



HAL
open science

Dependency Pairs Termination in Dependent Type Theory Modulo Rewriting

Frédéric Blanqui, Guillaume Genestier, Olivier Hermant

► **To cite this version:**

Frédéric Blanqui, Guillaume Genestier, Olivier Hermant. Dependency Pairs Termination in Dependent Type Theory Modulo Rewriting. FSCD, Jun 2019, Dortmund, Germany. hal-01943941v1

HAL Id: hal-01943941

<https://inria.hal.science/hal-01943941v1>

Submitted on 23 Apr 2019 (v1), last revised 15 Oct 2019 (v4)

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

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

1 Dependency Pairs Termination in Dependent 2 Type Theory Modulo Rewriting

3 Frédéric Blanqui^{1,2}
4 Guillaume Genestier^{2,3}
5 Olivier Hermant³

6 ¹ INRIA

7 ² LSV, ENS Paris-Saclay, CNRS, Université Paris-Saclay

8 ³ MINES ParisTech, PSL University

9 — Abstract —

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

15 **2012 ACM Subject Classification** Theory of computation → Equational logic and rewriting; Theory
16 of computation → Type theory

17 **Keywords and phrases** Termination, Higher-Order Rewriting, Dependent Types, Dependency Pairs

18 **Digital Object Identifier** 10.4230/LIPIcs...

19 **1** Introduction

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

24 This paper introduces a new termination criterion for a large class of programs whose
25 operational semantics can be described by higher-order rewriting rules [33] typable in the
26 $\lambda\Pi$ -calculus modulo rewriting ($\lambda\Pi/\mathcal{R}$ for short). $\lambda\Pi/\mathcal{R}$ is a system of dependent types where
27 types are identified modulo the β -reduction of λ -calculus and a set \mathcal{R} of rewriting rules given
28 by the user to define not only functions but also types. It extends Barendregt's Pure Type
29 System (PTS) λP [3], the logical framework LF [16] and Martin-Löf's type theory. It can
30 encode any functional PTS like System F or the Calculus of Constructions [10].

31 Dependent types, introduced by de Bruijn in AUTOMATH, subsume generalized algebraic
32 data types (GADT) used in some functional programming languages. They are at the core of
33 many proof assistants and programming languages: COQ, TWELF, AGDA, LEAN, IDRIS, ...

34 Our criterion has been implemented in DEDUKTI, a type-checker for $\lambda\Pi/\mathcal{R}$ that we will
35 use in our examples. The code is available in [12] and could be easily adapted to a subset of
36 other languages like AGDA. As far as we know, this tool is the first one to automatically
37 check termination in $\lambda\Pi/\mathcal{R}$, which includes both higher-order rewriting and dependent types.

38 This criterion is based on dependency pairs, an important concept in the termination
39 of first-order term rewriting systems. It generalizes the notion of recursive call in first-
40 order functional programs to rewriting. Namely, the dependency pairs of a rewriting rule
41 $f(l_1, \dots, l_p) \rightarrow r$ are the pairs $(f(l_1, \dots, l_p), g(m_1, \dots, m_q))$ such that $g(m_1, \dots, m_q)$ is a
42 subterm of r and g is a function symbol defined by some rewriting rules. Dependency pairs
43 have been introduced by Arts and Giesl [2] and have evolved into a general framework for
44 termination [13]. It is now at the heart of many state-of-the-art automated termination



© Frédéric Blanqui, Guillaume Genestier and Olivier Hermant;
licensed under Creative Commons License CC-BY

Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

45 provers for first-order rewriting systems and HASKELL, JAVA or C programs.

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

51 In [38], Wahlstedt considered a system slightly less general than $\lambda\Pi/\mathcal{R}$ for which he
 52 provided conditions that imply the weak normalization, that is, the existence of a finite
 53 reduction to normal form. In his system, \mathcal{R} uses matching on constructors only, like in
 54 the languages OCAML or HASKELL. In this case, \mathcal{R} is orthogonal: rules are left-linear (no
 55 variable occurs twice in a left-hand side) and have no critical pairs (no two rule left-hand side
 56 instances overlap). Wahlstedt’s proof proceeds in two modular steps. First, he proves that
 57 typable terms have a normal form if there is no infinite sequence of function calls. Second,
 58 he proves that there is no infinite sequence of function calls if \mathcal{R} satisfies Lee, Jones and
 59 Ben-Amram’s size-change termination criterion (SCT) [26].

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

64 In [5], the first author developed a termination criterion for a calculus slightly more
 65 general than $\lambda\Pi/\mathcal{R}$, based on the notion of computability closure, assuming that type-level
 66 rules are orthogonal. The computability closure of a term $f(l_1, \dots, l_p)$ is a set of terms that
 67 terminate whenever l_1, \dots, l_p terminate. It is defined inductively thanks to deduction rules
 68 preserving this property, using a precedence and a fixed well-founded ordering for dealing
 69 with function calls. Termination can then be enforced by requiring each rule right-hand side
 70 to belong to the computability closure of its corresponding left-hand side.

71 We extend this work as well by replacing that fixed ordering by the dependency pair
 72 relation. In [5], there must be a decrease in every function call. Using dependency pairs
 73 allows one to have non-strict decreases. Then, following Wahlstedt, SCT can be used to
 74 enforce the absence of infinite sequence of dependency pairs. But other criteria have been
 75 developed for this purpose that could be adapted to $\lambda\Pi/\mathcal{R}$.

76 **Outline**

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

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

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

90 Finally, in Section 7 we compare our criterion with other criteria and tools and, in Section
 91 8, we summarize our results and give some hints on possible extensions.

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

93 **2 Terms and types**

94 The set \mathbb{T} of terms of $\lambda\Pi/\mathcal{R}$ is the same as those of Barendregt's λP [3]:

$$95 \quad t \in \mathbb{T} = s \in \mathbb{S} \mid x \in \mathbb{V} \mid f \in \mathbb{F} \mid \forall x : t, t \mid tt \mid \lambda x : t, t$$

96 where $\mathbb{S} = \{\text{TYPE}, \text{KIND}\}$ is the set of sorts¹, \mathbb{V} is an infinite set of variables and \mathbb{F} is a set of
97 function symbols, so that \mathbb{S} , \mathbb{V} and \mathbb{F} are pairwise disjoint.

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

$$101 \quad \rightarrow = \rightarrow_\beta \cup \rightarrow_{\mathcal{R}}$$

102 where \rightarrow_β is the β -reduction of λ -calculus and $\rightarrow_{\mathcal{R}}$ is the smallest relation containing \mathcal{R}
103 and closed by substitution and context: we consider rewriting with syntactic matching only.
104 Following [6], it should however be possible to extend the present results to rewriting with
105 matching modulo $\beta\eta$ or some equational theory. Let SN be the set of terminating terms and,
106 given a term t , let $\rightarrow(t) = \{u \in \mathbb{T} \mid t \rightarrow u\}$ be the set of immediate reducts of t .

107 A typing environment Γ is a (possibly empty) sequence $x_1 : T_1, \dots, x_n : T_n$ of pairs of
108 variables and terms, where the variables are distinct, written $\vec{x} : \vec{T}$ for short. Given an
109 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
110 term T is the integer $n \in \mathbb{N}$ such that $T = \forall x_1 : T_1, \dots, \forall x_n : T_n, U$ and U is not a product.
111 Let \vec{t} denote a possibly empty sequence of terms t_1, \dots, t_n of length $|\vec{t}| = n$, and $\text{FV}(t)$ be
112 the set of free variables of t .

113 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
114 such that $\Theta_f = \forall\Gamma_f, U$ and $|\Gamma_f| = \text{ar}(\Theta_f)$.

115 The application of a substitution σ to a term t is written $t\sigma$. Given a substitution σ ,
116 let $\text{dom}(\sigma) = \{x \mid x\sigma \neq x\}$, $\text{FV}(\sigma) = \bigcup_{x \in \text{dom}(\sigma)} \text{FV}(x\sigma)$ and $[x \mapsto a, \sigma]$ ($[x \mapsto a]$ if σ is the
117 identity) be the substitution $\{(x, a)\} \cup \{(y, b) \in \sigma \mid y \neq x\}$. Given another substitution σ' ,
118 let $\sigma \rightarrow \sigma'$ if there is x such that $x\sigma \rightarrow x\sigma'$ and, for all $y \neq x$, $y\sigma = y\sigma'$.

119 The typing rules of $\lambda\Pi/\mathcal{R}$, in Figure 1, add to those of λP the rule (fun) similar to
120 (var). Moreover, (conv) uses \downarrow instead of \downarrow_β , where $\downarrow = \rightarrow^* \ast \leftarrow$ is the joinability relation
121 and \rightarrow^* the reflexive and transitive closure of \rightarrow . We say that t has type T in Γ if $\Gamma \vdash t : T$
122 is derivable. A substitution σ is well-typed from Δ to Γ , written $\Gamma \vdash \sigma : \Delta$, if, for all
123 $(x : T) \in \Delta$, $\Gamma \vdash x\sigma : T\sigma$ holds.

124 The word ‘‘type’’ is used to denote a term occurring at the right-hand side of a colon in
125 a typing judgment (and we usually use capital letters for types). Hence, KIND is the type
126 of TYPE, Θ_f is the type of f , and s_f is the type of Θ_f . Common data types like natural
127 numbers \mathbb{N} are usually declared in $\lambda\Pi$ as function symbols of type TYPE: $\Theta_{\mathbb{N}} = \text{TYPE}$ and
128 $s_{\mathbb{N}} = \text{KIND}$.

129 The dependent product $\forall x : A, B$ generalizes the arrow type $A \Rightarrow B$ of simply-typed
130 λ -calculus: it is the type of functions taking an argument x of type A and returning a term
131 whose type B may depend on x . If B does not depend on x , we sometimes simply write
132 $A \Rightarrow B$.

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

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

<p>(ax) $\frac{}{\vdash \text{TYPE} : \text{KIND}}$</p> <p>(var) $\frac{\Gamma \vdash A : s \quad x \notin \text{dom}(\Gamma)}{\Gamma, x : A \vdash x : A}$</p> <p>(weak) $\frac{\Gamma \vdash A : s \quad \Gamma \vdash b : B \quad x \notin \text{dom}(\Gamma)}{\Gamma, x : A \vdash b : B}$</p> <p>(prod) $\frac{\Gamma \vdash A : \text{TYPE} \quad \Gamma, x : A \vdash B : s}{\Gamma \vdash (x : A)B : s}$</p>	<p>(abs) $\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (x : A)B : s}{\Gamma \vdash \lambda x : A. b : (x : A)B}$</p> <p>(app) $\frac{\Gamma \vdash t : (x : A)B \quad \Gamma \vdash a : A}{\Gamma \vdash ta : B[x \mapsto a]}$</p> <p>(conv) $\frac{\Gamma \vdash a : A \quad A \downarrow B \quad \Gamma \vdash B : s}{\Gamma \vdash a : B}$</p> <p>(fun) $\frac{\vdash \Theta_f : s_f}{\vdash f : \Theta_f}$</p>
--	--

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

137 ► **Example 1** (Filter function on dependent lists). To illustrate the kind of systems we consider,
 138 we give an extensive example in the new DEDUKTI syntax combining type-level rewriting rules
 139 (El converts datatype codes into DEDUKTI types), dependent types (\mathbb{L} is the polymorphic
 140 type of lists parameterized with their length), higher-order variables (fil is a function
 141 filtering elements out of a list along a boolean function f), and matching on defined function
 142 symbols (fil can match a list defined by concatenation). Note that this example cannot be
 143 represented in COQ or AGDA because of the rules using matching on app. And its termination
 144 can be handled neither by [38] nor by [5] because the system is not orthogonal and has no
 145 strict decrease in every recursive call. It can however be handled by our new termination
 146 criterion and its implementation [12]. For readability, we removed the & which are used to
 147 identify pattern variables in the rewriting rules.

```

148
149 symbol Set: TYPE          symbol arrow: Set => Set => Set
150
151 symbol El: Set => TYPE    rule El (arrow a b) -> El a => El b
152
153 symbol Bool: TYPE        symbol true: Bool          symbol false: Bool
154 symbol Nat: TYPE         symbol zero: Nat           symbol s: Nat => Nat
155
156 symbol plus: Nat => Nat => Nat    set infix 1 "+" := plus
157 rule zero + q -> q              rule (s p) + q -> s (p + q)
158
159 symbol List: Set => Nat => TYPE
160 symbol nil: <math>\forall a, \text{List } a \text{ zero}</math>
161 symbol cons: <math>\forall a, \text{El } a \Rightarrow \forall p, \text{List } a \text{ } p \Rightarrow \text{List } a \text{ (s } p)</math>
162
163 symbol app: <math>\forall a \text{ } p, \text{List } a \text{ } p \Rightarrow \forall q, \text{List } a \text{ } q \Rightarrow \text{List } a \text{ (p+q)}</math>
164 rule app a _ (nil _)          q m -> m
165 rule app a _ (cons _ x p l) q m -> cons a x (p+q) (app a p l q m)
166
167 symbol len_fil: <math>\forall a, (\text{El } a \Rightarrow \text{Bool}) \Rightarrow \forall p, \text{List } a \text{ } p \Rightarrow \text{Nat}</math>
168 symbol len_fil_aux: <math>\text{Bool} \Rightarrow \forall a, (\text{El } a \Rightarrow \text{Bool}) \Rightarrow \forall p, \text{List } a \text{ } p \Rightarrow \text{Nat}</math>
169 rule len_fil a f _ (nil _)    -> zero
170 rule len_fil a f _ (cons _ x p l) -> len_fil_aux (f x) a f p l
    
```

```

171 rule len_fil a f _ (app _ p l q m)
172   → (len_fil a f p l) + (len_fil a f q m)
173 rule len_fil_aux true a f p l → s (len_fil a f p l)
174 rule len_fil_aux false a f p l → len_fil a f p l
175
176 symbol fil: ∀a f p l, List a (len_fil a f p l)
177 symbol fil_aux: ∀b a f, El a ⇒ ∀p l, List a (len_fil_aux b a f p l)
178 rule fil a f _ (nil _) → nil a
179 rule fil a f _ (cons _ x p l) → fil_aux (f x) a f x p l
180 rule fil a f _ (app _ p l q m)
181   → app a (len_fil a f p l) (fil a f p l)
182     (len_fil a f q m) (fil a f q m)
183 rule fil_aux false a f x p l → fil a f p l
184 rule fil_aux true a f x p l
185   → cons a x (len_fil a f p l) (fil a f p l)
186

```

187 **Assumptions:** Throughout the paper, we assume that \rightarrow is locally confluent ($\leftarrow\rightarrow \subseteq \downarrow$)
188 and preserves typing (for all Γ, A, t and u , if $\Gamma \vdash t : A$ and $t \rightarrow u$, then $\Gamma \vdash u : A$).

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

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

194 3 Interpretation of types as reducibility candidates

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

204 So, following Girard [15], to prove the termination of $\rightarrow_\beta \cup \rightarrow_{\mathcal{R}}$, we build a model of our
205 calculus by interpreting types into sets of terminating terms. To this end, we need to find an
206 interpretation $\llbracket \cdot \rrbracket$ having the following properties:

- 207 ■ Because types are identified modulo conversion, we need $\llbracket \cdot \rrbracket$ to be invariant by reduction:
208 if T is typable and $T \rightarrow T'$, then we must have $\llbracket T \rrbracket = \llbracket T' \rrbracket$.
- 209 ■ As usual, to handle β -reduction, we need a product type $\forall x : A, B$ to be interpreted by
210 the set of terms t such that, for all a in the interpretation of A , ta is in the interpretation
211 of $B[x \mapsto a]$, that is, we must have $\llbracket \forall x : A, B \rrbracket = \Pi a \in \llbracket A \rrbracket. \llbracket B[x \mapsto a] \rrbracket$ where $\Pi a \in$
212 $P. Q(a) = \{t \mid \forall a \in P, ta \in Q(a)\}$.

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

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

XX:6 Dependency Pairs Termination in Dependent Type Theory Modulo Rewriting

219 ■ The domain of $F(I)$ is the set $D(I)$ of all the terminating terms T such that, if T reduces
 220 to some product term $\forall x : A, B$ (not necessarily in normal form), then $A \in \text{dom}(I)$ and,
 221 for all $a \in I(A)$, $B[x \mapsto a] \in \text{dom}(I)$.

222 ■ If $T \in D(I)$ and the normal form² of T is not a product, then $F(I)(T) = \text{SN}$.

223 ■ If $T \in D(I)$ and $T \downarrow = \forall x : A, B$, then $F(I)(T) = \Pi a \in I(A). I(B[x \mapsto a])$.

224 We now introduce $\mathcal{D} = D(\mathcal{I})$ and define the interpretation of a term T wrt to a substitution
 225 σ , $\llbracket T \rrbracket_\sigma$ (and simply $\llbracket T \rrbracket$ if σ is the identity), as follows:

226 ■ $\llbracket s \rrbracket_\sigma = \mathcal{D}$ if $s \in \mathbb{S}$,

227 ■ $\llbracket \forall x : A, K \rrbracket_\sigma = \Pi a \in \llbracket A \rrbracket_\sigma. \llbracket K \rrbracket_{[x \mapsto a, \sigma]}$ if $K \in \mathbb{K}$ and $x \notin \text{dom}(\sigma)$,

228 ■ $\llbracket T \rrbracket_\sigma = \mathcal{I}(T\sigma)$ if $T \notin \mathbb{K} \cup \{\text{KIND}\}$ and $T\sigma \in \mathcal{D}$,

229 ■ $\llbracket T \rrbracket_\sigma = \text{SN}$ otherwise.

230 A substitution σ is adequate wrt an environment Γ , $\sigma \models \Gamma$, if, for all $x : A \in \Gamma$, $x\sigma \in \llbracket A \rrbracket_\sigma$.

231 A typing map Θ is adequate if, for all f , $f \in \llbracket \Theta_f \rrbracket$ whenever $\vdash \Theta_f : s_f$ and $\Theta_f \in \llbracket s_f \rrbracket$.

232 Let \mathbb{C} be the set of terms of the form $f\vec{t}$ such that $|\vec{t}| = \text{ar}(\Theta_f)$, $\vdash \Theta_f : s_f$, $\Theta_f \in \llbracket s_f \rrbracket$ and,
 233 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
 234 fully applying some function symbol to computable arguments.)

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

238 ► **Definition 3** (Computability predicates). A term is neutral if it is of the form $(\lambda x : A, t)u\vec{v}$,
 239 $x\vec{v}$ or $f\vec{v}$ with, for every rule $f\vec{l} \rightarrow r \in \mathcal{R}$, $|\vec{l}| \leq |\vec{v}|$.

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

242 Note that neutral terms satisfy the following key property: if t is neutral then, for all u ,
 243 tu is neutral and every reduct of tu is either of the form $t'u$ with t' a reduct of t , or of the
 244 form tu' with u' a reduct of u .

245 One can easily check that SN is a computability predicate.

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

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

249 ► **Lemma 4.** (a) For all terms T and substitutions σ , $\llbracket T \rrbracket_\sigma \in \mathbb{P}$.

250 (b) If T is typable, $T\sigma \in \mathcal{D}$ and $T \rightarrow T'$, then $\llbracket T \rrbracket_\sigma = \llbracket T' \rrbracket_\sigma$.

251 (c) If T is typable, $T\sigma \in \mathcal{D}$ and $\sigma \rightarrow \sigma'$, then $\llbracket T \rrbracket_\sigma = \llbracket T \rrbracket_{\sigma'}$.

252 (d) If $\forall x : A, B$ is typable and $\forall x : A\sigma, B\sigma \in \mathcal{D}$,

253 then $\llbracket \forall x : A, B \rrbracket_\sigma = \Pi a \in \llbracket A \rrbracket_\sigma. \llbracket B \rrbracket_{[x \mapsto a, \sigma]}$.

254 (e) If $\Delta \vdash U : s$, $\Gamma \vdash \gamma : \Delta$ and $U\gamma\sigma \in \mathcal{D}$, then $\llbracket U\gamma \rrbracket_\sigma = \llbracket U \rrbracket_{\gamma\sigma}$.

255 (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 $a \rightarrow a'$. Then,
 256 $\lambda x : A, b \in \Pi a \in P. Q(a)$ if $A \in \text{SN}$ and, for all $a \in P$, $b[x \mapsto a] \in Q(a)$.

257 We can finally prove that our model is adequate, that is, every term of type T belongs to
 258 $\llbracket T \rrbracket$, if the typing map Θ itself is adequate. This reduces the termination of well-typed terms
 259 to the computability of function symbols.

260 ► **Lemma 5** (Adequacy). If Θ is adequate, $\Gamma \vdash t : T$ and $\sigma \models \Gamma$, then $t\sigma \in \llbracket T \rrbracket_\sigma$.

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

261 **Proof.** First note that, if $\Gamma \vdash t : T$, then either $T = \text{KIND}$ or $\Gamma \vdash T : s$ [4, Lemma 28].
 262 Moreover, if $\Gamma \vdash a : A$, $A \downarrow B$ and $\Gamma \vdash B : s$ (the premises of the (conv) rule), then $\Gamma \vdash A : s$
 263 [4, Lemma 42] (because \rightarrow preserves typing). Hence, the relation \vdash is unchanged if one
 264 adds the premise $\Gamma \vdash A : s$ in (conv), giving the rule (conv'). Similarly, we add the premise
 265 $\Gamma \vdash \forall x : A, B : s$ in (app), giving the rule (app'). We now prove the lemma by induction on
 266 $\Gamma \vdash t : T$ using (app') and (conv'):
 267 **(ax)** It is immediate that $\text{TYPE} \in \llbracket \text{KIND} \rrbracket_\sigma = \mathcal{D}$.
 268 **(var)** By assumption on σ .
 269 **(weak)** If $\sigma \models \Gamma, x : A$, then $\sigma \models \Gamma$. So, the result follows by induction hypothesis.
 270 **(prod)** Is $(\forall x : A, B)\sigma$ in $\llbracket s \rrbracket_\sigma = \mathcal{D}$? Wlog we can assume $x \notin \text{dom}(\sigma) \cup \text{FV}(\sigma)$. So,
 271 $(\forall x : A, B)\sigma = \forall x : A\sigma, B\sigma$. By induction hypothesis, $A\sigma \in \llbracket \text{TYPE} \rrbracket_\sigma = \mathcal{D}$. Let now $a \in$
 272 $\mathcal{I}(A\sigma)$ and $\sigma' = [x \mapsto a, \sigma]$. Note that $\mathcal{I}(A\sigma) = \llbracket A \rrbracket_\sigma$. So, $\sigma' \models \Gamma, x : A$ and, by induction
 273 hypothesis, $B\sigma' \in \llbracket s \rrbracket_{\sigma'} = \mathcal{D}$. Since $x \notin \text{dom}(\sigma) \cup \text{FV}(\sigma)$, we have $B\sigma' = (B\sigma)[x \mapsto a]$.
 274 Therefore, $(\forall x : A, B)\sigma \in \llbracket s \rrbracket_\sigma$.
 275 **(abs)** Is $(\lambda x : A, b)\sigma$ in $\llbracket \forall x : A, B \rrbracket_\sigma$? Wlog we can assume that $x \notin \text{dom}(\sigma) \cup \text{FV}(\sigma)$.
 276 So, $(\lambda x : A, b)\sigma = \lambda x : A\sigma, b\sigma$. By Lemma 4d, $\llbracket \forall x : A, B \rrbracket_\sigma = \Pi a \in \llbracket A \rrbracket_\sigma. \llbracket B \rrbracket_{[x \mapsto a, \sigma]}$.
 277 By Lemma 4c, $\llbracket B \rrbracket_{[x \mapsto a, \sigma]}$ is an $\llbracket A \rrbracket_\sigma$ -indexed family of computability predicates such
 278 that $\llbracket B \rrbracket_{[x \mapsto a', \sigma]} = \llbracket B \rrbracket_{[x \mapsto a, \sigma]}$ whenever $a \rightarrow a'$. Hence, by Lemma 4f, $\lambda x : A\sigma, b\sigma \in$
 279 $\llbracket \forall x : A, B \rrbracket_\sigma$ if $A\sigma \in \text{SN}$ and, for all $a \in \llbracket A \rrbracket_\sigma$, $(b\sigma)[x \mapsto a] \in \llbracket B \rrbracket_{\sigma'}$ where $\sigma' = [x \mapsto$
 280 $a, \sigma]$. By induction hypothesis, $(\forall x : A, B)\sigma \in \llbracket s \rrbracket_\sigma = \mathcal{D}$. Since $x \notin \text{dom}(\sigma) \cup \text{FV}(\sigma)$,
 281 $(\forall x : A, B)\sigma = \forall x : A\sigma, B\sigma$ and $(b\sigma)[x \mapsto a] = b\sigma'$. Since $\mathcal{D} \subseteq \text{SN}$, we have $A\sigma \in \text{SN}$.
 282 Moreover, since $\sigma' \models \Gamma, x : A$, we have $b\sigma' \in \llbracket B \rrbracket_{\sigma'}$ by induction hypothesis.
 283 **(app')** Is $(ta)\sigma = (t\sigma)(a\sigma)$ in $\llbracket B[x \mapsto a] \rrbracket_\sigma$? By induction hypothesis, $t\sigma \in \llbracket \forall x : A, B \rrbracket_\sigma$, $a\sigma \in$
 284 $\llbracket A \rrbracket_\sigma$ and $(\forall x : A, B)\sigma \in \llbracket s \rrbracket_\sigma = \mathcal{D}$. By Lemma 4d, $\llbracket \forall x : A, B \rrbracket_\sigma = \Pi \alpha \in \llbracket A \rrbracket_\sigma. \llbracket B \rrbracket_{[x \mapsto \alpha, \sigma]}$.
 285 Hence, $(t\sigma)(a\sigma) \in \llbracket B \rrbracket_{\sigma'}$ where $\sigma' = [x \mapsto a\sigma, \sigma]$. Wlog we can assume $x \notin \text{dom}(\sigma) \cup$
 286 $\text{FV}(\sigma)$. So, $\sigma' = [x \mapsto a]\sigma$. Hence, by Lemma 4e, $\llbracket B \rrbracket_{\sigma'} = \llbracket B[x \mapsto a] \rrbracket_\sigma$.
 287 **(conv')** By induction hypothesis, $a\sigma \in \llbracket A \rrbracket_\sigma$, $A\sigma \in \llbracket s \rrbracket_\sigma = \mathcal{D}$ and $B\sigma \in \llbracket s \rrbracket_\sigma = \mathcal{D}$. By
 288 Lemma 4b, $\llbracket A \rrbracket_\sigma = \llbracket B \rrbracket_\sigma$. So, $a\sigma \in \llbracket B \rrbracket_\sigma$.
 289 **(fun)** By induction hypothesis, $\Theta_f \in \llbracket s_f \rrbracket_\sigma = \mathcal{D}$. Therefore, $f \in \llbracket \Theta_f \rrbracket_\sigma = \llbracket \Theta_f \rrbracket$ since Θ is
 290 adequate. \blacktriangleleft

291 4 Dependency pairs theorem

292 Now, we prove that the adequacy of Θ can be reduced to the absence of infinite sequences of
 293 dependency pairs, as shown by Arts and Giesl for first-order rewriting [2].

294 **► Definition 6** (Dependency pairs). *Let $f\vec{l} > g\vec{m}$ iff there is a rule $f\vec{l} \rightarrow r \in \mathcal{R}$, g is defined*
 295 *and $g\vec{m}$ is a subterm of r such that \vec{m} are all the arguments to which g is applied. The*
 296 *relation $>$ is the set of dependency pairs.*

297 *Let $\tilde{>} = \rightarrow_{\text{arg}}^* >_s$ be the relation on the set \mathbb{C} (Def. 2), where $f\vec{t} \rightarrow_{\text{arg}} f\vec{u}$ iff $\vec{t} \rightarrow_{\text{prod}} \vec{u}$*
 298 *(reduction in one argument), and $>_s$ is the closure by substitution and left-application of $>$:*
 299 *$ft_1 \dots t_p \tilde{>} gu_1 \dots u_q$ iff there are a dependency pair $fl_1 \dots l_i > gm_1 \dots m_j$ with $i \leq p$ and*
 300 *$j \leq q$ and a substitution σ such that, for all $k \leq i$, $t_k \rightarrow^* l_k\sigma$ and, for all $k \leq j$, $m_k\sigma = u_k$.*

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

304 $\text{app } x \ y \rightarrow x \ y$ $f \ x \ y \rightarrow \text{app } (f \ x) \ y$
 305

XX:8 Dependency Pairs Termination in Dependent Type Theory Modulo Rewriting

whose dependency pairs are $f\ x\ y > \text{app}\ (f\ x)\ y$ 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):

```

312 A:          El (arrow a b) > El a
313 B:          El (arrow a b) > El b
314 C:          (s p) + q > p + q
315 D:  app a _ (cons _ x p l) q m > p + q
316 E:  app a _ (cons _ x p l) q m > app a p l q m
317 F: len_fil a f _ (cons _ x p l) > len_fil_aux (f x) a f p l
318 G: len_fil a f _ (app _ p l q m) >
319                                     (len_fil a f p l) + (len_fil a f q m)
320 H: len_fil a f _ (app _ p l q m) > len_fil a f p l
321 I: len_fil a f _ (app _ p l q m) > len_fil a f q m
322 J:  len_fil_aux true  a f p l > len_fil a f p l
323 K:  len_fil_aux false a f p l > len_fil a f p l

```

In [2], a sequence of dependency pairs interleaved with \rightarrow_{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 \succ terminates. Moreover, in a first-order term algebra, \succ terminates if and only if, for all f and \vec{t} , $f\vec{t}$ terminates wrt \succ whenever \vec{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 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 $f\vec{l} \rightarrow r \in \mathcal{R}$ with g (defined or undefined) occurring in r . Then, let $\succ = \succeq \setminus \preceq$ be the strict part of \succeq . A rewriting system \mathcal{R} is well-structured if:*

- (a) \succ is well-founded;
- (b) for every rule $f\vec{l} \rightarrow r$, $|\vec{l}| \leq \text{ar}(\Theta_f)$;
- (c) for every dependency pair $f\vec{l} > g\vec{m}$, $|\vec{m}| \leq \text{ar}(\Theta_g)$;
- (d) every rule $f\vec{l} \rightarrow r$ is equipped with an environment $\Delta_{f\vec{l} \rightarrow r}$ such that, if $\Theta_f = \forall \vec{x} : \vec{T}, U$ and $\pi = [\vec{x} \mapsto \vec{l}]$, then $\Delta_{f\vec{l} \rightarrow r} \vdash_{f\vec{l}} r : U\pi$, 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 $|\vec{t}| = \text{ar}(\Theta_f)$. Condition (c) ensures that $>$ is included in \succ .

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 $\Delta_{f\vec{l} \rightarrow r}$ 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 $f\vec{l} \rightarrow r \in \mathcal{R}$ and $\vec{l}\sigma$ are computable, then σ is computable, a condition called accessibility in [5]:

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

$$\begin{array}{c}
\text{(ax)} \quad \frac{}{\vdash_{f\vec{l}} \text{TYPE} : \text{KIND}} \quad \text{(weak)} \quad \frac{\Gamma \vdash_{\prec f} A : s \quad \Gamma \vdash_{f\vec{l}} b : B \quad x \notin \text{dom}(\Gamma)}{\Gamma, x : A \vdash_{f\vec{l}} b : B} \\
\text{(var)} \quad \frac{\Gamma \vdash_{\prec f} A : s \quad x \notin \text{dom}(\Gamma)}{\Gamma, x : A \vdash_{f\vec{l}} x : A} \quad \text{(prod)} \quad \frac{\Gamma \vdash_{f\vec{l}} A : \text{TYPE} \quad \Gamma, x : A \vdash_{f\vec{l}} B : s}{\Gamma \vdash_{f\vec{l}} \forall x : A, B : s} \\
\text{(abs)} \quad \frac{\Gamma, x : A \vdash_{f\vec{l}} b : B \quad \Gamma \vdash_{\prec f} \forall x : A, B : s}{\Gamma \vdash_{f\vec{l}} \lambda x : A, b : \forall x : A, B} \\
\text{(app')} \quad \frac{\Gamma \vdash_{f\vec{l}} t : \forall x : A, B \quad \Gamma \vdash_{f\vec{l}} a : A \quad \Gamma \vdash_{\prec f} \forall x : A, B : s}{\Gamma \vdash_{f\vec{l}} ta : B[x \mapsto a]} \\
\text{(conv')} \quad \frac{\Gamma \vdash_{f\vec{l}} a : A \quad A \downarrow B \quad \Gamma \vdash_{\prec f} B : s \quad \Gamma \vdash_{\prec f} A : s}{\Gamma \vdash_{f\vec{l}} a : B} \\
\text{(dp)} \quad \frac{\vdash_{\prec f} \Theta_g : s_g \quad \Gamma \vdash_{f\vec{l}} \gamma : \Sigma}{\Gamma \vdash_{f\vec{l}} g\vec{y}\gamma : V\gamma} \quad (\Theta_g = (\forall \vec{y} : \vec{U}, V), \Sigma = \vec{y} : \vec{U}, g\vec{y}\gamma < f\vec{l}) \\
\text{(const)} \quad \frac{\vdash_{\prec f} \Theta_g : s_g}{\vdash_{f\vec{l}} g : \Theta_g} \quad (g \text{ undefined})
\end{array}$$

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

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

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

357 This property is not always satisfied because the subterm relation does not preserve
358 computability in general. Indeed, if C is an undefined type constant, then $\llbracket C \rrbracket = \text{SN}$.
359 However, $\llbracket C \Rightarrow C \rrbracket \neq \text{SN}$ since $\omega = \lambda x : C, xx \in \text{SN}$ and $\omega\omega \notin \text{SN}$. Hence, if c is an
360 undefined function symbol of type $\Theta_c = (C \Rightarrow C) \Rightarrow C$, then $c\omega \in \llbracket C \rrbracket$ but $\omega \notin \llbracket C \Rightarrow C \rrbracket$.

361 We can now state the main lemma:

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

363 **Proof.** Since \mathcal{R} is well-structured, \succ is well-founded by condition (a). We prove that,
364 for all $f \in \mathbb{F}$, $f \in \llbracket \Theta_f \rrbracket$, by induction on \succ . So, let $f \in \mathbb{F}$ with $\Theta_f = \forall \Gamma_f, U$ and
365 $\Gamma_f = x_1 : T_1, \dots, x_n : T_n$. By induction hypothesis, we have that, for all $g < f$, $g \in \llbracket \Theta_g \rrbracket$.

366 Since \rightarrow_{arg} and $\tilde{\succ}$ terminate on \mathbb{C} and $\rightarrow_{\text{arg}} \tilde{\succ} \subseteq \tilde{\succ}$, we have that $\rightarrow_{\text{arg}} \cup \tilde{\succ}$ terminates.
367 We now prove that, for all $f\vec{l} \in \mathbb{C}$, we have $f\vec{l} \in \llbracket U \rrbracket_\theta$ where $\theta = [\vec{x} \mapsto \vec{l}]$, by a second
368 induction on $\rightarrow_{\text{arg}} \cup \tilde{\succ}$. By condition (b), $f\vec{l}$ is neutral. Hence, by definition of computability,
369 it suffices to prove that, for all $u \in \rightarrow(f\vec{l})$, $u \in \llbracket U \rrbracket_\theta$. There are 2 cases:

- 370 ■ $u = f\vec{v}$ with $\vec{t} \rightarrow_{\text{prod}} \vec{v}$. Then, we can conclude by the first induction hypothesis.
- 371 ■ There are $f l_1 \dots l_k \rightarrow r \in \mathcal{R}$ and σ such that $u = (r\sigma)t_{k+1} \dots t_n$ and, for all $i \in \{1, \dots, k\}$,
372 $t_i = l_i\sigma$. Since $f\vec{l} \in \mathbb{C}$, we have $\pi\sigma \models \Gamma_f$. Since \mathcal{R} is accessible, we get that $\sigma \models \Delta_{f\vec{l} \rightarrow r}$.
373 By condition (d), we have $\Delta_{f\vec{l} \rightarrow r} \vdash_{f\vec{l}} r : V\pi$ where $V = \forall x_{k+1} : T_{k+1}, \dots, \forall x_n : T_n, U$.
374 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 $\sigma \models \Gamma$,
375 then $t\sigma \in \llbracket T \rrbracket_\sigma$, by a third induction on the structure of the derivation of $\Gamma \vdash_{f\vec{l}} t : T$

XX:10 Dependency Pairs Termination in Dependent Type Theory Modulo Rewriting

376 ($\Gamma \vdash_{\prec_f} t : T$ resp.), as in the proof of Lemma 5 except for (fun) replaced by (fun $_{\prec_f}$) in
 377 one case, and (const) and (dp) in the other case.

378 **(fun $_{\prec_f}$)** We have $g \in \llbracket \Theta_g \rrbracket$ by the first induction hypothesis on g .

379 **(const)** Since g is undefined, it is neutral and normal. Therefore, it belongs to every
 380 computability predicate and in particular to $\llbracket \Theta_g \rrbracket_\sigma$.

381 **(dp)** By the third induction hypothesis, $y_i \gamma \sigma \in \llbracket U_i \gamma \rrbracket_\sigma$. By Lemma 4e, $\llbracket U_i \gamma \rrbracket_\sigma = \llbracket U_i \rrbracket_{\gamma \sigma}$.
 382 So, $\gamma \sigma \models \Sigma$ and $g \vec{y} \gamma \sigma \in \mathbb{C}$. Now, by condition (c), $g \vec{y} \gamma \sigma \prec f \vec{l} \sigma$ since $g \vec{y} \gamma < f \vec{l}$.
 383 Therefore, by the second induction hypothesis, $g \vec{y} \gamma \sigma \in \llbracket V \gamma \rrbracket_\sigma$.

384 So, $r \sigma \in \llbracket V \pi \rrbracket_\sigma$ and, by Lemma 4d, $u \in \llbracket U \rrbracket_{[x_n \mapsto t_n, \dots, x_{k+1} \mapsto t_{k+1}, \pi \sigma]} = \llbracket U \rrbracket_\theta$. \blacktriangleleft

385 Note that the proof still works if one replaces the relation \succeq of Definition 8 by any
 386 well-founded quasi-order such that $f \succeq g$ whenever $f \vec{l} > g \vec{m}$. The quasi-order of Definition
 387 8, defined syntactically, relieves the user of the burden of providing one and is sufficient in
 388 every practical case met by the authors. However it is possible to construct ad-hoc systems
 389 which require a quasi-order richer than the one presented here.

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

392 **► Theorem 11.** *The relation $\rightarrow = \rightarrow_\beta \cup \rightarrow_{\mathcal{R}}$ terminates on terms typable in $\lambda\Pi/\mathcal{R}$ if \rightarrow is*
 393 *locally confluent and preserves typing, \mathcal{R} is well-structured and accessible, and $\tilde{\succ}$ terminates.*

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

5 Checking accessibility

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

399 As seen with the definition of accessibility, the main problem is to deal with subterms
 400 of higher-order type. A simple condition is to require higher-order variables to be direct
 401 subterms of the left-hand side, a condition called plain function-passing (PFP) in [25], and
 402 satisfied by Example 1.

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

406 **► Lemma 13.** *PFP systems are accessible.*

407 **Proof.** Let $f \vec{l} \rightarrow r$ be a PFP rule with $\Theta_f = \forall \Gamma, U$, $\Gamma = \vec{x} : \vec{T}$, $\pi = [\vec{x} \mapsto \vec{l}]$. Following
 408 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
 409 $(y : T) \in \Delta_{f \vec{l} \rightarrow r}$, $y \sigma \in \llbracket T \rrbracket_\sigma$.

410 \blacksquare Suppose $y = l_i$ and $T = T_i \pi$. Then, $y \sigma = l_i \sigma \in \llbracket T_i \rrbracket_{\pi \sigma}$. Since $\vdash \Theta_f : s_f$, $T_i \notin \mathbb{K} \cup \{\text{KIND}\}$.
 411 Since $\Theta_f \in \mathcal{D}$ and $\pi \sigma \models \Gamma$, we have $T_i \pi \sigma \in \mathcal{D}$. So, $\llbracket T_i \rrbracket_{\pi \sigma} = \mathcal{I}(T_i \pi \sigma)$. Since $T_i \notin$
 412 $\mathbb{K} \cup \{\text{KIND}\}$ and $\vec{l} \notin \mathbb{K} \cup \{\text{KIND}\}$, $T_i \pi \notin \mathbb{K} \cup \{\text{KIND}\}$. Since $T_i \pi \sigma \in \mathcal{D}$, $\llbracket T_i \pi \rrbracket_\sigma = \mathcal{I}(T_i \pi \sigma)$.
 413 Thus, $y \sigma \in \llbracket T \rrbracket_\sigma$.

414 \blacksquare Suppose $y \in \text{FV}(l_i)$ and T is of the form $C \vec{t}$ with $|\vec{t}| = \text{ar}(C)$. Then, $\llbracket T \rrbracket_\sigma = \text{SN}$ and
 415 $y \sigma \in \text{SN}$ since $l_i \sigma \in \llbracket T_i \rrbracket_\sigma \subseteq \text{SN}$. \blacktriangleleft

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

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

```
419 symbol Ord: TYPE
420 symbol zero: Ord symbol suc: Ord⇒Ord symbol lim: (Nat⇒Ord)⇒Ord
421
422 symbol ordrec: A⇒(Ord⇒A⇒A)⇒((Nat⇒Ord)⇒(Nat⇒A)⇒A)⇒Ord⇒A
423 rule ordrec u v w zero → u
424 rule ordrec u v w (suc x) → v x (ordrec u v w x)
425 rule ordrec u v w (lim f) → w f (λn, ordrec u v w (f n))
426
```

428 The above example is not PFP because $f:\text{Nat}\Rightarrow\text{Ord}$ is not argument of `ordrec`. Yet,
429 it is accessible if one takes for $\llbracket\text{Ord}\rrbracket$ the least fixpoint of the monotone function $F(S) =$
430 $\{t \in \text{SN} \mid \text{if } t \rightarrow^* \text{lim } f \text{ then } f \in \llbracket\text{Nat}\rrbracket \Rightarrow S, \text{ and if } t \rightarrow^* \text{suc } u \text{ then } u \in S\}$ [5].

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

433 $\llbracket\text{T } c\rrbracket = \text{if } c \downarrow = \text{arrow } a b \text{ then } \{t \in \text{SN} \mid \text{if } t \rightarrow^* \text{lam } f \text{ then } f \in \llbracket\text{T } a\rrbracket \Rightarrow \llbracket\text{T } b\rrbracket\}$ else SN

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

```
435 symbol Sort : TYPE symbol arrow : Sort ⇒ Sort ⇒ Sort
436
437 symbol T : Sort ⇒ TYPE
438 symbol lam : ∀ a b, (T a ⇒ T b) ⇒ T (arrow a b)
439 symbol app : ∀ a b, T (arrow a b) ⇒ T a ⇒ T b
440 rule app a b (lam _ _ f) x → f x
441
```

443 6 Size-change termination

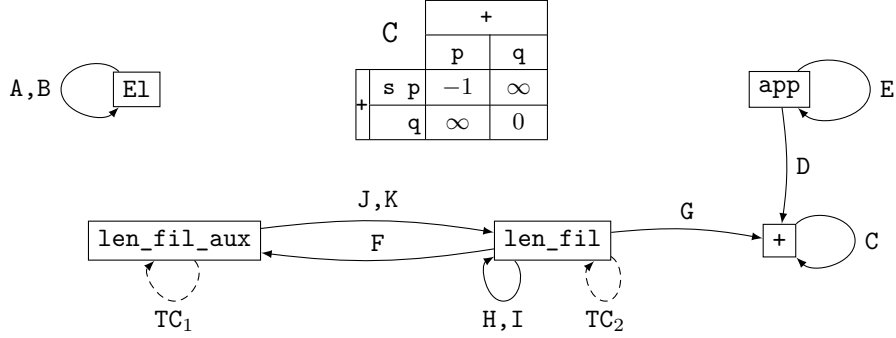
444 In this section, we give a sufficient condition for \succsim to terminate. For first-order rewriting,
445 many techniques have been developed for that purpose. To cite just a few, see for instance
446 [17, 14]. Many of them can probably be extended to $\lambda\Pi/\mathcal{R}$, either because the structure of
447 terms in which they are expressed can be abstracted away, or because they can be extended
448 to deal also with variable applications, λ -abstractions and β -reductions.

449 As an example, following Wahlstedt [38], we are going to use Lee, Jones and Ben-Amram's
450 size-change termination criterion (SCT) [26]. It consists in following arguments along function
451 calls and checking that, in every potential loop, one of them decreases. First introduced for
452 first-order functional languages, it has then been extended to many other settings: untyped
453 λ -calculus [21], a subset of OCAML [32], Martin-Löf's type theory [38], System F [27].

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

455 ▶ **Definition 16** (Size-change termination). *Let \triangleright be the smallest transitive relation such that*
456 *$ft_1 \dots t_n \triangleright t_i$ when $f \in \mathbb{F}$. The call graph $\mathcal{G}(\mathcal{R})$ associated to \mathcal{R} is the directed labeled graph*
457 *on the defined symbols of \mathbb{F} such that there is an edge between f and g iff there is a dependency*
458 *pair $fl_1 \dots l_p > gm_1 \dots m_q$. This edge is labeled with the matrix $(a_{i,j})_{i \leq \text{ar}(\Theta_f), j \leq \text{ar}(\Theta_g)}$ where:*
459 *■ if $l_i \triangleright m_j$, then $a_{i,j} = -1$;*
460 *■ if $l_i = m_j$, then $a_{i,j} = 0$;*
461 *■ otherwise $a_{i,j} = \infty$ (in particular if $i > p$ or $j > q$).*
462 *\mathcal{R} is size-change terminating (SCT) if, in the transitive closure of $\mathcal{G}(\mathcal{R})$ (using the min-plus*
463 *semi-ring to multiply the matrices labeling the edges), all idempotent matrices labeling a loop*
464 *have some -1 on the diagonal.*

■ **Figure 3** Matrix of dependency pair C and call graph of the dependency pairs of Example 7



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

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

471 ► **Lemma 17.** \succ terminates if \mathbb{F} is finite and \mathcal{R} is SCT.

472 **Proof.** Suppose that there is an infinite sequence $\chi = f_1 \vec{t}_1 \succ f_2 \vec{t}_2 \succ \dots$. Then, there is an
 473 infinite path in the call graph going through nodes labeled by f_1, f_2, \dots . Since \mathbb{F} is finite,
 474 there is a symbol g occurring infinitely often in this path. So, there is an infinite sequence
 475 $g\vec{u}_1, g\vec{u}_2, \dots$ extracted from χ . Hence, for every $i, j \in \mathbb{N}^*$, there is a matrix in the transitive
 476 closure of the graph which labels the loops of g corresponding to the relation between \vec{u}_i and
 477 \vec{u}_{i+j} . By Ramsey's theorem, there is an infinite sequence (ϕ_i) and a matrix M such that M
 478 corresponds to all the transitions $g\vec{u}_{\phi_i}, g\vec{u}_{\phi_j}$ with $i \neq j$. M is idempotent, indeed $g\vec{u}_{\phi_i}, g\vec{u}_{\phi_{i+2}}$
 479 is labeled by M^2 by definition of the transitive closure and by M due to Ramsey's theorem,
 480 so $M = M^2$. Since, by hypothesis, \mathcal{R} satisfies SCT, there is j such that $M_{j,j} = -1$. So, for
 481 all $i, u_{\phi_i}^{(j)} (\rightarrow^* \triangleright)^+ u_{\phi_{i+1}}^{(j)}$. Since $\triangleright \rightarrow \subseteq \rightarrow \triangleright$ and \rightarrow_{arg} is well-founded on \mathbb{C} , the existence of
 482 an infinite sequence contradicts the fact that \triangleright is well-founded. ◀

483 By combining all the previous results, we get:

484 ► **Theorem 18.** The relation $\rightarrow = \rightarrow_\beta \cup \rightarrow_{\mathcal{R}}$ terminates on terms typable in $\lambda\Pi/\mathcal{R}$ if \rightarrow is
 485 locally confluent and preserves typing, \mathbb{F} is finite and \mathcal{R} is well-structured, plain-function
 486 passing and size-change terminating.

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

488 7 Implementation and comparison with other criteria and tools

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

491 If we restrict ourselves to simply-typed rewriting systems, then we can compare it with
 492 the termination checkers participating in the category “higher-order rewriting union beta” of
 493 the termination competition: WANDA uses dependency pairs, polynomial interpretations,

494 HORPO and many transformation techniques [24]; SOL uses the General Schema [6] and other
 495 techniques. As these tools implement various techniques and SIZECHANGETOOL only one, it is
 496 difficult to compete with them. Still, there are examples that are solved by SIZECHANGETOOL
 497 and not by one of the other tools, demonstrating that these tools would benefit from
 498 implementing our new technique. For instance, the problem Hamana_Kikuchi_18/h17 is
 499 proved terminating by SIZECHANGETOOL but not by WANDA because of the rule:

```
500 rule map f (map g l) → map (comp f g) l
501
502
```

503 And the problem Kop13/kop12thesis_ex7.23 is proved terminating by SIZECHANGETOOL
 504 but not by SOL because of the rules:³

```
505 rule f h x (s y) → g (c x (h y)) y   rule g x y → f (λ_,s 0) x y
506
507
```

508 One could also imagine to translate a termination problem in $\lambda\Pi/\mathcal{R}$ into a simply-typed
 509 termination problem. Indeed, the termination of $\lambda\Pi$ alone (without rewriting) can be reduced
 510 to the termination of the simply-typed λ -calculus [16]. This has been extended to $\lambda\Pi/\mathcal{R}$ when
 511 there are no type-level rewrite rules like the ones defining E1 in Example 1 [22]. However,
 512 this translation does not preserve termination as shown by the Example 15 which is not
 513 terminating if all the types Tx are mapped to the same type constant.

514 In [30], Roux also uses dependency pairs for the termination of simply-typed higher-order
 515 rewriting systems, as well as a restricted form of dependent types where a type constant C is
 516 annotated by a pattern l representing the set of terms matching l . This extends to patterns
 517 the notion of indexed or sized types [18]. Then, for proving the absence of infinite chains, he
 518 uses simple projections [17], which can be seen as a particular case of SCT where strictly
 519 decreasing arguments are fixed (SCT can also handle permutations in arguments).

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

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

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

529 **8 Conclusion and future work**

530 We proved a general modularity result extending Arts and Giesl's theorem that a rewriting
 531 relation terminates if there are no infinite sequences of dependency pairs [2] from first-order
 532 rewriting to dependently-typed higher-order rewriting. Then, following [38], we showed how
 533 to use Lee, Jones and Ben-Amram's size-change termination criterion to prove the absence
 534 of such infinite sequences [26].

535 This extends Wahlstedt's work [38] from weak to strong normalization, and from orthog-
 536 onal to locally confluent rewriting systems. This extends the first author's work [5] from
 537 orthogonal to locally confluent systems, and from systems having a decreasing argument in
 538 each recursive call to systems with non-increasing arguments in recursive calls. Finally, this

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

539 also extends previous works on static dependency pairs [25] from simply-typed λ -calculus to
 540 dependent types modulo rewriting.

541 To get this result, we assumed local confluence. However, one often uses termination to
 542 check (local) confluence. Fortunately, there are confluence criteria not based on termination.
 543 The most famous one is (weak) orthogonality, that is, when the system is left-linear and
 544 has no critical pairs (or only trivial ones) [36], as it is the case in functional programming
 545 languages. A more general one is when critical pairs are “development-closed” [37].

546 This work can be extended in various directions.

547 First, our tool is currently limited to PFP rules, that is, to rules where higher-order
 548 variables are direct subterms of the left-hand side. To have higher-order variables in deeper
 549 subterms like in Example 14, we need to define a more complex interpretation of types,
 550 following [5].

551 Second, to handle recursive calls in such systems, we also need to use an ordering more
 552 complex than the subterm ordering when computing the matrices labeling the SCT call
 553 graph. The ordering needed for handling Example 14 is the “structural ordering” of COQ
 554 and AGDA [9, 6]. Relations other than subterm have already been considered in SCT but in
 555 a first-order setting only [35].

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

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

```
559 symbol minus: Nat⇒Nat⇒Nat          set infix 1 "-" := minus
560   rule 0 - n → 0          rule m - 0 → m          rule (s m) - (s n) → m - n
561   symbol div: Nat⇒Nat⇒Nat        set infix 1 "/" := div
562   rule 0 / (s n) → 0          rule (s m) / (s n) → s ((m - n) / (s n))
```

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

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

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

576 where the number of s 's strictly decreases between two calls to f although the first rule
 577 makes the number of s 's increase. Hyvernat enriched SCT to handle such systems [19].

578 **Acknowledgments.** The authors thank the anonymous referees for their comments,
 579 which have improved the quality of this article.

580 — References —

- 581 1 A. Abel. MiniAgda: integrating sized and dependent types. PAR'10.
- 582 2 T. Arts, J. Giesl. Termination of term rewriting using dependency pairs. TCS 236:133–178,
 583 2000.
- 584 3 H. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, T. S. E. Maibaum,
 585 editors, Handbook of logic in computer science. Volume 2. Background: computational
 586 structures, p. 117–309. Oxford University Press, 1992.

- 587 4 F. Blanqui. Théorie des types et réécriture. PhD thesis, Université Paris-Sud, France, 2001.
- 588 5 F. Blanqui. Definitions by rewriting in the calculus of constructions. *MSCS* 15(1):37–92, 2005.
- 589 6 F. Blanqui. Termination of rewrite relations on λ -terms based on Girard's notion of reducibility.
590 *TCS* 611:50–86, 2016.
- 591 7 F. Blanqui. Size-based termination of higher-order rewriting. *JFP* 28(e11), 2018. 75 pages.
- 592 8 W. N. Chin, S. C. Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*,
593 14(2-3):261–300, 2001.
- 594 9 T. Coquand. Pattern matching with dependent types. *TYPES'92*.
- 595 10 D. Cousineau, G. Dowek. Embedding pure type systems in the $\lambda\Pi$ -calculus modulo. *TLCA'07*.
- 596 11 P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory.
597 *JSL* 65(2):525–549, 2000.
- 598 12 G. Genestier. *SizeChangeTool*. <https://github.com/Deducteam/SizeChangeTool>, 2018.
- 599 13 J. Giesl, R. Thiemann, P. Schneider-Kamp. The dependency pair framework: combining
600 techniques for automated termination proofs. *LPAR'04*.
- 601 14 J. Giesl, R. Thiemann, P. Schneider-Kamp, S. Falke. Mechanizing and improving dependency
602 pairs. *JAR* 37(3):155–203, 2006.
- 603 15 J.-Y. Girard, Y. Lafont, P. Taylor. *Proofs and types*. Cambridge University Press, 1988.
- 604 16 R. Harper, F. Honsell, G. Plotkin. A framework for defining logics. *JACM* 40(1):143–184,
605 1993.
- 606 17 N. Hirokawa, A. Middeldorp. Tyrolean Termination Tool: techniques and features. *IC*
607 205(4):474–511, 2007.
- 608 18 J. Hughes, L. Pareto, A. Sabry. Proving the correctness of reactive systems using sized types.
609 *POPL'96*.
- 610 19 P. Hyvernats. The size-change termination principle for constructor based languages. *LMCS*
611 10(1):1–30, 2014.
- 612 20 P. Hyvernats, C. Raffalli. Improvements on the "size change termination principle" in a
613 functional language. *WST'10*.
- 614 21 N. D. Jones, N. Bohr. Termination analysis of the untyped lambda-calculus. *RTA'04*.
- 615 22 J.-P. Jouannaud, J. Li. Termination of Dependently Typed Rewrite Rules. *TLCA'15*.
- 616 23 J. W. Klop, V. van Oostrom, F. van Raamsdonk. Combinatory reduction systems: introduction
617 and survey. *TCS* 121:279–308, 1993.
- 618 24 C. Kop. Higher order termination. PhD thesis, VU University Amsterdam, 2012.
- 619 25 K. Kusakari, M. Sakai. Enhancing dependency pair method using strong computability in
620 simply-typed term rewriting systems. *AAECC* 18(5):407–431, 2007.
- 621 26 C. S. Lee, N. D. Jones, A. M. Ben-Amram. The size-change principle for program termination.
622 *POPL'01*.
- 623 27 R. Lepigre, C. Raffalli. Practical subtyping for System F with sized (co-)induction. 2017.
- 624 28 G. Markowsky. Chain-complete posets and directed sets with applications. *Algebra Universalis*,
625 6:53–68, 1976.
- 626 29 R. Mayr, T. Nipkow. Higher-order rewrite systems and their confluence. *TCS* 192(2):3–29,
627 1998.
- 628 30 C. Roux. Refinement Types as Higher-Order Dependency Pairs. *RTA'11*.
- 629 31 R. Saillard. Type checking in the Lambda-Pi-calculus modulo: theory and practice. PhD
630 thesis, Mines ParisTech, France, 2015.
- 631 32 D. Sereni, N. D. Jones. Termination analysis of higher-order functional programs. *APLAS'05*.
- 632 33 *TeReSe*. Term rewriting systems, volume 55 of Cambridge Tracts in Theoretical Computer
633 Science. Cambridge University Press, 2003.
- 634 34 R. Thiemann. The DP framework for proving termination of term rewriting. PhD thesis,
635 RWTH Aachen University, 2007. Technical Report AIB-2007-17.
- 636 35 R. Thiemann, J. Giesl. The size-change principle and dependency pairs for termination of
637 term rewriting. *AAECC* 16(4):229–270, 2005.

XX:16 Dependency Pairs Termination in Dependent Type Theory Modulo Rewriting

- 638 **36** V. van Oostrom. Confluence for abstract and higher-order rewriting. PhD thesis, Vrije
639 Universiteit Amsterdam, 1994.
- 640 **37** V. van Oostrom. Developing developments. TCS 175(1):159–181, 1997.
- 641 **38** D. Wahlstedt. Dependent type theory with first-order parameterized data types and well-
642 founded recursion. PhD thesis, Chalmers University of Technology, 2007.

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) $T \in \text{SN}$ and (2) if $T \rightarrow^* (x : A)B$ then $A \in \text{dom}(J)$ and, for all $a \in J(A)$, $B[x \mapsto a] \in \text{dom}(J)$:

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

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 $A \in \text{dom}(I)$ and, for all $a \in I(A)$, $B[x \mapsto a] \in \text{dom}(I)$. Since $\text{dom}(I) \subseteq \text{dom}(J)$, 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 $F(I)(T) = F(J)(T) = \text{SN}$. ◀

A.2 Computability predicates

► **Lemma 21.** *D is a computability predicate.*

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

1. $\mathcal{D} \subseteq \text{SN}$ by definition of D .
2. Let $T \in \mathcal{D}$ and T' such that $T \rightarrow T'$. We have $T' \in \text{SN}$ since $T \in \text{SN}$. Assume now that $T' \rightarrow^* (x : A)B$. Then, $T \rightarrow^* (x : A)B$, $A \in \mathcal{D}$ 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 $\rightarrow(T) \subseteq \mathcal{D}$. Since $\mathcal{D} \subseteq \text{SN}$, $T \in \text{SN}$. 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 $a \in \mathcal{I}(A)$, $B[x \mapsto a] \in \mathcal{D}$. ◀

► **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 $t \in \text{SN}$. Let $x \in \mathbb{V}$. Since $P \in \mathbb{P}$, $x \in P$. So, $tx \in Q(x)$. Since $Q(x) \in \mathbb{P}$, $Q(x) \subseteq \text{SN}$. Therefore, $tx \in \text{SN}$, and $t \in \text{SN}$.
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 $a \in \text{SN}$ 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 $t \in R$. ◀

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

XX:18 Dependency Pairs Termination in Dependent Type Theory Modulo Rewriting

684 **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
 685 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
 686 $T \in D(I)$, $F(I)(T) \in \mathbb{P}$. There are two cases:

- 687 ■ 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,
 688 for $a \in I(A)$, $I(B[x \mapsto a]) \in \mathbb{P}$. Hence, by Lemma 22, $F(I)(T) \in \mathbb{P}$.
- 689 ■ Otherwise, $F(I)(T) = \text{SN} \in \mathbb{P}$. ◀

690 ► **Lemma 4a.** *For all terms T and substitutions σ , $\llbracket T \rrbracket_\sigma \in \mathbb{P}$.*

691 **Proof.** By induction on T . If $T = s$, then $\llbracket T \rrbracket_\sigma = \mathcal{D} \in \mathbb{P}$ by Lemma 21. If $T = (x : A)K \in \mathbb{K}$,
 692 then $\llbracket T \rrbracket_\sigma = \Pi a \in \llbracket A \rrbracket_\sigma. \llbracket K \rrbracket_{[x \mapsto a, \sigma]}$. By induction hypothesis, $\llbracket A \rrbracket_\sigma \in \mathbb{P}$ and, for all $a \in \llbracket A \rrbracket_\sigma$,
 693 $\llbracket K \rrbracket_{[x \mapsto a, \sigma]} \in \mathbb{P}$. Hence, by Lemma 22, $\llbracket T \rrbracket_\sigma \in \mathbb{P}$. If $T \notin \mathbb{K} \cup \{\text{KIND}\}$ and $T\sigma \in \mathcal{D}$, then
 694 $\llbracket T \rrbracket_\sigma = \mathcal{I}(T\sigma) \in \mathbb{P}$ by Lemma 23. Otherwise, $\llbracket T \rrbracket_\sigma = \text{SN} \in \mathbb{P}$. ◀

695 A.3 Invariance by reduction

696 We now prove that the interpretation is invariant by reduction.

697 ► **Lemma 24.** *If $T \in \mathcal{D}$ and $T \rightarrow T'$, then $\mathcal{I}(T) = \mathcal{I}(T')$.*

698 **Proof.** First note that $T' \in \mathcal{D}$ since $\mathcal{D} \in \mathbb{P}$. Hence, $\mathcal{I}(T')$ is well defined. Now, we have
 699 $T \in \text{SN}$ since $\mathcal{D} \subseteq \text{SN}$. So, $T' \in \text{SN}$ and, by local confluence and Newman's lemma,
 700 $T \downarrow = T' \downarrow$. 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,
 701 $\mathcal{I}(T) = \text{SN} = \mathcal{I}(T')$. ◀

702 ► **Lemma 4b.** *If T is typable, $T\sigma \in \mathcal{D}$ and $T \rightarrow T'$, then $\llbracket T \rrbracket_\sigma = \llbracket T' \rrbracket_\sigma$.*

703 **Proof.** By assumption, there are Γ and U such that $\Gamma \vdash T : U$. Since \rightarrow preserves typing,
 704 we also have $\Gamma \vdash T' : U$. So, $T \neq \text{KIND}$, and $T' \neq \text{KIND}$. Moreover, $T \in \mathbb{K}$ iff $T' \in \mathbb{K}$ since
 705 $\Gamma \vdash T : \text{KIND}$ 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
 706 $\mathcal{D} \in \mathbb{P}$.

707 We now prove the result, with $T \rightarrow^= T'$ instead of $T \rightarrow T'$, by induction on T . If
 708 $T \notin \mathbb{K}$, then $T' \notin \mathbb{K}$ and, since $T\sigma, T'\sigma \in \mathcal{D}$, $\llbracket T \rrbracket_\sigma = \mathcal{I}(T\sigma) = \mathcal{I}(T'\sigma) = \llbracket T' \rrbracket_\sigma$ by Lemma
 709 24. If $T = \text{TYPE}$, then $\llbracket T \rrbracket_\sigma = \mathcal{D} = \llbracket T' \rrbracket_\sigma$. Otherwise, $T = (x : A)K$ and $T' = (x : A')K'$
 710 with $A \rightarrow^= A'$ and $K \rightarrow^= K'$. By inversion, we have $\Gamma \vdash A : \text{TYPE}$, $\Gamma \vdash A' : \text{TYPE}$,
 711 $\Gamma, x : A \vdash K : \text{KIND}$ and $\Gamma, x : A' \vdash K' : \text{KIND}$. So, by induction hypothesis, $\llbracket A \rrbracket_\sigma = \llbracket A' \rrbracket_\sigma$
 712 and, for all $a \in \llbracket A \rrbracket_\sigma$, $\llbracket K \rrbracket_{\sigma'} = \llbracket K' \rrbracket_{\sigma'}$, where $\sigma' = [x \mapsto a, \sigma]$. Therefore, $\llbracket T \rrbracket_\sigma = \llbracket T' \rrbracket_\sigma$. ◀

713 ► **Lemma 4c.** *If T is typable, $T\sigma \in \mathcal{D}$ and $\sigma \rightarrow \sigma'$, then $\llbracket T \rrbracket_\sigma = \llbracket T \rrbracket_{\sigma'}$.*

714 **Proof.** By induction on T .

- 715 ■ If $T \in \mathbb{S}$, then $\llbracket T \rrbracket_\sigma = \mathcal{D} = \llbracket T \rrbracket_{\sigma'}$.
- 716 ■ If $T = (x : A)K$ and $K \in \mathbb{K}$, then $\llbracket T \rrbracket_\sigma = \Pi a \in \llbracket A \rrbracket_\sigma. \llbracket K \rrbracket_{[x \mapsto a, \sigma]}$ and $\llbracket T \rrbracket_{\sigma'} = \Pi a \in$
 717 $\llbracket A \rrbracket_{\sigma'}. \llbracket K \rrbracket_{[x \mapsto a, \sigma']}$. By induction hypothesis, $\llbracket A \rrbracket_\sigma = \llbracket A \rrbracket_{\sigma'}$ and, for all $a \in \llbracket A \rrbracket_\sigma$,
 718 $\llbracket K \rrbracket_{[x \mapsto a, \sigma]} = \llbracket K \rrbracket_{[x \mapsto a, \sigma']}$. Therefore, $\llbracket T \rrbracket_\sigma = \llbracket T \rrbracket_{\sigma'}$.
- 719 ■ If $T\sigma \in \mathcal{D}$, then $\llbracket T \rrbracket_\sigma = \mathcal{I}(T\sigma)$ and $\llbracket T \rrbracket_{\sigma'} = \mathcal{I}(T\sigma')$. Since $T\sigma \rightarrow^* T\sigma'$, by Lemma 4b,
 720 $\mathcal{I}(T\sigma) = \mathcal{I}(T\sigma')$.
- 721 ■ Otherwise, $\llbracket T \rrbracket_\sigma = \text{SN} = \llbracket T \rrbracket_{\sigma'}$. ◀

A.4 Adequacy of the interpretation

► **Lemma 4d.** *If $(x : A)B$ is typable, $((x : A)B)\sigma \in \mathcal{D}$ and $x \notin \text{dom}(\sigma) \cup \text{FV}(\sigma)$, then*

$$\llbracket (x : A)B \rrbracket_\sigma = \Pi a \in \llbracket A \rrbracket_\sigma. \llbracket B \rrbracket_{[x \mapsto a, \sigma]}.$$

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

Since $(x : A)B$ is typable, A is of type TYPE and $A \notin \mathbb{K} \cup \{\text{KIND}\}$. Hence, $\llbracket A \rrbracket_\sigma = \mathcal{I}(A\sigma)$ 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 $B \notin \mathbb{K} \cup \{\text{KIND}\}$. Hence, $\llbracket B \rrbracket_{[x \mapsto a, \sigma]} = \mathcal{I}(B[x \mapsto a, \sigma])$. Since $x \notin \text{dom}(\sigma) \cup \text{FV}(\sigma)$, $B[x \mapsto a, \sigma] = (B\sigma)[x \mapsto a]$. Hence, $\llbracket B \rrbracket_{[x \mapsto a, \sigma]} = \mathcal{I}((B\sigma)[x \mapsto a])$ and, by Lemma 24, $\mathcal{I}((B\sigma)[x \mapsto a]) = \mathcal{I}((B\sigma\downarrow)[x \mapsto a])$.

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

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

► **Lemma 4e.** *If $\Delta \vdash U : s$, $\Gamma \vdash \gamma : \Delta$ and $U\gamma\sigma \in \mathcal{D}$, then $\llbracket U\gamma \rrbracket_\sigma = \llbracket U \rrbracket_{\gamma\sigma}$.*

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 $\llbracket U\gamma \rrbracket_\sigma = \mathcal{I}(U\gamma\sigma) = \llbracket U \rrbracket_{\gamma\sigma}$ since $U\gamma\sigma \in \mathcal{D}$.
- Otherwise, $s = \text{KIND}$ and $U \in \mathbb{K}$.
 - If $U = \text{TYPE}$, then $\llbracket U\gamma \rrbracket_\sigma = \mathcal{D} = \llbracket U \rrbracket_{\gamma\sigma}$.
 - Otherwise, $U = (x : A)K$ and, by Lemma 4d, $\llbracket U\gamma \rrbracket_\sigma = \Pi a \in \llbracket A\gamma \rrbracket_\sigma. \llbracket K\gamma \rrbracket_{[x \mapsto a, \sigma]}$ and $\llbracket U \rrbracket_{\gamma\sigma} = \Pi a \in \llbracket A \rrbracket_{\gamma\sigma}. \llbracket K \rrbracket_{[x \mapsto a, \gamma\sigma]}$. By induction hypothesis, $\llbracket A\gamma \rrbracket_\sigma = \llbracket A \rrbracket_{\gamma\sigma}$ and, for all $a \in \llbracket A\gamma \rrbracket_\sigma$, $\llbracket K\gamma \rrbracket_{[x \mapsto a, \sigma]} = \llbracket K \rrbracket_{\gamma[x \mapsto a, \sigma]}$. Wlog we can assume $x \notin \text{dom}(\gamma) \cup \text{FV}(\gamma)$. So, $\llbracket K \rrbracket_{\gamma[x \mapsto a, \sigma]} = \llbracket K \rrbracket_{[x \mapsto a, \gamma\sigma]}$. ◀

► **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 $a \rightarrow a'$. 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 $a \in \rightarrow^*(a_0)$. We can prove that $(\lambda x : A. b)a \in Q(a_0)$ by induction on (A, b, a) ordered by $(\rightarrow, \rightarrow, \rightarrow)_{\forall \cdot}$. Since $Q(a_0) \in \mathbb{P}$ and $(\lambda x : A. b)a$ is neutral, it suffices to prove that $\rightarrow((\lambda x : A. b)a) \subseteq Q(a_0)$. If the reduction takes place in A , b or a , we can conclude by induction hypothesis. Otherwise, $(\lambda x : A. b)a \rightarrow b[x \mapsto a] \in Q(a)$ by assumption. Since $a_0 \rightarrow^* a$ and $Q(a') \subseteq Q(a)$ whenever $a \rightarrow a'$, 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) >

```

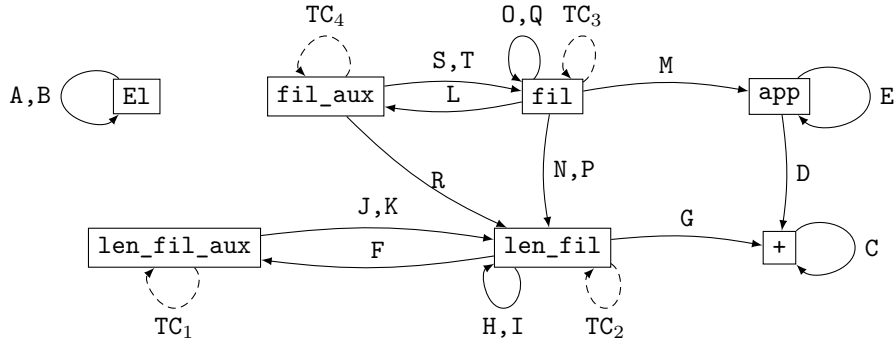
XX:20 Dependency Pairs Termination in Dependent Type Theory Modulo Rewriting

```

764                                     (len_fil a f p l) + (len_fil a f q m)
765 H: len_fil a f _ (app _ p l q m) > len_fil a f p l
766 I: len_fil a f _ (app _ p l q m) > len_fil a f q m
767 J:   len_fil_aux true  a f p l > len_fil a f p l
768 K:   len_fil_aux false a f p l > len_fil a f p l
769 L:   fil a f _ (cons _ x p l) > fil_aux (f x) a f x p l
770 M:   fil a f _ (app _ p l q m) >
771       app a (len_fil a f p l) (fil a f p l)
772       (len_fil a f q m) (fil a f q m)
773 N:   fil a f _ (app _ p l q m) > len_fil a f p l
774 O:   fil a f _ (app _ p l q m) > fil a f p l
775 P:   fil a f _ (app _ p l q m) > len_fil a f q m
776 Q:   fil a f _ (app _ p l q m) > fil a f q m
777 R:   fil_aux true  a f x p l > len_fil a f p l
778 S:   fil_aux true  a f x p l > fil a f p l
779 T:   fil_aux false a f x p l > fil a f p l
780

```

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.



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

$$\begin{aligned}
 786 \quad A, B &= (-1), \quad C = \begin{pmatrix} -1 & \infty \\ \infty & 0 \end{pmatrix}, \quad D = \begin{pmatrix} \infty & \infty \\ -1 & 0 \end{pmatrix}, \quad E = \begin{pmatrix} \infty & \infty & \infty & \infty \\ -1 & -1 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix}, \quad F = \begin{pmatrix} \infty & 0 & \infty & \infty \\ \infty & \infty & -1 & -1 \\ \infty & \infty & \infty & \infty \end{pmatrix}, \quad J=K = \begin{pmatrix} \infty & \infty & \infty \\ 0 & \infty & \infty \\ \infty & 0 & \infty \\ \infty & \infty & 0 \end{pmatrix}, \\
 787 \quad G &= \begin{pmatrix} \infty & \infty \\ \infty & \infty \\ \infty & \infty \end{pmatrix}, \quad H=I=N=O=P=Q = \begin{pmatrix} 0 & \infty & \infty \\ \infty & -1 & -1 \\ \infty & \infty & \infty \end{pmatrix}, \quad L = \begin{pmatrix} \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & -1 & -1 & -1 \\ \infty & \infty & \infty & \infty & \infty \end{pmatrix}, \quad M = \begin{pmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{pmatrix}, \\
 788 \quad R=S=T &= \begin{pmatrix} 0 & \infty & \infty \\ \infty & 0 & \infty \\ \infty & \infty & 0 \end{pmatrix}.
 \end{aligned}$$

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

$$\begin{aligned}
 790 \quad TC_1 &= J \times F = \begin{pmatrix} \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & -1 & -1 \\ \infty & \infty & \infty & \infty \end{pmatrix}, \quad TC_4 = S \times L = \begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & -1 & -1 & -1 \\ \infty & \infty & \infty & \infty & \infty \end{pmatrix}, \\
 791 \quad TC_3 &= L \times S = TC_2 = F \times J = \begin{pmatrix} 0 & \infty & \infty \\ \infty & \infty & \infty \\ \infty & -1 & -1 \end{pmatrix} = O = H.
 \end{aligned}$$

It would be useless to compute matrices labeling edges which are not in a strongly connected component of the call-graph (like $S \times R$), 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 $fl \rightarrow r$, we take the environment $\Delta_{fl \rightarrow r}$ made of all the variables of r with the following types: $a: \text{Set}$, $b: \text{Set}$, $p: \mathbb{N}$, $q: \mathbb{N}$, $x: \text{El } a$, $l: \mathbb{L} \ a$, $m: \mathbb{L} \ a$, $f: \text{El } a \Rightarrow \mathbb{B}$.

799 The precedence inferred for this example is the smallest containing:

800 ■ comparisons linked to the typing of symbols:

801

Set	⊢	arrow	Set,ℒ,0	⊢	nil
Set	⊢	El	Set,El,ℕ,ℒ,s	⊢	cons
ℬ	⊢	true	Set,ℕ,ℒ,+	⊢	app
ℬ	⊢	false	Set,El,ℬ,ℕ,ℒ	⊢	len_fil
ℕ	⊢	0	ℬ,Set,El,ℕ,ℒ	⊢	len_fil_aux
ℕ	⊢	s	Set,El,ℬ,ℕ,ℒ,len_fil	⊢	fil
ℕ	⊢	+	ℬ,Set,El,ℕ,ℒ,len_fil_aux	⊢	fil_aux
Set,ℕ	⊢	ℒ			

802 ■ and comparisons related to calls:

803

s	⊢	+	s,len_fil	⊢	len_fil_aux
cons,+	⊢	app	nil,fil_aux,app,len_fil	⊢	fil
0,len_fil_aux,+	⊢	len_fil	fil,cons,len_fil	⊢	fil_aux

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

