Partial Type Assignment in Left Linear Applicative Term Rewriting Systems*

Theory, Applications and Implementation.

Steffen van Bakel[1†] Sjaak Smetsers[1] Simon Brock[2]

[1]) Department of Informatics, Faculty of Mathematics and Informatics, University of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands.

[2]) School of Information Systems, University of East Anglia, Norwich NR4 7TJ, United Kingdom.

Abstract

This paper introduces a notion of partial type assignment on left linear applicative term rewriting systems that is based on the extension defined by Mycroft of Curry's type assignment system. The left linear applicative TRS we consider 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. As such there is no distinction between function symbols (such as append and plus) and constructor symbols (such as cons and succ). Terms and rewrite rules will be written as trees, and type assignment will consist of assigning types to function symbols, nodes and edges between nodes. The only constraints on this system are imposed by the relation between the type assigned to a node and those assigned to its incoming and out-going edges. We will show that every typeable term has a principal type, and formulate a needed and sufficient condition typeable rewrite rules should satisfy in order to gain preservance of types under rewriting. As an example we will show that the optimisation function performed after bracket abstraction is typeable. Finally we will present a type check algorithm that checks if rewrite rules are correctly typed, and finds the principal pair for typeable terms.

Introduction

In the recent years several paradigms have been investigated for the implementation of functional programming languages. Not only the lambda calculus [Barendregt '84], but also term rewriting systems [Klop '92] and graph rewriting systems [Barendregt *et al.* '87] are topics of research. Lambda calculus (or rather combinator systems) forms the underlying model for the functional programming language Miranda [Turner '85], term rewriting systems are used in the underlying model for the language OBJ [Futatsugi *et al.* '85], and graph rewriting systems is the model for the language Clean [Brus *et al.* '87, Nöcker *et al.* '91].

There exists a well understood and well defined notion of type assignment on lambda terms,

*Supported by the Esprit Basic Research Action 3074 "Semagraph".

†Partially supported by the Netherlands Organisation for the advancement of pure research (N.W.O.).

known as the Curry type assignment system [Curry & Feys '58]. This type assignment system is the basis for many type checkers and inferers used in functional programming languages. For example the type assignment system for the language ML [Milner '78], as defined by R. Milner forms in fact an extension of Curry's system. The type inference algorithm for the functional programming language Miranda works in roughly the same way as the one for ML. A real difference between these languages lies in the fact that Miranda also contains a type check algorithm, which is based on the type assignment system defined by A. Mycroft, an extension of Milner's type assignment system [Mycroft '84, Kfoury *et al.* '88].

To provide a formal type system for all languages that use pattern matching this paper presents a formal notion of type assignment on left linear applicative term rewriting systems. As shown in this paper type assignment in term rewriting systems in general does not satisfy the subject reduction property: i.e. types are not preserved under rewriting. The main result of this paper, the formulation of a needed and sufficient condition on rewrite rules to obtain subject reduction, could be used to prove that all rewrite rules that can be defined in a language like Miranda are safe in that respect. Also if at first sight one could think that the generalization of type assignments from functional programming languages to term rewriting systems would be straightforward, this is not true, as is shown from the conditions given to guarantee preservance of types under rewriting.

It is our aim to extend the notion presented in this paper to a notion of type assignment for term graph rewriting systems, and, possibly, general graph rewriting systems.

The type assignment system we present in this paper is a partial system in the sense of [Pfenning '88], because we not only define how terms and rewrite rules can be typed, but also provide a type for each function symbol. There are several reasons to do so.

For symbols for which there is a rewrite rule (called defined symbols), and for symbols for which such a rule does not exist (called constants), there must be some way of determining what type can be used for an occurrence. Instead of for defined symbols investigating their rule 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 to types, and to define type assignment using that mapping. This mapping is also convenient to make sure that types assigned to different occurrences of constants do not conflict.

In fact, the approach we take here is very much the same as the one taken by Hindley in [Hindley '69], where he defines the principal type scheme of an object in Combinatory Logic. Even his notion of type assignment could be regarded as a partial one. Moreover, since combinator systems can easily be translated into left linear applicative term rewriting systems, the results of this paper, when restricting the allowed rewrite rules to those that correspond to combinators, are the same as in [Hindley '69].

In section *1* we define the left linear applicative term rewriting systems, as the subclass of term rewriting systems that contain a predefined symbol Ap, and in which all rewrite rules are left linear. The definition of applicative term rewriting systems is not the one normally used, but more general: in the systems we consider, Ap is not the only function symbol.

The notion of type assignment on left linear applicative term rewriting systems as defined in subsection *3.2* is in fact defined on the tree representation of terms and rewrite rules, defined in section *2*, by assigning in a consistent way types (as defined in subsection *3.1*) to nodes and edges. The only constraints on this system are imposed by the relation between the type assigned to a node and those assigned to its incoming and out-going edges. It is based on the

Mycroft type assignment system as used in Miranda.

In subsection *3.3* we will show that for every basis B, term M, type σ and substitution S: if $B \vdash M{:}\sigma$ (i.e. M is typeable by σ starting from the set B of typed term variables), then also $S(B) \vdash M{:}S(\sigma)$. In that subsection we will also show that for every typeable term M, there is a principal pair $<P, \pi>$ for M, i.e. for every pair $<B, \sigma>$ such that $B \vdash M{:}\sigma$ there is a substitution S such that $S(<P, \pi>) = <B, \sigma>$.

Another result presented in that subsection is that if $<B, \sigma>$ is the principal pair for M, and M' is obtained from M by replacing (in a consistent way) term variables by terms, and M' is typeable by τ, then there is a substitution S such that: $S(\sigma) = \tau$, and for every $x{:}\rho$ occurring in B, the replacement of x is typeable by $S(\rho)$.

In subsection *4.1* we show that if type assignment is done in the most obvious, straightforward way, there are rewrite rules that are typeable, that match a term M typeable with σ, but for which the result of the application of the rewrite rule on M is not typeable with σ. In subsection *4.2* we formulate a condition that typeable rewrite rules should satisfy in order to guarantee preservance of types under rewriting. In that subsection we prove this condition to be needed and sufficient.

In section *5* we show that the usually defined optimisation algorithm after bracket abstraction (a translation of lambda terms to combinator expressions) is typeable in our system. In the appendix we present a type check algorithm, that checks if rewrite rules are correctly typed, and finds the principal pair for typeable terms.

1 Left linear applicative term rewriting systems

In this paper we study left linear applicative term rewriting systems (LLATRS), which are a subclass of term rewriting systems, as defined in [Klop '92]. LLATRS's are defined as the class of term rewriting systems that (can) contain a special binary operator Ap, and in which all rewrite rules are left linear. To distinguish them from the term rewriting systems that have *only* the function symbol Ap, we call the latter the *pure* 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 example: If we would want to translate Combinatory Logic (CL)

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

into a term rewriting system, then it could look like (making the implicit application explicit):

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

However, we prefer to see the symbols S, K and I as functions, with 3, 2 and 1 operands respectively. If we try to capture that view in our translation, we would need to define also the Curried versions of those symbol:

$$\begin{aligned} S\,(x,y,z) &\rightarrow Ap\,(Ap\,(x,z),Ap\,(y,z)) \\ Ap\,(S_2\,(x,y),z) &\rightarrow S\,(x,y,z) \end{aligned}$$

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

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. Since the pure left linear applicative term rewriting systems are a subclass of the left linear 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, and it is clear that the rules added to obtain the Curried versions of symbols in the translation of CL into a rewriting system are not intended as definitions for Ap, which is more or less a 'predefined function', but as definitions for the Curried versions.

However, in general term rewriting systems are not sensitive for the names used for functions symbols. So the rewriting system given above is in fact the same as the one obtained by replacing all Ap's by F, and then all rewrite rules starting with F could be seen as rules that define F. In order to avoid this problem, we regard those rewriting systems that have a 'predefined' binary function, called Ap, which then cannot be renamed. The symbol Ap is neglected when we are looking for the symbol that is defined in a rewrite rule.

In the lambda calculus there is a clear difference between free and bound variables of a term. In term rewriting systems a term variable x that occurs in the left hand side of a rewrite rule can be seen as the binding occurrence of x, binding the occurrences of x in the right hand side. However, in general x can occur more than once in the left hand side, making the notion of *the* binding occurrence obscure. In this paper we consider left linear rewriting systems, that contain only rewrite rules for which the left hand side is linear (term variables occur only once), because for those rules the binding occurrence of a term variable is unique.

The following definitions are based on definitions given in [Klop '92]. Definition *1.1* defines LLATRS's, in the same way as the definition given by Klop for term rewriting systems, extended with part *(i.c)* to express the existence of the predefined symbol Ap. Definition *1.2* defines a notion of rewriting on LLATRS's, in the same way as the definition of rewriting given by Klop for term rewriting systems, extended with part *(ii.a.2)* to express the left linearity of rewrite rules, part *(ii.a.4)* to express that the possible use of the symbol Ap in the left hand side is restricted, and part *(iii)* to define the notion of defined symbol of a rewrite rule. In fact, parts *(ii.a.4)* and *(iii)* are related.

We introduce in some parts a different notation, because some of the symbols or definitions normally used are also used in papers on type assignment, but with a different meaning. For example we use the word 'replacement' for the operation that replaces term variables by terms, instead of the word 'substitution', which will be used for operations that replace type variables by types.

Substitution and replacement are also operations defined in [Curry & Feys '58]. Both operations are there defined as operations on terms, where substitution is defined as the operation that replaces term variables by terms, and replacement is defined as the operation that

replaces subterms by terms. Note that our definition therefore differs from the one given in [Curry & Feys '58].

To denote a replacement, we use capital characters like 'R', instead of greek symbols like 'σ', which are used to denote types. We use the symbol '$\rightarrow$' for the rewriting symbol, instead of '$\rightarrow$' which is used as a type constructor. We use the notion 'constant symbol' for a symbol that cannot be rewritten, instead of for a function symbol with arity 0.

Definition 1.1 A *Left Linear Applicative Term Rewriting System* (LLATRS) is a pair (Σ, $\mathbf{R}$) of an *alphabet* or *signature* Σ and a set of *rewrite rules* $\mathbf{R}$.

i) The alphabet Σ consists of:

a) A countable infinite set of variables x_1, x_2, x_3, ... (or x, y, z, x', y', ...).

b) A non empty set $\mathcal{F}$ of *function symbols* or *operator symbols* F, G, ..., each equipped with an 'arity' (a natural number), i.e. the number of 'arguments' it is supposed to have. We have 0-ary, unary, binary, ternary etc function symbols.

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, ... $\in T(\mathcal{F},\mathcal{X})$.

b) If $F \in \mathcal{F} \cup \{\mathsf{Ap}\}$ is an n-ary symbol, and t_1, ..., $t_n \in T(\mathcal{F},\mathcal{X})$ ($n \geq 0$), then

$$F(t_1, \ldots, t_n) \in T(\mathcal{F},\mathcal{X}).$$

The t_i ($i = 1, \ldots, n$) are the arguments of the last term.

iii) Terms in which no variable occurs twice or more, are called *linear*.

Definition 1.2 Let (Σ, $\mathbf{R}$) be a LLATRS.

i) A *replacement* R is a map from $T(\mathcal{F},\mathcal{X})$ to $T(\mathcal{F},\mathcal{X})$ satisfying

$$\mathrm{R}(F(t_1, \ldots, t_n)) = F(\mathrm{R}(t_1), \ldots, \mathrm{R}(t_n))$$

for every n-ary function symbol F (here $n \geq 0$). So, R is determined by its restriction to the set of variables. We also write T^{R} instead of $\mathrm{R}(T)$.

ii) a) A *rewrite rule* $\in \mathbf{R}$ is a pair (Lhs, Rhs) of terms $\in T(\mathcal{F},\mathcal{X})$. Often a rewrite rule will get a name, e.g. $\mathbf{r}$, and we write $\mathbf{r} : Lhs \rightarrow Rhs$. Four conditions will be imposed:

1) Lhs is not a variable.

2) Lhs is linear.

3) The variables occurring in Rhs are contained in Lhs.

4) For every *Ap* in Lhs, the left hand argument is not a variable.

b) A rewrite rule $\mathbf{r} : Lhs \rightarrow Rhs$ 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*:

$$\mathrm{C}[\,\mathrm{Lhs}^{\mathrm{R}}\,] \rightarrow_{\mathbf{r}} \mathrm{C}[\,\mathrm{Rhs}^{\mathrm{R}}\,].$$

c) We call $\rightarrow_{\mathbf{r}}$ the *one-step rewrite relation* generated by $\mathbf{r}$. Concatenating rewrite steps we have (possibly infinite) *rewrite sequences* $t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots$ or *rewrites* for short. If $t_0 \rightarrow \cdots \rightarrow t_n$ we also write $t_0 \rightarrow^* t_n$, and t_n is a *rewrite* of t_0.

iii) a) 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.

b) If the symbol F is the defined symbol of $\mathbf{r}$, then $\mathbf{r}$ *defines* F.

c) F is *a defined symbol*, if there is a rewrite rule that defines F.

d) $A \in \mathcal{F}$ is called a *constant symbol* if A is not a defined symbol.

Part *(ii.a.4)* of definition *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.

Proposition 1.3 Let F be the defined symbol of the rewrite rule $\mathbf{r} : Lhs \to Rhs$. Then there are $n \geq j \geq 0$, and $A_1, \ldots, A_n$ such that:

$$Lhs = Ap\,(Ap\,(\ldots Ap\,(F(A_1, \ldots, A_j), A_{j+1}), \ldots), A_n).$$

and $A_1, \ldots, A_n$ are called the patterns *of* $\mathbf{r}$.

Proof: Easy. ■

We will consider rewriting systems that are *Curry-closed*, i.e. for every rewrite rule that defines the symbol F with arity $n \geq 0$, we will assume that there are n additional rewrite rules that define the function symbols F_0 upto F_{n-1} as follows:

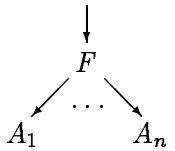
$$\begin{array}{lcl}
Ap(F_{n-1}(x_1, \ldots, x_{n-1}), x_n) & \to & F(x_1, \ldots, x_n) \\
Ap(F_{n-2}(x_1, \ldots, x_{n-2}), x_{n-1}) & \to & F_{n-1}(x_1, \ldots, x_{n-1}) \\
\quad \vdots & & \\
Ap(F_1(x_1), x_2) & \to & F_2(x_1, x_2) \\
Ap(F_0, x_1) & \to & F_1(x_1)
\end{array}$$

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

We could have defined a closure operation on LLATRS's, by adding rules and extending the alphabet Σ, but it is easier to assume that every LLATRS is Curry-closed. When presenting a rewrite system however, we will only show the rules that are essential: we do not show the rules that define the Curried-versions.

2 Tree representation of terms and rewrite rules

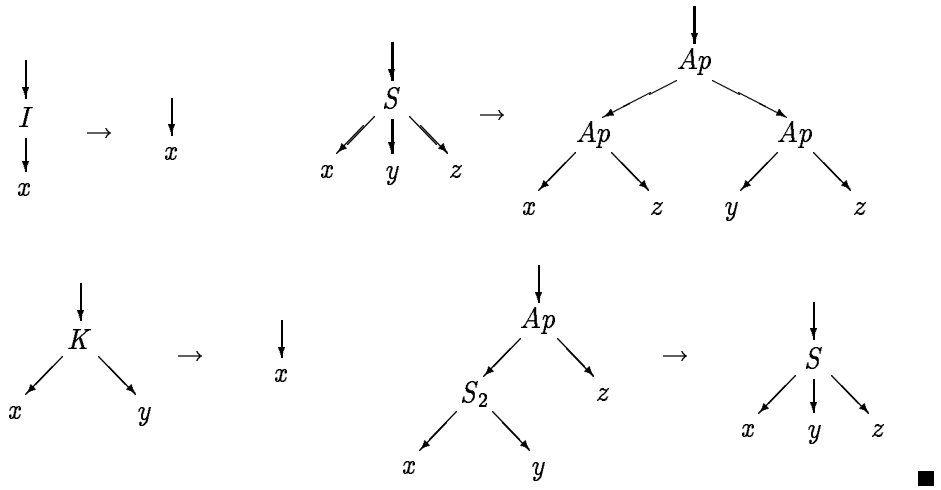
Definition 2.1 *i)* The tree representation of terms and rewrite rules is obtained in a straightforward way, by representing a term $F(A_1, \ldots, A_n)$ by:



ii) The *spine* of a tree is defined as usual, i.e.: the root node of the tree is on the spine, and if a node is on the spine, then its left most descendant is on the spine.

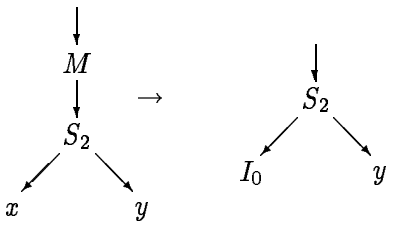
iii) In the tree representation of a rewrite rule, the first node on the spine of the left hand side, starting from the root node, that does not contain an Ap, is called the *defining node* of that rule. (Notice that if F is the defined symbol of a rewrite rule, then it occurs in the defining node of the tree representation of that rule.)

Example 2.2 We give some of the rewrite rules of CL in tree representation:



■

Example 2.3 Rewrite rules can of course be more complicated than illustrated above by the rules for CL. In general, if the left hand side of a rewrite rule is $F(t_1, \ldots, t_n)$, then the t_i need not be simple variables but can be terms as well, as for example in the rewrite rule $M\,(S_2\,(x, y)) \rightarrow\ S_2\,(I_0, y)$. In tree representation, this rule looks like:



■

3 Type assignment in LLATRS's

In this section we present a notion of partial type assignment on LLATRS's, based on the Mycroft type assignment system [Mycroft '84, Kfoury *et al.* '88]. Assigning types to a LLATRS will consist of labelling the nodes and edges in the tree representation of terms and rewrite rules with type information. Types are assigned to nodes to capture the notion of 'type of a function', 'type of a constant' or 'type of a variable', and are assigned to edges to capture the notion of 'type of a subterm' (or tree). The edge pointing to the the root of a term is called the *root edge*.

The difference between Milner's and Mycroft's type assignment system lies in the way they handle recursion. Would we have used Milner's system, then we would have demanded that all occurrences of recursively defined objects are, within their definition, typed with the same type. Mycroft's approach is a more general one. Using his system the only requirement for recursive definitions would be to demand that the separate occurrences of the object within the definition are typed with types that are substitution instances of the type of the object.

There is one major difference between the notion of type assignment we introduce in this paper and Curry's type assignment system. In that system a basis is usually defined as a mapping from term variables to types, or, equivalently, as a set of statements with distinct term variables as subjects. The bases allowed in the system we present however can contain several different statements for the same term variable. This in fact corresponds to the definition of ML-type

assignment as given in [Damas '85], and is used there for dealing with the let-construct. It could also be compared with using intersection types [Barendregt *et al.* '83] on free variables of terms. Unlike in lambda calculus, in term rewriting systems this causes no difficulties, since there is no notion of 'abstraction' in this world. Moreover, condition *(i.b)* of definition *3.2.3* deals with this apparent anomaly. So in the system we present, it is possible to assign a type to the term $Ap(x, x)$.

3.1 Types

The type system we define in this subsection is based on the Curry type system, extended with type constants.

Definition 3.1.1 *i)* $\mathcal{T}_C$, the set of Curry types is inductively defined by:

a) All type variables $\varphi_0, \varphi_1, \ldots \in \mathcal{T}_C$.
b) All type constants $c_0, c_1, \ldots \in \mathcal{T}_C$.
c) If $\sigma, \tau \in \mathcal{T}_C$, then $\sigma \rightarrow \tau \in \mathcal{T}_C$.

ii) A *statement* is an expression of the form $M{:}\sigma$, where $M \in T(\mathcal{F}, \mathcal{X})$ and $\sigma \in \mathcal{T}_C$. M is the *subject* and σ the *predicate* of $M{:}\sigma$.

iii) A *basis* B is a set of statements with term variables, not necessarily distinct, as subjects. If a basis is denoted as $\{x_1{:}\rho_1, \ldots, x_n{:}\rho_n\}$, the x_i are distinct.

Remark 3.1.2 In the notation of types, often outermost and rightmost brackets are omitted. We will use the symbol φ to denote a type variable and all other greek characters to denote arbitrary types. Also, instead of φ indexed by a number, we will write the number.

Definition 3.1.3 *i)* A *substitution* $S : \mathcal{T}_C \rightarrow \mathcal{T}_C$ is as usual inductively defined by:

a) The substitution $(\varphi \mapsto \alpha)$, where φ is a type variable and $\rho \in \mathcal{T}_C$, is defined by:

1) $(\varphi \mapsto \alpha)(\varphi) = \rho$.
2) $(\varphi \mapsto \alpha)(\varphi_0) = \varphi_0$, if $\varphi \neq \varphi_0$.
3) $(\varphi \mapsto \alpha)(c_i) = c_i$.
4) $(\varphi \mapsto \alpha)(\sigma \rightarrow \tau) = (\varphi \mapsto \alpha)(\sigma) \rightarrow (\varphi \mapsto \alpha)(\tau)$.

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

ii) If for σ, τ there is a substitution S such that $S(\sigma) = \tau$, then τ is called a *(substitution) instance* of σ.

iii) If σ is an instance of τ, and τ is an instance of σ, then σ is called a *trivial variant* of τ. We identify types that are trivial variants of each other.

iv) $S(B) = \{x{:}S(\rho) \mid x{:}\rho \in B\}$.

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

3.2 Type assignment

Type assignment on a LLATRS (Σ, R) is defined as the labelling of nodes and edges in the tree representation of terms and rewrite rules with types. If a node or edge is labelled with a type σ, we say that it is *typed* with σ, and σ is *assigned* to it.

In this labelling, we use that there is a mapping that provides a type in $\mathcal{T}_C$ for every

$F \in \mathcal{F} \cup \{\mathsf{Ap}\}$. Such a mapping is called an environment.

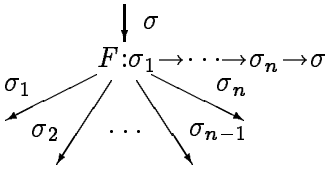
Definition 3.2.1 Let $(\Sigma, \mathbf{R})$ be a LLATRS. A mapping $\mathcal{E} : \mathcal{F} \cup \{\mathsf{Ap}\} \rightarrow \mathcal{T}_{\mathsf{C}}$ is called an *environment* if $\mathcal{E}(Ap) = (1 \rightarrow 2) \rightarrow 1 \rightarrow 2$, and for every $F \in \mathcal{F}$ with arity n, $\mathcal{E}(F) = \mathcal{E}(F_{n-1}) = \cdots = \mathcal{E}(F_0)$.

Definition 3.2.2 Let $(\Sigma, \mathbf{R})$ be a LLATRS.

i) We say that $M \in T(\mathcal{F}, \mathcal{X})$ *is typeable by* $\sigma \in \mathcal{T}$ *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 M is typed with σ.

b) If a node contains a symbol $F \in \mathcal{F} \cup \{\mathsf{Ap}\}$ that has arity n ($n \geq 0$), then there are σ_1, ..., σ_n and σ, such that this node is typed with $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma$, the n out-going edges are from left to right typed with σ_1 up to σ_n, and the in-going edge is typed with σ.



c) If a node containing a symbol $F \in \mathcal{F} \cup \{\mathsf{Ap}\}$ is typed with σ, then there is a substitution S, such that $S(\mathcal{E}(F)) = \sigma$.

ii) Let $M \in T(\mathcal{F}, \mathcal{X})$ be typeable by σ with respect to $\mathcal{E}$. If B is a basis containing all statements with variables as subjects that appear in the typed tree for $M{:}\sigma$, we write $B \vdash_{\mathcal{E}} M{:}\sigma$.

Notice that if $B \vdash_{\mathcal{E}} M{:}\sigma$, then B can contain more statements than needed to obtain $M{:}\sigma$.

Definition 3.2.3 Let $(\Sigma, \mathbf{R})$ be a LLATRS.

i) We say that $\mathbf{r}$: $Lhs \rightarrow Rhs \in \mathbf{R}$ with defined symbol F *is naively typeable with respect to* $\mathcal{E}$, if the following constraints hold:

a) There are $\sigma \in \mathcal{T}_{\mathsf{C}}$ and basis B such that $B \vdash_{\mathcal{E}} Lhs{:}\sigma$ and $B \vdash_{\mathcal{E}} Rhs{:}\sigma$.

b) All nodes within $\mathbf{r}$ containing the same term variable x, are typed with the same type.

c) The defining node of $\mathbf{r}$, containing F, is typed with $\mathcal{E}(F)$.

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

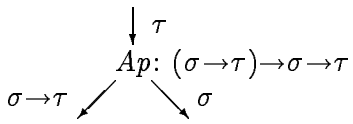
Condition *(i.c)* is in fact added to make sure that the type provided by the environment for a function symbol F is not in conflict with the rewrite rules that define F. By restricting the type that can be assigned to the defining node to the type provided by the environment, we are sure that the rewrite rule is typed using that type, and not using a substitution instance.

In the rest of this paper, we will assume the environment to be fixed, so we omit the subscript on $\vdash_{\mathcal{E}}$. Also, instead of saying '(naively) typeable with respect to $\mathcal{E}$', we just say 'typeable'.

It is easy to check that if F is a function symbol with artity n, and all rewrite rules that define F are typeable, then there are γ_1, ..., γ_n, γ such that $\mathcal{E}(F) = \gamma_1 \rightarrow \cdots \rightarrow \gamma_n \rightarrow \gamma$.

The use of an environment corresponds to the use of 'axiom-schemes', and part *(i.c)* of definition *3.2.2* to the use of 'axioms' as in [Hindley '69].

A typical example for part *(i.b)* of definition *3.2.2* is the function symbol Ap, which has the type $(1\to 2)\to 1\to 2$. So for every occurrence of Ap in a tree, there are σ and τ such that the following is part of the tree.



3.3 The principal pair for a term

In this subsection we define the principal pair for a typeable term M with respect to $\mathcal{E}$, consisting of basis B and type σ, by defining the notion $pp(M)$ using Robinsons unification algorithm $unify_{\mathrm{R}}$ [Robinson '65]. (Notice that, from a formal point of view, we would have to define $pp_{\mathcal{E}}(M)$, but that again we are omitting the subscript $\mathcal{E}$.) In the following we show that for every typeable term, this is a legal pair and is indeed the most general one.

We recall the following well known property of $unify_{\mathrm{R}}$.

Property 3.3.1 ([Robinson '65]) *If two types have an instance in common, they have a highest common instance which is returned by* $unify_{\mathrm{R}}$*, so for all* σ, τ*: if* $S_1 = unify_{\mathrm{R}}(\sigma, \tau)$ *and* S_2 *is a substitution such that* $S_2(\sigma) = S_2(\tau)$*, then there is a substitution* S_3 *such that* $S_2(\sigma) = S_3 \circ S_1(\sigma) = S_3 \circ S_1(\tau) = S_2(\tau)$. ■

Definition 3.3.2 We define, for every term M the notion $pp(M) = <P, \pi>$ inductively by:

i) For all x, φ: $pp(x) = <\{x{:}\varphi\}, \varphi>$.

ii) If for every $1 \le i \le n$: $pp(A_i) = <P_i, \pi_i>$, $\mathcal{E}(F) = \sigma_1\to\cdots\to\sigma_n\to\sigma$, (we choose if necessary trivial variants such that the $<P_i, \pi_i>$ are pairwise disjoint and these pairs share no type variables with $\sigma_1\to\cdots\to\sigma_n\to\sigma$), and

$$S = unify_{\mathrm{R}}(\sigma_1\to\cdots\to\sigma_n\to\sigma,\ \pi_1\to\cdots\to\pi_n\to\varphi),$$

where φ does not occur in any of the pairs $<P_i, \pi_i>$, nor in $\sigma_1\to\cdots\to\sigma_n\to\sigma$, then:

$$pp(F(A_1, \ldots, A_n)) = <S(P_1 \cup \cdots \cup P_n), S(\sigma)>.$$

The following theorem shows that substitution is a sound operation on trees. As illustrated in section *3.5*, we cannot show such a result for rewrite rules.

Theorem 3.3.3 **Soundness of substitution.** *If* $B \vdash M{:}\sigma$*, then for every substitution* S*:* $S(B) \vdash M{:}S(\sigma)$.

Proof: By easy induction on the structure of M. ■

By induction on the definition of $pp(M)$, using theorem *3.3.3* it is easy to verify that $pp(M) = <P, \pi>$ implies $P \vdash M{:}\pi$.

In the following theorem we show that the operation of substitution is complete.

Theorem 3.3.4 **Completeness of substitution.** *If* $B \vdash M{:}\sigma$*, then there are* P, π, *and a substitution* S *such that:* $pp(M) = <P, \pi>$*, and* $S(P) \subseteq B$, $S(\pi) = \sigma$.

Proof: By induction on the structure of M.

i) $M \equiv x$. Then $\{x{:}\sigma\} \subseteq B$. Then there is a φ such that $pp(x) = <\{x{:}\varphi\}, \varphi>$. Take $S = (\varphi \mapsto \sigma)$.

ii) $M \equiv F(A_1, \ldots, A_n)$. Then there are $\sigma_1, \ldots, \sigma_n$ such that

F is typed with $\sigma_1 \to \cdots \to \sigma_n \to \sigma$, and for every $1 \leq i \leq n$: $B \vdash A_i{:}\sigma_i$.

Also there are $\gamma_1, \ldots, \gamma_n, \gamma$, and a substitution S_0 such that

$\mathcal{E}(F) = \gamma_1 \to \cdots \to \gamma_n \to \gamma$, and $\sigma_1 \to \cdots \to \sigma_n \to \sigma = S_0(\gamma_1 \to \cdots \to \gamma_n \to \gamma)$.

By induction for every $1 \leq i \leq n$ there are $<P_i, \pi_i>$, (pairwise disjoint) and a substitution S_i, such that

$S_i(P_i) \subseteq B$, $S_i(\pi_i) = \sigma_i$, and $pp(A_i) = <P_i, \pi_i>$.

Assume, without loss of generality, that none of the type variables of the types $\gamma_1, \ldots, \gamma_n$, γ occur in any pair $<P_i, \pi_i>$. Take $S' = S_n \circ \cdots \circ S_0$, then:

$S'(\gamma) = \sigma$, $S'(P_1 \cup \cdots \cup P_n) \subseteq B$, and for every $1 \leq i \leq n$: $S'(\gamma_i) = S'(\pi_i) = \sigma_i$.

Let φ be a type variable not occurring in any other type. Since $\sigma_1 \to \cdots \to \sigma_n \to \sigma$ is a common instance of both $\gamma_1 \to \cdots \to \gamma_n \to \gamma$ and $\pi_1 \to \cdots \to \pi_n \to \varphi$, by property *3.3.1* there are substitutions S_g and S such that:

$S_g = unify_{\mathrm{R}}(\gamma_1 \to \cdots \to \gamma_n \to \gamma, \pi_1 \to \cdots \to \pi_n \to \varphi)$ and
$\sigma_1 \to \cdots \to \sigma_n \to \sigma = S \circ S_g(\gamma_1 \to \cdots \to \gamma_n \to \gamma) = S \circ S_g(\pi_1 \to \cdots \to \pi_n \to \varphi)$.

We can assume without loss of generality, that $S'(\beta) = S \circ S_g(\beta)$ for all β. By *3.3.2(ii)*:

$pp(F(A_1, \ldots, A_n)) = <S_g(P_1 \cup \cdots \cup P_n), S_g(\gamma)>$.

and $S'(P_1 \cup \cdots \cup P_n) = S \circ S_g(P_1 \cup \cdots \cup P_n)$, and $S'(\gamma) = S \circ S_g(\gamma)$. ■

In the following lemma we show that if F is the defined symbol of a rewrite rule, then the type $\mathcal{E}(F)$ dictates not only the type for the left and right hand side of that rule, but also the principal type for the left hand side.

Lemma 3.3.5 If F is the defined symbol of the typeable rewrite rule $\mathbf{r} : Lhs \to Rhs$, then there are P, B, σ_1, ..., σ_n, and σ such that

$\mathcal{E}(F) = \sigma_1 \to \cdots \to \sigma_n \to \sigma$, $pp(Lhs) = <P, \sigma>$, $B \vdash Lhs{:}\sigma$ and $B \vdash Rhs{:}\sigma$.

Proof: Easy, using *3.3.4*, *1.3* and the fact that if τ is a substitution instance of σ, and σ a substitution instance of τ, then $\sigma = \tau$. ■

The following lemma plays an important part in the proof that the condition, as defined in definition *4.2.1*, imposed on typeable rewrite rules is needed and sufficient, as given in theorem *4.2.3* and formulates the relation between replacements performed on a term and possible type assignments for that term.

Lemma 3.3.6 i) If $pp(M) = <P, \pi>$, and for the replacement R there are B and σ such that $B \vdash M^{\mathrm{R}}{:}\sigma$, then there is a substitution S, such that $S(\pi) = \sigma$, and for every statement $x{:}\rho \in P$: $B \vdash x^{\mathrm{R}}{:}S(\rho)$.

ii) If $B \vdash M{:}\sigma$, and R is a replacement and B' a basis such that for every statement $x{:}\rho \in B$: $B' \vdash x^{\mathrm{R}}{:}\rho$, then $B' \vdash M^{\mathrm{R}}{:}\sigma$.

Proof: By induction on the structure of M.

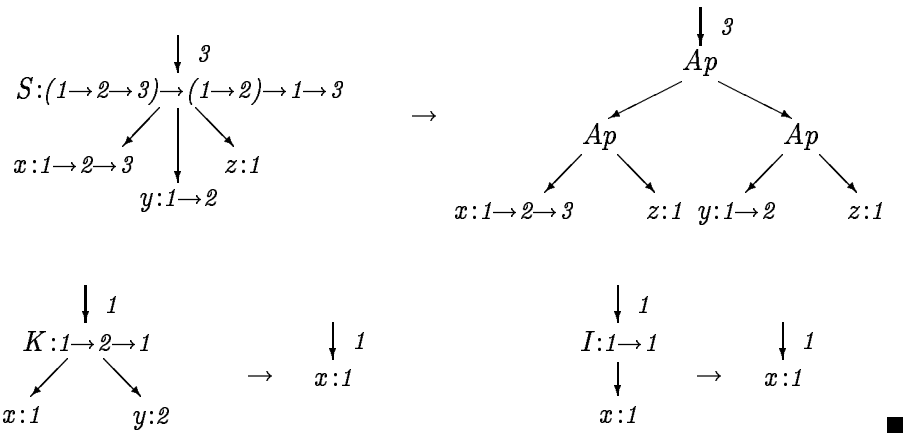
i) The proof is very much the same as the one for theorem *3.3.4*.

ii) Easy. ■

3.4 Examples

Example 3.4.1 Typed variants of some of the rewrite rules given in example *2.2*. We have only inserted those types that are not immediately clear. Notice that we have assumed that

$\mathcal{E}(S) = 1\to 2\to 3)\to(1\to 2)\to 1\to 3$,
$\mathcal{E}(K) = 1\to 2\to 1$, and
$\mathcal{E}(I) = 1\to 1$.

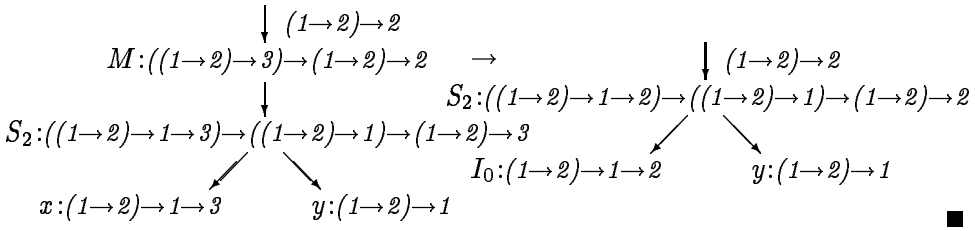


■

Example 3.4.2 Using

$\mathcal{E}(M) = ((1\to 2)\to 3)\to(1\to 2)\to 2$,
$\mathcal{E}(S) = (1\to 2\to 3)\to(1\to 2)\to 1\to 3$, and
$\mathcal{E}(I) = 1\to 1$

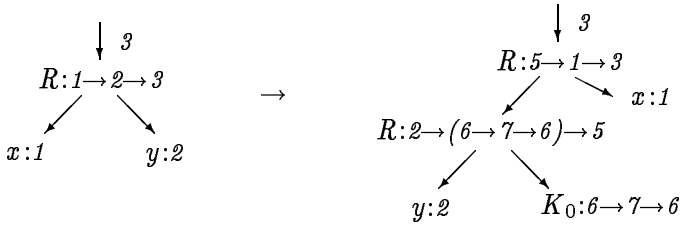
the rule of example *2.3* can be typed as follows:



■

3.5 About recursion

The rule $R(x,y) \to R(R(y,K_0),x)$ can be typed using $\mathcal{E}(R) = 1\to 2\to 3$:



Notice that because of definition *3.2.2(i.c)* the types assigned to the nodes containing R in the right hand side need only be instances of $1\to 2\to 3$.

Not every substitution instance of the type $1\to 2\to 3$ can be proven to be a correct type for R. For example when using $\mathcal{E}(R) = 1\to 1\to 2$, this rule cannot be typed. If first and second argument of R should have the same type, then the only types possible for R are the

substitution instances of $(1\to 2\to 1)\to(1\to 2\to 1)\to 1\to 2\to 1$.

However, by definition *3.2.2(i.c)* it is possible that R occurs in a term with type $1\to 1\to 2$. This means of course that although we allow all substitutions on the environment types of defined symbols, we cannot prove that all the types that are obtained in this way, are correct types for the rewrite rule.

This means in fact that the Mycroft approach is not very well suited for type assignment in term rewriting systems. If instead of Mycroft's approach we would use Milner's, which would require that all occurrences of the defined symbol F of a rule within that rule are typed with $\mathcal{E}(F)$, it is easy to verify that soundness of substitution can even be shown for rewrite rules.

4 Safe type assignment

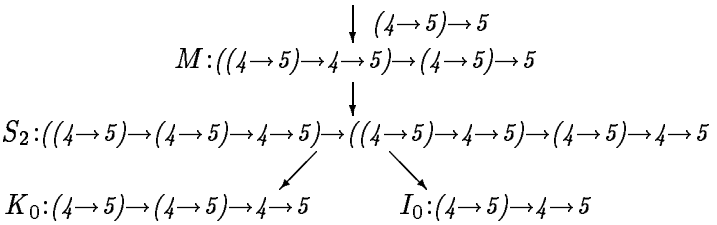
In this section we show that the naive type assignment is not safe: there are typeable rewrite rules that match a term M typeable with σ, for which the result of the application of the rewrite rule on M is not typeable with σ. We will formulate a condition typeable rewrite rules should satisfy in order to obtain preservance of types under rewriting. We will show that this condition is needed and sufficient.

4.1 Patterns cause problems

Definitions *3.2.1*, *3.2.2* and *3.2.3* define what a type assignment should be, just using the strategy as used in languages like for example Miranda. It is called *naive*, because it not sufficient to guarantee preservance of types under rewriting. (This is also called the 'subject reduction property'.) Not even typeability is kept under rewriting.

Take for example the definition of M as in example *2.3*, which can be typed as shown in example *3.4.2*. If we take the term $M(S_2(K_0, I_0))$ 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 in the following way:



the term $S_2(I_0, I_0)$ is not typeable with the type $(4\to 5)\to 5$. In fact, it is not typeable at all.

4.2 A needed and sufficient condition for preservance of types under rewriting

By definition *1.2*, if a term M is rewritten to the term M' using the rewrite rule $Lhs \to Rhs$, there is a subterm M_0 of M, and a replacement R, such that $\mathrm{Lhs}^{\mathrm{R}} = M_0$. M' is obtained by replacing M_0 by $\mathrm{Rhs}^{\mathrm{R}}$. To guarantee the subject reduction property, we should accept only those rewrite rules $Lhs \to Rhs$, that satisfy:

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

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

In the notion of type assignment as presented in this paper, it is easy to formulate a condition that rewrite rules should satisfy in order to be accepted.

Definition 4.2.1 *i)* We call a rewrite rule $Lhs \rightarrow Rhs$ *safe* if:

If $pp(Lhs) = <P, \pi>$, then $P \vdash Rhs{:}\pi$.

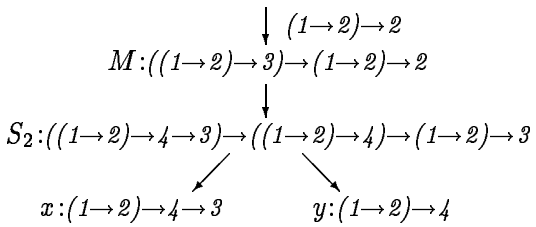
ii) The definition of a *safe type assignment with respect to* $\mathcal{E}$ is the same as the one for a naive type assignment, by replacing in definition *3.2.3* condition *(i.a)* and *(i.b)* by:

If $pp(Lhs) = <P, \pi>$, then $P \vdash Rhs{:}\pi$.

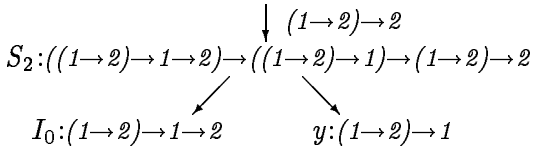
Notice that the notion $pp(M)$ is defined independently from the definition of typeable rewrite rules.

As an example of a rule that is not safe, take the typed rewrite rule in the next example: the types assigned to the nodes containing x and y are not the most general ones needed to find the type for the left hand side of the rewrite rule.

Example 4.2.2 Take the definition of M as in example *2.3*. To obtain $pp(M\,(S_2\,(x, y)))$, we assign types to nodes in the tree in the following way:



If the right hand side should be typed with $(1{\rightarrow}2){\rightarrow}2$, the type needed for the node containing y is $(1{\rightarrow}2){\rightarrow}1$.



So this rule is not safe, and should therefore be rejected. ■

In the following theorem, we prove that our solution is correct. The structure of the first proof depends greatly on the fact that for every type σ we can trivially find an $A \in \mathcal{F}$, such that $\mathcal{E}\,(A) = \sigma$: we just pick a constant A, not already used. We do not require in the proof that for every symbol A there is a rewrite rule that defines A.

Theorem 4.2.3 i) **The condition is needed.** *Let* $Lhs, Rhs \in T(\mathcal{F}, \mathcal{X})$, *and* $\mathbf{r} : Lhs \rightarrow Rhs$ *be a typeable rewrite rule that is not safe. Then there there exists a replacement* R, *and a type* μ, *such that* $\vdash Lhs^{\mathrm{R}}{:}\mu$ *&* $\neg \vdash Rhs^{\mathrm{R}}{:}\mu$.

ii) **The condition is sufficient.** *Let* $Lhs, Rhs \in T(\mathcal{F}, \mathcal{X})$, *and* $\mathbf{r} : Lhs \rightarrow Rhs$ *be a safe rewrite rule. Then for every replacement* R, *basis* B *and a type* μ: $B \vdash Lhs^{\mathrm{R}}{:}\mu \Rightarrow B \vdash Rhs^{\mathrm{R}}{:}\mu$.

Proof:i) Since $\mathbf{r}$ is typeable and left linear, we know that there are $\beta_1, \ldots, \beta_n, \tau$, and distinct $x_1, \ldots, x_n$, such that:

$\{x_1{:}\beta_1, \ldots, x_n{:}\beta_n\} \vdash Lhs{:}\tau$ & $\{x_1{:}\beta_1, \ldots, x_n{:}\beta_n\} \vdash Rhs{:}\tau$.

Then by theorem *3.3.4*, and lemma *3.3.5* we know that there are bases P_l, P_r, types α_1, $\ldots$, α_n, δ and a substitution S_0, such that:

$$pp(Lhs) = <P_l, \tau> \ \&\ pp(Rhs) = <P_r, \delta> \ \&\ P_l = \{x_1{:}\rho_1, \ldots, x_n{:}\rho_n\} \ \&$$
$$S_0(\tau) = S_0(\delta) = \tau \ \&\ S_0(P_l) = S_0(P_r) = \{x_1{:}\beta_1, \ldots, x_n{:}\beta_n\}.$$

Let S_1 be the substitution such that for every i: $S_1(\varphi_i) = c_i$ (the i-th type constant), μ be a type such that $S_1(\tau) = \mu$, $A_1, \ldots, A_n$ be constants such that for every $1 \leq i \leq n$, $\mathcal{E}(A_i) = S_1(\rho_i)$, and R be the replacement such that for every $1 \leq i \leq n$: ${x_i}^{\mathrm{R}} = A_i$. Then by lemma *3.3.6(ii)*: $\vdash \mathrm{Lhs}^{\mathrm{R}}{:}\mu$. (Notice that $\mathrm{Lhs}^{\mathrm{R}}$ does not contain term variables.) Since **r** is not safe, we know that

$$\neg\, \{x_1{:}\rho_1, \ldots, x_n{:}\rho_n\} \vdash Rhs{:}\tau.$$

Suppose towards a contradiction that $\vdash \mathrm{Rhs}^{\mathrm{R}}{:}\mu$.
Then by lemma *3.3.6(i)* there is a substitution S_2 such that

$$S_2(\delta) = \mu \text{ and for every } x{:}\gamma \in P_r{:}\ \vdash \mathrm{x}^{\mathrm{R}}{:}S_2(\gamma).$$

By definition *1.2(ii.a.3)* for every $x{:}\gamma \in P_r$ there is a $1 \leq i \leq n$ such that $x = x_i$. Since S_1 replaces type variables by type constants, the type assigned to A_i can only be $S_1(\rho_i)$. This implies that for every $x{:}\gamma \in P_r$ there is a $1 \leq i \leq n$ such that $x = x_i$ and $S_2(\gamma) = S_1(\rho_i)$. It is straightforward to verify that, since S_1 replaces type variables by type constants, there is a substitution S_3 such that $S_2 = S_1 \circ S_3$. So for every $x{:}\gamma \in P_r$ there is a $1 \leq i \leq n$ such that: $x = x_i$ and $\rho_i = S_3(\gamma)$. But then by lemma *3.3.3*:

$$\{x_1{:}\rho_1, \ldots, x_n{:}\rho_n\} \vdash Rhs{:}S_3(\delta).$$

Moreover, $\mu = S_1(\tau) = S_2(\delta) = S_1 \circ S_3(\delta)$, so $\tau = S_3(\delta)$.

ii) Since **r** is safe, there are P, π such that: if $pp(Lhs) = <P, \pi>$, then $P \vdash Rhs{:}\pi$
Suppose $pp(Lhs) = <P, \pi>$, and R is a replacement such that there are basis B and type μ such that $B \vdash \mathrm{Lhs}^{\mathrm{R}}{:}\mu$, then by lemma *3.3.6(i)* there is a substitution S such that

$$S(\pi) = \mu \ \&\ \forall\, x{:}\rho \in P \ [\ B \vdash \mathrm{x}^{\mathrm{R}}{:}S(\rho)\ ].$$

But then by lemma *3.3.3*:

$$S(P) \vdash Rhs{:}S(\pi) \ \&\ \forall\, x{:}\rho \in P \ [\ B \vdash \mathrm{x}^{\mathrm{R}}{:}S(\rho)\ ].$$

So by lemma *3.3.6(ii)*: $B \vdash \mathrm{Rhs}^{\mathrm{R}}{:}\mu$. ■

Notice that although the proof of part *(i)* uses explicitly the presence of type constants, the problem of loss of subject reduction also arises if type constants are not in the type system. However, we do not believe that it is possible to prove that the condition is also needed in that case. (See also conjecture *5.1*).

5 Optimisation after bracket abstraction

Bracket abstraction algorithms are used to translate lambda expressions to combinators. With some accompanying optimization rules they provide an interesting example. If in the bracket abstraction we would use the following combinator set:

$$\begin{aligned} I\ x &= x \\ K\ x\ y &= x \\ B\ x\ y\ z &= x\ (y\ z) \end{aligned}$$

$C\ x\ y\ z = x\ z\ y$
$S\ x\ y\ z = x\ z\ (y\ z)$

then we could define, as Curry did in [Curry & Feys '58], the following optimizations:

$S\ (K\ x)\ (K\ y) = K\ (x\ y)$
$S\ (K\ x)\ I \quad\ = x$
$S\ (K\ x)\ y \quad\ = B\ x\ y$
$S\ x\ (K\ y) \quad\ = C\ x\ y$

We can see that these rules work on combinators that have not yet been applied to all their arguments. When using the environment:

$\mathcal{E}\,(Opt) = (1 \to 2) \to 1 \to 2,$
$\mathcal{E}\,(I) = 1 \to 1,$
$\mathcal{E}\,(K) = 1 \to 2 \to 1,$
$\mathcal{E}\,(B) = (1 \to 2) \to (3 \to 1) \to 3 \to 2,$
$\mathcal{E}\,(C) = (1 \to 2 \to 3) \to 2 \to 1 \to 3,$
$\mathcal{E}\,(S) = (1 \to 2 \to 3) \to (1 \to 2) \to 1 \to 3$

the rewrite rules for I, K, B, C, and S and their Curried versions, and rewrite rules for a function Opt as below (where the rules that are not safe are preceded by '•'), can be typed. These rules for Opt are only typeable using Mycroft's approach, not using Milner's.

- $Opt\,(S_2\,(K_1\,(x), K_1\,(y))) \to K_1\,(Ap\,(Opt\,(x), Opt\,(y)))$
  $Opt\,(S_2\,(K_1\,(x), I_0)) \to Opt\,(x)$
  $Opt\,(S_2\,(K_1\,(x), y)) \to B_2\,(Opt\,(x), Opt\,(y))$
- $Opt\,(S_2\,(x, K_1\,(y))) \to C_2\,(Opt\,(x), Opt\,(y))$

(The rules for Opt are not intended to be complete.) They can now be used in combination with the previous rules for bracket abstraction to produce simplified combinator expressions. For example, the combinator expression that would be produced for $\lambda x.\lambda y.x$ (i.e. $S\ (K\ K)\ I$) will be evaluated as below:

$Opt(Ap\,(Ap\,(S_0, Ap\,(K_0, K_0)), I_0)) \to^* K_0$

This is what we would expect for the above lambda expression.

Conjecture 5.1 As indicated above, some of the rewrite rules for Opt are not safe. By theorem 4.2.3(i) we can give a term M, typeable by σ such that one of the unsafe rewrite rules matches M, but the result of rewriting M with this rule is not typeable with σ. However, in this construction we use that the type system contains also type constants. If we would use a type system without type constants, we conjecture that the safeness condition can be formulated in such a way that the rewrite rules for Opt are all safe. ■

Future work

Although we could very well motivate that for type assignment with Curry types the LLATRS's are the best equipped, it seems natural to generalize our notion of type assignment to the non-left linear systems, since in a number of functional programming languages it is possible to

have non-left linear rewrite rules. The only property that is lost in this generalization is the neededness of the condition on rewrite rules, since only in that proof the left linearity of rewrite rules is used. Another approach to this problem could be to apply our notion of type assignment to conditional term rewriting systems.

It seems straightforward to use our notion of type assignment in term graph rewriting systems. However, some accuracy will very likely be lost, and it seems necessary to base such a notion on a (restriction of) the intersection type assignment system. At the moment a paper on partial intersection type assignment of Rank 2 is in preparation. This system also seems promising for the definition of functions in functional programming languages that have internal polymorphism and for overloaded definitions.

Jan Willem Klop asked us if we could characterize the kind of rewrite rules that are typeable in our system. At this moment we have no direct answer to that question, but it will be interesting to explore this.

We are looking for good examples of areas where our notion of type assignment can be useful. It is probably not very interesting in an executable system, because by allowing arbitrary function symbols in patterns we loose the Church-Rosser property. Their main use could be providing type checking (and eventually type inference) for people who are writing specifications in the style of the rewrite systems we deal with. One of the main areas could be in program transformation and semantics.

Acknowledgement

The main problem solved in this paper (as given in section *4.1*) was first brought to our attention by M. Coppo and M. Dezani-Ciancaglini of the university of Turin, Italy. The construction used in the proof of theorem *4.2.3(i)* is partly due to M. van Eekelen, of the university of Nijmegen, the Netherlands.

Appendix: A type check algorithm for LLATRS's

In this section we present a type check algorithm that, when applied to a LLATRS and a environment determines whether this LLATRS is safely typeable with respect to the environment. The algorithm is specified in a notation based on the programming language Clean.

A.1 Representing LLATRS's and Curry types

Since type assignment is syntax directed, it is convenient to make the structure of an LLATRS explicit, for such a structure enables us to use pattern matching to distinguish the different syntactical parts. In Clean new data structures are defined by means of algebraic type definitions, and Clean uses square brackets (i.e. '[' and ']') to denote list types.

The algebraic type definitions for LLATRS's that we use are the following:

```
TYPE
:: TRS      → [Rule]                                              ;
:: Rule     → (Term, Term)                                        ;
:: Term     → Var Variable | Symb Symbol [Term] | DefSymb Symbol [Term] ;
:: Variable → x | ...                                             ;
:: Symbol   → Ap | ...                                            ;
```

The last alternative for Term (defining the constructor DefSymb) is used to indicate that the corresponding (sub)term starts with the defined symbol (definition *1.2(iii.a)*). So, using DefSymb makes only sense when specifying the left hand side of a rewrite rule. Note that it is very well possible to determine the defining node of a certain left hand side of a rewrite rule. This however would make the algorithm somewhat more complicated and therefore less clear.

A basis (definition *3.1.1(iii)*) is a set of statements with term variables as subjects. Since there are no sets in Clean we use lists instead. Hence, the representation of bases becomes:

```
TYPE
:: Statement → (Variable, CurryType) ;
:: Basis     → [Statement]           ;
```

A.2 The type check algorithm

The goal of the type check algorithm presented below is to determine whether a safe type assignment can be constructed such that all the conditions of definitions *3.2.2* and *3.2.3* are satisfied. The main function of the algorithm, called TypeTRS, expects a LLATRS as well as a environment as parameters. It returns a boolean that indicates whether the construction of the type assignment was successful.

It is easy to prove that the algorithm presented here is correct and complete:

Theorem A.2.1 i) If M is typeable with respect to the environment $\mathcal{E}$, then TypeTerm M $\mathcal{E}$ *returns pp(M).*

ii) If TypeTerm M $\mathcal{E}$ *returns the pair $<B, \sigma>$, then pp(M) $= <B, \sigma>$.*

iii) There is a safe type assignment with respect to $\mathcal{E}$ for the LLATRS TRS*, if and only if* TypeTRS TRS $\mathcal{E}$ *returns* TRUE.

Proof: By straightforward induction on the structure of terms and rewrite rules. ■

The algorithm does not perform any consistency check on its input so it assumes the input to be correct according to definitions *1.1* and *1.2*. Moreover, all possible error messages and error handling cases are omitted, and the algorithm returns only TRUE for rewrite systems that are safely typeable. It could easily be extended to an algorithm that accepts unsafe rewrite rules.

The algorithm as presented here is not purely functional. The 0-ary function FreshVariable (used amongst others by TypeSymb and TypeVar) is supposed to return a new, unused type variable. It is obvious that such a function is not referential transparent, but for the sake of readability, we prefer not to be explicit on the handling of typevariables.

```
:: TypeTRS TRS Environment → BOOL                                    ;
   TypeTRS [ ] E           → TRUE                                    |
   TypeTRS [r | rules] E   → And (TypeRule r E) (TypeTRS rules E) ;
```

Constraint *3.2.2(i.b)* describes that for every term variable occurring in a rewrite rule, all the nodes containing this variable in the bases for both the left and the right hand side should contain equal predicates. This is provided in TypeRule by unifying all predicates for each subject occurring in the bases after a rule has been type checked. These unifications are performed by the function UnifyBasis. TypeRule also takes care of checking the safeness constraint as given in definition *4.2.1(ii)*, by checking, using EqualBases, if the unification of left and right hand sides of a rewrite rule has changed the left hand side basis.

:: TypeRule Rule Environment → BOOL ;
TypeRule (Lhs, Rhs) $\mathcal{E}$ → EqualBases ($S_2(S_1(B_l))$) B_l
S_2: UnifyBasis ($S_1(B_l) \cup S_1(B_r)$),
S_1: UnifyTypes σ τ,
(B_l, σ): TypeTerm Lhs $\mathcal{E}$,
(B_r, τ): TypeTerm Rhs $\mathcal{E}$;

The type of a symbol is either an instance of the type for that symbol given by the environment (in case of a Symb) or that type itself (in case of a DefSymb). The distinction between the two is determined by the function TypeTerm. The defining node of a rewrite rule can only be typed with *one* type, so any substitution resulting from a unification is forbidden to change this type. This is solved by the unification algorithm UnifyTypes, an extension of Robinsons unification algorithm; it is capable of deciding whether or not a type variable is special (so it appears in some environment type) and it refuses to substitute this variable by other types.

:: TypeTerm Term Environment → (Basis, CurryType) ;
TypeTerm (Var x) $\mathcal{E}$ → TypeVar x |
TypeTerm (Symb F args) $\mathcal{E}$ → TypeSymb (Instance ($\mathcal{E}(F)$)) args $\mathcal{E}$ |
TypeTerm (DefSymb F args) $\mathcal{E}$ → TypeSymb ($\mathcal{E}(F)$) args $\mathcal{E}$;

The function TypeSymb verifies the constraints *3.2.2(i.b)* by the means of the method indicated by definition *3.3.2*.

:: TypeSymb CurryType [Term] Environment → (Basis, CurryType) ;
TypeSymb σ [ ] $\mathcal{E}$ → (B, σ) |
TypeSymb σ terms $\mathcal{E}$ → ($S(B)$, $S(\varphi)$),
S: UnifyTypes τ σ,
(B, τ): TypeArguments terms φ $\mathcal{E}$,
φ: FreshVariable ;

TypeArguments is an auxiliary function that derives the types for all arguments of a node and constructs an arrow type out of these types. Note that this function expects the last type of the arrow type that is constructed as a parameter: the type variable φ is passed to the function TypeArguments to make sure that the type $\sigma_1 \to \cdots \to \sigma_n \to \varphi$ is created, instead of $(\sigma_1 \to \cdots \to \sigma_n) \to \varphi$. Besides the adjusted basis, this arrow type is delivered as a result.

::TypeArguments [Term] CurryType Environment → (Basis, CurryType) ;
TypeArguments [ ] σ $\mathcal{E}$ → ($\emptyset$, σ) |
TypeArguments [M | terms] σ $\mathcal{E}$ → (B, $\sigma_1 \to \sigma_2$),
B: $B_1 \cup B_2$,
(B_1, σ_1): TypeTerm M $\mathcal{E}$,
(B_2, σ_2): TypeArguments terms σ $\mathcal{E}$;

TypeVar assigns a new type variable φ to a term variable x.

:: TypeVar Variable → (Basis, CurryType) ;
TypeVar x → ([(x, φ)], φ),
φ: FreshVariable ;

References

[Barendregt '84] Barendregt H. *The Lambda Calculus: its Syntax and Semantics.* North-Holland, Amsterdam, revised edition, 1984.

[Barendregt *et al.* '83] Barendregt H., 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.

[Barendregt *et al.* '87] Barendregt H.P., 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.

[Brus *et al.* '87] Brus T., 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.

[Curry & Feys '58] Curry H.B. and R. Feys. *Combinatory Logic.* volume 1. North-Holland, Amsterdam, 1958.

[Damas '85] Damas L.M.M. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, Department of Computer Science, Edinburgh, 1985. Thesis CST-33-85.

[Futatsugi *et al.* '85] Futatsugi K., J. Goguen, J.P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proceedings 12^{th} ACM Symposium on Principles of Programming Languages*, pages 52–66, 1985.

[Hindley '69] Hindley J.R. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[Kfoury *et al.* '88] Kfoury A.J., J. Tiuryn, and P. Urzyczyn. A proper extension of ML with an effective type-assignment. In *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 58–69, San Diego, California, 1988.

[Klop '92] Klop J.W. 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.

[Milner '78] Milner R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mycroft '84] Mycroft A. Polymorphic type schemes and recursive definitions. In *Proceedings of the International Symposium on Programming, Toulouse*, volume 167 of *Lecture Notes Computer Science*, pages 217–239. Springer-Verlag, 1984.

[Nöcker *et al.* '91] Nöcker E.G.J.M.H., 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.

[Pfenning '88] Pfenning F. 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.

[Robinson '65] Robinson J.A. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Turner '85] Turner D.A. 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.