Strong Normalization of Typeable Rewrite Systems

Steffen van Bakel[1*] Maribel Fernández[2]

[1] Afdeling Informatica, Universiteit Nijmegen,
Toernooiveld 1, 6525 ED Nijmegen, Nederland.
steffen@cs.kun.nl
[2] LRI Bât. 490, CNRS / Université de Paris-Sud,
91405 Orsay Cedex, France.
maribel@lri.fr

Abstract

This paper studies termination properties of rewrite systems that are typeable using intersection types. It introduces a notion of partial type assignment on Curryfied Term Rewrite Systems, that consists of assigning intersection types to function symbols, and specifying the way in which types can be assigned to nodes and edges between nodes in the tree representation of terms. Two operations on types are specified that are used to define type assignment on terms and rewrite rules, and are proven to be sound on both terms and rewrite rules. Using a more liberal approach to recursion, a general scheme for recursive definitions is presented, that generalizes primitive recursion, but has full Turing-machine computational power. It will be proved that, for all systems that satisfy this scheme, every typeable term is strongly normalizable.

Introduction

Most functional programming languages, like Miranda [23] or ML [19] for instance, although implemented through an extended Lambda Calculus (LC) or a combinator system, allow programmers to specify an algorithm (function) as a set of equations using pattern-matching, i.e. the formal parameter of a function is allowed to have structure. Functional programs can then be seen as Term Rewriting Systems (TRS). Because of the underlying formalism, however, formal notions as type assignment (and other abstract interpretations) are studied in LC rather than on the level of the term rewriting language; the type assignment systems incorporated in most functional languages are in fact extensions of type assignment systems for LC.

It may seem straightforward to generalize formal systems defined on LC to the (significantly larger) world of TRS, but it is not evident that those ported 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 [4]. Also, as argued in [2], not every notion of type assignment for LC can be used for TRS, and vice versa.

To study the problem of termination a notion of types can be of significant value. For LC, there exists a well understood and well defined notion of type assignment, known as the Curry Type Assignment System [11] which expresses abstraction and application, and it can be shown that, for this notion of type assignment, all typeable terms are strongly normalizable.

*Supported by the Netherlands Organisation for the advancement of pure research (N.W.O.).

Curry's system forms the basis for a number of notions of type assignment used in functional programming, like for example the ones used in ML, and Miranda. In [9] the Intersection Type Discipline (the BCD-system) for LC is presented, which is a very powerful extension of Curry's system: it is closed under β-equality. Moreover, the set of terms having a head-normal form, the set of terms having a normal form, and the set of strongly normalizable terms can all be characterized by the set of assignable types. Because of this power, type assignment in this system is undecidable. In [1] a system is studied that is a variant of the BCD-system: also in this system type assignment is undecidable. That paper also contains a proof, using a Computability Predicate [22], of the statement that all typeable terms are strongly normalizable; also the converse holds.

In this paper we study the problem of termination of functional programs. Instead of studying the problem of termination using types in the world of LC, the approach taken will be to study the desired property directly on the level of a programming language with patterns, i.e. in the world of TRS. For this purpose, we introduce a notion of type assignment on Curryfied TRS (CTRS) that uses intersection types. CTRS are defined as a slight extension of the first order TRS defined in [12], in that functional types are allowed. They are restrictions of the Applicative TRS (ATRS) as defined in [4], in that the role of *Ap* in the left-hand side is restricted further.

In the past the role of types in TRS has been studied within the framework of first-order sorted rewrite systems [12], as used in the underlying model for the language OBJ (see e.g. [14]). For notions of type assignment that use the sorted approach, an enumerable collection of *sorts* is defined, and it is assumed that every F with arity n has a type $s_1 \times \cdots \times s_n \rightarrow s_{n+1}$, where $s_1, \ldots, s_{n+1}$ are sorts; functional types are not allowed. This implies that, by definition, there are sorts s_1, s_2 and s_3 such that (the binary operator) Ap has type $s_1 \times s_2 \rightarrow s_3$.

The disadvantage of this approach is, however, that the collection of typeable rewrite rules is very restricted. For example, the rewrite rules that correspond to Combinatory Logic,

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

cannot be typed using the types associated to the lambda term that correspond to the combinators S, K, and I.

The notion of type assignment presented in this paper combines the approach taken in those many-sorted, first-order rewrite systems, with the one commonly used for type assignment in LC (normally defined by presenting derivation rules). First of all, by introducing Ap next to other function symbols, we are able to express partial applications of those symbols. Secondly, using for Ap the type implicitly used in the derivation rule ($\rightarrow$E), as defined in type assignment systems for LC,

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

i.e. $(\sigma\rightarrow\tau)\times\sigma\rightarrow\tau$ – or, in a Curryfied notation, $(\sigma\rightarrow\tau)\rightarrow\sigma\rightarrow\tau$ – invokes the possibility to assign arrow-types to all objects: we consider *higher order types*.

Our type assignment system is *partial* in the sense of [20]: we will assume that every function symbol already has a type (given in an environment), whose structure is usually motivated by a rewrite rule. In fact, the approach we take here is very much the same as the one taken by Hindley in [16], where he defines the principal Curry-type scheme of an object in Combinatory Logic. Even this notion of type assignment could be regarded as a partial one.

In [6] another partial type assignment system for higher-order rewrite systems that uses intersection types is defined. It differs from ours in that function symbols are strongly-typed with sorts only, whereas we allow for types to contain type-variables as well, and in this way we can model polymorphism (by allowing for the replacement of type-variables in types by other types).

In [4] and [2] two partial intersection type assignment systems for ATRS are presented. Apart from the difference in the syntactic definition, the system we present here is a restriction of the first one, mainly because we do not consider the type-constant ω. The second system is a *decidable* restriction of the one presented here; the restriction lies in the structure of types.

Unlike typeable terms in LC, typeable terms in CTRS need not be strongly normalizable (consider a typeable term t and a rule $t \rightarrow t$). In order to ensure strong normalization of typeable terms in CTRS we will impose some syntactical restrictions on the rewrite rules: we will present a *general scheme* for recursive definitions that generalizes primitive recursion. This kind of recursive definition was presented by Jouannaud and Okada in [17] for the incremental definition of higher order functionals based on first order definitions, so that the whole system is terminating. The general scheme of [17] was also used in [6] and [7] for defining higher order functions compatible with different lambda calculi.

We will prove (using the well-known method of Computability Predicates [15], [22]) that for all typeable CTRS satisfying this scheme, every typeable term is strongly normalizable.

In Sect. 1 we define CTRS. The type assignment system is presented in Sect. 2. In Sect. 3 we introduce the general scheme and we prove that typeable systems satisfying this scheme are strongly normalizing on typeable terms. Section 4 contains the conclusions.

1 Preliminaries

We assume the reader to be familiar with LC [8], and refer to the papers [9], [1], and [3] for an overview of intersection type assignment. For full definitions of rewrite systems we refer to [18] and [12]. Typeability of lambda terms in the system as presented in [4] is denoted in this paper by the symbol $\vdash_{\lambda\cap}$, and by $\vdash_{\lambda\cap-\omega}$ typeability in the system that is obtained from that one by removing the type-constant ω completely.

The intersection type assignment system for LC satisfies the following properties:

Theorem 1.1 *i)* $B \vdash_{\lambda\cap} M{:}\sigma$ *&* $M =_\beta N \Rightarrow B \vdash_{\lambda\cap} N{:}\sigma$.

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

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

iv) $\exists\, B, \sigma\ [B \vdash_{\lambda\cap-\omega} M{:}\sigma] \Leftrightarrow M$ *is strongly normalizable.* ■

1.1 Curryfied Term Rewriting Systems

In this subsection we will present Curryfied Term Rewriting Systems as an extension of first-order TRS ([18], [12]) that allow partial application of function symbols. It is easy to see that the systems presented here are equivalent to another popular way to write TRS, i.e. as the pure applicative systems, that contain only *one* function symbol (an implicit application that is normally omitted when writing terms and rules). In view of this equivalence, we have chosen to develop theory and results in the first-order setting. The language of our systems is first-order, and we *add* the binary operator *Ap* rather that restricting us to systems with only that function symbol.

CTRS are also an extension of the *function constructor* systems used in most functional programming languages. In function constructor systems the collection of function symbols is divided in two categories: constructor symbols that, given sufficient data of the right kind, create an object of a specific algebraic data-type, and function symbols that specify arbitrary operations; and in rewrite rules, function symbols are not allowed to occur in patterns, and constructor symbols can not occur at the left-most, outermost position of the left-hand side.

The systems proposed here do not discriminate in this way. The extension we 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.

Definition 1.1.1 An *alphabet* or *signature* Σ consists of:

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

ii) A non-empty set $\mathcal{F}$ of *function symbols* F, G, ..., each with an 'arity'.

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

Definition 1.1.2 The set $T(\mathcal{F},\mathcal{X})$ of *terms* (or *expressions*) is defined inductively:

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

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

Definition 1.1.3 A *replacement* R is a map from $T(\mathcal{F},\mathcal{X})$ to $T(\mathcal{F},\mathcal{X})$ satisfying

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

So, R is determined by its restriction to the set of variables, and sometimes we will use the notation $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ to denote a replacement. We also write t^{R} instead of $\mathrm{R}(t)$.

Definition 1.1.4 *i)* A *rewrite rule* is a pair (l, r) of terms in $T(\mathcal{F},\mathcal{X})$; we write $\mathbf{r} : l \to r$. 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:

$$Ap\,(F_i\,(x_1, \ldots, x_i), x_{i+1}) \;\to\; F_{i+1}\,(x_1, \ldots, x_{i+1}).$$

ii) For every rewrite rule with left-hand side $F(t_1, \ldots, t_n)$ there are n additional rewrite rules:

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

iii) 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*:

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

iv) We call $\to_{\mathbf{r}}$ the *one-step rewrite relation* generated by $\mathbf{r}$. Concatenating rewrite steps we have (possibly infinite) *rewrite sequences* $t_0 \to t_1 \to t_2 \to \cdots$ (or *derivations* for short). If $t_0 \to \cdots \to t_n$, we write $t_0 \to^* t_n$.

v) We write $t \to_{\mathbf{R}} t'$, if there is a $\mathbf{r} \in \mathbf{R}$ such that $t \to_{\mathbf{r}} t'$.

The added rules in part *(ii)* with $F_{n-1}, \ldots, F_1, F_0$, etc. give the 'Curried'-versions of F, and the rewrite systems are called *Curry-closed*. When presenting a rewrite system, however, we will only show the rules that are essential; we will not show the rules that define the Curried versions.

Definition 1.1.5 A *Curryfied Term Rewriting System* (CTRS) is a pair $(\Sigma, \mathbf{R})$ of an alphabet Σ and a set $\mathbf{R}$ of rewrite rules.

We take the view that in a rewrite rule a certain symbol is defined; it is this symbol to which the structure of the rewrite rule gives a type.

Definition 1.1.6 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*. $Q \in \mathcal{F}$ is called a *constant symbol*, if there is no rewrite rule that defines Q.

We can draw the dependency-graph of the defined function-symbols, i.e. we can construct a graph whose nodes are labeled by the defined symbols of the rewrite rules, and draw an 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 rules

$$\begin{array}{l} F(x) \rightarrow G(x) \\ G(x) \rightarrow F(x). \end{array}$$

We will call a defined symbol F a *recursive symbol* if F occurs on a cycle in the dependency-graph, and call every rewrite rule that defines F *recursive*. All function-symbols that occur on one cycle in the dependency-graph depend on each other and are, therefore, defined *simultaneously*, so we are in fact forced 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.

It is always possible to introduce tupels into the language, and solve the problem of mutual recursion using them, so without loss of generality, we will assume that rules are *not* mutually recursive.

Definition 1.1.7 A TRS whose dependency-graph is acyclic is called a *hierarchical* TRS. The rewrite rules of a hierarchical TRS can be regrouped in such a way that they are *incremental* definitions of the defined symbols $F^1, \ldots, F^k$, so that the rules defining F^i only depend on $F^1, \ldots, F^{i-1}$.

Example 1.1.8 Our 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

$$\begin{array}{lcl} D(x) & \rightarrow & Ap(x,x) \\ Ap(D_0,x) & \rightarrow & D(x) \end{array}$$

– or, equivalently, $Ap(D,x) \rightarrow Ap(x,x)$ – is *not* considered a recursive system. Notice that, for example, the term $D(D_0)$ (or $Ap(D,D)$) has no normal form (these terms play 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 normalizable.

Definition 1.1.9 *Applicative Combinatory Logic* (ACL) is the CTRS $(\Sigma, \mathbf{R})$, where $\mathcal{F} = \{S, S_2, S_1, S_0, K, K_1, K_0, I, I_0\}$, and $\mathbf{R}$ contains the rewrite rules

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

Because ACL is Curry-closed, it is in fact combinatory complete: every lambda term can be translated into a term in ACL; for details of such a translation, see [8, 13].

2 Type assignment in CTRS

The set of types that will be used in the remainder of this paper is a subset of the one used in [4]: since we are going to use types to study strong normalization, we will not consider the type constant ω. This means that the system we present here is a subsystem of that of [4], and, in particular, no properties proved there are automatically 'inherited'.

Definition 2.1 *i)* $\mathcal{T}_{-\omega}$, the set of *strict types*, is inductively defined by:

a) All type-variables $\varphi_0, \varphi_1, \ldots \in \mathcal{T}_{-\omega}$.

b) If $\tau, \sigma_1, \ldots, \sigma_n \in \mathcal{T}_{-\omega}$ $(n \geq 1)$, then $\sigma_1 \cap \cdots \cap \sigma_n \rightarrow \tau \in \mathcal{T}_{-\omega}$.

ii) $\mathcal{T}_{\cap}$, the set of *strict intersection types*, is defined by: If $\sigma_1, \ldots, \sigma_n \in \mathcal{T}_{-\omega}$ $(n \geq 1)$, then $\sigma_1 \cap \cdots \cap \sigma_n \in \mathcal{T}_{\cap}$.

iii) On $\mathcal{T}_{\cap}$, the relation $\leq_{\cap}$ is defined by:

a) $\forall\, 1 \leq i \leq n$ $(n \geq 1)$ $[\sigma_1 \cap \cdots \cap \sigma_n \leq_{\cap} \sigma_i]$.

b) $\forall\, 1 \leq i \leq n$ $(n \geq 1)$ $[\sigma \leq_{\cap} \sigma_i] \Rightarrow \sigma \leq_{\cap} \sigma_1 \cap \cdots \cap \sigma_n$.

c) $\sigma \leq_{\cap} \tau \leq_{\cap} \rho \Rightarrow \sigma \leq_{\cap} \rho$.

iv) On $\mathcal{T}_{\cap}$, the relation $\sim_{\cap}$ is defined by:

a) $\sigma \leq_{\cap} \tau \leq_{\cap} \sigma \Rightarrow \sigma \sim_{\cap} \tau$.

b) $\rho \sim_{\cap} \sigma \;\&\; \tau \sim_{\cap} \mu \Rightarrow \sigma \rightarrow \tau \sim_{\cap} \rho \rightarrow \mu$.

$\mathcal{T}_{\cap}$ may be considered modulo $\sim_{\cap}$. Then $\leq_{\cap}$ becomes a partial order, and in this paper we consider types modulo $\sim_{\cap}$.

Unless stated otherwise, if $\sigma_1 \cap \cdots \cap \sigma_n$ is used to denote a type, then by convention all $\sigma_1, \ldots, \sigma_n$ are assumed to be strict. Notice that $\mathcal{T}_{-\omega}$ is a proper subset of $\mathcal{T}_{\cap}$. The notion of type assignment as presented in [2] is a (decidable) restriction of the one presented in this paper (and of the one presented in [4]). Decidability is, in that paper, achieved by limiting the structure of types, by requiring that in $\sigma_1 \cap \cdots \cap \sigma_n \rightarrow \tau$, the types $\sigma_1, \ldots, \sigma_n$ do not contain intersections.

Definition 2.2 *i)* A *statement* is an expression $t{:}\sigma$, with $t \in T(\mathcal{F}, \mathcal{X})$ and $\sigma \in \mathcal{T}_{\cap}$. t is the *subject* and σ the *predicate* of $t{:}\sigma$.

ii) If B is a basis and $\sigma \in \mathcal{T}_{\mathrm{S}}$, then $\mathcal{T}_{\langle B,\sigma \rangle}$ is the set of all strict subtypes occurring in the pair $\langle B, \sigma \rangle$.

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

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

2.1 Operations on pairs

In this subsection we present two different operations on pairs of $\langle$*basis*, *type*$\rangle$, namely substitution and expansion, that are variants of similar definitions given in [4, 5, 3]. The operation of substitution deals with the replacement of type-variables by types and is the one normally used. The operation of expansion replaces types by the intersection of a number of copies of that type and coincides with the one given in [10, 21].

Definition 2.1.1 *i)* The *substitution* $(\varphi \mapsto \alpha) : \mathcal{T}_\cap \to \mathcal{T}_\cap$, where φ is a type-variable and $\alpha \in \mathcal{T}_{-\omega}$, is defined by:

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

b) $(\varphi \mapsto \alpha)(\varphi') = \varphi'$, if $\varphi \not\equiv \varphi'$.

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

d) $(\varphi \mapsto \alpha)(\sigma_1 \cap \cdots \cap \sigma_n) = (\varphi \mapsto \alpha)(\sigma_1) \cap \cdots \cap (\varphi \mapsto \alpha)(\sigma_n)$.

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

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

Definition 2.1.2 The *last type-variable* of a strict type is defined by:

i) The last type-variable of φ is φ.

ii) The last type-variable of $\sigma_1 \cap \cdots \cap \sigma_n \to \tau$ is the last type-variable of τ.

Definition 2.1.3 For every $\mu \in \mathcal{T}_{-\omega}$, $n \geq 2$, basis B and $\sigma \in \mathcal{T}_\cap$, the quadruple $\langle \mu, n, B, \sigma \rangle$ determines an *expansion* $E_{\langle \mu, n, B, \sigma \rangle} : \mathcal{T}_\cap \to \mathcal{T}_\cap$, that is constructed as follows.

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

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

b) If the last type-variable of $\tau \in \mathcal{T}_{\langle B, \sigma \rangle}$ is in $\mathcal{V}_\mu(\langle B, \sigma \rangle)$, then all type-variables that occur in τ are in $\mathcal{V}_\mu(\langle B, \sigma \rangle)$.

ii) Suppose $\mathcal{V}_\mu(\langle B, \sigma \rangle) = \{\varphi_1, \ldots, \varphi_m\}$. Choose $m \times n$ different type-variables $\varphi_1^1, \ldots, \varphi_n^1, \ldots, \varphi_1^m, \ldots, \varphi_n^m$, such that each φ_i^j $(1 \leq j \leq m,\ 1 \leq i \leq n)$ does not occur in $\langle B, \sigma \rangle$. Let $S_i = (\varphi_1 \mapsto \varphi_i^1) \circ \cdots \circ (\varphi_m \mapsto \varphi_i^m)$.

iii) Let $\alpha \in \mathcal{T}_\cap$. $E_{\langle \mu, n, B, \sigma \rangle}(\alpha)$ is obtained by traversing α top-down and replacing, in α, a subtype β which last type-variable is an element of $\mathcal{V}_\mu(\langle B, \sigma \rangle)$, by $S_1(\beta) \cap \cdots \cap S_n(\beta)$.

Both operations are, in a natural way, extended into operations that are also defined on bases and pairs of basis and type.

For these operations, the following properties hold:

Lemma 2.1.4 Let S be a substitution, and $E = E_{\langle \mu, n, B, \sigma \rangle}$ an expansion.

i) If $\tau \leq_\cap \rho$, then $S(\tau) \leq_\cap S(\rho)$, and $E(\tau) \leq_\cap E(\rho)$.

ii) If $B \leq_\cap B'$, then $S(B) \leq_\cap S(B')$, and $E(B') \leq_\cap E(B'')$.

iii) *If* $\tau \in \mathcal{T}_{\langle B,\sigma\rangle}$, *then either:* $E\,(\tau) = \tau_1\cap\cdots\cap\tau_n$ *where for every* $1 \leq i \leq n$, τ_i *is a trivial variant of* τ, *or* $E\,(\tau) \in \mathcal{T}_{-\omega}$.

iv) $E\,(\Pi\{B_1, \ldots, B_n\}) = \Pi\{E\,(B_1), \ldots, E\,(B_n)\}$.

Definition 2.1.5 A *chain* is an object $<O_1, \ldots, O_n>$, with each O_i an operation of substitution or expansion, such that:

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

2.2 Type assignment

The notion of type assignment on CTRS will in fact be defined on the tree-representation of terms and rewrite rules of these systems.

Definition 2.2.1 *i)* The tree-representation of terms and rewrite rules is obtained in a straightforward way by representing a term $F(t_1, \ldots, t_n)$ by:

$$\begin{tikzcd} & {} \arrow[d] & \\ & F \arrow[dl] \arrow[dr] & \\ t_1 & \cdots & t_n \end{tikzcd}$$

The *spine* of a term-tree is defined as usual, i.e. the root node of the term-tree is on the spine, and if a node is on the spine, then its left most descendent is on the spine. The first node on the spine of the left hand side (starting from the root node) that does not contain an Ap is called the *defining node* of that rule. The edge pointing to the the root of a term is called the *root edge*, and a node containing a term-variable (a function symbol $F \in \mathcal{F}$, the symbol Ap) will be called a *variable node* (*function node*, *application node*). Notice that if F is the defined symbol of the rule, then it occurs in the defining node.

ii) Subterms can be numbered by *positions*, which are sequences of natural numbers denoting the path from the root of the term to the root of the subterms. The letters p and q stand for positions. The empty sequence (*root position*) is denoted by Λ. The subterm of t *at position* p is denoted by $t|_p$ and $t[u]_p$ is the result of replacing the subterm of t at position p by u.

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

Definition 2.2.2 Let $(\Sigma, \mathbf{R})$ be a CTRS.

i) An *environment* $\mathcal{E} : \mathcal{F} \cup \{Ap\} \rightarrow \mathcal{T}_{-\omega}$ is such that for every $F \in \mathcal{F}$ with arity n, $\mathcal{E}\,(F) = \mathcal{E}\,(F_{n-1}) = \cdots = \mathcal{E}(F_0)$, and $\mathcal{E}(Ap) = (\varphi_1\rightarrow\varphi_2)\rightarrow\varphi_1\rightarrow\varphi_2$.

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

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

Type assignment on CTRS is defined in two stages. In the next definition we define type assignment on terms, in Def.*2.2.7* we define type assignment on term rewrite rules.

Definition 2.2.3 Let $(\Sigma,\, \mathbf{R})$ be a CTRS, and $\mathcal{E}$ an environment.

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

a) The root edge of t is typed with σ.

b) The type assigned to a function node containing $F \in \mathcal{F} \cup \{Ap\}$ (where F has arity $n \geq 0$) is $\tau_1\cap\cdots\cap\tau_m$, if and only if, for every $1 \leq i \leq m$, there are $\sigma^i_1, \ldots, \sigma^i_n \in \mathcal{T}_\cap$, and $\sigma_i \in \mathcal{T}_{-\omega}$, such that $\tau_i = \sigma^i_1\rightarrow\cdots\rightarrow\sigma^i_n\rightarrow\sigma_i$, the type assigned to the j-th $(1 \leq j \leq n)$ out-going edge is $\sigma^1_j\cap\cdots\cap\sigma^m_j$, and the type assigned to the incoming edge is $\sigma_1\cap\cdots\cap\sigma_m$.

$$\begin{array}{c}
\downarrow\, \sigma_1\cap\cdots\cap\sigma_m \\
F{:}(\sigma^1_1\rightarrow\cdots\rightarrow\sigma^1_n\rightarrow\sigma_1)\cap\cdots\cap\,(\sigma^m_1\rightarrow\cdots\rightarrow\sigma^m_n\rightarrow\sigma_m) \\
\swarrow \sigma^1_1\cap\cdots\cap\sigma^m_1 \quad \swarrow \sigma^1_2\cap\cdots\cap\sigma^m_2 \quad \cdots \quad \searrow \sigma^1_{n-1}\cap\cdots\cap\sigma^m_{n-1} \quad \searrow \sigma^1_n\cap\cdots\cap\sigma^m_n
\end{array}$$

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

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

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

The use of an environment and part *(i.c)* of Def.*2.2.3* introduce a notion of polymorphism into our type assignment system. The environment returns the 'principal type' for a function symbol; this symbol can be used with types that are 'instances' of its principal type.

A typical example for part *(i.b)* of Def.*2.2.3* is the symbol Ap; for every occurrence of Ap in a term-tree, there are $\sigma_1,\ldots,\sigma_n$ and $\tau_1,\ldots,\tau_n$ such that the following is part of the term-tree.

$$\begin{array}{c}
\downarrow\, \tau_1\cap\cdots\cap\tau_n \\
Ap{:}\ ((\sigma_1\rightarrow\tau_1)\rightarrow\sigma_1\rightarrow\tau_1)\cap\cdots\cap((\sigma_n\rightarrow\tau_n)\rightarrow\sigma_n\rightarrow\tau_n) \\
\swarrow (\sigma_1\rightarrow\tau_1)\cap\cdots\cap(\sigma_n\rightarrow\tau_n) \qquad \searrow \sigma_1\cap\cdots\cap\sigma_n
\end{array}$$

Example 2.2.4 The term $S\,(K_0, S_0, I_0)$ can be typed with the type $\sigma\rightarrow\sigma$, under the assumption that: $\mathcal{E}\,(S) = (1\rightarrow2\rightarrow3)\rightarrow(4\rightarrow2)\rightarrow1\cap4\rightarrow3$, $\mathcal{E}\,(K) = 5\rightarrow6\rightarrow5$, and $\mathcal{E}\,(I) = 7\rightarrow7$.

$$\begin{array}{c}
\downarrow\, \sigma\rightarrow\sigma \\
S{:}((\sigma\rightarrow\sigma)\rightarrow((\tau\rightarrow\rho)\rightarrow(\rho\rightarrow\mu)\cap\tau\rightarrow\mu)\rightarrow\sigma\rightarrow\sigma)\ \rightarrow \\
(((\rho\rightarrow\mu)\rightarrow\rho\rightarrow\mu)\rightarrow(\tau\rightarrow\rho)\rightarrow(\rho\rightarrow\mu)\cap\tau\rightarrow\mu)\ \rightarrow \\
(\sigma\rightarrow\sigma)\cap((\rho\rightarrow\mu)\rightarrow\rho\rightarrow\mu)\ \rightarrow\ \sigma\rightarrow\sigma \\
\swarrow \qquad \downarrow \qquad \searrow \\
K_0{:}(\sigma\rightarrow\sigma)\rightarrow((\tau\rightarrow\rho)\rightarrow(\rho\rightarrow\mu)\cap\tau\rightarrow\mu)\rightarrow\sigma\rightarrow\sigma \qquad I_0{:}(\sigma\rightarrow\sigma)\cap((\rho\rightarrow\mu)\rightarrow\rho\rightarrow\mu) \\
S_0{:}((\rho\rightarrow\mu)\rightarrow\rho\rightarrow\mu)\rightarrow(\tau\rightarrow\rho)\rightarrow(\rho\rightarrow\mu)\cap\tau\rightarrow\mu
\end{array}$$

Notice that to obtain the type for S in the root-node, we have used the chain $<(1\mapsto\sigma\rightarrow\sigma),\ (2\mapsto(\tau\rightarrow\rho)\rightarrow(\rho\rightarrow\mu)\cap\tau\rightarrow\mu),\ (3\mapsto\sigma\rightarrow\sigma),\ (4\mapsto(\rho\rightarrow\mu)\rightarrow\rho\rightarrow\mu)>$.

Notice, moreover, that to obtain the type for I_0, an expansion is needed.
If we define $D(x) \rightarrow Ap(x, x)$, then we can even check that for example $D(S(K_0, S_0, I_0))$ and $D(I_0)$ are both typeable by $\sigma \rightarrow \sigma$.

The here defined notion of type assignment satisfies the following properties:

Lemma 2.2.5 *i)* $B \vdash_{\mathcal{E}} t{:}\sigma_1 \cap \cdots \cap \sigma_n \Leftrightarrow \forall\, 1 \leq i \leq n\ [B \vdash_{\mathcal{E}} t{:}\sigma_i]$.
ii) $B \vdash_{\mathcal{E}} t{:}\sigma$ *&* $\sigma \leq_{\cap} \tau \Rightarrow B \vdash_{\mathcal{E}} t{:}\tau$.
iii) $B \vdash_{\mathcal{E}} F_n(t_1, \ldots, t_n){:}\sigma$ *&* $\sigma \in \mathcal{T}_{-\omega} \Rightarrow \exists\, \alpha \in \mathcal{T}_{\cap},\ \beta \in \mathcal{T}_{-\omega}\ [\sigma = \alpha \rightarrow \beta]$.
iv) $\Pi\{B, \{x{:}\alpha\}\} \vdash_{\mathcal{E}} Ap(t, x){:}\beta$ *& x does not occur in B* $\Rightarrow B \vdash_{\mathcal{E}} t{:}\alpha \rightarrow \beta$.

By Def.*1.1.4*, if a term t is rewritten to the term t' using the rewrite rule $l \rightarrow r$, there is a subterm t_0 of t, and a replacement R, such that $l^{\mathrm{R}} = t_0$, and t' is obtained by replacing t_0 by r^{R}. The subject reduction property for this notion of reduction is: If $B \vdash_{\mathcal{E}} t{:}\sigma$, and t can be rewritten to t', then $B \vdash_{\mathcal{E}} t'{:}\sigma$.

To ensure the subject reduction property, as in [4], type assignment on rewrite rules will be defined using the notion of principal pair for a typeable term.

Definition 2.2.6 Let $t \in T(\mathcal{F}, \mathcal{X})$. A pair $\langle P, \pi \rangle$ is called *a principal pair for t with respect to* $\mathcal{E}$, if $P \vdash_{\mathcal{E}} t{:}\pi$ and for every B, σ such that $B \vdash_{\mathcal{E}} t{:}\sigma$ there is a chain C such that $C(\langle P, \pi \rangle) = \langle B, \sigma \rangle$.

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

Definition 2.2.7 Let $(\Sigma, \mathbf{R})$ be a CTRS, and $\mathcal{E}$ an environment.

i) We say that $l \rightarrow r \in \mathbf{R}$ with defined symbol F *is typeable with respect to* $\mathcal{E}$, if there are basis P, type $\pi \in \mathcal{T}_{\cap}$, and an assignment of types to nodes and edges such that:

   a) $\langle P, \pi \rangle$ is a principal pair for l with respect to $\mathcal{E}$, and $P \vdash_{\mathcal{E}} r{:}\pi$.

   b) In $P \vdash_{\mathcal{E}} l{:}\pi$ and $P \vdash_{\mathcal{E}} r{:}\pi$, all nodes containing F are typed with $\mathcal{E}(F)$.

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

From now on, we will only consider CTRS that are typeable with respect to a certain environment.

Condition *(i.b)* of Def.*2.2.7* is in fact added to make sure that the type provided by the environment for a function symbol F is not in conflict with the rewrite rules that define F. By restricting the type that can be assigned to the defined symbol to the type provided by the environment, we are sure that the rewrite rule is typed using that type, and not using some other type. Since by part *(i.b)* of Def.*2.2.7* all occurrences of the defined symbol in a rewrite rule are typed with the same type, type assignment of rewrite rules is actually defined using Milner's way of dealing with recursion [19].

Using the same technique as in [4], it is possible to show that subject reduction holds.

Theorem 2.2.8 Subject Reduction Theorem. *If* $B \vdash_{\mathcal{E}} t{:}\sigma$, *and* $t \rightarrow_{\mathbf{R}} t'$, *then* $B \vdash_{\mathcal{E}} t'{:}\sigma$. ■

It is possible to show that the two operations on pairs (substitution and expansion) are sound on typed term-trees, and that part *(i.c)* of Def.*2.2.3* is sound in the following sense: if there is an operation O such that $O(\mathcal{E}(F)) = \sigma$, then for every type $\tau \in \mathcal{T}_{-\omega}$ such that $\sigma \leq_{\cap} \tau$, the rewrite rules that define F are typeable with respect to the changed environment $\mathcal{E}[F := \tau]$.

Theorem 2.2.9 Soundness of substitution. *Let S be a substitution.*

i) If $B \vdash_{\mathcal{E}} t{:}\sigma$, then $S(B) \vdash_{\mathcal{E}} t{:}S(\sigma)$.
ii) Let $\mathbf{r}$: $l \to r$ with defined symbol F be typeable with respect to $\mathcal{E}$. Then $\mathbf{r}$ is typeable with respect to $\mathcal{E}[F := S(\mathcal{E}(F))]$. ■

Theorem 2.2.10 Soundness of expansion. *Let E be an expansion.*

i) If $B \vdash_{\mathcal{E}} t{:}\sigma$, then $E(B) \vdash_{\mathcal{E}} t{:}E(\sigma)$.
ii) Let $\mathbf{r}$: $l \to r$ with defined symbol F be typeable with respect to $\mathcal{E}$. If $E(\mathcal{E}(F)) = \tau \in \mathcal{T}_{\cap}$, then for every $\mu \in \mathcal{T}_{-\omega}$ such that $\tau \leq_{\cap} \mu$, $\mathbf{r}$ is typeable with respect to $\mathcal{E}[F := \mu]$. ■

3 Strong Normalization and Typeability

Unlike typeable terms in $\vdash_{\lambda\cap}$, typeable terms in $\vdash_{\mathcal{E}}$ need not be normalizable. This means that the characterization of strongly normalizable terms and strongly normalizing CTRS can not be based on type conditions only, as it is possible for LC (see [1]).

In this section we analyze the relations between typeability and strong normalization in CTRS. First we show the relation between strong normalization in CTRS and normalization in $\vdash_{\lambda\cap}$; however, since the property we are characterizing is non-decidable, the condition we obtain is non-decidable as well. From a practical point of view, a decidable, sufficient (but not necessary) condition is clearly better. For this reason, in the second subsection we introduce a *general scheme*, inspired by [17], which imposes some syntactical restrictions on CTRS's rules so as to guarantee the strong normalization of all typeable terms.

From now on, we will abbreviate 't is strongly normalizable' by SN(t).

3.1 Strong Normalization of CTRS and Typeability in $\vdash_{\lambda\cap}$

Let us consider an interpreter of CTRS, i.e. a program such that, given a term t and a CTRS $(\Sigma, \mathbf{R})$, returns the set $\{t' \mid t \to_{\mathbf{R}} t'\}$ (the empty set if t is a normal form for $(\Sigma, \mathbf{R})$). In fact, we will consider a generalized interpreter P, which takes a set I of terms and a CTRS $(\Sigma, \mathbf{R})$ as input and returns the set $\bigcup_{t \in I} \{t' \mid t \to_{\mathbf{r}} t'\}$ (the empty set if I is a set of normal forms for $(\Sigma, \mathbf{R})$). We assume that the inputs and the output of P are sets of strings representing terms and rewrite rules. Besides, we assume that pure LC is used to write P, that is, P is a λ-term that will be applied to sets of strings (also coded in LC). We will use the symbol t for both the term in CTRS and its interpretation in LC. Then the λ-term $P\mathbf{R}\,t$ has a β-normal form which represents the set of one-step reducts of t (with respect to $(\Sigma, \mathbf{R})$).

The following property will be used to characterize strongly normalizable terms and CTRS:

Property 3.1.1 Let $(\Sigma, \mathbf{R})$ be a CTRS, and let $P^n\mathbf{R}\,t$ denote the term $(P\mathbf{R}\,(P\mathbf{R}\cdots(P\mathbf{R}\,t)\cdots))$, with n times P.

i) $SN_{\mathbf{R}}\,(\,)(t) \Leftrightarrow \exists\, n \geq 0\; [P^n\mathbf{R}\,t =_{\beta} \emptyset]$,
ii) $SN((\Sigma, \mathbf{R})) \Leftrightarrow \forall\, t \in T(\mathcal{F}, \mathcal{X})\; [SN_{\mathbf{R}}\,(\,)(t)]$.

Given t and $(\Sigma, \mathbf{R})$, we can obtain any t' such that $t \to_{\mathbf{R}}^* t'$ using a program P^* (also in pure LC) that applies P a number of times. By definition, the result of applying P^* to $(\Sigma, \mathbf{R})$ and t is the set of strings $\{t' \mid t \to_{\mathbf{R}}^* t'\}$.

Since β-reduction is confluent, property *3.1.1* can be reformulated in this way: For any strongly normalizable term t in a CTRS $(\Sigma, \mathbf{R})$, the λ-term $P^*\mathbf{R}t$ has a β-normal form. A CTRS $(\Sigma, \mathbf{R})$ is strongly normalizing if for all t the term $P^*\mathbf{R}t$ has a β-normal form. Therefore, using Thm.*1.1* we can characterize strongly normalizable terms and strongly normalizing systems using $\vdash_{\lambda\cap}$:

Property 3.1.2 Let $(\Sigma, \mathbf{R})$ *be a CTRS.*

i) $SN_{\mathbf{R}}(\,)(t) \Leftrightarrow P^*\mathbf{R}t$ *is typeable in* $\vdash_{\lambda\cap}$ *with a type different from* ω,

ii) $SN((\Sigma, \mathbf{R})) \Leftrightarrow$ *for all* t, $P^*\mathbf{R}t$ *is typeable in* $\vdash_{\lambda\cap}$ *with a type different from* ω.

Note that the above results show no relation between the types assignable to a term in a CTRS, and those assignable to its interpretation in LC. In fact, Prop.*3.1.2* shows only the relation between β-reduction and $\to_{\mathbf{R}}$, since we are implementing $\to_{\mathbf{R}}$ by means of $\to_\beta$.

3.2 Strong Normalization versus Typeability in CTRS

In this section we give decidable conditions for strong normalization of typeable CTRS. Since typeability alone is not sufficient, we will impose syntactic restrictions on the rules in order to get strongly normalizing systems.

We will prove that the class of typeable non-recursive CTRS is strongly normalizing. To appreciate the non-triviality of this condition, remember Ex.*1.1.8*: a non-recursive CTRS can be non-terminating. In fact, the main result of this section (every typeable term is strongly normalizable) shows that the term $D(D_0)$ (or $Ap(D,D)$) is not typeable.

Theorem 3.2.1 If $\mathbf{R}$ *contains no recursive rule and is typeable in* $\vdash_{\mathcal{E}}$, *then any typeable term is strongly normalizable with respect to* $\mathbf{R}$.

The converse of this theorem does not hold. However, the restriction to non-recursive systems is too strong indeed. In the following we will show that there is class of recursive functions that are safe: generalized primitive recursive functions satisfying the general scheme below. This scheme for definitions is inspired by [17] where generalized primitive recursive definitions were shown strongly normalizing when combined with typed LC. The same results were shown in [6] in the context of type assignment systems for LC and in [7] for typed LC of order ω.

Definition 3.2.2 Let $\mathcal{F}_n = \mathcal{C} \cup \{Ap\} \cup \{F^1, \ldots, F^n\}$, where $F^1, \ldots, F^n$ are the defined symbols of the signature, that are not Curried-versions, and assume that $F^1, \ldots, F^n$ are defined in an incremental way. Suppose, moreover, that the rules defining $F^1, \ldots, F^n$ satisfy the *general scheme*:

$$F^i(\overrightarrow{\mathrm{C}}[\overline{x}], \overline{y}) \to \mathrm{C}'[F^i(\overrightarrow{\mathrm{C}_1}[\overline{x}], \overline{y}), \ldots, F^i(\overrightarrow{\mathrm{C}_m}[\overline{x}], \overline{y}), \overline{x}, \overline{y}],$$

where $\overline{x}$, $\overline{y}$ are sequences of variables, and $\overline{x} \subseteq \overline{y}$. Also, $\overrightarrow{\mathrm{C}}[\,]$, $\mathrm{C}'[\,]$, $\overrightarrow{\mathrm{C}_1}[\,]$, and $\overrightarrow{\mathrm{C}_m}[\,]$ are sequences of contexts in $T(\mathcal{F}_{i-1}, \mathcal{X})$, and, for $1 \le j \le m$, $\overrightarrow{\mathrm{C}}[\overline{x}] \rhd_{mul} \overrightarrow{\mathrm{C}_j}[\overline{x}]$ (where $\lhd$ is the strict subterm ordering and *mul* denotes multiset extension).
The rules defining $F^1, \ldots, F^n$ and their Curry-closure together form a *safe recursive system*.

This general scheme imposes some restrictions on the definition of functions: the terms in every $\overrightarrow{\mathrm{C}_j}[\overrightarrow{x}]$ are subterms of terms in $\overrightarrow{\mathrm{C}}[\overrightarrow{x}]$ (this is the 'primitive recursive' aspect of the scheme), and the variables $\overrightarrow{x}$ must also appear as arguments in the left-hand side of the rule.

It is worthwhile noting that the rewrite rules of Def. *1.1.9* are *not* recursive, so, in particular, satisfy the scheme. Therefore, although the severe restriction imposed on rewrite rules, the systems satisfying the scheme still have full Turing-machine computational power, a property that systems without *Ap* would not possess.

Example 3.2.3 The following rewrite system satisfies the general scheme:

$$
\begin{array}{ll}
\mathit{Add}\,(\mathit{Zero}, y) & \to y \\
\mathit{Add}\,(\mathit{Succ}\,(x), y) & \to \mathit{Succ}\,(\mathit{Add}\,(x, y)) \\
\mathit{Mul}\,(\mathit{Zero}, y) & \to \mathit{Zero} \\
\mathit{Mul}\,(\mathit{Succ}\,(x), y) & \to \mathit{Add}\,(\mathit{Mul}\,(x, y), y) \\
\mathit{Fac}\,(\mathit{Zero}) & \to \mathit{Succ}\,(\mathrm{Zero}) \\
\mathit{Fac}\,(\mathit{Succ}\,(x)) & \to \mathit{Mul}\,(\mathit{Succ}\,(x), \mathit{Fac}\,(x))
\end{array}
$$

Note that if we extend the definition of *Add* with the rule that expresses the associativity of *Add*,

$$\mathit{Add}\,(\mathit{Add}\,(x, y), z) \to \mathit{Add}\,(x, \mathit{Add}\,(y, z))$$

the rewrite system is no longer safe.

Theorem *3.2.1* is actually a corollary of the Strong Normalization Theorem (Thm.*3.2.9*). To prove this theorem, we will use the well-known method of Computability Predicates ([15], [22]). The proof will have two parts; in the first one we give the definition of a predicate *Comp* on bases, terms, and types, and prove some properties of *Comp*. The most important one states that if for a term t there are a basis B and type σ such that $\mathit{Comp}\,(B,\, t,\, \sigma)$ holds, then t is strongly normalizable. In the second part *Comp* is shown to hold for each typeable term.

Definition 3.2.4 *i)* Let B be a basis, t a term, and σ a type, such that $B \vdash_{\mathcal{E}} t{:}\sigma$. We define the Computability Predicate $\mathit{Comp}\,(B,\, t,\, \sigma)$ by induction on σ:

a) $\mathit{Comp}\,(B,\, t,\, \varphi) \Leftrightarrow \mathrm{SN}(t)$.

b) $\mathit{Comp}\,(B,\, t,\, \sigma{\to}\tau) \Leftrightarrow$
$\quad\forall\, u \in T(\mathcal{F}, \mathcal{X})\ [\mathit{Comp}\,(B\text{'},\, u,\, \sigma) \Rightarrow \mathit{Comp}\,(\Pi\{B, B\text{'}\},\, \mathit{Ap}\,(t, u),\, \tau)]$.

c) $\mathit{Comp}\,(B,\, t,\, \sigma_1{\cap}\cdots{\cap}\sigma_n) \Leftrightarrow \forall\, 1 \leq i \leq n\ [\mathit{Comp}\,(B,\, t,\, \sigma_i)]$.

ii) We say that t is *computable* if there exist B, σ such that $\mathit{Comp}\,(B,\, t,\, \sigma)$.

iii) R is *computable in B* $\Leftrightarrow \exists\, B\text{'}\ [\forall\, x{:}\sigma \in B\ [\mathit{Comp}\,(B\text{'},\, x^{\mathrm{R}},\, \sigma)]]$.

Notice that because we use intersection types, and because of Def.*2.2 (iii)*, in part *(iii)* we need not consider the existence of different bases for each $x{:}\sigma \in B$.

Definition 3.2.5 A term is *neutral* if it is not of the form $F_n(t_1, \ldots, t_n)$ where F_n is a 'Curried'-version of some defined symbol F.

Property 3.2.6 Comp satisfies the standard properties of computability predicates:
C1) $\mathit{Comp}\,(B,\, t,\, \sigma) \Rightarrow \mathit{SN}(t)$.

C2) If Comp $(B,\ t,\ \sigma)$, and $t \to^* t'$, then Comp $(B,\ t',\ \sigma)$.

C3) If t is neutral, $B \vdash_{\mathcal{E}} t{:}\sigma$ for some B, σ, and for all v such that $t \to_{\mathbf{R}} v$, Comp $(B,\ v,\ \sigma)$ holds, then also Comp $(B,\ t,\ \sigma)$ holds.

Proof: By induction on the structure of types.

i) $\sigma = \varphi$.

C1) By Def.*3.2.4 (i.a)*.

C2) By Def.*3.2.4 (i.a)*, and Thm.*2.2.8*.

C3) By Def.*3.2.4 (i.a)*, SN(v). Then SN(t) and again by *3.2.4 (i.a)*, $Comp\,(B,\ t,\ \varphi)$.

ii) $\sigma = \alpha{\to}\beta$.

C1) Let $u \equiv x$ (a new variable). x is a neutral term in normal form, and $\{x{:}\alpha\} \vdash_{\mathcal{E}} x{:}\alpha$. Then, by induction hypothesis (*C3*), $Comp\,(\{x{:}\alpha\},\ x,\ \alpha)$, and by Def.*3.2.4 (i.b)*, $Comp\,(\Pi\{B{,}\{x{:}\alpha\}\},\ Ap\,(t,x),\ \beta)$. By induction hypothesis, SN($Ap\,(t,x)$), which implies SN(t).

C2) Assume $Comp\,(B',\ u,\ \alpha)$, then by Def.*3.2.4 (i.b)*, $Comp\,(\Pi\{B{,}B'\},\ Ap\,(t,u),\ \beta)$. Since $Ap\,(t,u) \to^* Ap\,(t',u)$, by induction we get $Comp\,(\Pi\{B{,}B'\},\ Ap\,(t',u),\ \beta)$. Then, by Def.*3.2.4 (i.b)*, $Comp\,(B,\ t',\ \alpha{\to}\beta)$.

C3) Since $\sigma = \alpha{\to}\beta$, we have to prove:

$$\forall\, u \in T(\mathcal{F}, \mathcal{X})\ [Comp\,(B\text{'},\ u,\ \alpha) \Rightarrow Comp\,(\Pi\{B{,}B\text{'}\},\ Ap\,(t,u),\ \beta)].$$

Since $Ap\,(t,u)$ is neutral of type β, by induction hypothesis, it is sufficient to prove that for all v' such that $Ap\,(t,u) \to_{\mathbf{R}} v'$, $Comp\,(\Pi\{B{,}B\text{'}\},\ v',\ \beta)$ holds. For this we apply induction on the length of the maximal derivation from u to its normal form (by Property (C1) we know that SN(u)).

Base: If u is a normal form, because t is neutral $Ap\,(t,u)$ reduces only inside t, so $Ap\,(t,u) \to_{\mathbf{R}} Ap\,(v,u)$ and $Comp\,(B,\ v,\ \alpha{\to}\beta)$ holds by assumption. Then, by Def.*3.2.4*, $Comp\,(\Pi\{B{,}B\text{'}\},\ Ap\,(v,u),\ \beta)$ holds.

Induction step: Consider all possible one step reductions from $Ap\,(t,u)$: In case $Ap\,(t,u) \to_{\mathbf{R}} Ap\,(v,u)$ we proceed as before. In case $Ap\,(t,u) \to_{\mathbf{R}} Ap\,(t,u')$, $Comp\,(\Pi\{B{,}B\text{'}\},\ Ap\,(t,u'),\ \beta)$ follows by induction hypothesis. And these are all possible cases, because $Ap\,(t,u)$ can not be a redex itself since t is neutral and the rewrite system is safe.

iii) $\sigma = \sigma_1{\cap}\cdots{\cap}\sigma_n$.

C1) By Def.*3.2.4 (i.c)*, $Comp\,(B,\ t,\ \sigma_1{\cap}\cdots{\cap}\sigma_n)$ implies $Comp\,(B,\ t,\ \sigma_i)$ for $1 \leq i \leq n$, and then by induction SN(t).

C2) As in the previous proof, $Comp\,(B,\ t,\ \sigma_i)$ holds for $1 \leq i \leq n$. By induction, for $1 \leq i \leq n$, $Comp\,(B,\ t',\ \sigma_i)$, then, by Def.*3.2.4 (i.c)*, also $Comp\,(B,\ t',\ \sigma_1{\cap}\cdots{\cap}\sigma_n)$.

C3) Using Lem.*2.2.5 (i)*, we obtain $B \vdash_{\mathcal{E}} t{:}\sigma_i$, and by theorem *2.2.8*, $B \vdash_{\mathcal{E}} v{:}\sigma_i$. Moreover, $Comp\,(B,\ v,\ \sigma_1{\cap}\cdots{\cap}\sigma_n)$ implies $Comp\,(B,\ v,\ \sigma_i)$ for $1 \leq i \leq n$. Then, by induction, $Comp\,(B,\ t,\ \sigma_i)$ for $1 \leq i \leq n$, and by Def.*3.2.4 (i.c)*, $Comp\,(B,\ t,\ \sigma_1{\cap}\cdots{\cap}\sigma_n)$.

In order to prove the Strong Normalization Theorem we shall prove a stronger property, for which we will need the following ordering.

Definition 3.2.7 Let $(\Sigma, \mathbf{R})$ be a CTRS. Let $>_{\mathbb{N}}$ denote the standard ordering on natural

numbers, $\trianglerighteq$ stand for the well-founded encompassment ordering, (i.e. $u \trianglerighteq v$ if $u \neq v$ and $u|_p = v^{\mathrm{R}}$ for some position $p \in u$ and replacement R), and *lex*, *mul* denote respectively the *lexicographic* and *multiset* extension of an ordering. Note that encompassment contains strict subterm.
We define the ordering $\gg$ on triples – consisting of a pair of natural numbers, a term, and a multiset of terms – as the object $((>_{\mathbb{N}}, >_{\mathbb{N}})_{lex}, \trianglerighteq, (\rightarrow_{\mathbf{R}} \cup \triangleright)_{mul})_{lex}$.

We will interpret the term u^{R} by the triple $\mathcal{I}(u^{\mathrm{R}}) = <(i,j), u, \{\mathrm{R}\}>$, where i is the maximal super-index of the function symbols belonging to u, j is the minimum of the differences $arity(F^i) - arity(F^i_j)$ such that F^i_j occurs in u, and $\{\mathrm{R}\}$ is the multiset $\{x^{\mathrm{R}} \mid x \in Var(u)\}$. These triples are compared in the ordering $\gg$.

When R is computable, then by Prop. C1 every t in the image of R is strongly normalizable, so $\rightarrow_{\mathbf{R}}$ is well-founded on the image of R. Also, because the union of the strict subterm relationship with a terminating rewrite relation is well-founded [12], the relation $(\rightarrow_{\mathbf{R}} \cup \triangleright)_{mul}$ is well-founded on $\{\mathrm{R}\}$. Hence, when restricted to computable replacements, $\gg$ is a well-founded ordering.

We use $\gg_n$ when we want to indicate that the nth element of the triple has decreased but not the n-1 first ones.

We would like to stress that we do not just interpret terms, but terms that are obtained by performing a replacement. This implies that when $t^{\mathrm{R}} = t'^{\mathrm{R}'}$, their interpretations are not necessarily equal.

We now come to the main theorem of this section, in which we show that for any typeable term and computable replacement R also the term t^{R} is computable. The strong normalization result then follows, using Prop.C1, for any typeable term t, taking for R the identity.

In the proof, the main idea is to write a term t^{R} like $t'^{\mathrm{R}'}$ (so they are equal as terms), where $t \trianglerighteq t'$, and R' is a computable extension of R. This is accomplished by taking a computable subterm v of t, to put a new variable z in its place and to define $\mathrm{R}' = \mathrm{R} \cup \{z \mapsto v\}$.

Property 3.2.8 Let t be such that $B \vdash_{\mathcal{E}} t{:}\sigma$, and R *be computable in B. Then there is a B' such that Comp (B', t^{R}, σ).*

Proof: By noetherian induction on $\gg$. Let $B = \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}$, and $\mathrm{R} = \{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}$. Because of part *(i.c)* of Def.*3.2.4*, we can restrict the proof to the case $\sigma \in \mathcal{T}_{-\omega}$. We distinguish the cases:

i) t is a neutral term. If t is a term-variable then the thesis follows trivially since R is computable. Let t be a non-variable term, so also t^{R} is neutral. If t^{R} is irreducible, then *Comp* (B', t^{R}, σ) holds by Prop.C3.
Otherwise, let $t^{\mathrm{R}} \rightarrow_{\mathbf{R}} w$ at position p. We will prove either *Comp* (B', t^{R}, σ) itself, or prove *Comp* (B', w, σ) and apply Prop.C3.

  a) $p = qp'$, $t|_q = x_i \in \mathcal{X}$, so the rewriting takes place in a subterm of t^{R} that is introduced by the replacement. Let z be a new term-variable.
  Take $\mathrm{R}' = \mathrm{R} \cup \{z \mapsto w|_q\}$, and note that $t^{\mathrm{R}}|_q \rightarrow_{\mathbf{R}} w|_q$ at position p'. Since $t^{\mathrm{R}}|_q \in \{\mathrm{R}\}$, and R is assumed to be computable, also *Comp* (B, $t^{\mathrm{R}}|_q$, σ_i) holds. So *Comp* (B, $w|_q$, σ_i) holds by Prop.C2, hence R' is computable in $B \cup \{z_i{:}\sigma_i\}$.
  Now, if the variable x_i has exactly *one* occurrence in t, then $t = t[z]_q$ modulo renaming of term-variables, and otherwise $t \trianglerighteq t[z]_q$. In the first case (since R contains a term that is rewritten to get R') we have $\mathcal{I}(t^{\mathrm{R}}) \gg_3 \mathcal{I}(t[z]_q^{\mathrm{R}'})$, and $\mathcal{I}(t^{\mathrm{R}}) \gg_2 \mathcal{I}(t[z]_q^{\mathrm{R}'})$ in the

second case. Both cases yield, by induction, $Comp\,(B', t[z]_q^{\mathrm{R}'}, \sigma)$ and note that $t[z]_q^{\mathrm{R}'} \equiv w$.

b) Now assume that p is a non-variable position in t. We analyze separatedly the cases:

1) $p \neq \Lambda$ (p is not the root position). Then $t \trianglerighteq t|_p$, hence $\mathcal{I}(t^{\mathrm{R}}) \gg_2 \mathcal{I}(t|_p^{\mathrm{R}})$, and $t|_p^{\mathrm{R}} = t^{\mathrm{R}}|_p$. Let τ be the type assigned to $t|_p$ in the derivation of $B \vdash_{\mathcal{E}} t{:}\sigma$, then $Comp\,(B, t^{\mathrm{R}}|_p, \tau)$ holds by induction.
Let z be a new variable, and $\mathrm{R}' = \mathrm{R} \cup \{z \mapsto t^{\mathrm{R}}|_p\}$, then R' is computable in $B \cup \{z{:}\tau\}$, and $B \cup \{z{:}\tau\} \vdash_{\mathcal{E}} t[z]_p{:}\sigma$. Now $t \trianglerighteq t[z]_p$, hence $\mathcal{I}(t^{\mathrm{R}}) \gg_2 \mathcal{I}(t[z]_p^{\mathrm{R}'})$, hence $Comp\,(B, t^{\mathrm{R}}, \sigma)$.

2) $p = \Lambda$. Then the possible cases for t are:

A) $t \equiv F(t_1, \ldots, t_n)$, where F is a defined symbol of arity n or $F \equiv Ap$ and $n = 2$, and at least one of the t_i is not a variable. Take $\mathrm{R}' = \{z_1 \mapsto t_1^{\mathrm{R}}, \ldots, z_n \mapsto t_n^{\mathrm{R}}\}$. Since $t \trianglerighteq t_i$, $\mathcal{I}(t^{\mathrm{R}}) \gg_2 \mathcal{I}(t_i^{\mathrm{R}})$. If $B \vdash_{\mathcal{E}} t_i{:}\sigma_i$, then by induction $Comp\,(B, t_i^{\mathrm{R}}, \sigma_i)$. Hence, R' is computable in $B \cup \{z_1{:}\sigma_1, \ldots, z_n{:}\sigma_n\}$. But $\mathcal{I}(t^{\mathrm{R}}) \gg_2 \mathcal{I}(F(z_1, \ldots, z_n)^{\mathrm{R}'})$, and $F(z_1, \ldots, z_n)^{\mathrm{R}'} = t^{\mathrm{R}}$ and $B \cup \{z_1{:}\sigma_1, \ldots, z_n{:}\sigma_n\} \vdash_{\mathcal{E}} F(z_1, \ldots, z_n){:}\sigma$. Hence $Comp\,(B, t^{\mathrm{R}}, \sigma)$.

B) $t \equiv F^k(z_1, \ldots, z_n)$ where $z_1, \ldots, z_n$ are different term-variables. (If $z_i = z_j$ for some $i \neq j$, we can reason as in part (i.a).) Then t^{R} must be an instance of the left hand side of a rule defining F^k: $t^{\mathrm{R}} = F^k(z_1, \ldots, z_n)^{\mathrm{R}} = F^k(\overrightarrow{\mathrm{C}}[\overrightarrow{M}], \overrightarrow{N}) \to_{\mathbf{R}} \mathrm{C}'[F^k(\overrightarrow{\mathrm{C}_1}[\overrightarrow{M}], \overrightarrow{N}), \ldots, F^k(\overrightarrow{\mathrm{C}_m}[\overrightarrow{M}], \overrightarrow{N}), \overrightarrow{M}, \overrightarrow{N}] = w$, where $\overrightarrow{\mathrm{C}}[\overrightarrow{M}]$, $\overrightarrow{N}$ are all terms in $\{R\}$, so are computable by hypothesis. Now we will deduce $Comp\,(B, w, \sigma)$ in three steps:

Step I: Let R' be the replacement that maps the left-hand side of the rewrite rule into t^{R}, so $\overrightarrow{x^{\mathrm{R}'}} = \overrightarrow{M}$. Since $\overrightarrow{M} \subseteq \overrightarrow{N}$, and all $\overrightarrow{N}$ are computable, also R' is computable. For every $1 \leq j \leq m$, F^k does not occur in $\overrightarrow{\mathrm{C}_j}$ (by definition of the general scheme), hence $\mathcal{I}(F^k(z_1, \ldots, z_n)^{\mathrm{R}}) \gg_1 \mathcal{I}(\mathrm{C}_j^{\mathrm{R}'})$, hence $\mathrm{C}_j^{\mathrm{R}'}$ is computable.

Step II: Let, for $1 \leq j \leq m$, R_j be the computable replacement such that $t^{\mathrm{R}_j} = F^k(\overrightarrow{\mathrm{C}_j}[\overrightarrow{x}], \overrightarrow{y})^{\mathrm{R}'}$. Since $\overrightarrow{\mathrm{C}} \triangleright_{mul} \overrightarrow{\mathrm{C}_j}$, and $\triangleright$ is closed under replacement, also $\overrightarrow{\mathrm{C}}^{\mathrm{R}'} \triangleright_{mul} \overrightarrow{\mathrm{C}_j}^{\mathrm{R}'}$, hence $\mathcal{I}(F^k(z_1, \ldots, z_n)^{\mathrm{R}}) \gg_3 \mathcal{I}(F^k(z_1, \ldots, z_n)^{\mathrm{R}_j})$, hence $F^k(z_1, \ldots, z_n)^{\mathrm{R}_j}$ is computable.

Step III: Let v be the term obtained by replacing, in the right hand side of the rule, the terms $F^k(\overrightarrow{\mathrm{C}_1}[\overrightarrow{M}], \overrightarrow{N}), \ldots, F^k(\overrightarrow{\mathrm{C}_m}[\overrightarrow{M}], \overrightarrow{N})$, $\overrightarrow{M}$, and $\overrightarrow{N}$ by fresh variables. Let R'' be the replacement such that $\mathrm{C}'[F^k(\overrightarrow{\mathrm{C}_1}[\overrightarrow{M}], \overrightarrow{N}), \ldots, F^k(\overrightarrow{\mathrm{C}_m}[\overrightarrow{M}], \overrightarrow{N}), \overrightarrow{M}, \overrightarrow{N}] = v^{\mathrm{R}''}$, then $t^{\mathrm{R}} \to_{\mathbf{R}} v^{\mathrm{R}''}$. Notice that above we have shown that R'' is computable. When F^j occurs in v, then by definition of the general scheme $j < k$, and therefore $\mathcal{I}(t^{\mathrm{R}}) \gg_1 \mathcal{I}(v^{\mathrm{R}''})$, hence $v^{\mathrm{R}''}$ is computable, and since $w = v^{\mathrm{R}''}$, we get $Comp\,(B, w, \sigma)$.

C) $t = Ap\,(z_1, z_2)$ where $z_1, z_2 \in \mathcal{X}$. By assumption, z_1^{R} and z_2^{R} are computable, and since t is well-typed, z_1 must have an arrow type. Then, by Def.*3.2.4*, $Ap\,(z_1^{\mathrm{R}}, z_2^{\mathrm{R}})$ is computable. But $Ap\,(z_1^{\mathrm{R}}, z_2^{\mathrm{R}})$ is the same as $Ap\,(z_1, z_2)^{\mathrm{R}}$.

ii) Let $t \equiv F_n(t_1, \ldots, t_n)$. Again we distinguish two cases:

a) Assume that at least one of the t_i is not a term-variable. Since $t \trianglerighteq t_i$ (for $1 \leq i \leq n$), by induction there exist B', σ_i such that $Comp\,(B', t_i, \sigma_i)$, and also the replacement $\mathrm{R}' =$

$\{z_1 \mapsto t_1, \ldots, z_n \mapsto t_n\}$ is computable. Since $t \trianglerighteq F_n(z_1, \ldots, z_n)$, we have $\mathcal{I}(t^{\mathrm{R}}) \gg \mathcal{I}(t^{\mathrm{R}'})$, and $t^{\mathrm{R}'}$ is computable by induction. Note that $t^{\mathrm{R}'} = t^{\mathrm{R}}$.

b) All t_i are term-variables. Since $B \vdash_{\mathcal{E}} t{:}\sigma$, by Lem.*2.2.5 (iii)* there exist $\alpha \in \mathcal{T}_{\cap}$, $\beta \in \mathcal{T}_{-\omega}$ such that $\sigma = \alpha \rightarrow \beta$. For all u such that *Comp* (B", u, α), we have to prove *Comp* ($\Pi\{B', B"\}$, *Ap* (t^{R}, u), β). Since *Ap* (t^{R}, u) is neutral, by Prop.C3, it is sufficient to prove *Comp* ($\Pi\{B', B"\}$, t', β) for all t' such that *Ap* (t^{R}, u) $\rightarrow_{\mathbf{R}} t'$. This will be proved by induction on the sum of the maximal length of the derivations out of u and out of R. Note that since u and R are computable, by Prop.C1, SN(u) and SN(R).

Base: If u and R are in normal form, the only reduction step out of *Ap* (t^{R}, u) could be: *Ap* ($F_n(z_1, \ldots, z_n)^{\mathrm{R}}, u$) $\rightarrow_{\mathbf{R}} t' \equiv F_{n+1}(z_1^{\mathrm{R}}, \ldots, z_n^{\mathrm{R}}, u)$. Then, since $\mathcal{I}(t^{\mathrm{R}}) \gg_1 \mathcal{I}(F_{n+1}(z_1^{\mathrm{R}}, \ldots, z_n^{\mathrm{R}}, u))$, t' is computable.

Induction step: If the reduction step out of *Ap* (t^{R}, u) takes place inside u or inside t^{R} (in the last case it must be inside R since the rewrite system is safe) then t' is computable by induction.

If *Ap* ($F_n(z_1, \ldots, z_n)^{\mathrm{R}}, u$) $\rightarrow_{\mathbf{R}} t' \equiv F_{n+1}(z_1^{\mathrm{R}}, \ldots, z_n^{\mathrm{R}}, u)$, then we proceed as in the base case. ∎

Theorem 3.2.9 Strong Normalization Theorem. *If* $(\Sigma, \mathbf{R})$ *is typeable in* $\vdash_{\mathcal{E}}$ *and safe, then any typeable term is strongly normalizable with respect to* $\mathbf{R}$.

Proof: From Prop. *3.2.8* and C1, taking R such that $x^{\mathrm{R}} = x$. ∎

4 Final remarks

The type assignment system defined in this paper for CTRS is undecidable: it is feasible to show that for every lambda term typeable in $\vdash_{\lambda\cap-\omega}$ there exists a term in ACL, obtained by bracket-abstraction, that is typeable as well, and vice versa. However, if we restrict the system as in [2], then typeability is decidable, and since the Strong Normalization Theorem we proved in Section *3.2* is still valid in this weaker system, any typeable CTRS satisfying the general scheme (as given in Def.*3.2.2*) is terminating on typeable terms.

References

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

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

[3] S. van Bakel. Essential Intersection Type Assignment. In R.K. Shyamasunda, editor, *Proceedings of FST&TCS '93. 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, *LNCS* 761, pages 13–23, Bombay, India, 1993.

[4] 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*, *LNCS* 664, pages 29–44, Utrecht, the Netherlands, 1993.

[5] S. van Bakel. Principal type schemes for the Strict Type Assignment System. *Logic and Computation*, 3(6):643–670, 1993.

[6] 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, *LNCS* 664, pages 60–74, Utrecht, the Netherlands, 1993.

[7] 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. 20th International Colloquium on Automata, Languages and Programming*, *LNCS* 700, pages 657–668, Lund, Sweden, 1993.

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

[9] 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.

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

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

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

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

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

[15] J.Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

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

[17] J.P. Jouannaud and M. Okada. Executable higher-order algebraic specification languages. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science*, pages 350–361, 1991.

[18] 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.

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

[20] F. Pfenning. Partial Polymorphic Type Inference and Higher-Order Unification. In *Proceedings of the 1988 ACM conference on LISP and Functional Programming Languages*, pages 153–163, 1988.

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

[22] W.W. Tait. Intensional interpretation of functional of finite types. *Journal of Symbolic Logic*, 32, 1967.

[23] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the conference on Functional Programming Languages and Computer Architecture*, *LNCS* 201, pages 1–16.