Refinement Types as Higher-Order Dependency Pairs

Cody Roux[1]

1 **INRIA - Lorraine**
615 r. du Jardin Botanique, 54600 Villers-lès-Nancy, France

Abstract

Refinement types are a well-studied manner of performing in-depth analysis on functional programs. The dependency pair method is a very powerful method used to prove termination of rewrite systems; however its extension to higher-order rewrite systems is still the subject of active research. We observe that a variant of refinement types allows us to express a form of higher-order dependency pair method: from the rewrite system labeled with typing information, we build a *type-level approximated dependency graph*, and describe a type level *embedding preorder*. We describe a syntactic termination criterion involving the graph and the preorder, which generalizes the *simple projection criterion* of Middeldorp and Hirokawa [21], and prove our main result: if the graph passes the criterion, then every well-typed term is strongly normalizing.

Keywords and phrases Dependency Pairs, Higher-Order, Refinement Types

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.299

Category Regular Research Paper

1 Introduction

Types are used to perform static analysis on programs. Various type systems have been developed to infer information about termination, run-time complexity, or the presence of uncaught exceptions.

We are interested in one such development, namely *dependent types* [30, 13]. Dependent types explicitly allow "object level" terms to appear in the types, and may be used to fully specify (extensional) program behavior using the so called *Curry-Howard isomorphism*. We are particularly interested here in *refinement types* [36, 17]. For a given base type B and a property P on programs, we may form a type R which is a *refinement of* B and which is intuitively given the semantics:

$$R = \{t : B \mid P(t)\}.$$

Programing languages based on dependent type systems have the reputation of being unwieldy, due to the perceived weight of proof obligations in heavily specified types. The field of dependently typed programing can be seen as a quest to find the compromise between expressivity of types and ease of use for the programmer. In this paper we propose a type system which we believe achieves such a compromise for a termination analysis based on the shape of constructor terms.

Dependency pairs are a highly successful technique for proving termination of first-order rewrite systems [4]. However, it is difficult to apply the method to higher-order rewrite systems. Indeed, the data-flow of such systems is significantly different than that of first-order ones. Let us examine the rewrite rule:

$$f\ (S\ x) \to (\lambda y.f\ y)\ x$$

22nd International Conference on Rewriting Techniques and Applications.
Editor: M. Schmidt-Schauß; pp. 299–312
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Which can for example be written in the higher-order rewriting framework of Jouannaud & Okada [26]. The termination of well-typed terms under this rewrite system combined with β-reduction cannot be inferred by simply looking at the left-hand side $f\ (S\ x)$ and the recursive call $f\ y$ in the right hand side as it could be in first-order rewriting. Here we need to infer that the variable y can only be instantiated by a subterm of $S\ x$. This can be done using dependent types, using a framework called *size-based termination* or sometimes *type-based termination* [22, 1, 5, 7, 10].

The dependency pair method rests on the examination of the aptly-named *dependency pairs*, which correspond to left-hand sides of rules and function calls with their arguments in the right-hand side of the rules. For instance with a rule

$$f(c(x,y),z) \to g(f(x,y))$$

We would have two dependency pairs, the pair $f(c(x,y),z) \to f(x,y)$ and the pair $f(c(x,y),z) \to g(f(x,y))$ (in the case that g is a defined symbol).

We can then define a *chain* to be a pair (θ,ϕ) of substitutions, and a couple $(t_1 \to u_1, t_2 \to u_2)$ of dependency pairs such that $u_1\theta$ reduces to $t_2\phi$. We may connect chains in an intuitive manner, and the fundamental theorem of dependency pairs may be stated: *a (first-order) rewrite system is terminating if and only if there are no infinite chains.* See also the original article [4] for details.

To prove that no infinite chains exist, one wants to work with the *dependency graph*: the graph built using the dependency pairs as nodes and with a vertex between $N_1 = t_1 \to u_1$ and $N_2 = t_2 \to u_2$ if there exist θ and ϕ such that $(\theta,\phi),(N_1,N_2)$ form a chain. It is then shown that if the system is finite, then it is sufficient to consider only the cycles in this graph and prove that they may not lead to infinite chains [18]. It is known that in general computing the dependency graph is undecidable (this is the *unification modulo rewriting* problem, see *e.g.* Jouannaud *et al* . [25]), so in practice we compute an approximation (or estimation) of the graph that is *conservative*: all edges in the dependency graph are sure to appear in the approximated graph. One common (see for instance Giesl [20]) and reasonable approximation is to perform ordinary unification on non-defined symbols (that is, symbols that are not at the head of a left-hand side), while replacing each subterm headed by a defined symbol by a fresh variable, ensuring that it may unify with any other term.

In this article, we show that the dependency pair technique with the approximated dependency graph can be modeled using a form of refinement types containing *patterns* which denote sets of possible values to which a term reduces. The syntax and type system are described in section 2. These type-patterns must be explicitly abstracted and applied, a choice that allows us to have very simple type inference. We then use these types to build a notion of type-based dependency pair for higher-order rewrite rules, as well as an approximated dependency graph which corresponds to the estimation described above. We describe an order on the type annotations, that essentially captures the subterm ordering, and use this order to express a *decrease condition* along cycles in the approximated dependency graph. In section 3 we describe a suitable generalization of the *simple projection criterion* first described by Middeldorp and Hirokawa [21]: if in every *strongly connected component* of the graph and every cycle in the component, the decrease condition holds, then every well-typed term is strongly normalizing under combination of the rewrite rules and β-reduction. The actual operational semantics are defined not on the terms themselves, but on *erased terms* in which we remove the explicit type information. Section 4 concludes with a comparison with other approaches to higher-order dependency pairs and possible extensions of our criterion.

2 Syntax and Typing Rules

The language we consider is simply a variant of the λ-calculus with constants. For simplicity we only consider the datatype of binary (unlabeled) trees. The development may be generalized without difficulty to other first-order datatypes, *i.e.* types whose constructors do not have higher-order recursive arguments. We define the syntax of *patterns*

$$p, q \in \mathcal{P} \coloneqq \alpha \mid \_ \mid \bot \mid \text{leaf} \mid \text{node}(p, q)$$

With $\alpha \in \mathcal{V}$ a set of *pattern variables*, and _ is called *wildcard*. Patterns appear in types to describe possible reducts of terms. We define the set of types:

$$T, U \in \mathcal{T} \coloneqq \boldsymbol{B}(p) \mid T \to U \mid \forall \alpha.T$$

An *atomic type* is a type of the form $\boldsymbol{B}(p)$. The set of terms of our language is defined by:

$$t, u \in \mathcal{T}rm \coloneqq x \mid f \mid t\ u \mid t\ p \mid \lambda x\!:\!T.t \mid \lambda \alpha.t \mid \text{Leaf} \mid \text{Node}$$

With $x \in \mathcal{X}$ a set of term variables, $f \in \Sigma$ is a set of *defined function symbols* and $\alpha \in \mathcal{V}$.

A *constructor* is either Leaf or Node. A *context* is a list of judgements $x\!:\!T$ with $x \in \mathcal{X}$ and $T \in \mathcal{T}$, with each variable appearing only once. Notice that application and abstraction of patterns is explicit.

Intuitively, $\boldsymbol{B}(p)$ denotes the set of terms for which *every* reduct in normal form *matches* the pattern p. For instance, any binary tree t is in the semantics of $\boldsymbol{B}(\_)$, only binary trees that reduce to Node $t_1\ t_2$ for some binary trees t_1 and t_2 are in $\boldsymbol{B}(\text{node}(\_, \_))$, and only terms that *never* reduce to a constructor are in $\boldsymbol{B}(\bot)$. Our operational semantics is defined by rewriting, which has the following consequences, which may be surprising to a programming language theorist:

- It may be the case that a term t has several distinct normal forms. Indeed we do not require our system to be orthogonal, or even confluent (we do require it to be finitely branching though). Therefore a term is in the semantics of $\boldsymbol{B}(\text{node}(\_, \_))$ if *all* its reducts reduce to a term of the form Node $t\ u$.
- It is possible for a term to be *stuck* in the empty context, that is in normal form and not headed by a constructor or an abstraction. Therefore $\boldsymbol{B}(\bot)$ is not necessarily empty even in the empty context.

We write $\mathcal{FV}(t)$ (resp. $\mathcal{FV}(T)$, $\mathcal{FV}(\Gamma)$) for the set of free (type or term) variables in a term t (resp. a type T, a context Γ). If a term (resp. pattern) does not contain any free variables, we say that it is *closed*. We write $\forall \vec{\alpha}.T$ for $\forall \alpha_1.\forall \alpha_2 \ldots \forall \alpha_n.T$, and arrows and application are associative to the left and right respectively, as usual. A pattern variable α appears in $\boldsymbol{B}(p)$ if it appears in p. It appears *positively* in a type T if:

- $T = \boldsymbol{B}(p)$ and α appears in p
- $T = T_1 \to T_2$ and α appears positively in T_2 or negatively in T_1 (or both),

with α appearing *negatively* in T if $T = T_1 \to T_2$ and α appears negatively in T_2 or positively in T_1 (or both).

We consider a *type assignment* $\tau\!:\!\Sigma \to \mathcal{T}$, such that for each $f \in \Sigma$, there is a number k such that τ_f is of the form

$$\tau_f = \forall \alpha_1, \ldots, \alpha_k.\boldsymbol{B}(\alpha_1) \to \ldots \to \boldsymbol{B}(\alpha_k) \to T_f$$

and each α_i may appear only *positively* in T_f. In this case k is called the number of *recursive arguments*.

$$\frac{}{\Gamma, x:T, \Delta \vdash x:T}\ \textbf{ax}$$

$$\frac{\Gamma, x:T \vdash t:U}{\Gamma \vdash \lambda x:T.t:T \to U}\ \textbf{t-lam} \qquad \alpha \notin \mathcal{FV}(\Gamma)\ \frac{\Gamma \vdash t:T}{\Gamma \vdash \lambda\alpha.t:\forall\alpha.T}\ \textbf{p-lam}$$

$$\frac{}{\Gamma \vdash \mathrm{Leaf}:\boldsymbol{B}(\mathrm{leaf})}\ \textbf{leaf-intro}$$

$$\frac{}{\Gamma \vdash \mathrm{Node}:\forall\alpha\beta.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\beta) \to \boldsymbol{B}(\mathrm{node}(\alpha,\beta))}\ \textbf{node-intro}$$

$$\frac{\Gamma \vdash t:T \to U \qquad \Gamma \vdash u:T}{\Gamma \vdash t\ u:U}\ \textbf{t-app} \qquad \frac{\Gamma \vdash t:\forall\alpha.T}{\Gamma \vdash t\ p:T\{\alpha \mapsto p\}}\ \textbf{p-app}$$

$$\frac{}{\Gamma \vdash f:\tau_f}\ \textbf{symb}$$

Figure 1 Typing Rules

The positivity condition is quite similar to the one used in the usual formulation of type-based termination, see for instance Abel [2] for an in depth analysis. The typing rules are also similar to the ones for type-based termination. The typing rules of our system are given by the typing rules in Figure 1.

To these rules we add the subtyping rule:

$$\frac{\Gamma \vdash t:T \qquad T \leq U}{\Gamma \vdash t:U}\ \textbf{sub}$$

Where the subtyping relation is defined, first on patterns, then on types by:

$$\frac{}{p \ll \_} \qquad \frac{}{\bot \ll p}$$

$$\frac{}{\alpha \ll \alpha} \qquad \frac{}{\mathrm{leaf} \ll \mathrm{leaf}}$$

$$\frac{p_1 \ll q_1 \qquad p_2 \ll q_2}{\mathrm{node}(p_1,p_2) \ll \mathrm{node}(q_1,q_2)}$$

$$\frac{p \ll q}{\boldsymbol{B}(p) \leq \boldsymbol{B}(q)}$$

$$\frac{T_2 \leq T_1 \qquad U_1 \leq U_2}{T_1 \to U_1 \leq T_2 \leq U_2}$$

$$\frac{T \leq U}{\forall\alpha.T \leq \forall\alpha.U}$$

This type system is similar to the refinement types described by Freeman *et al* . [17], for a subset of the ML language. However they consider more complex refinements in which arbitrary unions are allowed (and for which type checking is undecidable!) and which does not allow one to explicitly *name* the shape of a term in the type, *i.e.* it does not allow (our version of) type-level variables. Furthermore, our system is not very distant from *generalized algebraic datatypes* as are implemented in certain Haskell extensions [33], though subtyping is not present in that framework.

$$\frac{}{\Gamma, x : \boldsymbol{B}(\alpha), \Gamma' \vdash_{\min} x : \boldsymbol{B}(\alpha)}\ \alpha \notin \Gamma, \Gamma'$$

$$\frac{}{\Gamma \vdash_{\min} \text{Leaf} : \boldsymbol{B}(\text{leaf})}$$

$$\frac{\Gamma \vdash_{\min} c_1 : \boldsymbol{B}(p_1) \qquad \Gamma \vdash_{\min} c_2 : \boldsymbol{B}(p_2)}{\Gamma \vdash_{\min} \text{Node}\ \ p_1\ p_2\ c_1\ c_2 : \boldsymbol{B}(\text{node}(p_1, p_2))}$$

$$\frac{\Gamma \vdash_{\min} c_1 : \boldsymbol{B}(p_1) \qquad \ldots \qquad \Gamma \vdash_{\min} c_k : \boldsymbol{B}(p_k)}{\Gamma \vdash_{\min} f\ p_1 \ldots p_k\ c_1 \ldots c_k : T_f \phi}\ \vec{\alpha} \notin \Gamma$$

With $\tau_f = \forall \alpha_1 \ldots \alpha_k . \boldsymbol{B}(\alpha_1) \to \ldots \to \boldsymbol{B}(\alpha_k) \to T_f$ and $\phi(\alpha_i) = p_i$ for $1 \leq i \leq k$.

Figure 2 Minimal Typing Rules

It may seem surprising that we choose to represent pattern abstraction (by $\lambda \alpha . t$), and pattern application (by $t\ p$) explicitly in our system. This choice is justified by the simplicity of type inference with explicit parameters. In the author's opinion, implicit arguments should be handled by the following schema: at the user level a language without implicit parameters; these parameters are inferred by the compiler, which type-checks a language with all parameters present. Then at run-time they are once again erased. This is exactly analogous to a Hindley-Milner type language in which System F is used as an intermediate language [31, 24]. It is also our belief that explicit parameters will allow this criterion to be more easily integrated into languages with pre-existing dependent types, *e.g.* Agda [32], Epigram [29] or Coq [15].

A *constructor term* $c \in \mathcal{C}$ is a term built following the rules:

$$c_1, c_2 \in \mathcal{C} \coloneqq x \mid \text{Leaf} \mid \text{Node}\ \ p_1\ p_2\ c_1\ c_2$$

with $x \in \mathcal{X}$.

A rewrite rule is a pair of terms (l, r) which we write $l \to r$, such that l is of the form $f\ p_1 \ldots p_n\ c_1 \ldots c_k$ with $f \in \Sigma$, $p_i \in \mathcal{P}$ and $c_i \in \mathcal{C}$, and k is the number of recursive arguments of f. We suppose that the free variables of r appear in l. Note that there is no linearity restriction on the left-hand sides of rules, and that left-hand sides may not contain any abstractions.

We suppose in addition that every function symbol $g \in r$ is *fully applied* to its pattern arguments, that is if $\tau_g = \forall \alpha_1 \ldots \alpha_l . T$ then for each occurrence of g in r there are patterns $p_1, \ldots, p_l \in \mathcal{P}$ such that $g\ p_1 \ldots p_l$ appears at that position.

In the following we consider a *finite* set $\mathcal{R}$ of rewrite rules. The set $\mathcal{R}$ is *well-typed* if for each rule $l \to r \in \mathcal{R}$, there is a context Γ and a type T such that

$$\Gamma \vdash_{min} l : T$$

and

$$\Gamma \vdash r : T$$

with $\vdash_{\min}$ the *minimal typing relation* defined in Figure 2.

Notice that if $\Gamma \vdash_{\min} c_i : T$ then T is *unique*. Minimal typing is related to other work on size-based termination [11], in which it is called the *pattern condition*. Its purpose is to constrain the possible types of constructor terms in left-hand sides, so that the type gives an *exact* semantics of the matched terms. In particular, if $\Gamma \vdash_{min} x : \boldsymbol{B}(p)$, then $p = \alpha$, $x : \boldsymbol{B}(\alpha)$ and α may *not* appear in the type of any other variable. Furthermore, subtyping is forbidden, so that a constructor term of the form Node $p\ q\ t\ u$ is given a type of the form $\boldsymbol{B}(\text{node}(p, q))$, and Leaf is given the type $\boldsymbol{B}(\text{leaf})$.

We give the following theorem without proof.

▶ **Theorem 2.1.** *Type checking is* decidable*: there is a procedure which, given* Γ, t *and* T*, decides whether*

$$\Gamma \vdash t : T$$

is derivable.

It may be useful to note here that subtyping is necessary to type all but the most trivial of programs: let $f : \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\_)$ be the function that computes the mirror image of a binary tree, which may be defined in our system by the rules

$$\begin{aligned} f\ \text{leaf}\ \text{Leaf} &\to \text{Leaf} \\ f\ \text{node}(\alpha,\beta)\ (\text{Node}\ \alpha\ \beta\ x\ y) &\to \text{Node}\ \_\ \_\ (f\ \beta\ y)\ (f\ \alpha\ x) \end{aligned}$$

This function is well-typed using the minimal typing rules in the context $x : \boldsymbol{B}(\alpha),\ y : \boldsymbol{B}(\beta)$, but subtyping is necessary to type the second rule, as the term $\text{Node}\ \_\ \_\ (f\ \beta\ y)\ (f\ \alpha\ x)$ has type $\text{node}(\_,\_)$ and not $\_$ which is the required return type of f.

We can then define a higher-order analogue of dependency pairs, which uses type information instead of term information.

▶ **Definition 2.2.** *Let* $\rho = f\ \vec{p}\ \vec{c} \to r$ *be a rule in* $\mathcal{R}$*, with* Γ *such that* $\Gamma \vdash_{min} f\ \vec{p}\ \vec{c} : T$*, and* $\Gamma \vdash r : T$*. The set of* type dependency pairs $DP_{\mathcal{T}}(\rho)$ *is the set*

$$\left\{ f^\sharp(p_1,\dots,p_k) \to g^\sharp(q_1,\dots,q_l) \mid \forall i, \Gamma \vdash_{\min} c_i : \boldsymbol{B}(p_i) \wedge g\ q_1 \dots q_l \text{ appears in } r \right\}$$

The set $DP_{\mathcal{T}}(\mathcal{R})$ *is defined as the union of all* $DP_{\mathcal{T}}(\rho)$*, for* $\rho \in \mathcal{R}$*, where we suppose that all variables are disjoint between dependency pairs.*

The set of higher-order dependency pairs defined above should already be seen as an abstraction of the traditional dependency pair notion (for example those defined in [4]). Indeed, due to subtyping, there may be some information lost in the types, if for instance the wildcard pattern is used. As an example, if f, g and h all have type $\forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\_)$, consider the rule

$$f\ \alpha\ x \to g\ \_\ (h\ \alpha\ x)$$

The dependency pair we obtain is

$$f^\sharp(\alpha) \to g^\sharp(\_)$$

The information that g is called on the argument $h\ \alpha\ x$ is lost.

This approach can therefore be seen as a type-based manner to study an approximation of the dependency graph. Note that in the case where h is given a more precise type, like $\forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\text{leaf})$, which is the case if every normal form of $h\ p\ t$ is either neutral or Leaf, we have a more precise approximation.

Note that in the following definition, a dependency pair can not be easily seen as a rule of the system itself, though it may be seen as a first-order rewrite rule which operates on "type level" function symbols and constructors.

▶ **Definition 2.3.** *Let p and q be patterns. We define* pattern-unification *as the smallest relation verifying:*

$$\frac{}{p \bowtie \_} \qquad \frac{}{p \bowtie \alpha}$$

$$\frac{}{\text{leaf} \bowtie \text{leaf}} \qquad \frac{}{\bot \bowtie \bot} \qquad \frac{p \bowtie q}{q \bowtie p}$$

$$\frac{p_1 \bowtie q_1 \qquad p_2 \bowtie q_2}{\text{node}(p_1, p_2) \bowtie \text{node}(q_1, q_2)}$$

The standard typed dependency graph $\mathcal{G}_{\mathcal{R}}$ *is defined as the graph with*

- *As set of nodes the set $DP_{\mathcal{T}}(\mathcal{R})$.*
- *An edge from the dependency pair $t \to g^\sharp(p_1, \ldots, p_k)$ to $g^\sharp(q_1, \ldots, q_k) \to u$ if for every $1 \leq i \leq k, p_i \bowtie q_i$.*

This definition gives us an adequate higher-order notion of standard approximated dependency graph. We will now show that it is possible to give an order on the terms in the dependency pairs, which is similar to a simplification order and which will allow us to show termination of well-typed terms under the rules, if the graph satisfies an intuitive decrease criterion.

▶ **Definition 2.4.** *Given patterns p and q which do* not *contain $\_$, we define the* embeddeding preorder *on $\mathcal{P}$ written $p \rhd q$ by the following rules*

- $p_i \trianglerighteq q \Rightarrow \text{node}(p_1, p_2) \rhd q$ *for* $i = 1, 2$
- $p_1 \rhd q_1 \wedge p_2 \trianglerighteq q_2 \Rightarrow \text{node}(p_1, p_2) \rhd \text{node}(q_1, q_2)$
- $p_1 \trianglerighteq q_1 \wedge p_2 \rhd q_2 \Rightarrow \text{node}(p_1, p_2) \rhd \text{node}(q_1, q_2)$

With $\trianglerighteq$ as the reflexive closure of $\rhd$.

The preorder $\rhd$ can be used to verify a structural decrease in values: if $t : \boldsymbol{B}(p)$, $u : \boldsymbol{B}(q)$ in a common context and $p \rhd q$, then the maximum size of normal forms of t is strictly greater that the maximum size of normal forms in u. This explains why we must forbid $\_$ in the definition of $\rhd$, as a term can be simultaneously typed in $\boldsymbol{B}(\text{node}(\_, \_))$ and $\boldsymbol{B}(\_)$ (and so no decrease is possible).

Non termination can intuitively be traced to cycles in the dependency graph. We wish to consider termination on terms with erased pattern arguments and type annotations.

3 Operational Semantics and the Main Theorem

Rewriting needs to be performed over terms with erased pattern annotations. The problem with the naïve definition of rewriting arises when trying to match on patterns. Take the rule

$$f \;\; \text{node}(\alpha, \beta) \;\; (\text{Node} \;\; x \;\; y) \to \text{Leaf}$$

In the presence of this rule, we wish to have, for instance, the reduction

$$f \;\; \_ \;\; (\text{Node} \;\; (g \;\; x) \;\; (h \;\; x)) \to \text{Leaf}$$

where g and h are arbitrary defined symbols. However, there is no substitution θ such that $\text{node}(\alpha, \beta)\theta = \_$. There are two ways to deal with this. Either we take subtyping into account

when performing matching, or we erase the pattern arguments when performing reduction. We adopt the second solution, which has the advantage of requiring fewer reductions, and is closer to practice in languages with dependent type annotations (see for example McKinna [30]). Symmetrically, we erase pattern abstractions as well.

▶ **Definition 3.1.** *We define the set of* erased terms $\mathcal{T}rm^{|\cdot|}$ *as:*

$$t, u \in \mathcal{T}rm^{|\cdot|} \coloneqq x \mid f \mid \lambda x.t \mid t\ u \mid \text{Leaf} \mid \text{Node}$$

Where $x \in \mathcal{X}$ and $f \in \mathcal{F}$.

Given a term $t \in \mathcal{T}rm$, we define the erasure $|t| \in \mathcal{T}rm^{|\cdot|}$ *of t as:*

$$\begin{array}{ll} |x| & = x \\ |f| & = f \\ |\lambda x\!:\!T.t| & = \lambda x.|t| \\ |\lambda\alpha.t| & = |t| \\ |t\ u| & = |t|\ |u| \\ |t\ p| & = |t| \\ |\,\text{Leaf}\,| & = \text{Leaf} \\ |\,\text{Node}\,| & = \text{Node} \end{array}$$

An erased term can intuitively be thought of as the compiled form of a well typed term.

▶ **Definition 3.2.** *An erased term t* head rewrites *to a term u if there is some rule $l \to r \in \mathcal{R}$ and some substitution σ from $\mathcal{X}$ to terms in $\mathcal{T}rm^{|\cdot|}$ such that*

$$|l|\sigma = t \wedge |r|\sigma = u$$

We define β-reduction $\to_\beta$ as

$$\lambda x.t\ u \to_\beta t\{x \mapsto u\}$$

And we define the reduction $\to$ *as the closure of head-rewriting and β-reduction by term contexts. We then define $\to^*$ and $\to^+$ as the symmetric transitive and transitive closure of $\to$, respectively.*

We can now express our termination criterion. We need to consider the *strongly connected components*, or SCCs of the typed dependency graph. A strongly connected component of a graph $\mathcal{G}$ is a full subgraph such that each node is reachable from all the others.

▶ **Definition 3.3.** *Let $\mathcal{G}$ be the typed dependency graph for $\mathcal{R}$ and let $\mathcal{G}_1, \ldots, \mathcal{G}_n$ be the SCCs of $\mathcal{G}$. Suppose that for each $\mathcal{G}_i$, there is a* simple projection $\iota^i : \Sigma \to \mathbb{N}$ *which to $f \in \Sigma$ associates an integer $1 \leq \iota^i_f \leq k$ (with k the number of recursive arguments of f).*

We say that $\mathcal{R}$ passes the simple projection criterion *for ι if*

- *For each $1 \leq i \leq n$ and each rule $f^\sharp(p_1, \ldots, p_n) \to g^\sharp(q_1, \ldots, q_m)$ in $\mathcal{G}_i$, we have $p_{\iota^i_f} \trianglerighteq q_{\iota^i_g}$.*
- *For each cycle in $\mathcal{G}_i$, there is some rule $f^\sharp(p_1, \ldots, p_n) \to g^\sharp(q_1, \ldots, q_m)$ such that*

$$p_{\iota^i_f} \triangleright q_{\iota^i_g}$$

▶ **Theorem 3.4. (Main theorem)**
Suppose that there is ι such that $\mathcal{R}$ passes the simple projection criterion for ι. Then for every Γ, t, T such that $\Gamma \vdash t : T$,

$$|t| \in \mathcal{SN}_\mathcal{R}$$

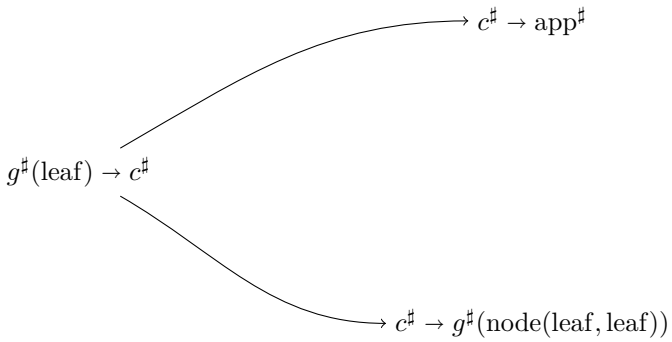


Figure 3 Dependency graph of Example 1

The proof of this theorem uses classic computability methods, and can be found in the online version, from the authors homepage. Let us give two examples of the application of this technique.

▶ **Example 1.** *Take the rewrite system given by the signature:*

$$\text{app} : \forall\alpha\beta.(\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\beta)) \to \boldsymbol{B}(\alpha) \to \boldsymbol{B}(\beta), \quad c : \boldsymbol{B}(\text{leaf}), \quad g : \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\text{leaf})$$

We give the rewrite rules:

$$\text{app} \ \to \lambda\alpha\beta.\lambda x : \boldsymbol{B}(\alpha) \to \boldsymbol{B}(\beta).\lambda y : \boldsymbol{B}(\alpha).x \ y$$

$$c \to \text{app} \ \ \text{node}(\text{leaf}, \text{leaf}) \ \ \text{leaf} \ \ (g \ \ \text{node}(\text{leaf}, \text{leaf})) \ \ (\text{Node}\, \text{leaf}\, \text{leaf} \ \ \text{Leaf} \ \ \text{Leaf})$$

$$g \ \ \text{node}(\alpha, \beta) \ \ (\text{Node}\, \alpha \ \ \beta \ \ x \ \ y) \to \text{Leaf}$$

$$g \ \ \text{leaf} \ \ \text{Leaf} \to c$$

or, in more readable form with pattern arguments and type annotations omitted:

$$\begin{aligned} \text{app} &\to \lambda x.\lambda y.x \ y \\ c &\to \text{app} \ \ g \ \ (\text{Node} \ \ \text{Leaf} \ \ \text{Leaf}) \\ g \ \ (\text{Node} \ \ x \ \ y) &\to \text{Leaf} \\ g \ \ \text{Leaf} &\to c \end{aligned}$$

It is possible to verify that the criterion can be applied and that in consequence, according to Theorem 3.4, all well typed terms are strongly normalizing under $\mathcal{R} \cup \beta$.

Indeed, we may easily check that each of these rules is minimally typed in some context. Furthermore, we can check that the dependency graph in Figure 3 has no cycles.

One may object that if we inline the definition of app and perform β-reduction on the right-hand sides of rules we obtain a rewrite system that can be treated with more conventional methods, such as those performed by the `AProVe` tool [19] (on terms without abstraction, and without β-reduction). However this operation can be very costly if performed automatically and is, in its most naïve form, ineffective for even slightly more complex higher-order programs such as *map*, which performs pattern matching and for which we need to instantiate. By resorting to typing, we allow termination to be proven using only "local" considerations, as the information encoding the semantics of app is contained in its type.

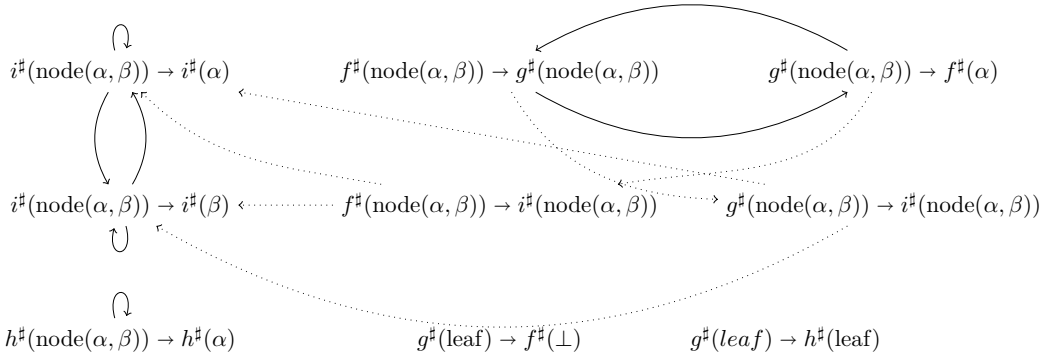


Figure 4 The dependency graph for Example 2

However it becomes necessary, if one desires a fully automated termination check on an unannotated system, to somehow infer the type of defined constants, and possibly perform an analysis quite similar in effect to the one proposed above. We believe that to this end one may apply known type inference technology, such as the one described in [14], to compute these annotated types. In conclusion, what used to be a termination problem becomes a type inference problem, and may benefit from the knowledge and techniques of this new community, as well as facilitate integration of these techniques into type-theoretic based proof assistants like Coq [15].

Let us examine a second, slightly more complex example, in which there is "real" recursion.

▶ **Example 2.** *Let $\mathcal{R}$ be the rewrite system defined by*

$$\begin{aligned}
f\ (\mathrm{Node}\ x\ y) &\to g\ (i\ (\mathrm{Node}\ x\ y)\\
g\ (\mathrm{Node}\ x\ y) &\to f\ (i\ x)\\
g\ \mathrm{Leaf} &\to f\ (h\ \mathrm{Leaf})\\
i\ (\mathrm{Node}\ x\ y) &\to \mathrm{Node}\ (i\ x)\ (i\ y)\\
i\ \mathrm{Leaf} &\to \mathrm{Leaf}\\
h\ (\mathrm{Node}\ x\ y) &\to h\ x
\end{aligned}$$

Again with the type arguments omitted for readability, and with types $f, g : \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\_)$, $h : \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\bot)$ and $i : \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\alpha)$. Every equation can be typed in the context $\Gamma = x : \boldsymbol{B}(\alpha), y : \boldsymbol{B}(\beta)$. The system with full type annotations is given in the appendix (or in the online version). The dependency graph is given in Figure 4, and has as SCCs the full subgraphs of $\mathcal{G}_\mathcal{R}$ with nodes $\{i^\sharp(\mathrm{node}(\alpha,\beta)) \to i^\sharp(\alpha), i^\sharp(\mathrm{node}(\alpha,\beta)) \to i^\sharp(\beta)\}$, $\{f^\sharp(\mathrm{node}(\alpha,\beta)) \to g^\sharp(\mathrm{node}(\alpha,\beta), g^\sharp(\mathrm{node}(\alpha,\beta) \to f^\sharp(\alpha)\}$ and $\{h^\sharp(\mathrm{node}(\alpha,\beta)) \to h^\sharp(\alpha)\}$ respectively.

Taking $\iota_s = 1$ for every SCC and every symbol $s \in \Sigma$, it is easy to show that every SCC respects the decrease criterion on cycles. For example, in the cycle

$$f^\sharp(\mathrm{node}(\alpha,\beta)) \to g^\sharp(\mathrm{node}(\alpha,\beta)) \leftrightarrows g^\sharp(\mathrm{node}(\alpha,\beta)) \to f^\sharp(\alpha)$$

we have $\mathrm{node}(\alpha,\beta) \trianglerighteq \mathrm{node}(\alpha,\beta)$ and $\mathrm{node}(\alpha,\beta) \triangleright \alpha$, so the cycle is weakly decreasing with at least one strict decrease.

We may then again apply the correctness theorem to conclude that the erasure of all well-typed terms are strongly normalizing with respect to $\mathcal{R} \cup \beta$.

Note that the minimality condition is important: otherwise one could take

$$f : \forall\alpha\beta.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\beta) \to \boldsymbol{B}(\_)$$

with the rule

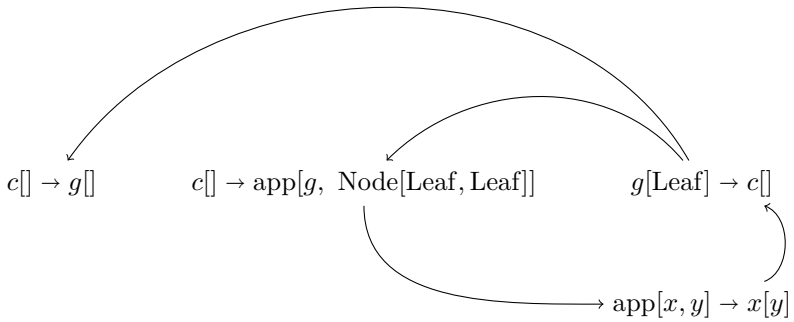
$$f\,\mathrm{node}(\mathrm{leaf},\mathrm{leaf})\,\mathrm{leaf}\;\; x\;y \to f\,\mathrm{leaf}\,\mathrm{leaf}\;\; y\;y$$

This rule can be typed in the context $x : \boldsymbol{B}(\mathrm{node}(\mathrm{leaf},\mathrm{leaf})), y : \boldsymbol{B}(\mathrm{leaf})$, but not minimally typed, as the variables x and y do not have type $\boldsymbol{B}(\alpha)$ for some variable α, and passes the termination criterion: the dependency graph is without cycles, as $\mathrm{node}(\mathrm{leaf},\mathrm{leaf})$ does not unify with leaf. However, this system leads to the non terminating reduction $f\,\mathrm{Leaf}\,\mathrm{Leaf} \to f\,\mathrm{Leaf}\,\mathrm{Leaf}$.

4 Comparison, future work

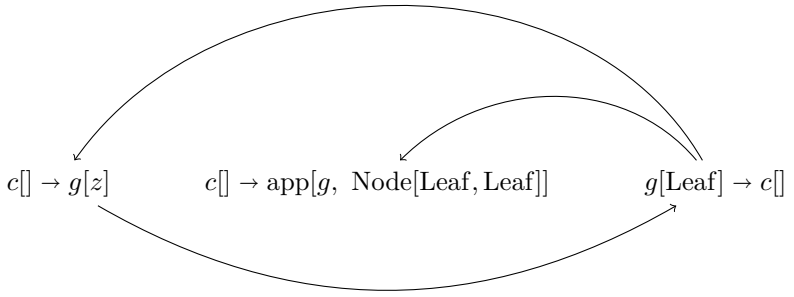
Several extensions of dependency pairs to different forms of higher-order rewriting have been proposed, first for applicative systems (variables may appear in application position, but there are no λ-abstractions) [19, 35, 3] and subsequently for more expressive systems including λ-abstractions [27, 8]. For the frameworks that do not handle the presence of bound variables, the usual approach is to defunctionalize (also called *lambda-lifting*) [16, 23] which is a whole program transformation which yields operationally equivalent terms for a given rewrite system.

All the techniques cited above, when applied to Example 1, where we may replace the rule $\mathrm{app} \to \lambda x.\lambda y.x\;y$ with the rule $\mathrm{app}\;\; x\;y \to x\;y$ (which does not involve bound variables), generate a dependency graph *with* cycles. For example, in Sakai & Kusakari [35], using the so-called "dynamic approach" the dependency graph is:



It is of course possible to prove that there are no infinite chains for this problem (the criterion is complete), but we have not much progressed from the initial formulation!

Using the so-called "static approach" from the same paper, which is based on computability (as is our framework), we obtain the following graph:



However it is not possible to prove that there are no infinite chains for this problem, as there is one! Therefore the criterion presented in the present paper allows a finer analysis of the possible calls.

The termination checking software `AProVE`, which used methods drawn in part from Giesl *et al* [19] succeeds in proving termination of Example 1, by using an analysis involving the computation of variable instances and symbolic reduction. As noted previously, our approach does not need such an expensive analysis as the information required *is already contained in the type information*. However it seems that such an analysis may be used to *infer* the type annotations required in our framework. At the moment it is unclear how the typing approach precisely compares to these techniques. More investigation is clearly needed in this direction.

`AProVE` can also easily prove termination of the second rewrite system (Example 2). However semantic information needs to be inferred (for example a polynomial interpretation needs to be given) when trying to well-order the cycle

$$f\ (\text{Node}\ x\ y) \to g\ (i\ (\text{Node}\ x\ y)) \leftrightarrows g\ (\text{Node}\ x\ y) \to f\ (i\ x)$$

This information is already supplied by our type system (through the fact that i is of type $\forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\alpha)$), and therefore it suffices to consider only syntactic information on the approximated dependency graph. The *subterm criterion* by Aoto and Yamada [3] is insufficient to treat this example.

Work by Bove and Capretta [12] allows one to use dependent types to encode functions that terminate for complex reasons, using functions which can be shown to be structurally recursive. While the theoretical power of this approach is stronger than that of ours, it not possible to give a straightforward encoding of our type-based framework using this approach, due to the presence of subtyping. Note also that our criterion applies to open terms with erased arguments, whereas the Bove-Capretta method does not.

The framework described here is only the first step towards a satisfactory *type-based dependency pair framework* using refinement types. We intuitively consider a "type level" first-order rewrite system, use standard techniques to show that that system is terminating, and show that this implies termination of the object level system. More work is required to obtain a satisfactory "dependency pairs by typing" framework.

Our work seems quite orthogonal to the *size-change principle* [28], which suggests we could apply this principle to treat cycles in the typed dependency graph, as a more powerful criterion than simple decrease on one indexed argument.

It is clear that the definitions and proofs in the current work extend to other first-order inductive types like lists, Peano natural numbers, etc. We conjecture that this framework can be extended to more general positive inductive types, like the type of Brower ordinals [9]. These kinds of inductive types seem to be difficult to treat with other (non type-based) methods.

For now types have to be explicitly given by the user, and for complete automation of our criterion it is necessary to infer the type annotations. Notice that trivial annotations (return type always $\boldsymbol{B}(\_)$) can very easily be infered automatically. Some work on automatic inference of type-level annotations has been carried out by Chin *et al* . [14] which considers annotations in the language of linear arithmetic, and by Barthe, Gregoire and Pastawski [6] for a more restricted language of size-types. We believe that inference of explicit type information in the terms is quite feasible with current state-of-the-art methods, for example those used for inferring the type of functional programs using GADTs [33].

We only consider matching on non-defined symbols, though an extension to a framework with matching on defined symbols seems feasible if we add some conversion rule to our type system.

We believe that refinement types are simply an alternative way of presenting the dependency pair method for higher-order rewrite systems. It is the occasion to draw a parallel between the types community and the rewriting community, by emphasizing that techniques used for the inference of dependent type annotations (for example work on *liquid types* [34]), may in fact be used to infer information necessary for proving termination and (we believe) vice-versa. It may also be interesting in the case of a programming language for the user to supply the types as documentation, in what some call "type directed programing".

Acknowledgements We thank Frederic Blanqui and Andreas Abel for the discussions that led to the birth of this work and for very insightful comments concerning a draft of this paper, as well as anonymous referees for numerous corrections.

References

1 A. Abel. Termination checking with types. *Theoretical Informatics and Applications*, 38(4):277–319, 2004.

2 A. Abel. Semi-continuous sized types and termination. In Z. Ésik, editor, *CSL*, volume 4207 of *Lecture Notes in Computer Science*, pages 72–88. Springer, 2006.

3 T. Aoto and T. Yamada. Dependency pairs for simply typed term rewriting. In J. Giesl, editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 120–134. Springer, 2005.

4 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

5 G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.

6 G. Barthe, B. Grégoire, and F. Pastawski. Practical inference for typed-based termination in a polymorphic setting. In *Typed Lambda Calculi and Applications*, pages 71–85, 2009.

7 F. Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In *Proc. of the 15th International Conference on Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, 2004.

8 F. Blanqui. Higher-order dependency pairs. In *Proceedings of the 8th International Workshop on Termination*, 2006.

9 F. Blanqui, J.-P. Jouannaud, and M. Okada. Inductive-data-type Systems. *Theoretical Computer Science*, 272:41–68, 2002.

10 F. Blanqui and C. Riba. Combining typing and size constraints for checking the termination of higher-order conditional rewrite systems. In *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Lecture Notes in Computer Science 4246, 2006.

11 F. Blanqui and C. Roux. On the relation between sized-types based termination and semantic labelling. In E. Grädel and R. Kahle, editors, *CSL*, volume 5771 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2009.

12 A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005.

13 N. G. D. Bruijn. The mathematical language automath, its usage, and some of its extensions. In M. Laudet, editor, *Proceedings of the Symposium on Automatic Demonstration*, volume 125, pages 29–61. Springer-Verlag, 1968.

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

15 Coq Development Team. *The Coq Reference Manual, Version 8.2.* INRIA Rocquencourt, France, 2008. `http://coq.inria.fr/`.

16 O. Danvy and L. R. Nielsen. Defunctionalization at work. In *proceedings of PPDP*, pages 162–174. ACM, 2001.

17 T. Freeman and F. Pfenning. Refinement types for ML. *SIGPLAN Not.*, 26(6):268–277, 1991.

18 J. Giesl, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *J. Symb. Comput.*, 34:21–58, July 2002.

19 J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *proceedings of the 5th FROCOS conference*, pages 216–231. Springer, 2005.

20 J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

21 N. Hirokawa and A. Middeldorp. Tyrolean termination tool: Techniques and features. *Information and Compututation*, 205:474–511, April 2007.

22 J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23th ACM Symposium on Principles of Programming Language*, 1996.

23 T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, pages 190–203. Springer-Verlag, 1985.

24 S. Jones and E. Meijer. Henk: a typed intermediate language, 1997.

25 J.-P. Jouannaud, C. Kirchner, and H. Kirchner. Incremental construction of unification algorithms in equational theories. In *Proceedings of the 10th Colloquium on Automata, Languages and Programming*, pages 361–373, London, UK, 1983. Springer-Verlag.

26 J.-P. Jouannaud and M. Okada. A computational model for executable higher-order algebraic specification languages. In *Proceedings of the sixth annual IEEE Symposium on Logic in Computer Science (LICS '91)*, pages 350–361, 1991.

27 K. Kusakari, Y. Isogai, M. Sakai, and F. Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. *IEICE Transactions*, 92-D(10):2007–2015, 2009.

28 C. S. Lee, N. D. Jones, and A. Ben-Amram. The size-change principle for program termination. In *Proceedings of POPL'01*, pages 81–92. ACM Press, 2001.

29 C. McBride and J. McKinna. The view from the left. *Journal of Functionnal Programming*, 14(1):69–111, 2004.

30 J. McKinna. Why dependent types matter. *SIGPLAN Not.*, 41(1):1–1, 2006.

31 R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

32 U. Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

33 S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ICFP '06, pages 50–61, New York, NY, USA, 2006. ACM.

34 P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In R. Gupta and S. P. Amarasinghe, editors, *PLDI*, pages 159–169. ACM, 2008.

35 M. Sakai and K. Kusakari. On dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E88-D(3):583–593, 2005.

36 H. Xi and D. Scott. Dependent types in practical programming. In *In Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227. ACM Press, 1998.