Termination of Dependently Typed Rewrite Rules

Jean-Pierre Jouannaud[1,2] and Jianqi Li[2]

1 **LIX, École Polytechnique, INRIA, and Université Paris-Sud, France**
2 **School of Software, Tsinghua University, NLIST, Beijing, China**

Abstract

Our interest is in automated termination proofs of higher-order rewrite rules in presence of dependent types modulo a theory T on base types. We first describe an original transformation to a type discipline without type dependencies which preserves non-termination. Since the user must reason on expressions of the transformed language, we then introduce an extension of the computability path ordering CPO for comparing dependently typed expressions named DCPO. Using the previous result, we show that DCPO is a well-founded order, behaving well in practice.

1998 ACM Subject Classification F.4.1 Mathematical Logic, F.4.2 Other Rewriting Systems

Keywords and phrases rewriting, dependent types, strong normalization, path orderings

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.257

1 Introduction

This paper addresses the problem of (semi-)automating termination proofs for typed higher-order calculi defined by rewrite rules. Since many automated techniques exist for showing termination of simply typed higher-order rewrite rules, our first approach is to reduce the former to the latter.

To this end, we introduce a non-termination preserving transformation from dependently typed algebraic λ-terms to simply typed algebraic λ-terms. Unlike the transformation used for showing strong normalization of LF [13], the present one uses algebraic symbols and type constructors in an essential way. Dependently typed rewrite rules can then be shown terminating via the transformation. The user can therefore benefit from all existing tools allowing to check termination of higher-order rewrite rules. The drawback is that these tools will operate on the transformed rules.

Among all termination proof techniques, we favour the one reducing termination proofs to ordering comparisons between lefthand and righthand sides of rules. These comparisons require well-founded orders on typed algebraic λ-terms which are stable by context application and substitution instance. CPO is such an order on *simply typed* algebraic λ-terms, defined recursively on the structure of the compared terms [9]. CPO is indeed well-founded on *weakly polymorphic* λ-terms, the familiar ML-discipline for which quantifiers on types can only occur in prefix position. A recent extension of core CPO to appear in LMCS handles inductive types, constructors possibly taking functional arguments, and function symbols smaller than application and abstraction.

We formulate here a new extension DCPO of CPO for dependently typed algebraic λ-terms. DCPO is then viewed as an infinite set of dependently typed rewrite rules which are shown terminating by checking the transformed rules with CPO. It follows that DCPO is a well-founded order of the set of dependently typed λ-terms, whose syntax-directed comparisons require little input from the user, in the form of a precedence on the algebraic symbols used in the rules. DCPO is our answer for practice.

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 257–272

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Variables:	**Abstraction:**	**Application:**	**Functions:**
$\dfrac{x{:}\sigma \in \Gamma}{\Gamma \ \vdash_\Sigma x{:}\sigma}$	$\dfrac{\Gamma,\, x{:}\sigma \ \vdash_\Sigma t{:}\tau}{\Gamma \ \vdash_\Sigma (\lambda x{:}\sigma.t){:}\sigma \to \tau}$	$\dfrac{\Gamma \ \vdash_\Sigma s{:}\sigma \to \tau \quad \Gamma \ \vdash_\Sigma t{:}\sigma}{\Gamma \ \vdash_\Sigma @(s,t){:}\tau}$	$\dfrac{f^n{:}\overline{\sigma} \to \sigma \in \mathcal{F} \quad \Gamma \ \vdash_\Sigma \overline{t}{:}\overline{\sigma}}{\Gamma \ \vdash_\Sigma f(\overline{t}){:}\sigma}$

Figure 1 Type system for monomorphic higher-order algebras.

Dependent programming has become a major trend in recent years [23, 16]. In practice, many types depend on natural numbers. Typing dependent definitions requires then a convertibility relation T including arithmetic laws [20], see Example 3.4. Our results allow for *dependent types modulo* T.

Sections 4 and 5 describe the non-termination preserving transformation and DCPO.

2 Higher-Order Algebras $\lambda^{\to}_{\Sigma}$

We assume a *signature* $\Sigma = \mathcal{S} \uplus \mathcal{F}$ of *sort symbols* in $\mathcal{S}$ and *function symbols* in $\mathcal{F}$. The set $\mathcal{T}^{\to}_{\mathcal{S}}$ of *simple types* (in short, *types*) is generated by the grammar $\sigma, \tau := a \in \mathcal{S} \mid \sigma \to \tau$. The (arrow) type constructor $\to$ associates to the right. The output sort of a type σ is itself if $\sigma \in \mathcal{S}$ and the output sort of τ if $\sigma = \nu \to \tau$. We use σ, τ, μ, ν for simple types.

Function symbols are meant to be algebraic operators upper-indexed by their fixed *arity* n. *Function declarations* are written $f^n : \sigma_1 \to \ldots \to \sigma_n \to \sigma$ (in short, $f : \overline{\sigma} \to \sigma$), where $\overline{\sigma}$ and σ are the *input* and *output types* of f^n. We use f^n, g^m for function symbols, possibly omitting m, n. $\lambda x : \sigma.s$, $@(s,t)$ and $f^n(t_1, \ldots, t_n)$ (or $f(\overline{t})$) are an *abstraction*, an *application*, and a *pre-algebraic* raw term. $f^0()$ is identified with f. We use x, y, z for variables, s, t, u, v, w, l, r for raw terms, $\mathrm{FV}(s)$ for the set of free variables of s and $|s|$ for the *size* of s.

Raw terms are seen as finite labeled trees by considering $\lambda x : \sigma.s$, for each $x : \sigma$, as a unary abstraction operator taking s as argument to construct the raw term $\lambda x : \sigma.s$. We abbreviate abstraction operators by λ. *Positions* are strings of strictly positive integers. We use i, j for positive integers, p, q for arbitrary positions. The empty string Λ is the *root* or *head* position and $\cdot$ is string concatenation. $\mathcal{P}os(t)$ is the set of positions of t.

Given a raw term s, $s(p)$ and $s|_p$ denote respectively the *symbol* and *subterm* of s at position p. For example, $(\lambda x : \sigma.u)|_1 = u$. The result of replacing the subterm $s|_p$ by the term t is written $s[t]_p$. A context term $s[x]_p$, in short $s[\,]_p$ or even $s[\,]$, is a term s in which x is a fresh variable, called *hole*, occurring at position p. All these notions extend as expected to a set P of disjoint positions, writing $s[t]_P$ for replacement of all terms in $s|_P$ by a single term t, and $s[\overline{x}]_P$ for a context with many holes.

An *environment* Γ is a finite set of pairs $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ where x_i is a variable, σ_i is a type, and $x_i \neq x_j$ for $i \neq j$. $\mathrm{FV}(\Gamma) = \{x_1, \ldots, x_n\}$ is the set of variables of Γ. Our typing judgements are written as $\Gamma \ \vdash_\Sigma s : \sigma$. A raw term s has type σ in the environment Γ if the judgement $\Gamma \ \vdash_\Sigma s : \sigma$ is provable in the inference system given in Figure 1. Typable raw terms are called *algebraic* λ*-terms* (in short, *terms* or *objects*), and their set is denoted by $\lambda_{\mathcal{F}}$ (or $\mathcal{O}^{\to}_{\mathcal{F}}$). Objects have a unique type in a given, possibly omitted, environment.

Substitutions are type-preserving homomorphisms avoiding captures, see for example [2], written here in postfix form, using the notation $\{x_i \mapsto s_i\}_i$. The congruence on terms generated by renaming the free variable x in s by the *fresh* variable $z \notin \mathrm{FV}(\lambda x.s)$ to yield the term $\lambda z.s\{x \mapsto z\}$ is called α*-conversion*, denoted by $=_\alpha$. We use γ, θ for substitutions.

A higher-order rewrite rule is a quadruple $\Delta \ \vdash_\Sigma l \to r : \sigma$ made of lefthand and righthand side terms l, r, and possibly omitted environment Δ and type σ. Rules β and η are particular

rewrite rule schemas. Reductions are defined as usual, and are a particular case of reductions in presence of dependent types, see Definition 3.3. A *higher-order reduction ordering* $\succeq$ is a quasi-order on terms satisfying: (i) its strict part $\succ$ is well-founded ; (ii) its equivalence is a congruence ; (iii) *monotonicity*: $s \succ t$ implies $u[s]_p \succ u[t]_p$ (assuming typability); (iv) *stability*: $s \succ t$ implies $s\gamma \succ t\gamma$ for all substitutions γ; and (v) *functionality*: $\longrightarrow_\beta \cup \longrightarrow_\eta \subseteq \succ$.

Given a set E the notation $\overline{s}$ shall be used for a list, multiset, or set of elements of E. Given a binary relation $\succ$ on E, we use $\overline{s} \succ_{lex} \overline{t}$ and $\overline{s} \succ_{mul} \overline{t}$ for its lexicographic and multiset extensions respectively. We use $s \succ \overline{t}$ for $(\forall t \in \overline{t})\, s \succ t$, and $\overline{s} \succ t$ for $(\exists s \in \overline{s})\, s \succ t$.

A rewrite relation generated by a set of rules $R \cup \{\beta, \eta\}$ can be proved terminating by checking whether $l \succ r$ for all rules in R with some higher-order reduction ordering $\succ$ [14]. CPO is such a higher-order reduction ordering based on three ingredients [9]:

- an order $\geq^{\to}$ on simple types, whose strict part $>^{\to}$ satisfies
  - **(i)** *well-foundedness:* $>^{\to} \cup \{\langle \sigma \to \tau, \sigma \rangle \mid \sigma, \tau \in \mathcal{T}^{\to}_{\mathcal{S}}\}$ is well-founded ;
  - **(ii)** *right arrow subterm:* $\sigma \to \tau >^{\to} \tau$;
  - **(iii)** *preservation:* $\sigma \to \tau =^{\to} \nu$ iff $\nu = \sigma' \to \tau'$, $\sigma =^{\to} \sigma'$, $\tau =^{\to} \tau'$ and
  - **(iv)** *decreasingness:* $\sigma \to \tau >^{\to} \nu$ implies $\tau \geq^{\to} \nu$ or $\nu = \sigma \to \mu$ and $\tau >^{\to} \mu$.
- a quasi-order $\geq_{\mathcal{F}}$ on $\mathcal{F} \cup \{@, \lambda\} \cup \mathcal{X}$ called *precedence* s.t.: (i) its strict part $>_{\mathcal{F}}$ restricts to $\mathcal{F} \cup \{@, \lambda\}$, is well-founded, and satisfies $(\forall f \in \mathcal{F})\ f >_{\mathcal{F}} @ >_{\mathcal{F}} \lambda$; (ii) its equivalence $=_{\mathcal{F}}$ contains pairs in $\mathcal{F} \times \mathcal{F}$ and all pairs $\{(x, x) \mid x \in \mathcal{X}\}$.
- $(\forall f \in \mathcal{F} \cup \{@, \lambda\})$, a status operator $\__f \in \{lex, mul\}$ in postfix index position such that $\__@ = mul$. Equivalent symbols have the same status.

The following auxiliary relations are used to define CPO [9] :

- $s \geq^X t$ iff $s =_\alpha t$ or $s >^X t$, for a set of variables X disjoint from $\mathrm{FV}(s)$, is the *main order*;
- $s >^X \overline{t}$ and $\overline{s} >^X t$ defined respectively as $(\forall v \in \overline{t})\, s >^X v$ and $(\exists u \in \overline{s})\, u >^X t$;
- $s : \sigma >^X t : \tau$ (resp., $s : \sigma \geq^X t : \tau$) for $s >^X t$ (resp., $s \geq^X t$) and $\sigma \geq^{\to} \tau$;
- we are interested in the *typed order* $s : \sigma >^\emptyset t : \tau$, written $s : \sigma > t : \tau$.

▶ **Definition 2.1.** Given $\Gamma \vdash_{\mathcal{F}} s : \sigma$ and $\Gamma \vdash_{\mathcal{F}} t : \tau$, $s >^X t$ iff either

$t \in X$ and $s \notin \mathcal{X}$	VAR
$s = \lambda x{:}\mu.u$ and $u\{x \mapsto z\} : \nu \geq^X t : \tau$	SUBTλ
$s = f(\overline{s})$, $t = \lambda x{:}\mu.v$ and $s >^{X \cup \{z:\mu\}} v\{x \mapsto z\}$	$\mathcal{F}\lambda$
$s = @(u, w)$, $t = \lambda x{:}\mu.v$, $x \notin \mathrm{FV}(v)$ and $s >^X v$	$@\lambda$
$s = \lambda x{:}\mu.u$, $t = \lambda y{:}\mu.v$ and $u\{x \mapsto z\} >^X v\{y \mapsto z\}$	$\lambda\lambda$
$s = @(\lambda x{:}\mu.u,\ w)$ and $u\{x \mapsto w\} \geq^X t$	BETA
$s = \lambda x{:}\mu.@(u, x)$, $x \notin \mathrm{FV}(u)$ and $u \geq^X t$	ETA

otherwise $s = f(\overline{s}), t = g(\overline{t})$ with $f, g \in \mathcal{F} \cup \{@, \lambda\} \cup \mathcal{X}$, and either

SUBT $\overline{s} : \overline{\sigma} \ \geq\ t : \tau$ PREC $f >_{\mathcal{F}} g$ and $s >^X \overline{t}$ STAT $f =_{\mathcal{F}} g,\ s >^X \overline{t}$ and $\overline{s} : \overline{\sigma} \ >_f\ \overline{t} : \overline{\tau}$

This definition of CPO is organized differently from [9] to be more compact. The last three cases originating in Dershowitz' recursive path ordering [12] describe the normal behaviour of head symbols, whether or not in $\mathcal{F}$. Note here that $>_f$ is the status extension (lexicographic or multiset) of the order $>$. The first 7 cases describe other behaviours, either specific (var, beta, eta), or using explicit α-conversion for the others. In its recursive call, case$\mathcal{F}\lambda$ increases the set X of upper variables, while other recursive calls either keep it unchanged or reset it to $\emptyset$. Relaxations of these recursive calls are indeed ill-founded.

▶ **Theorem 2.2** ([9]). *$>^+$ is a higher-order reduction ordering.*

3 Dependent algebras λ^{Π}_{Σ}

We move to a calculus with dependent types inspired by Edinburgh's LF [13], an extension of the simply typed λ-calculus which can be seen as a formal basis for dependently typed programming languages and their formal study as done with Elf [18].

In higher-order algebras, types are typable by a single, usually omitted constant TYPE. In dependent algebras, as in other type theories, types (called *type families*) are typed by *kinds*, TYPE being one of them, which describe their functional structure. Let $\mathcal{S}, \mathcal{F}$ and $\mathcal{V}$ be pairwise disjoint sets of respectively *type symbols*, *algebraic function symbols*, and *variables*. Algebraic function symbols in $\mathcal{F}$ may carry an arity upper-indexing their name. Type symbols in $\mathcal{S}$ are curried constants which kind may be functional. We use respectively f and a for a typical function or type symbol. Raw terms are given by the following grammar:

$$\begin{array}{rcl} \mathit{Kinds}\ K & := & \mathrm{TYPE} \mid \Pi x : A.K \\ \mathit{Types}\ A, B & := & a \mid \Pi x : A.B \mid \lambda x : A.B \mid A\ N \\ \mathit{Objects}\ M, N & := & x \mid \lambda x : A.M \mid M\ N \mid f^n(M_1, ..., M_n) \end{array}$$

λ and Π are LF's binders. Notation $=_\alpha$ stands for similarity. Here, λ and Π are binary operators, whose arguments are a type ($(\lambda x : A.u)|_1 = A$), and the body of the abstraction or product ($(\lambda x : A.u)|_2 = u$). Both may originate computations. We write f for $f^0()$.

3.1 Typing judgements

All LF expressions are typed, objects by types, types by kinds, kinds by a special untyped constant KIND (the *universe*) asserting their well-formedness, we also call them *valid*. Five kinds of judgement are recursively defined by the LF type system of Figure 2.

$\vdash_{sig} \Sigma$	Σ is a valid signature
$\Sigma \vdash_{\mathcal{C}} \Gamma$	Γ is a valid context assuming $\vdash_{sig} \Sigma$
$\Sigma\,;\,\Gamma \vdash_{\mathcal{K}} K\ :\ \mathrm{KIND}$	K is a valid kind assuming $\Sigma \vdash_{\mathcal{C}} \Gamma$
$\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} A\ :\ K$	type A has kind K assuming $\Sigma\,;\,\Gamma \vdash_{\mathcal{K}} K\ :\ \mathrm{KIND}$
$\Sigma\,;\,\Gamma \vdash_{\mathcal{O}} M : A$	object M has type A assuming $\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} A\ :\ K$
$\Sigma\,;\,\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} C\ :\ D$ and	$\Sigma\,;\,\Gamma \vdash_{\mathcal{T}\vee\mathcal{K}} C\ :\ D$ are self explanatory

Environments pair up a signature and a context. Signatures assign kinds to type symbols and product types to function symbols. Contexts assign product types to variables.

$$\begin{array}{rcll} \mathit{Env}\ \Theta & := & \Sigma\,;\,\Gamma & \\ \mathit{Sig}\ \Sigma & := & \mathtt{nil} \mid \Sigma, a : K \mid \Sigma, f^n : \Pi\{x_i : A_i\}_n.A & \\ \mathit{Con}\ \Gamma & := & \mathtt{nil} \mid \Gamma, x : A & \end{array}$$

where `nil` is the empty set, $\Pi\{x_i : A_i\}_n.A$ is $\Pi x_1 : A_1.(\ldots(\Pi x_n : A_n.A)\ldots)$

In dependent calculi, the order of constants or variables in the environment is determined by their types. This impacts the expression of the so-called substitution lemma:

▶ **Lemma 3.1.** *Let* $\Sigma\,;\,\Gamma \vdash_{\mathcal{O}} M : A$ *and* $\Sigma\,;\,\Gamma, x : A, \Gamma' \vdash_{\mathcal{T}\vee\mathcal{O}} s\ :\ \sigma$. *Then,* $\Sigma\,;\,\Gamma, \Gamma'\{x \mapsto M\} \vdash_{\mathcal{T}\vee\mathcal{O}} s\{x \mapsto M\}\ :\ \sigma\{x \mapsto M\}$.

Applying the substitution lemma several times introduces an order on the application of elementary substitutions. We use the notation $M \circ^n \{x_i \mapsto N_i\}_i$ to denote the sequential application to M of the *elementary* substitutions $\{x_1 \mapsto N_1\}$, ..., $\{x_n \mapsto N_n\}$ in this order. We use the word *dependent substitution* to stress this sequential behaviour of substitutions.

Given a valid signature Σ and a context Γ valid in Σ, the set λ_Σ of valid expressions, called *terms* is the (disjoint) union of the sets $\mathcal{K}^{\Pi}_{\Sigma}$ of valid *kinds*, $\mathcal{T}^{\Pi}_{\Sigma}$ of valid *types* and $\mathcal{O}^{\Pi}_{\Sigma}$ of

Signatures

$$[\textsc{Empty}]\ \frac{}{\vdash_{sig} \mathtt{nil}} \qquad [\textsc{TConst}]\ \frac{\Sigma\,;\,\mathtt{nil} \vdash_{\mathcal{K}} K\ :\ \mathrm{KIND}}{\vdash_{sig} \Sigma,\ a : K}\ a \notin dom(\Sigma)$$

$$[\textsc{Const}]\ \frac{\vdash_{sig} \Sigma \qquad \Sigma\,;\,\{x_i : A_i\}_k \vdash_{\mathcal{T}} A_{k+1}(k = 0..n)\ :\ \mathrm{TYPE}}{\vdash_{sig} \Sigma,\ f^n : \Pi\{x_i : A_i\}_n.A_{n+1}}\ f \notin dom(\Sigma)$$

Contexts

$$[\textsc{Empty}]\ \frac{\vdash_{sig} \Sigma}{\Sigma \vdash_{\mathcal{C}} \mathtt{nil}} \qquad [\textsc{Var}]\ \frac{\Sigma \vdash_{\mathcal{C}} \Gamma \qquad \Sigma\,;\,\Gamma \vdash_{\mathcal{T}} A\ :\ \mathrm{TYPE}}{\Sigma \vdash_{\mathcal{C}} \Gamma,\ x : A}\ x \notin dom(\Gamma)$$

Kinds

$$[\textsc{Univ}]\ \frac{\Sigma \vdash_{\mathcal{C}} \Gamma}{\Sigma\,;\,\Gamma \vdash_{\mathcal{K}} \mathrm{TYPE}\ :\ \mathrm{KIND}} \qquad [\textsc{Prod}]\ \frac{\Sigma\,;\,\Gamma,\ x : A \vdash_{\mathcal{K}} K\ :\ \mathrm{KIND}}{\Sigma\,;\,\Gamma \vdash_{\mathcal{K}} \Pi x : A.K\ :\ \mathrm{KIND}}$$

Types

$$[\textsc{Ax}]\ \frac{\Sigma \vdash_{\mathcal{C}} \Gamma}{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} a\ :\ K}\ a : K \in \Sigma \qquad [\textsc{Abs}]\ \frac{\Sigma\,;\,\Gamma,\ x : A \vdash_{\mathcal{T}} B\ :\ K}{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} \lambda x : A.B\ :\ \Pi x : A.K}$$

$$[\textsc{App}]\ \frac{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} A\ :\ \Pi x : B.K \qquad \Sigma\,;\,\Gamma \vdash_{\mathcal{O}} M : B}{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} A\,M\ :\ K\{x \mapsto M\}} \qquad [\textsc{Prod}]\ \frac{\Sigma\,;\,\Gamma,\ x : A \vdash_{\mathcal{T}} B\ :\ \mathrm{TYPE}}{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} \Pi x : A.B\ :\ \mathrm{TYPE}}$$

$$[\textsc{Conv}]\ \frac{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} A\ :\ K \qquad \Sigma\,;\,\Gamma \vdash_{\mathcal{K}} K'\ :\ \mathrm{KIND}}{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} A\ :\ K'}\ K \equiv K'$$

Objects

$$[\textsc{Ax}]\ \frac{\Sigma \vdash_{\mathcal{C}} \Gamma \qquad x : A \in \Gamma}{\Sigma\,;\,\Gamma \vdash_{\mathcal{O}} x : A} \qquad [\textsc{Fun}]\ \frac{\Sigma\,;\,\Gamma \vdash_{\mathcal{O}} M_i : A_i\{x_1 \mapsto M_1, ..., x_{i-1} \mapsto M_{i-1}\}}{\Sigma\,;\,\Gamma \vdash_{\mathcal{O}} f^n(M_1, ..., M_n) : A \circ^n \{x_i \mapsto M_i\}_i}\ \text{where } f^n : \Pi\{x_i : A_i\}_n.A \in \Sigma$$

$$[\textsc{Abs}]\ \frac{\Sigma\,;\,\Gamma,\ x : A \vdash_{\mathcal{O}} M : B}{\Sigma\,;\,\Gamma \vdash_{\mathcal{O}} \lambda x : A.M : \Pi x : A.B} \qquad [\textsc{App}]\ \frac{\Sigma\,;\,\Gamma \vdash_{\mathcal{O}} M : \Pi x : A.B \qquad \Sigma\,;\,\Gamma \vdash_{\mathcal{O}} N : A}{\Sigma\,;\,\Gamma \vdash_{\mathcal{O}} M\,N : B\{x \mapsto N\}}$$

$$[\textsc{Conv}]\ \frac{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} s\ :\ \sigma \qquad \Sigma\,;\,\Gamma \vdash_{\mathcal{T}\vee\mathcal{K}} \sigma'\ :\ \mathrm{TYPE/KIND}}{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} s\ :\ \sigma'}\ \sigma \equiv \sigma'$$

Figure 2 LF Typing rules.

valid *objects* (possibly abbreviated as $\mathcal{K}, \mathcal{T}, \mathcal{O}$). Our presentation of LF performs classically the necessary sanitary checks when forming the signatures and contexts and only those.

$\equiv$ denoting the *convertibility relation*, a congruence discussed next in more details, that is generated by $\beta\eta$-conversion on similar terms on the one hand and on the other hand by an additional arbitrary congruence between object terms stable by substitution (possibly identifying them all [4]). By its definition as a congruence, convertibility respects our syntactic categories, objects, types and kinds.

Our dependently typed calculus is referred to as λ^{Π}_{Σ} to stress the signature Σ, or simply LF.

Lexicography: we use K for kinds, A,B,C,D for types, M,N for objects, s,t,u,v,w,l,r for (objects or types), σ, τ, μ, ν for (type or kinds), and γ, θ for substitutions.

3.2 Dependent rewriting and convertibility

In LF, the usual rules of the λ-calculus apply at the object and type levels, making four different rules generating a congruence called *$\beta\eta$-convertibility*:

$$\begin{array}{lll|lll} \text{beta:} & (\lambda x\!:\!A.B)\ M & \to_{\beta_{\mathcal{T}}} \ B\{x \mapsto M\} & (\lambda x\!:\!A.M)\ N & \to_{\beta_{\mathcal{O}}} \ M\{x \mapsto N\} \\ \text{eta:} & \lambda x\!:\!A.(B\,x) & \to_{\eta_{\mathcal{T}}} \ B \ \text{ if } x \notin \mathrm{FV}(B) & \lambda x\!:\!A.(M\ x) & \to_{\eta_{\mathcal{O}}} \ M \ \text{ if } x \notin \mathrm{FV}(M) \end{array}$$

Convertibility plays a key role for typing via Conv. In LF, convertibility is defined as the congruence generated by $\beta\eta$-reductions. In reality, convertibility must be strengthened on object level terms in order to type most examples. This problem has been considered in the framework of the calculus of constructions with the Calculus on Inductive Constructions [17], the Calculus of Algebraic Constructions [7] and the Calculus of Constructions Modulo Theory [20, 3], for which convertibility includes respectively: primitive recursion at higher type generated by the user's inductive types; the user's higher-order rules; and a decidable object-level first-order theory like Presburger arithmetic. These frameworks can be restricted to the LF type system seen as a particular case of the calculus of constructions. In the context of LF, it *relates* to the liquid types discipline [19], which shares similar objectives.

We now introduce dependent rewrite rules and rewriting.

▶ **Definition 3.2.** Given a valid signature Σ, a *plain dependent rewriting system* is a set $\{\Delta_i \vdash l_i : \sigma_i \to r_i : \tau_i\}_i$ of quintuples made, for every index i, of a context Δ_i, lefthand and righthand side terms l_i, r_i, and terms σ_i, τ_i, s.t. $\mathrm{FV}(r_i) \subseteq \mathrm{FV}(l_i)$, $\Sigma\,;\, \Delta_i \vdash_{\mathcal{T}\vee\mathcal{O}} l_i \,:\, \sigma_i$ and $\Sigma\,;\, \Delta_i \vdash_{\mathcal{T}\vee\mathcal{O}} r_i \,:\, \tau_i$ with $\sigma_i \equiv \tau_i$. Δ_i, σ_i and τ_i may be omitted.

▶ **Definition 3.3** (Dependent rewriting)**.** Given a rewriting system R, one step rewriting is a relation over terms, written $\Sigma\,;\, \Gamma \vdash_R s \longrightarrow^p t$ (in short, $s \longrightarrow_R t$) defined as:

- s and t are both types or objects which are typable under $\Sigma;\ \Gamma$,
- $\Delta \vdash l \to r : A \in R$, where $\mathrm{FV}(\Delta) \cap \mathrm{FV}(\Gamma) = \emptyset$.
- $s = s[l \circ \gamma]_p$ and $t = s[r \circ \gamma]_p$, where γ is a dependent substitution wrt to Δ.

A major semantic property expected from rewrite rules is that rewriting preserves types. In presence of dependencies, types are usually preserved up to some congruence defined by the rules themselves, as in the Calculus of Inductive Constructions or the Calculus of Algebraic Constructions [7]. Here, preservation of typing by rewriting, up to *type erasures*, follows from Lemma 4.7.

▶ **Example 3.4.** Here is a simple example with dependent lists of elements of a given type A. We allow ourselves with some OBJ-like mixfix syntax, using "_" for arguments' positions:

$nat, A : \mathrm{TYPE};\ List : \Pi m : nat.\mathrm{TYPE};$
$0 : nat;\ \_ + 1 : \Pi n : nat.nat;\ \_ + \_ : \Pi\{m, n : nat\}.nat$
$cons : \Pi\{n : nat,\ a : A,\ l : List\ n\}.List\ (n + 1)$
$app : \Pi\{m, n : nat,\ k : List\, m,\ l : List\, n\}.\, List\, (m + n)$

$$0 + n \to n \qquad app(0, n, nil, l) \to l$$
$$(m + 1) + n \to (m + n) + 1 \quad app(m+1, n, cons(m, a, k), l) \to cons(m+n, a, app(m, n, k, l))$$

Using LF's typing rules given in Figure 2, we get:

$$\begin{array}{rcrcl}
\{m, n : nat\} & \vdash & n, 0 + n, (m + 1) + n, (m + n) + 1 & : & nat \\
\{n : nat, l : List\, n\} & \vdash & app(0, n, nil, l) & : & List\, (0 + n) \\
\{m, n : nat, k : List\, m, l : List\, n\} & \vdash & app(m+1, n, cons(m, a, k), l) & : & List\, ((m + 1) + n) \\
\{m, n : nat, k : List\, m, l : List\, n\} & \vdash & cons(m+n, a, app(m, n, k, l)) & : & List\, ((m + n) + 1)
\end{array}$$

Typing these rules requires a conversion relation extending $\beta\eta$-conversion on objects with Presburger arithmetic to identify $List\,(0+n), List\, n$ and $List\,((m+1)+n), List\,((m+n)+1)$.

4 Encoding LF in higher-order algebras

We define here a transformation from the source language to a target language which preserves non-termination of arbitrary reductions, not just β-reductions as in LF. Our target language is the simply-typed λ-calculus enriched with function symbols and type constants of Section 2, a choice which has three important advantages: the target vocabulary can be as close as possible from the source vocabulary; the transformation preserves termination as well as non-termination; the transformed rules can be checked for termination by CPO.

The higher-order algebra encoding λ^{Π}_{Σ} will be $\lambda^{\rightarrow}_{\Sigma^{flat}}$, where $\Sigma^{flat} = \mathcal{S}_{flat} \biguplus \Sigma_{flat}$ is a higher-order (non-dependent) signature whose two pieces are described next. The set of types in $\lambda^{\rightarrow}_{\Sigma^{flat}}$ is denoted by $\mathcal{T}^{\rightarrow}_{\mathcal{S}_{flat}}$. Terms in $\lambda^{\rightarrow}_{\Sigma^{flat}}$, whose set is denoted by $\lambda_{\Sigma^{flat}}$, are meant to encode LF objects and types in a way which mimics dependently typed computations. We do not encode LF kinds, since computations in kinds are indeed computations on types or objects. Indeed, dependent types will be encoded in $\lambda^{\rightarrow}_{\Sigma^{flat}}$ both as types and as terms.

4.1 Type erasures

Types. Types in $\lambda^{\rightarrow}_{\Sigma^{flat}}$ are arrow types built from the set of sorts $\mathcal{S}_{flat} = \{*\} \cup \{a \mid a : K \in \Sigma\}$.

The new sort $*$ will serve to encode LF product types as terms of sort $*$ in $\lambda^{\rightarrow}_{\Sigma^{flat}}$. LF objects will be encoded as terms whose types are erasures of LF types, as defined next.

Type erasures. The classical *erasing* transformation from families and kinds in λ^{Π}_{Σ} to types in $\lambda^{\rightarrow}_{\Sigma^{flat}}$ eliminates dependencies from objects by replacing product types by arrow types:

▶ **Definition 4.1.** The *erasing* map $|.| : \mathcal{T} \cup \mathcal{K} \to \mathcal{T}^{\rightarrow}_{\mathcal{S}_{flat}}$ is defined as:

$$(1)|a| = a \qquad (2)|\Pi x : A.B| = |A| \to |B| \qquad (3)|\lambda x : A.B| = |B|$$
$$(4)|\mathrm{TYPE}| = * \qquad (5)|\Pi x : A.K| = |A| \to |K| \qquad (6)|A\, N| = |A|$$

Sorts being constants, an induction shows that variables in types are eliminated in rule (6):

▶ **Lemma 4.2.** *Let $A \in \mathcal{T} \cup \mathcal{K}$. Then $FV(|A|) = \emptyset$.*

▶ **Corollary 4.3.** *For any $E \in \mathcal{K} \cup \mathcal{T}$, $y \in \mathcal{V}$, and $s \in \mathcal{O}^{\rightarrow}_{\Sigma^{flat}}$, $|E\{y \mapsto s\}| = |E|\{y \mapsto |s|\} = |E|$.*

▶ **Lemma 4.4** (Conversion equality)**.** *Let $D, E \in \mathcal{T}$ such that $D \equiv E$. Then, $|D| = |E|$.*

Proof. Conversion is a congruence generated by $\beta\eta$-rewriting and an equivalence $=_{\mathcal{O}}$ on object terms. We show that the property is true of both relations by induction on D.

1. Case $D = a : K$. Then $E = D$.
2. Case $D = \Pi x : A.B : \text{TYPE}$. Then $E = \Pi x : A'.B' : \text{TYPE}$ with $A \equiv A'$ and $B \equiv B'$. By induction hypothesis (and definition of the erasing map).
3. Case $D = \lambda x : A.B : \Pi x : A.K$. There are two cases:
   (a) $E = \lambda x : A'.B' : \Pi x : A'.K$, with $A \equiv A'$ and $B \equiv B'$. By induction hypothesis.
   (b) $D = \lambda x{:}A.(E\ x) \to_\eta E$ where $x \notin \text{FV}(E)$. Then $|D| = |\lambda x{:}A.(E\,x)| = |E\,x| = |E|$.
3. Case $D = A\ M : K$, where M is an object. Again two cases:
   (a) $E = A'\ M'$, $A \equiv A'$ and $M \equiv M'$. Then, $|D| = |A|$ and $|E| = |A'|$. By induction.
   (b) $D = (\lambda x{:}F.G)\ M \to_\beta G\{x \mapsto M\} = E$. By Corol. 4.3, $|D| = |G| = |G\{x \mapsto M\}| = |E|$.

◀

Erasing is extended to environments $\Sigma\,;\Gamma$ by: $|\Gamma| = \{x : |A| \mid x : A \in \Gamma\}$ and $|\Sigma| = \{a : |K| \mid a : K \in \Sigma\} \cup \{f^n : |A_1| \to \ldots |A_n| \to |A| \text{ where } f^n : \Pi\{x_i : A_i\}_n.A \in \Sigma\}$.

Note here that a constructor like $cons : \Pi n : Nat, x : Nat, l : List(n).List(n+1) \in \Sigma$ becomes $cons : Nat \to Nat \to List \to List \in |\Sigma|$. Eliminating the first (type) argument from *cons* would easily allow writing rules for which termination is not preserved by the transformation. This does not mean, however, that writing such rules is actually impossible.

4.2 Term flattening

Besides types and function symbols in $|\Sigma|$, the signature Σ_{flat} will contain algebraic symbols, called *flattening constructors*, used to mimic LF's abstraction and product.

$$\Sigma_{flat} = |\Sigma| \cup \{lo^2_\sigma : * \to \sigma \to \sigma, \quad lf^2_\sigma : * \to \sigma \to \sigma, \quad pif^2_\sigma : * \to (\sigma \to *) \to * \quad | \quad \sigma \in \mathcal{T}^{\to}_{\mathcal{S}_{flat}}\}$$

We may omit subscripts σ, τ in constructor's names, and use lof^2 for $(lo^2_\sigma / lf^2_\sigma)$. The first argument of the flattening constructors, of type $*$, is the flattening of some dependent type A. The second argument of lo^2_σ is the interpretation of some object $M : A$, hence $\sigma = |A|$, while that of lf^2_σ is the interpretation of a type $A : K$, hence $\sigma = |K|$. Since signatures are monomorphic and σ is arbitrary, the flattening constructors must be indexed by these types. We now define a *flattening* transformation for expressions of LF which are typed in some environment left unspecified to expressions of $\lambda^{\to}_{\Sigma^{flat}}$:

▶ **Definition 4.5.** The *flattening* function $\|\_$ from λ_Σ to $\lambda_{\Sigma^{flat}}$ (the context in which the input term is valid is omitted) is defined as

$$\begin{array}{rcl|rcl} \|x : A & = & x : |A| & \|\Pi x : A.B : \text{TYPE} & = & pif^2_{|A|}(\|A, \lambda x : |A|.\|B) \\ \|M\ N : A & = & @(\|M, \|N) & \|\lambda x : A.M : \Pi x : A.B & = & \lambda x : |A|.lo^2_{|B|}(\|A, \|M) \\ \|A\ N : K & = & @(\|A, \|N) & \|\lambda x : A.B : \Pi x : A.K & = & \lambda x : |A|.lf^2_{|K|}(\|A, \|B) \\ \|a : K & = & a & \|f^n(M_1, ..., M_n) : A & = & f^n(\|M_1, ..., \|M_n) \end{array}$$

Since flattening is not surjective, we denote by $\|\lambda_\Sigma \subset \lambda_{\Sigma^{flat}}$ its target.

Note that type symbols in $\mathcal{S}$ become both sorts in $\mathcal{S}_{flat}$ and function symbols in Σ_{flat}, with the same (overloaded) name. This definition obeys the following principles: (i) flattening is a homomorphism, hence commutes with substitutions; (ii) because types may depend on objects, the type information associated with bound variables must be recorded by the encoding to preserve non-termination; (iii) the types of the flattening constructors are compatible with the erasing transformation, which allows to trace the syntactic categories in the transformed world; (iv) the encoding of abstractions and products preserves their variable's binding, but two different encodings are used. The encoding of abstractions is an

abstraction, in order to preserve beta-redexes via the transformation. Nothing like that is needed for product types which cannot be applied, and can therefore be transformed into terms of sort $|\text{TYPE}| = *$. In that case, the abstraction is encapsulated in the flattening constructor. This allows to single out easily products' encodings in the flattened world. Now,

an LF object $M : A$ is translated as a term $\|M$ of type $|A|$,
an LF type $A : K$ is translated as both a type $|A|$ and a term $\|A$ of type $|K|$,
an LF kind $K : \text{KIND}$ is translated as a type $|K|$.

▶ **Lemma 4.6.** *The following properties hold:*

1. *soundness: let* $\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} A : K$. *Then* $\Sigma_{flat}\,;\,|\Gamma| \vdash \|A : |K|$;
2. *soundness: let* $\Sigma\,;\,\Gamma \vdash_{\mathcal{O}} M : A$. *Then* $\Sigma_{flat}\,;\,|\Gamma| \vdash \|M : |A|$;
3. *preservation: let* $\Sigma\,;\,\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} s : \sigma$. *Then* $FV(s) = FV(\|s)$;
4. *stability: let* $\Sigma\,;\,\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} s, t : \sigma, \tau$ *and* $x : \tau \in \Gamma$. *Then* $\|s\{x \mapsto t\} = \|s\{x \mapsto \|t\}$.

Proof. The first three are proved by induction on the typing derivations of A, M and s respectively, using Lemma 4.4 for the third (translation of products) and for the conversion rule. Stability follows by induction on the typing derivation of s. ◀

4.3 Preservation of reductions by flattening

Our goal now is to show that the reductions on objects and types in λ^{Π}_{Σ} are mimicked in $\lambda^{\rightarrow}_{\Sigma^{flat}}$. Since rewriting a term cannot not increase its set of free variables, rewriting commutes with the η-rule. A consequence is that, given a dependent rewrite system R, $\longrightarrow_{\beta\eta R}$ terminates iff $\longrightarrow_{\beta R}$ terminates: encoding the η-rule will not be necessary. To ease the reading, we use $\rightharpoonup$ for our rewriting symbol in $\lambda^{\rightarrow}_{\Sigma^{flat}}$, and decorate subterms in flattened rules by their type.

$$[\beta_{\mathcal{O}}] \quad @(\lambda x : \sigma.lo^2_\tau(A : *, M : \tau), N : \sigma) \rightharpoonup M\{x \mapsto N\}$$
$$[\beta_{\mathcal{T}}] \quad @(\lambda x : \sigma.lf^2_\tau(A : *, B : \tau), N : \sigma) \rightharpoonup B\{x \mapsto N\}$$

In these rules, A and σ are the term flattening and type erasure of the same dependent type D, a relationship that cannot be expressed in $\lambda^{\rightarrow}_{\Sigma^{flat}}$, hence is not kept in the transformed rules. For example, assuming $A, s, t \in \|\lambda_\Sigma$ and $|\|A^{-1}| \neq \sigma$, then $u = @(\lambda x : \sigma.lof^2(A, s), t)$ is a redex which has no counter-part in λ^{Π}_{Σ}. There are indeed new rewrites in the flattened world, making preservation of non-termination a weaker property.

We use $[R]$, $[\beta]$, $[\beta R]$ and $\beta[\beta R]$ for $\{\|l \rightharpoonup \|r \,|\, l \to r \in R\}$, $\{[\beta_{\mathcal{T}}], [\beta_{\mathcal{O}}]\}$, $[\beta]\cup[R]$, and $\{\beta\}\cup[\beta R]$.

As with A, M, N, σ, τ used in $[\beta]$, variables of R that denote expressions in λ^{Π}_{Σ} keep their name in $[R]$, denoting now expressions of $\lambda^{\rightarrow}_{\Sigma^{flat}}$ belonging to the same syntactic categories as in the dependent world. Then, an instance of a $[\beta]$-rule may not be the encoding of an instance of the β-rule in the dependent world if M/B match flattened terms which are no encoding of dependent expressions. Despite these approximations, reductions are preserved:

▶ **Lemma 4.7.** *Let* $\Sigma;\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} s : \sigma$, $s \to_{\beta R} t$ *and* $\Sigma;\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} t : \tau$. *Then,* $|\sigma| = |\tau|$ *and* $\|s \rightharpoonup_{[\beta R]} \|t$.

Proof. The proof is by induction on the typing derivation of s. All cases are by induction except those where the βR redex is at the top. We carry out four typical cases:

1. $[\text{App}]\ \dfrac{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} A : \Pi x{:}B.K \quad \Sigma\,;\,\Gamma \vdash_{\mathcal{O}} M : B}{\Sigma\,;\,\Gamma \vdash_{\mathcal{T}} s = A\,M : K\{x \mapsto M\}}$ and $M \to_{\beta R} M'$, hence $A\,M \to_{\beta R} A\,M' = t$. By induction hypothesis, $\Sigma\,;\,\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} M' : B'$, $|B'| = |B|$ and $\|M \rightharpoonup_{[\beta R]} \|M'$. By definition of flattening, $\|A\,M = @(\|A, \|M) : |K\{x \mapsto M\}| = |K|$ by Corollary 4.3. By

definition of erasing, $|\Pi x : B.K| = |B| \to |K|$. By Lemma 4.6 (soundness) $\|A : |B| \to |K|$ and $\|M' : |B'|$. Finally, $\|t = @(\|A, \|M') : |K|$, we are done.

2. $[\textsc{Conv}]\ \dfrac{\Sigma;\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} s : \sigma' \quad \Sigma;\Gamma \vdash_{\mathcal{T}\vee\mathcal{K}} \sigma : \text{TYPE/KIND}}{\Sigma\,;\,\Gamma\vdash_{\mathcal{O}} s : \sigma}\ \sigma \equiv \sigma'.$ By induction hypothesis, $\Sigma;\Gamma\vdash_{\mathcal{T}\vee\mathcal{O}} t:\tau$, $|\tau|=|\sigma'|$ and $\|s \rightharpoonup_{[\beta R]} \|t$. By Lemma 4.6, $\|t:|\tau|$. Lemma 4.4 concludes.

3. $[\textsc{App}]\ \dfrac{\Sigma\,;\,\Gamma\vdash_{\mathcal{O}} \lambda x : A.u : \Pi x : A.B \qquad \Sigma\,;\,\Gamma\vdash_{\mathcal{O}} M : A}{\Sigma\,;\,\Gamma\vdash_{\mathcal{O}} s = \lambda x : A.u\,M : B\{x \mapsto M\}}$ and $t = u\{x \mapsto M\}$. By inversion, $\Sigma\,;\Gamma\vdash_{\mathcal{O}} u:B$, thus $t:B\{x\mapsto M\}$ by Lemma 3.1. $\|\lambda x:A.u\,M = @(\lambda x:A.lo^2_{|B|}(\|A, \|u), \|M)$ $\rightharpoonup_{[\beta]} \|u\{x \mapsto \|M\}$ by definition of $[\beta_{\mathcal{O}}]$. Lemma 4.6 and Corollary 4.3 conclude.

4. $\Sigma\,;\,\Gamma\vdash_{\mathcal{T}\vee\mathcal{O}} s\,:\,\sigma$, $s = l\circ\gamma$ and $t = r\circ\gamma$ for some $l \to r \in R$ such that l, r have convertible types μ, ν in their environment. By Lemma 3.1 applied repeatedly, s, t have types $\sigma = \mu\circ\gamma$ and $\tau = \nu \circ \gamma$, and since $\equiv$ is a congruence, $\sigma \equiv \tau$. By Lemma 4.6(stability), $\|l\gamma = \|l\|\gamma$ and $\|r\gamma = \|r\|\gamma$ hence $\|s \rightharpoonup_{[R]} \|t$ by definition of $\rightharpoonup_{[R]}$. ◀

This Lemma contains the analog of the type-preservation property of non-dependent rewriting, equivalence by conversion being here equivalence modulo type erasures: subject reduction holds for dependent rewriting modulo type erasures.

Thus λ^{Π}_{Σ} is terminating for an empty set R, implying strong normalisation of $\beta\eta$-reductions at both object and type level, for any convertibility relation $\equiv$ containing $\beta\eta$-convertibility. In particular, $\equiv$ can contain Presburger arithmetic, an important known extension of LF.

▶ **Example 4.8.** Here are the transformed signature and rules for our example on Lists:

$$\begin{array}{rcl||rcl||rcl}
nat, A & : & * & 0 & : & nat & _+1 & : & nat \to nat \\
List & : & nat \to * & nil & : & List & + & : & nat \to nat \to nat \\
\multicolumn{3}{r||}{cons : nat \to A \to List \to List} & \multicolumn{3}{r||}{} & app & : & nat \to nat \to List \to List \to List \\
\multicolumn{3}{r||}{0 + n \to n} & \multicolumn{6}{l}{(m+1) + n \to (m+n) + 1} \\
\multicolumn{3}{r||}{app(0, n, nil, l) \to l} & \multicolumn{6}{l}{app(m{+}1, n, cons(m, a, k), l) \to cons(m{+}n, a, app(m, n, k, l))}
\end{array}$$

These dependently typed rewrite rules being algebraic, their encoding is the identity. CPO proves their termination with $app >_{\mathcal{F}} cons >_{\mathcal{F}} nil >_{\mathcal{F}} + >_{\mathcal{F}} \_ + 1 >_{\mathcal{F}} 0$, and $List >^{\to} \{nat, A\}$.

▶ **Theorem 4.9.** *Given a signature Σ, a dependent term rewriting system $\beta\eta R$ is terminating in λ^{Π}_{Σ} if its flattening $\beta\|\beta R$ is terminating in $\lambda^{\to}_{\Sigma^{flat}}$.*

Proof. By using Lemma 4.7 and a commutation argument for η. ◀

4.4 Inverse encoding

Lemma 4.7 justifies our method for checking strong normalization by a transformation to a higher-order algebra where we can use standard techniques including CPO. But the flattened world is richer than the dependent world, there are more terms and rewrites. Nonetheless, we can also show that termination is partly preserved by defining an inverse transformation such that composing both is the identity. First, we define the inverse transformation on the subset $\|\lambda_{\Sigma} \subseteq \lambda_{\Sigma^{flat}}$, hence allowing us to show that flattening is injective.

▶ **Definition 4.10.** The inverse $\|\_^{-1} : \|\lambda_{\Sigma} \to \lambda_{\Sigma}$ is defined as:

$$\begin{array}{rcl||rcl||rcl}
\|x^{-1} & = & x & \|a^{-1} & = & a & \|f^n(s_1, ..., s_n)^{-1} & = & f^n(\|s_1{}^{-1}, ..., \|s_n{}^{-1}) \\
\|@(s,\, t)^{-1} & = & \|s^{-1}\ \|t^{-1} & & & & \|pif^2(s, \lambda x : \sigma.t)^{-1} & = & \Pi x : \|s^{-1}.\|t^{-1} \\
\|\lambda x{:}\sigma.lf^2(s,t)^{-1} & = & \lambda x{:}\|s^{-1}.\|t^{-1} & & & & \|\lambda x : \sigma.lo^2_{\tau}(s,t)^{-1} & = & \lambda x : \|s^{-1}.\|t^{-1}
\end{array}$$

▶ **Lemma 4.11** (Reversibility). *Assume* $\Sigma;\Gamma \vdash s:\sigma$ *in* λ^{Π}_{Σ}. *Then* $\Sigma;\Gamma \vdash \|(\|s)^{-1} = s:\sigma$.

Proof. By induction on s. Variables, type constants, applications and pre-algebraic terms are clear. We carry out one remaining case, the others being similar. $\|(\|\Pi x:A.B)^{-1} = \|pif^2_{|A|}(\|A, \lambda x:|A|.\|B)^{-1} = \Pi x:\|(\|A)^{-1}.\|(\|B)^{-1}$. We conclude by induction. ◀

▶ **Corollary 4.12.** *Let* $\sigma;\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} s:\sigma$ *and* $\|s|_p \in \|\lambda_\Sigma$ *for some* p. *Then* $\|s|_p = \|s|_q$ *for some* q.

Flattening being an injection from λ_Σ to $\lambda_{\Sigma^{flat}}$, hence a bijection between λ_Σ and its target $\|\lambda_\Sigma$, provides strong evidence that λ^{Π}_{Σ} is faithfully encoded by flattening. It follows that, if $\|s = \|t$ for $s,t \in \lambda_\Sigma$, then $s = t$ since $s = \|\|s^{-1} = \|\|t^{-1} = t$. We apply to Lemma 4.6:

▶ **Corollary 4.13** (inverse stability). *Assume* $u\{x \mapsto v\} \in \|\lambda_\Sigma$. *Then,* $\|u\{x \mapsto v\}^{-1} = \|u^{-1}\{x \mapsto \|v^{-1}\}$.

Therefore, rewrites in $\|\lambda_\Sigma$ can be mapped back to λ_Σ, showing both that $\|\lambda_\Sigma$ is closed under rewriting by $[\beta R]$, and that termination in λ_Σ implies termination in $\|\lambda_\Sigma$:

▶ **Lemma 4.14.** *Let* $\Sigma;\Gamma \vdash_{\mathcal{T}\vee\mathcal{O}} s:\sigma$ *and* $\|s \rightharpoonup_{[\beta R]} u$. *Then* $u = \|t$ *for some* t *st* $s \to_{\beta R} t$.

Proof. By assumption $\|s|_p = \|l\gamma \rightharpoonup_{[\beta R]} u|_p$ for some $p \in \mathcal{P}os(\|s)$, where $l \to r \in \{\beta\} \cup R$, γ a substitution in $\|\lambda_\Sigma$ and $u|_p = \|r\gamma$. By Corollary 4.12, $\|s|_p = \|s|_q$ for some q. By Lemma 4.11, $s|_q = \|\|l\gamma^{-1} = \|\|l^{-1}\|\gamma^{-1}$ (by Corollary 4.13) $= l\theta$ with $\|\theta = \gamma$ by Lemma 4.11. Hence $s \longrightarrow_{\beta R} s[r\theta]_q = t$. Now, since flattening is a homomorphism, $\|t = \|s[\|r\theta]_p$ (by definition of q) $= \|s[\|r\gamma]_p$ (by definition of θ) $= u$. ◀

This does not prove, however, that termination in λ^{Π}_{Σ} implies termination in $\lambda^{\rightarrow}_{\Sigma^{flat}}$. The problem is that flattening is non-surjective, since many terms, like those headed by lof^2, are no flattening of a dependent term. Further, some seemingly good-looking terms, like $pif^2_\sigma(s, \lambda x{:}\sigma.t)$ may not be either, even assuming that s,t are themselves flattening of dependent terms. This is the case because the flattened signature checks that term s has type $*$, hence is the encoding of some dependent type A, but cannot check whether $|A| = \sigma$ as it should. The same happens with applications and pre-algebraic terms: (s,t) may not be typable in λ^{Π}_{Σ} when s,t are typable in λ^{Π}_{Σ} and $@(\|s, \|t)$ is typable in $\lambda^{\rightarrow}_{\Sigma^{flat}}$. Indeed, termination in λ^{Π}_{Σ} does not imply termination in $\lambda^{\rightarrow}_{\Sigma^{flat}}$ as shown by this example: let o : TYPE; $List$: Πx : o.TYPE; $a, b : o$; $la : List\, a$; $lb : List\, b$; $f^2 : \Pi m : o, l : List\, m\, .\, o$; and $\{f(a, la) \to f(b, lb);\ b \to a, lb \to la\}$, whose dependent derivations are all finite since b cannot rewrite to a in $f(b, lb)$, a non-typable term. In the flattened signature, o and $List$ are sorts, and $a, b : o$; $la, lb : List$; $f^2 : o \to List \to o$; and $\{f(a, la) \to f(b, lb);\ b \to a;\ lb \to la\}$. We now have the following infinite derivation: $f(a, la) \to f(b, lb) \to f(a, lb) \to f(a, la) \ldots$. Restricting rewrites on parameters could be a solution.

5 The Dependent Computability Path Ordering

DCPO is an extension of CPO obtained by adding new cases for products. All notations for DCPO are simply and systematically obtained by replacing the CPO ordering notation $>$ by $\succ$. For example, the precedence will be denoted by $\succ_\Sigma$. Type families are compared alternatively as types with $\succeq^{\Pi}$ and as terms with $\succ$. The basic ingredients of DCPO are:

- a *precedence* $\succeq_\Sigma$ on $\mathcal{F} \cup \mathcal{S} \cup \{@, \lambda, \Pi\}$ s.t. $\succ_\Sigma$ is well-founded, and $(\forall f \in \mathcal{F}) f \succ_\Sigma @ \succ_\Sigma \{\lambda, \Pi\}$.
- a status $_f \in \{Mul, Lex\}$ for every symbol $f \in \mathcal{F} \cup \{@\}$ with $_@ \in Mul$.
- a quasi-order $\succeq^{\Pi}$ on $\mathcal{K} \cup \mathcal{T}$, whose strict part $\succ^{\Pi}$ and equivalence $=^{\Pi}$ satisfy:

- **(i)** *compatibility*: $\equiv \subseteq =^{\Pi}$;
- **(ii)** *well-foundedness*: $\succ^{\Pi} \cup \{\langle \Pi x : A.\sigma, A\rangle \mid A \in \mathcal{T},\ \sigma \in \mathcal{T} \cup \mathcal{K}\}$ is well-founded;
- **(iii)** *product body subterm*: $(\forall \sigma \in \mathcal{T} \cup \mathcal{K})\, \Pi x : A.\sigma \succ^{\Pi} \sigma$;
- **(iv)** *product preservation*: $|\Pi x\!:\!A.\sigma| {=^{\Pi}} |\tau|$ implies $\tau = \Pi x\!:\!B.\mu$ for some B, μ, such that $|A| {=^{\Pi}} |B|, |\sigma| {=^{\Pi}} |\mu|$;
- **(v)** *decreasingness*: $\Pi x\!:\!A.\sigma \succeq^{\Pi} \tau$ implies $\sigma \succeq^{\Pi} \tau$ or else $\tau = \Pi x\!:\!B.\nu$, $A {=^{\Pi}} B$ and $\tau \succeq^{\Pi} \nu$.

Building quasi-orders $\succeq^{\Pi}$ on types and kinds with a non-trivial equivalence $=^{\Pi}$ is not hard [14].

We now define a first version of DCPO which can be justified by using CPO [9]. We will then discuss briefly an enhanced version justified by a more elaborated version of CPO [10]. In the following definition, z denotes a fresh variable of type A/B.

▶ **Definition 5.1** (DCPO). Given $\Gamma \vdash_{\Sigma} s : \sigma$ and $\Gamma \vdash_{\Sigma} t : \tau$, then $s \succ^{X} t$ iff either:

1. $t \in X$ and $s \notin \mathcal{X}$ (*var*)
2. $s = (u\ N)$ and $u \succeq^{X} t$ or $N : B \succeq^{X} t : \tau$ (*subt@*)
3. $s = \lambda/\Pi x : A.u$ and $u\{x \mapsto z\} : \mu \succeq^{X} t : \tau$ (*subt*$\lambda\Pi$)
4. $s = f(\overline{s})$, $f \in \Sigma$, $t = (\lambda/\Pi y\!:\!B.v)\!:\!\Pi y\!:\!B.K$, and $s \succ^{X} B$ and $s \succ^{X \cup \{z\}} v\{y \mapsto z\}$ (Σ*prec*$\lambda\Pi$)
5. $s = \lambda x : A.u : \Pi x : A.C$, $t = \lambda y : B.v : \Pi y : B.D$, $|A| = |B|$, $A \succ^{X} B$ and $u\{x \mapsto z\} : C \succ^{X} v\{y \mapsto z\} : D$ (*stat*λ)
6. $s = \Pi x\!:\!A.u$, $t = \Pi y\!:\!B.v$, $|A| = |B|$, $A \succ^{X} B$ and $u\{x \mapsto z\} \succ^{X} v\{y \mapsto z\}$ (*stat*Π)
7. $s = ((\lambda x : A.u) : (\Pi x : A.C)\ \ w : A)$, and $u\{x \mapsto w\} \succeq t$ (*beta*)
8. otherwise $s = f(\overline{s}), t = g(\overline{t})$ with $f, g \in \mathcal{F} \cup \mathcal{S} \cup \{@\} \cup \mathcal{X}$, and either of (*rpo*)
   $\overline{s}\!:\!\overline{\sigma} \succeq t\!:\!\tau$ (*subt*) $\quad f >_{\mathcal{F}} g$ and $s \succ^{X} \overline{t}$ (*prec*) $\quad f =_{\Sigma} g$, $s \succ^{X} \overline{t}$ and $\overline{s}\!:\!\overline{\sigma}\ (\succ)_f\ \overline{t}\!:\!\overline{\tau}$ (*stat*)

All terms built by DCPO are well-typed under the assumption that both starting terms s, t are well-typed. Note (i) the importance of all our assumptions on $\succeq^{\Pi}$, see for example Case stat$\lambda\Pi$; (ii) the order may recursively compare objects with types, even when the input comparison operates on expressions in $\mathcal{T}^2 \cup \mathcal{O}^2$; (iii) compared products cannot be kinds.

5.1 Example

We now consider our list example, skipping the rules on natural numbers which do not have dependencies. We shall use the (user defined) precedence $app \succ_{\Sigma} \{+, cons, nil\}$, multiset status for app, and a quasi-order on types in which, for all $n : nat$, then $List\, n \succeq^{\Pi} A$ and $List\, m {=^{\Pi}} List\, n$ for all m, n of type nat. Such a type order is easy to get by using a restricted RPO on type erasures, which equivalence therefore contains Presburger arithmetic. See [14]. The goal $app(0, n, nil, l) : List\, 0 + n \succ l : List\, n$ is easy, although already requiring identification of $0 + n$ in $List\, 0 + n$ with n in $List\, n$. We proceed with the second goal:
$app(m{+}1, n, cons(m, a, k), l) : List\,(m{+}1){+}n \succ cons(m{+}n, a, app(m, n, k, l))\!:\!List\,(m{+}n){+}1$
The type comparison $List\,(m+1)+n \succeq^{\Pi} List\,(m+n)+1$ succeeds by using our type ordering which indeed equates both types, and we are left with the term comparison: $app(m+1, n, cons(m, a, k), l) \succ cons(m+n, a, app(m, n, k, l))$ Using *(prec)*, we get three subgoals, which are processed in turn, using indentation to identify the dependencies between recursive calls, the used Case being indicated between parentheses:

$app(m+1,n,cons(m,a,k),l) \succ m+n$ $(prec)$
$app(m+1,n,cons(m,a,k),l) \succ m$ $(subt)$
$m+1 : nat \succeq m : nat$ which yields
$nat \succeq^{\Pi} nat$ which succeeds and $m+1 \succeq m$ which succeeds by $(subt)$
$app(m+1,n,cons(m,a,k),l) \succ n$ which succeeds by $(subt)$
$app(m+1,n,cons(m,a,k),l) \succ a$ $(subt)$
$cons(m,a,k) : List\,(m+1) \succ a : A$ which yields
$List\,(m+1) \succeq^{\Pi} A$ which succeeds and $cons(m,a,k) \succ a$ which succeeds by $(subt)$
$app(m+1,n,cons(m,a,k),l) \succ app(m,n,k,l)$ $(stat)$
$\{m+1 : nat, n : nat, cons(m,a,k) : List\ m+1, l : List\ n\}$
$\succ_{mul} \{m : nat, n : nat, k : List\ m, l : List\ n\}$ which yields
$m+1 : nat \succ m : nat$ and $cons(m,a,k) : List\ m+1 \succ k : List\ m$, for the reader.

5.2 Properties of DCPO

▶ **Lemma 5.2** (Monotonicity, stability). *Let $s,t \in \lambda_\Sigma$ st $s : \sigma \succ t : \sigma$, $C(x:\sigma)$ a context, and γ a substitution. Then $C\{x \mapsto s\gamma\} \succ C\{x \mapsto t\gamma\}$.*

Proof. By induction on the definition of $s \succ t$ and stability of the type order for monotonicity. By induction on the context, and use of the status rules for stability. ◀

As anticipated, we now reduce the well-foundedness of DCPO to the well-foundedness of CPO by using Theorem 4.9, termination in the target higher-order algebra being checked by CPO. To this end, we need to show that, whenever $s : \sigma \succ t : \tau$, then $\|s : |\sigma| > \|t : |\tau|$.

We start showing that an order on types and kinds of LF satisfying the requirements for DCPO becomes naturally an order on types of $\lambda^{\rightarrow}_{\Sigma^{flat}}$ satisfying the requirements for CPO.

▶ **Definition 5.3.** Given $\sigma,\tau \in \mathcal{T}^{\rightarrow}_{\mathcal{S}_{flat}}$, let $\sigma \succeq^{\rightarrow}_{\Sigma^{flat}} \tau$ iff $\exists \mu,\nu \in \mathcal{T} \cup \mathcal{K}$, $|\mu| = \sigma$, $|\nu| = \tau$, and $\mu \succeq^{\Pi} \nu$.

Transitivity is clear. Of course, different choices of μ,ν may sometimes lead to contradictory orderings for σ,τ, hence the equality $=^{\rightarrow}_{\Sigma^{flat}}$ may be strictly larger than the corresponding equality $=^{\Pi}$. This has indeed no negative impact since the strict part $\succ^{\Pi}$ is never used in comparisons. Further, the properties of $\succeq^{\Pi}$ transfer naturally to $\succeq^{\rightarrow}_{\Sigma^{flat}}$. Soundness follows:

▶ **Lemma 5.4.** *Assume $\succeq^{\Pi}$ is a DCPO type order. Then $\succeq^{\rightarrow}_{\Sigma^{flat}}$ is a CPO type order.*

The set Σ^{flat} of function symbols of Σ^{flat} is the union of $\mathcal{S}$, $\mathcal{F}$ and the flattening constructors. The precedence $>_{\Sigma^{flat}}$ is obtained by letting the flattening constructors be equivalent minimal symbols. The strict part of $>_{\Sigma^{flat}}$ is clearly well-founded, while its equivalence is increased by the equality of the flattening constructors. We take status *Mul* for lo^2, lf^2, pif^2.

▶ **Lemma 5.5.** *Let $s:\sigma, t:\tau$ and $X \subseteq \mathcal{X}$ such that $s \succ^X t$ by any DCPO rule. Then $\|s >^X \|t$.*

Proof Sketch. The proof is by induction on the definition of DCPO, assuming by induction hypothesis that the property holds at every recursive call. Note that if $s : \sigma$, $t : \tau$, $\sigma \succeq^{\Pi} \tau$ and $\|s >^X \|t$, then $\|s : |\sigma| >^X \|t : |\tau|$ by Definition 5.3. ◀

We then obtain the second main result of the paper as a corollary:

▶ **Theorem 5.6** (Well-foundedness of DCPO). *$\succ^{\Pi}$ is well-founded.*

Proof. By theorem 4.9. Note that we use the full strength of that result, including the need for type-level rules instances of the various DCPO rules instances which compare types. This is possible since we only need preservation of non-termination by the flattening transformation, that is, the if direction of Theorem 4.9. ◀

It follows that $\succ^{\Pi} \cup \eta$ is well-founded. One may wonder why we did not include η-rules in the definition of DCPO, since it is in the definition of CPO. The reason is the comparison of lefthand and righthand sides of $[\eta]$, $\lambda x{:}|A|.lof^2(\|A,\ @(\|B,\ x)) > \|B$ which does not go through: the type comparison of the subgoal $lof^2(\|A,\ @(\|B,\ x)){:}\sigma > \|B{:}|A| \to \sigma$ fails.

5.3 A realistic example

We consider here a more complex, non-algebraic, higher-order example. Given a list l of natural numbers $x_1, ..., x_m$, and a natural number y, a higher-order variable g which is meant to be instantiated by $+$, we define a higher-order function *foldr* such that $(foldr(m, l, y)\ g)$ calculates $g(x_1, g(x_1, ...g(x_m, y)))$, while $(map(m,\ l)\ f)$ calculates the list $f(x_1), ..., f(x_m)$.

$nat : \mathrm{TYPE} \quad | \quad 0 : nat \quad | \quad +1 : \Pi x : nat.nat$
$List : \Pi m{:}nat.\mathrm{TYPE} \quad | \quad nil : List\ 0 \quad | \quad cons : \Pi\{m{:}nat,\ x{:}nat,\ l{:}List\ m\}.List\ m+1$
$map : \Pi\{m : nat,\ l : List\ m\}.(\Pi f : (\Pi x : nat.nat).List\ m)$
$foldr : \Pi\{m : nat,\ l : list\ m,\ y : nat\}.\Pi g : (\Pi\{x_1 : nat,\ x_2 : nat\}.nat).nat$

$map(0, l) \to \lambda f : (\Pi x{:}nat.nat).nil \quad | \quad foldr(0,\ l,\ y) \to \lambda g{:}\Pi\{x_1{:}nat,\ x_2{:}nat\}.nat.y$
$map(m+1,\ cons(m,\ x,\ l)) \to \lambda f : \Pi\{x : nat\}.nat.cons((f\ x),\ map(m,\ l))$
$foldr(m+1,\ cons(m,\ z,\ l),\ y) \to \lambda g : \Pi\{x_1 : nat,\ x_2 : nat\}.nat.(g\ z\ (foldr(m,\ l,\ y)\ g))$

To carry out the example, we let $List >_{\mathcal{T}} nat$, $foldr > map > cons > nil > lf^2 > lo^2 > List > nat$ and $(\forall \sigma >_{\mathcal{T}} \tau)\ lf^2_\sigma > lf^2_\tau$ and $lo^2_\sigma > lo^2_\tau$. We consider the last rule only.

CPO comparison. It generates the following goals (omitting the type comparisons):

$\|l > \lambda g : nat \to nat \to nat.lo^2_{nat}$
$(lf^2_{nat}(nat,\ \lambda x_1 : nat.lf^2_{nat}(nat,\ \lambda x_2 : nat.nat)),\ @(@(g, z), @(foldr(m,\ l,\ y)\ g)))$ $\mathcal{F}\lambda$
$\|l >^{\{g\}} lo^2_{nat}(lf^2_{nat}(nat, \lambda x_1{:}nat.lf^2_{nat}(nat, \lambda x_2{:}nat.nat)), @(@(g, z), @(foldr(m, l, y), g)))$
$\|l >^{\{g\}} lf^2_{nat}(nat,\ \lambda x_1 : nat.lf^2_{nat}(nat,\ \lambda x_2 : nat.nat))$ PREC
$\|l >^{\{g\}} nat$ succeeds by PREC
$\|l >^{\{g\}} \lambda x_1 : nat.lf^2_{nat}(nat,\ \lambda x_2 : nat.nat)$ $\mathcal{F}\lambda$
$\|l >^{\{g,x_1\}} lf^2_{nat}(nat,\ \lambda x_2 : nat.nat)$ PREC
$\|l >^{\{g,x_1\}} nat$ succeeds by PREC
$\|l >^{\{g,x_1\}} \lambda x_2 : nat.nat$ $\mathcal{F}\lambda$
$\|l >^{\{g,x_1,x_2\}} nat$ succeeds by PREC
$\|l >^{\{g\}} @(@(g, z), @(foldr(m,\ l,\ y)\ g))$ PREC
$\|l >^{\{g\}} @(g, z)$ PREC
$\|l >^{\{g\}} g$ succeeds by VAR
$\|l = foldr(m+1,\ cons(m,\ z,\ l),\ y) >^{\{g\}} z$ SUBT
$cons(m, z, l) : List(m+1) >^{\{g\}} z : nat$ which succeeds by SUBT
$\|l >^{\{g\}} @(foldr(m,\ l,\ y), g)$ PREC
$\|l = foldr(m+1,\ cons(m,\ z,\ l),\ y) >^{\{g\}} foldr(m,\ l,\ y)$ STAT
$\{m+1,\ cons(m,\ z,\ l),\ y\}(>^{\{g\}})_{mul}\{m,\ l,\ y\}$ which succeeds by repeated SUBT
$\|l >^{\{g\}} g$ which succeeds by VAR, therefore ending the computation successfully.

DCPO comparison. We now carry out the same computation with DCPO directly:

$foldr(m+1, cons(m, z, l), y) \succ \lambda g{:}\Pi\{x_1{:}nat, x_2{:}nat\}.nat.(g\ z\ (foldr(m, l, y)\ g))$ (Σprec$\lambda\Pi$)
$l \succ \Pi\{x_1 : nat,\ x_2 : nat\}.nat$ (Σprec$\lambda\Pi$)
$l \succ nat$ which succeeds by (prec)
$l \succ^{\{x_1\}} \Pi x_2 : nat.nat$ (Σprec$\lambda\Pi$)
$l \succ^{\{x_1\}} nat$ which succeeds by (prec)

$l \succ^{\{x_1,x_2\}} nat$ which succeeds by (prec)
$l \succ^{\{g\}} (g\ z\ (foldr(m,\ l,\ y)\ g))$ -we keep the @ operator implicit this time- (prec)
$l \succ^{\{g\}} g$ (var)
$l \succ^{\{g\}} z$ (subt)
$cons(m, z, l) : List(m+1) \succ^{\{g\}} z : nat$ succeeds by (subt)
$l \succ^{\{g\}} (foldr(m,\ l,\ y)\ g)\}$ (prec)
$l \succ^{\{g\}} foldr(m,\ l,\ y)$ (stat)
$\{m+1,\ cons(m,\ z,\ l),\ y\} \succ^{\{g\}} \{m, l, y\}$ which succeed by repeated (subt)
$l \succ^{\{g\}} g$ succeeds by (var), therefore ending the computation successfully.

6 Conclusion

The amount of research work targeting automatic termination is vast. Among the most popular techniques are dependency pairs, introduced by Aart and Giesl and the size-changing principle, pioneered by Neil Jones, which have been generalized to dependently-typed rules [5]. Dependent types can also be *used* to store annotations useful for proving termination [22]. Despite these proposals that recent prototypes try to combine, techniques used in Coq and Agda are still poor, as acknowledged by the authors on their websites. Using our techniques would improve this situation.

Our first contribution is a new transformation for eliminating type dependencies using a framework richer than the simply-typed λ-calculus, which provides with a natural encoding, and allows us to consider arbitrary rewrite rules, not only β- and η-reductions. Furthermore, these results hold for a practical dependent type system made richer than LF's via a convertibility relation possibly stronger than $\beta\eta$-convertibility, hence allowing us to type many more terms. This easily implementable transformation allow us using existing implementations targeting termination of rules in presence of simple types. The transformation also allows us to show well-foundedness of DCPO, a version of CPO applying to dependently typed terms directly. This is done by considering pairs ordered by DCPO as dependently typed rewrite rules to which the transformation applies. Note that DCPO will naturally benefit from improvements of CPO, without changing the proof technique. In particular, we could easily accommodate size interpretations by using them as a precedence as in [11], or type level rules such as $s = (A\ u)$, $t = \Pi y : B.v$, with $s \succ^X B$ and $s \succ^{X\cup\{z\}} v\{y \mapsto z\}$ flattened as $\|s = @(\|A, \|u)$, and $\|t = pik_{|B|}(\|B, \lambda y : |B|.\|v)$, with $\|s >^X \|B$ and $\|s >^{X\cup\{z\}} \|v\{y \mapsto z\}$, whose justification requires the extension of CPO with *small* function symbols, here $pik_{|B|}$, which behave as if they were smaller than application and abstraction [10]. We can then break the main goal into the above solvable subgoals.

Our main interest is indeed to prove termination directly at the dependent type level. Using DCPO allows the programmer, in case of failure, to get an error message in her/his own dependently typed syntax, rather than in the transformed syntax as would be the case when using CPO on the transformed rules. To our knowledge, this is the very first general – Coq and Agda's techniques are very limited –, purely syntactic method that allows one to show termination of a set of dependently typed rewrite rules via computations taking place on the user's dependently typed rules.

Acknowledgements. Work supported by NSFC grants 61272002, 91218302, 973 Program 2010CB328003 and Nat. Key Tech. R&D Program SQ2012BAJY4052 of China. This work was done in part during a visit of the first author supported by Tsinghua University.

References

1 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.

2 H. P. Barendregt. Lambda calculus with types. In *Handbook of Logic in Computer Science, Volume 2*, pages 117–309. Oxford Univ. Press, 1993.

3 B. Barras, J.-P. Jouannaud, P.-Y. Strub, and Q. Wang. CoqMTU: A higher-order type theory with a predicative hierarchy of universes parametrized by a decidable first-order theory. In *LICS*, pages 143–151. IEEE Computer Society, 2011.

4 G. Barthe. The relevance of proof-irrelevance. In *ICALP*, volume 1443 of *LNCS*, pages 755–768. Springer, 1998.

5 F. Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In *RTA*, volume 3091 of *LNCS*, pages 24–39. Springer, 2004.

6 F. Blanqui. Definitions by rewriting in the calculus of constructions. *CoRR*, /abs/cs/0610065, 2006.

7 F. Blanqui, J.-P. Jouannaud, and M. Okada. The calculus of algebraic constructions. In *RTA*, volume 1631 of *LNCS*, pages 301–316. Springer, 1999.

8 F. Blanqui, J.-P. Jouannaud, and A. Rubio. HORPO with computability closure : A reconstruction. In *LPAR*, volume 4790 of *LNCS*. Springer, 2007.

9 F. Blanqui, J.-P. Jouannaud, and A. Rubio. The computability path ordering: The end of a quest. In *CSL*, *LNCS* 5213. 2008.

10 F. Blanqui, J.-P. Jouannaud, and A. Rubio. The computability path ordering. To appear in LMCS.

11 C. Borralleras and A. Rubio. A monotonic higher-order semantic path ordering. In *LPAR*, volume 2250 of *LNCS*, pages 531–547. Springer, 2001.

12 Nachum Dershowitz. Orderings for term-rewriting systems. *TCS*, 17:279–301, 1982.

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

14 J.-P. Jouannaud and A. Rubio. Polymorphic higher-order recursive path orderings. *J. ACM*, 54(1), 2007.

15 K. Kusakari, Y. Isogai, M. Sakai, and F. Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. *CoRR*, abs/1109.5468, 2011.

16 U. Norell. Dependently typed programming in Agda. In *TLDI*, pages 1–2. ACM, 2009.

17 C. Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In *TLCA*, volume 664 of *LNCS*, pages 328–345. Springer, 1993.

18 F. Pfenning. Elf: A language for logic definition and verified metaprogramming. otoIn *In Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 313–322. 1989.

19 P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, pages 159–169. ACM, 2008.

20 P.-Y. Strub. Coq modulo theory. In *CSL*, volume 6247 of *LNCS*, pages 529–543. Springer, 2010.

21 S. Suzuki, K. Kusakari, and F. Blanqui. Argument filterings and usable rules in higher-order rewrite systems. *CoRR*, abs/1109.4357, 2011.

22 H. Xi. Dependent ML an approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, 2007.

23 H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227. ACM, 1999.