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 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-order*. We describe a syntactic termination criterion involving the graph and the order, and prove our main result: if the graph passes the criterion, then every well-typed term is strongly normalizing.

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. Let us examine the rewrite rule:

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

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* [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 $f(c(x,y),z) \to f(x,y)$ and the pair $f(c(x,y),z) \to g(f(x,y))$.

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 $u_1\theta \to^* t_2\phi$. 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 [AG00] 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 [GAO02]. 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 edges in the dependency graph are sure to appear in the approximated graph. One common (see for instance Giesl [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 using a form of refinement types containing *patterns* which denote sets of possible values to which a term reduces. These type-patterns must be explicitly abstracted and applied, a choice that allows us to have very simple type inference. This allows us 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 capture the subterm ordering, and use this order to express a *decrease condition* along cycles in the approximated dependency graph. We then state the correctness of the criterion: 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 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. We then conclude 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*

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

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:

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

With $x \in \mathcal{X}$ a set of term variables, $f \in \Sigma$ is a set of *function symbols* and $\alpha \in \mathcal{V}$. Defined symbols are in lower case. Notice that application and abstraction of patterns is explicit. A *constructor* is either Node or Leaf. A *context* is a list of judgements $x\colon T$ with $x \in \mathcal{X}$ and $T \in \mathcal{T}$, with each variable appearing only once.

Intuitively, $\boldsymbol{B}(p)$ denotes the set of terms that reduce to some term that *matches* the pattern p. For instance, any binary tree t is in the semantics of $\boldsymbol{B}(\_)$, only binary trees that reduce to $\mathsf{Node}\ t_1\ t_2$ for some binary trees t_1 and t_2 are in $\boldsymbol{B}(\mathsf{node}(\_,\_))$, and only terms that *never* reduce to a constructor are in $\boldsymbol{B}(\bot)$. Our operational semantics are 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 $\boldsymbol{B}(\mathsf{node}(\_,\_))$ if *all* its reducts reduce to a term of the form $\mathsf{Node}\ t\ u$.
- 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 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 (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 $\tau_f = \forall \alpha_1, \ldots, \alpha_n.A_1 \to \ldots \to A_k \to T_f$ with

- $n \geq k$
- $A_i = \boldsymbol{B}(\alpha_i)$
- $\forall 1 \leq i \leq k, \alpha_i$ appears *positively* in T_f.

In this case k is called the number of *recursive arguments*.

The positivity condition is quite similar to the one used in the usual formulation of type-based termination, see for instance Abel [Abe06] 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.

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

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

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

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

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

$$\frac{\Gamma \vdash t : T \to U \qquad \Gamma \vdash u : T}{\Gamma \vdash t\ u : U}\ \textbf{t-app}$$

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

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

Fig. 1. Typing Rules

To these rules we add the subtyping rule:

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

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

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

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

- $p \ll q \Rightarrow \boldsymbol{B}(p) \leq \boldsymbol{B}(q)$
- $T_2 \leq T_1 \wedge U_1 \leq U_2 \Rightarrow T_1 \to U_1 \leq T_2 \to U_2$
- $T \leq U \Rightarrow \forall \alpha . T \leq \forall \alpha . U$

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 [JVWW06], though subtyping is not present in that framework.

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. 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 [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].

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

$$l_1, l_2 \in \mathcal{L} := x \mid \mathrm{Leaf} \mid \mathrm{Node}\ \ l_1\ l_2$$

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 $f\ \ p_1 \ldots p_n\ l_1 \ldots l_k$ with $f \in \Sigma$, $p_i \in \mathcal{P}$ and $l_i \in \mathcal{L}$, such that k is the number of recursive arguments of f. We suppose that the free variables of r appear in l.

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 $\vdash_{\text{min}}$ defined in figure 2.

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

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

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

$$\frac{\Gamma \vdash_{\text{min}} l_1 : \boldsymbol{B}(p_1) \qquad \dots \qquad \Gamma \vdash_{\text{min}} l_k : \boldsymbol{B}(p_k)}{\Gamma \vdash_{\text{min}} f\ p_1 \dots p_k\ \beta_{k+1} \dots \beta_l\ l_1 \dots l_k : T_f \phi}\ \boldsymbol{\alpha} \notin \Gamma$$

With $\tau_f = \forall \alpha_1 \dots \alpha_l . A_1 \to \dots \to A_k \to T_f$ and $\phi(\alpha_i) = p_i$ if $1 \leq i \leq k$ and $\phi(\alpha_j) = \beta_j$ for $k < i \leq l$.

Fig. 2. Minimal Typing Rules

Notice that if $\Gamma \vdash_{\text{min}} l_i : T$ then T is *unique*. Minimal typing is present in other work on size-based termination [BR09], in which it is called the *pattern condition*. The purpose of minimal typing is to constrain the possible types of constructor terms in left hand sides.

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

Definition 1 Let $\rho = f\ \boldsymbol{p}\ \boldsymbol{l} \to r$ be a rule in $\mathcal{R}$, with Γ such that $\Gamma \vdash_{min} f\ \boldsymbol{p}\ \boldsymbol{l} : T$, and $\Gamma \vdash r : T$. The set of *type dependency pairs* $DP_{\mathcal{T}}(\rho)$ is the set

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

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 [AG00]). 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

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

The information that g is called on the argument $h\ x$ 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 $\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\mathsf{leaf})$, which is the case if every normal form of $h\ t$ is either neutral or Leaf, we have a more precise approximation.

Note that, in addition, a dependency pair is not formally a (higher-order) rewrite rule, though it may be seen as a first-order one.

Definition 2 Let p and q be patterns. We say that p and q are *pattern-unifyable*, and write $p \bowtie q$, if p' and q' are unifyable, where p' and q' are the patterns p and q in which each occurrence of $\_$ and each occurrence of a variable is replaced by some *fresh* variable.

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 between the dependency pairs $t \to g^\sharp(p_1, \ldots, p_k)$ and $h^\sharp(q_1, \ldots, q_l) \to u$ if $g = h$, $k = l$ and 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 3 We define the *embeddeding preorder* on $\mathcal{P}$ written $p \rhd q$ by the following rules

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

With $\trianglerighteq$ as the reflexive closure of $\rhd$ and with the further condition that if $p \rhd q$, then p and q may not contain any occurrence of $\_$.

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

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

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

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

However, there is no substitution θ such that $\mathsf{node}(\alpha, \beta)\theta = \_$. There are two ways to deal with this. Either we take subtyping into account when performing matching, or we do away with the pattern arguments when performing reduction. We adopt the second solution, as it is used in practice when dealing with languages with dependent type annotations (see for example McKinna [McK06]). Symmetrically, we erase pattern abstractions as well.

Definition 4 We define the set of *erased terms* $\mathcal{T}rm^{|\cdot|}$ as:

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

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:

$$\begin{array}{lcl} |x| & = & x \\ |f| & = & f \\ |\lambda x\colon T.t| & = & \lambda x.|t| \\ |\lambda \alpha.t| & = & |t| \\ |t\ u| & = & |t|\ |u| \\ |t\ p| & = & |t| \\ |\mathsf{Leaf}| & = & \mathsf{Leaf} \\ |\mathsf{Node}| & = & \mathsf{Node} \end{array}$$

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

Definition 5 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.

Theorem 6 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 *recursive index* $\iota^i : \Sigma \to \mathbb{N}$ which to $f \in \Sigma$ associates an integer $1 \leq \iota^i_f \leq k$ (with k the number of recursive arguments of f).

Suppose that 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 $p_{\iota^i_f} \trianglerighteq q_{\iota^i_g}$. Finally suppose that 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

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

then for every Γ, t, T such that $\Gamma \vdash t : T$,

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

The proof of this theorem can be found in the appendix. Let us give two examples of the application of this technique.

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

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

$$f \to \mathsf{app}\ \ \mathsf{node}(lea\, f, lea\, f)\ \ \mathsf{leaf}\ \ (g\ \ \mathsf{node}(\mathsf{leaf}, \mathsf{leaf}))\ (\mathsf{Node}\ \mathsf{leaf}\ \mathsf{leaf}\ \ \mathsf{Leaf}\ \ \mathsf{Leaf})$$

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

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

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

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

It is possible to verify that the criterion can be applied and that in consequence, according to theorem 6, 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 [GTSK05] (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.

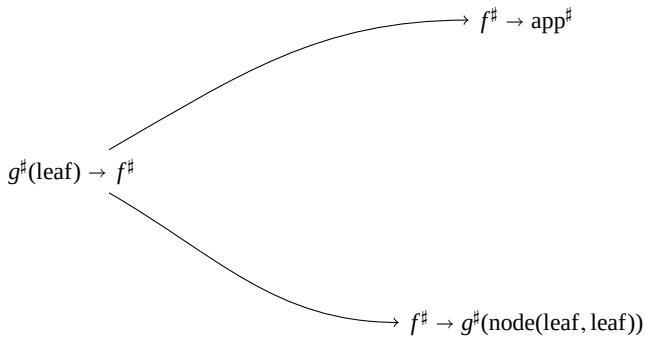


Fig. 3. Dependency graph of example 1

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.

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].

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

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

Again with the type arguments omited, 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 by typed in the context $\Gamma = x : \boldsymbol{B}(\alpha), y : \boldsymbol{B}(\beta)$, The system with full type annotations is given in the appendix.

The dependency graph is given in figure 3, and has as SCCs the full subgraphs of $\mathcal{G}_\mathcal{R}$ with nodes $\{i^\sharp(\mathsf{node}(\alpha,\beta)) \to i^\sharp(\alpha), i^\sharp(\mathsf{node}(\alpha,\beta)) \to i^\sharp(\beta)\}$, $\{f^\sharp(\mathsf{node}(\alpha,\beta)) \to g^\sharp(\mathsf{node}(\alpha,\beta), g^\sharp(\mathsf{node}(\alpha,\beta) \to f^\sharp(\alpha)\}$ and $\{h^\sharp(\mathsf{node}(\alpha,\beta)) \to h^\sharp(\alpha)\}$ 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

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

we have $\mathsf{node}(\alpha,\beta) \trianglerighteq \mathsf{node}(\alpha,\beta)$ and $\mathsf{node}(\alpha,\beta) \triangleright \alpha$, 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 $\mathcal{R} \cup \beta$.

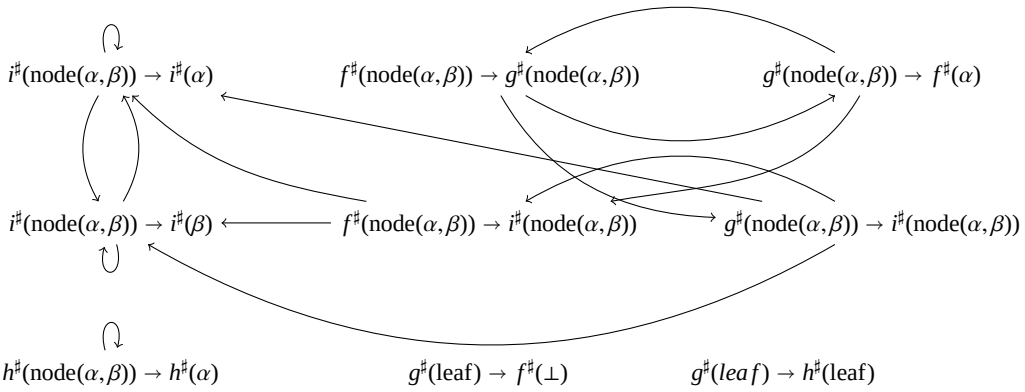


Fig. 4. The dependency graph for example 2

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

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

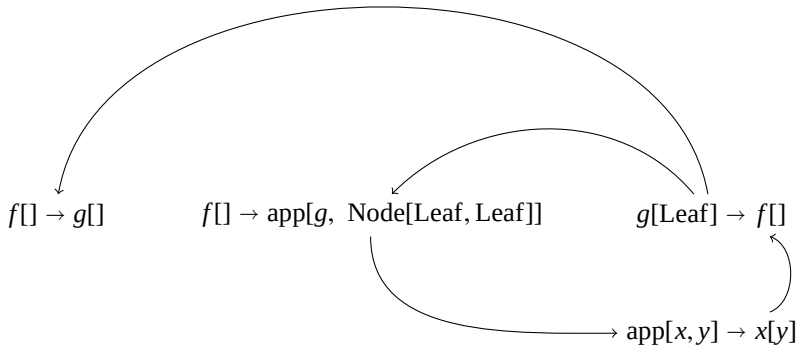
This rule can be typed in the context $x : \boldsymbol{B}(\mathsf{node}(\mathsf{leaf},\mathsf{leaf})), y : \boldsymbol{B}(\mathsf{leaf})$, but not minimally typed, and passes the termination criterion: the dependency graph is without cycles, as $\mathsf{node}(\mathsf{leaf},\mathsf{leaf})$ does not unify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

4 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].

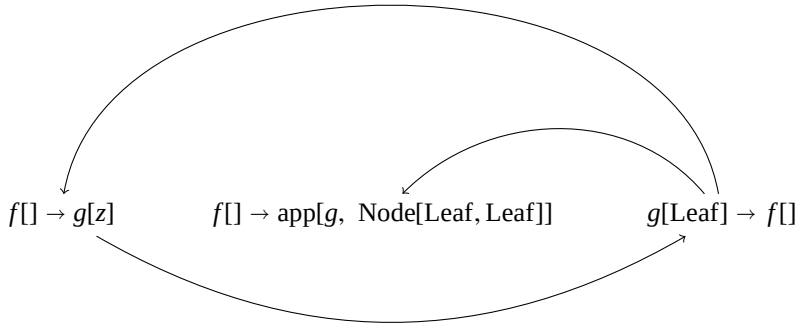
In particular, all the techniques cited above, when applied to example 1, where we replace the rule $\mathsf{app} \to \lambda x.\lambda y.x\ y$ with the rule $\mathsf{app}\ \ x\ y \to x\ y$ (which

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.

`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

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

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 [AY05] is insufficient to treat this example.

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.

Our work seems quite orthogonal to the *size-change principle* [LJBa01], 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 [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. On the other hand, we believe that the inference of the 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 [JVWW06].

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* [RKJ08]), 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".

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.

Acknowledgements

We thank Frederic Blanqui 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 on a previous version of this paper.

References

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

Abe06. Andreas Abel. Semi-continuous sized types and termination. In Zoltán Ésik, editor, *CSL*, volume 4207 of *Lecture Notes in Computer Science*, pages 72–88. Springer, 2006.

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.

Ber05. U. Berger. Continuous semantics for strong normalization. In S.B. Cooper, B. Löwe, and L. Torenvliet, editors, *CiE 2005: New Computational Paradigms*, volume 3526 of *Lecture Notes in Computer Science*, pages 23–34. Springer-Verlag, 2005.

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.

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.

GAO02. J. Giesl, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *J. Symb. Comput.*, 34:21–58, July 2002.

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.

JVWW06. S. P. 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.

KISB09. K. Kusakari, Y. Isogai, M. Sakai, and F. Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. *IEICE TRANSACTIONS on Information and Systems*, E92-D No.10:2007–2015, 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.

RKJ08. P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In Rajiv Gupta and Saman P. Amarasinghe, editors, *PLDI*, pages 159–169. ACM, 2008.

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.

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.

A The full system of example 2

Every rule is typed in the context $x : \boldsymbol{B}(\alpha), y : \boldsymbol{B}(\beta)$, and we remind that the types of defined functions are:

$$f, g : \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\_) \quad h : \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\bot) \quad i : \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\alpha)$$

The rewrite system with all type annotations is then

$$\begin{aligned}
f\ \mathsf{node}(\alpha,\beta)\ (\mathsf{Node}\ \alpha\ \beta\ x\ y) &\to g\ \mathsf{node}(\alpha,\beta)\ (i\ \mathsf{node}(\alpha,\beta)\ (\mathsf{Node}\ \alpha\ \beta\ x\ y)\\
g\ \mathsf{node}(\alpha,\beta)\ (\mathsf{Node}\ \alpha\ \beta\ x\ y) &\to f\ \alpha\ (i\ \alpha\ x)\\
g\ \mathsf{leaf}\ \mathsf{Leaf} &\to f\ \bot\ (h\ \mathsf{leaf}\ \mathsf{Leaf})\\
i\ \mathsf{node}(\alpha,\beta)\ (\mathsf{Node}\ \alpha\ \beta\ x\ y) &\to \mathsf{Node}\ \mathsf{node}(\alpha,\beta)\ (i\ \alpha\ x)\ (i\ \beta\ y)\\
i\ \mathsf{leaf}\ \mathsf{Leaf} &\to \mathsf{Leaf}\\
h\ \mathsf{node}(\alpha,\beta)\ (\mathsf{Node}\ \alpha\ \beta\ x\ y) &\to h\ \alpha\ x
\end{aligned}$$

B Proof of theorem 6

The proof uses computability predicates (or candidates). As mentioned before, the absence of control, and particularly the lack of orthogonality makes giving accurate semantics somewhat difficult. We draw inspiration from the termination semantics of Berger [Ber05], which uses sets of values to denote terms. As is standard in computability proofs, each type will be interpreted as a set of strongly normalizing (erased) terms. Suppose a term t reduces to the normal forms Leaf and $\mathsf{Node}\ \mathsf{Leaf}\ \mathsf{Leaf}$. In that case t is in the candidate that contains all terms that reduce to Leaf or $\mathsf{Node}\ \mathsf{Leaf}\ \mathsf{Leaf}$, or are hereditarily neutral. If t the erasure of a term of type $\boldsymbol{B}(\alpha)$ for some pattern variable α, the interpretation $[\![\boldsymbol{B}(\alpha)]\!]$ must depend on some valuation of the free variable α. If we valuate α by some closed pattern p and interpret $[\![\boldsymbol{B}(\alpha)]\!]$ by the set of terms whose normal forms are neutral or match p, then the only possible choice for p is $\_$. Clearly this does not give us the most precise possible semantics for t, as it also includes terms such as $u = \mathsf{Node}\ (\mathsf{Node}\ x\ y)\ \mathsf{Leaf}$. However we need precise semantics if we are to capture the information needed for the dependency analysis: if we take the constructor term $l = \mathsf{Node}\ \mathit{Leaf}\ x$, then a reduct of t does match l, but this can never happen for u. To give sufficiently precise semantics to terms, we therefore need to interpret pattern variables with *sets of closed patterns*. In this case we will interpret α by the set $\{\mathsf{leaf}, \mathsf{node}(\mathsf{leaf}, \mathsf{leaf})\}$ to capture the most precise semantics possible for t.

We define the interpretation of types, and prove that they satisfy the Girard conditions. We then show that correctness of the defined function symbols implies correctness of the semantics.

Definition 7 A *value* is a term $v \in \mathcal{T}rm^{\lfloor\cdot\rfloor}$ of the form:

– $\lambda x.t$

– Node $t\ u$
– Leaf

For any $t \in \mathcal{T}rm^{|\cdot|}$ we say v is a *value of* t if $t \to^*_{\mathcal{R}\cup\beta} v$ and v is a value.

A term is *neutral* if it is not a value, and is *hereditarily neutral* if it has no values.

Definition 8 Let $\mathcal{P}_c$ be the set of *closed* patterns, and $\mathcal{NF}$ is the set of $\mathcal{R}\beta$-*normal forms* in $\mathcal{T}rm^{|\cdot|}$. The *term matching relation* $\ll\!\downarrow \subseteq \mathcal{NF} \times \mathcal{P}_c$ is defined in the following way:

– $v \ll\!\downarrow \_$
– $v \ll\!\downarrow p$ if v is neutral.
– $v \ll\!\downarrow \mathsf{node}(p, q)$ if $v = \mathsf{Node}\ v_1\ v_2$ with $v_1 \ll\!\downarrow p \wedge v_2 \ll\!\downarrow q$.
– $v \ll\!\downarrow \mathsf{leaf}$ if $v = \mathsf{Leaf}$.

A *pattern valuation*, or *valuation* if the context is clear, is a *partial* function with finite support from pattern variables $\mathcal{V}$ to *non-empty* sets of *closed* patterns. If p is a pattern, θ is a pattern valuation and $\mathcal{FV}(p) \subseteq \mathsf{dom}(\theta)$ then $p\theta$ is the set defined inductively by:

– $\alpha\theta = \theta(\alpha)$
– $\mathsf{leaf}\,\theta = \mathsf{leaf}$
– $\_\theta = \_$
– $\bot\theta = \bot$
– $\mathsf{node}(p_1, p_2)\theta = \{\mathsf{node}(q_1, q_2) \mid q_1 \in p_1\theta \wedge q_2 \in p_2\theta\}$

We may write $p\theta = \{p \mid \alpha_1 \leftarrow \theta(\alpha_1), \ldots, \alpha_n \leftarrow \theta(\alpha_n)\}$, using inspiration from *list comprehension* notation (as in Berger [Ber05]). If $\alpha \notin \mathsf{dom}(\theta)$ and P is a non-empty set of closed patterns, we write θ^{al}_P for the valuation that sends $\beta \in \mathsf{dom}(\theta)$ to $\theta(\beta)$ and α to P. Notice that $p\theta$ is a set of *closed* patterns.

Finally if θ is a valuation and t is a term in $\mathcal{SN}$, we write $t \ll\!\downarrow p\theta$ if for every normal form v of t:

$$\exists q \in p\theta,\ v \ll\!\downarrow q$$

The *type interpretation* $[\![\_]\!]\_$ is a function that to each $T \in \mathcal{T}$ and each valuation θ such that $\mathcal{FV}(T) \subseteq \mathsf{dom}(\theta)$ associates a set $[\![T]\!]_\theta \subseteq \mathcal{SN}_{\mathcal{R}\cup\beta}$. We define it by induction on the structure of T:

– $[\![\boldsymbol{B}(p)]\!]_\theta = \{t \in \mathcal{B} \mid t \ll\!\downarrow p\theta\}$
– $[\![T \to U]\!]_\theta = \{t \in \mathcal{SN} \mid \forall u \in [\![T]\!]_\theta, t\ u \in [\![U]\!]_\theta\}$
– $[\![\forall\alpha.T]\!]_\theta = \{t \in \mathcal{SN} \mid \forall P, t \in [\![T]\!]_{\theta^\alpha_P}\}$

Where $\mathcal{B}$ is the smallest set that verifies:

$$\mathcal{B} = \{t \in \mathcal{SN} \mid \forall v \text{ a value of } t, v = \mathsf{Leaf} \vee v = \mathsf{Node}\ t_1\ t_2 \wedge t_1, t_2 \in \mathcal{B}\}$$

The next step in the reducibility proof is to verify that the interpretation of terms verify the *Girard conditions*: A subset $X \subseteq \mathcal{T}rm^{|\cdot|}$ satisfies the Girard conditions if

1. strong normalization: $X \subseteq \mathcal{SN}$
2. stability by reduction: for every term $t \in X$, if u is such that $t \to^* u$, then $u \in X$.
3. "sheaf condition": if t is *neutral*, and for every term u such that $t \to u$, $u \in X$, then $t \in X$.

These embody the exact combinatorial properties required to carry through the inductive proof of correctness, namely that every well-typed term is in the interpretation of its type.

Lemma 9 If $T \in \mathcal{T}$ is a type, then for every valuation θ,

$$[\![T]\!]_\theta \text{ satisfies the girard conditions}$$

Proof. We proceed by induction on the structure of T.

- $T = \boldsymbol{B}(p)$
  - Strong normalization: by definition of $\mathcal{B}$.
  - Stability by reduction. Suppose that $t \in [\![\boldsymbol{B}(p)]\!]_\theta$. If $t \to^* u$, then the set of normal forms of u is contained in the set of normal forms of t.
  - Sheaf condition. Suppose that t is neutral and that each one step reduct of t is in $[\![\boldsymbol{B}(p)]\!]_\theta$. Now either t is in normal form, and then $t \ll\downarrow p\theta$ (as it is non empty), or for every one step reduct u of t, $u \ll\downarrow p\theta$. But in this case every normal form of t is the normal form of some $t \to u$, and thus $t \ll\downarrow p\theta$.
- $T = T_1 \to T_2$
  - Strong normalization: by definition.
  - Stability by reduction. Simple application of induction hypothesis.
  - Sheaf condition: Let t be neutral and suppose that $t' \in [\![T \to U]\!]_\theta$ for every t' a reduct of t. Let u be an arbitrary element of $[\![T]\!]_\theta$. Then $t'\ u' \in [\![U]\!]_\theta$ for every reduct $u \to^* u'$, by definition of the interpretation and stability by reduction. By induction hypothesis, this implies that $t\ u \in [\![U]\!]_\theta$, as it is again a neutral term. As u was chosen arbitrarily, then t is in $[\![T \to U]\!]_\theta$.
- $T = \forall\alpha.U$
  - Strong normalization: by definition.
  - Stability by reduction: Let $t \in [\![\forall\alpha.U]\!]_\theta$. We have for every set P of closed terms, $t \in [\![U]\!]_{\theta^\alpha_P}$. By induction, every reduct u of t is also in $[\![U]\!]_{\theta^\alpha_P}$. As P was chosen arbitrarily, u is also in $[\![\forall\alpha.U]\!]_\theta$.
  - Sheaf condition. Let t be neutral and suppose that one step reducts of t are in $[\![\forall\alpha.U]\!]_\theta$. Take an arbitrary P. Every reduct of t is in $[\![U]\!]_{\theta^\alpha_P}$. By induction hypothesis, t is in $[\![U]\!]_{\theta^\alpha_P}$, from which we may conclude.

■

Now we give the conditional correctness theorem, which states that if the function symbols belong to the interpretation of their types, then so does every well-typed term.

Definition 10 Let θ be a pattern valuation, σ a substitution from term variables to erased terms, and Γ a context. We say that (θ, σ) *validates* Γ, and we write $\sigma \models_\theta \Gamma$, if the set of free pattern variables in Γ is contained in $\mathsf{dom}(\theta)$, and if for every $x \in \mathsf{dom}(\Gamma)$

$$\sigma(x) \in [\![\Gamma x]\!]_\theta$$

Likewise, we write $\sigma \models_\theta t : T$ if $\mathcal{FV}(t) \subseteq \mathsf{dom}(\sigma)$, $\mathcal{FV}(T) \subseteq \mathsf{dom}(\theta)$ and

$$|t|\sigma \in [\![T]\!]_\theta$$

Theorem 11 Suppose that for each $f \in \Sigma$ and each valuation θ,

$$f \in [\![\tau_f]\!]_\theta$$

then for every context Γ, term t and type T, if $\Gamma \vdash t : T$

$$\forall(\theta, \sigma),\ \sigma \models_\theta \Gamma \Rightarrow \sigma \models_\theta t : T$$

We need the classic *substitution lemma* for types:

Lemma 12 For every patterns q, p and valuation θ, if α is not in the domain of θ, then

$$p\{\alpha \mapsto q\}\theta = p\theta^{\alpha}_{q\theta}$$

Proof. We proceed by induction on the structure of p:

- $p = \alpha$: trivial.
- $p = \beta \neq \alpha$: We have $p\{\alpha \mapsto q\} = \beta$ and therefore $p\{\alpha \mapsto q\}\theta = \theta(\beta) = \theta^{\alpha}_{q\theta}(\beta)$.
- $p = \mathsf{leaf}, \_, \bot$: trivial.
- $p = \mathsf{node}(p_1, p_2)$: We have $p\{\alpha \mapsto q\}\theta = \mathsf{node}(p_1\{\alpha \mapsto q\}, p_2\{\alpha \mapsto q\})\theta$. But this last term is equal to

$$\{\mathsf{node}(q_1, q_2) \mid q_i \in p_i\{\alpha \mapsto q\}\theta, i = 1, 2\}$$

which by induction is equal to

$$\{\mathsf{node}(q_1, q_2) \mid q_i \in p_i\theta^{\alpha}_{q\theta}, i = 1, 2\}$$

which allows us to conclude.

■

Lemma 13 (substitution lemma)
Let T be a type and θ a pattern valuation. If α does not appear in the domain of θ then:

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

Proof. We proceed by induction on the type.

– Atomic case:

$$[\![\boldsymbol{B}(q)\{\alpha \mapsto p\}]\!]_\theta = \{t \in \mathcal{SN} \mid t \ll\downarrow q\{\alpha \mapsto p\}\theta\}$$

But by lemma 12, $q\{\alpha \mapsto p\}\theta = p\theta^{\alpha}_{p\theta}$, 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:

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

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:

$$\{t \in \mathcal{SN} \mid \forall Q, t \in [\![T]\!]_{\theta^{\alpha}_{p\theta}{}^{\beta}_{Q}}\} = [\![\forall\beta.T]\!]_{\theta^{\alpha}_{p\theta}}$$

Which concludes the argument.

■

We may easily generalize this result to:

Corollary 14 Let T be a type. If ϕ is a substitution, and θ is a valuation 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 valuation defined by $\theta \circ \phi(\alpha) = \phi(\alpha)\theta$.

Another useful lemma states that type interpretations only depend on the value of the pattern substitutions in the free variables of the type.

Lemma 15 Let T be some type and θ, θ' be two closed pattern substitutions. If $\theta(\alpha) = \theta'(\alpha)$ for every $\alpha \in \mathcal{FV}(T)$, then $[\![T]\!]_\theta = [\![T]\!]_{\theta'}$.

Proof. Straightforward induction on T.

The next lemmas show correctness of the interpretation with respect to subtyping.

Definition 16 Let P and Q be sets of closed patterns. We write $P \ll Q$ if for each $p \in P$, there is a $q \in Q$ such that $p \ll q$.

Lemma 17 Let θ be a pattern valuation. If $p \ll q$, then $p\theta \ll q\theta$

Proof. Induction on the derivation of $p \ll q$. The only interesting case is $\mathsf{node}(p_1, p_2) \ll \mathsf{node}(q_1, q_2)$ with $p_i \ll q_i$ for $i = 1, 2$. In that case, if $r \in \mathsf{node}(p_1, p_2)\theta$, we have $r = \mathsf{node}(r_1, r_2)$ with $r_i \in p_i\theta$ for $i = 1, 2$. By induction hypothesis, there is r'_1, r'_2 in $\mathsf{node}(q_1, q_2)\theta$ such that $r_i \ll r'_i$ for each i. Then we take $\mathsf{node}(r'_1, r'_2) \in \mathsf{node}(q_1, q_2)\theta$ to conclude.

■

Lemma 18 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, and every non-empty set of closed patterns P and Q, if $P \ll Q$, then $t \ll\downarrow P \Rightarrow t \ll\downarrow Q$. This follows from the following fact: if v is in normal form and $r \ll s$, then

  $$v \ll\downarrow r \Rightarrow v \ll\downarrow s$$

  To show this we proceed by induction on the $\ll$ judgement. The first three cases are easy. In the fourth case, $v \ll\downarrow \mathsf{node}(r_1, r_2)$ which by definition implies that $v = \mathsf{Node}\ v_1\ v_2$, with $v_1 \ll\downarrow r_1$ and $v_2 \ll\downarrow r_2$. We can then conclude by the induction hypothesis.
  Now using lemma 17, we have, if $p \ll q$, $t \ll\downarrow p\theta \Rightarrow t \ll\downarrow q\theta$.

  Now let $t \in [\![\boldsymbol{B}(p)]\!]_\theta$, we have by definition $t \ll\downarrow p\theta$, and by the previous remark, $t \ll\downarrow q\theta$ which implies $t \in [\![\boldsymbol{B}(q)]\!]_\theta$.
- 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 $[\![T_1 \to U_1]\!]_\theta$), $t\ u$ 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 set of closed patterns, and suppose that α is a variable not appearing in the domain of θ. We then have

  $$t \in [\![T]\!]_{\theta^\alpha_P}$$

  Since $\forall\alpha.T \leq \forall\alpha.U$, we have $T \leq U$. The induction hypothesis gives:

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

  for all valuations θ'. Take θ' to be θ^α_P. We have:

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

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

■

We shall also need the fact that given T and a valuation θ, then $[\![T]\!]_\theta$ is included in $[\![T]\!]_{\theta'}$ if θ' is a *weakening* of θ on the variables in *positive position* in T.

Lemma 19 Let T be a type and θ, θ' two pattern valuations. 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

$$[\![T]\!]_\theta \subseteq [\![T]\!]_{\theta'}$$

Conversely if $\theta(\alpha) \ll \theta'(\alpha)$ for every free variable α in a *negative* position, then

$$[\![T]\!]_{\theta'} \subseteq [\![T]\!]_\theta$$

Proof. First notice that if p is a pattern, then $p\theta \ll p\theta'$, by a simple induction on p. We prove both propositions simultaneously by induction on T:

- $T = \boldsymbol{B}(p)$. All variables of p appear positively in T. Then by the above remark, $p\theta \ll p\theta'$, and therefore $[\![\boldsymbol{B}(p)]\!]_\theta \subseteq [\![\boldsymbol{B}(p)]\!]_{\theta'}$.
- $T = T_1 \to T_2$. We treat the positive case. We have by induction hypothesis $[\![T_1]\!]_{\theta'} \subseteq [\![T_1]\!]_\theta$, as all variable of T_1 that appear positively in T appear negatively in T_1, and $[\![T_2]\!]_\theta \subseteq [\![T_2]\!]_{\theta'}$. Therefore, by definition of $[\![T_1 \to T_2]\!]_\phi$, we have:
  $$[\![T_1 \to T_2]\!]_\theta \subseteq [\![T_1 \to T_2]\!]_{\theta'}$$
  The negative case is treated in the same fashion.
- $T = \forall \alpha.U$: straightforward induction.

■

We can now prove the correctness of the interpretation relative to that of the function symbols (theorem 11).

Proof. We proceed by induction on the typing derivation.

- ax: by definition of $\sigma \models_\theta \Gamma$.
- t-lam: By induction hypothesis, for all σ', θ' such that $\sigma' \models_{\theta'} \Gamma, x\colon T$, $t\sigma'$ is in $[\![U]\!]_{\theta'}$. By definition of $[\![T \to U]\!]_\theta$, we need to show that for any $u \in [\![T]\!]_\theta$, $(\lambda x\colon T.t)\sigma u$ is in $[\![U]\!]_\theta$. Now as this term is neutral, it suffices to show that every reduct is in $[\![U]\!]_\theta$. We proceed by well founded induction on the reducts of t and u. Thus if $(\lambda x\colon T.t)\sigma u \to (\lambda x\colon T.t')\sigma u'$ with $t \to t'$ or $u \to u'$, then we may conclude by well-founded induction hypothesis. The remaining case is $(\lambda x\colon T.t)\sigma u \to t\sigma\{x \mapsto u\}$. To show that this is in $[\![U]\!]_\theta$, we apply the main induction hypothesis with $\sigma' = \sigma^x_u$ and $\theta' = \theta$.
  It can be argued that this argument is the fundamental combinatory explanation for normalization of β-reduction.
- 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 a set of closed patterns. 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 15. We may therefore conclude that $|t|\sigma$ is in $[\![T]\!]_{\theta^\alpha_P}$.
- leaf-intro: Clear by definition of $[\![\boldsymbol{B}(\mathsf{leaf})]\!]_\theta$

– node-intro: let t, u be terms in $[\![\boldsymbol{B}(\alpha)]\!]_\theta$ and $[\![\boldsymbol{B}(\beta)]\!]_\theta$, respectively. The normal forms of $\mathsf{Node}\ t\ u$ are of the form $\mathsf{Node}\ t'\ u'$, with t' and u' normal forms of t and u, respectively. Therefore, to check if $\mathsf{Node}\ t\ u \ll\downarrow \mathsf{node}(\theta(\alpha), \theta(\beta))$, it suffices to check $t \ll\downarrow \theta(\alpha)$ and $u \ll\downarrow \theta(\beta)$, both of which are true by hypothesis.
– t-app: straightforward by the 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 13), $|t|\sigma \in [\![T\{x \mapsto p\}]\!]_\theta$, therefore

$$|t\ p|\sigma \in [\![T\{x \mapsto p\}]\!]_\theta$$

– symb: By hypothesis.
– sub: By application of the correctness of subtyping (lemma 18), and the induction hypothesis.

■

Now it remains to show that each function symbol is computable. By analogy with the first-order dependency pair framework, we need to build an order on terms that is in relation to the approximated dependency graph. Then sequences of decreasing terms will be the analogue of *chains*, and we will show that there can be no infinite decreasing sequences. Instead of actual terms, it is more convenient, when dealing with higher-order rewriting, to order tuples of terms labeled by a head function symbol, *i.e.* instead of having $f\boldsymbol{t} > g\boldsymbol{u}$ we have $(f, \boldsymbol{t}) > (g, \boldsymbol{u})$. The reason for this is that recursive calls in the right-hand side of rewrite rules needn't be applied to all their arguments. We will therefore need a way of using typing to "predict" which arguments may be applied, using the order on tuples as above.

However it is quite subtle to build this order in practice: indeed, a natural candidate for such an order is (the transitive closure of) the order defined by $(f, \boldsymbol{t}) > (g, \boldsymbol{u})$ if and only if

$$\exists \theta, \phi,\ f^\sharp(p_1, \ldots, p_n) \to g^\sharp(q_1, \ldots, q_m) \in \mathcal{G},\ \forall i, j,\ t_i \in [\![\boldsymbol{B}(p_i)]\!]_\theta \wedge u_j \in [\![\boldsymbol{B}(q_j)]\!]_\phi$$

This would allow us to easily build the relation between the graph and the order, and show that each call induces a decrease in this order. Sadly, this order may not be well founded even in the event that the termination criterion is satisfied. Consider for example the rule $f\ \mathsf{node}(\alpha,\beta)\ (\mathsf{Node}\ x\ y) \to f\ \alpha\ x$, typeable in the context $\Gamma = x{:}\boldsymbol{B}(\alpha), y{:}\boldsymbol{B}(\beta)$. Given the above definition, we have $(f, t) > (f, u)$ provided that there are closed p and q such that $t \ll\downarrow p$ and $u \ll\downarrow q$. But then we may take $p = q = \_$ and if $t = z$ and $u = z$ with z a variable, then $(f, z) > (f, z)$. The rewrite system does satisfy the criterion, as $\mathsf{node}(\alpha,\beta) \rhd \alpha$, but the order is not well founded.

One possible solution is to restrict the reduction to call-by value on closed terms, where a reduction in $\mathcal{R}$ can occur only if the arguments to the defined function are in normal form, and values (although β-reduction can occur at any moment). However we strive for more generality.

Another solution, in the previous example, is to impose the condition that t must be equal to $\mathsf{Node}\ t_1\ t_2$, which makes the counter-example invalid. However,

we still do not have any necessary relationship between t and u, and we may take in particular $t = \mathsf{Node}\ x\ y$ and $u = \mathsf{Node}\ x\ y$, which again results in a non well founded sequence. The solution is to take, instead of just a particular instance of the pattern variables, the *most general* possible instance.

Definition 20 Take the set $\mathcal{P}_{min}$ of *minimal patterns* to be the subset of $\mathcal{P}$ defined by:

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

Let t be a term in normal form. We inductively define the *pattern form* $\mathit{pat}(t)$ of t inductively:

- $\mathit{pat}(t) = \bot$ if t is neutral.
- $\mathit{pat}(\mathsf{Leaf}) = \mathsf{leaf}$
- $\mathit{pat}(\mathsf{Node}\ t\ u) = \mathsf{node}(\mathit{pat}(t), \mathit{pat}(u))$
- $\mathit{pat}(t) = \_$ otherwise.

We define the partial *type matching* function $\mathsf{match}_{\mathcal{P}}$ that takes terms $t_1, \ldots, t_n$ in $\mathcal{T}rm^{\lfloor\cdot\rfloor}$, and minimal patterns $p_1, \ldots, p_n$ in $\mathcal{P}_{min}$ and returns a pattern valuation:

- if $p_1 = \alpha_1, \ldots, p_n = \alpha_n$ and $t_i = t_j$ whenever $\alpha_i = \alpha_j$, then

$$\mathsf{match}_{\mathcal{P}}(\boldsymbol{t}, \boldsymbol{p})(\alpha_i) = t_i$$

- if $p_i = \mathsf{node}(q_1, q_2)$ and $t_i = \mathsf{Node}\ u_1\ u_2$ then

$$\mathsf{match}_{\mathcal{P}}(\boldsymbol{t}; \boldsymbol{p}) = \mathsf{match}_{\mathcal{P}}(t_1, \ldots, t_{i-1}, u_1, u_2, t_{i+1}, \ldots, t_n\ ;\ p_1, \ldots, p_{i-1}, q_1, q_2, p_{i+1}, \ldots, p_n)$$

- if $p_i = \mathsf{leaf}$ and $t_i = \mathsf{Leaf}$ then

$$\mathsf{match}_{\mathcal{P}}(\boldsymbol{p}, \boldsymbol{t}) = \mathsf{match}_{\mathcal{P}}(t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n\ ;\ p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n)$$

- $\mathsf{match}_{\mathcal{P}}$ is undefined in other cases.

The type matching can be seen as a way of giving the most precise possible valuation for terms that match some left-hand side of a rule. Notice that for each $f^\sharp(\boldsymbol{p}) \to g^\sharp(\boldsymbol{q}) \in \mathcal{G}$, each p_i is in $\mathcal{P}_{min}$. Indeed, an examination of the minimal typing rules show that only minimal patterns may appear in types.

Note also that if $\mathsf{match}(\boldsymbol{t}, \boldsymbol{p}) = \theta$, then for each i, $t_i \ll\!\downarrow\ p_i\theta$, by a simple induction.

Definition 21 A *link* is a tuple $(n, \boldsymbol{t}, \boldsymbol{u})$ such that

- $\boldsymbol{t}, \boldsymbol{u} \in \mathcal{SN}$
- $n = f^\sharp(\boldsymbol{p}) \to g^\sharp(\boldsymbol{q}) \in \mathcal{G}$
- $\mathsf{match}_{\mathcal{P}}(\boldsymbol{t}, \boldsymbol{p})$ is defined, and if it is equal to θ, then

$$\forall j, u_j \ll\!\downarrow q_j\theta'$$

For some extension θ' of θ such that $\mathcal{FV}(\boldsymbol{q}) \subseteq \mathsf{dom}(\theta')$.

A *chain* is an eventually infinite sequence $c_1, c_2, \ldots$ of links such that if $c_i = (n_i, \boldsymbol{t}_i, \boldsymbol{u}_i)$, then for each i,

$$\boldsymbol{u}_i \to^* \boldsymbol{t}_{i+1}$$

and if $n_i = f_i^\sharp(\boldsymbol{p}) \to g_i^\sharp(\boldsymbol{q})$ then $g_i = f_{i+1}$.

Notice that if $\mathcal{FV}(\boldsymbol{q}) \subseteq \mathcal{FV}(\boldsymbol{p})$ then we may take $\theta' = \theta$ in the definition of chains.

We first need to show a correspondence between the chains and the graph, that is:

Lemma 22 For each chain $c_1, c_2, \ldots$ such that $c_i = (n_i, \boldsymbol{t}_i, \boldsymbol{u}_i)$, there is a path $n_1 \to n_2 \to \ldots$ in $\mathcal{G}_\mathcal{R}$.

Proof. It suffices to show that if $c_1 = (n_1, \boldsymbol{t}, \boldsymbol{u}), c_2 = (n_2, \boldsymbol{u}', \boldsymbol{v})$ is a chain, then there is an edge between $n_1 = f^\sharp(\boldsymbol{p}) \to g^\sharp(\boldsymbol{q})$ and $n_2 = g^\sharp(\boldsymbol{r}) \to h^\sharp(\boldsymbol{s})$. First note that the variables of $\boldsymbol{q}$ and $\boldsymbol{r}$ are distinct by hypothesis. Notice that for each i, $u_i \ll\!\downarrow\ q_i\theta$ for some t and $\mathsf{match}_\mathcal{P}(r_i, u_i')$ is defined. We need to prove for each i that $q_i \bowtie r_i$. As $u_i \to^* u_i'$, all normal forms of u_i' are also normal forms of u_i. We proceed by induction on $\mathsf{match}_\mathcal{P}(r_i, u_i')$.

- r_i is a variable. We can conclude immediately by the definition of $\bowtie$, as a fresh variable can unify with any pattern.
- $u_i' = \mathsf{Leaf}$ and $r_i = \mathsf{leaf}$. In this case Leaf is a normal form of u_i, so there is some $q' \in q_i\theta$ such that $\mathsf{Leaf} \ll\!\downarrow q'$. From this it follows that q_i is either leaf, $\_$ or some variable. This allows us to conclude that $q_i \bowtie \mathsf{leaf}$.
- $u_i' = \mathsf{Node}\ u_i'^1\ u_i'^2$ and $r_i = \mathsf{node}(r_i^1, r_i^2)$. Now let us examine q_i. We may exclude the cases $q_i = \mathsf{leaf}$ and $q_i = \bot$, as every normal form of u_i' is a normal form of u_i and is of the form $\mathsf{Node}\ v\ v'$. In the case $q_i = \alpha$ or $q_i = \_$ we may easily conclude. The only remaining case is $q_i = \mathsf{node}(q_i^1, q_i^2)$. From the induction hypothesis we get $q_i^1 \bowtie r_i^1$ and $q_i^2 \bowtie r_i^2$, which imply $q_i \bowtie r_i$

■

If the conditions of the termination theorem are satisfied, the there are no infinite chains, in the same way as for the first-order dependency pair approach.

Theorem 23 Suppose that the conditions of theorem 6 are satisfied. Then there are no infinite chains.

We need to define and establish the well foundedness of the embedding order on terms.

Definition 24 We mutually define the strict and large *embedding preorder* on erased terms in normal form $\triangleright$ and $\trianglerighteq$ by:

- $t_1 \trianglerighteq u \Rightarrow \mathsf{Node}\ t_1\ t_2 \triangleright u$
- $t_2 \trianglerighteq u \Rightarrow \mathsf{Node}\ t_1\ t_2 \triangleright u$

– $t_1 \rhd u_1 \wedge t_2 \trianglerighteq u_2 \Rightarrow \mathsf{Node}\ t_1\ t_2 \rhd \mathsf{Node}\ u_1\ u_2$
– $t_1 \trianglerighteq u_1 \wedge t_2 \rhd u_2 \Rightarrow \mathsf{Node}\ t_1\ t_2 \rhd \mathsf{Node}\ u_1\ u_2$
– $\mathsf{Leaf} \trianglerighteq \mathsf{Leaf}$
– $t \trianglerighteq u$ if t and u are neutral.
– $t \rhd u \Rightarrow t \trianglerighteq u$

Note that the preorder is *not* an order: for instance, $x \trianglerighteq y$ and $y \trianglerighteq x$.

Lemma 25 The preorder $\rhd$ is well-founded.

Proof. Given a term in normal form t, define $\mathsf{size}(t)$ inductively:

– $\mathsf{size}(\mathsf{Node}\ t_1\ t_2) = \mathsf{size}(t_1) + \mathsf{size}(t_2) + 1$
– $\mathsf{size}(t) = 0$ otherwise.

It is then easy to verify by mutual induction that if $t \rhd u$, $\mathsf{size}(t) > \mathsf{size}(u)$ and if $t \trianglerighteq u$ then $\mathsf{size}(t) \geq \mathsf{size}(u)$. Well foundedness of the order on naturals yields the desired conclusion.

■

To show that there are no infinite chains, we will exploit the fact that if $c = (n, \boldsymbol{t}, \boldsymbol{u})$ is a link, that is decreasing in the embedding order on patterns, then there is a decrease in the normal forms from $\boldsymbol{t}$ to $\boldsymbol{u}$.

To show this, we must prove that pattern-matching does indeed *completely* capture the "pattern semantics" of a term in $\mathcal{B}$.

Lemma 26 Suppose $\boldsymbol{t}$ are terms in $\mathcal{B}$ and $\boldsymbol{p}$ are minimal patterns. If $\mathsf{match}_{\mathcal{P}}(\boldsymbol{t}, \boldsymbol{p})$ is defined and equal to θ, then for each $q \in p_i\theta$, there is a normal form v of t_i such that $\mathit{pat}(v) = q$.

Proof. We proceed by induction on the definition of $\mathsf{match}_{\mathcal{P}}$:

– $p_i = \alpha_i$. In this case (as $\mathsf{match}_{\mathcal{P}}(\boldsymbol{t}, \boldsymbol{p})$ is defined) By definition $\alpha\theta$ is equal to $\{\mathit{pat}(v) \mid v \text{ is a normal form of } t_i\}$.
– $p_i = \mathsf{leaf}$. In this case $t_i = \mathsf{Leaf}$ and therefore we can take $v = \mathsf{Leaf}$.
– $p_i = \mathsf{node}(p_i^1, p_i^2)$. In this case, $t_i = \mathsf{Node}\ t_i^1\ t_i^2$. By the induction hypothesis, for any $q_1 \in p_i^1\theta$ and $q_2 \in p_i^2\theta$ there are normal forms v_1 and v_2 of t_i^1 and t_i^2 such that $q_j = \mathit{pat}(v_j)$ for $j = 1, 2$. It is easy to observe that $\mathsf{Node}\ v_1\ v_2$ is a normal form of t_i, and that $q = \mathsf{node}(q_1, q_2)$ is an element of $p_i\theta$, and $\mathit{pat}(\mathsf{Node}\ v_1\ v_2) = q$ allows us to conclude.

■

To prove that there are no infinite chains, we need to relate the decrease of the patterns to the decrease of the normal forms of the terms that appear in chains.

Lemma 27 Suppose that p and q are closed patterns such that $p \rhd q$ (respectively $p \trianglerighteq q$), and v_1, v_2 normal forms such that $\mathit{pat}(v_1) = p$ and $v_2 \ll\!\downarrow q$. Then $v_1 \rhd v_2$ (respectively $v_1 \trianglerighteq v_2$).

Proof. We prove both properties simultaneously by induction on the derivation of $p \rhd q$:

- $p = \mathsf{node}(p_1, p_2)$ and $p_1 \trianglerighteq q$. We have $v_1 = \mathsf{Node}\ u_1\ u_2$ with $pat(u_1) = p_1$. By induction hypothesis $u_1 \trianglerighteq v_2$, and therefore $\mathsf{Node}\ u_1\ u_2 \rhd v_2$.
- $p = \mathsf{node}(p_1, p_2), q = \mathsf{node}(q_1, q_2)$ with $p_1 \rhd q_1$ and $p_2 \trianglerighteq q_2$. In that case $v_1 = \mathsf{Node}\ v_1^1\ v_1^2$ and $v_2 = \mathsf{Node}\ v_2^1\ v_2^2$. The induction hypothesis gives $v_1^1 \rhd v_2^1$ and $v_1^2 \trianglerighteq v_2^2$, from which we may conclude.
- The symmetrical cases are treated in the same manner.
- $p = \mathsf{leaf}$ and $q = \mathsf{leaf}$. In this case, $v_1 = v_2 = \mathsf{Leaf}$, and $v_1 \trianglerighteq v_2$.

Lemma 28 Let $c = (n, \boldsymbol{t}, \boldsymbol{u})$ be some link such that $n = f^\sharp(\boldsymbol{p}) \to g^\sharp(\boldsymbol{q})$. Suppose that there is i such that $p_i \rhd q_i$, (respectively $p_i \trianglerighteq q_i$). Then if v is a normal form of u, there exists some normal form v' of t such that $v' \rhd v$, (respectively $v' \trianglerighteq v$).

Proof. Let $\theta = \mathsf{match}_{\mathcal{P}}(\boldsymbol{t}, \boldsymbol{p})$, which is guaranteed to exist by hypothesis. First notice that for every $\alpha \in \mathcal{FV}(\boldsymbol{p})$, $\theta(\alpha)$ does *not* contain $\_$. Indeed, given $t \in \mathcal{B}$, the normal form of t is also in $\mathcal{B}$. It can only be neutral, equal to Leaf, or in the form $\mathsf{Node}\ t_1\ t_2$ with t_i in the above form.

We treat the $\rhd$ case first. Suppose that v is a normal form of u_i. By definition, we have $u_i \ll\!\downarrow q_i\theta$, which means by definition that there is some $r \in q_i\theta$ such that $v \ll\!\downarrow r$. Since $p_i \rhd q_i$, this implies that there is some $r' \in p_i$ such that $r' \rhd r$. We have by lemma 26 that there exists some v' a normal form of t_i such that $pat(v') = r'$, which allows us to conclude using lemma 27.

■

We finally have all the tools to give the proof of well foundedness of chains.

Proof. of theorem 23.

By contradiction, let $c_1, c_2, \ldots$ be an infinite chain, such that for each i, $c_i = (n_i, \boldsymbol{t}_i, \boldsymbol{u}_i)$. By lemma 22, $n_1, n_2, \ldots$ is an infinite path in $\mathcal{G}$. By finiteness of $\mathcal{G}$, there is some SCC $\mathcal{G}'$ and some natural number k such that $n_k, n_{k+1}, \ldots$ is contained in $\mathcal{G}'$. By hypothesis, if $n_i = f_i^\sharp(\boldsymbol{p}^i) \to g_i^\sharp(\boldsymbol{q}^i)$, there is an index j such that for each i, $p_j^i \trianglerighteq q_j^i$ or $p_j^i \rhd p_i^j$. Furthermore, again by hypothesis, there are an infinite number of indexes i such that $p_j^i \rhd q_j^i$. Let $V_i = \{v \mid v \text{ is a normal form of } t_j^i\}$ and $U_i = \{v \mid v \text{ is a normal form of } u_j^i\}$. We apply lemma 28 to show that for each $v_i' \in U_i$ there exists $v_i \in V_i$ such that $v_i \rhd v_i'$ for these indexes and $v_i \trianglerighteq v_i'$ for the others.

We wish to show that there is an infinite chain $v_1, v_2, \ldots$ such that $v_i \trianglerighteq v_{i+1}$ for each i and $v_i \rhd v_{i+1}$ for an infinite number of indexes i, contradicting well-foundedness of $\rhd$ (lemma 25).

To do this we first notice that $V_{i+1} \subseteq U_i$, as $\boldsymbol{u}_i \to^* \boldsymbol{t}_{i+1}$. Then we build the following tree:

- We have a node at the top, connected to every element of V_k.
- We have a node between $a \in V_i$ and b in U_i if $a \trianglerighteq b$ or $a \rhd b$.
- We have a node between $a \in U_i$ and $b \in V_{i+1}$ if $a = b$.

Notice first that every V_i, U_i is finite, as the rewrite system is finite (each strongly normalizing term therefore has a finite number of normal forms). We wish to apply *König's lemma* which states: every finitely branching infinite tree has an infinite path. It is easy to see that the tree is finitely branching: every V_i and U_i is finite, and it is equally easy to verify that the tree is infinite, as no V_i or U_i is empty (the t_i and u_i are strongly normalizing and therefore have at least one normal form). This give us the existence of an infinite path in the tree, which concludes the proof.

■

To prove that the function symbols are in the interpretation of their type, we shall (obviously) need to consider the rewrite rules. In particular, we need to relate the minimal typing used to derive the types of left hand sides and pattern matching, in order to prove that our notion of chain is the correct one.

Lemma 29 Suppose that Γ is a context, that $l_1, \ldots, l_k$ are constructor terms and that $\Gamma \vdash_{min} l_1 : \boldsymbol{B}(p_1), \ldots, \Gamma \vdash_{min} l_k : \boldsymbol{B}(p_k)$. Suppose that $t_1, \ldots, t_k$ *match* $l_1, \ldots, l_k$. Then $\mathsf{match}_{\mathcal{P}}(\boldsymbol{t}, \boldsymbol{p})$ is defined.

Proof. We proceed by induction on the structures of l_i (matching the cases of the $\mathsf{match}_{\mathcal{P}}$ judgement)

- $l_1 = x_1, \ldots, l_n = x_n$. In this case, the only applicable case for $\vdash_{min}$ is the variable case. If $x_i = x_j$, then $t_i = t_j$. Furthermore $p_i = \alpha_i$ for some variable α_i and again, $\alpha_i = \alpha_j$ if and only if $x_i = x_j$, by linearity of α_i and α_j in Γ. Therefore if $\alpha_i = \alpha_j$, then $t_i = t_j$, and $\mathsf{match}_{\mathcal{P}}(\boldsymbol{t}, \boldsymbol{p})$ is defined.
- $l_i = \mathsf{Leaf}$. In this case the only applicable rule is the leaf rule, and $p_i = \mathsf{leaf}$ and $t_i = \mathsf{Leaf}$. By induction $\mathsf{match}_{\mathcal{P}}(\boldsymbol{t}, \boldsymbol{p})$ is defined.
- $l_i = \mathsf{Node}\ l_i^1\ l_i^2$. In this case we apply the node rule, and we have $p_i = \mathsf{node}(p_i^1, p_i^2)$. Again, we have $t_i = \mathsf{Node}\ t_i^1\ t_i^2$, and we may conclude by the induction hypothesis.

■

Our reason for defining pattern matching is to provide the "closest" possible pattern semantics for a term. In fact we have the following result, which states that any valuation θ such that t is in $[\![\boldsymbol{B}(\alpha)]\!]_\theta$ can be "factored through" $\mathsf{match}(t, p)$:

Lemma 30 Suppose that $\boldsymbol{t}$ is a tuple of strongly normalizing terms, that $\boldsymbol{\alpha}$ is a tuple of pattern variables, and θ' is a valuation that verifies:

$$\forall i, t_i \in [\![\boldsymbol{B}(\alpha_i)]\!]_{\theta'}$$

Suppose in addition that $\boldsymbol{p}$ are minimal patterns such that $\mathsf{match}_{\mathcal{P}}(\boldsymbol{t}, \boldsymbol{p})$ is defined and equal to θ. Let ϕ be the substitution that sends α_i to p_i. Then

$$\forall i, \theta \circ \phi(\alpha_i) \ll \theta'(\alpha_i)$$

Proof. We proceed by induction on the judgment $\mathsf{match}_{\mathcal{P}}(\boldsymbol{t}, \boldsymbol{p})$.

- $p_i = \beta_i$ for each p_i, and therefore $\phi(\alpha_i) = \beta_i$. In that case, $\theta \circ \phi(\alpha_i) = \{pat(v) \mid v$ normal form of $t_i\}$. Furthermore, $t_i \ll\downarrow \theta'(\alpha_i)$. Take some v a normal form of t_i. We have some $q \in \theta'(\alpha_i)$ such that $v \ll\downarrow q$. We then verify that $pat(v) \ll q$, which implies $\theta \circ \phi(\alpha_i) \ll \theta'(\alpha_i)$
- $p_i = \mathsf{leaf}$. In this case, $\theta' \circ \phi(\alpha_i) = \mathsf{leaf}$. By $t_i \ll\downarrow \theta'(\alpha_i)$ and $t_i = \mathsf{Leaf}$, we have that $\theta'(\alpha_i)$ contains leaf or $\_$, and in each case we can conclude.
- $p_i = \mathsf{node}(p_i^1, p_i^2)$. In this case, $t_i = \mathsf{Node}\ t_1^1\ t_i^2$, and

$$\theta \circ \phi(\alpha_i) = \{\mathsf{node}(r_1, r_2) \mid r_1 \in p_i^1\theta \wedge r_1 \in p_i^2\theta\}$$

  By $t_i \ll\downarrow \theta'(\alpha_i)$ we have for each normal form v of t_i some q in $\theta'(\alpha_i)$ such that $v \ll\downarrow q$. In addition v is of the form $\mathsf{Node}\ v_1\ v_2$, where v_1 is a normal form of t_i^1 and v_2 is a normal form of t_i^2. From this we get that either $q = \_$, in which case we are done, or $q = \mathsf{node}(q_1, q_2)$ with $v_1 \ll\downarrow q_1$ and $v_2 \ll\downarrow q_2$. In this case we apply the induction hypothesis to deduce that there is some $r_1 \in p_i^1\theta$ and $r_2 \in p_i^2\theta$ such that $r_1 \ll q_1$ and $r_2 \ll q_2$, and thus $\mathsf{node}(r_1, r_2) \ll \mathsf{node}(q_1, q_2)$.

■

Definition 31 We define the following order $>_{dp}$ on pairs $(f, \boldsymbol{t})$ with $f \in \Sigma$ and $\boldsymbol{t}$ a tuple of terms:

$$(f, \boldsymbol{t}) >_{dp} (g, \boldsymbol{u}) \Leftrightarrow \exists \boldsymbol{t'}, n = f^\sharp(\boldsymbol{p}) \to g^\sharp(\boldsymbol{q}),\ \boldsymbol{t} \to^* \boldsymbol{t'} \wedge (n, \boldsymbol{t'}, \boldsymbol{u}) \text{ is a link}$$

That is, if $\boldsymbol{t}$ reduces to $\boldsymbol{t'}$ such that there is a link between $\boldsymbol{t'}$ and $\boldsymbol{u}$, and where the associated node corresponds to a call from f to g.

Lemma 32 If the conditions of theorem 6 are satisfied then the order $>_{dp}$ is well-founded.

Proof. Any infinite decreasing sequence $(f_1, \boldsymbol{t}_1) > (f_1, \boldsymbol{t}_2) > \ldots$ gives rise to an infinite chain, which is not possible by theorem 23.

We have enough to prove the main theorem, that is correctness of defined symbols.

Theorem 33 Suppose that the conditions of theorem 6 are satisfied. Then for each $f \in \Sigma$ and each valuation θ, $f \in [\![\tau_f]\!]$.

Proof. Suppose that $\tau_f = \forall \boldsymbol{\alpha}.\boldsymbol{B}(\alpha_1) \to \ldots \to \boldsymbol{B}(\alpha_k) \to T_f$. Take θ a valuation and $t_1, \ldots, t_n$ in $[\![\boldsymbol{B}(\alpha_1)]\!]_\theta, \ldots, [\![\boldsymbol{B}(\alpha_k)]\!]_\theta$. We need to show that

$$f\ t_1 \ldots t_n \in [\![T_f]\!]_\theta$$

Note that each t_i is strongly normalizing. We proceed first by induction on $\boldsymbol{t}$ ordered by strict reduction. As $t = f\ \boldsymbol{t}$ is neutral, it suffices to consider all the one step reducts t' of t. These reducts are of two forms:

– $t' = f\ t_1 \dots t'_i \dots t_k$ with $t_i \to t'_i$. We conclude by the induction hypothesis.
– There is some rule $l \to r \in \mathcal{R}$, and some substitution σ such that $|l|\sigma = t$, and $|r|\sigma = t'$. We then proceed by induction on $(f, \boldsymbol{t})$ ordered by $>_{dp}$. We have by hypothesis that there is some context Γ and some derivation $\Gamma \vdash_{min} l_i : \boldsymbol{B}(p_i)$ for each i, and a derivation $\Gamma \vdash r : T_f\phi$, with ϕ the substitution that sends α_i to p_i.
By lemma 29, $\mathsf{match}_{\mathcal{P}} = \psi$ is defined. We therefore have $t_i \in [\![\boldsymbol{B}(p_i)]\!]_\psi$ for each i, which gives $t_i \in [\![\boldsymbol{B}(\alpha_i)]\!]_{\psi\circ\phi}$ by the substitution lemma. By lemma 30, $\psi \circ \phi \ll \theta$. We may then apply the positivity condition of τ_f using lemma 19 to deduce that $[\![T_f]\!]_{\psi\circ\phi} \subseteq [\![T_f]\!]_\theta$. Therefore it suffices to show that t' is in $[\![T_f]\!]_{\psi\circ\phi}$, which is equal to $[\![T_f\phi]\!]_\psi$ by the substitution lemma. By hypothesis, $\Gamma \vdash r : T_f\phi$, so we would like to apply the correctness theorem 11 to show that $t' = |r|\sigma \in [\![T_f\phi]\!]_\psi$. The correctness theorem itself can not be applied, as it takes as hypothesis the correctness of function symbols, which we are trying to prove. But we will proceed in the same manner, making essential use of the well-founded induction hypothesis.
Let us first show by induction on the derivation of $\Gamma \vdash_{min} l_i$ that for each $x \in \mathsf{dom}(\sigma)$, $\sigma(x) \in [\![\Gamma(x)]\!]_\psi$.
  - $l_i = x$. We have $\sigma(x) = t_i \in [\![\boldsymbol{B}(\gamma)]\!]_\psi$ with $\gamma = p_i$ and $\psi(\beta_i) = \{pat(v) \mid v \text{ normal form of } t_i\}$
  - $l_i = \mathsf{Leaf}$. We have nothing to show here.
  - $l_i = \mathsf{Node}\ l^1\ l^2$. Simple application of the induction hypothesis.

  Now we prove by induction on the derivation of $\Gamma \vdash r : T_f\phi$ that $|r|\sigma \in [\![T_f\phi]\!]_\psi$. We can exactly mimic the proof of theorem 11, except for the **symb** case. In this case, there is a g such that $r = g\boldsymbol{q}$, and if $\tau_g = \forall\boldsymbol{\beta}.\boldsymbol{B}(\beta_1) \to \dots \to \boldsymbol{B}(\beta_m) \to T_g$, we need to show that, for some extension ψ' of psi, $g \in [\![\boldsymbol{B}(q_1) \to \dots \to \boldsymbol{B}(q_m) \to T_g]\!]_{\psi'}$. Recall the induction hypothesis on $(f, \boldsymbol{t})$, which states that for every θ, if $(f, \boldsymbol{t}) >_{dp} (g, \boldsymbol{u})$, then $g\boldsymbol{u} \in [\![T_g]\!]_\theta$. Now take θ to be $\psi' \circ \zeta$ where ζ is the substitution that sends β_i to q_i. It suffices to show that if for $i = 1, \dots, m$ $u_i \in [\![\boldsymbol{B}(\beta_i)]\!]_{\psi'\circ\zeta}$, then $(f, \boldsymbol{t}) >_{dp} (g, \boldsymbol{u})$. For this we need to show that there exists $n \in \mathcal{G}$ such that:
  - $n = f^\sharp(\boldsymbol{r}) \to g^\sharp(\boldsymbol{s})$
  - $\mathsf{match}_{\mathcal{P}}(\boldsymbol{t}, \boldsymbol{r}) = \theta$
  - There is an extension θ' of θ such that

$$\boldsymbol{u} \ll\downarrow \sigma\theta'$$

  We just take n to be the node that corresponds to the call site of $g\boldsymbol{q}$. In this case, $\boldsymbol{r} = \boldsymbol{p}$ and $\boldsymbol{s} = \boldsymbol{q}$. By definition, $\mathsf{match}_{\mathcal{P}}(\boldsymbol{t}, \boldsymbol{p})$ is defined and equal to ψ. Then ψ' is an extension of ψ and as $u_i \in [\![\boldsymbol{B}(\beta_i)]\!]_{\psi'\circ\zeta} = [\![\boldsymbol{B}(q_i)]\!]_{\psi'}$, we have $u_i \ll\downarrow q_i\psi'$.

■

Corollary 34 Every well-typed term is in the interpretation of its type, that is

$$\forall \Gamma, t, T\ \Gamma \vdash t : T \Rightarrow |t| \in [\![T]\!]$$

Where $[\![T]\!]$ is $[\![T]\!]_\theta$ where θ is the valuation that sends every variable to the set $\{\_\}$.

Proof. In fact it does not matter which θ we choose: let θ be any valuation. Given a variable x and a type T, by lemma 9, $x \in [\![T]\!]_\theta$, as x is neutral and in normal form. Given $\Gamma \vdash t : T$, we can therefore take the substitution σ that sends every variable $x \in \mathsf{dom}(\Gamma)$ to itself. In that case $\sigma(x) \in [\![\Gamma(x)]\!]_\theta$ by the above remark, and by the combination of theorem 11 and theorem 33, $|t|\sigma \in [\![T]\!]_\theta$. But in this case $|t|\sigma = |t|$.

■

We obtain the statement of theorem 6 as a corollary: every well typed term is in the interpretation of its type, but this interpretation only contains strongly normalizing terms by lemma 9.