Partial Intersection Type Assignment in Applicative Term Rewriting Systems

Steffen van Bakel*

Department of Informatics, Faculty of Mathematics and Informatics,
University of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands.
steffen@cs.kun.nl

Abstract

This paper introduces a notion of partial type assignment on applicative term rewriting systems that is based on a combination of an essential intersection type assignment system, and the type assignment system as defined for ML [16], both extensions of Curry's type assignment system [11]. Terms and rewrite rules will be written as trees, and type assignment will consists of assigning intersection types function symbols, and specifying the way in which types can be assigned to nodes and edges between nodes. The only constraints on this system are local: they are imposed by the relation between the type assigned to a node and those assigned to its incoming and out-going edges. In general, given an arbitrary typeable applicative term rewriting system, the subject reduction property does not hold. We will formulate a sufficient but undecidable condition typeable rewrite rules should satisfy in order to obtain this property.

Introduction

In the recent years several paradigms have been investigated for the implementation of functional programming languages. Not only the lambda calculus [5], but also term rewriting systems [15] and term graph rewriting systems [7] are topics of research. Lambda calculus (or rather combinator systems) forms the underlying model for the functional programming language Miranda [22], term rewriting systems are used in the underlying model for the language OBJ [14], and term graph rewriting systems is the model for the language Clean [8, 17].

The lambda calculus, term rewriting systems and graph rewriting systems themselves are type free, whereas in programming the notion of types plays an important role. Type assignment to programs and objects is in fact a way of performing abstract interpretation that provides necessary information for both compilers and programmers. Since the lambda calculus is a fundamental basis for many functional programming languages, a type assignment system for the pure untyped lambda calculus, capable of deducing meaningful and expressive types, has been a topic of research for many years.

There exists a well understood and well defined notion of type assignment on lambda terms, known as the Curry type assignment system [11] which expresses abstraction and application. Many of the now existing type assignment systems for functional programming languages are based on (extensions of) the Curry type assignment system.

In [9, 6] the intersection type discipline for the lambda calculus is presented, an extension of Curry's type assignment system. The extension consists of allowing more than one type for term-variables and adding a type constant 'ω' and, next to the type constructor '$\rightarrow$', the type constructor '$\cap$'. This yields a type assignment system that is very powerful: it is closed under

*Partially supported by the Esprit Basic Research Action 3074 "Semagraph" and the Netherlands Organisation for the advancement of pure research (N.W.O.).

β-equality. Because of this power, type assignment in this system (and even in the system that does not contain ω, see [1]) is undecidable. The essential type assignment system as presented in this paper is a restriction of the intersection type discipline presented in [6] that satisfies all properties of that system, and is also an extension of the Curry type assignment system. The main advantage of the essential system over the intersection system is that the set of types assignable to a term is significantly smaller.

Most functional programming languages, like Miranda for instance, allow programmers to specify an algorithm (function) as a set of rewrite rules. The type assignment systems incorporated in most term rewriting languages are in fact extensions of type assignment systems for a(n extended) lambda calculus, and although it seems straightforward to generalize those systems to the (significantly larger) world of term rewriting systems, it is at first look not evident that those borrowed systems have still all the properties they possessed in the world of lambda calculus. For example, type assignment in term rewriting systems in general does not satisfy the subject reduction property: i.e. types are not preserved under rewriting, as illustrated in [4]. In order to be able to study the details of type assignment for term rewriting systems, a formal notion of type assignment on term rewriting systems is needed, that is more close to the approach of type assignment in lambda calculus than the algebraic one [12].

The aim of this paper is to present a formal notion of type assignment on term rewriting systems that is close to those defined for the lambda calculus and use intersection types.

The notion of type assignment presented here for term rewriting systems is based on both an essential type assignment system for the lambda calculus and the polymorphic type assignment system for the functional programming language ML [16]. The polymorphic aspect of type assignment can be found in the use of an environment, that provides a type for every function symbol F; for every occurrence of F the way in which its type can be obtained from the one provided by the environment is specified.

Intersection types are studied because they are a good means to perform abstract interpretation, better than Curry types, also even better than the kind of types used in languages like ML. Also, the notion of type assignment presented in this paper could be extended to the world of term graph rewriting systems, and in that world intersection types are the natural tool to type nodes that are shared. Moreover, intersection types seem to be promising for use in functional programming languages, since they seem to provide a good formalism to express overloading (see also [20]).

In this paper we define applicative term rewriting systems (ATRS), a slight extension of the term rewriting systems as defined in [15], as the term rewriting systems that contain a special binary operator Ap. The applicative term rewriting systems defined in this paper are extensions to those suggested by most functional programming languages in that they do not discriminate against the varieties of function symbols that can be used in patterns.

In [4] and [2] partial type assignment systems for (left linear) applicative term rewriting systems are presented. The system presented here can be seen as a variant of those systems; the main difference between those two systems and the one presented here are in the set of types that can be assigned to nodes and edges: Curry types in [4], intersection types of Rank 2 in [2], and strict intersection types in this one. Also, type assignment in those systems is decidable, but in the one presented in this paper it is not.

Throughout this paper, the symbol φ (often indexed, like in φ_i) will be a type-variable; when writing a type-variable φ_i, sometimes only the index i is used, so as to obtain more readable types. Greek symbols like α, β, γ, μ, ν, η, ρ, σ and τ (often indexed) will range over types. To avoid parentheses in the notation of types, '$\rightarrow$' is assumed to associate to the right – so right-most, outer-most brackets will be omitted – and, as in logic, '$\cap$' binds stronger than '$\rightarrow$'. The symbol B is used for bases.

Because of the restricted length of this paper, all results are presented without proofs.

1 Essential type assignment for the lambda calculus

In this section we present the essential type assignment system, a restricted version of the system presented in [6], together with some of its properties. The major feature of this restricted system is, compared to that system, a restricted version of the derivation rules and it is based on a set of strict types.

Strict types are the types that are strictly needed to assign a type to a term in the system of [6]. We will assume that ω is the same as an intersection over zero elements: if $n = 0$, then $\sigma_1 \cap \cdots \cap \sigma_n = \omega$, so ω does not occur in an intersection subtype. Moreover, intersection type schemes (so also ω) occur in strict types only as subtypes at the left hand side of an arrow type scheme. We could have omitted the type constant ω completely from the presentation of the system, because we can always assume that $n = 0$ in $\sigma_1 \cap \cdots \cap \sigma_n$, but some of the definitions and the results we obtain are more clear when ω is dealt with explicitly.

1.1 Essential type assignment

Definition 1.1.1 (cf. [1]) *i)* $\mathcal{T}_s$, the set of *strict types*, is inductively defined by:

a) All type-variables $\varphi_0, \varphi_1, \ldots \in \mathcal{T}_s$.
b) If $\tau, \sigma_1, \ldots, \sigma_n \in \mathcal{T}_s$ $(n \geq 0)$, then $\sigma_1 \cap \cdots \cap \sigma_n \rightarrow \tau \in \mathcal{T}_s$.

ii) $\mathcal{T}_S$, the set of *strict intersection types*, is defined by: If $\sigma_1, \ldots, \sigma_n \in \mathcal{T}_s$ $(n \geq 0)$, then $\sigma_1 \cap \cdots \cap \sigma_n \in \mathcal{T}_S$.

iii) On $\mathcal{T}_S$, the relation $\leq_S$ is defined by:

a) $\forall\, 1 \leq i \leq n$ $(n \geq 1)$ $[\, \sigma_1 \cap \cdots \cap \sigma_n \leq_S \sigma_i \,]$.
b) $\forall\, 1 \leq i \leq n$ $(n \geq 0)$ $[\, \sigma \leq_S \sigma_i \,] \Rightarrow \sigma \leq_S \sigma_1 \cap \cdots \cap \sigma_n$.
c) $\sigma \leq_S \tau \leq_S \rho \Rightarrow \sigma \leq_S \rho$.

iv) We define the relation $\leq_E$ on $\mathcal{T}_S$ like the relation $\leq_S$, that is only defined for strict intersection types, but we add an extra alternative.

d) $\rho \leq_E \sigma \;\&\; \tau \leq_E \mu \Rightarrow \sigma \rightarrow \tau \leq_E \rho \rightarrow \mu$.

v) On $\mathcal{T}_S$, the relation $\sim_E$ is defined by: $\sigma \sim_E \tau \Leftrightarrow \sigma \leq_E \tau \leq_E \sigma$.

vi) A *statement* is an expression of the form $M{:}\sigma$, where $M \in \Lambda$ and $\sigma \in \mathcal{T}_S$. M is the *subject* and σ the *predicate* of $M{:}\sigma$.

vii) A *basis* is a set of statements with only distinct variables as subjects.
If $\sigma_1 \cap \cdots \cap \sigma_n$ is a predicate in a basis, then $n \geq 1$.

$\mathcal{T}_S$ may be considered modulo $\sim_E$. Then $\leq_E$ becomes a partial order, and in this paper we consider types modulo $\sim_E$.

Unless stated otherwise, if $\sigma_1 \cap \cdots \cap \sigma_n$ is used to denote a type, then by convention all $\sigma_1, \ldots, \sigma_n$ are assumed to be strict. Notice that $\mathcal{T}_s$ is a proper subset of $\mathcal{T}_S$.

Definition 1.1.2 *i) Essential type assignment* and *essential derivations* are defined by the following natural deduction system (where all types displayed are strict, except σ in the derivation rules $(\rightarrow I)$ and $(\leq_E)$):

$$(\to\mathrm{I}):\quad \frac{\begin{array}{c}[x{:}\sigma]\\ \vdots\\ M{:}\tau\end{array}}{\lambda x.M{:}\sigma\to\tau}\,(a) \qquad (\leq_{\mathrm{E}}):\quad \frac{x{:}\sigma \quad \sigma\leq_{\mathrm{E}}\tau}{x{:}\tau}$$

$$(\to\mathrm{E}):\quad \frac{M{:}\sigma_1\cap\cdots\cap\sigma_n\to\tau \;\; N{:}\sigma_1 \;\ldots\; N{:}\sigma_n}{MN{:}\tau}\,(n\geq 0)$$

(a)) If $x{:}\sigma$ is the only statement about x on which $M{:}\tau$ depends.

If $M{:}\sigma$ is derivable from B using an essential derivation, we write $B \vdash_{\mathrm{e}} M{:}\sigma$.

ii) We define $\vdash_{\mathrm{E}}$ by: $B \vdash_{\mathrm{E}} M{:}\sigma$ if and only if: there are $\sigma_1, \ldots, \sigma_n$ $(n \geq 0)$ such that $\sigma = \sigma_1\cap\cdots\cap\sigma_n$ and for every $1 \leq i \leq n$ $B \vdash_{\mathrm{e}} M{:}\sigma_i$.

Although the derivation rule $(\leq_{\mathrm{E}})$ is not allowed on all terms, we can prove that if $B \vdash_{\mathrm{E}} M{:}\sigma$, and $\sigma \leq_{\mathrm{E}} \tau$, then $B \vdash_{\mathrm{E}} M{:}\tau$. It is then easy to prove that type assignment in this system is closed under η-reduction. It is also possible to prove that the essential type assignment system satisfies the main properties of the BCD-system:

Property 1.1.3 *i)* $B \vdash_{\mathrm{E}} M{:}\sigma \;\&\; M =_\beta N \;\Rightarrow\; B \vdash_{\mathrm{E}} N{:}\sigma$.

ii) $\exists\, B, \sigma\, [\, B \vdash_{\mathrm{E}} M{:}\sigma \;\&\; B, \sigma$ *ω-free* $]\;\Leftrightarrow\; M$ *has a normal form.*

iii) $\exists\, B, \sigma\, [\, B \vdash_{\mathrm{E}} M{:}\sigma \;\&\; \sigma \neq \omega\,] \;\Leftrightarrow\; M$ *has a head normal form.* ∎

1.2 Operations on pairs

In this subsection we present three different operations on pairs of <*basis*, *type*>, namely substitution, expansion, and lifting as defined in [3]. The operation of substitution deals with the replacement of type-variables by types and is a slight modification of the one normally used; this modification is needed to make sure that substitution is closed on strict types. The operation of expansion replaces types by the intersection of a number of copies of that type and coincides with the one given in [10, 21]. The operation of lifting deals with the introduction of extra (types to) statements in the basis of a derivation, or introduces extra types to term-variables that are bound.

Substitution is normally defined on types as the operation that replaces type-variables by types. For strict types this definition would not be correct. For example, the replacement of φ by ω would transform $\sigma\to\varphi$ (or $\sigma\cap\varphi$) into $\sigma\to\omega$ $(\sigma\cap\omega)$, which is not a strict type. Therefore, for strict types substitution is not defined as an operation that replaces type-variables by types, but as a mapping from types to types.

Definition 1.2.1 ([3]) *i)* The *substitution* $(\varphi \mapsto \alpha) : \mathcal{T}_{\mathrm{S}} \to \mathcal{T}_{\mathrm{S}}$, where φ is a type-variable and $\alpha \in \mathcal{T}_{\mathrm{s}} \cup \{\omega\}$, is defined by:

a) $(\varphi \mapsto \alpha)\,(\varphi) = \alpha$.

b) $(\varphi \mapsto \alpha)\,(\varphi') = \varphi'$, if $\varphi \neq \varphi'$.

c) $(\varphi \mapsto \alpha)\,(\sigma\to\tau) = \omega$, if $(\varphi \mapsto \alpha)\,(\tau) = \omega$.

d) $(\varphi \mapsto \alpha)\,(\sigma\to\tau) = (\varphi \mapsto \alpha)\,(\sigma) \to (\varphi \mapsto \alpha)\,(\tau)$, if $(\varphi \mapsto \alpha)\,(\tau) \neq \omega$.

e) $(\varphi \mapsto \alpha)\,(\sigma_1\cap\cdots\cap\sigma_n) = (\varphi \mapsto \alpha)\,(\sigma_1{}')\cap\cdots\cap(\varphi \mapsto \alpha)\,(\sigma_m{}')$, where $\{\sigma_1{}', \ldots, \sigma_m{}'\} = \{\, \sigma_i \in \{\sigma_1, \ldots, \sigma_n\} \mid (\varphi \mapsto \alpha)\,(\sigma_i) \neq \omega\,\}$.

ii) If S_1 and S_2 are substitutions, then so is $S_1{\circ}S_2$, where $S_1{\circ}S_2\,(\sigma) = S_1\,(S_2\,(\sigma))$.

iii) $S\,(B) = \{\, x{:}S\,(\alpha) \mid x{:}\alpha \in B \;\&\; S\,(\alpha) \neq \omega\,\}$.

iv) $S(<B, \sigma>) = <S(B), S(\sigma)>$.

Notice that in part *(i.e)*, if for $1 \leq i \leq n$ $(\varphi \mapsto \alpha)(\sigma_i) = \omega$, then $(\varphi \mapsto \alpha)(\sigma_1 \cap \cdots \cap \sigma_n) = \omega$.

The operation of expansion is an operation on types that deals with the replacement of (sub)types by an intersection of a number of copies of that type. In this process it can be that also other types need to be copied. An expansion indicates not only the type to be expanded, but also the number of copies that has to be generated.

Definition 1.2.2 ([3]) *i)* The *last variable* of a strict type is defined by:

a) The last variable of φ is φ.

b) The last variable of $\sigma_1 \cap \cdots \cap \sigma_n \rightarrow \tau$ $(n \geq 0)$ is the last variable of τ.

ii) A strict type σ is said to *end with* φ, if φ is the last variable of σ.

Definition 1.2.3 ([3]) For every $\mu \in \mathcal{T}_S$, $n \geq 2$, basis B and $\sigma \in \mathcal{T}_S$, the quadruple $<\mu, n, B, \sigma>$ determines an *expansion* $E_{<\mu,n,B,\sigma>} : \mathcal{T}_S \rightarrow \mathcal{T}_S$, that is constructed as follows.

i) The set of type-variables $\mathcal{V}_\mu(<B, \sigma>)$ is constructed by:

a) If φ occurs in μ, then $\varphi \in \mathcal{V}_\mu(<B, \sigma>)$.

b) Let τ be a strict (sub)type occurring in $<B, \sigma>$, with last variable φ_0. If $\varphi_0 \in \mathcal{V}_\mu(<B, \sigma>)$, then for all type-variables φ that occur in τ: $\varphi \in \mathcal{V}_\mu(<B, \sigma>)$.

ii) Suppose $\mathcal{V}_\mu(<B, \sigma>) = \{\varphi_1, \ldots, \varphi_m\}$. Choose $m \times n$ different type-variables $\varphi_1^1, \ldots, \varphi_1^n, \ldots, \varphi_m^1, \ldots, \varphi_m^n$, such that each φ_j^i does not occur in $<B, \sigma>$, for $1 \leq i \leq n$ and $1 \leq j \leq m$. Let S_i be the substitution that replaces every φ_j by φ_j^i.

iii) $E_{<\mu,n,B,\sigma>}(\alpha)$ is obtained by traversing α top-down and replacing, in α, a subtype β that ends with an element of $\mathcal{V}_\mu(<B, \sigma>)$ by $S_1(\beta) \cap \cdots \cap S_n(\beta)$.

iv) $E_{<\mu,n,B,\sigma>}(B') = \{ x{:}E_{<\mu,n,B,\sigma>}(\rho) \mid x{:}\rho \in B' \}$.

v) $E_{<\mu,n,B,\sigma>}(<B', \sigma'>) = <E_{<\mu,n,B,\sigma>}(B'), E_{<\mu,n,B,\sigma>}(\sigma')>$.

The last operation on pairs defined in this subsection is the operation of lifting.

Definition 1.2.4 ([3]) A *lifting* L is denoted by a pair $<<B_0, \tau_0>, <B_1, \tau_1>>$ such that $\tau_0 \leq_E \tau_1$ and $B_1 \leq_E B_0$, and is defined by:

i) $L(\sigma) = \tau_1$ if $\sigma = \tau_0$; $L(\sigma) = \sigma$ otherwise.

ii) $L(B) = B_1$ if $B = B_0$; $L(B) = B$ otherwise.

iii) $L(<B, \sigma>) = <L(B), L(\sigma)>$.

Definition 1.2.5 ([3]) A *chain* is an object $<O_1, \ldots, O_n>$, with each O_i an operation of substitution, expansion or lifting; $<O_1, \ldots, O_n>(<B, \sigma>) = O_n(\cdots(O_1(<B, \sigma>))\cdots)$.

It is possible to prove the principal type property for the essential type assignment system, in the same way as done in [21] for the BCD-system. The operations needed for this proof are substitution, expansion, and lifting, and it is possible to show that all pairs for a term can be generated by chains that exist of expansions, and substitutions (in that order) and that end with one lifting. Moreover, all three operations can be proven to be sound on all pairs.

2 Type assignment in Applicative Term Rewriting Systems

In this paper we study type assignment on applicative term rewriting systems, which is a slight extension of the term rewriting systems as defined in [15]. Applicative term rewriting systems are defined as term rewriting systems that (can) contain a special binary operator Ap, this in contrast to the *pure* applicative term rewriting systems, that contain *only* the binary operator Ap.

2.1 Applicative Term Rewriting Systems

The motivation for the use of applicative term rewriting systems instead of the general term rewriting systems can be illustrated by the following: There is a clear translation (embedding) of combinator systems into term rewriting systems, in which the implicit application of the world of combinators is made explicit. The kind of term rewriting system that is needed for such a translation contains only one function symbol, called Ap, and is therefore often called an applicative term rewriting system. A translation of for example Combinatory Logic (CL)

$$\begin{array}{ll} S\ x\ y\ z & = x\ z\ (y\ z) \\ K\ x\ y & = x \\ I\ x & = x \end{array}$$

into such a term rewriting system then looks like:

$$\begin{array}{ll} Ap\,(Ap\,(Ap\,(S,x),y),z) & \rightarrow Ap\,(Ap\,(x,z),Ap\,(y,z)) \\ Ap\,(Ap\,(K,x),y) & \rightarrow x \\ Ap\,(I,x) & \rightarrow x \end{array}$$

The definition of applicative systems we present in this paper is, however, more general: in the systems we consider, Ap is a *special* function symbol; in particular it is *one of the function symbols*, not the only one. To distinguish between the term rewriting systems that contain *only* the function symbol Ap and those that contain Ap next to other function symbols, we call the former the *pure* applicative term rewriting systems.

We prefer to see the symbols S, K and I as functions, with 3, 2 and 1 operands respectively. This means that we have to introduce extra rewrite rules to express the Curried versions of these symbols. Moreover, to get some computational power, some rewrite rule starting with Ap should be added. Such an extended CL system could look like:

$$\begin{array}{ll} S\,(x,y,z) & \rightarrow Ap\,(Ap\,(x,z),Ap\,(y,z)) \\ Ap\,(S_2\,(x,y),z) & \rightarrow S\,(x,y,z) \\ Ap\,(S_1\,(x),y) & \rightarrow S_2\,(x,y) \\ Ap\,(S_0,x) & \rightarrow S_1\,(x) \\ K\,(x,y) & \rightarrow x \\ Ap\,(K_1\,(x),y) & \rightarrow K\,(x,y) \\ Ap\,(K_0,x) & \rightarrow K_1\,(x) \\ I\,(x) & \rightarrow x \\ Ap\,(I_0,x) & \rightarrow I\,(x) \end{array}$$

We consider the applicative rewriting systems, because they are far more general than the subclass of systems in which there exists only the function symbol Ap. Moreover, they are a natural extension of those rewrite systems considerd in papers on type assignment on term rewriting systems that follow the 'algebraic' approach [12], and are also the kind of rewrite systems an effecient implementation of a functional language would be based upon [18]. Since the pure applicative term rewriting systems are a subclass of the applicative term rewriting systems, all results obtained in this paper are also valid for that subclass.

We take the view that in a rewrite rule a certain symbol is defined; it is this symbol to which the structure of the rewrite rule gives a type. We treat Ap as a predefined symbol; the symbol Ap is neglected when we are looking for the symbol that is defined in a rewrite rule.

The type assignment system we present in this paper is a partial system in the sense of [19]: we not only define how terms and rewrite rules can be typed, but assume that every function symbol already has a type, which structure is usually motivated by a rewrite rule. There are several reasons to do so.

First of all a term rewriting system can contain symbols that is not the defined symbol of a rewrite rule (such a symbol is called a constant). A constant can appear in a rewrite rule more or less as a symbol that 'has to be there', but for which it is impossible to determine any functional characterisation, apart from what is demanded by the immediate context. If we provide a type for every constant, then we can formulate some consistency requirement, by saying that the types used for a constant must be related to the provided type.

Moreover, even for every defined symbol there must be some way of determining what type can be used for an occurrence. Normally the rewrite rules that define such a symbol are investigated, and from analyzing the structure of those rules the 'most general type' for that symbol can be constructed. Instead of for defined symbols investigating all their defining rules every time the symbol is encountered, we can store the type of the symbol in a mapping from symbols to types, and use this mapping instead. Of course it makes no difference to assume the existence from the start of such a mapping from symbols (both defined and constant) to types, and to define type assignment using that mapping (in the following such a mapping is called an 'environment').

Definition 2.1.1 (cf. [15, 4]) An *Applicative Term Rewriting System* (ATRS) is a pair (Σ, **R**) of an *alphabet* or *signature* Σ and a set of *rewrite rules* **R**.

i) The alphabet Σ consists of:
  - *a)* A countable infinite set of variables $x_1, x_2, x_3, \ldots$ (or $x, y, z, x', y', \ldots$).
  - *b)* A non empty set $\mathcal{F}$ of *function symbols* $F, G, \ldots$, each equipped with an 'arity'.
  - *c)* A special binary operator, called *application* (Ap).

ii) The set of *terms* (or *expressions*) 'over' Σ is $T(\mathcal{F}, \mathcal{X})$ and is defined inductively:
  - *a)* $x, y, z, \ldots \in T(\mathcal{F}, \mathcal{X})$.
  - *b)* If $F \in \mathcal{F} \cup \{\mathsf{Ap}\}$ is an n-ary symbol, and $t_1, \ldots, t_n \in T(\mathcal{F}, \mathcal{X})$, then $\mathsf{F}(t_1, \ldots, t_n) \in T(\mathcal{F}, \mathcal{X})$.

Definition 2.1.2 (cf. [15, 4]) Let (Σ, **R**) be an ATRS.

i) A *replacement* is a map $\mathrm{R} : T(\mathcal{F}, \mathcal{X}) \rightarrow T(\mathcal{F}, \mathcal{X})$ satisfying $\mathrm{R}(\mathsf{F}(t_1, \ldots, t_n)) = F(\mathrm{R}(t_1), \ldots, \mathrm{R}(t_n))$ for every n-ary function symbol $F \in \mathcal{F} \cup \{\mathsf{Ap}\}$ (here $n \geq 0$). We also write T^{R} instead of $\mathrm{R}(T)$.

ii) a) A *rewrite rule* $\in$ **R** is a pair (Lhs, Rhs) of terms $\in T(\mathcal{F}, \mathcal{X})$. Often a rewrite rule will get a name, e.g. **r**, and we write **r** : $Lhs \rightarrow Rhs$. Three conditions will be imposed:
  - *1)* Lhs is not a variable.
  - *2)* The variables occurring in Rhs are contained in Lhs.
  - *3)* For every Ap in Lhs, the left hand argument is not a variable.

  b) A rewrite rule **r** : $Lhs \rightarrow Rhs$ determines a set of *rewrites* $\mathrm{Lhs}^{\mathrm{R}} \rightarrow \mathrm{Rhs}^{\mathrm{R}}$ for all replacements R. The left hand side $\mathrm{Lhs}^{\mathrm{R}}$ is called a *redex*; it may be replaced by its '*contractum*' $\mathrm{Rhs}^{\mathrm{R}}$

inside a context C[ ]; this gives rise to *rewrite steps*:
C[$\mathrm{Lhs}^{\mathrm{R}}$] $\to_{\mathrm{r}}$ C[$\mathrm{Rhs}^{\mathrm{R}}$].

iii) In a rewrite rule, the leftmost, outermost symbol in the left hand side that is not an Ap, is called the *defined symbol* of that rule. If the symbol F is the defined symbol of a rewrite rule **r**, then **r** *defines* F. F is a *defined symbol*, if there is a rewrite rule that defines F. $Q \in \mathcal{F}$ is called a *constant symbol* if Q is not a defined symbol.

iv) For every defined symbol F with arity $n \geq 1$, there are n additional rewrite rules that define the function symbols F_0 upto F_{n-1} as follows:

$$\begin{array}{ll} Ap\,(F_{n-1}\,(x_1, \ldots, x_{n-1}), x_n) & \to F\,(x_1, \ldots, x_n) \\ Ap\,(F_{n-2}\,(x_1, \ldots, x_{n-2}), x_{n-1}) & \to F_{n-1}\,(x_1, \ldots, x_{n-1}) \\ \quad\vdots & \\ Ap\,(F_0, x_1) & \to F_1\,(x_1) \end{array}$$

The added rules with $F_{n-1}, \ldots, F_1, F_0$, etc. give in fact the 'Curried'-versions of F.

Part *(ii.a.3)* of definition *2.1.2* is added in order to avoid rewrite rules with left hand sides like $Ap\,(x, y)$, because such a rule would not have a defined symbol.

We will, for the sake of simplicity, assume that rewrite rules are not mutually recursive; for those rules the definition of defined symbol, and also that of type assignment, would be more complicated, and this would unnecessarily obscure this paper.

In the following definition we give a special applicative term rewriting system.

Definition 2.1.3 *Applicative Combinatory Logic* (ACL) is the ATRS (Σ, **R**), where $\mathcal{F}$ = $\{S, S_2, S_1, S_0, K, K_1, K_0, I, I_0\}$, and **R** contains the rewrite rules

$$\begin{array}{ll} S\,(x, y, z) & \to Ap\,(Ap\,(x, z), Ap\,(y, z)) \\ K\,(x, y) & \to x \\ I\,(x) & \to x. \end{array}$$

For ACL we have for example the following rewriting sequence:

$S\,(K_0, S_0, I_0) \to Ap\,(Ap\,(K_0, I_0), Ap\,(S_0, I_0)) \to Ap\,(K_1\,(I_0), Ap\,(S_0, I_0)) \to$
$K\,(I_0, Ap\,(S_0, I_0)) \to I_0$.

Notice that a term like $K_1\,(I_0)$ itself cannot be rewritten. This corresponds to the fact that in CL the term $K\,I$ is not a redex. Because ACL is Curry-closed, it is in fact combinatory complete: every lambda term can be translated into a term in ACL; for details of such a translation, see [5, 13].

Example 2.1.4 If the left hand side of a rewrite rule is $\mathsf{F}\,(t_1, \ldots, t_n)$, then the t_i need not be simple variables, but can be terms as well, as for example in the rewrite rule $H\,(S_2\,(x, y)) \to S_2\,(I_0, y)$.

It is also possible that for a certain symbol F, there are more than one rewrite rule that define F. For example the rewrite rules $F\,(x) \to x$, $F\,(x) \to Ap\,(x, x)$ are legal.

2.2 Essential type assignment in ATRS's

Partial intersection type assignment on an ATRS (Σ, R) is defined as the labelling of nodes and edges in the tree-representation of terms and rewrite rules with types in $\mathcal{T}_{\mathrm{S}}$. In this labelling, we use that there is a mapping that provides a type in $\mathcal{T}_{\mathrm{S}}$ for every $F \in \mathcal{F} \cup \{\mathsf{Ap}\}$. Such a mapping is called an environment.

Definition 2.2.1 Let $(\Sigma, \mathbf{R})$ be an ATRS.

i) A mapping $\mathcal{E} : \mathcal{F} \cup \{\mathsf{Ap}\} \to \mathcal{T}_s$ is called an *environment* if $\mathcal{E}(Ap) = (1\to2)\to1\to2$, and for every $F \in \mathcal{F}$ with arity n, $\mathcal{E}(F) = \mathcal{E}(F_{n-1}) = \cdots = \mathcal{E}(F_0)$.

ii) For $F \in \mathcal{F}$ with arity $n \geq 0$, $\sigma \in \mathcal{T}_s$, and $\mathcal{E}$ an environment, the environment $\mathcal{E}[\mathrm{F} := \sigma]$ is defined by:

a) $\mathcal{E}[\mathrm{F} := \sigma](G) = \sigma$, if $G \in \{F, F_{n-1}, \ldots, F_0\}$.

b) $\mathcal{E}[\mathrm{F} := \sigma](G) = \mathcal{E}(G)$, otherwise.

Since $\mathcal{E}$ maps all $F \in \mathcal{F}$ to types in $\mathcal{T}_s$, no function symbol is mapped to ω.

Type assignment on applicative term rewriting systems is defined in two stages. In the next definition we define type assignment on terms, in definition *2.2.6* we define type assignment on term rewrite rules.

Definition 2.2.2 Let $(\Sigma, \mathbf{R})$ be an ATRS, and $\mathcal{E}$ an environment.

i) We say that $T \in T(\mathcal{F}, \mathcal{X})$ *is typeable by* $\sigma \in \mathcal{T}_S$ *with respect to* $\mathcal{E}$, if there exists an assignment of types to edges and nodes that satisfies the following constraints:

a) The root edge of T is typed with σ; if $\sigma = \omega$, then the root edge is the only thing in the term-tree that is typed.

b) The type assigned to a function node containing $F \in \mathcal{F} \cup \{\mathsf{Ap}\}$ (where F has arity $n \geq 0$) is $\tau_1\cap\cdots\cap\tau_m$, if and only if for every $1 \leq i \leq m$ there are $\sigma_1^i, \ldots, \sigma_n^i \in \mathcal{T}_S$, and $\sigma_i \in \mathcal{T}_s$, such that $\tau_i = \sigma_1^i\to\cdots\to\sigma_n^i\to\sigma_i$, the type assigned to the j-th ($1 \leq j \leq n$) out-going edge is $\sigma_j^1\cap\cdots\cap\sigma_j^m$, and the type assigned to the incoming edge is $\sigma_1\cap\cdots\cap\sigma_m$.

$$\begin{tikzcd}[column sep=small]
& & {} \arrow[d, "\sigma_1\cap\cdots\cap\sigma_m"] & & \\
& & F{:}(\sigma_1^1\to\cdots\to\sigma_n^1\to\sigma_1)\cap\cdots\cap(\sigma_1^m\to\cdots\to\sigma_n^m\to\sigma_m) \arrow[dll, "\sigma_1^1\cap\cdots\cap\sigma_1^m"'] \arrow[dl, "\sigma_2^1\cap\cdots\cap\sigma_2^m"'] \arrow[d, phantom, "\cdots"] \arrow[dr, "\sigma_{n-1}^1\cap\cdots\cap\sigma_{n-1}^m"] \arrow[drr, "\sigma_n^1\cap\cdots\cap\sigma_n^m"] & & \\
{} & {} & {} & {} & {}
\end{tikzcd}$$

c) If the type assigned to a function node containing $F \in \mathcal{F} \cup \{\mathsf{Ap}\}$ is τ, then there is a chain C, such that $C(\mathcal{E}(F)) = \tau$.

ii) Let $T \in T(\mathcal{F}, \mathcal{X})$ be typeable by σ with respect to $\mathcal{E}$. If B is a basis such that for every statement $x{:}\tau$ occurring in the typed term-tree there is a $x{:}\tau' \in B$ such that $\tau' \leq_E \tau$, we write $B \vdash_{\mathcal{E}} T{:}\sigma$.

Notice that if $B \vdash_{\mathcal{E}} T{:}\sigma$, then B can contain more statements than needed to obtain $T{:}\sigma$. Notice also that parts *(i.a)* and *(ii)* are not in conflict so for every B and T: $B \vdash_{\mathcal{E}} T{:}\omega$.

A typical example for part *(i.b)* of definition *2.2.2* is the symbol Ap; for every occurrence of Ap in a term-tree, there are σ and τ such that the following is part of the term-tree.

$$\begin{tikzcd}[column sep=small]
& {} \arrow[d, "\tau"] & \\
& Ap{:}\ (\sigma\to\tau)\to\sigma\to\tau \arrow[dl, "\sigma\to\tau"'] \arrow[dr, "\sigma"] & \\
{} & & {}
\end{tikzcd}$$

Notice that the type the environment provides for Ap is crucial; it is the type suggested by the $(\to\mathrm{E})$ derivation rule, and gives structure to the type assignment.

Example 2.2.3 The term $S(K_0, S_0, I_0)$ can be typed with the type $7\to7$, under the assumption that: $\mathcal{E}(S) = (1\to2\to3)\to(4\to2)\to1\cap4\to3$, $\mathcal{E}(K) = 5\to\omega\to5$, $\mathcal{E}(I) = 6\to6$.

$$\begin{tikzcd} & {} \arrow[d, "7\to7"] & \\ & S{:}((7\to7)\to\omega\to7\to7)\to\omega\to(7\to7)\to7\to7 \arrow[dl] \arrow[d] \arrow[dr] & \\ K_0{:}(7\to7)\to\omega\to7\to7 & S_0 & I_0{:}7\to7 \end{tikzcd}$$

Notice that to obtain the type $((7\to7)\to\omega\to7\to7)\to\omega\to(7\to7)\to7\to7$ for S, we have used the chain $<(1 := 7\to7), (2 := \omega), (3 := 7\to7), (4 := \omega)>$, and that the node containing S_0 is not typed since the incoming edge is typed with ω. If we define $D(x) \to Ap(x, x)$, then we can even check that for example $D(S(K_0, S_0, I_0))$ and $D(I_0)$ are both typeable by $8\to8$.

The following definition introduces some terminology and notations for bases.

Definition 2.2.4 *i)* The relation $\leq_{\rm E}$ is extended to bases by: $B \leq_{\rm E} B'$ if and only if for every $x{:}\sigma' \in B'$ there is an $x{:}\sigma \in B$ such that $\sigma \leq_{\rm E} \sigma'$.

ii) If $B_1, \ldots, B_n$ are bases, then $\Pi\{B_1, \ldots, B_n\}$ is the basis defined as follows: $x{:}_1\cap\cdots\cap\sigma_m \in \Pi\{B_1, \ldots, B_n\}$ if and only if $\{x{:}\sigma_1, \ldots, x{:}\sigma_m\}$ is the set of all statements whose subject is x that occur in $B_1 \cup \ldots \cup B_n$.

Notice that if $n = 0$, then $\Pi\{B_1, \ldots, B_n\} = \emptyset$.

In the next definition we introduce the notion of used bases. The idea is to collect all types assigned to term-variables that are actually used for the typed term-tree, but the collected types need not occur in the original bases themselves.

Definition 2.2.5 *i)* The *used bases of* $B \vdash_{\mathcal{E}} T{:}\sigma$ are inductively defined by:

a) $\sigma \in \mathcal{T}_{\rm s}$.

1) $T \equiv x$. Take $\{x{:}\sigma\}$.

2) $T \equiv \mathsf{F}(t_1, \ldots, t_n)$. There are $\sigma_1, \ldots, \sigma_n$ such that for every $1 \leq i \leq n$ $B \vdash_{\mathcal{E}} t_i{:}\sigma_i$. Let for $1 \leq i \leq n$, B_i be a used basis of $B \vdash_{\mathcal{E}} t_i{:}\sigma_i$.
Take $\Pi\{B_1, \ldots, B_n\}$.

b) If $\sigma = \sigma_1\cap\cdots\cap\sigma_n$ $(n \geq 0)$, then for every $1 \leq i \leq n$ $B \vdash_{\mathcal{E}} T{:}\sigma_i$. Let for every $1 \leq i \leq n$, B_i be a used basis of $B \vdash_{\mathcal{E}} T{:}\sigma_i$. Take $\Pi\{B_1, \ldots, B_n\}$.

ii) A basis B *is used for* $T{:}\sigma$ *with respect to* $\mathcal{E}$ if and only if there is a basis B' such that $B' \vdash_{\mathcal{E}} T{:}\sigma$ and B is a used basis of $B' \vdash_{\mathcal{E}} T{:}\sigma$.

Notice that in part (i.b), if $n = 0$, then $\sigma = \omega$, and $\Pi\{B_1, \ldots, B_n\} = \emptyset$.

We will say 'B is used for $T{:}\sigma$' instead of 'B is used for $T{:}\sigma$ with respect to $\mathcal{E}$'. A used basis for a statement $T{:}\sigma$ is not unique, but the results of this paper do not depend on the actual structure of such a basis, only on its existence. Thanks to the notion of used basis, we can give a clear definition of a typeable rewrite rule and a typeable rewrite system. The condition 'B is used for $Lhs{:}\sigma$' in definition *2.2.6(i.a)* is crucial. Just saying:

We say that $Lhs \to Rhs \in \mathbf{R}$ with defined symbol F *is typeable with respect to* $\mathcal{E}$, if there are basis B, and type $\sigma \in \mathcal{T}_{\rm s}$ such that: $B \vdash_{\mathcal{E}} Lhs{:}\sigma$ and $B \vdash_{\mathcal{E}} Rhs{:}\sigma$,

would give a notion of type assignment that is not closed under rewriting (i.e. does not satisfy the subject reduction property), and is not a natural extension of the essential intersection type assignment system for the λ-calculus. For an example of the first, take the rewrite system $I(x) \to x$, $K(x, y) \to x$, $F(I_0) \to I_0$, $G(x) \to F(x)$.

Take the environment $\mathcal{E}\,(G) = (5{\rightarrow}\omega{\rightarrow}5){\rightarrow}6{\rightarrow}6$, $\mathcal{E}\,(F) = (3{\rightarrow}3){\rightarrow}4{\rightarrow}4$, $\mathcal{E}\,(K) = 2{\rightarrow}\omega{\rightarrow}2$, $\mathcal{E}\,(I) = 1{\rightarrow}1$. Take $B = \{x{:}(7{\rightarrow}7){\cap}(5{\rightarrow}\omega{\rightarrow}5)\}$, then $B \vdash_{\mathcal{E}} G\,(x){:}6{\rightarrow}6$, and $B \vdash_{\mathcal{E}} F\,(x){:}6{\rightarrow}6$. Notice that $\vdash_{\mathcal{E}} G\,(K_0){:}7{\rightarrow}7$, but not $\vdash_{\mathcal{E}} F\,(K_0){:}7{\rightarrow}7$.

Therefore, a minimal requirement for subject reduction is to demand that all types assigned to term-variables in the typed term-tree for the right hand side of a rewrite rule already occurred in the typed term-tree for the left hand side. This is accomplished by restricting the possible bases to those that contain nothing but the types actually used for the left hand side.

Definition 2.2.6 Let (Σ, **R**) be an ATRS, and $\mathcal{E}$ an environment.

i) We say that $Lhs \rightarrow Rhs \in$ **R** with defined symbol F *is typeable with respect to* $\mathcal{E}$, if there are basis B, type $\sigma \in \mathcal{T}_S$, and an assignment of types to nodes and edges such that:

a) B is used for $Lhs{:}\sigma$ and $B \vdash_{\mathcal{E}} Rhs{:}\sigma$.

b) In $B \vdash_{\mathcal{E}} Lhs{:}\sigma$ and $B \vdash_{\mathcal{E}} Rhs{:}\sigma$, all nodes containing F are typed with $\mathcal{E}\,(F)$.

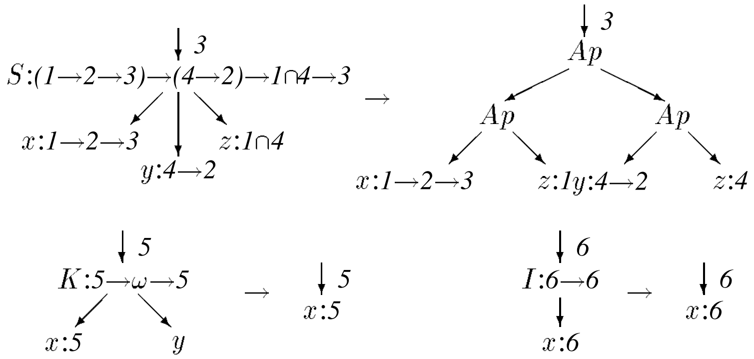
ii) We say that (Σ, **R**) *is typeable with respect to* $\mathcal{E}$, if every $\mathbf{r} \in$ R is typeable with respect to $\mathcal{E}$.

Condition *(i.b)* of definition *2.2.6* is in fact added to make sure that the type provided by the environment for a function symbol F is not in conflict with the rewrite rules that define F. By restricting the type that can be assigned to the defined symbol to the type provided by the environment, we are sure that the rewrite rule is typed using that type, and not using some other type. Since by part *(i.b)* of definition *2.2.6* all occurrences of the defined symbol in a rewrite rule are typed with the same type, type assignment of rewrite rules is actually defined using Milner's way of dealing with recursion.

It is easy to check that if F is a function symbol with arity n, and all rewrite rules that define F are typeable, then there are $\gamma_1, \ldots, \gamma_n, \gamma$ such that $\mathcal{E}\,(F) = \gamma_1{\rightarrow}\cdots{\rightarrow}\gamma_n{\rightarrow}\gamma$.

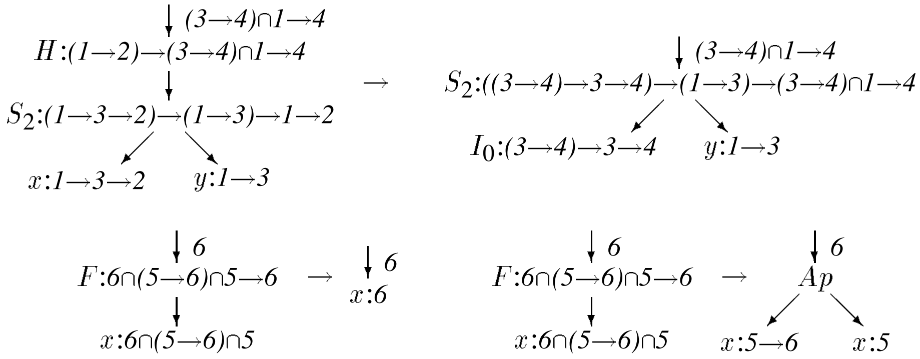
The use of an environment and part *(i.c)* of definition *2.2.2* introduces a notion of polymorphism into our type assignment system. The environment returns the 'principal type' for a function symbol; this symbol can be used with types that are 'instances' of its principal type.

Example 2.2.7 Typed versions of some of the rewrite rules given in definition *2.1.3*, under the assumption that: $\mathcal{E}\,(S) = (1{\rightarrow}2{\rightarrow}3){\rightarrow}(4{\rightarrow}2){\rightarrow}1{\cap}4{\rightarrow}3$, $\mathcal{E}\,(K) = 5{\rightarrow}\omega{\rightarrow}5$, $\mathcal{E}\,(I) = 6{\rightarrow}6$.



Notice that the node containing y in the rule for K is not typed; its incoming edge is typed with ω.

Example 2.2.8 Typed versions of the the rewrite rules given in example *2.1.4*, using: $\mathcal{E}\,(H) = (1{\rightarrow}2){\rightarrow}(3{\rightarrow}4){\cap}1{\rightarrow}4$, $\mathcal{E}\,(S) = (7{\rightarrow}8{\rightarrow}9){\rightarrow}(10{\rightarrow}8){\rightarrow}7{\cap}10{\rightarrow}9$, $\mathcal{E}\,(I) = 11{\rightarrow}11$, $\mathcal{E}\,(F) = 6{\cap}(5{\rightarrow}6){\cap}5{\rightarrow}6$.



2.3 Soundness of operations

It is possible to show that the three operations on pairs (substitution, expansion, and lifting) are sound on typed term-trees. For the operations of substitution and expansion it is also possible to show that part *(i.c)* of definition *2.2.2* is sound in the following sense: if there is an operation O (either a substitution or an expansion) such that $O(\mathcal{E}(F)) = \sigma$, then for every type $\tau \in \mathcal{T}_S$ such that $\sigma \leq_S \tau$, the rewrite rules that define F are typeable with respect to the changed environment $\mathcal{E}[F := \tau]$. It is not possible to prove such a property for the operation of lifting.

Theorem 2.3.1 Soundness of substitution. *Let S be a substitution.*

- *i) If* $B \vdash_{\mathcal{E}} T{:}\sigma$, *then* $S(B) \vdash_{\mathcal{E}} T{:}S(\sigma)$.
- *ii) If* B *is used for* $T{:}\sigma$, *then* $S(B)$ *is used for* $T{:}S(\sigma)$.
- *iii) Let* $\mathbf{r}$: $Lhs \rightarrow Rhs$ *be a rewrite rule typeable with respect to the environment* $\mathcal{E}$, *and let* F *be the defined symbol of* $\mathbf{r}$. *Then* $\mathbf{r}$ *is typeable with respect to* $\mathcal{E}[F := S(\mathcal{E}(F))]$. ■

Theorem 2.3.2 Soundness of expansion. *Let E be an expansion such that* $E(<B, \sigma>) = <B', \sigma'>$.

- *i) If* $B \vdash_{\mathcal{E}} T{:}\sigma$, *then* $B' \vdash_{\mathcal{E}} T{:}\sigma'$.
- *ii) If* B *is used for* $T{:}\sigma$, *then* B' *is used for* $T{:}\sigma'$.
- *iii) Let* $\mathbf{r}$: $Lhs \rightarrow Rhs$ *be a rewrite rule typeable with respect to the environment* $\mathcal{E}$, *and let* F *be the defined symbol of* $\mathbf{r}$. *If* $E(\mathcal{E}(F)) = \tau \in \mathcal{T}_S$, *then for every* $\mu \in \mathcal{T}_S$ *such that* $\tau \leq_S \mu$, $\mathbf{r}$ *is typeable with respect to* $\mathcal{E}[F := \mu]$. ■

Notice that in part (iii) the relation $\leq_S$ *is used, not* $\leq_E$.

Theorem 2.3.3 Soundness of lifting. *If* $B \vdash_{\mathcal{E}} T{:}\sigma$ *and L is a lifting , then* $L(B) \vdash_{\mathcal{E}} T{:}L(\sigma)$. ■

Obviously not every lifting performed on a pair $<B, \sigma>$ such that B is used for $T{:}\sigma$ produces a pair with this same property. Since type assignment of rewrite rules is defined using the notion of used bases, it is clear that lifting cannot be a sound operation on rewrite rules. This can also be illustrated by the rewrite system $I(x) \rightarrow x$, $F(I_0) \rightarrow I_0$, that is typeable with respect to the environment $\mathcal{E}_1(I) = 1{\rightarrow}1$, $\mathcal{E}_1(F) = (2{\rightarrow}2){\rightarrow}3{\rightarrow}3$. Notice that $(2{\rightarrow}2){\rightarrow}3{\rightarrow}3 \leq_E (2{\rightarrow}2)\cap 4{\rightarrow}3{\rightarrow}3$, so $<<\emptyset, (2{\rightarrow}2){\rightarrow}3{\rightarrow}3>, <\emptyset, (2{\rightarrow}2)\cap 4{\rightarrow}3{\rightarrow}3>>$ is a lifting. It is not possible to show that the rewrite rule that defines F is typeable with respect to $\mathcal{E}[F := (2{\rightarrow}2)\cap 4{\rightarrow}3{\rightarrow}3]$, since all types in $(2{\rightarrow}2)\cap 4$ should be types for I.

Combining the above results for the different operations, we have:

Theorem 2.3.4 *i) If $B \vdash_{\mathcal{E}} T{:}\sigma$ then for every chain C such that C(<B, σ>) = <B′, σ′>: $B' \vdash_{\mathcal{E}} T{:}\sigma'$.*

ii) If B is used for $T{:}\sigma$, and C is a chain that contains no lifting, then: if C(<B, σ>) = <B′, σ′>, then B' is used for $T{:}\sigma'$.

iii) Let **r**: *$Lhs \rightarrow Rhs$ be a rewrite rule typeable with respect to the environment $\mathcal{E}$, and let F be the defined symbol of* **r**. *If C is a chain that contains no lifting, then: if C($\mathcal{E}(F)$) = $\tau \in \mathcal{T}_S$, then for every $\mu \in \mathcal{T}_S$ such that $\tau \leq_S \mu$,* **r** *is typeable with respect to $\mathcal{E}[F := \mu]$.* ∎

3 The loss of the subject reduction property

By definition *2.1.2 (i)*, if a term T is rewritten to the term T' using the rewrite rule $Lhs \rightarrow Rhs$, there is a subterm t_0 of T, and a replacement R, such that $\mathrm{Lhs}^{\mathrm{R}} = t_0$, and T' is obtained by replacing t_0 by $\mathrm{Rhs}^{\mathrm{R}}$. The subject reduction property for this notion of reduction is: If $B \vdash_{\mathcal{E}} T{:}\sigma$, and T can be rewritten to T', then $B \vdash_{\mathcal{E}} T'{:}\sigma$.

This is of course an important property of reduction systems. To guarantee the subject reduction property, we should accept only those rewrite rules $Lhs \rightarrow Rhs$, that satisfy:

For all replacements R, bases B and types σ: if $B \vdash_{\mathcal{E}} \mathrm{Lhs}^{\mathrm{R}}{:}\sigma$, then $B \vdash_{\mathcal{E}} \mathrm{Rhs}^{\mathrm{R}}{:}\sigma$.

because then we are sure that all possible rewrites are safe.

Definitions *2.2.1*, *2.2.2* and *2.2.6* define what a type assignment should be, just using the strategy as used in languages like for example Miranda. Unfortunately, it is not sufficient to guarantee the subject reduction property. Take for example the definition of H as in example *2.1.4*, and the following environment $\mathcal{E}_0(H) = ((1\rightarrow2)\rightarrow3)\rightarrow(1\rightarrow2)\rightarrow2$, $\mathcal{E}_0(S) = (1\rightarrow2\rightarrow3)\rightarrow(1\rightarrow2)\rightarrow1\rightarrow3$, $\mathcal{E}_0(I) = 1\rightarrow1$. The rule that defines H is typeable with respect to $\mathcal{E}_0$:

$$\begin{array}{ccc}
\begin{array}{c}
\downarrow (1\rightarrow2)\rightarrow2 \\
H{:}((1\rightarrow2)\rightarrow3)\rightarrow(1\rightarrow2)\rightarrow2 \\
\downarrow \\
S_2{:}((1\rightarrow2)\rightarrow1\rightarrow3)\rightarrow((1\rightarrow2)\rightarrow1)\rightarrow(1\rightarrow2)\rightarrow3 \\
\swarrow \qquad \searrow \\
x{:}(1\rightarrow2)\rightarrow1\rightarrow3 \qquad y{:}(1\rightarrow2)\rightarrow1
\end{array}
& \rightarrow &
\begin{array}{c}
\downarrow (1\rightarrow2)\rightarrow2 \\
S_2{:}((1\rightarrow2)\rightarrow1\rightarrow2)\rightarrow((1\rightarrow2)\rightarrow1)\rightarrow(1\rightarrow2)\rightarrow2 \\
\swarrow \qquad \searrow \\
I_0{:}(1\rightarrow2)\rightarrow1\rightarrow2 \qquad y{:}(1\rightarrow2)\rightarrow1
\end{array}
\end{array}$$

If we take the term $H(S_2(K_0, I_0))$ then it is easy to see that the rewrite is allowed, and that this term will be rewritten to $S_2(I_0, I_0)$. Although the first term is typeable with respect to $\mathcal{E}_0$:

$$\begin{array}{c}
\downarrow (4\rightarrow5)\rightarrow5 \\
H{:}((4\rightarrow5)\rightarrow4\rightarrow5)\rightarrow(4\rightarrow5)\rightarrow5 \\
\downarrow \\
S_2{:}((4\rightarrow5)\rightarrow(4\rightarrow5)\rightarrow4\rightarrow5)\rightarrow((4\rightarrow5)\rightarrow4\rightarrow5)\rightarrow(4\rightarrow5)\rightarrow4\rightarrow5 \\
\swarrow \qquad \searrow \\
K_0{:}(4\rightarrow5)\rightarrow(4\rightarrow5)\rightarrow4\rightarrow5 \qquad I_0{:}(4\rightarrow5)\rightarrow4\rightarrow5
\end{array}$$

the term $S_2(I_0, I_0)$ is not typeable with respect to $\mathcal{E}_0$ with the type $(4\rightarrow5)\rightarrow5$.

We should emphasize that the loss of the subject reduction property is not caused by the fact that we use intersection types. The environment $\mathcal{E}_0$ maps function symbols to Curry-types, so even for a notion of type assignment based on Curry-types types are not preserved under rewriting.

In [4] and [2] two restrictions (variants) of the notion of type assignment of this paper are discussed for which a decidable and sufficient condition is formulated that rewrite rules should satisfy in order to reach the subject reduction property.

The construction of those conditions is made using a notion of principal pairs; the condition a rewrite rule should satisfy is that the principal pair for the left hand side term is also a pair for the right hand side term. For the notion of type assignment defined in this section we are not able to formulate this condition in a constructive way, since it is not clear how we should define *the* principal pair for a term. This problem is overcome in [4] and [2] by defining a most general unification algorithm for types and defining principal pairs using that algorithm. At this moment there is no general unification algorithm for types in $\mathcal{T}_{\mathrm{S}}$ that works well on all types, so we cannot take this approach.

For the notion of type assignment as defined in this paper the only result we can obtain is to show that *if* a left hand side of a rewrite rule has *a* principal pair and using that pair the rewrite rule can be typed, then rewriting using this rule is safe with respect to subject reduction.

Definition 3.1 *i)* Let $T \in T(\mathcal{F}, \mathcal{X})$. A pair $<P, \pi>$ is called *a principal pair for* T *with respect to* $\mathcal{E}$, if $P \vdash_{\mathcal{E}} T{:}\pi$ and for every B, σ such that $B \vdash_{\mathcal{E}} T{:}\sigma$ there is a chain *C* such that $C\,(<P, \pi>) = <B, \sigma>$.

ii) The definition of a *safe type assignment with respect to* $\mathcal{E}$ is the same as the one for a type assignment as defined in definition *2.2.6*, by replacing condition *(i.a)* by:

$<B, \sigma>$ is a principal pair for Lhs with respect to $\mathcal{E}$, and $B \vdash_{\mathcal{E}} Rhs{:}\sigma$.

Then rewrite rule $Lhs \rightarrow Rhs$ is called a *safe rewrite rule*.

Notice that we do not show that every typeable term *has* a principal pair with respect to $\mathcal{E}$; at the moment we cannot give a construction of such a pair for every term. But even with this non-constructive approach we can show that the condition is sufficient.

Theorem 3.2 The condition is sufficient. *Let* $\mathbf{r} : Lhs \rightarrow Rhs$ *be a safe rewrite rule. Then for every replacement* R*, basis* B *and a type* μ*: if* $B \vdash_{\mathcal{E}} Lhs^{\mathrm{R}}{:}\mu$*, then* $B \vdash_{\mathcal{E}} Rhs^{\mathrm{R}}{:}\mu$. ■

Future work

We intend to formulate a decidable condition rewrite rules should satisfy in order to obtain the subject reduction property. In the near future characterizations of typeable rewrite systems will be looked for, like for example normalizability of non-recursive typeable systems.

References

[1] S. van Bakel. Complete restrictions of the Intersection Type Discipline. *Theoretical Computer Science*, 102:135–163, 1992.

[2] S. van Bakel. Partial Intersection Type Assignment of Rank 2 in Applicative Term Rewriting Systems. Technical Report 92-03, Department of Computer Science, University of Nijmegen, 1992.

[3] S. van Bakel. Principal type schemes for the Strict Type Assignment System. *Logic and Computation*, 1993. To appear.

[4] S. van Bakel, S. Smetsers, and S. Brock. Partial Type Assignment in Left Linear Applicative Term Rewriting Systems. In J.-C. Raoult, editor, *Proceedings of CAAP '92. 17th Colloquim on Trees in Algebra and Programming, Rennes, France*, volume 581 of *Lecture Notes in Computer Science*, pages 300–321. Springer-Verlag, 1992.

[5] H. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, Amsterdam, revised edition, 1984.

[6] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The Journal of Symbolic Logic*, 48(4):931–940, 1983.

[7] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In *Proceedings of PARLE, Parallel Architectures and Languages Europe, Eindhoven, The Netherlands*, volume 259-II of *Lecture Notes in Computer Science*, pages 141–158. Springer-Verlag, 1987.

[8] T. Brus, M.C.J.D. van Eekelen, M.O. van Leer, and M.J. Plasmeijer. Clean - A Language for Functional Graph Rewriting. In *Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture, Portland, Oregon, USA*, volume 274 of *Lecture Notes in Computer Science*, pages 364–368. Springer-Verlag, 1987.

[9] M. Coppo and M. Dezani-Ciancaglini. An Extension of the Basic Functionality Theory for the λ-Calculus. *Notre Dame, Journal of Formal Logic*, 21(4):685–693, 1980.

[10] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and λ-calculus semantics. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry, Essays in combinatory logic, lambda-calculus and formalism*, pages 535–560. Academic press, New York, 1980.

[11] H.B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[12] N. Dershowitz and J.P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 245–320. North-Holland, 1990.

[13] M. Dezani-Ciancaglini and J.R. Hindley. Intersection types for combinatory logic. *Theoretical Computer Science*, 100:303–324, 1992.

[14] K. Futatsugi, J. Goguen, J.P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proceedings 12th ACM Symposium on Principles of Programming Languages*, pages 52–66, 1985.

[15] J.W. Klop. Term Rewriting Systems. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 1, pages 1–116. Clarendon Press, 1992.

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

[17] E.G.J.M.H. Nöcker, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J. Plasmeijer. Concurrent Clean. In *Proceedings of PARLE '91, Parallel Architectures and Languages Europe, Eindhoven, The Netherlands*, volume 506-II of *Lecture Notes in Computer Science*, pages 202–219. Springer-Verlag, 1991.

[18] S. Peyton Jones and J. Salkild. The spineless tagless G-machine. In *Functional Programming Languages and Computer Architecture*, pages 184–201. ACM press, 1989.

[19] F. Pfenning. Partial Polymorphic Type Inference and Higher-Order Unification. In *Proceedings of the 1988 conference on LISP and Functional Programming Languages*, volume 201 of *Lecture Notes in Computer Science*, pages 153–163. Springer-Verlag, 1988.

[20] B.C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pitssburgh, 1991. CMU-CS-91-205.

[21] S. Ronchi della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theoretical Computer Science*, 28:151–169, 1984.

[22] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1985.