Rewrite Systems with Abstraction and β-rule: Types, Approximants and Normalization

Steffen van Bakel [1]   Franco Barbanera [2]   Maribel Fernández [3]

[1] Department of Computing, Imperial College, 180 Queens Gate, London SW7 2BZ, svb@doc.ic.ac.uk

[2] Dipartimento di Informatica, Università degli Studi di Torino, Corso Svizzera 185, 10149 Torino, Italia, barba@di.unito.it

[3] DMI - LIENS (CNRS URA 1327), Ecole Normale Supérieure, 45, rue d'Ulm, 75005 Paris, France, maribel@ens.fr

Abstract

In this paper we define and study intersection type assignment systems for first-order rewriting extended with application, λ-abstraction, and β-reduction ($\mathrm{TRS}+\beta$). One of the main results presented is that, using a suitable notion of approximation of terms, any typeable term of a $\mathrm{TRS}+\beta$ that satisfies a general scheme for recursive definitions has an approximant of the same type. From this result we deduce, for different classes of typeable terms, a head-normalization and a normalization theorem.

Introduction

Lambda Calculus (LC) and *Term Rewriting Systems* (TRS) are two computational paradigms that have been thoroughly investigated because of their adaptness to modeling fundamental aspects of computing. In the past, these fields were often studied separately. This enabled a better understanding of particular features of the actual practice of computing, by isolating and abstracting those from the wider context in which they are usually found.

Recently, a greater interest has developed for the study of a combination of these two formalisms. This combination is interesting not only from the point of view of programming languages, but also from a more theoretical side. Indeed, such a combination allows to investigate the interactions of the different aspects of computing, and enables either to develop new computational methods and paradigms, or to better understand and improve the actual computing practice.

Various combinations of these two formalisms have been studied extensively in recent years, both in typed and untyped contexts. In the absence of types, the two systems do not interact in a very smooth manner. For instance, in [21] Klop showed that confluence, a highly desirable property in practice, is lost if a surjective pairing operation is added to the untyped LC. In [16], Dougherty provided some restrictions on terms, thus ensuring that properties that LC and TRS both possess can be preserved when these systems are combined.

Instead, in the presence of types the combination proved to be much safer. Type disciplines provide an environment in which rewrite rules and β-reduction can be combined without loss of their useful properties (for example, strong normalization and confluence are preserved under the combination of typed LC and first-order TRS). This is supported by a number of results for a broad range of type systems and calculi [12, 13, 14, 20, 23, 9], but still lacks evidence in order to be completely accepted in its full generality. More specifically, all the systems studied in the papers mentioned above have *explicit type disciplines* (also called *à la Church*), i.e. type disciplines where terms come together with types and, hence, each term has exactly *one* type. When types are considered to be functional properties of terms, this way of using types forces to prove a property of a term at the same time that term is constructed.

Type disciplines à la Church, however, are not the only ones used within the setting of programming languages. In some languages it is possible to write type-free programs and construct their functional characterizations at a later stage, i.e. *to assign types to them*. This sort of type discipline (also called *à la Curry*) is fruitfully exploited in several functional programming languages, like ML [18] and Miranda[1] [26]. So, before stating in full generality that type disciplines provide a good environment for a smooth interaction of computing modeled by LC and TRS, also disciplines of type assignment have to considered.

Type assignment disciplines were widely investigated in contexts of LC, but very little was done in this direction for TRS. The system presented in [8], for example, combines a type assignment system for LC with TRS that are typed à la Church. This means that [8] did not present really a type assignment environment for LC and TRS, but rather a way to embed explicitly typed TRS in a type assignment discipline for LC.

Recently, however, new ideas and results have come in aid to the search for a type assignment environment for both LC and TRS. For example, in [3] a notion of type assignment for TRS has been developed. In particular, that paper considered systems in which it is possible to make hypotheses about the functional characterization of the function symbols in the signature of the TRS. The soundness of these hypotheses should then be checked against the structure of the rewrite rules, and, using these hypotheses, types can be derived for terms. This type assignment system enjoys interesting normalization properties [5, 6].

Having now a good notion of type assignment at hand for TRS as well, in the present paper we are going to define a type assignment environment for the combination of TRS and LC. To our knowledge, this is the first presentation of a type assignment system where both formalisms are treated in the same way. We hope that the design of such system will provide evidence for the claim stated above, i.e. that type disciplines are a good setting for sound interaction of computational paradigms. In fact, we already have positive results concerning the normalization properties of the combined system.

More precisely, in this paper we present an *intersection* type assignment system with ω and sorts (i.e. constant types) for TRS extended with application, λ-abstraction and β-reduction. This system is an extension of the type assignment systems for TRS presented in [3]. It exploits the power and generality of intersection types with ω (see, e.g., [11, 2, 4]), managing to type broad and meaningful sets of terms and rewrite rules. We will show that the normalization properties of LC and TRS are preserved in our system.

It is well-known that intersection type systems for LC are useful not only in the study of normalization properties, but also in the study of the semantics of the LC (see, e.g., [11, 2]). The notion of intersection type assignment for TRS developed in [3, 5, 6] enables the study of the relation between semantics of reduction and type assignment in the framework of TRS. In [7] the notion of *approximant* and the related approximation model defined by Thatte [25] are used to show that every type that can be assigned to a term, can also be assigned to one of its approximants (provided the TRS satisfies certain conditions). In this sense, the type assigned to the term gives finitary information about the reduction process. This paper presents that result for the combination of LC and TRS, but because of the presence of abstraction, the applied technique differs significantly.

On the other hand, the use of intersection types models in a very elegant way the distribution of the actual argument of a function during the computation. That more than one type can be assigned to a term corresponds, in this setting, to the fact that an operand is used more than once during reduction, even at a later point than just during the contraction of the redex at hand.

In the present paper we define approximants for the combination of TRS and LC. This notion of approximant is a combination of similar definitions given by Thatte [25] and Wadsworth [27] for TRS and LC, respectively. We show that also in the combination of TRS and LC every typeable term has an approximant of the same type. This *Approximation Theorem* will be proved for systems that use

[1]Miranda is a trade mark of Research Software LTD.

recursion in a restricted way: we will consider rewrite rules that satisfy a variant of the *general schemes* defined in [6, 7]. We will then use this result to prove a head-normalization and a normalization theorem for different classes of typeable terms. Worth noting is that, applying the technique used in [8, 5] it is also possible to prove that if the type constant ω is not in the type system, then typeable terms are strongly normalizable; we will not discuss that result for the calculus presented here, because of the great similarities with those two papers.

This paper is organized as follows: In Section *1* we define TRS with application, λ-abstraction and β-reduction (TRS+β), and in Section *2* the type assignment system for $\mathrm{TRS}+\beta$. In Section *3* we define approximants and prove the approximation theorem, and in Section *4* we prove the normalization theorems. Section *5* contains the conclusions.

1 Term Rewriting Systems with β-reduction rule

In this section we present a combination of untyped Lambda Calculus with untyped Algebraic Rewriting, obtained by extending first-order TRS with notions of application and abstraction, and a β-reduction rule. We can look at such calculi also as extensions of the Curryfied Term Rewriting Systems ($\mathbb{C}$TRS) considered in [3, 5, 6], by adding λ-abstraction and a β-reduction rule. We assume the reader to be familiar with LC [10] and refer to [22, 15] for rewrite systems.

Definition 1.1 An *alphabet* or *signature* Σ consists of:

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

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

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

Definition 1.2 *i)* The set $T(\mathcal{F},\mathcal{X})$ of *terms* is defined inductively:

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

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

c) If $t \in T(\mathcal{F},\mathcal{X})$, and $x \in \mathcal{X}$, then $\lambda x.t \in T(\mathcal{F},\mathcal{X})$.

We will consider terms modulo α-conversion.

A *context* is a term with a hole, and it is denoted as usual by C[ ].

ii) *a)* A *neutral* term is a term not of the form $\lambda x.t$.

b) A *lambda term* is a term not containing function symbols.

The set of *free variables* of a term t is defined as usual, and denoted by $FV(t)$.

To denote a term-substitution, we use capital characters like 'R', instead of Greek characters like 'σ', which will be used to denote types. Sometimes we use the notation $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. We write t^{R} for the result of applying the term-substitution R to t.

In the next definition, we present a notion of rewriting on $T(\mathcal{F},\mathcal{X})$ that is defined through rewrite rules together with a β-reduction rule.

Definition 1.3 (Reduction) *i)* A *rewrite rule* is a pair (l, r) of terms. Often, a rewrite rule will get a name, e.g. **r**, and we write $l \rightarrow_{\mathbf{r}} r$. Three conditions are imposed: l is not a variable or an abstraction $\lambda x.t$, $FV(r) \subseteq FV(l)$, and Ap does not occur in l.

The *patterns* of a rewrite rule $F(t_1, \ldots, t_n) \rightarrow_{\mathbf{r}} r$ are the terms t_i, $1 \leq i \leq n$, such that either t_i is not a variable, or t_i is variable and there is a $1 \leq i \neq j \leq n$ such that $t_i = t_j$.

ii) On terms we define the usual notion of β-reduction: $Ap(\lambda x.t, u) \rightarrow_\beta t^{\{x \mapsto u\}}$.

iii) A rewrite rule $l \rightarrow_{\mathbf{r}} r$ determines a set of *rewrites* $l^{\mathrm{R}} \rightarrow r^{\mathrm{R}}$ for all term-substitutions R. The left hand side l^{R} is called a *redex*, the right hand side r^{R} its *contractum*. Likewise, for any t and u,

$Ap\,(\lambda x.t, u) \rightarrow_\beta t^{\{x \mapsto u\}}$ is also a rewrite; $Ap\,(\lambda x.t, u)$ is called a redex, and $t^{\{x \mapsto u\}}$ its contractum.

iv) A redex t may be substituted by its contractum t' inside a context C[ ]; this gives rise to *rewrite steps* C[t] $\rightarrow$ C[t']. Concatenating rewrite steps we have *rewrite sequences* $t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots$. If $t_0 \rightarrow \cdots \rightarrow t_n$ ($n \geq 0$) we also write $t_0 \rightarrow^* t_n$, and $t_0 \rightarrow^+ t_n$ if $t_0 \rightarrow^* t_n$ in one step or more.

Definition 1.4 A *Term Rewriting System with β-reduction rule* (TRS+β) is defined by a pair $(\Sigma, \mathbf{R})$ of an alphabet Σ and a set **R** of rewrite rules.

Note that in contrast with $@$TRS, the rewrite rules considered in this paper can contain λ-abstractions. We take the view that in a rewrite rule a certain symbol is defined.

Definition 1.5 In a rewrite rule $F(t_1, \ldots, t_n) \rightarrow_{\mathbf{r}} r$, F is called *the defined symbol* of **r**, and **r** is said to *define* F. F is *a defined symbol*, if there is a rewrite rule that defines F, and $Q \in \mathcal{F}$ is called a *constructor* if Q is not a defined symbol. (Notice that Ap is never a defined symbol.)

Example 1.6 The following is a set of rewrite rules that defines the functions append and map on lists and establishes the associativity of append. The function symbols nil and cons are constructors.

$$\begin{array}{ll}
\mathsf{append}\,(\mathsf{nil}, l) & \rightarrow l \\
\mathsf{append}\,(\mathsf{cons}\,(x, l), l') & \rightarrow \mathsf{cons}\,(x, \mathsf{append}\,(l, l')) \\
\mathsf{append}\,(\mathsf{append}\,(l, l'), l'') & \rightarrow \mathsf{append}\,(l, (\mathsf{append}\,(l', l''))) \\
\mathsf{map}\,(\lambda x.t, \mathsf{nil}) & \rightarrow \mathsf{nil} \\
\mathsf{map}\,(\lambda x.t, \mathsf{cons}\,(y, l)) & \rightarrow \mathsf{cons}\,(Ap\,(\lambda x.t, y), \mathsf{map}\,(\lambda x.t, l))
\end{array}$$

Since variables in TRS+β can be substituted by λ-expressions, we obtain the usual functional programming paradigm, extended with definitions of operators and data structures.

Definition 1.7 Let $(\Sigma, \mathbf{R})$ be a TRS+β.

i) A term is in *normal form* if it contains no redex.

ii) A term t is in *head normal form* if for all t' such that $t \rightarrow^* t'$:

a) t' is not itself a redex, and

b) if $t' = Ap\,(v, u)$, then v is in head normal form,

c) if $t' = \lambda x.u$, then u is in head normal form.

Note that t itself cannot be a redex.

iii) A term is *(head) normalizable* if it can be reduced to a term in (head) normal form; a term is *strongly normalizable* if all the rewrite sequences starting with t are finite.

iv) $(\Sigma, \mathbf{R})$ is *strongly normalizing* (*normalizing, head-normalizing*) if every term is.

v) $(\Sigma, \mathbf{R})$ is *confluent* if for all t such that $t \rightarrow^* u$ and $t \rightarrow^* v$, there exists s such that $u \rightarrow^* s$ and $v \rightarrow^* s$.

Example 1.8 Take the TRS+β

$$\begin{array}{ll}
F\,(G, x) & \rightarrow A\,(H) \\
B\,(C) & \rightarrow G \\
H & \rightarrow H
\end{array}$$

then the term $F\,(B\,(C), \lambda y.Ap\,(G, y))$ is not a redex. It is not a head-normal form either, since it reduces to $F\,(G, \lambda y.Ap\,(G, y))$ which is a redex. This term reduces to $A\,(H)$ that is a head-normal form (it rewrites only to itself, so it will never become a redex). Another term in head-normal form is, for instance, $\lambda y.Ap\,(y, B\,(C))$.

Our definition of head normal form is an extension to rewrite systems with Ap of the notion of root stable form defined in [1]. Note that the head of a term of the form $Ap\,(v,u)$ is in v, since we think of Ap as an invisible symbol.

2 Type assignment in TRS+β

Type assignment systems are formal systems defined by specifying a set of terms, a set of types, and a set of type assignment rules. In this section we define a type assignment system for TRS+β, that can be seen as an extension of the intersection type assignment system presented in [3]. The LC-fragment of our type assignment system corresponds directly to the system presented in [4].

We assume the reader to be familiar with intersection type assignment systems; we refer to [11, 2, 3] for details.

2.1 Types

As in [6], we will use strict intersection types over type-variables and *sorts* (constant types). We assume that ω is the same as an intersection over zero elements: if $n = 0$, then $\sigma_1\cap\cdots\cap\sigma_n = \omega$; so in an intersection $\sigma_1\cap\cdots\cap\sigma_n$, no σ_i can be ω. Moreover, intersection types (so also ω) occur in strict types only in the left-hand side of an arrow type.

Definition 2.1 (Types) *i)* $\mathcal{T}_s$, the set of *strict types*, and $\mathcal{T}_S$, the set of *strict intersection types*, are defined by mutual induction:

a) all type-variables $\varphi_0, \varphi_1, \ldots \in \mathcal{T}_s$, and all sorts $s_0, s_1, \ldots \in \mathcal{T}_s$,

b) if $\tau \in \mathcal{T}_s$ and $\sigma \in \mathcal{T}_S$, then $\sigma\rightarrow\tau \in \mathcal{T}_s$.

c) If $\sigma_1, \ldots, \sigma_n \in \mathcal{T}_s$ $(n \geq 0)$, then $\sigma_1\cap\cdots\cap\sigma_n \in \mathcal{T}_S$.

ii) On $\mathcal{T}_S$, 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) $\forall\, 1 \leq i \leq n$ $(n \geq 0)$ $[\sigma \leq \sigma_i] \Rightarrow \sigma \leq \sigma_1\cap\cdots\cap\sigma_n$.

c) $\sigma \leq \tau \leq \rho \Rightarrow \sigma \leq \rho$.

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

iii) On $\mathcal{T}_S$, the relation $\sim$ is defined by: $\sigma \sim \tau \iff \sigma \leq \tau \leq \sigma$.

Notice that $\mathcal{T}_s$ is a proper subset of $\mathcal{T}_S$, and that $\sigma\rightarrow(\tau\cap\rho)$ is not a type in $\mathcal{T}_S$, and neither is $\sigma\rightarrow\omega$.

2.2 Type assignment

Before coming to the definition of type assignment, we first introduce the notions of basis and operations on types.

Definition 2.2 (Statements and Bases) *i)* A *statement* is an expression of the form $t{:}\sigma$, where $t \in T(\mathcal{F}, \mathcal{X})$ and $\sigma \in \mathcal{T}_S$. t is the *subject* and σ the *predicate* of $t{:}\sigma$.

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

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 (non-empty) set of all statements about x that occur in $B_1 \cup \cdots \cup B_n$.

iv) We extend $\leq$ and $\sim$ to bases by: $B \leq B'$ if and only if for every $x{:}\sigma' \in B'$ there is an $x{:}\sigma \in B$ such that $\sigma \leq \sigma'$, and $B \sim B' \iff B \leq B' \leq B$.

Notice that if $n = 0$, then $\Pi\{B_1, \ldots, B_n\} = \emptyset$. We will often write $B, x{:}\sigma$ for the basis $\Pi\{B, \{x{:}\sigma\}\}$, when x does not occur in B.

The notion of type assignment of this paper uses three operations on types. The first is called *substitution* and is the operation that instantiates a type (i.e. that replaces type variables by types). It is defined as a mapping from strict types to strict types. The operation of *expansion* replaces types by the intersection of a number of copies of that type. Substitutions and expansions can be extended to bases in the natural way. The last operation is that of *lifting*, that replaces a type by a larger type, and a basis by a smaller basis, in the sense of $\leq$. (Although sound type assignment could be defined with less – or less complicated – operations, more terms are typeable with these three, and more expressive types can be assigned.)

The operations of substitution, expansion and lifting can be composed to form *chains* of operations. In other words, the set of chains is the smallest set containing expansions, substitutions and liftings, and closed under composition. See [3] for the formal definition of chain and of the above operations on types. We will use the notation $C(\sigma)$ for the application of a chain C of operations to the type σ. Given a basis-type pair $\langle B, \sigma\rangle$ and a chain C of operation on types, we define $C(\langle B, \sigma\rangle) = \langle \{x{:}C(\tau) \mid x{:}\tau \in B\}, C(\sigma)\rangle$.

In the definition of the type assignment system for TRS$+\beta$, we assume that there is an environment that assigns types to function symbols.

Definition 2.3 (Environments) A mapping $\mathcal{E}\colon \mathcal{F} \cup \{Ap\} \to \mathcal{T}_s$ is called an *environment* if $\mathcal{E}(Ap) = (\varphi_1{\to}\varphi_2){\to}\varphi_1{\to}\varphi_2$.

The use of an environment introduces a notion of polymorphism into our type assignment system. The environment returns the 'principal type' for a function symbol; this symbol will be used with types that are 'instances' of its principal type, obtained by applying operations of *substitution, expansion,* and *lifting*.

We specify how to type terms and rewrite rules through the presentation of type assignment rules.

Definition 2.4 *i) Type assignment on terms* (with respect to $\mathcal{E}$) is defined by the following natural deduction system (where all types displayed are in $\mathcal{T}_s$, except for $\sigma_1, \ldots, \sigma_n$ in rule ($\to$E)). Note the use of a chain of operations in rule ($\to$E).

$$(\to\mathrm{I}):\ \frac{\begin{array}{c}[x{:}\sigma]\\ \vdots\\ t{:}\tau\end{array}}{\lambda x.t{:}\sigma{\to}\tau}\ (a) \qquad\qquad (\to\mathrm{E}):\ \frac{t_1{:}\sigma_1 \quad \ldots \quad t_n{:}\sigma_n}{F(t_1, \ldots, t_n){:}\sigma}\ (b)$$

$$(\leq):\ \frac{x{:}\sigma \qquad \sigma \leq \tau}{x{:}\tau} \qquad\qquad (\cap\mathrm{I}):\ \frac{t{:}\sigma_1 \quad \ldots \quad t{:}\sigma_n}{t{:}\sigma_1{\cap}\cdots{\cap}\sigma_n}\ (n \geq 0)$$

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

(b)) If $F \in \mathcal{F} \cup \{Ap\}$, and there exists a chain C such that $\sigma_1{\to}\cdots{\to}\sigma_n{\to}\sigma = C(\mathcal{E}(F))$.

ii) We write $B \vdash_{\mathcal{E}} t{:}\sigma$ if and only if $t{:}\sigma$ is derivable from the basis B using the above rules.

Notice that, by rule ($\cap$I), $B \vdash_{\mathcal{E}} t{:}\omega$ for all terms t and bases B. We will call those terms *typeable* that can be assigned a type different from ω.

To ensure the subject reduction property, as in [3], type assignment on rewrite rules will be defined using the notion of principal pair for a typeable term.

Definition 2.5 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 $C(\langle P, \pi\rangle) = \langle B, \sigma\rangle$.

We define now type assignment on rewrite rules. The typeability of rules ensures consistence with respect to the environment used in the type assignment for terms.

Definition 2.6 *i)* We say that $l \to r \in \mathbf{R}$ with defined symbol F *is typeable with respect to* $\mathcal{E}$, if there are P, and $\pi \in \mathcal{T}_{\mathrm{S}}$ 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 occurrences of F are typed with $\mathcal{E}\,(F)$.

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

From now on, we will only consider $\mathrm{TRS}{+}\beta$ that are typeable with respect to a given environment $\mathcal{E}$.

Using a combination of the techniques used in [3, 4], it is possible to show that the three operations (substitution, expansion, and lifting) are sound on typed terms. . That is, we have:

Theorem 2.7 *If* $B \vdash_{\mathcal{E}} t{:}\sigma$ *then, for every* C *such that* $C\,(\langle B, \sigma \rangle) = \langle B', \sigma' \rangle$, $B' \vdash_{\mathcal{E}} t{:}\sigma'$. ■

Then it is possible to prove that type assignment is closed under reduction.

Theorem 2.8 (Subject Reduction) *If* $B \vdash_{\mathcal{E}} t{:}\sigma$, *and* $t \to t'$, *then* $B \vdash_{\mathcal{E}} t'{:}\sigma$.

Proof: The case of a β-reduction follows from the fact that it is possible to prove that, for every t, u, $B \vdash_{\mathcal{E}} (\lambda x.t)u{:}\sigma \iff B \vdash_{\mathcal{E}} t^{\{x \mapsto u\}}{:}\sigma$. The case of a rewriting can be proved using the same technique as in [3]. ■

3 Approximation results

In this section we define approximants of the terms of our calculus, and prove the approximation theorem (any typeable term has an approximant with the same type). Our definition of approximants is inspired by the one given by Wadsworth [27] for the LC, and the notion of approximants for Term Rewriting Systems given by Thatte [25], which in turn is based on the definition of Ω-normal forms of Huet and Lévy [19]. As in those papers, in the sequel we will only consider confluent systems.

We start by adding a special symbol $\bot$ (*bottom*) to the language.

Definition 3.1 The set $T(\mathcal{F}, \mathcal{X}, \bot)$ of *partial terms* is defined in the same way as the set $T(\mathcal{F}, \mathcal{X})$, by adding to Definition *1.1* the case

iv) A special symbol $\bot$.

and to Definition *1.2-(i)* the case

i) *d)* $\bot \in T(\mathcal{F}, \mathcal{X}, \bot)$.

Notice that $\bot \notin \mathcal{F}$ and $\bot \notin \mathcal{X}$.

To define type assignment on $T(\mathcal{F}, \mathcal{X}, \bot)$, the type assignment rules (Definition *2.4*) need not be changed, it suffices that terms are allowed to be in $T(\mathcal{F}, \mathcal{X}, \bot)$. Since $\bot \notin \mathcal{F} \cup \{Ap\}$, $\bot$ can only be typed with ω or appear in subterms that are typed with ω (i.e. for which the derivation rule ($\cap$I) is used with $n = 0$).

We define the following relation on partial terms.

Definition 3.2 *i)* $t \sqsubseteq u$ is inductively defined by:

a) For every $u \in T(\mathcal{F}, \mathcal{X}, \bot)$, $\bot \sqsubseteq u$.

b) For every $t \in T(\mathcal{F}, \mathcal{X}, \bot)$, $t \sqsubseteq t$.

c) $t \sqsubseteq u \Rightarrow \lambda x.t \sqsubseteq \lambda x.u$.

d) $\forall 1 \leq i \leq n\, [t_i \sqsubseteq u_i] \Rightarrow F(t_1, \ldots, t_n) \sqsubseteq F(u_1, \ldots, u_n)$, for $F \in \mathcal{F} \cup \{Ap\}$.

ii) We write $t \uparrow u$ (and say that t and u *are compatible*) if there is a $s \in T(\mathcal{F}, \mathcal{X}, \bot)$ such that $t \sqsubseteq s$ and $u \sqsubseteq s$. We write $t \uparrow T$ if there is a $s \in T$ such that $t \uparrow s$.

For the type assignment system of Section *2*, extended to $T(\mathcal{F}, \mathcal{X}, \bot)$, the following property holds:

Lemma 3.3 $B \vdash_{\mathcal{E}} t{:}\sigma \;\&\; t \sqsubseteq u \;\Rightarrow\; B \vdash_{\mathcal{E}} u{:}\sigma$. ■

We will now develop the notion of approximant of a term with respect to a given TRS+β $(\Sigma, \mathbf{R})$. As mentioned above, this definition is a combination of the notion of approximant for terms in LC [27] and that for terms in TRS [25]. A particular difference with those definitions is that our definition is 'static', whereas both other notions were defined as normal forms with respect to an extended notion of reduction, adding, for example, for LC the reduction rule $\bot M \to \bot$. This approach would not be appropriate for our paper, because, to name just one problem, we would not be able to prove a subject reduction result for such a notion of reduction. Instead, we will recursively replace redexes by $\bot$. While doing this, it can be that a term is created that itself is not a redex, but looks like one, in the sense that is compatible to a left-hand side of a rewrite rule (where variables are replaced by $\bot$). Also such 'possible redexes' will be replaced by $\bot$.

We will use the symbol $\bot$ also for the term-substitution that replaces variables by $\bot$: $\bot = \{x \mapsto \bot \mid x \in \mathcal{X}\}$. Also, $Lhs^{\bot} = \{l^{\bot} \mid \exists r \; [l \to r \in \mathbf{R}]\}$. First we will define direct approximants of terms by replacing by $\bot$ potential redexes. The set of approximants of a term will then be defined by taking the downward closure of the direct approximants of all its reducts.

Definition 3.4 $\mathcal{DA}(t)$, the *direct approximant of t* is defined by cases:

i) $t = x$. $\mathcal{DA}(x) = x$.

ii) $t = F(t_1, \ldots, t_n)$, $F \in \mathcal{F}$; let, for $1 \leq i \leq n$, $a_i = \mathcal{DA}(t_i)$.
$\mathcal{DA}(t) = \bot$, if $F(a_1, \ldots, a_n) \uparrow Lhs^{\bot}$; otherwise, $\mathcal{DA}(t) = F(a_1, \ldots, a_n)$.

iii) $t = Ap(t_1, t_2)$; let $a_1 = \mathcal{DA}(t_1)$, and $a_2 = \mathcal{DA}(t_2)$.
$\mathcal{DA}(t) = \bot$, if $a_1 = \bot$, or $a_1 = \lambda x.a'$; otherwise, $\mathcal{DA}(t) = Ap(a_1, a_2)$.

iv) $t = \lambda x.t'$; let $a = \mathcal{DA}(t')$. $\mathcal{DA}(t) = \bot$, if $a = \bot$; otherwise, $\mathcal{DA}(t) = \lambda x.a$.

Example 3.5 Take the TRS+β of Example *1.8*

$$\begin{array}{ll} F(G,x) & \to A(H) \\ B(C) & \to G \\ H & \to H \end{array}$$

and consider again the term $F(B(C), \lambda y.Ap(G,y))$. Since $B(C)$ is a redex, in particular it is compatible with a left-hand side (begin that term itself), so $\mathcal{DA}(B(C)) = \bot$. Since $F(\bot, \lambda y.Ap(G,y))$ is compatible to $F(G,\bot)$, we get that $\mathcal{DA}(F(B(C), \lambda y.Ap(G,y))) = \bot$.

Also, $\mathcal{DA}(\lambda y.Ap(y, B(C))) = \lambda y.Ap(y, \bot)$, and $\mathcal{DA}(\lambda y.Ap(B(C), y)) = \bot$.

Definition 3.6 *i)* $\mathcal{DA}$, the set of *approximate normal forms* is defined as

$$\{a \in T(\mathcal{F}, \mathcal{X}, \bot) \mid \mathcal{DA}(a) = a\}.$$

ii) $\mathcal{A}(t)$, the *set of approximants of t*, is defined by:

$$\mathcal{A}(t) = \{a \in \mathcal{DA} \mid \exists u \; [t \to^* u \;\&\; a \sqsubseteq \mathcal{DA}(u)]\}.$$

Example 3.7 Take again the TRS+β of Example *1.8*. Then

$$F(B(C), \lambda y.Ap(G,y)) \to F(G, \lambda y.Ap(G,y)) \to A(H) \to A(H) \to \cdots.$$

Then $\mathcal{DA}(F(B(C), \lambda y.Ap(G,y))) = \mathcal{DA}(F(G, \lambda y.Ap(G,y))) = \bot$, $\mathcal{DA}(A(H)) = A(\bot)$, so $\mathcal{A}(F(B(C), \lambda y.Ap(G,y)))$ $= \{\bot, A(\bot)\}$.

Instead, $\mathcal{A}(F(H, \lambda y.Ap(G,y))) = \{\bot\}$.

Lemma 3.8 *i) If t is irreducible, then $t \in \mathcal{DA}$.*

ii) t is in head-normal form if and only if there exists $a \in \mathcal{DA}$ such that $a \neq \bot$, and $a \sqsubseteq t$. ■

The approximation theorem does not hold for arbitrary typeable TRS+β: let t be a term typeable by φ, then the rewrite rule $t \to t$ is typeable, but the only approximant of t is $\bot$, which has only type ω. Therefore, as in [5, 8, 6], we will control the use of recursion in the rewrite rules by imposing syntactical restrictions inspired by the general scheme of Jouannaud and Okada [20].

The recursive schemes defined in [5, 8] ensure strong normalization of typeable terms when the constant ω is not included in the type assignment system. But in a type system with ω there are two kinds of typeable recursion: the one explicitly present in the syntax, and the one obtained by the so-called *fixed-point combinators*. Hence, further restrictions have to be imposed. Take for instance the rewrite system:

$$\begin{array}{ll} F\,(C\,(x)) & \to F\,(x), \\ A\,(x,y) & \to Ap\,(y, Ap\,(Ap\,(x,x),y)), \end{array}$$

that satisfies the scheme of [5], and is typeable with respect to

$$\begin{array}{l} \mathcal{E}(F) = \omega{\to}\sigma, \\ \mathcal{E}(C) = \omega{\to}\sigma, \\ \mathcal{E}(A) = ((\alpha{\to}\mu{\to}\beta){\cap}\alpha){\to}((\beta{\to}\rho){\cap}\mu){\to}\rho. \end{array}$$

Let $A_0 = \lambda xy.A\,(x,y)$, then $B \vdash_{\mathcal{E}} F\,(A\,(A_0, \lambda x.C\,(x))){:}\sigma$, but

$$F\,(A\,(A_0, \lambda x.C\,(x))) \to^{*} F\,(C\,(A\,(A_0, \lambda x.C\,(x)))) \to F\,(A\,(A_0, \lambda x.C\,(x))).$$

The underlying problem is that $Ap\,(A_0, A_0)$ is acting as a fixed-point combinator: for every G that has type $\omega{\to}\sigma$, the term $A\,(A_0, G_0)$ has type σ, and $A\,(A_0, G_0) \to^{*}_{\mathbf{R}} G\,(A\,(A_0, G_0))$ (taking for G_0 of course the term $\lambda x.G\,(x)$). The solution for this particular problem will be to demand *patterns that cannot be typed using the type constant ω.*

The recursive scheme that we will use in this paper was introduced in [7], to prove the Approximation Theorem for $\mathcal{C}$TRS. It is a generalization of the scheme defined in [6] to prove that all typeable terms in typeable $\mathcal{C}$TRS are head-normalizable. In fact, the head-normalization property of typeable terms in typeable TRS+β follows directly from the Approximation Theorem for TRS+β, as we will see below.

Definition 3.9 (Safe systems) A typeable TRS+β over a signature with a set of function symbols $\mathcal{F}_n = \mathcal{C} \cup \{Ap\} \cup \{F^1, \ldots, F^n\}$, where $F^1, \ldots, F^n$ are the defined symbols and $\mathcal{C}$ is the set of constructors, is called *safe* if it fulfills the following conditions:

i) $F^1, \ldots, F^n$ are defined in an incremental way (i.e. there is no mutual recursion) by rules that satisfy the *general scheme*:

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

where $\vec{x}$, $\vec{y}$ are sequences of variables and $\vec{x} \subseteq \vec{y}$; $\overrightarrow{\mathrm{C}}[\;]$, $\mathrm{C}'[\;]$, $\overrightarrow{\mathrm{C}_1}[\;]$, and $\overrightarrow{\mathrm{C}_m}[\;]$ are sequences of contexts in $T(\mathcal{F}_{i-1}, \mathcal{X})$; and, for $1 \leq j \leq m$, $\overrightarrow{\mathrm{C}}[\vec{x}] \rhd_{mul} \overrightarrow{\mathrm{C}_j}[\vec{x}]$, where $\lhd$ is the strict subterm ordering (so $\rhd$ denotes superterm) and *mul* denotes multiset extension,

and moreover, in every rewrite rule

ii) *a)* patterns cannot be typed with ω (i.e. no variable typed with ω occurs twice in $F^i\,(\overrightarrow{\mathrm{C}}[\vec{x}], \vec{y})$, and no subterm of $\overrightarrow{\mathrm{C}}[\vec{x}]$ can be typed with ω), and

b) the type derivations for $\overrightarrow{\mathrm{C}_j}[\vec{x}]$ $(1 \leq j \leq m)$ in the right-hand side are subderivations of those of $\overrightarrow{\mathrm{C}}[\vec{x}]$.

Note that the rewrite system of the example above is not safe: the pattern $C\,(x)$ in the first rule must be typed with ω in a type derivation of $F\,(C\,(x)){:}\sigma$. However, the system containing only the second rule is safe.

The rewrite system of Example *1.6* is not safe because the rule stating the associativity of append does not satisfy the general scheme. If we do not consider that rule, then it is possible to define an environment where the system is safe. For example, assuming we are working with lists of natural numbers we can use the following environment:

$\mathcal{E}$ (nil) = *natlist,*
$\mathcal{E}$ (cons) = *nat→natlist→natlist,*
$\mathcal{E}$ (append) = *natlist→natlist→natlist,*
$\mathcal{E}$ (map) = *(nat→nat)→natlist→natlist.*

Note that the definition of safe system given in [6] is a particular case of the one given above, so all the systems that are safe in that sense, are also safe according to the previous definition.

The rest of this section will be devoted to the proof of the theorem stating that $B \vdash_{\mathcal{E}} t{:}\sigma$ implies $\mathcal{A}pprox\,(B,t,\sigma)$ (which stands for $\exists\, a \in \mathcal{A}\,(t)\ [B \vdash_{\mathcal{E}} a{:}\sigma]$), for all typeable $\mathrm{TRS}{+}\beta$ that are safe. To prove this, we will use the well-known method of Computability Predicates [24] (see also [17]). 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 $Comp\,(B,t,\sigma)$ holds, then $\mathcal{A}pprox\,(B,t,\sigma)$. In the second part *Comp* is shown to hold for each typeable term.

Definition 3.10 *i)* Let B be a basis, $t \in T(\mathcal{F},\mathcal{X})$, and σ a type such that $B \vdash_{\mathcal{E}} t{:}\sigma$. We define the Computability Predicate $Comp\,(B,t,\sigma)$ recursively on σ by:

a) $\sigma = \varphi$, or $\sigma = s$. $Comp\,(B,t,\sigma) \iff \mathcal{A}pprox\,(B,t,\sigma)$.

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

c) $\sigma = \sigma_1\cap\cdots\cap\sigma_n$.
$$Comp\,(B,t,\sigma_1\cap\cdots\cap\sigma_n)\ (n \geq 0) \iff \forall\, 1 \leq i \leq n\ [Comp\,(B,t,\sigma_i)].$$

ii) We say that a term-substitution R is *computable in a basis* B if there is a basis B' such that for every $x{:}\sigma \in B$, $Comp\,(B',x^{\mathrm{R}},\sigma)$ holds.

Notice that $Comp\,(B,t,\omega)$ holds as special case of part *(i.c)*.

Lemma 3.11 Let $\sigma \leq \tau$. *Then* $Comp\,(B,t,\sigma)$ *implies* $Comp\,(B,t,\tau)$.

Comp satisfies the standard properties *C1* and *C3* of computability predicates (see [17]). *C1* states that computable terms have the desired property (approximation in this case). *C3* concerns neutral terms, and will be divided in two parts to gain readability. In general, computability predicates are closed under reduction (property *C2*). With the notion of computability we are using, *C2* is not needed.

Property 3.12 C1. $Comp\,(B,t,\sigma)$ *implies* $\mathcal{A}pprox\,(B,t,\sigma)$.

C3. Let t be neutral. If $B \vdash_{\mathcal{E}} t{:}\sigma$ *and there exists* v *such that* $t \to^* v$ *and* $Comp\,(B,v,\sigma)$, *then* $Comp\,(B,t,\sigma)$.

C3'. Let t be neutral. If $B \vdash_{\mathcal{E}} t{:}\sigma$ *and there exists a such that* $a \in \mathcal{A}\,(t)$, $a \sqsubseteq t$, *and* $B \vdash_{\mathcal{E}} a{:}\sigma$, *then* $Comp\,(B,t,\sigma)$.

In order to prove the Approximation Theorem it would be enough to prove that for every term *t*, basis B and type σ, if $B \vdash_{\mathcal{E}} t{:}\sigma$, then $Comp\,(B,t,\sigma)$. As usual in proofs by computability, we need to prove a stronger property: if *t* has type σ in a basis B and R is computable in B, then there exists a B' such that $Comp\,(B',t^{\mathrm{R}},\sigma)$. We will prove this property by noetherian induction, for which we will need the following:

Definition 3.13 *i)* Let *t* be a term such that $B \vdash_{\mathcal{E}} t{:}\sigma$. We write $t \to^{a}_{B\sigma} u$ if

a) $t \to^{+} u$,

b) there is no $a \in \mathcal{A}(t)$ such that $a \sqsubseteq t$ and $B \vdash_{\mathcal{E}} a{:}\sigma$,

c) there exists $a \in \mathcal{A}(u)$ such that $a \sqsubseteq u$ and $B \vdash_{\mathcal{E}} a{:}\sigma$.

ii) Let t, u be terms such that $B \vdash_{\mathcal{E}} t{:}\sigma$ and $B \vdash_{\mathcal{E}} u{:}\sigma$. We write $t \rhd^{a}_{B\sigma} u$ if $t \rhd u$, and there exist $a_1 \in \mathcal{A}(t)$ and $a_2 \in \mathcal{A}(u)$ such that $a_1 \sqsubseteq t$, $a_2 \sqsubseteq u$, $B \vdash_{\mathcal{E}} a_1{:}\sigma$,$B \vdash_{\mathcal{E}} a_2{:}\sigma$, and $a_1 \rhd a_2$.

Intuitively, $t \rightarrow^{a}_{B\sigma} u$ if u is a reduct of t for which there is an approximant with the same form and the same type. The relation $\rhd^{a}_{B\sigma}$ is a strict subterm ordering that preserves the previous property.

Definition 3.14 Let $\blacktriangleright$ stand for the well-founded encompassment ordering, i.e. $u \blacktriangleright v$ if $u \neq v$ modulo renaming of variables, and $u|_p = v^{\mathrm{R}}$ for some position $p \in u$ and substitution R. Let $>_{\mathbb{N}}$ denote the standard ordering on natural numbers, and *lex, mul* denote respectively the *lexicographic* (from left to right) and *multiset* extension of an ordering.

Let $(\Sigma, \mathbf{R})$ be a TRS$+\beta$. We define the ordering $\gg$ on triples – a natural number, a term, and a multiset of terms that are typeable in a basis B' with types $\{\rho_i\}$ – as the object $(>_{\mathbb{N}}, \blacktriangleright\ , (\rightarrow^{a}_{B'\rho_i} \cup \rhd^{a}_{B'\rho_i})_{mul})_{lex}$.

Property 3.15 Let t be such that $B \vdash_{\mathcal{E}} t{:}\sigma$, and R *be computable in B (i.e. for every $x{:}\rho_i$ in B, Comp $(B', x^{\mathrm{R}}, \rho_i)$ holds). Then Comp $(B', t^{\mathrm{R}}, \sigma)$ holds.*

Proof: We will interpret a term u^{R} by the triple $\langle i, u, \{R\}\rangle$, where i is the maximal super-index of the function symbols (see Definition *3.9*) belonging to u, and $\{\mathrm{R}\}$ is the multiset of typeable terms $\{x^{\mathrm{R}} \mid x \in FV\,(u)\}$. These triples are compared in the ordering $\gg$.

Since R is computable in B, $\rightarrow^{a}_{B'\rho_i}$ is well-founded on the image of R. The union of $\rhd^{a}_{B'\rho_i}$ and $\rightarrow^{a}_{B'\rho_i}$ is also well-founded. Hence, $\gg$ is a well-founded ordering. The proof of the property goes by noetherian induction on $\gg$ and case analysis. ■

With this result we are able to prove the main theorem of this section.

Theorem 3.16 (Approximation Theorem) *If $(\Sigma, \mathbf{R})$ is typeable in $\mathcal{E}$ and safe, then for every term t such that $B \vdash_{\mathcal{E}} t{:}\sigma$, there is an $a \in \mathcal{A}(t)$ such that $B \vdash_{\mathcal{E}} a{:}\sigma$.*

Proof: The theorem follows from Properties *3.15* and *C1*, taking R such that $x^{\mathrm{R}} = x$. ■

4 Normalization results

In this section we will use the Approximation Theorem to prove theorems of head-normalization and normalization. We will also state a strong-normalization theorem for a restricted system.

Theorem 4.1 Let $(\Sigma, \mathbf{R})$ be typeable in $\mathcal{E}$ and safe. If $B \vdash_{\mathcal{E}} t{:}\sigma$, and $\sigma \neq \omega$, then t has a head-normal form.

Proof: If $B \vdash_{\mathcal{E}} t{:}\sigma$, then by Theorem *3.16*, there is an $a \in \mathcal{A}(t)$ such that $B \vdash_{\mathcal{E}} a{:}\sigma$. Since $\sigma \neq \omega$, $a \neq \perp$, and, since $a \in \mathcal{A}(t)$, there is a v such that $t \rightarrow^{*} v$ and $a \sqsubseteq \mathcal{DA}(v)$. Then, by Lemma *3.8-(ii)*, v is in head-normal form, so, in particular, t has a head-normal form. ■

In the intersection type assignment system for LC, terms that are typeable with a type σ from a basis B such that ω does not occur in B and σ, are normalizable [11]. In the framework of $\mathbb{C}$TRS this property holds for non-Curryfied terms (i.e. terms without *Ap* and Curryfied functions), provided the rewrite rules satisfy certain conditions: the function definitions have to be sufficiently complete (see [6] for more details). In the case of TRS$+\beta$, Curryfied versions of the function symbols of the signature are obtained through the use of λ-abstraction (we do not need rules to define them since we have β-reduction). The only terms that we have to exclude are those containing subterms of the form $Ap\,(F\,(t_1, \ldots, t_n), u)$, where $F \in \mathcal{F}$ with arity n and $t_1, \ldots, t_n, u$ are arbitrary terms. This is because a term of this form can have a type without ω even if F is used with a type containing ω. To exclude these terms, we will assume that the environment $\mathcal{E}$ is such that $F(t_1, \ldots, t_n)$ cannot have an arrow type if F has arity n. The definition of complete TRS+β is similar to the definition of complete $\mathcal{C}\!\mathcal{U}$TRS [6, 7].

Definition 4.2 Let $\mathcal{E}$ be an environment such that for any $F \in \mathcal{F}$ of arity n, $F(t_1, \ldots, t_n)$ cannot have an arrow type. A TRS+β is *complete* in the environment $\mathcal{E}$ if whenever a typeable term t, of which the type does not contain ω, is reducible at a position p such that $t|_p$ can be assigned a type containing ω, there exists $q < p$ such that $t|_q$ has a type without ω and $t|_q[x]_p$ (where x is a fresh variable) is not in head normal form.

Intuitively, in a complete TRS+β a term $F(t_1, \ldots, t_n)$ that has an ω-free type, and where there is a redex t_i that can be assigned a type containing ω, will be reducible either at the root or in some t_j with an ω-free type. This means that the rules defining F cannot have patterns that have types with ω, and also that constructors cannot accept arguments having a type which contains ω. Moreover, if a defined function accepts arguments having types with ω then its definition must be exhaustive.

Defined functions of safe systems satisfy the first condition. So, a safe system is complete whenever constructors have ground types and for all defined function F that accept arguments with types that contain ω, the rules defining F cover all possible cases.

The following lemma is easy to prove for complete TRS+β.

Lemma 4.3 Let $(\Sigma, \mathbf{R})$ be a complete TRS+β *in $\mathcal{E}$. Let $a \in \mathcal{DA}$. If $B \vdash_{\mathcal{E}} a{:}\sigma$, and ω does not occur in B and σ, then a contains no $\bot$.*

With the help of this lemma and the Approximation Theorem, we can show the following:

Theorem 4.4 Let $(\Sigma, \mathbf{R})$ be typeable in $\mathcal{E}$, safe and complete. If $B \vdash_{\mathcal{E}} t{:}\sigma$, and ω does not occur in B and σ, then t is normalizable.

Proof: If $B \vdash_{\mathcal{E}} t{:}\sigma$, then by Theorem *3.16*, there is an $a \in \mathcal{A}(t)$ such that $B \vdash_{\mathcal{E}} a{:}\sigma$. So there is a v such that $t \rightarrow^* v$ and $a \sqsubseteq \mathcal{DA}(v)$. Then by the above lemma, a is free of $\bot$, so in particular $a \equiv v$, so t has a normal form. ■

When the type constant ω is removed from the system, all typeable terms are strongly normalizable. The technique required to prove this property is very similar to the one used in [2, 5], so we will not give the details of the proof.

Theorem 4.5 Let $\vdash_{\mathcal{E}}^{\not\omega}$ denote the notion of type assignment obtained from $\vdash_{\mathcal{E}}$ by removing the type constant ω, and let t be a term in a TRS+β *that satisfies the general scheme. Then $B \vdash_{\mathcal{E}}^{\not\omega} t{:}\sigma$ implies that t is strongly normalizable.*

5 Conclusions

We have extended first-order term rewriting systems with application, abstraction and β-reduction, and have proposed a type assignment system for this language. Term rewriting systems with abstraction and application combine the advantages of algebraic rewrite systems, which model algebraic operations on data structures, with the power of LC. The type assignment system that we defined is a true extension of the intersection system for LC, so the pure LC-fragment of the language has the well-known normalization properties:

i) the set of terms typeable without ω is the set of strongly normalizable terms,

ii) the set of terms typeable with type σ from a basis B, such that ω does not occur in B and σ, is the set of normalizable terms, and *iii)* the set of terms typeable with type $\sigma \neq \omega$ is the set of terms having a head normal form.

If we do not allow abstractions in right-hand sides of rewrite rules, and consider the algebraic fragment of our language, we obtain a @TRS, for which the following properties hold [7]:

i) terms typeable without ω are strongly normalizable,

ii) non-Curryfied terms typeable with type σ from a basis B, such that ω does not occur in B and σ are normalizable, and

iii) terms typeable with type $\sigma \neq \omega$ have a head normal form.

Notice that the converses of the previous properties do not hold, because the environment is given (and fixed).

In [7], these properties were proved directly from the strong normalization property of "derivation reduction," a rewrite relation on derivations that is strongly normalizing even in type systems with ω. The Approximation Theorem is also a consequence of this property. Since it is at this moment not clear if that technique extends to systems with abstraction, in this paper we have given a direct proof of the Approximation Theorem from which we can easily deduce the head-normalization and normalization properties (at the expense of a more complicated strong normalization proof).

We have shown that the normalization properties that are enjoyed by both languages when considered separately, are inherited by the combined language. This supports our initial claim that type assignment systems provide a sound environment for the combination of the programming paradigms based on TRS and LC. But in order to provide more evidence for this claim, other important properties (such as confluence, preservation of normalizing strategies) have to be studied. This will be a subject of future work.

Acknowledgements

The second author wishes to thank prof. Mariangiola Dezani for her gentle guidance, and Salvatore Favata for making the department of Computer Science of Torino a more pleasant environment to work in.

References

[1] Ariola Z., R. Kennaway, J.W. Klop, R. Sleep and F-J. de Vries. Syntactic definitions of undefined: on defining the undefined. In *Proceedings of TACS '94*, volume 789 of *LNCS*, pages 543–554, 1994.

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

[3] S. van Bakel. Partial Intersection Type Assignment in Applicative Term Rewriting Systems. In *Proceedings of TLCA '93*, volume 664 of *LNCS*, pages 29–44, 1993.

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

[5] S. van Bakel and M. Fernández. Strong Normalization of Typeable Rewrite Systems. In *Proceedings of HOA '93*, volume 816 of *LNCS*, pages 20–39, 1994.

[6] S. van Bakel and M. Fernández. (Head-)Normalization of Typeable Rewrite Systems. In *Proceedings of RTA '95*, volume 914 of *LNCS*, pages 279–293, 1995.

[7] S. van Bakel and M. Fernández. Approximation and Normalization Results for Typeable Rewrite Systems. To appear in *Proceedings of HOA '95*, Paderborn, Germany, 1995.

[8] F. Barbanera and M. Fernández. Combining first and higher order rewrite systems with type assignment systems. In *Proceedings of TLCA '93*, volume 664 of *LNCS*, pages 60–74, 1993.

[9] F. Barbanera, M. Fernández, and H. Geuvers. Modularity of Strong Normalization and Confluence in the λ-algebraic-cube. In *Proceedings of LICS'94*, 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] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proceedings of LICS'88*, pages 82–90, 1988.

[13] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization. *Theoretical Computer Science*, 83(1):3–28, 1991.

[14] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic confluence. *Information and Computation*, 82:3–28, 1992.

[15] 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.

[16] D. J. Dougherty. Adding Algebraic Rewriting to the Untyped Lambda Calculus. In *Proceedings of RTA '91*, volume 488 of *LNCS*, pages 37–48. 1991.

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

[18] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF. *Lecture Notes in Computer Science*, volume 78, 1979.

[19] G. Huet and J.J. Lévy. Computations in Orthogonal Rewriting Systems. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in Honour of Alan Robinson*. MIT Press, 1991.

[20] J.P. Jouannaud and M. Okada. Executable higher-order algebraic specification languages. In *Proceedings of LICS'91*, pages 350–361, 1991.

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

[22] 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.

[23] Mitsuhiro Okada. Strong normalizability for the combined system of the types lambda calculus and an arbitrary convergent term rewrite system. In *Proceedings of ISSAC 89, Portland, Oregon*, 1989.

[24] W.W. Tait. Intensional interpretation of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–223, 1967.

[25] S.R. Thatte. Full Abstraction and Limiting Completeness in Equational Languages. *Theoretical Computer Science*, 65:85–119, 1989.

[26] 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 *LNCS*, pages 1–16, 1985.

[27] C.P. Wadsworth. The relation between computational and denotational properties for Scott's D_∞-models of the lambda-calculus. *SIAM J. Comput.*, 5:488–521, 1976.