Normalisation and Approximation results for typeable Term Rewriting Systems (with abstraction and β-rule)

Summary

Steffen van Bakel, Franco Barbanera, and Maribel Fernández

[1] Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, U.K.
[2] Dipartimento di Informatica, Università degli Studi di Torino, Corso Svizzera 185, 10149 Torino, Italia,
[3] DMI - LIENS, CNRS / Ecole Normale Supérieure, 45, rue d'Ulm, 75005 Paris, France.
`maribel@dmi.ens.fr`

This paper gives an overview of results presented in (in order of appearance) [3], [8], [9] (that together will appear as [11]), and [10], [7].

1 Introduction

There are essentially three paradigms in common use for the design of functional programming languages: the λ-calculus (LC for short), Term Rewriting Systems (TRS), and Term Graph Rewriting Systems (TGRS). The LC, or rather combinator systems, forms the underlying model for the functional programming language Miranda[1] [35], TRS are used in the language OBJ [24], and TGRS form the base model for the language Clean [17].

For LC, there exists a well understood notion of type assignment known as the *Curry type assignment system* [20], that expresses abstraction and application. The *intersection type discipline* [18, 15] is an extension of Curry's system that consists of allowing more than one type for term-variables and terms, adding a type constant 'ω', and considering the type constructor '$\cap$' in addition to the type constructor '$\rightarrow$'. One of the most appealing features of intersection type assignment in LC is the fact that normalisation of terms can be studied through assignable types (see e.g. [15] and [2]):

- M has a head-normal form iff $B \vdash M : \sigma$ and $\sigma \neq \omega$.
- M has a normal form iff $B \vdash M : \sigma$ and ω does not occur in B and σ.
- M is strongly normalisable iff $B \vdash M : \sigma$ and ω is not used at all.

The *essential intersection system for LC* defined by [5] is a restriction of the intersection type discipline that satisfies all the properties above. Its main advantage is that the set of types assignable to a term is significantly smaller than in the full intersection system.

Though many functional programming languages allow programmers to specify an algorithm (function) as a set of rewrite rules, type assignment for TRS has not attracted much attention until now. This is remarkable, since TRS and LC are essentially different: although both formalisms are Turing-complete, there exists no direct translation of TRS to LC. For example, adding the definition of surjective pairing,

$$\begin{array}{ll} \mathsf{Fst}\,(\mathsf{Pair}\,(\mathsf{x},\mathsf{y})) & \mathsf{x} \\ \mathsf{Snd}\,(\mathsf{Pair}\,(\mathsf{x},\mathsf{y})) & \mathsf{y} \\ \mathsf{Pair}\,(\mathsf{Fst}\,(\mathsf{x}),\mathsf{Snd}\,(\mathsf{x})) & \mathsf{x} \end{array}$$

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

to LC gives a system in which the Church-Rosser property no longer holds [28]; this implies that the above TRS cannot be expressed in LC.

Although it seems straightforward to extend type assignment systems for LC to TRS, it is not evident that those borrowed systems will still have, for general TRS, all the properties they possessed in the setting of LC. For example, some restrictions have to be imposed in the assignment of types to rewrite rules in order to ensure the subject reduction property (i.e. preservation of types under rewriting), as illustrated in [12].

The aim of this paper is to define a notion of (essential) intersection type assignment *directly for TRS* and to study normalisation properties in that setting. We use intersection types because more meaningful terms can be typed in this way. Also, the notion of type assignment presented in this paper applies to TGRS and in that framework intersection types are the natural tool to type nodes that are shared (another notion of type assignment on TGRS was defined by [16], to study safeness of destructive updates). Intersection types are also promising for use in functional languages, since they provide a good formalism to express overloading, see [32].

We consider *Curryfied* TRS (CuTRS), a slight extension of the TRS defined by [29], and [21]. CuTRS contain a special binary operator Ap, that models application and allows for partial application of function symbols (Curryfication). CuTRS are also extensions of the constructor systems used in most functional programming languages in that they do not discriminate against the varieties of function symbols that can be used in patterns. However, we will in some cases make this distinction when we will study normalisation properties of CuTRS.

Recently, some results have been obtained in the field of *typed TRS* [21] and the combination of those with intersection type assignment systems for LC [13]. The idea behind those systems is that rewrite rules aim to describe manipulations of objects of an algebraic data-type and, therefore, concepts like polymorphism are not introduced within TRS. In contrast, in this paper we present a *type assignment system for* CuTRS that is closer to the approach of intersection type assignment in LC; in particular, rewrite rules can be polymorphic.

The type assignment system on CuTRS that we define is based on a combination of the essential intersection system for LC and the type assignment system of ML [30], both extensions of Curry's type assignment system. Type assignment will be defined through a natural deduction system, assuming that every function symbol has a predefined type, given in an *environment*. This approach is similar to the one taken by [26] to define the principal Curry type of an object in Combinatory Logic.

The polymorphic aspect of our type assignment system becomes apparent in the derivation rule that deals with the assignment of a type to a term like $F(t_1,\ldots,t_n)$. There the type predefined for F in the environment can be 'instantiated' by applying operations of substitution, expansion, and lifting (see [4]). The operation of substitution deals with the replacement of type-variables by types, the operation of expansion replaces types by the intersection of a number of copies of that type and coincides with the one given by [19], and the operation of lifting deals with both taking more specific types in bases and assigning a more general type to terms. We use these three operations, instead of just substitution, not only because more terms are typeable in this way, but also to obtain a natural embedding of LC in TRS that preserves assignable types (with just substitution, this would not be possible).

The type assignment system presented in this paper can be seen as a generalization of the systems of [12] and [6]; the main difference is the set of types used: Curry types in [12], intersection types of Rank 2 in [6], and strict intersection types in this paper. Type assignment in those systems is decidable, whereas in the one presented here it is not. However, the normalisation results we will prove hold also for free in these decidable restrictions of the system.

In contrast with LC, typeable terms in CuTRS need not even be head-normalisable; for example, consider a typeable term t and a rule $t \to t$. That is why we need to control the use of recursion by imposing some syntactical conditions on the rewrite rules (a generalization of primitive recursion). We will define a recursive scheme for rewrite rules that is inspired by the *general scheme* of [27]. The general scheme was devised for the incremental definition of higher order functionals based on first order definitions, such that their combination with polymorphic LC is terminating. It was also used for defining higher order functions compatible with other lambda calculi by [13] and [14].

It is worthwhile to notice that, even with the severe restrictions imposed on rewrite rules by the general scheme, the class of CuTRS that satisfies the scheme is Turing-complete, a property that systems without Ap would not possess.

For a type assignment system in which the type ω is not used, we will prove (adapting the method of Computability Predicates of [25] and [33]) that for all typeable CuTRS satisfying the general scheme, typeable terms are strongly normalisable.

Perhaps surprisingly, in the type system with ω, the general scheme is not enough to ensure head-normalisation of typeable terms. Therefore, to study head-normalisation of typeable CuTRS we will define a suitable restriction of the general scheme, called the *HNF-scheme*, where the patterns of rewrite rules are constructor terms that have sorts as types. We should remark here that our notion of head-normal form for CuTRS is similar in spirit to the notion of weak head-normal form in LC; the latter is used in most functional programming languages based on LC, see [1]. We will again use the method of Computability Predicates to prove that for all typeable CuTRS satisfying the HNF-scheme, every typeable term has a head-normal form. We will also show that if Curryfication is not allowed, under certain restrictions, terms typeable with a type that does not contain ω are normalisable.

These results apply in particular to Combinator Systems, a class of CuTRS that satisfies the required conditions. For Combinator Systems that are combinatory complete, a type assignment system was defined by [22]. Our system can be seen as a generalization of that one.

The lay-out of this paper is as follows: We present CuTRS in Section 2. In Section 3 we briefly recall the essential intersection system for LC. In Section 3 we introduce the essential intersection system for CuTRS, and compare it with the one for LC. In Section 4 we present the general scheme and prove the strong normalisation theorem for the type assignment system without ω. We then show that in the system with ω, and considering the restrictions formulated in the HNF-scheme, all typeable terms have a head-normal form. Finally we prove the normalisation result.

The results presented in this paper were first published, in a much condensed form, as [3, 8], and [9].

2 Curryfied Term Rewriting Systems

In this section we present Curryfied Term Rewriting Systems (CuTRS), an extension of the Term Rewriting Systems (TRS) defined by [29], and [21]. CuTRS are first-order TRS extended with a binary function symbol which models partial application of functions. This feature allows us to make a straightforward translation of Lambda Calculus (LC) to CuTRS, i.e. to a first-order rewrite system.

CuTRS are also an extension of the constructor systems used in most functional programming languages in that not only constructor symbols can be used in the operand space of the left-hand side of rewrite rules, but all function symbols.

Definition 2.1 An *alphabet* or *signature* Σ consists of a countable infinite set X of variables x, $y, z, x', y', \ldots$; a non-empty set $\mathcal{F}$ of *function symbols* $F, G, \ldots$, each equipped with an 'arity' (a natural number); and a special binary operator, called *application* (Ap).

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

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 $F(t_1,\ldots,t_n) \in T(\mathcal{F},\mathcal{X})$. The t_i ($i \in \underline{n}$) are the *arguments* of the last term. We will omit the brackets when $n = 0$.

We will write $Var(t)$ for the set $\{x \in \mathcal{X} \mid x \textit{ occurs in } t\}$.

Definition 2.3 A *term-substitution* R is a mapping from $T(\mathcal{F},\mathcal{X})$ to $T(\mathcal{F},\mathcal{X})$ satisfying

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

for every n-ary ($n \geq 0$) function symbol F, and is determined by its restriction to a finite set of variables. Sometimes we will use the notation $\{x_1 \mapsto t_1,\ldots,x_n \mapsto t_n\}$ for term-substitutions. We will also write t^{R} instead of $\mathrm{R}(t)$.

Definition 2.4 *i*) Given a signature Σ with a set X of variables and a set $\mathcal{F}$ of function symbols, a *rewrite rule* in σ is a pair (l,r) of terms in T(F, X), such that l is not a variable, and thevariables occurring in r appear in l. Often a rewrite rule will get a name, e.g. $\mathbf{r}$, and we will write $\mathbf{r} : l \to r$.
If $F(t_1,\ldots,t_n)$ is the left-hand side of a rule $\mathbf{r}$, and, for $i \in \underline{n}$, either t_i is not a variable, or t_i is a variable and there is a $1 \leq i \neq j \leq n$ such that $t_i = t_j$, then t_i is called a *pattern* of $\mathbf{r}$.

ii) A *Curryfied Term Rewriting System* (ℭTRS) is a pair $(\Sigma,\mathbf{R})$ of a signature $\Sigma = (\mathcal{F},\mathcal{X})$, and a set $\mathbf{R}$ of rewrite rules in Σ, such that, for every $F \in \mathcal{F}$ of arity $n > 0$, there exist n additional function symbols $F_{n-1},\ldots,F_1,F_0$ in $\mathcal{F}$, the *Curryfied-versions of* F, and $\mathbf{R}$ contains the n rewrite rules:

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

If F_i is a Curryfied version of a function symbol F, then its Curryfied versions coincide with the corresponding Curryfied versions of F, being $F_{i-1},\ldots,F_0$. Moreover, we will assume that for any rule $\mathbf{r} : l \to r$ in $\mathbf{R}$, if Ap occurs in l, then $\mathbf{r}$ is of the shape:

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

for some Curryfied version F_{i-1}, and that Curryfied versions do not appear in the root of any left-hand side.

iii) Terms that do not contain Curryfied versions of symbols are called *non-Curryfied terms*.

iv) A rewrite rule $\mathbf{r} : l \to r$ determines a set of *reductions* $l^{\mathrm{R}} \to r^{\mathrm{R}}$ for all term-substitutions R. The term l^{R} is called a *redex*; it may be replaced by its *contractum* r^{R} inside any context C[ ]; this gives rise to *rewrite steps*:

$$\mathrm{C}[l^{\mathrm{R}}] \to_r \mathrm{C}[r^{\mathrm{R}}].$$

We will write $t \to_{\mathbf{R}} t'$ if there is a $\mathbf{r} \in \mathbf{R}$ such that $t \to_r t'$.
Concatenating rewrite steps we have (possibly infinite) *rewrite sequences* $t_0 \to t_1 \to \cdots$. If $t_0 \to \cdots \to t_n$ ($n \geq 0$) we will also write $t_0 \to^* t_n$.

Because of the extra rules for $F_{n-1}, \ldots, F_1, F_0$, etc., the rewrite systems are called *Curry-closed*. When presenting a rewrite system we will sometimes omit the rules that define the Curryfied versions.

Definition 2.5 A rewrite rule $\mathbf{r} : l \to r$ *defines* F if F is the leftmost, outermost symbol in l that is not an Ap; we call F *the defined symbol of* $\mathbf{r}$. We say that $F \in \mathcal{F}$ is a *defined symbol* if there is a rewrite rule that defines F. Otherwise, it is a *constructor*.

Example 2.6 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 GTRS

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

is *not* considered a recursive system. Notice that, for example, the term $D\,(D_0)$ has no normal form (this term plays the role of $(\lambda x.xx)(\lambda x.xx)$ in LC). This means that, in the formalism of this paper, there exist non-recursive first-order rewrite systems that are not normalising.

Definition 2.7 *i*) A term is in *normal form* if it is irreducible.
ii) A term t is in *head-normal form* if for all t' such that $t \to^* t'$:
  a) t' is not itself a redex and
  b) if $t' = Ap\,(v,u)$, then v is in head-normal form.
iii) A term is *(head) normalisable* if it can be reduced to a term in (head/constructor-hat) normal form.
iv) A rewrite system is *strongly normalising* (or terminating) if all the rewrite sequences are finite; it is *(head) normalising* if every term is (head) normalisable.

Example 2.8 Take the GTRS

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

where F, B, and H are defined symbols, and A is a constructor (notice the use of a defined symbol as a pattern in the second rule). The term $F\,(B\,(H),H)$ is not a redex, but it is not a head-normal form either, since it reduces to $F\,(H,H)$ which is a redex. This term reduces to $A\,(H)$, which is a head-normal form.

This notion of head-normal form corresponds to the notion of weak ead-normal form in LC. For instance, if F is a function symbol of arity n, $F_i\,(t_1,\ldots,t_i)$ is a head-normal form according to the previous definition. Clearly it corresponds to the λ-term

$$\lambda x_{i+1} \ldots x_n.F\,(t_1,\ldots,t_i,x_{i+1},\ldots,x_n),$$

which is in weak head-normal form.

3 Essential Intersection System for GTRS

In this section we present an intersection type assignment system for GTRS. It is a *partial* system in the sense of [31]: not only will we define how terms and rewrite rules can be typed, but we will also assume that every function symbol in the signature *has* a type, provided by an *environment* (i.e. a mapping from function symbols to types).

To assign types to terms in the $\mathcal{C}$TRS framework, we are going to use three *operations* on types (that extend to bases and to pairs of $\langle$*basis*,*type*$\rangle$), namely *substitution*, *expansion*, and *lifting*. These were first defined in [4] to show that the strict type assignment system of [2] has the principal type property. Substitution is the operation that instantiates a type (i.e. that replaces type variables by types). The operation of expansion replaces types by the intersection of a number of copies of that type. The operation of lifting replaces basis and type by a smaller basis and a larger type, in the sense of $\leq$.

The operations of substitution, expansion, and lifting can be composed to form *chains* of operations. The *set* $\mathcal{C}h$ *of chains* is defined as the smallest set containing expansions, substitutions, and liftings, that is closed under composition $\circ$.

See [3] for formal definitions.

Definition 3.1 *i*) $\mathcal{T}_S$, the set of *strict types*, and T, the set of *strict intersection types*, are defined through mutual induction by:

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

b) All sorts $s_1, s_2, \ldots \in \mathcal{T}_S$.

c) If $\tau \in \mathcal{T}_S$ and $\sigma \in \mathcal{T}$, then $\sigma \rightarrow \tau \in \mathcal{T}_S$.

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

ii) The type ω is defined as an intersection of zero types: if $n = 0$, then $\sigma_1 \cap \cdots \cap \sigma_n = \omega$.

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

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

b) $\forall\, n \geq 0\ [\forall\, i \in \underline{n}\ [\sigma \leq \sigma_i] \Rightarrow \sigma \leq \sigma_1 \cap \cdots \cap \sigma_n]$.

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

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

iv) The relation $\sim_E$ is defined by: $\sigma \sim_E \tau \iff \sigma \leq_E \tau \leq_E \sigma$.

Notice that intersection types (so also ω) occur in strict types only as subtypes at the left-hand side of an arrow type. According to the previous definition, if $\sigma_1 \cap \cdots \cap \sigma_n$ is used to denote a type, then all $\sigma_1, \ldots, \sigma_n$ are strict, therefore they cannot be ω. Notice also that $\mathcal{T}_S$ is a proper subset of $\mathcal{T}$.

To obtain readable types, instead of φ_i we often write only the number i.

Definition 3.2 *i*) A *statement* is an expression of the form $M : \sigma$, where M is a λ-term and $\sigma \in \mathcal{T}$. M is the *subject* and σ the *predicate* of $M : \sigma$.

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 $n \geq 1$.

The relations $\leq_E$ and $\sim_E$ extend to bases in the natural way: $B \leq B' \iff \forall x : \sigma' \in B'\ \exists x : \sigma \in$B$[\sigma \leq \sigma']$, and $B \sim B' \iff B \leq B' \leq B$.

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

Notice that if $n = 0$, then $\cap\{B_1, \ldots, B_n\} =$. Often $B \cup \{x : \sigma\}$ (or $B, x : \sigma$) will be written for the basis $\cap\{B, \{x : \sigma\}\}$, when x does not occur in B.

The three operations on types will be used in this section to define type assignment on $\mathcal{C}$TRS: the types assigned to occurrences of function symbols will be obtained from the type provided by the environment by making a chain of operations.

We will start by defining an environment, which is a mapping from function symbols to strict types. Since we want to associate the Curryfied versions of a function symbol not only through their rewrite rules, but also through their assignable types, we will require that the environment maps a function F and all its Curryfied versions F_i to the same type.

Definition 3.3 Let $(\Sigma,\mathbf{R})$ be a GTRS, and $\mathcal{F}$ the set of function symbols in Σ.

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

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

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

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

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

In the following we will assume that $\mathcal{E}$ is a given environment for a GTRS$(\Sigma,\mathbf{R})$.

Definition 3.4 *i*) *Type assignment* and *derivations* are defined by the following natural deduction system (where all types displayed are in $\mathcal{T}_S$, except for σ in rule $(\leq)$ and $(\to E)$, and $\sigma_1,\ldots,\sigma_n$ in rule (F)):

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

$$(\to E) : \frac{B \vdash_E t_1 : \sigma \to \tau \quad B \vdash_E t_2 : \tau}{B \vdash_E Ap(t_1, t_2) : \tau} \qquad (\mathcal{F}) : \frac{B \vdash_E t_1 : \sigma_1 \quad \ldots \quad B \vdash_E t_n : \sigma_n}{B \vdash_E F(t_1, \ldots, t_n) : \sigma}\ (a)$$

(a): If there exists $C \in \mathcal{C}h$ such that $Ch(\mathcal{E}(F)) = \sigma_1 \to \cdots \to \sigma_n \to \sigma$.

ii) We write $B \vdash_E t : \sigma$ if and only if $t : \sigma$ is derivable from B by using the natural deduction system above. We will say that t *is typeable with respect to* $\mathcal{E}$ (or simply that t is typeable, if the environment is clear from the context) if there exists a basis B and a type $\sigma \neq \omega$ such that $B \vdash_E t : \sigma$.

Notice that if $B \vdash_E t : \sigma$, then B can contain more statements than needed to obtain $t : \sigma$. Moreover, by $(\cap I)$, for every B and t, $B \vdash_E t : \omega$.

The use of an environment in derivation rule $(\to E)$ introduces a notion of polymorphism into our type assignment system. The environment returns the 'principal type' for a function symbol; this symbol can be used with types that are 'instances' of its principal type, obtained by applying chains of operations.

We will now define the type assignment for rewrite rules. This will be done in a careful way to ensure that the subject reduction property (i.e. preservation of types under rewriting) holds.

In [12] and [6] two restrictions of the type assignment system defined above are discussed, for which there is a decidable and sufficient condition on rewrite rules that ensures subject reduction. The condition a rewrite rule should satisfy is that the *principal pair* for the left-hand side is also a correct pair for the right-hand side of the rule. The notion of principal pair is in those papers defined in a constructive way, by defining a unification algorithm for types and defining principal pairs using that algorithm.

Since at this moment there is no general unification algorithm for types in $\mathcal{T}_S$ that works well on all types, we cannot take this approach here. Therefore, for the notion of type assignment defined in this paper we will show that *if* a left-hand side of a rewrite rule has *a* principal pair and using that pair the rewrite rule can be typed, then rewriting using this rule preserves types.

Definition 3.5 A pair $\langle P,\pi\rangle$ is called *a principal pair for a term t with respect to an environment* $\mathcal{E}$ if $P \vdash_{\mathrm{E}} t:\pi$ and, for every B,σ such that $B \vdash_{\mathrm{E}} t:\sigma$, there is a chain Ch such that $C(\langle P,\pi\rangle) = \langle B,\sigma\rangle$.

Definition 3.6 Let $(\Sigma,\mathbf{R})$ be a GTRS, and $\mathcal{E}$ an environment.

i) We will 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 $\pi \in \mathcal{T}_{\mathrm{S}}$, and an assignment of types to l and r such that:

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

b) In $P \vdash_{\mathrm{E}} l:\pi$ and $P \vdash_{\mathrm{E}} r:\pi$, the type actually used for each occurrence of F (or Curryfied versions of F) is $\mathcal{E},(F)$.

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

Condition (*i.b*) of Definition 3.6 is in fact added to make sure that the type provided by the environment for a function symbol F is not in conflict with the rewrite rules that define F. Since by part (*i.b*) of Definition 3.6, all occurrences of the defined symbol in a rewrite rule are typed with the same type, type assignment of rewrite rules is actually defined using Milner's way of dealing with recursion [30].

We have shown a soundness result for chains of operations.

Theorem 3.7 *i) If* $B \vdash_{\mathrm{E}} t:\sigma$ *then for every chain* Ch *such that* $Ch(\langle B,\sigma\rangle) = \langle B',\sigma'\rangle$, *and* $B' \vdash_{\mathrm{E}} t:\sigma'$.

ii) Let $\mathbf{r}: l \rightarrow r$ *be a rewrite rule typeable with respect to the environment* $\mathcal{E}$ *and let F be the defined symbol of* $\mathbf{r}$. *If* Ch *is a chain that contains no lifting, then: if* $Ch(\mathcal{E}(F)) = \sigma_1\cap\cdots\cap\sigma_n$, *then for every* $i \in \underline{n}$, $\mathbf{r}$ *is typeable with respect to* $\mathcal{E}[F{:=}\sigma_i]$.

We have also proved a subject reduction result.

Theorem 3.8 (SUBJECT REDUCTION THEOREM) *Let* $(\Sigma,\mathbf{R})$ *be a typeable* GTRS *with respect to an environment* $\mathcal{E}$. *If* $B \vdash_{\mathrm{E}} t:\sigma$ *and* $t \rightarrow_{\mathbf{R}} t'$, *then* $B \vdash_{\mathrm{E}} t':\sigma$.

It is important to note that, for this last theorem, the condition '$\langle P,\pi\rangle$ is a principal pair for l with respect to $\mathcal{E}$' in Definition 3.6 is crucial. Just saying naively:

" $l \rightarrow r \in \mathbf{R}$ *is typeable with respect to* $\mathcal{E}$ *if there are basis B and type* $\sigma \in \mathcal{T}_{\mathrm{S}}$ *such that* $B \vdash_{\mathrm{E}} l:\sigma$ *and* $B \vdash_{\mathrm{E}} r:\sigma$, "

would give a notion of type assignment that is not closed under rewriting and is not a natural extension of the essential intersection system for LC. The following is an example of the loss of subject reduction (see [12] for more details).

Example 3.9 Consider the rewrite system

$$\begin{array}{lll} H(S_2(x,y)) & \rightarrow & S_2(I_0,y) \\ S(x,y,z) & \rightarrow & Ap(Ap(x,z),Ap(y,z)) \\ K(x,y) & \rightarrow & x \\ I(x) & \rightarrow & x \end{array}$$

and the environment

$$\begin{array}{ll} \mathcal{E}_0(H) & = ((1\rightarrow 2)\rightarrow 3)\rightarrow(1\rightarrow 2)\rightarrow 2, \\ \mathcal{E}_0(S) & = (1\rightarrow 2\rightarrow 3)\rightarrow(1\rightarrow 2)\rightarrow 1\rightarrow 3, \\ \mathcal{E}_0(K) & = 1\rightarrow 2\rightarrow 1, \\ \mathcal{E}_0(I) & = 1\rightarrow 1. \end{array}$$

The first rule is naively typeable with respect to $\mathcal{E}_0$:

$$\dfrac{\dfrac{\overline{x:(1\to2)\to1\to3} \qquad \overline{y:(1\to2)\to1}}{S_2(x,y):(1\to2)\to3}}{H(S_2(x,y)):(1\to2)\to2} \quad\to\quad \dfrac{\overline{I_0:(1\to2)\to1\to2} \qquad \overline{y:(1\to2)\to1}}{S_2(I_0,y):(1\to2)\to2}$$

Take the term $H(S_2(K_0,I_0))$, which reduces to $S_2(I_0,I_0)$. Although the first term is typeable with respect to $\mathcal{E}_0$:

$$\dfrac{\dfrac{\overline{K_0:(4\to5)\to(4\to5)\to4\to5} \qquad \overline{I_0:(4\to5)\to4\to5}}{S_2(K_0,I_0):(4\to5)\to4\to5}}{H(S_2(K_0,I_0)):(4\to5)\to5}$$

the term $S_2(I_0,I_0)$ is not typeable with respect to $\mathcal{E}_0$ with the type $(4\to5)\to5$. In our system the rule is not typeable in this way, because the type assignment used for $H(S_2(x,y))$ is not a principal one. To illustrate this, consider the following derivation:

$$\dfrac{\dfrac{\overline{x:(1\to2)\to4\to3} \qquad \overline{y:(1\to2)\to4}}{S_2(x,y):(1\to2)\to3}}{H(S_2(x,y)):(1\to2)\to2}$$

The pair $\langle\{x:(1\to2)\to4\to3, y:(1\to2)\to4\},(1\to2)\to2\rangle$ can by no chain of operations be obtained from the pair $\langle\{x:(1\to2)\to1\to3, y:(1\to2)\to1\},(1\to2)\to2\rangle$; in the opposite direction, the operation needed is that of $(4\mapsto1)$.

We should emphasise that, when defining type assignment in a naive way, the loss of the subject reduction property is not caused by the fact that intersection types are used. The environment E_0 maps function symbols to Curry-types, so even for a notion of type assignment based on Curry-types, types are not preserved under rewriting.

4 Normalisation Properties of Typeable $\mathcal{G}$TRS

In this section we study normalisation properties of $\mathcal{G}$TRS, using the type assignment system defined above. As in LC, the type ω plays an important role in this study.

As mentioned in the introduction, in the rewriting framework typeability alone does not ensure any normalisation property (for example, consider a typeable term t and a one-rule recursive $\mathcal{G}$TRS of the form $t\to t$). This means that the characterisation of normalisability of terms in $\mathcal{G}$TRS cannot be based on type conditions only, as is possible for LC, but that also syntactic restrictions on the rules have to be imposed. For this reason, we will introduce a *general scheme of recursion*, inspired by [27], that restricts the use of recursion to ensure strong normalisation of all terms typeable without using ω. Moreover, by restricting the scheme a little further, we will show that when ω is used, all typeable terms have a head-normal form.

4.1 Strong Normalisation

In the following we will define the general scheme and the class of SN-safe recursive systems, and prove that, using the type assignment system without ω, typeable SN-safe systems are strongly normalising. We will consider environments that map function symbols to types without ω. Such environments will be called *ω-free*, and in general we will use the expression *ω-free* as prefix to indicate that ω does not appear in an object.

Definition 4.1 Let σ be a signature with a set of function symbols $\mathcal{F}_n = \mathcal{C} \cup \{F^1, \ldots, F^n\}$, where $F^1, \ldots, F^n$ will be the defined symbols that are not Curryfied-versions, and $\mathcal{C}$ the set of constructors and Curryfied versions of symbols. Assume that $F^1, \ldots, F^n$ are defined incrementally, by rules that satisfy the *general scheme*:

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

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

Then the hierarchical GTRS that contains the rules defining $F^1, \ldots, F^n$ is a *SN-safe recursive system*.

This general scheme is a generalisation of primitive recursion. It imposes two main restrictions on the definition of functions: the terms in the multisets $\vec{\mathrm{C}_j}[\vec{x}]$ are subterms of terms in $\vec{\mathrm{C}}$ (this is the 'primitive recursive' aspect of the scheme), and the variables $\vec{x}$ must also appear as arguments in the left-hand side of the rule. Both restrictions are essential to prove the Strong Normalisation Theorem (Theorem 4.4 below). The last one can be replaced by a typing condition, requiring that the variables in $\vec{x}$ that are not included in $\vec{y}$ can only be assigned base types. Also, instead of the multiset extension of the subterm ordering, a lexicographic extension can be used, or even a combination of lexicographic and multiset (see [23] for details about these variants of the scheme).

Example 4.2 The following rewrite system on natural numbers is SN-safe: it is a hierarchical system, the variables that do not appear as arguments in the left-hand sides can only have base types, and the recursive calls in the right-hand sides satisfy the required subterm condition. The signature contains the constructors *Succ*, and *Zero*, and the defined symbols *Add*, and *Mul*.

$$\begin{array}{lll}
Add\,(Zero, y) & \to & y \\
Add\,(Succ\,(x), y) & \to & Succ\,(Add\,(x, y)) \\
Mul\,(Zero, y) & \to & Zero \\
Mul\,(Succ\,(x), y) & \to & Add\,(Mul\,(x, y), y) \\
Con\,(Nil, l) & \to & l \\
Con\,(Cons\,(a, b), l) & \to & Cons\,(a, Con\,(b, l))
\end{array}$$

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

$$Add\,(Add\,(x, y), z) \to Add\,(x, Add\,(y, z))$$

the rewrite system is no longer SN-safe.

Note that although the general scheme has a primitive recursive aspect, it allows the definition of non-primitive functions thanks to the higher-order features available in GTRS: for example, Ackermann's function can be represented.

Example 4.3 Ackermann's function as a $\mathcal{G}$TRS.

$$\begin{array}{lcl} h(0) & = & Succ_0 \\ h(Succ(x)) & = & H_1(h(x)) \\ H(g,0) & = & Ap(g,1) \\ H(g,Succ(y)) & = & Ap(g,H(g,y)) \end{array}$$

Moreover, the rewrite rules of CCL (the rules for S, K, and I in Example 3.9) are *not* recursive, so, in particular, satisfy the scheme. Therefore, even with the severe restrictions imposed on rewrite rules by the general scheme, the class of SN-safe $\mathcal{G}$TRS is Turing-complete, a property that systems without Ap would not possess.

Then the following holds.

Theorem 4.4 (STRONG NORMALISATION THEOREM) *If* $(\Sigma,\mathbf{R})$ *is typeable in* $\vdash_{\mathcal{E}}^{\not\omega}$ *and SN-safe, then any typeable term is strongly normalisable with respect to* $\mathbf{R}$.

4.2 Head-normalisation

As shown in the previous subsection, in a type assignment system without ω the conditions imposed by the general scheme are sufficient to ensure strong normalisation of typeable terms. Unfortunately, the general scheme is not enough to ensure head-normalisation of typeable terms in a type system with ω: take the rewrite system

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

that is typeable with respect to the environment

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

then $B \vdash_{\mathrm{E}} F(A(A_0,G_0)) : \sigma$, but

$$F(A(A_0,G_0)) \to_{\mathbf{R}}^{*} F(G(A(A_0,G_0))) \to_{\mathbf{R}} F(A(A_0,G_0)).$$

The underlying problem is that, using ω, there are two kinds of typeable recursion in $\mathcal{G}$TRS: the one explicitly present in the syntax, as well as the one obtained by the so-called *fixed-point combinators*; for every H that has type $\omega{\to}\sigma$, the term $A(A_0,H_0)$ has type σ, and $A(A_0,H_0) \to_{\mathbf{R}}^{*} H(A(A_0,H_0))$. In fact, the term $A_1(A_0)$ corresponds to $\Theta = (\lambda xy.y(xxy))(\lambda xy.y(xxy))$, Turing's fixed-point combinator for LC.

So, to obtain a head-normalisation theorem in a type system with ω, we will have to impose stronger conditions than just those imposed by the general scheme. In this subsection we will only consider environments that map constructors to types without type variables and ω. In other words, we will consider those $\mathcal{G}$TRS having an alphabet with a set $\mathcal{C}$ of constructors, such that, for every environment $\mathcal{E}$, if $H \in \mathcal{C}$ then $\mathcal{E}(H) = s_1{\to}\ldots{\to}s_n{\to}s$, where n is the arity of H and $s_1,\ldots,s_n,s$ are sorts, i.e. type constants. In the following, $\mathcal{E}$ will denote an environment satisfying this condition.

Definition 4.5 (HNF-SCHEME) A rewrite rule

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

satisfies the *HNF-scheme* in the environment $\mathcal{E}$, if it satisfies the conditions of the general scheme, where we replace the condition of Definition 4.1:

" for $j \in \underline{m}$, $\vec{C}[\vec{x}] \rhd_{mul} \vec{C_j}[\vec{x}]$ "
by the condition:
" for $j \in \underline{m}$, $\vec{C}[\vec{x}] \rhd_{mul} \vec{C_j}[\vec{x}], \vec{C}[\vec{x}], \vec{C_j}[\vec{x}] \in T(\mathcal{C}, \mathcal{X})$, and the patterns (see Definition 2.4(i)) appear at positions where $\mathcal{E}(F^i)$ requires arguments of sort type' "

The systems that satisfy the HNF-scheme will be called *HNF-safe*.

With this restriction, we can prove the following.

Theorem 4.6 (HEAD NORMALISATION THEOREM) *If the HNF-safe $(\Sigma, \mathbf{R})$ is typeable in $\vdash_{\mathrm{E}}$ and HNF-safe, then $\mathcal{H}t$ for every term t such that $B \vdash_{\mathrm{E}} t : \sigma$ and $\sigma \neq \omega$.*

4.3 Normalisation

In the intersection system for LC it is well-known that terms that are typeable without ω in basis and type are normalisable. This is not true in the rewriting framework, even if one considers HNF-safe recursive systems only. Take for instance the HNF-safe system:

$$\begin{array}{lcl} Z(x,y) & \to & y \\ D(x) & \to & Ap(x,x). \end{array}$$

The term $Z_1(D(D_0))$ has type $\varphi \to \varphi$ in an environment where Z is typed with $\varphi_1 \to \varphi_2 \to \varphi_2$ and D with $(\varphi_3 \to \varphi_4) \cap \varphi_3 \to \varphi_4$, but is not normalisable. The characterisation of normalisation can therefore only be obtained for a restricted class of terms. We will consider only non-Curryfied terms and GTRS where reduction is closed on non-Curryfied terms (the latter will be called non-Curryfied GTRS). Actually, to get a normalisation result similar to that of LC we will also need to impose the following condition on GTRS:

Definition 4.7 A GTRS is *complete* if whenever a typeable non-Curryfied term t of which the type does not contain ω has a reducible subterm $t|_p$ that is typeable with a type containing ω, there exists $q < p$ such that $t|_q$ is typeable with a type without ω and $t|_q[x]_p$ (where x is a fresh variable) is not in head-normal form.

Definition 4.8 Let $(\Sigma, \mathbf{R})$ be a non-Curryfied, HNF-safe and complete GTRS. Let $\succ$ denote the following well-founded ordering between terms: $t \succ t'$ if $t \rhd t'$ or t' is obtained from t by replacing the subterm $t|_p = F(t_1, \ldots, t_n)$ by the term $F(s_1, \ldots, s_n)$ where $\{t_1, \ldots, t_n\} \rhd_{mul} \{s_1, \ldots, s_n\}$. We define the ordering $>>_{\mathcal{NF}}$ on triples composed of a natural number and two terms, as the object $(>_{\mathbb{N}}, \trianglerighteq, \succ)_{lex}$.

Theorem 4.9 (NORMALISATION THEOREM) *Let t be a non-Curryfied term in a typeable, and NF-safe GTRS. If $B \vdash_{\mathrm{E}} t : \sigma$ and ω does not appear in σ, then t is normalisable.*

5 Approximation and normalisation properties

This section gives the construction of the proof of the approximation theorem.

In order to define approximants of terms, we start by introducing a special symbol $\bot$ (*bottom*) into the language (so $\bot$ is not in $\mathcal{X}$, and neither in $\mathcal{F}$), that is intended to represent meaningless terms. (The definition of this new set of terms $T(\mathcal{F}, \mathcal{X}, \bot)$ is straightforward.) To define type assignment on $T(\mathcal{F}, \mathcal{X}, \bot)$, the type assignment rules given in Def. 3.4 need not be changed, it suffices that terms are allowed to be in $T(\mathcal{F}, \mathcal{X}, \bot)$. Since $\bot \notin \mathcal{F} \cup \{Ap\}$, this implies that $\bot$ can only be given the type ω, or appear in subterms that are given the type ω.

Terms in $T(\mathcal{F}, \mathcal{X}, \bot)$ can be ordered using the relation $\sqsubseteq$:

Definition 5.1 *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) $F(t_1,\ldots,t_n) \sqsubseteq F(u_1,\ldots,u_n)$, if and only if, for all $i \in \underline{n}$, $t_i \sqsubseteq u_i$.

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

By abuse of notation, we will use the symbol $\bot$ also for the term-substitution that replaces term-variables by $\bot$: $\bot$ $\{x \mapsto \bot \mid x \in \mathcal{X}\}$. In the following we consider a given $\mathcal{C}$TRS $(\Sigma,\mathbf{R})$, and $Lhs^\bot = \{l^\bot \mid \exists r\, [l \rightarrow r \in \mathbf{R}]\}$.

We will now develop the notion of approximant of a term with respect to a given $\mathcal{C}$TRS. A particular difference with approximants for lambda terms [36] is that our definition is 'static', whereas the other notion was defined as normal forms with respect to an extended notion of reduction. This approach did not work 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 $\bot$.

Definition 5.2 $\mathcal{DA}\,(t)$, the *direct approximant of t* with respect to $(\Sigma,\mathbf{R})$ is defined by:

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

ii) $t = F(t_1,\ldots,t_n)$; let, for $i \in \underline{n}$, $a_i = \mathcal{DA}\,(t_i)$.
$\mathcal{DA}\,(t) = \bot$, 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 $a_1 = F_i(a_1,\ldots,a_i)$; otherwise, $\mathcal{DA}\,(t) = Ap(a_1,a_2)$.

Approximants of terms are obtained by taking direct approximants of their reducts (and making a downward closure).

Definition 5.3 *i*) $\mathcal{DA}$, the set of *approximate normal forms* is defined as

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

ii) A(t), the *set of approximants of t*, is defined by:

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

Intuitively, the terms whose only approximant is $\bot$ are undefined (i.e. meaningless). We will see below (Corollary 5.11) that typeable terms cannot be undefined. A particular problem for this result was that the normal approach, i.e. reasoning on terms, did not give a solution, since the join a two SN-terms is not always an SN-term. Take for example the $\mathcal{C}$TRS:

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

then $E(\bot,D_0) \sqcup E(D_0,\bot) = E(D_0,D_0)$. Notice that $E(D_0,D_0) \rightarrow D(D_0) \rightarrow D(D_0) \rightarrow \cdots$, $E(\bot,D_0) \rightarrow Ap(\bot,D_0)$, and $E(D_0,\bot) \rightarrow D(\bot) \rightarrow Ap(\bot,\bot)$, and therefore that the terms $E(\bot,D_0)$ and $E(D_0,\bot)$ are both strongly normalisable, while $E(D_0,D_0)$ is not. The solution to this apparent problem lies in the fact that the term $E(D_0,D_0)$ is, in $\vdash_{\text{E}}$, only typeable by ω. By defining reduction on derivations rather than on terms, the difficulty is overcome, since the derivation for $\vdash_{\text{E}} E(D_0,D_0) : \omega$ is without redexes, so in normal form.

In this section, we will show the approximation theorem

“ *If $B \vdash_{\text{E}} t : \sigma$, then there exists $a \in \mathcal{A}(t)$ such that $B \vdash_{\text{E}} a : \sigma$,* ”

as well as the three normalisation properties stated above, are consequences of a strong normalisation result for derivation reduction.

The approximation result has been reached also in [5] for the essential system for LC, $\vdash_{\lambda\cap}$. That result, however, cannot be transferred to typed GTRS, and neither can the there used technique. The crucial point in the problem is that the property *“If z does not occur in M, then ‘there is an $A \in \mathcal{A}(Mz)$ such that $B, z : \alpha \vdash_{\lambda\cap} A : \sigma$’ implies ‘there is an $A \in \mathcal{A}(M)$ such that $B \vdash_{\lambda\cap} A : \alpha{\to}\beta$’.”* is relatively easy to prove, since the following holds: *“If $A \in \mathcal{A}(Mz)$ and $z \notin FV(M)$, then either: $A \equiv A'z$ & $z \notin FV(A')$ & $A' \in \mathcal{A}(M)$, or $\lambda z.A \in \mathcal{A}(M)$.”*

The first of these properties is hard to prove in arbitrary GTRS, because there is no known way to express abstraction adequately in GTRS that are not combinatory complete. Take, for example, the term $S_2(K_0,y)$, $B = \{z : \alpha\}$, and notice that

$$\frac{\dfrac{\overline{B \vdash_{\text{E}} K_0 : \alpha{\to}\omega{\to}\alpha} \qquad \overline{B \vdash_{\text{E}} y : \omega}}{B \vdash_{\text{E}} S(K_0,y) : \alpha{\to}\alpha} \qquad \dfrac{}{B \vdash_{\text{E}} z : \alpha}\,(z : \alpha \in B)}{B \vdash_{\text{E}} Ap(S(K_0,y),z) : \alpha}$$

Notice that $\{\bot,z\} = \mathcal{A}(Ap(S(K_0,y),z))$ and also $\{z : \alpha\} \vdash_{\mathcal{E}_{\text{CL}}} z : \alpha$. Following the above property, since none of the approximants of $Ap(S(K_0,y),z)$ is an application term, we would then like to obtain, with $a = \llbracket \lambda z.\langle z\rangle_\lambda \rrbracket_{\text{CL}}$ (where $\llbracket\,\rrbracket_{\text{CL}}$ is the mapping that translates lambda terms to terms in CCL, and $\langle\rangle_\lambda$ is its inverse), that $a \in \mathcal{A}(S_2(K_0,y))$ and $\emptyset \vdash_{\mathcal{E}_{\text{CL}}} a : \alpha{\to}\alpha$. However, $\llbracket \lambda z.\langle z\rangle_\lambda \rrbracket_{\text{CL}} = I$ and $\mathcal{A}(S_2(K_0,y)) = \{\bot, S_2(\bot,\bot), S_2(K_0,\bot), S_2(\bot,y), S_2(K_0,y)\}$.

That is why, to be able to prove the approximation result for GTRS, it was necessary to develop a new technique. We proved that, for a slight restriction of the intersection type assignment system as defined in this paper, derivation reduction is strongly normalisable; this then gives the desired approximation result in an easy way.

In order to prove the approximation theorem 5.10, we need first to define an auxiliary notion of type assignment, $\vdash_{\text{E}}'$, that differs from $\vdash_{\text{E}}$ in the rule for term-variables only.

Definition 5.4 *Strict type assignment* and *strict derivations* are defined as in Definition 3.4 by a natural deduction system, by replacing rule $(\leq)$ by:

$$(\cap E) : \;\frac{}{B \vdash_{\text{E}}' x : \sigma_i}\;(x : \sigma_1 \cap \cdots \cap \sigma_n \in B, i \in \underline{n})$$

The relation between the two notions of type assingment is formulated by:

Lemma 5.5 *i) If $B \vdash_{\text{E}} t : \sigma$, then there is a B' such that $B \leq B'$, and $B' \vdash_{\text{E}}' t : \sigma$.*
ii) If $B \vdash_{\text{E}}' t : \sigma$, and B' is such that $B' \leq B$, then $B' \vdash_{\text{E}} t : \sigma$.

We have also proved a subject reduction result for this notion of type assignment.

Theorem 5.6 (SUBJECT REDUCTION THEOREM) *Let $(\Sigma, \mathbf{R})$ be a typeable* GTRS *with respect to an environment $\mathcal{E}$. If $B \vdash_{\text{E}}' t : \sigma$ and $t \to_{\mathbf{R}} t'$, then $B \vdash_{\text{E}}' t' : \sigma$.*

For this notion of type assignment $\vdash_{\text{E}}'$, we managed to prove that derivation reduction is strongly normalisable. Reduction derivation is defined through the following.

First, we needed a notion of substitution on derivations (for lack of space, we do not give a formal definition):

Definition 5.7 More precisely, for each of those leaves there is a subderivation

$$\frac{\begin{array}{c}\boxed{\qquad\qquad\qquad}\\ B, x_i : \sigma_i \vdash_{\mathrm{E}}{}' x_i : \rho^i_j\end{array}}{A}$$

$$\frac{\begin{array}{c}\boxed{\quad\mathcal{D}^i_1\quad}\\ B \vdash_{\mathrm{E}}{}' x_i{}^{\mathrm{R}} : \rho^i_1\end{array} \quad \cdots \quad \begin{array}{c}\boxed{\quad\mathcal{D}^i_{m_i}\quad}\\ B \vdash_{\mathrm{E}}{}' x_i{}^{\mathrm{R}} : \rho^i_{m_i}\end{array}}{B \vdash_{\mathrm{E}}{}' x_i{}^{\mathrm{R}} : \sigma_i}$$

The derivation $\mathcal{D}\,[\mathcal{D}_i / x_i : \sigma_i]$ is defined as the derivation obtained from D by replacing all occurrences of $\mathcal{D}_{i,j}$ such that $\sigma_i = \rho^i_j$ by $\mathcal{D}_i$, and the others by the corresponding $\mathcal{D}^i_j$, and making in t the corresponding replacement of x_i by $x_i{}^{\mathrm{R}}$.

We now give the definition of reduction on derivations.

Definition 5.8 The *derivation reduction* relation, denoted by $\mathcal{D} :: B \vdash_{\mathrm{E}}{}' t : \sigma \rightarrow_{\mathcal{D}} \mathcal{D}' :: B \vdash_{\mathrm{E}}{}' t' : \sigma$, is defined as follows: Suppose there is a rewrite rule $l \rightarrow r$ where $Var(l) = \{x_1, \ldots, x_n\}$, and a subterm of t at position p (denoted $t|_p$) such that: $t|_p = l^{\mathrm{R}} = F(t_1, \ldots, t_m)$ where $\mathrm{R} = \{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}$. Assume moreover that for $t|_p$, D contains at least one subderivation $\mathcal{D}_0$ of the form:

$$\frac{\begin{array}{c}\boxed{\quad\mathcal{D}_1\quad}\\ B \vdash_{\mathrm{E}}{}' u_1 : \sigma_1\end{array} \quad \begin{array}{c}\boxed{\quad\mathcal{D}_n\quad}\\ B \vdash_{\mathrm{E}}{}' u_n : \sigma_n\end{array}}{B \vdash_{\mathrm{E}}{}' F(t_1, \ldots, t_m) : \tau}$$

such that $\tau \neq \omega$, and the root of $\mathcal{D}_0$ is the first occurrence of the statement $t|_p : \tau$ in a path from the root of $\mathcal{D}$ to a leaf. Then by the Theorem 5.6 there exists $\mathcal{D}'_0 :: \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} \vdash_{\mathrm{E}}{}' r : \tau$. Let $\mathcal{D}'$ be obtained from $\mathcal{D}$ by replacing each subderivation $\mathcal{D}_0$ of $t|_p$ satisfying the previous conditions by the corresponding

$$\mathcal{D}'_0\,[\mathcal{D}_1 / x_1 : \sigma_1, \ldots, \mathcal{D}_n / x_n : \sigma_n] :: B \vdash_{\mathrm{E}}{}' r^{\mathrm{R}} : \tau,$$

and propagating the replacement of $t|_p$ by r^{R} along all the derivation tree. Let t' be obtained from t by replacing $t|_p$ with r^{R}. Then we write $\mathcal{D} :: B \vdash_{\mathrm{E}}{}' t : \sigma \rightarrow_{\mathcal{D}} \mathcal{D}' :: B \vdash_{\mathrm{E}}{}' t' : \sigma$.

The reflexive and transitive closure of $\rightarrow_{\mathcal{D}}$ is denoted by $\rightarrow_{\mathcal{D}}{}^*$, and we write $SN\,(\mathrm{D})$ to indicate that $\mathcal{D}$ is strongly normalisable with respect to $\rightarrow_{\mathcal{D}}$.

For this notion of reduction, we are able to show the following termination result.

Theorem 5.9 Strong Normalisation of Derivation Reduction. *If* TRS *is typeable in $\vdash_{\mathrm{E}}{}'$ and HNF-safe, then for every $\mathcal{D} :: B \vdash_{\mathrm{E}}{}' t : \sigma$, SN(D).*

Because of the precise relation between $\vdash_{\mathrm{E}}{}'$ and $\vdash_{\mathrm{E}}$, as formulated by Lemma 5.5, we can prove the following:

Theorem 5.10 Approximation Theorem. *If $(\Sigma, \mathbf{R})$ is typeable in $\vdash_{\mathrm{E}}$ and HNF-safe, then for every t such that $B \vdash_{\mathrm{E}} t : \sigma$ there is an $a \in \mathcal{A}(t)$ such that $B \vdash_{\mathrm{E}} a : \sigma$.*

Corollary 5.11 (TYPEABLE TERMS ARE MEANINGFUL.) *Let $(\Sigma, \mathbf{R})$ be typeable with respect to $\mathcal{E}$ and HNF-safe. If $B \vdash_{\mathrm{E}} t : \sigma$, and $\sigma \neq \omega$, then there exists $a \in \mathcal{A}(t)$ such that $a \neq \bot$.*

As a matter of fact, it is possible to show that both the head-normalisation theorem 4.6 as well as the normalisation theorem 4.9 follow directly from the approximation result 5.10.
Moreover, the strong normalisation result 4.4 is a direct result from 5.9.

6 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 Beta-reduction rule. Terms are defined for this calculus as in Definition 2.2, with the extension of the introduction of abstraction.

Definition 6.1 The set $T(\mathcal{F},\mathcal{X})$ of *terms* 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 $F(t_1,\ldots,t_n) \in T(\mathcal{F},\mathcal{X})$.

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

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 Beta-reduction rule. Again, this definition is similar to Definition 2.4. The differences lie in the fact that now also the β-contraction is added, and that the rules for the Curryfied versions need not longer be added; they can now be expressed using abstraction.

Definition 6.2 (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 $l \rightarrow_r r$. 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.

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 $l \rightarrow_r r$ 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* $\mathrm{C}[t] \rightarrow \mathrm{C}[t']$. 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 6.3 *Type assignment on terms* (with respect to $\mathcal{E}$) is defined as in Definition 3.4 by a natural deduction system, adding the following rule:

$$(\rightarrow I): \ \frac{B, x:\sigma \vdash_{\mathrm{E}} t:\tau}{B \vdash_{\mathrm{E}} \lambda x.t : \sigma \rightarrow \tau} \ (a)$$

(*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 Ch such that $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma = Ch(\mathcal{E},(F))$.

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 [36] and that for terms in TRS [34]. 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 $\bot M \to \bot$.

Definition 6.4 $\mathcal{DA}(t)$, 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 $i \in \underline{n}, a_i = \mathcal{DA}(t_i)$.
$\mathcal{DA}(t) = \bot$, 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 6.5 Take the TRS $+\beta$

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

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

The sets $\mathcal{DA}$ and $\mathcal{A}(t)$ are defined as before.

Example 6.6 Take again the TRS $+\beta$ of Example 6.5. Then

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

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

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

Using the HNF-scheme, we were able to prove:

Theorem 6.7 (APPROXIMATION THEOREM) *If @TRS is typeable in $\mathcal{E}$ and HNF-safe, then for every term t such that $B \vdash_{\mathrm{E}} t : \sigma$, there is an $a \in \mathcal{A}(t)$ such that $B \vdash_{\mathrm{E}} a : \sigma$.*

As in the previous section, this result leads to the head-normalisation result, and a normalisation theorem. The strong normalisation result needs to be proven independently, however.

References

[1] S. Abramsky. The lazy lambda calculus. In *Research topics in functional programming*, pages 65–116. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

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

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

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

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

[6] S. van Bakel. Rank 2 Intersection Type Assignment in Term Rewriting Systems. *Fundamenta Informaticae*, 2(26):141–166, 1996.

[7] S. van Bakel, F. Barbanera, and M. Fernández. Rewrite Systems with Abstraction and β-rule: Types, Approximants and Normalization. In Hanne Riis Nielson, editor, *Programming Languages and Systems – ESOP'96. Proceedings of 6th European Symposium on Programming,* Linköping, Sweden, volume 1058 of *Lecture Notes in Computer Science*, pages 387–403. Springer Verlag, 1996.

[8] S. van Bakel and M. Fernández. Strong Normalization of Typeable Rewrite Systems. In Jan Heering, Karl Meinke, Bernhard Möller, and Tobias Nipkow, editors, *Proceedings of HOA'93. First International Workshop on Higher Order Algebra, Logic and Term Rewriting,* Amsterdam, the Netherlands. *Selected Papers*, volume 816 of *Lecture Notes in Computer Science*, pages 20–39. Springer Verlag, 1994.

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

[10] S. van Bakel and M. Fernández. Approximation and Normalization Results for Typeable Term Rewriting Systems. In Gilles Dowek, Jan Heering, Karl Meinke, and Bernhard Möller, editors, *Proceedings of HOA'95. Second International Workshop on Higher Order Algebra, Logic and Term Rewriting,* Paderborn, Germany. *Selected Papers*, volume 1074 of *Lecture Notes in Computer Science*, pages 17–36. Springer Verlag, 1996.

[11] S. van Bakel and M. Fernández. Normalization Results for Typeable Rewrite Systems. *Information and Computation*, 2(133):73–116, 1997.

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

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

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

[15] 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.

[16] E. Barendsen and S. Smetsers. Conventional and Uniqueness Typing in Graph Rewrite Systems. In R.K. Shyamasunda, editor, *Proceedings of FST&TCS'93. 13$^{\text{th}}$ Conference on Foundations of Software Technology and Theoretical Computer Science,* Bombay, India, volume 761 of *Lecture Notes in Computer Science*, pages 41–52. Springer Verlag, 1993.

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

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

[19] 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.

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

[21] 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.

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

[23] M. Fernández and J.P. Jouannaud. Modular termination of term rewriting systems revisited. In E. Astesiano and A. Tarlecki, editors, *Recent Trends in Data Type Specification. 10th Workshop on Specification of Abstract Data Types,* S. Margherita, Italy, volume 906 of *Lecture Notes in Computer Science*, pages 255–272. Springer Verlag, 1994.

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

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

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

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

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

[29] 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.

[30] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[31] 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.

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

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

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

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

[36] 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.