above, "\texttt{O,Q}"] \arrow[loop right, dashed, "\texttt{TC}_3"] \arrow[ll, bend left, "\texttt{L}"] \arrow[r, "\texttt{M}"] \arrow[dl, "\texttt{N,P}"] & \boxed{\texttt{app}} \arrow[loop right, "\texttt{E}"] \arrow[d, "\texttt{D}"] \\
& \boxed{\texttt{len\_fil\_aux}} \arrow[loop below, dashed, "\texttt{TC}_1"] \arrow[rr, bend left, "\texttt{J,K}"] & & \boxed{\texttt{len\_fil}} \arrow[loop below, "\texttt{H,I}"] \arrow[loop right, dashed, "\texttt{TC}_2"] \arrow[ll, bend left, "\texttt{F}"] \arrow[rr, "\texttt{G}"] & & \boxed{\texttt{+}} \arrow[loop right, "\texttt{C}"]
\end{tikzcd}$$

The argument `a` is omitted everywhere on the matrices presented below:

$$\texttt{A,B}=\begin{pmatrix} -1 \end{pmatrix},\ \texttt{C}=\begin{pmatrix} -1 & \infty \\ \infty & 0 \end{pmatrix},\ \texttt{D}=\begin{pmatrix} \infty & \infty \\ -1 & \infty \\ \infty & 0 \\ \infty & \infty \end{pmatrix},\ \texttt{E}=\begin{pmatrix} \infty & \infty & \infty & \infty \\ -1 & -1 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix},\ \texttt{F}=\begin{pmatrix} \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & -1 & -1 \end{pmatrix},\ \texttt{J}=\texttt{K}=\begin{pmatrix} \infty & \infty & \infty \\ 0 & \infty & \infty \\ \infty & 0 & \infty \\ \infty & \infty & 0 \end{pmatrix},$$

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

$$\texttt{R}=\texttt{S}=\texttt{T}=\begin{pmatrix} \infty & \infty & \infty \\ 0 & \infty & \infty \\ \infty & \infty & \infty \\ \infty & 0 & \infty \\ \infty & \infty & 0 \end{pmatrix}.$$

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

$$\texttt{TC}_1=\texttt{J}\times\texttt{F}=\begin{pmatrix} \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & -1 & -1 \end{pmatrix},\ \texttt{TC}_4=\texttt{S}\times\texttt{L}=\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & -1 & -1 & -1 \end{pmatrix},$$

$$\texttt{TC}_3=\texttt{L}\times\texttt{S}=\texttt{TC}_2=\texttt{F}\times\texttt{J}=\begin{pmatrix} 0 & \infty & \infty \\ \infty & \infty & \infty \\ \infty & -1 & -1 \end{pmatrix}=\texttt{O}=\texttt{H}.$$

It would be useless to compute matrices labeling edges which are not in a strongly connected component of the call-graph (like `S`×`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.

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

The precedence infered for this example is the smallest containing:

- comparisons linked to the typing of symbols:

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

$$\begin{tikzcd}
 & & & & & & \boxed{\texttt{fil,fil\_aux}} & & & & \\
 & & & \boxed{\texttt{app}} \arrow[urrr] & & & & \boxed{\texttt{len\_fil,len\_fil\_aux}} \arrow[ul] & & & \\
 & & \boxed{\texttt{cons}} \arrow[ur] & & \boxed{\texttt{nil}} \arrow[uurr] & & & \boxed{\texttt{+}} \arrow[ullll] \arrow[u] & & & \\
\boxed{\texttt{arrow}} & \boxed{\texttt{El}} \arrow[ur] \arrow[uurrrrrr, bend left] & & \boxed{\mathbb{L}} \arrow[ul] \arrow[ur] \arrow[uurrrr] & & \boxed{\texttt{0}} \arrow[ul] \arrow[uurr] & & \boxed{\texttt{s}} \arrow[ulllll] \arrow[u] & \boxed{\texttt{true}} & & \boxed{\texttt{false}} \\
 & \boxed{\texttt{Set}} \arrow[ul] \arrow[u] \arrow[urr] & & & & & \boxed{\mathbb{N}} \arrow[ulll] \arrow[ul] \arrow[ur] & & & \boxed{\mathbb{B}} \arrow[ul] \arrow[ur] \arrow[uuull, bend right] &
\end{tikzcd}$$