Rank 2 Intersection Type Assignment in Term Rewriting Systems

(Fundamenta Informaticae, 26(2):141-166, 1996)

Steffen van Bakel

Dipartimento di Informatica, Università degli Studi di Torino,
Corso Svizzera 185, 10149 Torino, Italia
bakel@di.unito.it

Abstract

A notion of type assignment on Curryfied Term Rewriting Systems is introduced that uses Intersection Types of Rank 2, and in which all function symbols are assumed to have a type. Type assignment will consist of specifying derivation rules that describe how types can be assigned to terms, using the types of function symbols.
Using a modified unification procedure, for each term the principal pair (of basis and type) will be defined in the following sense: from these all admissible pairs can be generated by chains of operations on pairs, consisting of the operations substitution, copying, and weakening.
In general, given an arbitrary typeable $\mathcal{G}i$TRS, the subject reduction property does not hold. Using the principal type for the left-hand side of a rewrite rule, a sufficient and decidable condition will be formulated that 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 (LC) [10], but also Term Rewriting Systems (TRS) [28] and Term Graph Rewriting Systems (TGRS) [12] are topics of research. LC (or rather Combinator Systems) constitutes the underlying model for the functional programming language Miranda [37], TRS were used in the underlying model for the language OBJ [22], and TGRS were the model for the language Clean [13, 32].

For the implementation of a language, independent of the chosen implementation model, the notion of types plays an important role. Types are essential to obtain efficient machine code when compiling a program and are also used to make sure that the programmer has a clearer understanding of the programs that are written. In fact, type assignment to programs and objects is a way of performing abstract interpretation that provides necessary information for both compilers and programmers.

Since functional programming languages had their origin in LC, formal notions like type assignment or strictness analysis, used for those languages, are often studied on the level of LC. Several authors, when presenting new notions of type assignment to be used in programming, presented their results within (extended) lambda calculi (see, for example, [30], and [31]) that are normally extensions of the Curry Type Assignment System [19].

The scope of this paper is to develop decidable intersection type assignment in the context of TRS. The restriction studied in this paper of the Intersection Type Discipline (ITD) for LC as presented in [16, 11, 1, 3] is the one suggested in [29]: the limitation of the set of types to intersection types of Rank 2. In that paper is stated that (part of) the type assignment system

for ML can be seen as a restriction of ITD when types are limited in that way. However, using Rank 2 intersection types significantly extends the set of typeable terms as well as the accuracy of derivable types. Moreover, sometimes a program that is correct in the programmers mind can be rejected because of type errors, while it could be accepted after the programmer has rewritten the specification. Such a rewrite would not be necessary if Rank 2 types are used (see Subsection 1.2 for an example).

In [2], Curryfied Term Rewriting Systems (*Gi*TRS) were defined as the TRS that contain a special binary operator Ap.[1] The motivation for the use of *Curryfied* TRS instead of the general first-order TRS is the following. The notion of type assignment as studied in this paper uses higher-order types: in general, a term t can have a type $\sigma \to \tau$, a type that expresses that t is considered to be a function. Given a term t' of type σ, it is natural to allow for the application of t to t', creating in this way a term of type τ. This calls for the introduction of a notion of explicit application into the first-order rewrite systems, and, therefore, in this paper systems equipped with a binary function Ap are considered. Using this Ap, also Curryfied versions of function symbols (other than Ap), and related rewrite rules can be expressed, thus enlarging the expressive power of the rewrite system. In this way, it is possible to formulate a typeable first-order rewrite system that has full Turing-machine power (see Definition 2.7).

For these systems, in [4] strong normalization of typeable terms has been proved, provided that recursive rules do not surpass the complexity of primitive recursion (for details, see [4]); a head-normalization result was shown to hold in [5] for systems that are a slight restriction of those considered in [4]. The notion of type assignment presented here is a restriction of those presented in [2, 4, 5] in that types are restricted to those of Rank 2; in particular, the result of [4] applies: given the there defined restriction on recursive rewrite rules, typeable terms are strongly normalizable.

Although it may seem straightforward to generalize type assignment systems for LC to the (significantly larger) world of TRS, it is not evident that those borrowed systems have still all the properties they possessed in the world of LC. For example, type assignment in TRS in general does not satisfy the subject reduction property, i.e.: types are not preserved under rewriting, as illustrated in [6]. Moreover, this paper aims to present type assignment for languages that allow for patterns, and as discussed in Section 6.1, notions of type assignment with intersection types for LC and TRS are, in general, incomparable, and cannot be ported from one to the other.

Recently, some results have been obtained in the field of typed TRS [20] and the combination of those with (intersection) type assignment systems for LC (e.g. [7], [8], [9]).

Using essentially the solution of [6], also for the system as presented in this paper we will prove that type assignment is closed for subject reduction. To obtain this result, first the three operations specified (Subsection 3.1) are proven to be sound on typeable terms (Theorem 3.8). Then principal pairs are defined for terms (Definition 4.2), followed by the proof that every typeable term has a principal pair (Theorem 4.13). Using the principal pair of the left-hand side, type assignment on rewrite rules is defined (Definition 5.1) that is proven to be sufficient for the subject reduction result (Theorem 5.6). Since it is decidable if a term has a principal type, also the restrictions that rewrite rules should satisfy to obtain subject reduction are decidable.

[1] In fact, there the name *Applicative* TRS is used; the set-up of the systems defined in this paper is almost the same as the one used there.

1 Context of this paper

1.1 Rank 2 type assignment for Lambda Calculus

In this subsection, we will briefly discuss *a* notion of Rank 2 type assignment for LC (the system presented here is not the only one possible: a variant could be to consider also the empty intersection, but we will not take that direction here).

Intersection types of Rank 2 are a true subset of the set of intersection types as defined in [16, 11, 1, 3], and only a minor extension of the set of Curry-types. They are defined by:

Definition 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*) If $\sigma, \tau \in \mathcal{T}_C$, then $\sigma{\to}\tau \in \mathcal{T}_C$.

ii) $\mathcal{T}_1$ is defined by: If $\sigma_1 \ldots, \sigma_n \in \mathcal{T}_C$ $(n \geq 1)$ then $\sigma_1 \cap \cdots \cap \sigma_n \in \mathcal{T}_1$.

iii) $\mathcal{T}_2$ is inductively defined by:
- *a*) If $\sigma \in \mathcal{T}_C$, then $\sigma \in \mathcal{T}_2$.
- *b*) If $\sigma \in \mathcal{T}_1$, $\tau \in \mathcal{T}_2$, then $\sigma{\to}\tau \in \mathcal{T}_2$.

iv) $\mathcal{T}_R$, the set of *intersection types of Rank 2* is defined by: if $\sigma_1 \ldots, \sigma_n \in \mathcal{T}_2$ $(n \geq 1)$ then $\sigma_1 \cap \cdots \cap \sigma_n \in \mathcal{T}_R$.

The next definition presents a partial order relation $\leq$ on $\mathcal{T}_R$, that is induced by intersections. This relation is used to define an equivalence relation $\sim$ on types. Types σ and τ are equivalent under this relation if σ can be obtained from τ by permuting subtypes that are part of an intersection subtype.

Definition 1.2 *i*) On $\mathcal{T}_R$, the relation $\leq$ is defined by:
- *a*) $\forall\, 1 \leq i \leq n\ (n \geq 1)\ [\sigma_1 \cap \cdots \cap \sigma_n \leq \sigma_i]$.
- *b*) $\forall\, 1 \leq i \leq n\ (n \geq 1)\ [\sigma \leq \sigma_i] \Rightarrow \sigma \leq \sigma_1 \cap \cdots \cap \sigma_n$.
- *c*) $\sigma \leq \tau \leq \rho \Rightarrow \sigma \leq \rho$.

ii) On $\mathcal{T}_R$, the relation $\sim$ is defined by:
- *a*) For $\sigma, \tau \in \mathcal{T}_R$: $\sigma \leq \tau \leq \sigma \Rightarrow \sigma \sim \tau$.
- *b*) For $\sigma{\to}\tau, \rho{\to}\mu \in \mathcal{T}_R$: $\sigma \sim \rho \ \&\ \tau \sim \mu \Rightarrow \sigma{\to}\tau \sim \rho{\to}\mu$.

In this paper, types are considered modulo $\sim$. Therefore, $\rho \cap (\sigma \cap \tau) = (\rho \cap \sigma) \cap \tau$, and $\sigma \cap \sigma{\to}\tau = \sigma{\to}\tau$. Unless stated otherwise, if $\sigma_1 \cap \cdots \cap \sigma_n$ is used to denote a type, all $\sigma_1 \ldots, \sigma_n$ are assumed to be in $\mathcal{T}_2$.

Definition 1.3 *i*) A *statement* is an expression of the form $M{:}\sigma$, where $M \in \Lambda$ and $\sigma \in \mathcal{T}_R$. M is the *subject* and σ the *predicate* of $M{:}\sigma$.

ii) A *basis* is a set of statements with distinct term-variables as subjects and types in $\mathcal{T}_1$ as predicates.

iii) Two types (bases, pairs of basis and type) are *disjoint* if and only if they have no type-variables in common.

Notice that, in bases, only types in $\mathcal{T}_1$ are allowed as predicates.

Definition 1.4 *i*) The relation $\leq$ is extended to bases by:

$$B \leq B' \iff \forall\, x{:}\sigma' \in B'\ \exists\, x{:}\sigma \in B\ [\sigma \leq \sigma'].$$

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

Notice that $\Pi\{B_1, \ldots, B_n\}$ is well defined, since if $\sigma_1, \ldots, \sigma_m$ are predicates of statements in $B_1 \cup \cdots \cup B_n$, then all $\sigma_1, \ldots, \sigma_m$, and $\sigma_1 \cap \cdots \cap \sigma_m$ are elements of $\mathcal{T}_1$.

Definition 1.5 For LC, *Rank 2 type assignment* and *Rank 2 derivations* are defined by:

$$(\to\text{I}):\ \frac{\begin{array}{c}[x{:}\sigma_1]\ \cdots\ [x{:}\sigma_n] \\ \vdots \\ M{:}\tau\end{array}}{\lambda x.M{:}\sigma\to\tau}\ (a) \qquad (\to\text{E}):\ \frac{M{:}\sigma\to\tau \quad N{:}\sigma}{MN{:}\tau}$$

$$(\cap\text{I}):\ \frac{M{:}\sigma_1 \quad \ldots \quad M{:}\sigma_n}{M{:}\sigma_1 \cap \cdots \cap \sigma_n}\ (n \geq 1)$$

(a) If $x{:}\sigma_1, \ldots, x{:}\sigma_n$ are all and nothing but the statements about x on which $M{:}\tau$ depends, and $\sigma \leq \sigma_1 \cap \cdots \cap \sigma_n$. If x does not occur free in M, so no statement with subject x is used to obtain $M{:}\tau$, then $\sigma \in \mathcal{T}_1$.

It is possible to show that the system presented in this way has the principal type property, and that type assignment is decidable. The technique to prove the first of these properties is very similar to the one used in this paper: specifying operations on types that are proven sound, to define principal pairs and to show that the specified operations are complete, i.e. every correct pair for a typeable term can be obtained from its principal pair by applying some sequence of operations to it. Because of the strong similarity in approach, we will not present the details of such a result; instead, in Subsection 6.1, we will briefly discuss the fundamental differences between the two systems.

To avoid confusion, it is necessary to point out that there also exists a notion of type assignment that is called the Rank 2 *Polymorphic* Type Assignment System, defined in [25]. This system is an extension of Milner's system, by allowing for the $\forall$-type constructor to occur also on the left-hand side of an arrow-type, instead of only at top level. (It is also a restriction of the Polymorphic Type Discipline [23], where types are restricted to polymorphic types of Rank 2.) As in the system presented here, type assignment in that system is decidable.

1.2 Intersection type assignment versus ML type assignment

In [29] was remarked that (part of) the ML Type Assignment System [30] can be seen as a restriction of the ITD [11] by limitation of the set of types to intersection types of Rank 2. This observation can be understood by the following intuitive argument:

The ML Type Assignment System is in fact a type assignment system for an extended lambda calculus. This calculus is defined by:

Definition 1.6 *i*) The set of ML terms, Exp, is defined as Λ, the set of lambda terms, extended by:
  a) If $M, N \in \mathsf{Exp}$, and x a term-variable, then $(\mathsf{let}\ x = N\ \mathsf{in}\ M) \in \mathsf{Exp}$.
  b) $\mathsf{Y} \in \mathsf{Exp}$.

ii) The notion of reduction on Exp, $\to_{\text{ML}}$, is defined as $\to_\beta$, extended by:
  a) $(\mathsf{let}\ x = N\ \mathsf{in}\ M) \to_{\text{ML}} M[N/x]$.
  b) $\mathsf{Y}M \to_{\text{ML}} M(\mathsf{Y}M)$.

With this extended notion of reduction, the terms (let $x = N$ in M) and $((\lambda x.M)N)$ are both denotations for reducible expressions (redexes) that both reduce to the term $M[N/x]$. However, the semantic interpretation of these terms is different (for details of this semantic, see [30]). The term $((\lambda x.M)N)$ is interpreted as a function with an operand, whereas the term (let $x = N$ in M) is interpreted as the term $M[N/x]$ would be interpreted. This difference is reflected in the way the type assignment system treats these terms.

In fact, the let-construct is added to ML to cover precisely those cases in which the term $((\lambda x.M)N)$ is not typeable, but the contraction $M[N/x]$ is, while it is desirable for the term $((\lambda x.M)N)$ to be typeable. The problem to overcome is that, in assigning a type to $((\lambda x.M)N)$, the term-variable x can only be typed with *one* Curry-type; this is not required for x in (let $x = N$ in M). When assigning a type to that term, first the 'operand' N is typed by, say, the Curry-type σ. Suppose M is typeable with the type τ, and the n free occurrences of x in M are typed by the Curry-types $\sigma_1 \ldots, \sigma_n$ respectively. If for every σ_i there is a substitution S_i such that $S_i(\sigma) = \sigma_i$, then also (let $x = N$ in M) is typeable by τ.

$$\frac{\dfrac{B \cup \{x{:}\sigma\} \vdash x{:}\sigma}{B \cup \{x{:}\sigma\} \vdash x{:}\sigma_1}\ (\text{INST}) \quad \cdots \quad \dfrac{B \cup \{x{:}\sigma\} \vdash x{:}\sigma}{B \cup \{x{:}\sigma\} \vdash x{:}\sigma_n}\ (\text{INST}) \qquad \vdots \qquad B \cup \{x{:}\sigma\} \vdash M{:}\tau \qquad B \vdash N{:}\sigma}{B \vdash (\text{let } x = N \text{ in } M){:}\tau}\ (\text{LET})$$

Under those conditions, however, the term $((\lambda x.M)N)$ can be typed in the Rank 2 system, because there the term $(\lambda x.M)$ can be typed by $\sigma_1 \cap \cdots \cap \sigma_n \rightarrow \tau$. Also, since $B \vdash N{:}\sigma$, and type assignment in the Rank 2 system is closed for substitution of types, N is typeable by every σ_i. So, when using intersection types, the let-construct is not needed.

$$\frac{\dfrac{\begin{array}{c}[x{:}\sigma_1] \quad \cdots \quad [x{:}\sigma_n] \\ \vdots \\ M{:}\tau\end{array}}{\lambda x.M{:}\sigma_1 \cap \cdots \cap \sigma_n \rightarrow \tau} \qquad \dfrac{N{:}\sigma_1 \quad \cdots \quad N{:}\sigma_n}{N{:}\sigma_1 \cap \cdots \cap \sigma_n}}{(\lambda x.M)N{:}\tau}$$

Notice that the construction sketched above uses only Rank 2 intersection types.

The Rank 2 system and Milner's system are not really equivalent, because there are terms that are typeable in the former and not typeable in the latter, like the term $\lambda x.xx$. Moreover, when using the ML-type checker it can be that a program is rejected because of occurring type conflicts, whereas it could be accepted after the programmer has rewritten the specification, by performing in advance some of the reductions. Such a rewrite would not be necessary if Rank 2 types are used.

Example 1.7 Take the following Miranda program:

```
Add x y          = x + y
LengthList [ ]   = 0
LengthList (a:b) = 1 + (LengthList b)
F f g c i        = f (g c) (g i)
F Add LengthList ['a','b','c'] [1,2,3]
```

When using Milner's approach to type this program (as is done in Miranda), the last term in this program gives a type-error, since the type derived for the symbol F is:

$$\mathsf{F} :: (**\rightarrow**\rightarrow***) \rightarrow (*\rightarrow**) \rightarrow * \rightarrow * \rightarrow ***,$$

and the types [char] and [num] cannot be unified. (Notice that the definition for F corresponds to the ML-term $(\lambda fgci.f(gc)(gi))$.) It is possible to modify this into a typeable program, by replacing the definition for F:

```
Add x y           = x + y
LengthList [ ]    = 0
LengthList (a:b)  = 1 + (LengthList b)
F f c i           = f (LengthList c) (LengthList i)
F Add ['a','b','c'] [1,2,3]
```

but of course this is not the same program. Notice that in this modification, the definition for F has been replaced by the 'ML-term' (let g = LengthList in $\lambda fci.f(gc)(gi)$).

Using intersection types, however, the first definition of F is typeable with

$$\mathsf{F} :: (**\rightarrow*****\rightarrow***) \rightarrow ((*\rightarrow**)\cap(****\rightarrow*****)) \rightarrow * \rightarrow **** \rightarrow ***,$$

so third and fourth argument need not be of the same type, which makes the last term typeable. Notice that this last type for F is an intersection type of Rank 2.

But not only the class of typeable terms is significantly extended when intersection types of Rank 2 are used, also more accurate types can be deduced for terms. For example, the term $SKSI$ (where S, K and I are the well-known lambda terms) has in the Rank 2 system a more general principal type than in the ML system; in the notion of type assignment as presented in the previous subsection, the principal type for $SKSI$ is $\varphi\rightarrow\varphi$, whereas in Milner's system it is $(\varphi_0\rightarrow\varphi_1)\rightarrow\varphi_0\rightarrow\varphi_1$ (see Example 4.8). This implies, for instance, that more accurate types can be deduced for programs that are translated into combinator expressions.

The here noted equivalence gives rise to the idea that the ML-Type Assignment System (and in particular, the unification algorithm for that system), and the limitation of the ITD to Rank 2 are as far as decidability is concerned, equivalent. In fact, the results of this paper show that type assignment in the here presented notion of Rank 2 type assignment is decidable. This is accomplished mainly by showing that the unification procedure as defined in this paper is always terminating.

1.3 *Gi*TRS versus Function-Constructor systems

The kind of rewrite systems presented in this paper is an extension to those suggested by most functional programming languages. Such languages, like Miranda for instance, allow for the formal operand of a function to have structure. This makes definitions like

```
In-left(Pair(x,y))    → x
In-right(Pair(x,y))   → y
```

possible. The subterm Pair(x,y) in the definitions of both In-left and In-right is called a *pattern*, and the term In-right t, for example, can only be reduced when there are terms t_1 and t_2 such that $t \equiv \mathsf{Pair}(t_1,t_2)$. As suggested by this example, languages like Miranda allow programmers to specify an algorithm (function) as a set of rewrite rules, although there is a restriction on

the *kind* of patterns that is allowed: the symbols of the language are divided in two groups, *function symbols* and *constructors*. Constructors are meant to construct objects of a specific algebraic data type (hence their name), and are only allowed to occur in a pattern when supplied with all the required arguments.

The reason to distinguish between function symbols and constructors is fundamental, and lies directly in the fact that programming languages in this class are in fact sugared lambda calculi: only those patterns are allowed that can be translated to LC. When translating function definitions using patterns to pure lambda terms through a mapping [[ ]], at a certain stage it is necessary to deal with the pattern. One approach could be to, given the rewrite rule $\mathsf{F(P)} \to \mathsf{E}$, define that

$$[\![\mathsf{F}]\!] = (\lambda v.\text{IF } (v = [\![\mathsf{P}]\!])\ [\![\mathsf{E}]\!]\ \text{FAIL})$$

but this gives only a solution for certain cases. The problem is that the function '=', i.e. equality between lambda terms, cannot be expressed *in* LC: there exist no lambda term that is capable of deciding if two terms are the same (this is known as the problem of separability). Only in specific cases, like for example when dealing with Church-numerals, lists, or pairs, it is possible to express equality. To guarantee that patterns in function-constructor systems can be adequately translated to LC, the only patterns allowed are those based on data structures, using constructors (see, for an extensive treatment, Chapter Six of [34]). So functions symbols are *not* allowed to occur in patterns; for example, a definition like

$$\mathsf{Pair(In\text{-}left(x),In\text{-}right(x))} \to \mathsf{x}.$$

is, given the two rules above, not allowed.

A difficulty with these three rules together, is that they form Klop's famous 'Surjective Pairing' example [27]; this function cannot be expressed in LC because when added to LC, the Church-Rosser property no longer holds. This implies that, although both LC and TRS are Turing-machine complete, there is no general syntactic solution for patterns in LC, so a full-purpose translation (interpretation) of TRS in LC is not feasible. It is this fundamental impossibility that prohibits rules like the last one: in order to be able to apply that rule to a term

$$\mathsf{Pair(In\text{-}left}(t_1)\mathsf{),(In\text{-}right}(t_2)\mathsf{)},$$

the terms t_1 and t_2 have to be *equivalent*, something that cannot be expressed in LC.

For a very elegant discussion of a lambda calculus with patterns, see [33].

The kind of programming language we aim at uses more general rewrite systems than just function-constructor systems. The systems considered in this paper do not discriminate against the varieties of function symbols that can be used in patterns. As such there is no distinction between function symbols and constructor symbols; the extension made consists of allowing for not only constructor-symbols in the operand space of the left-hand side of rewrite rules, but all function symbols. Since function-constructor systems are a true restriction of the systems considered here, the results obtained in this paper apply also there.

1.4 The limitations of many-sorted rewrite systems

One way to study type assignment on TRS is to work within the framework of first-order many-sorted rewrite systems, as used in the underlying model for the language OBJ [22]. The differences between that approach and the one taken in this paper are significant.

First of all, first-order many-sorted rewrite systems are far less general than those suggested by functional programming languages: rewrite rules are considered to specify operations over data-types, instead of over arbitrary objects. This implies that an enumerable collection of *sorts* is defined, and it is assumed that every F with arity n has a type $s_1 \times \cdots \times s_n \to s_{n+1}$, where s_1, $\ldots$, s_{n+1} are sorts. Using this approach, every F has in fact *only one* type, so in particular no function symbol can be called polymorphic. Moreover, the biggest shortcoming of this approach is that neither one of the arguments of a function symbol, nor the result of applying a function symbol to sufficiently many arguments can have a type that is not a sort: 'higher-order' types are not allowed.

The notion of type assignment as presented in this paper is combining the approach taken in those multi-sorted, first-order rewrite systems, with the one commonly used for type assignment in LC. Compared to the multi-sorted systems, the main change is to allow for higher-order types. In multi-sorted systems, a term containing the binary function Ap can be typed, but *only in one* way; by definition, there are sorts s_1, s_2 and s_3 such that Ap has type $s_1 \times s_2 \to s_3$. In order to get a notion of type assignment that resembles notions for LC, in this paper the type used for Ap is the one implicitly used in the derivation rule ($\to$E). That rule describes what the relation is between the types assigned to the left-hand term in an application, to the right-hand term, and to the application itself.

$$(\to\text{E}): \quad \frac{M{:}\sigma \to \tau \qquad N{:}\sigma}{MN{:}\tau}$$

This scheme gives that the natural type-scheme for Ap should be $(\sigma \to \tau) \times \sigma \to \tau$ – or, in a different notation, $(\sigma \to \tau) \to \sigma \to \tau$ – so in particular the left-hand argument has an arrow-type. This extension invokes the possibility to assign arrow-types to all objects.

2 Curryfied Term Rewriting Systems

In this paper, type assignment on Curryfied Term Rewriting Systems is studied, that are defined as a slight extension of TRS as defined in [28] or [20]. In the literature, several different formal definitions of TRS exist. The one chosen in this paper is that of functional, first-order systems: terms are constructed from term variables and function symbols that have a fixed arity greater than or equal to zero, and each function symbol can only be used with the right amount of arguments present.

Definition 2.1 An *alphabet* or *signature* Σ consists of:

i) A countable infinite set $\mathcal{X}$ of variables x_1, x_2, x_3, $\ldots$ (or x, y, z, $\ldots$).

ii) A non-empty set $\mathcal{F}$ of *function symbols* F, G, $\ldots$, each with an 'arity' (a natural number), i.e. the number of 'arguments' it is supposed to have.

iii) A special binary operator, called *application* (Ap).

Definition 2.2 The set $T(\mathcal{F},\mathcal{X})$ of *terms* (or *expressions*) is defined inductively by:

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

ii) If $\mathsf{F} \in \mathcal{F} \cup \{\mathsf{Ap}\}$ is an n-ary symbol ($n \geq 0$), and $t_1, \ldots, t_n \in T(\mathcal{F},\mathcal{X})$, then $\mathsf{F}(t_1, \ldots, t_n) \in T(\mathcal{F},\mathcal{X})$. The t_i ($1 \leq i \leq n$) are the *arguments* of the last term.

Definition 2.3 A *replacement* R is a map from $T(\mathcal{F},\mathcal{X})$ to $T(\mathcal{F},\mathcal{X})$ satisfying

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

So, R is determined by its restriction to the set of variables; we will write t^{R} instead of $\mathrm{R}(t)$.

Definition 2.4 *i*) A *rewrite rule* $\mathbf{r}$ is a pair (l,r) of terms in $T(\mathcal{F},\mathcal{X})$, also written as $\mathbf{r} : l \to r$. Three conditions will be imposed:

a) l is not a variable.

b) The variables occurring in r are contained in l.

c) If Ap occurs l, then $\mathbf{r}$ is of the shape:

$$\mathsf{Ap}(\mathsf{F}_{i-1}(\mathsf{x}_1, \ldots, \mathsf{x}_{i-1}), \mathsf{x}_i) \quad \to \quad \mathsf{F}_i(\mathsf{x}_1, \ldots, \mathsf{x}_i)$$

For every (unindexed) $\mathsf{F} \in \mathcal{F} \cup \{\mathsf{Ap}\}$ with arity n there are n additional rewrite rules:

$$\begin{aligned}
&\mathsf{Ap}(\mathsf{F}_{n-1}(\mathsf{x}_1, \ldots, \mathsf{x}_{n-1}), \mathsf{x}_n) \;\to\; \mathsf{F}(\mathsf{x}_1, \ldots, \mathsf{x}_n)\\
&\qquad\vdots\\
&\mathsf{Ap}(\mathsf{F}_1(\mathsf{x}_1),\mathsf{x}_2) \;\to\; \mathsf{F}_2(\mathsf{x}_1,\mathsf{x}_2)\\
&\mathsf{Ap}(\mathsf{F}_0, \mathsf{x}_1) \;\to\; \mathsf{F}_1(\mathsf{x}_1)
\end{aligned}$$

The function symbols $\mathsf{F}_n, \ldots, \mathsf{F}_1, \mathsf{F}_0$, are the *Curryfied versions of* F.

ii) A rewrite rule $\mathbf{r} : l \to r$ determines a set of *rewrites* $l^{\mathrm{R}} \to r^{\mathrm{R}}$ for all replacements R. The left-hand side l^{R} is called a *redex*; it may be replaced by its '*contractum*' r^{R} inside a context C[ ]; this gives rise to *rewrite steps*:

$$\mathrm{C}[\, l^{\mathrm{R}} \,] \to_{\mathbf{r}} \mathrm{C}[\, r^{\mathrm{R}} \,].$$

iii) $\to_{\mathbf{r}}$ is called the *one-step rewrite relation* generated by $\mathbf{r}$. Concatenating rewrite steps (possibly infinite) *rewrite sequences* $t_0 \to t_1 \to t_2 \to \cdots$ are obtained.

Because of the added rules for $\mathsf{F}_0, \ldots, \mathsf{F}_n$, the rewrite systems considered in this paper are called *Curry-closed*. When presenting a rewrite system, however, only the rules that are essential are shown, not the rules that define the Curryfied versions.

Definition 2.5 A *Curryfied Term Rewriting System* (*Cl*TRS) is a pair $(\Sigma, \mathbf{R})$ of an alphabet Σ and a set $\mathbf{R}$ of rewrite rules.

In a rewrite rule a certain symbol is defined; it is this symbol to which the structure of the rule gives a type.

Definition 2.6 In a rewrite rule $\mathbf{r}$, the leftmost, outermost symbol in the left-hand side that is not an Ap, is called *the defined symbol* of $\mathbf{r}$. Then $\mathbf{r}$ *defines* F, and F is *a defined symbol*. $\mathsf{Q} \in \mathcal{F}$ is called a *constant symbol*, if there is no rewrite rule that defines Q.

When the dependency-graph of the defined function-symbols of a *Cl*TRS is drawn (i.e. a graph is constructed whose nodes are labeled by the defined symbols of the rewrite rules, with a directed edge going from F to G if G occurs in the right-hand side of one of the rules that define F) then in that graph cycles can occur, like for the rewrite system

$$\begin{aligned}
&\mathsf{F}(\mathsf{x}) \to \mathsf{G}(\mathsf{x})\\
&\mathsf{G}(\mathsf{x}) \to \mathsf{F}(\mathsf{x})
\end{aligned}$$

A defined symbol F is called a *recursive symbol* if it occurs on a cycle in the dependency-graph, and every rewrite rule that defines F is called *recursive*. All function-symbols that occur on one cycle in the dependency-graph depend on each other and are, therefore, defined *simultaneously*. This in fact forces to give a different notion of defined symbol; the two rewrite rules above are called *mutually recursive*, and *both* define the symbols F *and* G. To avoid this

problem, rules are assumed to be *not* mutually recursive.

Notice that the definition of recursive symbols, using the notion of defined symbols, is different from the one normally considered. Since Ap is never a defined symbol, the following rewrite system

$$\begin{aligned} \mathsf{D}(\mathsf{x}) &\to \mathsf{Ap}(\mathsf{x}, \mathsf{x}) \\ \mathsf{Ap}(\mathsf{D}_0, \mathsf{x}) &\to \mathsf{D}(\mathsf{x}) \end{aligned}$$

is *not* considered a recursive system. Moreover, the term $\mathsf{D}(\mathsf{D}_0)$ has no normal form (this term plays the role of $(\lambda x.xx)(\lambda x.xx)$ in LC). This means that, in the formalism of this paper, there exist non-recursive first-order rewrite systems that are not normalizing.

Definition 2.7 *Curryfied Combinatory Logic* (CCL) is the *Gi*TRS $(\Sigma, \mathbf{R})$, where $\mathcal{F} = \{\mathsf{S}, \mathsf{S}_2, \mathsf{S}_1, \mathsf{S}_0, \mathsf{K}, \mathsf{K}_1, \mathsf{K}_0, \mathsf{I}, \mathsf{I}_0\}$, and $\mathbf{R}$ contains the rewrite rules

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

and their Curryfied versions. Since CCL is Curry-closed, it is even combinatory complete: every lambda term can be translated into a term in CCL; for details of such a translation, see [10, 21].

Example 2.8 In general, 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

$$\mathsf{M}(\mathsf{S}_2(\mathsf{x}, \mathsf{y})) \to \mathsf{S}_2(\mathsf{I}_0, \mathsf{y})$$

It is also possible that for a certain symbol F, there are more than one rewrite rule that define F, as for example for the rewrite rules:

$$\begin{aligned} \mathsf{F}(\mathsf{x}) &\to \mathsf{x} \\ \mathsf{F}(\mathsf{x}) &\to \mathsf{Ap}(\mathsf{x}, \mathsf{x}) \end{aligned}$$

3 Rank 2 Intersection Type Assignment

The notion of type assignment presented here is defined following the type assignment strategy as used for languages like ML and Miranda. In particular, the way of dealing with function symbols that are defined by more than one rewrite rule as used in Miranda is copied, as well as the way of dealing with untyped recursive definitions. (This is a slightly more liberal way of dealing with recursion than used for ML. In [31, 26] another extension of the way of dealing with recursion in the ML-system is presented, in which type assignment is no longer decidable, but that is nevertheless used for type checking in Miranda. This system was used for the notion of type assignment defined in [6], but will not be used here.)

Compared to the notion of type assignment used in OBJ, the system here is an extension by allowing for higher-order types as well as polymorphism.

3.1 Operations on pairs

In this subsection, three operations on pairs of basis and type are defined, namely substitution, copying, and weakening. In Theorem 3.8 it will be proved that these operations are sound:

they return admissible pairs for a term when applied to an admissible pair for that term (see Definition 3.7 (*ii*), and in Theorem 4.12 that they are complete: they are sufficient to generate all admissible pairs for a term from its principal pair.

In this paper, substitution is defined as the operation that replaces type-variables by elements of $\mathcal{T}_C$. Although perhaps this is a more restricted kind of substitution than could be expected, it is a sound operation and will be proven to be sufficient.

Definition 3.1 The *substitution* $(\varphi \mapsto \alpha) : \mathcal{T}_R \to \mathcal{T}_R$, where φ is a type-variable and $\alpha \in \mathcal{T}_C$, is defined by:

$$\begin{array}{ll}(\varphi \mapsto \alpha)(\varphi) & = \alpha \\ (\varphi \mapsto \alpha)(\varphi') & = \varphi', \text{ if } \varphi \neq \varphi' \\ (\varphi \mapsto \alpha)(\sigma \to \tau) & = (\varphi \mapsto \alpha)(\sigma) \to (\varphi \mapsto \alpha)(\tau) \\ (\varphi \mapsto \alpha)(\sigma_1 \cap \cdots \cap \sigma_n) & = (\varphi \mapsto \alpha)(\sigma_1) \cap \cdots \cap (\varphi \mapsto \alpha)(\sigma_n).\end{array}$$

If S_1 and S_2 are substitutions, then so is $S_1 \circ S_2$, where $S_1 \circ S_2(\sigma) = S_1(S_2(\sigma))$. Substitutions are extended to bases by $S(B) = \{x{:}S(\sigma) \mid x{:}\sigma \in B\}$, and $S(\langle B, \sigma \rangle) = \langle S(B), S(\sigma) \rangle$.

Substitution is normally defined as the operation that replaces type-variables by types, without restriction. In general, this definition would not be correct for the Rank 2 system, since, for example, the replacement of the type-variable φ in $\varphi \to \varphi$ by the type $(\sigma \to \tau) \cap \sigma \to \tau$ would give a type that is not an element of $\mathcal{T}_R$.

The next operation on pairs, copying, can be seen as a very simple version of the various operations of expansion as defined in [17, 36, 3]. For readers familiar with those definitions of expansion: copying is a total expansion, that is not 'computed': all type-variables occurring in basis and type are copied. It is an operation on types that deals with the replacement of a type by an intersection of a number of copies of that type.

Definition 3.2 Let B be a basis, $\sigma \in \mathcal{T}_R$, and $n \geq 1$. The triple $<n, B, \sigma>$ determines a *copying* $C_{<n,B,\sigma>} : \mathcal{T}_R \to \mathcal{T}_R$, that is constructed as follows: Suppose $V = \{\varphi_1, \ldots, \varphi_m\}$ is the set of all type-variables occurring in $\langle B, \sigma \rangle$. 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 $(1 \leq i \leq n,\ 1 \leq j \leq m)$ does not occur in V. Let S_i be the substitution that replaces every φ_j by φ_j^i. Then

$$C_{<n,B,\sigma>}(\tau) = S_1(\tau) \cap \cdots \cap S_n(\tau).$$

Copying is extended to bases and pairs by: $C_{<n,B,\sigma>}(B') = \{x{:}C_{<n,B,\sigma>}(\rho) \mid x{:}\rho \in B'\}$, and $C_{<n,B,\sigma>}(\langle B', \sigma' \rangle) = \langle C_{<n,B,\sigma>}(B'), C_{<n,B,\sigma>}(\sigma') \rangle$.
To simplify notation, $<n, B, \sigma>$ will be written instead of $C_{<n,B,\sigma>}$.

Notice that if τ does not contain type-variables that occur in V, then $<n, B, \sigma>(\tau) = \tau \cap \cdots \cap \tau$, which is by definition of $\sim$ the same as τ.

The last operation is that of weakening; it replaces a basis by a more informative one.

Definition 3.3 A *weakening* W is an operation characterized by a pair of bases $\langle B_0, B_1 \rangle$ such that $B_1 \leq B_0$, and is defined by: if $B = B_0$, then $W(\langle B, \sigma \rangle) = \langle B_1, \sigma \rangle$, and $W(\langle B, \sigma \rangle) = \langle B, \sigma \rangle$, otherwise.

Definition 3.4 *i*) A *transformation sequence* is an object $<O_1,\ldots,O_n>$, where each O_i is an operation of substitution, copying, or weakening, and

$$<O_1,\ldots,O_n>(\langle B,\sigma\rangle) \;=\; O_n(\cdots(O_1(\langle B,\sigma\rangle))\cdots).$$

ii) On transformation sequences the operation of concatenation is denoted by $*$, and:

$$<O_1,\ldots,O_i> * <O_{i+1},\ldots,O_n> \;=\; <O_1,\ldots,O_n>.$$

iii) A *type-chain* is a transformation sequence $<O_1,\ldots,O_n>$ of operations of substitution and copying only, and is extended to types by:

$$<O_1,\ldots,O_n>(\sigma) \;=\; O_n(\cdots(O_1(\sigma))\cdots).$$

iv) A *chain* is a type-chain concatenated with one operation of weakening.

v) We say that $Ch_1 = Ch_2$, if for all σ, $Ch_1(\sigma) = Ch_2(\sigma)$.

For type-chains, the following properties hold:

Lemma 3.5 Let Ch be a type-chain.

i) There are a copying C and substitutions $S_1, \ldots, S_n$ such that $Ch = <C,S_1,\ldots,S_n>$.

ii) If $\sigma \in \mathcal{T}_2$, and $Ch(\sigma) \in \mathcal{T}_2$, then there is a substitution S such that $Ch(\sigma) = S(\sigma)$. Without loss of generality, there is also a type-chain Ch′ such that $Ch = <S> * Ch'$.

iii) If $\sigma \in \mathcal{T}_2$, and $Ch(\sigma) \in \mathcal{T}_R$, then there are $\sigma_1\ldots,\sigma_n$ and substitutions $S_1, \ldots, S_n$ such that $Ch(\sigma) = \sigma_1\cap\cdots\cap\sigma_n$, and, for every $1 \le i \le n$, $S_i(\sigma) = \sigma_i$.

Proof: Easy, using part (*i* in part (*iii*. □

3.2 Rank 2 type assignment in *Gi*TRS

The type assignment system presented in this paper is a partial system in the sense that not only will be defined how terms and rewrite rules can be typed, but it is also assumed that every function symbol already has a type, stored in an environment, of which the structure is usually motivated by a rewrite rule. In fact, this approach is very close to the one taken in [24], where the principal Curry-type scheme of an object in Combinatory Logic is defined.

Definition 3.6 Let $(\Sigma,\mathbf{R})$ be a *Gi*TRS.

i) A mapping $\mathcal{E} : \mathcal{F} \to \mathcal{T}_2$ is called an *environment* if, for every $\mathsf{F} \in \mathcal{F}$ with arity n, $\mathcal{E}(\mathsf{F}) = \mathcal{E}(\mathsf{F}_{n-1}) = \cdots = \mathcal{E}(\mathsf{F}_0)$.

ii) For $\mathsf{F} \in \mathcal{F}$, $\sigma \in \mathcal{T}_2$, and $\mathcal{E}$ an environment, the environment $\mathcal{E}[\mathsf{F}:=\sigma]$ is defined by:

$\mathcal{E}[\mathsf{F}:=\sigma](\mathsf{G}) = \sigma$, if $\mathsf{G} \in \{\mathsf{F}, \mathsf{F}_{n-1}, \ldots, \mathsf{F}_0\}$

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

Type assignment on *Gi*TRS is defined in two stages. In the next definition type assignment on terms is defined, and in Definition 5.1 type assignment on term rewrite rules will be defined.

Definition 3.7 Let $(\Sigma,\mathbf{R})$ be a *Gi*TRS, and $\mathcal{E}$ an environment.

i) *Type assignment* and *derivations* are defined by the following natural deduction system.

$$(\leq):\ \frac{\mathsf{x}{:}\sigma \quad \sigma \leq \tau}{\mathsf{x}{:}\tau}\ (\sigma \in \mathcal{T}_1,\ \tau \in \mathcal{T}_C) \qquad (\cap\mathrm{I}):\ \frac{t{:}\sigma_1 \quad \ldots \quad t{:}\sigma_n}{t{:}\sigma_1 \cap \cdots \cap \sigma_n}\ (a)$$

$$(\mathsf{Ap}):\ \frac{t_1{:}\sigma \to \tau \qquad t_2{:}\sigma}{\mathsf{Ap}(t_1,t_2){:}\tau}\ (\tau \in \mathcal{T}_2,\ \sigma \in \mathcal{T}_1) \qquad (\mathcal{F}):\ \frac{t_1{:}\sigma_1 \quad \ldots \quad t_n{:}\sigma_n}{\mathsf{F}(t_1,\ldots,t_n){:}\sigma}\ (b)$$

(*a*) If $n \geq 1$, and for every $1 \leq i \leq n$, $\sigma_i \in \mathcal{T}_2$.

(*b*) if there exists a type-chain *Ch* such that $Ch(\mathcal{E}(F)) = \sigma_1 \to \cdots \to \sigma_n \to \sigma$, and for every $0 \leq i \leq n$, $\sigma_i \in \mathcal{T}_1$.

ii) Let $t \in T(\mathcal{F},\mathcal{X})$ be typeable by σ with respect to $\mathcal{E}$. The notation $B \vdash_{\mathcal{E}} t{:}\sigma$ is used to express that B is a basis that contains at least all the statements with variables as subject that occur in the derivation for $t{:}\sigma$. Then $\langle B,\sigma\rangle$ is called *an admissible pair for* t.

An environment does not provide a type for Ap; instead in rule (Ap) it is defined how an application should be typed; this is because although $\sigma \to \tau$ and $\sigma \in \mathcal{T}_1$, not necessarily $(\sigma \to \tau) \to \sigma \to \tau \in \mathcal{T}_2$.

The use of an environment corresponds to the use of 'axiom-schemes', and the use of a chain in rule ($\mathcal{F}$) to the use of 'axioms' as in [24], and corresponds to the use of a 'combinator basis' and the axioms in Definition 3.2 of [21]. The combination of those two definitions also introduces a notion of polymorphism into the type assignment system of this paper. The environment returns the 'principal type' for a function symbol; this symbol can be used with types that are 'instances' of its principal type.

The following theorem shows the operations are sound on derivations; in Theorem 5.4, we will prove a soundness result for rewrite rules.

Theorem 3.8 *i) Let S be a substitution. If* $B \vdash_{\mathcal{E}} t{:}\sigma$, *then* $S(B) \vdash_{\mathcal{E}} t{:}S(\sigma)$.

ii) Let C be a copying such that $C(\langle B,\sigma\rangle) = \langle B',\sigma'\rangle$. *If* $B \vdash_{\mathcal{E}} t{:}\sigma$, *then* $B' \vdash_{\mathcal{E}} t{:}\sigma'$.

iii) For every $t \in T(\mathcal{F},\mathcal{X})$: *if* $B \vdash_{\mathcal{E}} t{:}\sigma$, *then, for every weakening W: if* $W(\langle B,\sigma\rangle) = \langle B',\sigma'\rangle$, *then* $B' \vdash_{\mathcal{E}} t{:}\sigma'$.

Proof: Part (*i* follows by straightforward induction, part (*ii* follows by Definition 3.2, part (*i* and rule ($\cap$I), and part (*iii* follows by an easy induction. □

4 Completeness of operations on pairs

In this section, the principal type property will be shown to hold for the here presented type assignment system: for every term t typeable with respect to $\mathcal{E}$, there exists a pair $PP_{\mathcal{E}}(t) = \langle P,\pi\rangle$, the principal pair of t with respect to $\mathcal{E}$, such that $P \vdash_{\mathcal{E}} t{:}\pi$, and, for every pair $\langle B,\sigma\rangle$ such that $B \vdash_{\mathcal{E}} t{:}\sigma$, there exists a chain of operations *Ch* such that $Ch(\langle P,\pi\rangle) = \langle B,\sigma\rangle$.

As in [24], principal types are defined using a notion of unification.

4.1 Unification of intersection types of Rank 2

In the context of types, unification is a procedure normally used to find a common instance for demanded and provided type for applications, i.e: if t_1 has type $\sigma \to \tau$, and t_2 has type α, then unification looks for a common instance of the types σ and α such that $\mathsf{Ap}(t_1,t_2)$ can be typed properly. The unification algorithm $unify_{R2}$ presented in the next definition deals with

just that problem. This means that it is not a full unification algorithm for intersection types of Rank 2, but only an algorithm that finds the most general unifying chain for demanded and provided type. It is defined using Robinson's well-known unification algorithm *unify*.

Definition 4.1 (Robinson's Unification Algorithm [35]) Let $\mathcal{S}$ be the set of all substitutions.

$unify : \mathcal{T}_C \times \mathcal{T}_C \to \mathcal{S}$

$unify(\varphi, \varphi') = (\varphi \mapsto \varphi')$

$unify(\varphi, \tau) = (\varphi \mapsto \tau)$, if φ does not occur in τ and τ is not a type-variable

$unify(\sigma, \varphi) = unify(\varphi, \sigma)$

$unify(\sigma \to \tau, \rho \to \mu) = S_2 \circ S_1$

where $S_1 = unify(\sigma, \rho)$

$S_2 = unify(S_1(\tau), S_1(\mu))$

Property 4.2 ([35]) *unify returns the most general unifier of two Curry-types σ and τ (if it exists), i.e.: For all $\sigma, \tau \in \mathcal{T}_C$, substitutions S: if $S(\sigma) = S(\tau)$, then there are substitutions S_u and S' such that*

$$S_u = unify(\sigma, \tau), \text{ and } S(\sigma) = S' \circ S_u(\sigma) = S' \circ S_u(\tau) = S(\tau). \qquad \square$$

Since the substitution returned by *unify* is defined only on type-variables occurring in σ and τ, it is even possible to show that $S = S' \circ S_u$.

The unification algorithm works roughly as follows: in finding the principal pair for the term $\mathsf{Ap}(t_1,t_2)$, by construction the demanded type σ in $\sigma \to \tau$ is in $\mathcal{T}_1$ and the provided type α is in $\mathcal{T}_2$. The unification algorithm looks for types that can be assigned to the terms t_1 and t_2 such that the application term can be typed properly. In order to be consistent, the result of the unification of σ and α – a chain Ch – should always be such that $Ch(\alpha) \in \mathcal{T}_1$. However, if $\alpha \notin \mathcal{T}_C$, then in general $Ch(\alpha) \notin \mathcal{T}_1$. To overcome this difficulty, an algorithm $to\mathcal{T}_C$ will be inserted that, when applied to the type α, returns a type-chain of operations that removes, if possible, intersections in α.

Definition 4.3 Let $\mathcal{C}$ be the set of all type-chains, and let Id_S be the substitution that replaces all type-variables by themselves.

$to\mathcal{T}_C : \mathcal{T}_2 \to \mathcal{C}$

$to\mathcal{T}_C(\sigma) = <Id_S>$, if $\sigma \in \mathcal{T}_C$

$to\mathcal{T}_C(\sigma_1 \cap \cdots \cap \sigma_n \to \sigma) = <S_1, \ldots, S_{n-1}> * Ch$, otherwise

where $S_i = unify(<S_1, \ldots, S_{i-1}>(\sigma_1), <S_1, \ldots, S_{i-1}>(\sigma_{i+1}))$, for every $1 \le i \le n-1$

$Ch = to\mathcal{T}_C(<S_1, \ldots, S_{n-1}>(\sigma))$

The algorithm $unify_{R2}$ is called with the types σ and α', the latter being α in which the intersections are removed (so $\alpha' = to\mathcal{T}_C(\alpha)(\alpha)$; notice that $to\mathcal{T}_C(\alpha)$ is an operation on types that removes all intersections in α).

It is possible that $\sigma \notin \mathcal{T}_C$, so it can be that α' must be duplicated. Since such an operation affects also the basis, the third argument of $unify_{R2}$ is a basis.

Definition 4.4 (Rank 2 Unification) Let $\mathcal{B}$ be the set of all bases, and $\mathcal{C}$ the set of all type-chains.

$unify_{R2} : \mathcal{T}_1 \times \mathcal{T}_C \times \mathcal{B} \to \mathcal{C}$

$unify_{R2}(\sigma, \alpha, B) = unify(\sigma, \alpha)$, if $\sigma \in \mathcal{T}_C$

$unify_{R2}(\sigma_1\cap\cdots\cap\sigma_n, \alpha, B) = <C,S_1,\ldots,S_n>$, otherwise
where $C = <n,B,\alpha>$
$\alpha_1\cap\cdots\cap\alpha_n = C(\alpha)$
$S_i = unify(<S_1,\ldots,S_{i-1}>(\sigma_i), \alpha_i)$, for every $1\leq i\leq n$

Notice that $unify_{R2}$ and $to\mathcal{T}_C$ only fail when *unify* fails, and that $<n,B,\alpha>$ never fails. Because of this relation between $unify_{R2}$ and $to\mathcal{T}_C$ on one side, and *unify* on the other, the procedures defined here are terminating and type assingment in the system defined in this paper is decidable.

With Property 4.2, it is possible to prove the following lemma.

Lemma 4.5 i) For every $\sigma\in\mathcal{T}_2$, type-chain Ch: if $Ch(\sigma)=\tau\in\mathcal{T}_1$, then there is a type-chain Ch′ such that $to\mathcal{T}_C(\sigma)*Ch'(\sigma)=\tau$. (Without loss of generality, $Ch = to\mathcal{T}_C(\sigma)*Ch'$.)

ii) For every $\sigma\in\mathcal{T}_1$, $\alpha\in\mathcal{T}_C$ that are disjoint: if there exists a type-chain Ch such that $Ch(\sigma) = Ch(\alpha)$, then, for every basis B that shares no type-variables with σ, there are type-chains Ch′ and Ch_u such that $Ch_u = unify_{R2}(\sigma,\alpha,B)$ and $Ch(\sigma) = Ch_u * Ch'(\sigma) = Ch_u * Ch'(\alpha) = Ch(\alpha)$. (Without loss of generality, $Ch = Ch_u * Ch'$.)

Proof: i) By 3.5 (*iii* there are substitutions $S_1,\ldots,S_n$ such that $Ch(\sigma) = S_1(\sigma)\cap\cdots\cap S_n(\sigma)$. Let $\sigma = \alpha_1\cap\cdots\cap\alpha_m\to\beta$. Since, for every $1\leq i\leq n$, $S_i(\sigma)\in\mathcal{T}_C$, also for $1\leq i\leq n$, $1\leq j\neq k\leq m$, $S_i(\alpha_j) = S_i(\alpha_k)$. The result follows from Property 4.2 and Definition 4.3.

ii) If $\sigma\in\mathcal{T}_C$, then it is easy to show that σ and α must have a common substitution-instance, so from Property 4.2 the result follows. If $\sigma=\sigma_1\cap\cdots\cap\sigma_n$, then likewise it is easy (but laborious) to show that, for every $1\leq i\leq n$, σ_i and α have a substitution-instance in common. Then by induction on Definition 4.4, using Property 4.2, the result follows. □

4.2 Principal pairs for terms

In this subsection, the principal pair for a term t with respect to $\mathcal{E}$ – $PP_{\mathcal{E}}(t)$ – is defined, consisting of basis P and type π. In Theorem 4.13 it will be shown that, for every term, this is indeed the principal one.

Definition 4.6 For every $t\in T(\mathcal{F},\mathcal{X})$, using $unify_{R2}$, $PP_{\mathcal{E}}(t) = \langle P,\pi\rangle$ is defined by:

i) $t\equiv \mathsf{x}$. Then $\langle P,\pi\rangle = \langle\{\mathsf{x}{:}\varphi\},\varphi\rangle$.

ii) $t\equiv \mathsf{Ap}(t_1,t_2)$. Let $PP_{\mathcal{E}}(t_1) = \langle P_1,\pi_1\rangle$, $PP_{\mathcal{E}}(t_2) = \langle P_2,\pi_2\rangle$, (choose, if necessary, trivial variants such that these pairs are disjoint), and $S_2 = to\mathcal{T}_C(\pi_2)$, then:

a) If $\pi_1=\varphi$, then:
$PP_{\mathcal{E}}(\mathsf{Ap}(t_1,t_2)) = \langle S_2,S_1\rangle(\langle\Pi\{P_1,P_2\},\varphi'\rangle)$
where $S_1 = unify(\varphi, S_2(\pi_2)\to\varphi')$,
and φ' is a type-variable not occurring in any other type.

b) If $\pi_1=\sigma\to\tau$, then:
$PP_{\mathcal{E}}(\mathsf{Ap}(t_1,t_2)) = \langle S_2\rangle * Ch(\langle\Pi\{P_1,P_2\},\tau\rangle)$
where $Ch = unify_{R2}(\sigma, S_2(\pi_2), S_2(P_2))$.

iii) $t\equiv \mathsf{F}(t_1,\ldots,t_n)$. If $\mathcal{E}(\mathsf{F}) = \gamma_1\to\cdots\to\gamma_n\to\gamma$, and, for every $1\leq i\leq n$, $PP_{\mathcal{E}}(t_i) = \langle P_i,\pi_i\rangle$, (choose, if necessary, trivial variants such that the $\langle P_i,\pi_i\rangle$ are disjoint in pairs and these pairs share no type-variables with $\gamma_1\to\cdots\to\gamma_n\to\gamma$), then:
$PP_{\mathcal{E}}(\mathsf{F}(t_1,\ldots,t_n)) = Ch(\langle\Pi\{P_1,\ldots,P_n\},\gamma\rangle)$
where $Ch = <S_1,\ldots,S_n> * Ch_1 * \cdots * Ch_n$

$$S_i = to\mathcal{T}_C(\pi_i)$$
$$Ch_i = unify_{R2}(Ch_1 * \cdots * Ch_{i-1}(\gamma_i), S_i(\pi_i), S_i(P_i)).$$

Note that, since $unify_{R2}$ may fail, not every term has a principal pair.

Example 4.7 The typed rules for F as in Example 5.2 seem perhaps somewhat ad hoc, but using the environment: $\mathcal{E}(\mathsf{K}) = 1{\to}2{\to}1$, $\mathcal{E}(\mathsf{Z}) = 3{\to}4{\to}4$, $\mathcal{E}(\mathsf{I}) = 5{\to}5$, and $\mathcal{E}(\mathsf{F}) = 7{\cap}(6{\to}7){\cap}6{\to}7$, where Z is defined by $\mathsf{Z}(x, y) \to y$, and using Definition 4.6, the following can easily be checked:

i) $\vdash_{\mathcal{E}} \mathsf{F}(\mathsf{I}_0){:}8{\to}8$, $\vdash_{\mathcal{E}} \mathsf{I}{:}8{\to}8$, and $\vdash_{\mathcal{E}} \mathsf{I}(\mathsf{I}_0){:}8{\to}8$.

ii) $\vdash_{\mathcal{E}} \mathsf{F}(\mathsf{Z}_0){:}(8{\to}8){\to}8{\to}8$, $\vdash_{\mathcal{E}} \mathsf{Z}_0{:}(8{\to}8){\to}8{\to}8$, and $\vdash_{\mathcal{E}} \mathsf{Z}_1(\mathsf{Z}_0){:}(8{\to}8){\to}8{\to}8$.

iii) $\vdash_{\mathcal{E}} \mathsf{F}(\mathsf{K}_0){:}(8{\to}9){\to}9{\to}8{\to}9$, $\vdash_{\mathcal{E}} \mathsf{K}_0{:}(8{\to}9){\to}9{\to}8{\to}9$, and $\vdash_{\mathcal{E}} \mathsf{K}_1(\mathsf{K}_0){:}(8{\to}9){\to}9{\to}8{\to}9$.

The given types are the principal types for respectively $\mathsf{F}(\mathsf{I}_0)$, $\mathsf{F}(\mathsf{Z}_0)$, and $\mathsf{F}(\mathsf{K}_0)$.

Example 4.8 Using Rank 2 intersection types, the term $\mathsf{S}(\mathsf{K}_0,\mathsf{S}_0,\mathsf{I}_0)$ has a more general principal type than using Curry-types. With the environment

$$\mathcal{E}(\mathsf{S}) = (1{\to}2{\to}3){\to}(4{\to}2){\to}(1{\cap}4){\to}3$$
$$\mathcal{E}(\mathsf{K}) = 5{\to}6{\to}5$$
$$\mathcal{E}(\mathsf{I}) = 7{\to}7,$$

and Definition 4.6, the following can easily be checked: $\vdash_{\mathcal{E}} \mathsf{S}(\mathsf{K}_0,\mathsf{S}_0,\mathsf{I}_0){:}8{\to}8$.

$$\frac{\mathsf{K}_0{:}(8{\to}8){\to}(((9{\to}10){\to}9){\to}(9{\to}10){\to}10){\to}8{\to}8 \qquad \mathsf{S}_0{:}((9{\to}10){\to}9{\to}10){\to}((9{\to}10){\to}9){\to}(9{\to}10){\to}10 \quad \mathsf{I}_0{:}(8{\to}8){\cap}((9{\to}10){\to}9{\to}10)}{\mathsf{S}(\mathsf{K}_0,\mathsf{S}_0,\mathsf{I}_0){:}8{\to}8}$$

Notice that in Curry's system – and in ML – the term $SKSI$ has the principal type $(9{\to}10){\to}9{\to}10$.

With D defined by $\mathsf{D}(x) \to \mathsf{Ap}(x,x)$, it is even possible to check that for example $\mathsf{D}(\mathsf{S}(\mathsf{K}_0,\mathsf{S}_0,\mathsf{I}_0))$ and $\mathsf{D}(\mathsf{I}_0)$ are typeable by $11{\to}11$. Notice that the term $\mathsf{I}(\mathsf{D}_0)$ is not typeable.

The following lemma expresses that a principal pair for the term t is an admissible pair for t.

Lemma 4.9 *If $PP_{\mathcal{E}}(t) = \langle P, \pi\rangle$, then $P \vdash_{\mathcal{E}} t{:}\pi$, and $\pi \in \mathcal{T}_2$.*

Proof: By induction on the definition of $PP_{\mathcal{E}}(t)$, using Theorem 3.8. □

The following lemmas are needed in the proofs of Theorem 4.12 and Lemma 5.5 (*iii*. The first states that if a type-chain maps the principal pairs of terms in an application to pairs that allows the application itself to be typed, then these pairs can also be obtained by first performing a unification. The second generalizes this result to arbitrary function applications.

Lemma 4.10 *Let $\sigma \in \mathcal{T}_2$, and for $i = 1,2$: $PP_{\mathcal{E}}(t_i) = \langle P_i, \pi_i\rangle$, such that these pairs are disjoint, and let Ch be a type-chain such that*

$$Ch(PP_{\mathcal{E}}(t_1)) = \langle B_1, \tau{\to}\sigma\rangle, \text{ and } Ch(PP_{\mathcal{E}}(t_2)) = \langle B_2, \tau\rangle.$$

Then there are type-chains Ch_g and Ch', and type $\alpha \in \mathcal{T}_2$ such that

$$PP_{\mathcal{E}}(\mathsf{Ap}(t_1,t_2)) = Ch_g(\langle \Pi\{P_1,P_2\},\alpha\rangle), \text{ and } Ch'(PP_{\mathcal{E}}(\mathsf{Ap}(t_1,t_2))) = \langle \Pi\{B_1,B_2\},\sigma\rangle.$$

Proof: Since $Ch(\pi_2) \in \mathcal{T}_1$, by 4.5 (*i* there is a Ch_1 such that $Ch = <S_2> * Ch_1$, with $S_2 = to\mathcal{T}_C(\pi_2)$.

i) $\pi_1 = \varphi$. Take $S_1 = unify(\varphi, S_2(\pi_2)\rightarrow\varphi')$, where φ' is a type-variable not occurring in any other type. Assume, without loss of generality, that $Ch_1(\varphi') = \sigma$. Then, by Definition 4.6 (*ii.a*),

$$PP_{\mathcal{E}}(\mathsf{Ap}(t_1,t_2)) = \langle S_2,\ S_1\rangle(\langle \Pi\{P_1,P_2\},\varphi'\rangle).$$

Since $\varphi \in \mathcal{T}_C$ and $\tau\rightarrow\sigma \in \mathcal{T}_2$, also $\tau\rightarrow\sigma \in \mathcal{T}_1$, so $\tau\rightarrow\sigma \in \mathcal{T}_C$ and $\tau \in \mathcal{T}_C$. So $Ch_1(S_2(\pi_2)\rightarrow\varphi') \in \mathcal{T}_C$, and, by Lemma 3.5 (*ii*, there are a substitution S_3 and a type-chain Ch_2 such that $S_3(S_2(\pi_2)\rightarrow\varphi') = \tau\rightarrow\sigma$, and $Ch_1 = <S_3> * Ch_2$. Assume, without loss of generality, that $S_3(\varphi) = \tau\rightarrow\sigma$. By Property 4.2, there is a substitution S_4 such that $S_3 = S_4 \circ S_1$. So

$$Ch = <S_2> * Ch_1 = <S_2> * <S_3> * Ch_2 = <S_2,\ S_1> * <S_4> * Ch_2.$$

Take $Ch_g = <S_2,\ S_1>$, $Ch' = <S_4> * Ch_2$ and $\alpha = \varphi'$.

ii) $\pi_1 = \rho\rightarrow\mu$. Since the pairs $\langle P_1,\rho\rightarrow\mu\rangle$ and $\langle P_2,\pi_2\rangle$ are disjoint, $Ch_1(\rho\rightarrow\mu) = \tau\rightarrow\sigma$. Since $Ch_1(\rho) = Ch_1(S_2(\pi_2))$, by Lemma 4.5 (*ii*, there are type-chains Ch_u and Ch_2 such that

$$Ch_u = unify_{R2}(\rho, S_2(\pi_2), S_2(P_2)), \text{ and } Ch_1 = Ch_u * Ch_2.$$

By Definition 4.6 (*ii.b*), $PP_{\mathcal{E}}(\mathsf{Ap}(t_1,t_2)) = \langle S_2\rangle * Ch_u(\langle \Pi\{P_1,P_2\},\mu\rangle)$. Then

$$Ch = <S_2> * Ch_1 = <S_2> * Ch_u * Ch_2.$$

Take $Ch_g = <S_2> * Ch_u$, $Ch' = Ch_2$, and $\alpha = \mu$. □

Lemma 4.11 Let $\sigma \in \mathcal{T}_2$, and, for every $1 \le i \le n$, $PP_{\mathcal{E}}(t_i) = \langle P_i,\pi_i\rangle$, such that the pairs $\langle P_i,\pi_i\rangle$ and the type $\mathcal{E}(\mathsf{F}) = \gamma_1\rightarrow\cdots\rightarrow\gamma_n\rightarrow\gamma$ are disjoint, and let Ch be a type-chain such that

$$Ch(\mathcal{E}(\mathsf{F})) = \sigma_1\rightarrow\cdots\rightarrow\sigma_n\rightarrow\sigma \text{ and, for every } 1 \le i \le n,\ Ch(\langle P_i,\pi_i\rangle) = \langle B_i,\sigma_i\rangle.$$

Then there are type-chains Ch_g and Ch_p such that

$$PP_{\mathcal{E}}(\mathsf{F}(t_1,\ldots,t_n)) = Ch_g(\langle \Pi\{P_1,\ldots,P_n\},\gamma\rangle), \text{ and}$$
$$Ch_p(PP_{\mathcal{E}}(\mathsf{F}(t_1,\ldots,t_n))) = \langle \Pi\{B_1,\ldots,B_n\},\sigma\rangle.$$

Proof: As part (*ii* of the proof for the previous lemma, constructing Ch_p by induction on n. □

In order to prove that the operations are complete, we prove that if $B \vdash_{\mathcal{E}} t{:}\sigma$, then t has a principal pair and there is a chain that maps this principal pair to $\langle B,\sigma\rangle$.

Theorem 4.12 *If $B \vdash_{\mathcal{E}} t{:}\sigma$, then there are a basis P, type π and a chain Ch such that $PP_{\mathcal{E}}(t) = \langle P,\pi\rangle$, and $Ch(\langle P,\pi\rangle) = \langle B,\sigma\rangle$.*

Proof: By induction on the structure of derivations.

(≤): Then $t \equiv \mathsf{x}$, $B \le \{\mathsf{x}{:}\sigma\}$, $\sigma \in \mathcal{T}_C$, and $PP_{\mathcal{E}}(\mathsf{x}) = \langle\{\mathsf{x}{:}\varphi\},\varphi\rangle$.
Take $Ch = <\varphi\mapsto\sigma, \langle\{\mathsf{x}{:}\sigma\},B\rangle>$.

(Ap): Then $t \equiv \mathsf{Ap}(t_1,t_2)$, and there are $\tau \in \mathcal{T}_1$ and bases B_1, B_2 such that $B_1 \vdash_{\mathcal{E}} t_1{:}\tau\rightarrow\sigma$, and $B_2 \vdash_{\mathcal{E}} t_2{:}\tau$. By induction for $i = 1,2$ there are P_i, π_i, and chain Ch_i such that

$$PP_{\mathcal{E}}(t_i) = \langle P_i,\pi_i\rangle,\ Ch_1(PP_{\mathcal{E}}(t_1)) = \langle B,\tau\rightarrow\sigma\rangle, \text{ and } Ch_2(PP_{\mathcal{E}}(t_2)) = \langle B,\tau\rangle.$$

Let $Ch_i = Ch_i' * W_i$, where Ch_i' is a type-chain, and $B_i \le B$ such that $W_i = \langle B_i,B\rangle$. Since the pairs $\langle P_i,\pi_i\rangle$ are disjoint, the type-chains Ch_i' do not interfere, so

$$Ch_1' * Ch_2'(PP_{\mathcal{E}}(t_1)) = \langle B_1,\tau\rightarrow\sigma\rangle, \text{ and } Ch_1' * Ch_2'(PP_{\mathcal{E}}(t_2)) = \langle B_2,\tau\rangle.$$

Then, by Lemma 4.10, there is a Ch' such that $Ch'(PP_{\mathcal{E}}(\mathsf{Ap}(t_1,t_2))) = \langle \Pi\{B_1,B_2\},\sigma\rangle$.
Take $Ch = Ch' * <\langle \Pi\{B_1,B_2\},B\rangle>$.

($\mathcal{F}$): Then $t \equiv \mathsf{F}(t_1,\ldots,t_n)$; let $\mathcal{E}(\mathsf{F}) = \gamma_1 \to \cdots \to \gamma_n \to \gamma$. There are $\sigma_1 \ldots, \sigma_n$ such that, for every $1 \leq i \leq n$, $B \vdash_{\mathcal{E}} t_i{:}\sigma_i$, and a type-chain Ch_F such that

$$Ch_F(\gamma_1 \to \cdots \to \gamma_n \to \gamma) = \sigma_1 \to \cdots \to \sigma_n \to \sigma.$$

By induction, for $1 \leq i \leq n$, there are $\langle P_i, \pi_i \rangle$, (disjoint in pairs) and chain Ch_i, such that

$$PP_{\mathcal{E}}(t_i) = \langle P_i, \pi_i \rangle, \text{ and } Ch_i(PP_{\mathcal{E}}(t_i)) = \langle B, \sigma_i \rangle.$$

Let $Ch_i = Ch_i' * W_i$, where Ch_i' is a type-chain, and $B_i \leq B$ such that $W_i = \langle B_i, B \rangle$. Since the pairs $\langle P_i, \pi_i \rangle$ are disjoint, the chains Ch_i' do not interfere. Assume, without loss of generality, that none of the type-variables occurring in $\gamma_1 \to \cdots \to \gamma_n \to \gamma$ occur in any of the pairs $\langle P_i, \pi_i \rangle$. Let $Ch' = Ch_F * Ch_1' * \cdots * Ch_n'$. Since, for every $1 \leq i \leq n$, $Ch'(\langle P_i, \pi_i \rangle) = \langle B_i, \sigma_i \rangle$, and $Ch'(\gamma_1 \to \cdots \to \gamma_n \to \gamma) = \sigma_1 \to \cdots \to \sigma_n \to \sigma$, by Lemma 4.11 there is a type-chain Ch'' such that $Ch''(PP_{\mathcal{E}}(\mathsf{F}(t_1,\ldots,t_n))) = \langle \Pi\{B_1,\ldots,B_n\}, \sigma \rangle$.

Take $Ch = Ch'' * <\langle \Pi\{B_1,\ldots,B_n\}, B \rangle>$.

($\cap I$): Then $\sigma = \sigma_1 \cap \cdots \cap \sigma_n$, and, for every $1 \leq i \leq n$, $B \vdash_{\mathcal{E}} t{:}\sigma_i$.

By induction there are P, π, such that $PP_{\mathcal{E}}(t) = \langle P, \pi \rangle$. Let $C = <n, P, \pi>$, then

$$C(\langle P, \pi \rangle) = \langle \Pi\{P_1,\ldots,P_n\}, \pi_1 \cap \cdots \cap \pi_n \rangle, \text{ with } PP_{\mathcal{E}}(t) = \langle P_i, \pi_i \rangle.$$

By induction there are type-chains $Ch_1, \ldots, Ch_n$ such that

$$\text{for } 1 \leq i \leq n,\ Ch_i(\langle P_i, \pi_i \rangle) = \langle B_i, \sigma_i' \rangle.$$

Let $Ch_i = Ch_i' * W_i$, where Ch_i' is a type-chain, and $B_i' \leq B_i$ such that $W_i = \langle B_i', B_i \rangle$. Without loss of generality, we can assume that every $Ch_i' = Ch_i'' * <S_i>$, such that the Ch_i'' do not interfere.

Take $Ch = <C> * Ch_1'' * \cdots * Ch_n'' * <S_1 \circ \cdots \circ S_n, \langle \Pi\{B_1,\ldots,B_n\}, B \rangle>$. □

Theorem 4.13 (Principal pair property) *i) Soundness. If $PP_{\mathcal{E}}(t) = \langle P, \pi \rangle$, and Ch is a chain such that $Ch(\langle P, \pi \rangle) = \langle B, \sigma \rangle$, then $B \vdash_{\mathcal{E}} t{:}\sigma$.*

ii) Completeness. If $B \vdash_{\mathcal{E}} t{:}\sigma$, then there are a basis P and type π such that $PP_{\mathcal{E}}(t) = \langle P, \pi \rangle$, and there is a chain Ch such that $Ch(\langle P, \pi \rangle) = \langle B, \sigma \rangle$.

Proof: i) By Lemma 4.9, and Theorem 3.8.

ii) By Theorem 4.12. □

5 Subject reduction

If a term t is rewritten to the term t' using the rewrite rule $l \to r$, there is a subterm t_0 of t, and a replacement R, such that $l^{\mathrm{R}} = t_0$; t' is obtained by replacing t_0 by r^{R}. The subject reduction property for this notion of reduction is:

$$\text{If } B \vdash_{\mathcal{E}} t{:}\sigma, \text{ and } t \text{ can be rewritten to } t', \text{ then } B \vdash_{\mathcal{E}} t'{:}\sigma.$$

In this section, it will be shown that the notion of type assignment as given in this paper satisfies the subject reduction property. We will present in Definition 5.1 a notion of type assignment on rewrite rules using a restriction on the possible type assignments that guarantees this property. We will also, through examples, show that if the restriction is weakened, the thus obtained systems do not satisfy the subject reduction property: then there are rewrite rules typeable with respect to an environment $\mathcal{E}$, that match a term t typeable by σ with respect to $\mathcal{E}$, for which the result of the application of the rewrite rule on t is not typeable by σ with respect to $\mathcal{E}$.

Using the notion of principal pair, we now give a definition of a typeable rewrite rule and a typeable rewrite system.

Definition 5.1 Let $(\Sigma, \mathbf{R})$ be a *Gi*TRS, and $\mathcal{E}$ an environment.

i) **r**: $l \to r \in \mathbf{R}$ with defined symbol F *is typeable with respect to* $\mathcal{E}$, if there are basis B, type $\sigma \in \mathcal{T}_2$, such that
  a) $PP_{\mathcal{E}}(l) = \langle B, \sigma \rangle$, and $B \vdash_{\mathcal{E}} r{:}\sigma$.
  b) In $B \vdash_{\mathcal{E}} l{:}\sigma$ and $B \vdash_{\mathcal{E}} r{:}\sigma$, all F are typed with $\mathcal{E}(\mathsf{F})$.

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

Example 5.2 Derivations for the rewrite rules as given in Example 2.8, using:

$\mathcal{E}_2(\mathsf{M}) = (1\to2)\to((3\to4)\cap1)\to4$
$\mathcal{E}_2(\mathsf{F}) = (6\cap(5\to6)\cap5)\to6$
$\mathcal{E}_2(\mathsf{S}) = (7\to8\to9)\to(10\to8)\to(7\cap10)\to9$
$\mathcal{E}_2(\mathsf{I}) = 11\to11$.

$$\dfrac{\dfrac{\mathsf{x}{:}1\to3\to2 \qquad \mathsf{y}{:}1\to3}{\mathsf{S}_2(\mathsf{x},\mathsf{y}){:}1\to2}}{\mathsf{M}(\mathsf{S}_2(\mathsf{x},\mathsf{y})){:}((3\to4)\cap1)\to4} \quad\to\quad \dfrac{\mathsf{I}_0{:}(3\to4)\to3\to4 \qquad \mathsf{y}{:}1\to3}{\mathsf{S}_2(\mathsf{I}_0,\mathsf{y}){:}((3\to4)\cap1)\to4}$$

$$\dfrac{\mathsf{x}{:}6\cap(5\to6)\cap5}{\mathsf{F}(\mathsf{x}){:}6} \;\to\; \dfrac{\mathsf{x}{:}6\cap(5\to6)\cap5}{\mathsf{x}{:}6} \qquad \dfrac{\mathsf{x}{:}6\cap(5\to6)\cap5}{\mathsf{F}(\mathsf{x}){:}6} \;\to\; \dfrac{\dfrac{\mathsf{x}{:}6\cap(5\to6)\cap5}{\mathsf{x}{:}5\to6} \quad \dfrac{\mathsf{x}{:}6\cap(5\to6)\cap5}{\mathsf{x}{:}5}}{\mathsf{Ap}(\mathsf{x},\mathsf{x}){:}6}$$

Example 5.3 ([6]) The condition '$PP_{\mathcal{E}}(l) = \langle B, \sigma \rangle$' in Definition 5.1 (*i.a*) is crucial. Just saying

$$B \vdash_{\mathcal{E}} l{:}\sigma \text{ and } B \vdash_{\mathcal{E}} r{:}\sigma$$

would give a notion of type assignment that is not closed under rewriting (i.e. does not satisfy the subject reduction property).

Take the rewrite system of Example 2.8, that then would be typeable with respect to the following environment:

$\mathcal{E}_3(\mathsf{M}) = ((1\to2)\to3)\to(1\to2)\to2$
$\mathcal{E}_3(\mathsf{S}) = (4\to5\to6)\to(4\to5)\to4\to6$
$\mathcal{E}_3(\mathsf{I}) = 7\to7$

$$\dfrac{\dfrac{\mathsf{x}{:}(1\to2)\to1\to3 \qquad \mathsf{y}{:}(1\to2)\to1}{\mathsf{S}_2(\mathsf{x},\mathsf{y}){:}(1\to2)\to3}}{\mathsf{M}(\mathsf{S}_2(\mathsf{x},\mathsf{y})){:}(1\to2)\to2} \quad\to\quad \dfrac{\mathsf{I}_0{:}(1\to2)\to1\to2 \qquad \mathsf{y}{:}(1\to2)\to1}{\mathsf{S}_2(\mathsf{I}_0,\mathsf{y}){:}(1\to2)\to2}$$

Take the term $\mathsf{M}(\mathsf{S}_2(\mathsf{K}_0,\mathsf{I}_0))$. It is easy to see that the rewrite rule is allowed, and that this term rewrites to $\mathsf{S}_2(\mathsf{I}_0,\mathsf{I}_0)$. Although the first term is typeable by $(4\to5)\to5$ with respect to $\mathcal{E}_3$,

$$\dfrac{\dfrac{\mathsf{K}_0{:}(4\to5)\to(4\to5)\to4\to5 \qquad \mathsf{I}_0{:}(4\to5)\to4\to5}{\mathsf{S}_2(\mathsf{K}_0,\mathsf{I}_0){:}(4\to5)\to4\to5}}{\mathsf{M}(\mathsf{S}_2(\mathsf{K}_0,\mathsf{I}_0)){:}(4\to5)\to5}$$

the term $\mathsf{S}_2(\mathsf{I}_0,\mathsf{I}_0)$ is not typeable by the type $(4\to5)\to5$ with respect to $\mathcal{E}_3$. (In fact, it is not typeable at all with respect to $\mathcal{E}_3$; when constructing the principal pair for this term with respect to $\mathcal{E}_3$, unification fails.)

Notice that this example shows that the loss of subject reduction is not connected to the fact that intersection types are allowed.

In building $PP_{\mathcal{E}_3}(\mathsf{M}(\mathsf{S}_2(\mathsf{x},\mathsf{y})))$, types are assigned in the following way:

$$\dfrac{\dfrac{\mathsf{x}:(1\to2)\to4\to3 \qquad \mathsf{y}:(1\to2)\to4}{\mathsf{S}_2(\mathsf{x},\mathsf{y}):(1\to2)\to3}}{\mathsf{M}(\mathsf{S}_2(\mathsf{x},\mathsf{y})):(1\to2)\to2}$$

The right-hand side $\mathsf{S}_2(\mathsf{I}_0,\mathsf{y})$ of the rewrite rule is not typeable with $(1\to2)\to2$ using the basis $\{\mathsf{x}:(1\to2)\to4\to3,\ \mathsf{y}:(1\to2)\to4\}$. If the right-hand side should be typed with $(1\to2)\to2$, the type needed for y is $(1\to2)\to1$.

$$\dfrac{\mathsf{I}_0:(1\to2)\to1\to2 \qquad \mathsf{y}:(1\to2)\to1}{\mathsf{S}_2(\mathsf{I}_0,\mathsf{y}):(1\to2)\to2}$$

Since types assigned to term-variables in the right-hand side should occur in type assigned to types in the left-hand side, the type-variable 4 should be replaced by 1, so in the typed rewrite rule no longer the most general pair for the left-hand side is used.

We will now show that the use of a type-chain in rule $(\mathcal{F})$ is sound in the following sense: if there is a type-chain Ch such that $Ch(\mathcal{E}(\mathsf{F})) = \sigma$, then, for every type $\tau \in \mathcal{T}_2$ such that $\sigma \leq \tau$, the rewrite rules that define F are typeable with respect to a changed environment, in which $\mathcal{E}(\mathsf{F})$ is replaced by τ.

Theorem 5.4 *i) Let S be a substitution. Let* $\mathbf{r}$*: $l \to r$ 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 $\mathcal{E}[\mathsf{F}:=S(\mathcal{E}(\mathsf{F}))]$.*

ii) Let C be a copying such that $C(\langle B,\sigma\rangle) = \langle B',\sigma'\rangle$. Let $\mathbf{r}$*: $l \to r$ be a rewrite rule typeable with respect to the environment $\mathcal{E}$, and let F be the defined symbol of* $\mathbf{r}$*. If $C(\mathcal{E}(\mathsf{F})) = \tau \in \mathcal{T}_R$, then, for every $\mu \in \mathcal{T}_2$ such that $\tau \leq \mu$,* $\mathbf{r}$ *is typeable with respect to $\mathcal{E}[\mathsf{F}:=\mu]$.*

Proof: i) Straightforward.

ii) For every $\mu \in \mathcal{T}_2$ such that $\tau \leq \mu$, by Definition 3.2 there is a substitution S such that $\mu = S(\mathcal{E}(\mathsf{F}))$. The proof is completed by part (*i*. □

Before coming to the proof that the condition is sufficient, some preliminary results are needed.

Lemma 5.5 i) If $B \vdash_{\mathcal{E}} t^R{:}\sigma$, then there is a basis B' such that $B' \vdash_{\mathcal{E}} t{:}\sigma$, and for all $\mathsf{x}{:}\alpha \in B'$, $B \vdash_{\mathcal{E}} \mathsf{x}^R{:}\alpha$.

ii) If $B \vdash_{\mathcal{E}} t{:}\sigma$, and R is a replacement and B' a basis such that, for every statement $\mathsf{x}{:}\alpha \in B$, $B' \vdash_{\mathcal{E}} \mathsf{x}^R{:}\alpha$, then $B' \vdash_{\mathcal{E}} t^R{:}\sigma$.

iii) Let t be typeable, $PP_{\mathcal{E}}(t) = \langle P,\pi\rangle$, and for the replacement R *there are B and σ such that $B \vdash_{\mathcal{E}} t^R{:}\sigma$, then there is a type-chain Ch, such that $Ch(\pi) = \sigma$, and, for every statement $\mathsf{x}{:}\alpha \in P$, $B \vdash_{\mathcal{E}} \mathsf{x}^R{:}Ch(\alpha)$.*

Proof: The proofs of parts (*i* and (*ii* follow by easy induction on the structure of t. For part (*iii*, use part (*i*, and Theorem 4.13. □

The following theorem shows that the condition suffices.

Theorem 5.6 (SUBJECT REDUCTION) *Let* $\mathbf{r} : l \to r$ *be a typeable rewrite rule. Then, for every replacement* R, *basis* B *and a type* μ: $B \vdash_{\mathcal{E}} l^{\mathrm{R}}{:}\mu \;\Rightarrow\; B \vdash_{\mathcal{E}} r^{\mathrm{R}}{:}\mu$.

Proof: Let $PP_{\mathcal{E}}(l) = \langle P, \pi \rangle$. Since $\mathbf{r}$ is typeable, $P \vdash_{\mathcal{E}} r{:}\pi$. Suppose R is a replacement such that there are B, μ such that $B \vdash_{\mathcal{E}} l^{\mathrm{R}}{:}\mu$. (Assume $\mu \in \mathcal{T}_2$.) By Lemma 5.5(*iii*, there is a type-chain Ch such that

$$Ch(\pi) = \mu \;\;\&\;\; \forall\, \mathsf{x}{:}\alpha \in P \;[\; B \vdash_{\mathcal{E}} \mathsf{x}^{\mathrm{R}}{:}Ch(\alpha) \;].$$

By Theorem 3.8, $Ch(P) \vdash_{\mathcal{E}} r{:}Ch(\pi)$, so $Ch(P) \vdash_{\mathcal{E}} r{:}\mu$, and

$$\forall\, \mathsf{x}{:}\alpha \in P \;[\; B \vdash_{\mathcal{E}} \mathsf{x}^{\mathrm{R}}{:}Ch(\alpha) \;] \;\Rightarrow\; \forall\, \mathsf{x}{:}\alpha \in Ch(P) \;[\; B \vdash_{\mathcal{E}} \mathsf{x}^{\mathrm{R}}{:}\alpha \;].$$

So, by Lemma 5.5(*ii*, $B \vdash_{\mathcal{E}} r^{\mathrm{R}}{:}\mu$. □

In [6] it is also shown that the there formulated condition is necessary. This result is reached by extending the set of types with type constants, as also used in Example 5.3, and, for every rewrite rule that is typeable in the less restrictive way, creating a specific replacement that gives the counterexample. In this construction it is used that every type σ can be inhabited in a trivial way: just pick a constant Q, not already used, and assume that $\mathcal{E}(\mathsf{Q}) = \sigma$.

In the notion of type assignment as defined in this paper this construction cannot be given, because not every type can be trivially inhabited. An environment in this paper returns types in $\mathcal{T}_2$, and a function symbol F can only have an intersection type $\alpha \cap \beta$ if there exists a type-chain Ch such that $Ch(\mathcal{E}(\mathsf{F})) = \alpha \cap \beta$. This means that it is not possible to show that there is for example a function symbol that can be assigned the type $\mathsf{int} \cap (\mathsf{int} \to \mathsf{int})$.

6 Concluding remarks

6.1 Rank 2 type assignment in LC versus Rank 2 type assignment in *Gi*TRS

This paper introduced a notion of type assignment for *Gi*TRS. Although it is possible, as illustrated in Section 1.1, to define a similar notion for LC, results for such a system cannot be directly brought to *Gi*TRS. This is caused by an important difference between LC and TRS: the concept of *abstraction* is part of the definition of the former, but not of the latter. (Abstraction can be modelled in TRS, but it is not an explicit syntactic constructor of terms or rules in TRS.)

As argued in that section, it is possible to show that the principal type property holds. However, an important difference between a proof of the principal type property for the system for LC and the one obtained here for *Gi*TRS, is that the collection of operations differs. For the system of this paper, this collection of operations consists of substitution, copying, and weakening. Notice that, using the approach sketched above, the principal type for the term $(\lambda x.x)$ would be $\varphi \to \varphi$. Notice that also a type like $\sigma \cap \tau \to \sigma$ can be derived for $(\lambda x.x)$, so, in order to prove the principal type property, an operation of lifting should be specified that allows for the introduction of more types to the left of an arrow-type constructor, an operation that can change $\sigma \to \sigma$ into $\sigma \cap \tau \to \sigma$. Although the *Gi*TRS equivalent of the term $(\lambda x.x)$ can be given the type $\sigma \cap \tau \to \sigma$, it is not possible to obtain that type from the type $\sigma \to \sigma$: none of the operations specified in this paper is capable of changing a type in the way needed to go from $\sigma \to \sigma$ to $\sigma \cap \tau \to \sigma$.

Moreover, the operation that performs this is not sound for *Gi*TRS. Take for example the rewrite system

$$\begin{aligned} \mathsf{I}(x) &\to x \\ \mathsf{G}(\mathsf{I}_0) &\to \mathsf{I}_0 \\ \mathsf{H}(x) &\to x \end{aligned}$$

that is typeable with respect to the environment $\mathcal{E}$:

$$\begin{aligned} \mathcal{E}(\mathsf{I}) &= 1{\to}1 \\ \mathcal{E}(\mathsf{G}) &= (1{\to}1){\to}1{\to}1 \\ \mathcal{E}(\mathsf{H}) &= (1{\to}1){\to}1{\to}1. \end{aligned}$$

The operations that can be applied to types are defined on types only, so there is no way of distinguishing the types for G and H. We can safely add types to the 'argument' type for the rule that defines G, since the rule is typeable with respect to $\mathcal{E}[\mathsf{G}{:=}((1{\to}1){\cap}\sigma){\to}1{\to}1]$, for all σ. For the rule that defines G, however, not all types can be safely inserted: since all types σ should be types for I, the rewrite rule for G is not typeable with respect to the environment $\mathcal{E}[\mathsf{G}{:=}(2{\to}2){\cap}4{\to}3{\to}3]$.

So, the principal type property for the Rank 2 type assignment for LC is no direct consequence of the results of this paper. The converse also does not hold, since the proof of that property for the system in LC requires the operation of lifting, that is not provided for in the system of this paper. This of course implies that a notion of type assignment for LC cannot be directly translated into a type assignment system for a programming language that allows for functions to be specified as rewrite rules.

6.2 On implementation

The results of this paper could be used to implement a type-check algorithm for *Gi*TRS. It should be pointed out that the notion of type assignment as defined in this paper is really a *type-check* system, in the sense that it is not possible to create a type-inference algorithm, based on the approach of this paper. To obtain a type-inference algorithm an operation should be inserted that allows for more specific types than generated by substitution and copying. Take for example the rewrite rules (see examples 2.8 and 5.2)

$$\begin{aligned} \mathsf{F}(x) &\to x \\ \mathsf{F}(x) &\to \mathsf{Ap}(x,x) \end{aligned}$$

A type-inference algorithm could for example type both alternatives separately and try to combine the results found. For the first rule it would find $\mathcal{E}(\mathsf{F}) = \varphi_1{\to}\varphi_1$, for the second $\mathcal{E}(\mathsf{F}) = (\varphi_2{\to}\varphi_3){\cap}\varphi_2{\to}\varphi_3$. The problem is that it is not possible to create the desired type, $(\varphi_5{\cap}(\varphi_4{\to}\varphi_5){\cap}\varphi_4){\to}\varphi_5$, from these other two applying the operations specified in this paper. To obtain the correct type for F, $\varphi_6{\cap}(\varphi_5{\to}\varphi_6){\cap}\varphi_5{\to}\varphi_6$, an operation is needed that inserts extra types in the left-hand side of the top arrow-type constructor, like the one needed for LC.

So, it is allowed to give an environment for function symbols that is not a combination of possible environments for the various rules. This implies that, in particular, combining types found for one function symbol defined by several rules, applying the here defined operations, does not always lead to the right solution. It can be that the user 'sees' the right type for the rules, which the type-check algorithm is not capable of deducing, but will be capable of checking on its correctness. This can be seen as a disadvantage of the system, but, on the other hand, for many people it is nowadays considered to be programming hygiene to explicitly state the types for function definitions.

Although type assignment (and type-checking) using the here defined notion of type assignment is decidable, the complexity of type-checking is bigger than for a system based on

Curry-types. The biggest problem arises when checking the type provided for a function symbol. Suppose $l \to r$ is a rewrite rule. One way to implement type-checking for this rule would be to construct the principal pair $\langle P, \pi \rangle$ for the term l and to try to type r using this pair. Let $\sigma_1 \cap \cdots \cap \sigma_n$ be the type assigned to the term-variable x in P. Then, for every occurrence of x in r, some selection of the types in $\sigma_1 \cap \cdots \cap \sigma_n$ should be made. In the worst case the number of possibilities that must be tried is huge: 2^n. There are some more efficient ways to type-check a rule, but the complexity is still exponential. However, in every day programming life n will rarely be larger than 2.

6.3 Overloading

The concept of overloading in programming languages is normally used to express that different objects (typically procedures) can have the same identifier. (For another approach to overloading, see [15, 14].) At first sight this seems to be nothing but a tool to obtain programming convenience, but the implementational aspects of languages with overloading are not at all trivial. In functional programming languages, functions are *first-order citizens* which means that they can be handled as any object, like for example numbers. In particular, a function can be passed as argument to another one, or could be its result. Especially in the first case it can occur that at compile time it is not possible to decide which of the several bodies (or pieces of code) for an overloaded identifier should be linked into the object-code. If this decision cannot be made, the compiler should generate code that contains all possible functions and some kind of case-construct that makes it possible to select at runtime which is the code to use. For reasons of efficiency – and to avoid run-time checks on function types – it seems natural to allow for overloaded objects only if at compile time it can be decided which of the different function definitions is meant, since then, for every occurrence of an overloaded symbol, the compiler can decide which of the several function definitions should be linked into the object code.

The intersection type constructor is a good candidate to express overloading. It seems natural to say for example that the type for addition Add is $(\mathsf{int}\to\mathsf{int}\to\mathsf{int})\cap(\mathsf{real}\to\mathsf{real}\to\mathsf{real})$. Bringing the notion of overloading into a formal system for type assignment as defined in this paper implies that the restriction on the types that can be provided by an environment should be dropped; in such a formalism, types provided by the environment should be in $\mathcal{T}_{\mathrm{R}}$, not just $\mathcal{T}_2$.

However, this extension itself creates strange effects. Let, for example, F be a function symbol that has type $(\mathsf{int}\to\mathsf{int}\to\mathsf{int}) \cap (\mathsf{real}\to\mathsf{real}\to\mathsf{real}) \to \alpha$. Then, by the notion of type assignment as defined here, the term $\mathsf{F}(\mathsf{Add})$ can be typed by α. Moreover, let G be a function symbol that has the type $\sigma\cap\tau\to\rho$, and let H be an overloaded function symbol with $\mathcal{E}(\mathsf{H}) = \alpha\cap\beta$. Then finding the principal pair for the term $\mathsf{G}(\mathsf{H})$ requirers more than just the kind of unification defined in this paper. In general, there can be several cases, since all possible combinations have to be tried:

- $\mathit{unify}(\sigma, \alpha)$ and $\mathit{unify}(\tau, \beta)$ are both successful.
- $\mathit{unify}(\sigma, \beta)$ and $\mathit{unify}(\tau, \alpha)$ are both successful.
- $\mathit{unify}_{\mathrm{R2}}(\sigma\cap\tau, \alpha)$ and $\mathit{unify}_{\mathrm{R2}}(\sigma\cap\tau, \beta)$ are both successful.
- $\mathit{unify}_{\mathrm{R2}}(\sigma\cap\tau, \beta)$ fails, $\mathit{unify}_{\mathrm{R2}}(\sigma\cap\tau, \alpha)$ is successful.
- $\mathit{unify}_{\mathrm{R2}}(\sigma\cap\tau, \alpha)$ fails, $\mathit{unify}_{\mathrm{R2}}(\sigma\cap\tau, \beta)$ is successful.

It can even be that more than one of these cases is true at the same time, like for example the first and second. This in particular is troublesome, since it is not obvious at all what in

this case the type of G(H) should be. One solution for this problem would be to allow, like in [18], for more than one principal pair for a term (notice that this is not the same as saying that a principal type can be an intersection). Another would be to introduce – formally – an extra type constructor $+$ with the same meaning as $\cap$, and to define overloading using this notion. Then the unification of $\sigma\cap\tau$ and $\alpha+\beta$ can be defined as the combination of the results of unifying $\sigma\cap\tau$ and α, and unifying $\sigma\cap\tau$ and β.

A good solution to all aforementioned problems is to *force selection* of one of the function definitions for an overloaded identifier. This can be accomplished by defining, as in Definition 5.1, how a rewrite rule can be typed, but by adding that, for every $\sigma \in \mathcal{T}_2$ such that $\mathcal{E}(\mathsf{F}) \leq \sigma$, all the rewrite rules that define F should be typeable using the type σ, for every occurrence of F. (Another approach would be to introduce a new syntactic construct into the language that is used to separate the rules that define F in groups, and to ask that, for every $\sigma \in \mathcal{T}_2$ such that $\mathcal{E}(\mathsf{F}) \leq \sigma$, there is at least one group of rules that can be typed using σ.) Moreover, it is possible to define, as in rule $(\mathcal{F})$ how a type for a function symbol can be obtained form the one provided by the environment, in the following way:

$$(\mathcal{F}): \quad \frac{t_1{:}\sigma_1 \quad \ldots \quad t_n{:}\sigma_n}{\mathsf{F}(t_1,\ldots,t_n){:}\sigma} \ (\exists\, \tau \in \mathcal{T}_2, Ch\,[\mathcal{E}(F) \leq \tau \ \&\ Ch(\tau) = \sigma_1 \to \cdots \to \sigma_n \to \sigma])$$

Then the term F(Add) mentioned above cannot be typed. This selection is then reflected in the way intersection types are unified. Since only *one* of the types in an 'overloaded' type can be used, the unification should try to unify the demanded type with *each individual type* occurring in the provided type.

Using this definition, the notion of 'principal pair' becomes slightly more complicated. This is best explained by discussing the implementation of the type-checker that is looking for such a pair. Take the well-known function foldr that is defined by

```
foldr f i [ ]    = i
foldr f i (a:b) = f a (foldr f i b)
```

and can be typed by *(1→2→2)→2→[1]→2*. Take the term foldr Add 1 [2,3,4], then it is clear that this term should be typeable by the type int. When constructing the type assignment for this term, the subterm foldr Add is typed. For this term as such the type needed for Add cannot be uniquely determined: it is the second argument of foldr that forces the selection. Since there is a chance of success, the type-checker should postpone the decision to reject the term and consider both possibilities simultaneously. This means that formally the term foldr Add has *two* principal types.

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 in Applicative Term Rewriting Systems. In M. Bezem and J.F. Groote, editors, *Proceedings of TLCA '93. International Conference on Typed Lambda Calculi and Applications,* Utrecht, the Netherlands, volume 664 of *Lecture Notes in Computer Science*, pages 29–44. Springer-Verlag, 1993.

[3] S. van Bakel. Intersection Type Assignment Systems. *Theoretical Computer Science*, 151(2):385–435, 1995.

[4] S. van Bakel and M. Fernández. Strong Normalization of Typeable Rewrite Systems. In Jan Heering, Karl Meinke, Bernhard Möller, and Tobias Nipkow, editors, *Proceedings of HOA '93. First Inter-*

national Workshop on Higher Order Algebra, Logic and Term Rewriting, Amsterdam, the Netherlands. *Selected Papers*, volume 816 of *Lecture Notes in Computer Science*, pages 20–39. Springer-Verlag, 1994.

[5] S. van Bakel and M. Fernández. (Head-)Normalization of Typeable Rewrite Systems. In Jieh Hsiang, editor, *Proceedings of RTA '95. 6th International Conference on Rewriting Techniques and Applications,* Kaiserslautern, Germany, volume 914 of *Lecture Notes in Computer Science*, pages 279–293. Springer-Verlag, 1995.

[6] 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.

[7] F. Barbanera and M. Fernández. Combining first and higher order rewrite systems with type assignment systems. In M. Bezem and J.F. Groote, editors, *Proceedings of TLCA '93. International Conference on Typed Lambda Calculi and Applications,* Utrecht, the Netherlands, volume 664 of *Lecture Notes in Computer Science*, pages 60–74. Springer-Verlag, 1993.

[8] F. Barbanera and M. Fernández. Modularity of Termination and Confluence in Combinations of Rewrite Systems with λ_ω. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *Proceedings of ICALP '93. 20*$^{\text{th}}$ *International Colloquium on Automata, Languages and Programming,* Lund, Sweden, volume 700 of *Lecture Notes in Computer Science*, pages 657–668. Springer-Verlag, 1993.

[9] F. Barbanera, M. Fernández, and H. Geuvers. Modularity of Strong Normalization and Confluence in the λ-algebraic-cube. In *Proceedings of the ninth Annual IEEE Symposium on Logic in Computer Science,* Paris, France, 1994.

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

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

[12] 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.

[13] 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.

[14] G. Castagna. A Meta-Language for Typed Object-Oriented Languages. In R.K. Shyamasunda, editor, *Proceedings of FST&TCS '93. 13*$^{\text{th}}$ *Conference on Foundations of Software Technology and Theoretical Computer Science,* Bombay, India, volume 761 of *Lecture Notes in Computer Science*, pages 52,71. Springer-Verlag, 1993.

[15] G. Castagna, G. Ghelli, and G. Longo. A Calculus for Overloaded Functions with Subtyping. *Information and Computation*, 117(1):115–135, 1995.

[16] 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.

[17] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 27:45–58, 1981.

[18] M. Coppo and P. Giannini. A complete type inference algorithm for simple intersection types. 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 102–123. Springer-Verlag, 1992.

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

[20] 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.

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

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

[23] J.Y. Girard. The System F of Variable Types, Fifteen years later. *Theoretical Computer Science*,

45:159–192, 1986.

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

[25] A.J. Kfoury and J. Tiuryn. Type reconstruction in finite-rank fragments of the second-order λ-calculus. *Information and Computation*, 98(2):228–257, 1992.

[26] A.J. Kfoury, 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,* San Diego, California, pages 58–69, 1988.

[27] J.W. Klop. Term Rewriting Systems: a tutorial. *EATCS Bulletin*, 32:143–182, 1987.

[28] 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.

[29] D. Leivant. Polymorphic Type Inference. In *Proceedings 10th ACM Symposium on Principles of Programming Languages*, pages 88–98, Austin Texas, 1983.

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

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

[32] 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.

[33] V. van Oostrom. Lambda Calculus with Patterns. Technical Report IR-228, Faculteit der Wiskunde en Informatica, Vrije Universiteit Amsterdam, 1990.

[34] S. Peyton Jones. *The Implementation of Functional Programming Languages*. Series in Computer Science. Prentice/Hall International, Englewood Cliffs, NJ, USA, 1987.

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

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

[37] 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.