Refinement Types as Higher Order Dependency Pairs

Cody Roux

Refinement Types as Higher Order Dependency Pairs

Cody Roux

INRIA-Nancy Grand Est

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 object of active research. We observe that a variant of refinement types allow us to express a form of higher-order dependency pair criterion that only uses information at the type level, and we prove the correctness of this criterion.

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*[McK06,Bru68]. Dependent types explicitly allow "object level" terms to appear in the types, and can express arbitrarily complex program properties using the so called *Curry-Howard isomorphism*. We are particularly interested here in *refinement types*[XS98,FP91]. 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.

Dependency pairs are a highly successful technique for proving termination of first-order rewrite systems [AG00]. However, without modifications, 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. Indeed, let us examine the rewrite rule:

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

The termination of this rule can not 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* [HPS96,Abe04,BFG+04,Bla04,BR06].

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 $\langle f(c(x,y),z) \mid f(x,y)\rangle$ and the pair $\langle f(c(x,y),z) \mid g(f(x,y))\rangle$.

We can then define a *link* to be a pair (θ,ϕ) of substitutions, and a couple $(\langle t_1 \mid u_1\rangle, \langle t_2 \mid u_2\rangle)$ of dependency pairs such that $u_1\theta \to t_2\phi$. We may link links in an intuitive manner to form a *chain*, 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 [AG00] for details.

To prove that no infinite chains exist, one wants to work with the *dependency graph*: the graph is constituted of the dependency pairs as nodes and there is a vertex between $N_1 = \langle t_1 \mid u_1\rangle$ and $N_2 = \langle t_2 \mid u_2\rangle$ if there exist θ and ϕ such that $(\theta,\phi),(N_1,N_2)$ form a link. It is then shown that it is sufficient to consider only the cycles in this graph and prove that they may not lead to infinite chains [GTSKF06].

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* [JKK83]), so in practice we compute an approximation (or estimation) of the graph that is *conservative*: all nodes in the dependency graph are sure to appear in the approximated graph. One common [GTSKF06] (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 in a satisfactory way using a form of refinement types containing *patterns* which denote sets of possible values to which a term reduces. These patterns must be explicitly abstracted and applied, a choice that allows us to have very simple type inference. This allows us to build a form of dependency pair at the type level for higher order rewrite rules, as well as an approximated graph which corresponds to the approximation described above. We then state and prove the correctness of our criterion: if the graph is *acyclic*, then the well typed *closed* terms, in which pattern arguments are erased, are strongly normalizing for reduction of the rewrite system combined with *weak β-reduction*.

We leave the treatment of graphs with cycles for future work. We then conclude with a comparison with other approaches to higher order dependency pairs and possible extensions.

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 arguments. We define the syntax of *patterns*

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

Where $\alpha, \beta \in \mathcal{V}$ are *pattern variables*, and $\_$ is called *wildcard*.
We then define the set of types:

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

An *atomic type* is a type of the form $\boldsymbol{B}(p)$.
We finally define the set of terms:

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

Where $f, g \in \Sigma$ are *defined function symbols* and $x, y \in \mathcal{X}$ are *object variables*. Notice that application and abstraction of patterns is explicit.

Intuitively, $\boldsymbol{B}(p)$ denotes the set of terms that reduce to some term that matches p. To each $f \in \Sigma$, we give an *arity* or *type declaration* $\tau_f \in \mathcal{T}$. We write $\mathcal{FV}(t)$ (resp. $\mathcal{FV}(T)$, $\mathcal{FV}(\Gamma)$) for the set of free 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 $\forall \boldsymbol{\alpha}.T$ 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.

It may seem surprising that we choose to explicitly represent pattern abstraction and application in our system. This choice is justified by the simplicity of type inference with explicit parameters. The author is of the opinion that 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 [Mil78,JM97].

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.* Adga [Nor07], Epigram [McK06] or Coq [Coq08].

The typing rules of our system are given by the typing rules in figure 1.

To these rules we add the subtyping rule:

$$\frac{\Gamma \vdash t\colon T \quad T \leq U}{\Gamma \vdash t\colon U}\textbf{sub}$$

Where the subtyping relation is defined by:
An order on patterns:

- $p \ll \_$

$$\frac{}{\Gamma, x:T, \Gamma' \vdash x:T}\,\mathbf{ax}$$

$$\frac{\Gamma, x:T \vdash t:U}{\Gamma \vdash \lambda x:T.t:T \to U}\,\mathbf{t-lam}$$

$$\frac{\Gamma \vdash t:T}{\Gamma \vdash \lambda\alpha.t:\forall\alpha.T}\,\alpha \notin \mathcal{FV}(\Gamma)\ \mathbf{p-lam}$$

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

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

$$\frac{\Gamma \vdash t:T \to U \quad \Gamma \vdash u:T}{\Gamma \vdash t\ u:U}\,\mathbf{t-app}$$

$$\frac{\Gamma \vdash t:\forall\alpha.T}{\Gamma \vdash t\ p:T\{\alpha \mapsto p\}}\,\mathbf{p-app}$$

$$\frac{}{\Gamma \vdash f:\tau_f}\,\mathbf{symb}$$

Fig. 1. Typing Rules

- $p \ll p$
- $p_1 \ll q_1 \wedge p_2 \ll q_2 \Rightarrow \mathrm{node}(p_1, p_2) \ll \mathrm{node}(q_1, q_2)$

For all patterns p, p_1, p_2, q_1, q_2. This order is carried to types by:

- $p \ll q \Rightarrow \boldsymbol{B}(p) \leq \boldsymbol{B}(q)$
- $T_2 \leq T_1 \wedge U_1 \leq U_2 \Rightarrow T_1 \to U_1 \leq T_2 \to U_2$
- $T\{\alpha \mapsto \gamma\} \leq U\{\beta \mapsto \gamma\}$ for γ not free in $T, U \Rightarrow \forall\alpha.T \leq \forall\beta.U$

This type system is quite similar to the refinement types described for mini-ML by Freeman *et al* [FP91], and is not very distant from *generalized algebraic datatypes* as are implemented in certain Haskell compilers [Jon06], though subtyping is not present in that framework.

We furthermore suppose that all type declarations τ_f are of the form $\forall\alpha_1 \ldots \alpha_k.T_f$, where T_f is of the form $A_1 \to \ldots \to A_n \to T_1 \to \ldots \to T_m \to A_f$ where the A_i are of the form $\boldsymbol{B}(\alpha_{k_i})$, with $\alpha_{k_i} \neq \alpha_{k_j}$ if $i \neq j$, and α_{k_i} only appears *positively* in $T_1 \to \ldots \to T_m \to A_f$. The A_i's are called the *recursive arguments* of f, and there are no bound variables in T. The positivity condition is quite similar to the one detailed in Abel [Abe04].

A *constructor* is either Node or Leaf.

A rewrite rule is a pair of terms (l, r) which we write $l \to r$, such that l is an *algebraic pattern, i.e.* a term of the form $f\ p_1 \ldots p_k\ l_1 \ldots l_n$ with $f \in \Sigma$, p_i patterns and l_i *constructor terms*: variables, or a constructor (possibly) applied to constructor terms or patterns. We suppose that the free variables of r appear in l.

We suppose that functions in right-hand sides are *completely applied* to their pattern arguments: if g appears in r, and $\tau_g = \forall\alpha_1 \ldots \alpha_k.T$, then g is applied to k pattern arguments at each occurrence in r.

We say a context Γ *types* a rule $l \to r$ if there is some $T \in \mathcal{T}$ such that $\Gamma \vdash l:T$ and $\Gamma \vdash r:T$.

We suppose that all left-hand sides match on all recursive arguments, *i.e.* if $f\,p_1 \ldots, p_k l_1 \ldots l_n$ is a left-hand side, then n is equal to the number of recursive arguments of f.

Let l be the left hand side of a rule. We say that a context Γ *minimally types* l of type T if $\Gamma \vdash_{\min} l : T$ is derivable, where $\vdash_{\min}$ is derived by the rules:

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

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

$$\frac{\Gamma \vdash_{\min} l_1 : \boldsymbol{B}(p_1) \quad \Gamma \vdash_{\min} l_2 : \boldsymbol{B}(p_2)}{\Gamma \vdash_{\min} \text{Node}\ \ p_1\ p_2\ l_1\ l_2 : \boldsymbol{B}(\text{node}(p_1, p_2))}$$

$$\frac{\Gamma \vdash_{\min} l_1 : \boldsymbol{B}(p_1) \ldots \Gamma \vdash_{\min} l_n : \boldsymbol{B}(p_n)}{\Gamma \vdash_{\min} f\ \boldsymbol{q}\ l_1 \ldots l_n : T_f \phi}$$

Where $\tau_f = \forall \boldsymbol{\alpha}.\boldsymbol{B}(\alpha_{i_1}) \to \ldots \to \boldsymbol{B}(\alpha_{i_n}) \to T_f$ and $\phi \coloneqq \{\boldsymbol{\alpha} \mapsto \boldsymbol{q}\}$ is the substitution that sends α_{i_j} to p_j and all other variables to themselves.

Minimality is important to constrain the possible typing of left-hand sides: We need the types to be the most general possible, so as to ensure the correctness of any possible instance of a left-hand side.

Notice that if $\Gamma \vdash_{\min} l_i : T$ then T is *unique*. Minimal typing is implicitly used in other work on size-based termination [BR09], in which it is called the pattern condition.

Definition 1 Let $\rho \coloneqq f\ p_1 \ldots p_k l_1 \ldots l_n \to r$ be a rule in $\mathcal{R}$, Γ be a context such that $\Gamma \vdash_{\min} l : T_f \psi$, with $\psi = \{\boldsymbol{\alpha} \mapsto \boldsymbol{p}\}$ and $t = g\ q_1 \ldots q_{k'}$ be a subterm of r.

Suppose $\tau_f = \forall \boldsymbol{\alpha}.A_1 \to \ldots \to A_n \to T_f$ and $\tau_g = \forall \boldsymbol{\beta}.B_1 \to \ldots \to B_m \to T_g$.

The *dependency pair* associated to ρ and t is the tuple:

$$\langle f; \boldsymbol{A}\{\boldsymbol{\alpha} \mapsto \boldsymbol{p}\} \mid g; \boldsymbol{B}\{\boldsymbol{\beta} \mapsto \boldsymbol{q}\}\rangle$$

The set $\mathcal{DP}$ of dependency pairs associated to $\mathcal{R}$ is the set of all dependency pairs for each rule $l \to r \in \mathcal{R}$ and each subterm t of r of the form $g\ \boldsymbol{p}$.

Definition 2 Let $A = \boldsymbol{B}(p)$ and $B = \boldsymbol{B}(q)$ be atomic types, with distinct variable sets. We say that A and B have *common inhabitants* if $p\_ \bowtie q\_$, with $p\_$ being the pattern p in which every variable is replaced by $\_$ (likewise for $q\_$) and $p \bowtie q$ is defined as the symmetric-transitive-congruence closure of $\ll$.

Note that the relation $\bowtie$ is decidable.

Let $\mathcal{R}$ be a rewrite system, such that each rule is minimally typed by some context. The *dependency graph* $\mathcal{G}_{\mathcal{R}}$ associated to $\mathcal{R}$ is the directed graph where:

- The set of nodes is the set $\mathcal{DP}$, such that there are no shared variables between the different dependency pairs.
- There is an edge between $\langle f; \boldsymbol{A} \mid g; B_1, \ldots, B_m\rangle$ and $\langle h; C_1, \ldots, C_n \mid i; \boldsymbol{D}\rangle$ if
  1. $g = h$

2. $\forall i,\ B_i$ and C_i have common inhabitants.

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

Definition 3 We define the set of *erased terms* $\mathcal{T}rm^{|\cdot|}$ as:

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

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:

- $|x| = x$
- $|f| = f$
- $|\lambda x\colon T.t| = \lambda x.|t|$
- $|\lambda\alpha.t| = |t|$
- $|t\ u| = |t|\ |u|$
- $|t\ p| = |t|$
- $|\,\text{Leaf}\,| = \text{Leaf}$
- $|\,\text{Node}\,| = \text{Node}$

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

Definition 4 A *value* is a term $t \in \mathcal{T}rm^{|\cdot|}$ in the following form:

- $t = \lambda x\colon .t'$
- $t = \text{Leaf}$
- $t = \text{Node}\, v_1\ v_2$ with v_1 and v_2 values.

If $t \to^*_{\mathcal{R}\beta} v$ and v is a value, we say that *v is a value of t*.

We finally define the operational semantics of our system.

Definition 5 Given a rewrite system $\mathcal{R}$, An erased term t *head rewrites* to a term u if there is some rule $f\ p_1 \ldots p_k l_1 \ldots l_k \to r$ and some substitution σ such that

1. $t = f\ |l_1|\sigma \ldots |l_n|\sigma$
2. $|l_i|\sigma$ is a value for every $1 \leq i \leq n$
3. $u = |r|\sigma$.

In this case we write $t \to^{hd}_{\mathcal{R}} u$.

We then define *weak rewriting*, written $\to_{\mathcal{R}\beta}$, inductively:

$$\frac{t \to^{hd}_{\mathcal{R}} u}{t \to_{\mathcal{R}\beta} u}$$

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

$$\frac{t \to_{\mathcal{R}\beta} t'}{t\ u \to_{\mathcal{R}\beta} t'\ u}$$

$$\frac{u \to_{\mathcal{R}\beta} u'}{t\ u \to_{\mathcal{R}\beta} t\ u'}$$

And define $\to^*_{\mathcal{R}\beta}$ to be the reflexive transitive closure of $\to_{\mathcal{R}\beta}$.

We say a term $t \in \mathcal{T}rm^{|\cdot|}$ *normalizes* if all sequences of reductions $t \to_{\mathcal{R}\beta} t_1 \to_{\mathcal{R}\beta} t_2 \to_{\mathcal{R}\beta} \ldots$ is finite. We write $\mathcal{SN}$ to denote the set of all normalizing terms.

In a sense, our strategy is call-by-value, as functions only reduce when applied to only values in their recursive arguments. However, nothing is said about the other arguments, and β-reduction can occur at any time. Notice also that we do not impose confluence, or even have exhaustivity of matching in the left hand sides.

Let us give an example of the application of this technique.

Example 1 Take the rewrite system given by the signature: $\{\text{app} : \forall\alpha\beta.(\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\beta)) \to \boldsymbol{B}(\alpha) \to \boldsymbol{B}(\beta),\ f : \boldsymbol{B}(\text{leaf}), g : \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\text{leaf})\}$, . We give the rewrite rules:

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

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

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

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

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

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

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 2 has no cycles.

It is easy to verify that the criterion can be applied and that in consequence, according to theorem 6 that is proven in the next section, all well typed terms in the empty context are strongly normalizing.

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 exposed in [AG00] (on terms without abstraction, and without β-reduction). However this operation can be very costly if performed automatically and is, in its most naive 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.

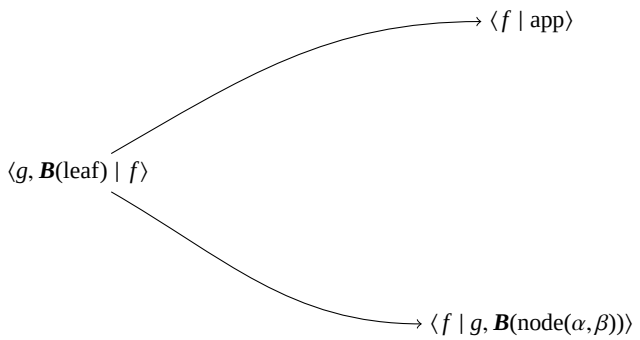


Fig. 2. Dependency graph for example 1

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 [CK01], 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 [Coq08].

4 Main Theorem and Proof

Theorem 6 Let $\mathcal{R}$ be a rewrite system such that each rule is minimally typed by some context.

If the dependency graph $\mathcal{G}_\mathcal{R}$ of the rewrite system $\mathcal{R}$ is acyclic and finite, then for every term t, if $\vdash t : T$ then $|t|$ is normalizing.

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, and leads to the non terminating reduction $f\ \mathrm{Leaf}\ \mathrm{Leaf} \to f\ \mathrm{Leaf}\ \mathrm{Leaf}$.

The proof uses a variation on the Tait-Girard technique of computability predicates [Tai67,Gir71]. The computability predicate method is a powerful method to carry out termination proofs in the higher order framework [BJR08]. It allows us to avoid considering higher order variables when constructing the dependency pairs [SK05]. It also avoids the use of *minimal bad sequences* [AG00] to prove termination, the existence of which can only be proven using a form of the Axiom of Choice. We do however use the Axiom of Choice implicitly in the proof of lemma 19.

We associate to each type some set of strongly normalizing closed terms. We then show correctness of our semantics, that is if a term is typed in the empty context, then it belongs to the interpretation of its type.

Definition 7 The *type interpretation* $[\![\_]\!]_\_$ is a function that to each $T \in \mathcal{T}$ and each *closed* pattern substitution θ, associates a set $[\![T]\!]_\theta \subseteq \{t \in \mathcal{T}rm^{|\cdot|} \mid t \text{ closed }\}$: let θ be some such valuation.

- $[\![\boldsymbol{B}(p)]\!]_\theta = \{t \in \mathcal{SN} \mid t \text{ closed } \wedge \forall v \text{ a value of } t, v \lhd\!\downarrow p\theta\}$
- $[\![T \to U]\!]_\theta = \{t \in \mathcal{SN} \mid \forall u \in [\![T]\!]_\theta, t\ u \in [\![U]\!]_\theta\}$
- $[\![\forall \alpha.T]\!]_\theta = \{t \in \mathcal{SN} \mid \forall \text{ closed } p \in \mathcal{P}, t \in [\![T]\!]_{\theta^\alpha_p}\}$

Where the *term matching relation* $\lhd\!\downarrow \subseteq \mathcal{T}rm^{|\cdot|} \times \mathcal{P}$ is defined in the following way:

- $t \lhd\!\downarrow \_$
- $t \lhd\!\downarrow \text{node}(p, q)$ iff $t = \text{Node}\ t_1\ t_2 \wedge t_1 \lhd\!\downarrow p \wedge t_2 \lhd\!\downarrow q$.
- $t \lhd\!\downarrow \text{leaf}$ iff $t = \text{Leaf}$.

The "magic" of computability proofs is contained in the following lemma:

Lemma 8 Let A, B be types, and θ be some closed pattern substitution. If $t \in \mathcal{T}rm^{|\cdot|}$ verifies:

$$\forall u \in [\![A]\!]_\theta, t\{x \mapsto u\} \in [\![B]\!]_\theta$$

Then $\lambda x.t \in [\![A \to B]\!]_\theta$

Proof. It suffices to prove: if $B = B_1 \to \ldots \to B_n \to B_o$ with B_o atomic, and $u_i \in B_i$ for all $i = 1 \ldots n$, then $(\lambda x.t)uu_1 \ldots u_n$ reduces to a value only if $t\{x \mapsto u\}u_1 \ldots u_n$ does. Details may be found in Barendregt [Bar84].

Let us prove correctness of the interpretation with respect to subtyping. We first need a substitution lemma:

Lemma 9 (substitution lemma)
Let T be a type. If α does not appear in the domain of θ then:

$$[\![T\{\alpha \mapsto p\}]\!]_\theta = [\![T]\!]_{\theta^\alpha_{p\theta}}$$

Proof. We first show that for all q a pattern, if α is not in the domain of θ, $q\{\alpha \mapsto p\}\theta = q\theta^\alpha_{p\theta}$: straightforward by induction on the structure of q.
We then proceed by induction on the type.

- Atomic case:

$$[\![\boldsymbol{B}(q)\{\alpha \mapsto p\}]\!]_\theta = \{t \in \mathcal{SN} \mid t \to^* v \text{ a value} \Rightarrow v \lhd\!\downarrow q\{\alpha \mapsto p\}\theta\}$$

  But by the previous remark, $q\{\alpha \mapsto p\}\theta = p\theta^\alpha_{p\theta}$, from which we can conclude.
- Arrow case: straightforward from induction hypothesis

– case $\forall\beta.T$. We may suppose by Barendregts convention that β is distinct from α, not in the domain of θ and distinct from all variables in p. We then have:

$$[\![(\forall\beta.T)\{\alpha \mapsto p\}]\!]_\theta = \{t \in \mathcal{SN} \mid \forall q \text{ closed}, t \in [\![T\{\alpha \mapsto p\}]\!]_{\theta^\beta_q}\}$$

Let $\theta' = \theta^\beta_q$. We may apply the induction hypothesis, which gives:

$$[\![T\{\alpha \mapsto p\}_{\theta'}]\!] = [\![T]\!]_{\theta'^\alpha_{p\theta'}}$$

And as β does not appear in p:

$$[\![T]\!]_{\theta'^\alpha_{p\theta'}} = [\![T]\!]_{\theta^\alpha_{p\theta}{}^\beta_q}$$

But we have:

$$\{t \in \mathcal{SN} \mid \forall q \text{ closed}, t \in [\![T]\!]_{\theta^\alpha_{p\theta}{}^\beta_q}\} = [\![\forall\beta.T]\!]_{\theta^\alpha_{p\theta}}$$

Which concludes the argument.

■

We may easily generalize this result to:

Corollary 10 Let T be a type. If ϕ is a substitution, and θ is a closed substitution such that the variables of T do not appear in the domain of θ, then:

$$[\![T\phi]\!]_\theta = [\![T]\!]_{\theta\circ\phi}$$

Where $\theta \circ \phi$ is the substitution defined by $\theta \circ \phi(\alpha) = \phi(\alpha)\theta$.

Lemma 11 Let T be some type and θ, θ' be two closed pattern substitutions. If $\theta = \theta'$ on the free variables of T, then $[\![T]\!]_\theta = [\![T]\!]_{\theta'}$.

Proof. Straightforward induction on T.

Lemma 12 Suppose $T \leq U$. Then for all θ, $[\![T]\!]_\theta \subseteq [\![U]\!]_\theta$

Proof. We proceed by induction on all the possible cases for the judgement $T \leq U$.

– $p \ll q$: We first show that for all terms t, $t{\lhd\downarrow}\ p \Rightarrow t{\lhd\downarrow}\ q$. We proceed by induction on the $\ll$ judgement. The first two cases are easy. In the third case, $t{\lhd\downarrow}\ \text{node}(p, q)$ which by definition implies that $t = \text{Node}\ u\ v$, with $u{\lhd\downarrow}\ p$ and $v{\lhd\downarrow}\ q$. We can then conclude by the induction hypothesis.
In addition it is easy to see that if $p \ll q$, then for all θ, $p\theta \ll q\theta$.
Now let $t \in [\![\boldsymbol{B}(p)]\!]_\theta$, and u be some value of t. Using the previous lemma, if $u{\lhd\downarrow}\ p\theta$ then $u{\lhd\downarrow}\ q\theta$, which allows us to conclude that $t \in [\![\boldsymbol{B}(p_1 \ldots q \ldots p_n)]\!]_\theta$.
– Suppose $T_2 \leq T_1$ and $U_1 \leq U_2$. Let t be in $[\![T_1 \to U_2]\!]_\theta$, we show that it is in $[\![T_2 \to U_2]\!]_\theta$. Let u be in $[\![T_2]\!]_\theta$. By the induction hypothesis, $u \in [\![T_1]\!]_\theta$, therefore (by definition of the interpretation), $t\ u$ is in $[\![U_1]\!]_\theta$, which by another application of the induction hypothesis, is included in $[\![U_2]\!]_\theta$. From this we can conclude that t is in $[\![T_2 \to U_2]\!]_\theta$.

– Let t be a term in $[\![\forall\alpha.T]\!]_\theta$ and p be some arbitrary closed pattern, and suppose that α, β are variables not appearing in the domain of θ. We then have

$$t \in [\![T]\!]_{\theta_p^\alpha}$$

Take some variable γ not free in T, U, and not in the domain of θ. Since $\forall\alpha.T \leq \forall\beta.U$, we have $T\{\alpha \mapsto \gamma\} \leq U\{\beta \mapsto \gamma\}$. Induction hypothesis gives:

$$[\![T\{\alpha \mapsto \gamma\}]\!]_{\theta'} \subseteq [\![U\{\beta \mapsto \gamma\}]\!]_{\theta'}$$

for all closed substitutions θ'. Take θ' to be the substitution everywhere equal to θ on the domain of θ and $\theta'(\gamma) = p$. The substitution lemma (lemma 9) gives:

$$[\![T]\!]_{\theta'^\alpha_p} \subseteq [\![U]\!]_{\theta'^\beta_p}$$

And lemma 11, with the hypothesis on γ gives:

$$[\![T]\!]_{\theta^\alpha_p} \subseteq [\![U]\!]_{\theta^\beta_p}$$

From this we can deduce $t \in [\![U]\!]_{\theta^\beta_p}$ and conclude.

■

Lemma 13 Let T, θ, θ' be as in lemma 11. If $\theta(\alpha) \ll \theta'(\alpha)$ for every free variable $\alpha \in T$ in a *positive* position, and $\theta(\beta) = \theta'(\beta)$ for every other variable, then $[\![T]\!]_\theta \subseteq [\![T]\!]_{\theta'}$.

Proof. First notice that if p is a pattern such that $\mathcal{FV}(p) \subseteq \mathcal{FV}(T)$, then $p\theta \ll p\theta'$. We then proceed by induction using lemma 12. ■

We wish to prove normalization of our system, if the rules are all typed by some minimal context and the dependency graph is without cycles.

Definition 14 Let Γ be a context, and σ a valuation from object variables to closed erased terms and θ a closed substitution on patterns. We say Γ is *satisfied* by (σ, θ) and write $\sigma \models_\theta \Gamma$, if $\sigma(x) \in [\![\Gamma x]\!]_\theta$ for every x in the domain of Γ.

Definition 15 We say that $[\![\_]\!]$ is *correct* if for every $\sigma \models_\theta \Gamma$,

$$\Gamma \vdash t : T \text{ implies } |t|\sigma \in [\![T]\!]_\theta$$

We first state that if each defined function symbol f belongs to $[\![\tau_f]\!]_\theta$ for any θ (τ_f has no free variables), then the interpretation is correct.

Theorem 16 (correction) Suppose that for all $f \in \Sigma$ and all θ a substitution, $f \in [\![\tau_f]\!]_\theta$. Then the interpretation is correct.

Proof. We proceed by induction on the typing derivation.

– ax: by definition of $\sigma \models_\theta \Gamma$.

– t-lam: By induction hypothesis, for all $\sigma', \theta \quad \sigma' \models_\theta \Gamma, x:T \Rightarrow |t|\sigma' \in [\![U]\!]_\theta$. By lemma 8 it suffices to show that $|t|\sigma\{x \mapsto u\}$ is in $[\![U]\!]_\theta$ for any $u \in [\![T]\!]_\theta$, where we suppose that x is not in the domain of σ by Barendregts convention. We take σ' to be equal to σ on the domain of σ and $\sigma'(x) = u$. It is easy to verify that $\sigma' \models_\theta \Gamma, x:T$.
– p-lam: by induction hypothesis, for all σ', θ' such that $\sigma \models_{\theta'} \Gamma, |t|\sigma'$ is in $[\![T]\!]_{\theta'}$. Let σ, θ be some such valuations and p be some closed pattern. As $|\lambda\alpha.t|\sigma = |t|\sigma$, we need to show that $|t|\sigma \in [\![T]\!]_{\theta^\alpha_p}$
  Observe that if α does not appear in Γ, then $\sigma \models_\theta \Gamma$ implies $\sigma \models_{\theta^\alpha_p} \Gamma$, by virtue of lemma 11. We may therefore conclude that $|t|\sigma$ is in $[\![T]\!]_{\theta^\alpha_p}$.
– leaf-intro: Clear by definition of $[\![\boldsymbol{B}(\text{leaf})]\!]_\theta$
– node-intro: let p, q be patterns, and t, u be terms in $[\![\boldsymbol{B}(p)]\!]_\theta$ and $[\![\boldsymbol{B}(q)]\!]_\theta$, respectively. The values of the term Node t u are exactly the terms of the form Node v_1 v_2, where v_1 is a value of t and v_2 is a value of u. Let v_1 and v_2 be two such terms. By definition $v_1 \triangleleft\!\downarrow p\theta$ and $v_2 \triangleleft\!\downarrow q\theta$. From this we have Node v_1 $v_2 \triangleleft\!\downarrow \text{node}(p, q)\theta$. From these statements we may conclude that Node t $u \in [\![\boldsymbol{B}(\text{node}(p, q))]\!]_\theta$.
– t-app: by definition and induction hypothesis.
– p-app: by hypothesis, $|t|\sigma \in [\![\forall x.T]\!]_\theta$, this gives by definition $|t|\sigma \in [\![T]\!]_{\theta^x_{p\theta}}$, and by the substitution lemma (lemma 9), $|t|\sigma \in [\![T\{x \mapsto p\}]\!]_\theta$
– symb: By hypothesis.
– sub: By application of the correctness of subtyping (lemma 12), and the induction hypothesis.

■

The next lemma assures that our definition of the graph $\mathcal{G}_\mathcal{R}$ is correct with respect to the semantics, in the sense that all possible recursive calls (themselves approximated by the $>_{dp}$ order) can be traced to some edge in the graph.

Lemma 17 If $(f, \boldsymbol{t}) >_{dp} (g, \boldsymbol{u}) >_{dp} (h, \boldsymbol{v})$ then if N_1 and N_2 are such that

– $(f, \boldsymbol{t}) >_{dp} (g, \boldsymbol{u}) \in [\![N_1]\!]_{(\theta,\theta')}$
– $(g, \boldsymbol{u}) >_{dp} (h, \boldsymbol{v}) \in [\![N_2]\!]_{(\sigma,\sigma')}$

Then there is an edge in $\mathcal{G}_\mathcal{R}$ between N_1 and N_2.

Proof. Take an arbitrary i. Let us show that $B_i = \boldsymbol{B}(p)$ and $C_i = \boldsymbol{B}(q)$ have common inhabitants, if $N_1 = \langle f; \boldsymbol{A} \mid g; \boldsymbol{B}\rangle$ and $N_2 = \langle g; \boldsymbol{C} \mid h; \boldsymbol{D}\rangle$. Take v a value such that $u_i \to^* v$. Then by definition of $>_{dp}$ and $[\![\_]\!]$, $v \triangleleft\!\downarrow p\theta'$ and $v \triangleleft\!\downarrow q\sigma$.

We show that $p\_ \bowtie q\_$, by double induction on the judgements $v \triangleleft\!\downarrow p\theta'$ and $v \triangleleft\!\downarrow q\sigma$.

– $p\theta'$ or $q\sigma = \_$. In this case we can only have p (or q) equal to $\_$ or some variable. We may then conclude by definition of $\bowtie$.
– $p\theta$ or $q\sigma = \alpha$ for some variable α. Suppose it is $p\theta$. Again we necessarily have $p = \beta$ for some variable β, which gives $p\_ = \_$ and we may conclude by definition of $\bowtie$. The other case is symmetric.
– $p\theta$ = leaf. In this case $q\sigma$ = leaf as well, since the other possible cases are treated above. We only have three possibilities:
  1. p is a variable. We can conclude by definition of $\bowtie$, as $p\_ = \_$.
  2. q is a variable. We conclude as above.

3. $p = q =$ leaf, we may again apply the definition of $\bowtie$.

– $p\theta = \text{node}(p_1, p_2)$. In this case, $q\sigma$ must be equal to $\text{node}(q_1, q_2)$, as other possible cases are treated above. Again we have three possible cases: p or q are variables, and we can conclude as above or $p = \text{node}(p'_1, p'_2)$ and $q = \text{node}(q'_1, q'_2)$, with $p_i = p'_i\theta$ and $q_i = q'_i\sigma$ for $i = 1, 2$. We then have $v = \text{Node}\, v_1\ v_2$, with $v_1 \triangleleft\downarrow\ p_1, q_1$ and $v_2 \triangleleft\downarrow\ p_2, q_2$. By induction hypothesis, $p'_i \bowtie q'_i$ for $i = 1, 2$. From this we may deduce $\text{node}(p'_1, p'_2) = \text{node}(p'_1, p'_2) \bowtie \text{node}(q'_1, q'_2) = \text{node}(q'_1, q'_2)$.
– other cases: The other cases are impossible, since we have both $v \triangleleft\downarrow p\theta$ and $v \triangleleft\downarrow q\sigma$. For example, if $p\theta =$ leaf and $q\sigma = \text{node}(q_1, q_2)$, then by examination of the rules defining $\triangleleft\downarrow$, we must have $v =$ Leaf, as $v \triangleleft\downarrow$ leaf, but then it is impossible to have $v \triangleleft\downarrow \text{node}(q_1, q_2)$.

■

We now need to show computability of defined symbols. We will proceed by induction on an order which subsumes the call relation, and which is well-founded if the dependency graph is without cycles (and finite).

Definition 18 We define the order $>_{dp} \subseteq \Sigma \times \left(\mathcal{T}rm^{|\cdot|}\right)^*$ by taking the transitive closure of the following relation: $(f, \boldsymbol{t}) >_{dp} (g, \boldsymbol{u})$ if there are valuations θ, θ' such that:

– for all i, t_i reduces to some value v.
– there is a node

$$\langle f; A_1, \ldots, A_n \mid g; B_1, \ldots, B_m \rangle \in \mathcal{G}_{\mathcal{R}}$$

such that $t_1 \in [\![A_1]\!]_\theta, \ldots, t_n \in [\![A_n]\!]_\theta$ and $u_1 \in [\![B_1]\!]_{\theta'}, \ldots, u_m \in [\![B_m]\!]_{\theta'}$.

We will write $(f, \boldsymbol{t}) >_{dp} (g, \boldsymbol{u}) \in [\![\langle f; A_1, \ldots, A_n \mid g; B_1, \ldots, B_m \rangle]\!]_{(\theta,\theta')}$ in such a case.

The first clause in the definition of $>_{dp}$ is important: define *hereditarily neutral* terms to be terms that never reduce to a value. Notice that all strongly normalizing hereditarily neutral terms are in $[\![T]\!]_\theta$ for all T, θ. We then have, if $\langle f; \boldsymbol{A} \mid g; \boldsymbol{B} \rangle$ is a dependency pair in $\mathcal{G}_{\mathcal{R}}$, for $t_1 \in [\![A_1]\!]_\theta, \ldots, t_n \in [\![A_n]\!]_\theta$, and $u_1, \ldots, u_m$ hereditarily neutral terms in $\mathcal{SN}$, $(f, \boldsymbol{t}) >_{dp} (g, \boldsymbol{u})$.

Thus, if the first clause were not required, it would be possible for $>_{dp}$ to not be well-founded, even if the graph $\mathcal{G}_{\mathcal{R}}$ is without cycles.

Lemma 19 If $\mathcal{G}_{\mathcal{R}}$ has no infinite paths, then $>_{dp}$ is well founded.

Proof. We show that an infinite reduction $(f_1, \boldsymbol{t}_1) >_{dp} (f_2, \boldsymbol{t}_2) >_{dp} \ldots$ would lead to an infinite path in the graph $\mathcal{G}_{\mathcal{R}}$. Let $(f_i, \boldsymbol{t}_i)_{i\in\mathbb{N}}$ be such a sequence. We may extract (using countable choice) a sequence $(N_i)_{i\in\mathbb{N}}$ and θ_i, θ'_i such that $(f_i, \boldsymbol{t}_i) >_{dp} (f_{i+1}, \boldsymbol{t}_{i+1}) \in [\![N_i]\!]_{(\theta_i,\theta'_i)}$. We may then apply lemma 17 to build an infinite path in $\mathcal{G}_{\mathcal{R}}$. ■

Definition 20 Let $g \in \Sigma$, with type $\forall \boldsymbol{\alpha}.\tau_g$, with $\tau_g = A_1 \to \ldots \to A_n \to T_g$, ϕ be some pattern substitution and θ be a closed pattern substitution. We define the *stratified interpretation* $[\![\tau_g\phi]\!]_\theta^{<^g_{dp}(f,\boldsymbol{t})}$ to be

$$\{t \in \mathcal{SN} \mid \forall \boldsymbol{u} \in [\![\boldsymbol{A}\phi]\!]_\theta, (g, \boldsymbol{u}) <_{dp} (f, \boldsymbol{t}) \Rightarrow tu_1 \ldots u_n \in [\![T_g\phi]\!]_\theta\}$$

The stratified interpretation is necessary to prove computability of symbols that are not fully applied to their arguments in right hand sides of rewrite rules as in example 1, when performing induction on $>_{dp}$. For this we need the following lemma:

Lemma 21 Let f and g be function symbols, of type τ_f and τ_g respectively. Let

$$\tau_g = \forall \boldsymbol{\beta}.B_1 \to \ldots \to B_m \to T_g$$

Suppose there exists some $\langle f; \boldsymbol{U} \mid g; \boldsymbol{V} \rangle \in \mathcal{G}_{\mathcal{R}}$, θ some closed pattern substitution and $t_i \in [\![U_i]\!]_\theta$ for $1 \leq i \leq n$. Then if ϕ is a pattern substitution such that $B_i\phi = V_i$, then for all closed substitution θ':

$$[\![\tau_g\phi]\!]_{\theta'} = [\![\tau_g\phi]\!]_{\theta'}^{>^g_{dp}(f,\boldsymbol{t})}$$

Proof.
Let $u_1 \in [\![V_1]\!]_{\theta'}, \ldots, u_m \in [\![V_m]\!]_{\theta'}$. It suffices to show that $(f, \boldsymbol{t}) >_{dp} (g, \boldsymbol{u})$. But this is exactly the definition of $>_{dp}$. ■

Lemma 22 Let $t \in \mathcal{T}rm^{|\cdot|}$ be a closed term, p an element of $\mathcal{P}$ and θ a closed pattern substitution. Suppose that for all u such that $t \to_{\mathcal{R}\beta} u$, $u \in \boldsymbol{B}(p)$. Then $t \in [\![\boldsymbol{B}(p)]\!]_\theta$.

Proof. It is clear that if all reducts of t are in $\mathcal{SN}$ then so is t. Suppose that v is a reduct of t that is a value. Then v is a reduct of u for some u a 1-step reduct of t, and by hypothesis $v \lhd\downarrow p\theta$. From this we can conclude that $t \in [\![\boldsymbol{B}(p)]\!]_\theta$. ■

Lemma 23 Let $f \in \Sigma$ of type $\tau_f = \forall \boldsymbol{\alpha}.A_1 \to \ldots \to A_n \to T_f$, $p_1 \ldots p_k$ be closed patterns, and $t_1, \ldots, t_n$ be in $[\![A_1]\!]_\theta, \ldots, [\![A_n]\!]_\theta$ respectively, where $\theta(\alpha_i) = p_i$. If for all r such that $f\ t_1 \ldots t_n \to^{hd}_{\mathcal{R}} r$, $r \in [\![T_f]\!]_\theta$ then $f\ t_1 \ldots t_n \in [\![T_f]\!]_\theta$

Proof. Suppose $T_f = B_1 \to \ldots B_m \to B_o$. Take $u_1 \in [\![B_1]\!]_\theta \ldots u_m \in [\![B_m]\!]_\theta$. By lemma 22, it suffices to prove that for all 1-step reducts v of $f\ \boldsymbol{t}\ \boldsymbol{u}$, $v \in [\![B_o]\!]_\theta$. We proceed by well founded induction on the tuple $\boldsymbol{t}\ \boldsymbol{u}$, ordered by the reduction order. This order is well-founded, as each t_i and u_i is in $\mathcal{SN}$. The reducts of $f\ \boldsymbol{t}\ \boldsymbol{u}$ are of three possible forms:

- $f\ t_1 \ldots t_i' \ldots t_n \boldsymbol{u}$ with t_i' a reduct of t_i. We can conclude by induction hypothesis.
- $f\ \boldsymbol{t} u_1 \ldots u_i' \ldots u_m$ with u_i' a reduct of u_i. We can conclude by induction hypothesis.
- $r\boldsymbol{u}$ with $f\boldsymbol{t} \to^{hd}_{\mathcal{R}} r$. We can conclude by hypothesis.

Notice that if $f\boldsymbol{t} \to^{hd}_{\mathcal{R}} u$, then t_i is a value, for every i, by definition of head reduction.

And finally, the correctness theorem for function symbols:

Theorem 24 Let $\mathcal{R}$ be a set of rewrite rule. Suppose each rule $f\ p_1 \ldots p_n l_1 \ldots l_n \to r$ is typed by some minimal context Γ. Suppose in addition that the dependency graph has no infinite paths. Then for all $f \in \Sigma, \theta$, $f \in [\![\tau_f]\!]_\theta$.

Proof. Let f, $\tau_f = \forall \boldsymbol{\alpha}.A_1 \to \ldots \to A_n \to T_f$, $\boldsymbol{p}$, θ and $\boldsymbol{t}$ be as in lemma 23. By the aforementioned lemma, it suffices to prove that $u \in [\![T_f]\!]_\theta$ for every $f\,\boldsymbol{t} \to^{hd}_{\mathcal{R}} u$. By definition, there is some rule $f\ \boldsymbol{q}\ \boldsymbol{l} \to r$ and some σ such that $|\boldsymbol{l}|\sigma = \boldsymbol{t}$ and $|r|\sigma = u$. By hypothesis, there is some Γ such that $\Gamma \vdash_{\min} f\ \boldsymbol{q}\ \boldsymbol{l} : T_f\psi$ and $\Gamma \vdash r : T_f\psi$ with $\psi := \{\boldsymbol{\alpha} \mapsto \boldsymbol{q}\}$. Let us show that there is some θ' such that $\theta' \circ \psi \ll \theta$. First observe that $x : \boldsymbol{B}(\alpha_x) \in \Gamma$ for some variable α_x for every $x \in \boldsymbol{l}$, and that furthermore $\alpha_x \neq \alpha_y$ if $x \neq y$. We may take θ' to be the substitution which sends α_x to the "patternification" of $\sigma(x)$: $\theta'(\alpha_x) = pat(\sigma(x))$ with pat defined by:

- $pat(\text{Leaf}) = \text{leaf}$
- $pat(\text{Node}\ t\ u) = \text{node}(pat(t), pat(u)$

We define θ' to be equal to θ on the variables not contained in Γ.

Suppose $A_i = \boldsymbol{B}(\alpha_{k_i})$ for all i, and $\Gamma \vdash_{\min} l_i : \boldsymbol{B}(q_i)$. We must show that $\theta' \circ \psi \ll \theta$. Now it is easy to see that $t_i \lhd\!\downarrow q_i\theta' = \psi(\alpha_{k_i})\theta'$.

We have by hypothesis $t_i \lhd\!\downarrow \theta(\alpha_{k_i})$. First we prove that $\theta' \circ \psi \ll \theta$ on α_{k_i} for each i. We proceed by induction on the derivation of $t_i \lhd\!\downarrow \alpha_{k_i}\theta$.

- $\alpha_{k_i}\theta = \_$. This case is trivial.
- $\alpha_{k_i}\theta = \text{leaf}$. In this case, $t_i = \text{Leaf}$, so we either have $l_i = \text{Leaf}$, which implies $\psi(\alpha_{k_i}) = \text{leaf}$, or $l_i = x$ for some variable x and $\psi(\alpha_{k_i}) = \alpha_x$ and $\theta'(\alpha_x) = \text{leaf}$. Either case allows us to conclude $psi(\alpha_{k_i})\theta' \ll \alpha_{k_i}\theta$
- $\alpha_{k_i}\theta = \text{node}(p^1, p^2)$. We proceed by cases: either $l_i = x$ and $\sigma(x) = \text{Node}\ t^1\ t^2$. We then have $\psi(\alpha_{k_i}) = \alpha_x$ and $\theta'(\alpha_x) = \text{node}(pat(t^1), pat(t^2))$, $t^1 \lhd\!\downarrow p^1$ and $t^2 \lhd\!\downarrow p^2$. We can then conclude by induction hypothesis.
  In the other case, $l_i = \text{Node}\ q^1\ q^2\ l^1\ l^2$. By definition of $\lhd\!\downarrow$, we have $|l^e|\sigma \lhd\!\downarrow p^e$ for $e = 1, 2$. We may then apply the induction hypothesis to conclude $q^e\theta' \ll p^e$, and therefore $\psi(\alpha_{k_i})\theta' = \text{node}(q^1, q^2)\theta' \ll \text{node}(p^1, p^2)$.

Now if α_j does not appear in the recursive types, then $\psi(\alpha_j) = \alpha_j$, and $\theta'(\alpha_j) = \theta(\alpha_j)$.

Observe now that $\sigma \models_{\theta'} \Gamma$. This is straightforward, as θ' models the action of σ on type level variables.

We need to show that $|r|\sigma$ is in $[\![T_f]\!]_\theta$. We will show that $|r|\sigma$ is in $[\![T_f\psi]\!]_{\theta'} = [\![T_f]\!]_{\theta' \circ \psi} \subseteq [\![T_f]\!]_\theta$ by corollary 10 and lemma 13, using the positivity hypothesis. We wish to apply theorem 16, knowing $\Gamma \vdash r : T_f\psi$ and $\sigma \models_{\theta'} \Gamma$. However, this fails as we still have not proven that any g that appears in r belongs to the interpretation of its type! To resolve this apparent circularity, we proceed by well founded induction on $(f, \boldsymbol{t})$ ordered by $>_{dp}$: by induction hypothesis, for all g of type $\forall\boldsymbol{\beta}.\tau_g = \forall\boldsymbol{\beta}.B_1 \to \ldots B_m \to T_g$, all $q'_1, \ldots, q'_k$ and all θ'' equal to θ' on its domain, and $u_1, \ldots, u_m$ in $[\![B\phi]\!]_{\theta''} \ldots [\![B\phi]\!]_{\theta''}$ respectively such that $(f, \boldsymbol{t}) >_{dp} (g, \boldsymbol{u})$, $g\ \boldsymbol{u}$ is in $[\![T_g\phi]\!]_{\theta''}$, where $\phi = \{\boldsymbol{\beta} \mapsto \boldsymbol{q'}\}$. But this means by definition that $g \in [\![\tau_g\phi]\!]^{<^g_{dp}(f,\boldsymbol{t})}_{\theta''}$. It is then sufficient to prove that we may apply lemma 21 to any $g\ \boldsymbol{q'}$ that appears in r.

The rule $f\ \boldsymbol{q}\ \boldsymbol{l} \to r$ and the subterm $g\ \boldsymbol{q'}$ correspond to some dependency pair $\langle f, \boldsymbol{U} \mid g, \boldsymbol{V}\rangle$ of $\mathcal{G}_{\mathcal{R}}$. First we observe that $t_i \in [\![A_i\psi]\!]_{\theta'}$, by consideration of the definition

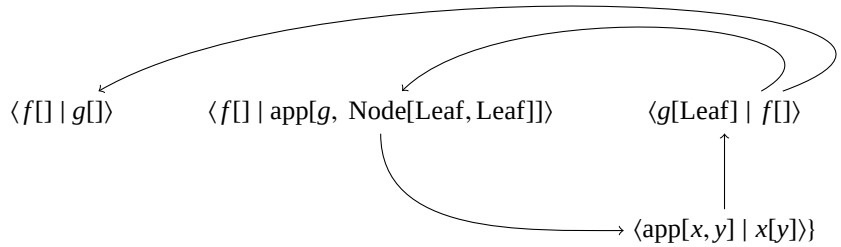
of ψ and θ'. Then by definition of dependency pairs, we have $A_i\psi = U_i$. Finally, if $\phi = \{\boldsymbol{\beta} \mapsto \boldsymbol{q'}\}$, then $B_j\phi = V_j$ for every j. This concludes the proof. ■

If the rewrite system $\mathcal{R}$ is finite then so is $\mathcal{G}_\mathcal{R}$, if in addition the graph is acyclic then it is well known that there are no infinite paths in the graph, and we may apply the above theorem.

5 Comparison, future work

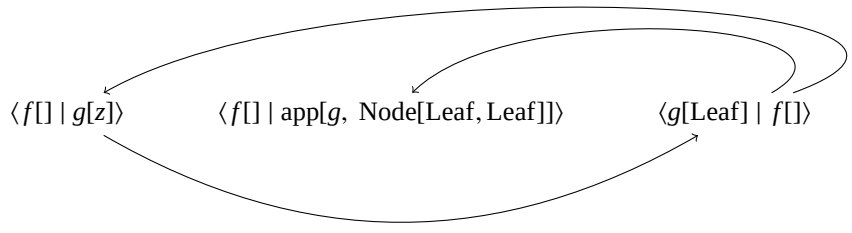
Several extensions of dependency pairs to different forms of higher order rewriting have been proposed [KISB09,Bla06,GTSK05,SK05,AY05]. However, these frameworks do not handle the presence of bound variables, for which the usual approach is to defunctionalize (also called *lambda-lifting*) [DN01,Joh85].

The criterion presented in this paper is quite weak, as it does not (yet) take into account the possibility of cycles in the dependency graph, an essential aspect of methods of termination using dependency pairs. However, it is a step in the right direction, as the graphs that result from this approach are often smaller than those generated in other approaches to higher order dependency pairs. In particular, all the techniques cited above, when applied to example 1, where we replace the rule app $\to \lambda x.\lambda y.x\ y$ with the rule app $x\ y \to x\ y$ which does not involve bound variables, generate a dependency graph with cycles. For example, in Sakai & Kusakari [SK05], using the SN framework 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 SC-framework 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 this paper allows a finer analysis of the possible calls.

The termination checking software `AProVE` [GTSK05] succeeds in proving termination of example 1, by using an analysis involving instance computation and symbolic reduction. As noted previously, 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 compares to these techniques. More investigation is clearly needed in this direction.

The framework described here is only the first step towards a satisfactory higher order 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: we need to be able to consider cycles in the dependency graph to be able to have a useful criterion. We believe that this is possible by combining this work with previous work on size types, especially work by Blanqui [Bla04]. Our work seems quite orthogonal to the *size-change principle* [LJBa01], which suggests we could apply this principle to treat the remaining cycles after having computed the dependency graph approximation.

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 [BJO02]. 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 it would be interesting to investigate inference of 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* [CK01] which may provide inspiration. There may also be connections with work on inferring the type of functional programs using GADTs [Jon06].

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 typing system.

Finally, it would be more pleasing to have normalization results on open terms and with strong reduction, but to do so requires a modification of our semantics, the definition of $>_{dp}$ in particular would have to be extended to handle open terms.

References

Abe04. A. Abel. Termination checking with types. *Theoretical Informatics and Applications*, 38(4):277–319, 2004.

AG00. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.

AY05. 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.

Bar84. H. P. Barendrecht. *The Lambda Calculus: its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1984.

BFG+04. 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.

BJO02. F. Blanqui, J.-P. Jouannaud, and M. Okada. Inductive-data-type Systems. *Theoretical Computer Science*, 272:41–68, 2002.

BJR08. F. Blanqui, J.-P. Jouannaud, and A. Rubio. The computability path ordering: the end of a quest. In *7th EACSL Annual Conference on Computer Science Logic - CSL'08*, volume 5213 of *LNCS*, Bertinoro Italie, 2008.

Bla04. 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.

Bla06. F. Blanqui. Higher-order dependency pairs. In *Proceedings of the 8th International Workshop on Termination*, 2006.

BR06. 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.

BR09. F. Blanqui and C. Roux. On the relation between sized-types based termination and semantic labelling. In Erich Grädel and Reinhard Kahle, editors, *CSL*, volume 5771 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2009.

Bru68. N. G. De 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.

CK01. W. N. Chin and S. C. Khoo. Calculating sized types. *Journal of Higher-Order and Symbolic Computation*, 14(2-3):261–300, 2001.

Coq08. Coq Development Team. *The Coq Reference Manual, Version 8.2*. INRIA Rocquencourt, France, 2008. `http://coq.inria.fr/`.

DN01. O. Danvy and L. R. Nielsen. Defunctionalization at work. In *proceedings of PPDP*, pages 162–174. ACM, 2001.

FP91. T. Freeman and F/ Pfenning. Refinement types for ML. *SIGPLAN Not.*, 26(6):268–277, 1991.

Gir71. J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. Fenstad, editor, *Proc. of the 2nd Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1971.

GTSK05. 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.

GTSKF06. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

HPS96. 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.

JKK83. 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.

JM97. S. Jones and E. Meijer. Henk: a typed intermediate language, 1997.

Joh85. T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, pages 190–203. Springer-Verlag, 1985.

Jon06. S. P. Jones. Simple unification-based type inference for GADTs. pages 50–61. ACM Press, 2006.

KISB09. K. Kusakari, Y. Isogai, M. Sakai, and F. Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. pages 2007–2015, 2009.

LJBa01. C. S. Lee, N. D. Jones, and A. M. Ben-amram. The size-change principle for program termination, 2001.

McK06. J. McKinna. Why dependent types matter. *SIGPLAN Not.*, 41(1):1–1, 2006.

Mil78. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

Nor07. 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.

SK05. 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.

Tai67. W. W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

XS98. 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.