Polymorphic Intersection Type Assignment for Rewrite Systems with Abstraction and β-rule[*]

Extended Abstract

(TYPES'99, LNCS 1956, pages 41-60, 2000)

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

[1] Department of Computing, Imperial College,
180 Queen's Gate, London SW7 2BZ.
[2] Dipartimento di Matematica, Università degli Studi di Catania,
Viale A. Doria 6, 95125 Catania, Italia.
[3] LIENS (CNRS URA 8548), Ecole Normale Supérieure,
45, rue d'Ulm, 75005 Paris, France.
maribel@ens.fr

Abstract

We define two type assignment systems for first-order rewriting extended with application, λ-abstraction, and β-reduction (TRS$+\beta$). The types used in these systems are a combination of (ω-free) intersection and polymorphic types. The first system is the general one, for which we prove a subject reduction theorem and show that all typeable terms are strongly normalisable. The second is a decidable subsystem of the first, by restricting types to Rank 2. For this system we define, using an extended notion of unification, a notion of principal type, and show that type assignment is decidable.

Introduction

The combination of λ-calculus (LC) and term rewriting systems (TRS) has attracted attention not only from the area of programming language design, but also from the rapidly evolving field of theorem provers. It is well-known by now that type disciplines provide an environment in which rewrite rules and β-reduction can be combined without loss of their useful properties. This is supported by a number of results for a broad range of type systems [11, 12, 20, 7, 8, 5].

In this paper we study the combination of LC and TRS as a basis for the design of a programming language. The type systems à la Curry, also called *type inference* or *type assignment systems*, are the most interesting ones from this point of view, since they save the programmer from specifying a type for each variable (no type annotations are required). Type assignment disciplines have been widely studied in the context of the LC, and some work has also been done for TRS, more precisely, for *Curryfied* TRS ($\mathcal{C}u$TRS) [6] which are first-order TRS with application, that correspond to the TRS that underlie the programming language Clean [26]. The interactions between LC and TRS in the context of type assignment were first studied in [5], where $\mathcal{C}u$TRS extended with λ-abstraction and β-reduction were defined, together with a notion of intersection type assignment for both the LC and the TRS fragments.

Other important features of a type system for a programming language are *polymorphism*, that is, the possibility of using the same program with arguments of different types, and

[*] Partially supported by NATO Collaborative Research Grant CRG 970285 'Extended Rewriting and Types'.

the existence of *principal types*, that is, a type from which all the other types of the term can be derived. The type system of ML has the three properties above, i.e., it is a *polymorphic* type *inference* system with *principal types*, but its polymorphism is limited (some programs that arise naturally cannot be typed). System F [18] provides a much more general notion of polymorphism, but lacks principal types, and type inference is undecidable in general (although it is decidable for some subsystems, in particular if we consider types of rank 2 [21]). Intersection type systems [10] are somewhere in the middle with respect to polymorphism (they provide less polymorphism than System F but more than ML) and principal types can be constructed for typeable terms. But type assignment is again undecidable; decidability is recovered if we restrict ourselves to the rank 2 types [4].

In view of the above results, two questions arise naturally:

- Is the rank 2 combination of System F and the Intersection System also decidable?
- Does it have principal types for all typeable terms?

A system for the LC that combines intersection types and System F with principal types has been studied in [19, 24]. In this paper we extend that system to a combination of LC and $\mathcal{C}u$TRS. In other words, we extend the type assignment system of [5] further, adding '$\forall$' as an extra type-constructor (i.e. explicit polymorphism). Although extending the set of types by adding '$\forall$' does not extend the expressivity of the system in terms of typeable terms, the set types of assignable types increases, and types can better express the behaviour of terms (see [13]). The resulting system has the expected properties: Subject Reduction, and Strong Normalization when the rewrite rules use a limited form of recursion (inspired by the General Schema of Jouannaud and Okada [20]). The proof of the latter follows the method of Tait-Girard's reducibility candidates, extended in order to take the presence of (higher-order) algebraic rewriting into account.

We also answer the first question in the affirmative. The restriction to types of rank 2 of the combined system of polymorphic and intersection types is decidable. This restricted system can be seen as a combination of the systems considered in [4] and [21]. The combination is twofold: not only the type systems of those two papers are combined (resp. intersection and polymorphic types of Rank 2), but also their calculi are combined (resp. $\mathcal{C}u$TRS and LC). In our Rank 2 system each typeable term has a principal type such that every type derivable for the term can be seen as an instance (under suitable operations) of the type. This is the case also in the Rank 2 intersection system of [4], but not in the Rank 2 polymorphic system of [21]. For the latter, a type inference algorithm of the same complexity of that of ML was given in [22], where the problems that occur due to the lack of principal types are discussed in detail.

This paper is organised as follows: In Section 1 we define TRS with application, λ-abstraction and β-reduction (TRS $+\,\beta$), and in Section 2 the type assignment system. Section 3 deals with the strong normalization property for typeable terms. In Section 4 we present the restriction of the general type assignment system to Rank 2.

1 Term Rewriting Systems with β-reduction

We present a combination of Lambda Calculus with Algebraic Rewriting, obtained by extending first-order TRS with notions of application and abstraction, and β-reduction. We can look at such calculi also as extensions of the Curryfied Term Rewriting Systems ($\mathcal{C}u$TRS) considered in [6], by adding λ-abstraction and a β-reduction rule. We assume the reader to be familiar with LC [9] and refer to [23, 14] for rewrite systems.

We consider terms built from a set $\mathcal{X}$ of variables, a set $\mathcal{F}$ of function symbols, application

(Ap) and λ-abstraction, modulo α-conversion, as usual. A *context* is a term with a hole, and it is denoted, as usual, by $\mathrm{C}[\,]$. A *lambda term* is a term not containing function symbols. An *algebraic term* is a term containing neither λ nor Ap. 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. Reductions are defined through rewrite rules together with a β-reduction rule.

Definition 1.1 (Reduction) A *rewrite rule* is a pair (l,r) of terms, written $l \to r$. Three conditions are imposed: $l \notin \mathcal{X}$, l is an algebraic term, and $FV(r) \subseteq FV(l)$. The β-rule is defined by: $Ap(\lambda x.t,u) \quad \to_\beta \quad t^{\{x \mapsto u\}}$. A rewrite rule $l \to r$ determines a set of *rewrite steps* $\mathrm{C}[l^{\mathrm{R}}] \to \mathrm{C}[r^{\mathrm{R}}]$ for all term-substitutions R and contexts $\mathrm{C}[\,]$. Likewise, for any t and u, $\mathrm{C}[Ap(\lambda x.t,u)] \to_\beta \mathrm{C}[t^{\{x \mapsto u\}}]$. Concatenating rewrite steps we have *rewrite sequences* $t_0 \to t_1 \to t_2 \to \cdots$. If $t_0 \to \cdots \to t_n$ ($n \geq 0$) we also write $t_0 \to^* t_n$, and $t_0 \to^+ t_n$ if $t_0 \to^* t_n$ in one step or more.

A *Term Rewriting System with β-reduction rule* (TRS$+\beta$) is defined by a set **R** of rewrite rules.

Note that, in contrast with $\mathcal{C}\!\mathcal{U}$TRS, the rewrite rules considered in this paper may contain λ-abstractions in the right-hand sides. We take the view that in a rewrite rule a certain symbol is defined: F is *a defined symbol*, if there is a rewrite rule $F(t_1,\ldots,t_n) \to r$. $Q \in \mathcal{F}$ is called a *constructor* if Q is not a defined symbol. Notice that Ap cannot be a defined symbol since it cannot appear in the left-hand side of a rewrite rule.

A TRS$+\beta$ is strongly normalizing, or terminating, if all the rewrite sequences are finite. It is confluent if for all t such that $t \to^* u$ and $t \to^* v$, there exists s such that $u \to^* s$ and $v \to^* s$.

Example 1.2 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.

$$\begin{array}{rcl}
\mathsf{append}(\mathsf{nil},l) & \to & l \\
\mathsf{append}(\mathsf{cons}(x,l),l') & \to & \mathsf{cons}(x,\mathsf{append}(l,l')) \\
\mathsf{append}(\mathsf{append}(l,l'),l'') & \to & \mathsf{append}(l,(\mathsf{append}(l',l''))) \\
\mathsf{map}(f,\mathsf{nil}) & \to & \mathsf{nil} \\
\mathsf{map}(f,\mathsf{cons}(y,l)) & \to & \mathsf{cons}(Ap(f,y),\mathsf{map}(f,l))
\end{array}$$

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

2 A Polymorphic Intersection System for TRS$+\beta$

We define a type assignment system for TRS$+\beta$, that can be seen as an extension (by adding $\forall$) of the intersection system presented in [5]. We use polymorphic strict intersection types, defined over a set of type-variables and sorts (type constants). The $\forall$-free, λ-free fragment of this system corresponds to the system studied in [6]. If sorts are not taken into account, the $\forall$-free LC-fragment of our type assignment system corresponds to the system presented in [3], and the intersection-free LC-fragment to System F [18].

For LC, a type assignment system that combines intersection types with polymorphic types has been defined in [19] and its principal type property has been studied in [24]. As far as types are concerned, the difference between our system and the latter is that we add constant types, and use *strict* intersection types [3] (i.e. an intersection type cannot be the right-hand side of an arrow type). The restriction to strict intersection types simplifies the typing procedures and does not affect the typing power of the system, because any term typeable in the full intersection type discipline can be given a strict type and vice-versa. We assume the reader to be familiar with intersection type assignment systems, and refer to [10, 1, 3] for more details.

2.1 Types

Definition 2.1 (Types) Let $V = \Phi \uplus \mathcal{A}$ be a set of *type-variables*, where $\Phi = \{\varphi_0, \varphi_1, \ldots\}$ is the set of *free* variables, and $\mathcal{A} = \{\alpha_0, \alpha_1, \ldots\}$ the set of *bound* variables. Let $\mathcal{S} = \{s_0, s_1, \ldots\}$ be a set of sorts. $\mathcal{T}_s$, the set of *polymorphic strict types*, and $\mathcal{T}$, the set of *polymorphic strict intersection types*, are defined by mutual induction:

$$\begin{array}{lcl} \mathcal{T}_s & ::= & \varphi \mid \alpha \mid s \mid (\mathcal{T} \rightarrow \mathcal{T}_s) \mid \forall \alpha . \mathcal{T}_s[\alpha/\varphi] \\ \mathcal{T} & ::= & (\mathcal{T}_s \cap \cdots \cap \mathcal{T}_s) \end{array}$$

For various reasons (definition of operations on types, definition of unification), we distinguish syntactically between free type-variables (in Φ) and bound type-variables (in $\mathcal{A}$). As usual '$\rightarrow$' associates to the right, and '$\cap$' binds stronger than '$\rightarrow$', which binds stronger than '$\forall$'; so $\rho\cap\mu\rightarrow\forall\alpha.\gamma\rightarrow\delta$ stands for $((\rho\cap\mu)\rightarrow(\forall\alpha.(\gamma\rightarrow\delta)))$. Also $\forall\overline{\alpha}.\sigma$ is used as abbreviation for $\forall\alpha_1.\forall\alpha_2\ldots\forall\alpha_n.\sigma$, and we assume that each variable is bound at most once in a type (renaming if necessary). In the meta-language, we denote by $\sigma[\tau/\varphi]$ (resp. $\sigma[\tau/\alpha]$) the substitution of the type-variable φ (resp. α) by τ in σ.

FV(σ), the set of *free variables* of a type σ is defined as usual (note that by construction, $FV(\sigma) \subseteq \Phi$). A type is called *closed* if it contains no free variables, and *ground* if it contains no variables at all.

Definition 2.2 (Relations on types) On $\mathcal{T}$, the pre-order (i.e. reflexive and transitive relation) $\leq$ is defined by:

$$\begin{array}{cc} \forall n \geq 1, \forall 1 \leq i \leq n \ [\sigma_1\cap\cdots\cap\sigma_n \leq \sigma_i] & \sigma \leq \tau \ \Rightarrow \ \forall\alpha.\sigma[\alpha/\varphi] \leq \forall\alpha.\tau[\alpha/\varphi] \\ \forall\alpha.\sigma\rightarrow\tau \leq \sigma\rightarrow\forall\alpha.\tau, (\alpha \text{ not in } \sigma) & \forall\alpha.\sigma \leq \sigma[\tau/\alpha] \\ \rho \leq \sigma \,\&\, \tau \leq \mu \ \Rightarrow \ \sigma\rightarrow\tau \leq \rho\rightarrow\mu & \sigma \leq \forall\alpha.\sigma, (\alpha \text{ is fresh}) \\ \multicolumn{2}{c}{\forall n \geq 1, \forall 1 \leq i \leq n \ [\sigma \leq \sigma_i] \ \Rightarrow \ \sigma \leq \sigma_1\cap\cdots\cap\sigma_n} \end{array}$$

The equivalence relation $\sim$ is defined by: $\sigma \sim \tau \iff \sigma \leq \tau \leq \sigma$. We work with types modulo $\sim$.

To obtain a notion of type assignment that is a true extension of System F, the $\forall$ type-constructor is allowed to occur on the right of an arrow, so a type like $\sigma\rightarrow\forall\alpha.\tau$ is well-defined. Also note that we cannot quantify intersection types, but we have equivalent types of the form $(\forall\alpha.\sigma_1)\cap(\forall\alpha.\sigma_2)$.

2.2 Type assignment

Definition 2.3 (Statement and Basis) *i*) A *statement* is an expression of the form $t{:}\sigma$, where $\sigma \in \mathcal{T}$. The term 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.

iii) For bases $B_1, \ldots, B_n$, $\Pi\{B_1, \ldots, B_n\}$ is the basis defined by: $x{:}\sigma_1\cap\cdots\cap\sigma_m \in \Pi\{B_1, \ldots, B_n\}$ 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 $B \sim B'$ if and only if $B \leq B' \leq B$.

We will often write $B, x{:}\sigma$ for the basis $\Pi\{B, \{x{:}\sigma\}\}$, when x does not occur in B, and write $B\backslash x$ for the basis obtained from B by removing the statement that has x as subject.

To assign types to terms we are going to consider an *environment*, and we will define type assignment in such a way that the types used for each occurrence of a function symbol will be consistent with the type provided for it by the environment.

Definition 2.4 (Environment) An *environment* is a mapping $\mathcal{E} : \mathcal{F} \to \mathcal{T}_s$.

As an abstraction of a type derivation for a term t we will use a triple $\langle B,\tau,E\rangle$ where B is a basis, τ a type, and E a set of types. These represent the types in a derivation that are affected by the operations we define below. In order to obtain valid instances of the types provided by the environment we are going to use *operations* on types (that extend to triples $\langle B,\tau,E\rangle$). In type systems based on arrow types with type-variables, the operation of *substitution* generates all the valid instances of a given type. In a system with intersection types, all the intersections of those instances should also be considered valid instances, which means that substitution alone is not enough to generate all the instances of a given type. We will also use *expansion*, which replaces (sub)types by the intersection of a number of renamed copies of that type, and *lifting*, which replaces basis and type by a smaller basis and a larger type, in the sense of $\leq$ (see [2] for details). These operations are standard in type systems with intersection types, we will extend them to take into account the presence of universal quantifiers.

The operation of lifting can be used not only to eliminate intersections but also quantifiers, since our $\leq$ relation takes into account universal quantifiers. We will introduce a fourth operation, called *closure*, to deal with the introduction of universal quantifiers.

Definition 2.5 The *substitution* $(\varphi \mapsto \rho) : \mathcal{T} \to \mathcal{T}$, where φ is a type-variable in Φ and $\rho \in \mathcal{T}_s$, applied to σ replaces all the occurrences of φ in σ by ρ. Substitutions extend to bases and triples in the natural way.

The operation of expansion deals with the replacement of a subtype of a type by an intersection of a number of renamed copies of that subtype. An expansion is determined by a pair $\langle\mu,n\rangle$ which indicates the subtype to be expanded and the number of copies that have to be generated. When a subtype is expanded new type variables are generated, and other subtypes might be affected (e.g. the expansion of τ in $\sigma{\to}\tau$ might affect also σ: intuitively, each renamed copy of τ will have an associated copy of σ; see [28] for a detailed explanation). Ground types are not affected by expansions since all renamed copies coincide (and $\sigma{\cap}\sigma \sim \sigma$). Before applying an operation of expansion, we need then to compute the set of types that will be affected by it, which is done with respect to a given triple $\langle B,\tau,E\rangle$.

Two different definitions of expansion appear in the literature for LC, depending on whether one uses a set of types (see e.g. [28]) or a set of type variables (see e.g. [3]) to characterise the set of types affected by the expansion. We will consider the definition given by Ronchi della Rocca and Venneri [28] for full intersection types without sorts and universal quantifiers, and adapt it to our system. The extension to deal with types containing sorts has already been done in [15], here we add quantifiers. For an expansion $\langle\mu,n\rangle$ and a triple $\langle B,\sigma,E\rangle$, an associated set $\mathcal{L}_\mu(\langle B,\sigma,E\rangle)$ of types is computed, and the types modified by the expansion are those that 'end' with a type in this set. The notion of *last subtypes* in a strict type plays an important rôle in this operation.

Definition 2.6 The set of *last subtypes* of a type $\tau \in \mathcal{T}_s$, denoted by $last(\tau)$, is defined by:

- For a type variable φ, $last(\varphi) = \{\varphi\}$
- For a sort s, $last(s) = \{s\}$
- $last(\sigma \to \rho) = \{\sigma \to \rho\} \cup last(\rho)$
- $(\forall\alpha.\sigma) = \{\forall\alpha.\sigma\} \cup last(\sigma[\varphi_\alpha/\alpha])$

Note that for types of the form $\forall\alpha.\sigma$, σ is not a well-formed subtype according to our convention (free variables must belong to Φ). For this reason we consider a mapping $\mathcal{A} \to \Phi$ that associates to each α a different fresh $\varphi_\alpha \in \Phi$, and rename α in σ, using φ_α. In this way we can define subtypes of types in $\mathcal{T}$, as usual.

The definition of expansion, already non-trivial in the intersection system, becomes quite involved in the presence of universal quantifiers. We define it in three steps. First we compute the set $\mathcal{L}$ of types affected by the expansion. Then we see which are the variables that will need to be renamed. Finally, to apply an expansion to a type τ, we traverse τ top-down searching for maximal subtypes whose last subtypes are in $\mathcal{L}$ or have an instance (obtained by replacing bound variables by types) in $\mathcal{L}$. Those subtypes of τ will be replaced by intersections of renamed copies.

Definition 2.7 (Expansion in $\mathcal{T}$) Let μ be a type in $\mathcal{T}$, $n \geq 2$, B a basis, $\sigma \in \mathcal{T}$ and E a set of types in $\mathcal{T}$ (we assume that each variable is bound at most once in μ, B, σ, E). The pair $\langle\mu, n\rangle$ determines an *expansion Ex* with respect to $\langle B, \sigma, E\rangle$, defined as follows.

i) Let $\mathcal{L}_\mu(\langle B, \sigma, E\rangle)$ be the set of types defined by:

a) Any non-closed strict subtype of μ is in $\mathcal{L}_\mu(\langle B, \sigma, E\rangle)$.

b) Let τ be a non-closed strict (sub)type occurring in $\langle B, \sigma, E\rangle$. If τ' is a most general instance (with respect to the universal quantifiers) of τ such that $last(\tau') \cap \mathcal{L}_\mu(\langle B, \sigma, E\rangle) \neq \emptyset$, then $\tau' \in \mathcal{L}_\mu(\langle B, \sigma, E\rangle)$.

c) Any non-closed proper strict subtype of $\tau \in \mathcal{L}_\mu(\langle B, \sigma, E\rangle)$ is in $\mathcal{L}_\mu(\langle B, \sigma, E\rangle)$.

ii) Let $\mathcal{V}_\mu(\langle B, \sigma, E\rangle) = \{\varphi_1, \ldots, \varphi_m\}$ be the set of free type variables occurring in B, σ, μ, E that appear in $\mathcal{L}_\mu(\langle B, \sigma, E\rangle)$, and let S_i $(1 \leq i \leq n)$ be the substitution that replaces every φ_j by a fresh variable φ^i_j, and every α_j and φ_{α_j} by α^i_j (actually, S_i is not a substitution according to our definition, it is just a renaming).

iii) For any $\tau \in \mathcal{T}$ (without loss of generality we assume that its bound variables are disjoint with those of μ, B, σ, τ, E), $Ex(\tau)$ is obtained by traversing τ top-down and replacing, in τ, a maximal non-closed subtype β such that there exists a most general instance (w.r.t. the universal quantifiers) β' of β with $last(\beta') \cap \mathcal{L}_\mu(\langle B, \sigma, E\rangle) \neq \emptyset$

a) by $S_1(\beta) \cap \cdots \cap S_n(\beta)$ if $\beta' = \beta$,

b) otherwise by

$$\bigcap_{1 \leq j \leq p} (S_1(\beta'_j) \cap \cdots \cap S_n(\beta'_j) \cap \forall\overline{S_i(\overline{\alpha})}Ex(\rho[\overline{c_j}/\overline{\alpha}])[\overline{\alpha}/\overline{c_j}])$$

if $\beta = \forall\overline{\alpha}.\rho$, β'_j $(1 \leq j \leq p)$ are all the most general instances of β satisfying the condition, and $\overline{c_j}$ are fresh constants replacing the variables instantiated in the instance β'_j of β.

Expansions extend to triples representing type derivations in the natural way.

Some explanations are in order. The result of an operation of expansion is not unique because it depends on the choice of new variables in part (*ii*) of the definition; but it is unique modulo renaming of variables (and this is sufficient for our purpose). It is always a type in $\mathcal{T}$: we never introduce an intersection at the right-hand side of an arrow type, and never quantify an intersection type (see part (*iii*)). A type might be affected by an expansion even if its free variables are disjoint with those of the subtype to be expanded. The reason is that universally quantified variables represent an infinite set of terms (their instances), so if one instance is affected, the whole type is affected. If we are applying an expansion operation to a universally quantified type, some instances may be expanded (if their last subtypes are in the set of computed types) whereas others are not (if their last subtypes are not in the set of computed types). In this case the expansion of the universally quantified type will be the intersection of the expansions of each class of instances. Since there is only a finite set of computed types, the operation is well defined.

Example 2.8 Let γ be $(\varphi_1{\to}\varphi_2){\to}(\varphi_3{\to}\varphi_1){\to}\varphi_3{\to}\varphi_2$, and Ex be the expansion determined by $\langle\varphi_1,2\rangle$ with respect to $\langle\emptyset,\gamma,\emptyset\rangle$. Then $\mathcal{L}_{\varphi_1}(\langle\emptyset,\gamma,\emptyset\rangle) = \{\varphi_1,\varphi_3{\to}\varphi_1,\varphi_3\}$, $\mathcal{V}_{\varphi_1}(\langle\emptyset,\gamma\rangle) = \{\varphi_1,\varphi_3\}$, and (to save space, we write ${}^j_i$ instead of φ^j_i)

$$Ex(\gamma) = (({}^1_1{\cap}{}^2_1){\to}{}_2){\to}(({}^1_3{\to}{}^1_1) \cap ({}^2_3{\to}{}^2_1)){\to}({}^1_3{\cap}{}^2_3){\to}{}_2.$$

Let now γ be $\forall\alpha_2\forall\alpha_3.({}_1{\to}\alpha_2){\to}(\alpha_3{\to}{}_1){\to}\alpha_3{\to}\alpha_2$, and Ex be the same expansion. Then $\mathcal{L}_1(\langle\emptyset,\gamma,\emptyset\rangle) = \{{}_1,\forall\alpha_3.({}_1{\to}{}_1){\to}(\alpha_3{\to}{}_1){\to}\alpha_3{\to}{}_1,({}_1{\to}{}_1){\to}({}_{\alpha_3}{\to}{}_1){\to}{}_{\alpha_3}{\to}{}_1,$
${}_1{\to}{}_{1,\alpha_3}{\to}{}_{1,\alpha_3}\}$, and

$$\begin{array}{rcl} Ex(\gamma) & = & (\forall\alpha_3.({}^1_1{\to}{}^1_1){\to}(\alpha_3{\to}{}^1_1){\to}\alpha_3{\to}{}^1_1) \cap (\forall\alpha_3.({}^2_1{\to}{}^2_1){\to}(\alpha_3{\to}{}^2_1){\to}\alpha_3{\to}{}^2_1)\cap \\ & & (\forall\alpha_2\forall\alpha^1_3\forall\alpha^2_3.({}^1_1{\cap}{}^2_1{\to}\alpha_2){\to}((\alpha^1_3{\to}{}^1_1) \cap (\alpha^2_3{\to}{}^2_1)){\to}\alpha^1_3 \cap \alpha^2_3{\to}\alpha_2). \end{array}$$

For types in $\mathcal{T}$ without sorts and $\forall$, the operation of expansion defined by Ronchi della Rocca and Venneri [28] gives the same results as ours, modulo the relation $\sim$ defined for full intersection types (but the representatives of equivalence classes chosen in [28] are not always types in $\mathcal{T}$).

The operation of lifting allows us to eliminate intersections and universal quantifiers, using the $\leq$ relation.

Definition 2.9 An operation of *lifting* is denoted by a pair $L = \langle\langle B_0,\tau_0\rangle,\langle B_1,\tau_1\rangle\rangle$ such that $\tau_0 \leq \tau_1$ and $B_1 \leq B_0$. L can be applied to a type, or a basis:

$$\begin{array}{rclll} L(\sigma) & = & \tau_1, & \text{if } \sigma = \tau_0 & \quad L(B) = B_1, \text{ if } B = B_0 \\ L(\sigma) & = & \sigma, & \text{otherwise} & \quad L(B) = B, \text{ otherwise} \end{array}$$

It is extended to triples in that it does not affect the added parameter.

The operation of closure introduces quantifiers, taking into account the basis where a type is used.

Definition 2.10 A *closure* is an operation characterised by three parameters $Cl = \langle B,\sigma,\varphi\rangle$. It is defined by:

$$\begin{array}{rcll} Cl(\tau) & = & \tau, & \text{if } \tau \neq \sigma \\ Cl(\sigma) & = & \forall\alpha.\sigma[\alpha/\varphi], & \text{if } \varphi \text{ does not appear in } B \ (\alpha \text{ is a fresh variable}) \\ Cl(\sigma) & = & \sigma, & \text{otherwise} \end{array}$$

Cl can also be applied to a basis, or to triples representing type derivations; on bases it is just the identity, as well as on triples when the basis is different from the basis of the closure *Cl*.

The *set Ch of chains* is defined as the smallest set containing expansions, substitutions, liftings, and closures, that is closed under composition. Chains are denoted as $[O_1,\ldots,O_n]$.

Notice that, although the operation of substitution seems redundant, in that one could simulate substitution via closure and lifting, this is only the case for type variables that *do not* *occur in the basis.*

Definition 2.11 (Type Assignment Rules) *i*) *Type assignment on terms* (with respect to $\mathcal{E}$) is defined by the following natural deduction system in sequent form (where all types displayed are in $\mathcal{T}_s$, except for $\sigma_1,\ldots,\sigma_n$ in rule $(\mathcal{F})$ and σ in rule $(\rightarrow E)$). Note the use of a chain of operations in rule $(\mathcal{F})$.

$$(\leq):\ \frac{x{:}\sigma \in B \quad \sigma \leq \tau}{B \vdash_{\mathcal{E}} x{:}\tau} \qquad (\cap I):\ \frac{B \vdash_{\mathcal{E}} t{:}\sigma_1 \quad \ldots \quad B \vdash_{\mathcal{E}} t{:}\sigma_n}{B \vdash_{\mathcal{E}} t{:}\sigma_1 \cap \cdots \cap \sigma_n}\ (n \geq 1)$$

$$(\rightarrow E):\ \frac{B \vdash_{\mathcal{E}} t_1{:}\sigma \rightarrow \tau \qquad B \vdash_{\mathcal{E}} t_2{:}\sigma}{B \vdash_{\mathcal{E}} Ap(t_1,t_2){:}\tau} \qquad (\rightarrow I):\ \frac{B, x{:}\sigma \vdash_{\mathcal{E}} t{:}\tau}{B \vdash_{\mathcal{E}} \lambda x.t{:}\sigma \rightarrow \tau}$$

$$(\mathcal{F}):\ \frac{B \vdash_{\mathcal{E}} t_1{:}\sigma_1 \quad \ldots \quad B \vdash_{\mathcal{E}} t_n{:}\sigma_n}{B \vdash_{\mathcal{E}} F(t_1,\ldots,t_n){:}\sigma}\ (a)$$

$$(\forall I):\ \frac{B \vdash_{\mathcal{E}} t{:}\sigma}{B \vdash_{\mathcal{E}} t{:}\forall \alpha.\sigma[\alpha/\varphi]}\ (b) \qquad (\forall E):\ \frac{B \vdash_{\mathcal{E}} t{:}\forall \alpha.\sigma}{B \vdash_{\mathcal{E}} t{:}\sigma[\tau/\alpha]}$$

(a): If $F \in \mathcal{F}$, and there exists a chain Ch such that $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma = Ch(\mathcal{E}(F))$.

(b): If φ does not occur (free) in B.

ii) We write $B \vdash_{\mathcal{E}} t{:}\sigma$, and say that t is *typeable*, if and only if this judgement is derivable using the above rules. If E is the set of types assigned to function symbols in such a derivation, we represent it by $\langle B,\sigma,E\rangle$.

The use of an environment in rule $(\mathcal{F})$ introduces a notion of polymorphism for our function symbols, which is an extension (with intersection types and general quantification) of the ML-style of polymorphism. 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.

Note that the rule $(\leq)$ is only defined for variables, and we have a $(\forall E)$-rule for arbitrary terms but not an $(\cap E)$-rule. Indeed, the $(\cap E)$-rule for arbitrary terms can be derived from this system of rules. On the other hand, the $(\forall E)$-rule cannot be derived if it is not present in the system. This asymmetry comes from the fact that our types are strict with respect to intersection, but not with respect to $\forall$.

2.2.1 Type Assignment for Rewrite Rules

To ensure the subject reduction property, as in [6], type assignment on rewrite rules will be defined using the notion of principal pair.

Definition 2.12 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 Ch such that $Ch(\langle P,\pi,E\rangle) = \langle B,\sigma,E'\rangle$.

The typeability of rules ensures consistency with respect to the environment.

Definition 2.13 *i*) We say that $l \rightarrow r \in \mathbf{R}$ with defined symbol F *is typeable with respect to* $\mathcal{E}$, if there are P, and $\pi \in \mathcal{T}$ 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 a TRS$+\beta$ *is typeable with respect to* $\mathcal{E}$, if all $\mathbf{r} \in$ R are.

Note that for a rule $F(t_1,\ldots,t_n) \to r$ to be typeable, $\mathcal{E}(F)$ must be of the form $\sigma_1 \to \ldots \to \sigma_n \to \sigma$. Although $\mathcal{E}(F)$ cannot have an outermost universal quantifier, its free variables play the same role as universally quantified variables (since they can be instantiated by substitution operations). In particular, for the polymorphic identity function I we will use $\mathcal{E}(I) = \varphi \to \varphi$.

Reductions preserve types in our system. To obtain this result, first we need to prove that the operations (substitution, expansion, lifting, and closure) are sound on typeable terms. Using that result, we can prove:

Theorem 2.14 (Subject Reduction) *If* $B \vdash_{\mathcal{E}} t{:}\sigma$ *and* $t \to t'$, *then* $B \vdash_{\mathcal{E}} t'{:}\sigma$.

3 Strong Normalisation

Types serve not only as specifications and as a way to ensure that programs 'cannot go wrong' during execution, but also to ensure that computations terminate. In fact, this is a well-known property of the intersection system for LC, and of System F, but the situation is different in TRS (a rule $t \to t$ may be typeable, although it is obviously non-terminating). Inspired by the work of Jouannaud and Okada [20], who defined a general scheme of recursion that ensures termination of higher-order rewrite rules combined with LC, we define a general scheme for TRS$+\beta$, such that typeability of the rewrite rules in the polymorphic intersection system defined in this paper implies strong normalization of all typeable terms.

Definition 3.1 (General Scheme of Recursion) Let $\mathcal{F}_n = \mathcal{Q} \cup \{F^1,\ldots,F^n\}$, where $F^1, \ldots, F^n$ will be the defined symbols, and $\mathcal{Q}$ the set of constructors. We will assume that $F^1, \ldots, F^n$ are defined incrementally (i.e. there is no mutual recursion), by typeable rules that satisfy the *general scheme*:

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

where $\overline{x}$, $\overline{y}$ are sequences of variables such that $\overline{x} \subseteq \overline{y}$; $\overline{\mathrm{C}}[\ ]$, C'[ ], $\overline{\mathrm{C}_1}[\ ], \ldots, \overline{\mathrm{C}_m}[\ ]$ are (sequences of) contexts with function symbols in $\mathcal{F}_{i-1}$; and for every $1 \leq j \leq m$, $\overline{\mathrm{C}}[\overline{x}] \rhd_{mul} \overline{\mathrm{C}_j}[\overline{x}]$, where $\lhd$ is the strict sub-term ordering (i.e. $\rhd$ denotes strict super-term) and '*mul*' denotes multi-set extension. Moreover, if $\langle P,\pi \rangle$ is the principal pair of $F^i(\overline{\mathrm{C}}[\overline{x}],\overline{y})$, the types associated to the variables $\overline{y}$ in P are the types of the corresponding arguments of F^i in $\mathcal{E}(F^i)$.

This general scheme is a generalization of primitive recursion. It imposes two main restrictions on the definition of functions: the terms in the multi-sets $\overline{\mathrm{C}_j}[\overline{x}]$ are sub-terms of terms in $\overline{\mathrm{C}}[\overline{x}]$ (this is the 'primitive recursive' aspect of the scheme), and the variables $\overline{x}$ must also appear as arguments in the left-hand side of the rule. Both restrictions are essential to prove the Strong Normalization Theorem below. 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 TRS$+\beta$: for example, Ackermann's function can be represented. Also the rewrite rules of Combinatory Logic are *not* recursive, so, in particular, satisfy the scheme, and therefore the systems that satisfy the scheme have full computational power.

Using the power of the general schema, it is possible to prove the following

Theorem 3.2 (Strong Normalization) *If the rewrite rules satisfy the general schema, any typeable term is strongly normalisable.*

The proof, which we omit for lack of space, can be carried on by using Tait-Girard's method [17] and the techniques devised in [20] in order to cope with some of the difficulties that arise because of the presence of algebraic rewriting.

It is possible to show that if we assume the rules to satisfy the general schema, a typeable TRS$+\beta$ without critical pairs in **R** is locally confluent on typeable terms (we omit also this proof for lack of space), and hence, by Newman's Lemma [25], we can deduce confluence from strong normalization and local confluence.

4 Restriction to Rank 2

In this section, we will present a decidable restriction of the type system, based on types of rank 2. Although the rank 2 intersection system and the rank 2 polymorphic system for LC type exactly the same set of terms [29], their combination results in a system with more expressive power: polymorphism can be expressed directly (using the universal quantifier) and, moreover, every typeable term has a principal type, as we will show below. The latter property does not hold in a system without intersection.

4.1 Rank 2 type assignment

Definition 4.1 We define polymorphic intersection types of Rank 2 in layers: $\mathcal{T}_C$ are Curry types, $\mathcal{T}_C^{\forall}$ are quantified Curry types, $\mathcal{T}_1$, types of rank 1, are intersections of quantified Curry types, and $\mathcal{T}_2$ are types of Rank 2:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathcal{T}_C & ::= & \varphi \mid s \mid (\mathcal{T}_C \rightarrow \mathcal{T}_C) & \mathcal{T}_C^{\forall} & ::= & \mathcal{T}_C \mid (\forall \alpha.\mathcal{T}_C^{\forall}[\alpha/\varphi]) \\ \mathcal{T}_1 & ::= & (\mathcal{T}_C^{\forall} \cap \cdots \cap \mathcal{T}_C^{\forall}) & \mathcal{T}_2 & ::= & \mathcal{T}_C \mid (\mathcal{T}_1 \rightarrow \mathcal{T}_2) \end{array}$$

Below, we will define a unification procedure that will recursively go through types. However, using the sets defined above, not every subtype of type in $\mathcal{T}_2$ is a type in that same set. For example, $\alpha\rightarrow\varphi$ is not a type in any of the sets defined above; however, $\forall\alpha.\alpha\rightarrow\varphi \in \mathcal{T}_C^{\forall}$, and therefore it can be that, when going through types in $\mathcal{T}_2$ recursively, $\alpha\rightarrow\varphi$ has to be dealt with. Since the distinction between free and bound variables is essential, we introduce, for every set $\mathcal{T}_i$ defined above, also the set $\mathcal{T}_i'$ of types, that contains also free occurrences of αs. We will not always use the $'$ when speaking of these sets, however; it will be clear from the context which set is intended.

As for $\mathcal{T}$, we will consider a relation on types, $\leq_2$, that is the restriction to $\mathcal{T}_2$ of the relation $\leq$ defined in Def. 2.2. Notice that part '$\forall\alpha.\sigma\rightarrow\tau \leq_2 \sigma\rightarrow\forall\alpha.\tau$, *if* α *does not occur in* σ' is omitted, since $\sigma\rightarrow\forall\alpha.\tau$ is not a type of Rank 2.

The first three operations used for the Rank 2 system are straightforward variants of operations defined for the full system.

Definition 4.2 *i*) *Substitution* $(\varphi \mapsto \rho) : \mathcal{T}_2 \rightarrow \mathcal{T}_2$ is defined as for the general system, but with the restriction that $\rho \in \mathcal{T}_C$. We use Id_S for the substitution that replaces all variables by themselves, and write $\mathcal{S}$ for the set of all substitutions.

ii) *Lifting* is defined as in Def. 2.9, but with the restriction that $\leq$ is taken to be $\leq_2$.

iii) *Closure* is defined as in Def. 2.10 as $\langle B,\sigma,\varphi\rangle$, with the restriction that $\sigma \in \mathcal{T}_C^{\forall}$.

The variant of expansion used in the Rank 2 system is quite different from that of Def. 2.7. The reason for this is that expansion, normally, increases the rank of a type, a feature that is of course not allowed within a system that limits the rank of types. Since below expansion is only used in very precise situations (within the procedure $unify_2^{\forall}\,()$, the solution is relatively easy: in the context of rank 2 types, expansion is only called on types in $\mathcal{T}_C^{\forall}$, so it is defined to work well there, by replacing *all* types by an intersection; in particular, intersections are not created at the left of an arrow.

Definition 4.3 Let B be a Rank 2 basis, $\sigma \in \mathcal{T}_C^{\forall}$, and $n \geq 1$. The triple $\langle n,B,\sigma\rangle$ determines a n-fold *expansion* with respect to the pair $\langle B,\sigma\rangle$, $Ex_{\langle n,B,\sigma\rangle} : \mathcal{T}_2 \rightarrow \mathcal{T}_2$, that is constructed as follows: Suppose $V = \{\varphi_1,\ldots,\varphi_m\}$ is the set of all (free) variables occurring in $\langle B,\sigma\rangle$. Choose $m \times n$ different variables $\varphi_1^1,\ldots,\varphi_1^n,\ldots,\varphi_m^1,\ldots,\varphi_m^n$, such that each φ_j^i $(1 \leq i \leq n,\ 1 \leq j \leq m)$ does not occur in V. Let S_i be the substitution that replaces every φ_j by φ_j^i. Then $Ex_{\langle n,B,\sigma\rangle}(\tau) = S_1(\tau) \cap \cdots \cap S_n(\tau)$.

Expansion is extended to bases and pairs in the straightforward way.

As before, operations will be grouped in chains.

Definition 4.4 *Rank 2 type assignment on terms* is defined by the following natural deduction system:

$$(\leq):\quad \frac{x{:}\sigma \in B \qquad \sigma \leq_2 \tau}{B \vdash_{\mathcal{E}}^2 x{:}\tau}\ (\sigma \in \mathcal{T}_C^{\forall}, \tau \in \mathcal{T}_C) \qquad (\mathcal{F}):\quad \frac{B \vdash_{\mathcal{E}}^2 t_1{:}\sigma_1 \quad \ldots \quad B \vdash_{\mathcal{E}}^2 t_n{:}\sigma_n}{B \vdash_{\mathcal{E}}^2 F(t_1,\ldots,t_n){:}\sigma}\ (a)$$

$$(\rightarrow E):\quad \frac{B \vdash_{\mathcal{E}}^2 t_1{:}\sigma\rightarrow\tau \qquad B \vdash_{\mathcal{E}}^2 t_2{:}\sigma}{B \vdash_{\mathcal{E}}^2 Ap(t_1,t_2){:}\tau} \qquad (\rightarrow I):\quad \frac{B, x{:}\sigma \vdash_{\mathcal{E}}^2 t{:}\tau}{B \vdash_{\mathcal{E}}^2 \lambda x.t{:}\sigma\rightarrow\tau}$$

$$(\forall I):\quad \frac{B \vdash_{\mathcal{E}}^2 t{:}\sigma}{B \vdash_{\mathcal{E}}^2 t{:}\forall\alpha.\sigma[\alpha/\varphi]}\ (b) \qquad (\cap I):\quad \frac{B \vdash_{\mathcal{E}}^2 t{:}\sigma_1 \quad \ldots \quad B \vdash_{\mathcal{E}}^2 t{:}\sigma_n}{B \vdash_{\mathcal{E}}^2 t{:}\sigma_1\cap\cdots\cap\sigma_n}\ (c)$$

(a): If $F \in \mathcal{F}$, and there exists an expansion-free chain of operations Ch such that $\sigma_1\rightarrow\cdots\rightarrow\sigma_n\rightarrow\sigma = Ch(\mathcal{E}(F))$.

(b): If φ does not occur in B.

(c): If $n \geq 1$, and $\sigma_i \in \mathcal{T}_C^{\forall}$, for every $1 \leq i \leq n$.

Notice that, since quantification elimination is implicit in rule $(\leq)$, when restricting the use of the quantifier to the left of arrows only, there is no longer need for a general $(\forall E)$ rule; as rule $(\cap E)$, its use is in a strict system limited to variables, and there its actions are already performed by $(\leq)$. In fact, this change is justified by the fact that it is possible to show that the operations (substitution, lifting, closure and expansion) are sound with respect to typing.

4.2 Unification of Rank 2 Types

In the context of types, unification is a procedure normally used to find a common instance for demanded and provided type for applications, i.e: if t_1 has type $\sigma\rightarrow\tau$, and t_2 has type ρ, then unification looks for a common instance of the types σ and ρ such that $Ap(t_1,t_2)$ can be typed properly. The unification algorithm $unify_2^{\forall}\,()$ presented in the next definition deals with just that problem. This means that it is not a full unification algorithm for types of Rank 2, but only an algorithm that finds the most general unifying chain for demanded and provided type. It is defined as a natural extension of Robinson's well-known unification algorithm $unify\,()$ [27].

Definition 4.5 Unification of Curry types (extended with non-unifiable variables and type constants), $unify : \mathcal{T}_C' \times \mathcal{T}_C' \rightarrow \mathcal{S}$, is defined by:

$$\begin{array}{lll@{\qquad}lll}
\mathit{unify}\,(\varphi_1,\varphi_2) & = & (\varphi_1 \mapsto \varphi_2) & \mathit{unify}\,(c_1,c_1) & = & Id_S \\
\mathit{unify}\,(\alpha_1,\alpha_1) & = & Id_S & \mathit{unify}\,(\sigma,\varphi) & = & \mathit{unify}\,(\varphi,\sigma) \\
\mathit{unify}\,(\varphi,\tau) & = & \multicolumn{4}{l}{(\varphi \mapsto \tau), \text{ if } \varphi \text{ does not occur in } \tau \text{ and } \tau \text{ is not a variable}} \\
\mathit{unify}\,(\sigma{\to}\tau,\rho{\to}\mu) & = & \multicolumn{4}{l}{S_2{\circ}S_1, \text{ where } S_1 = \mathit{unify}\,(\sigma,\rho), \text{ and } S_2 = \mathit{unify}\,(S_1\,(\tau),S_1\,(\mu))}
\end{array}$$

(All non-specified cases, like $\mathit{unify}\,(\alpha_1,\alpha_2)$, with $\alpha_1 \neq \alpha_2$, fail.)

The unification algorithm $\mathit{unify}_2^{\forall}\,()$ works roughly as follows: suppose we are trying to find a type for the term $Ap\,(t_1,t_2)$, and we know that both t_1 and t_2 are already typed by, respectively, $\sigma{\to}\tau$ and ρ. Thus the demanded type σ is in $\mathcal{T}_1$ and the provided type ρ is in $\mathcal{T}_2$. The unification algorithm looks for types that can be assigned to the terms t_1 and t_2 such that the application term can be typed properly, i.e. looks for types μ and ν such that $\mu{\to}\nu$ is an instance of $\sigma{\to}\tau$, and μ is an instance of ρ. It tries to unify the types σ and ρ, in trying to find a sequence of operations that does the right instantiation. In order to be consistent, the result of the unification of σ and ρ – a chain Ch – should always be such that $Ch(\rho) \in \mathcal{T}_C^{\forall}$. However, if $\rho \notin \mathcal{T}_C^{\forall}$, then in general $Ch\,(\rho) \notin \mathcal{T}_C^{\forall}$.

To overcome this difficulty, an algorithm $\mathit{to}\mathcal{T}_C^{\forall}\,()$ will be inserted that, when applied to the type ρ, returns a chain of operations that removes, if possible, intersections in ρ. This can be understood by the observation that, for example, $((\sigma{\to}\sigma){\to}\sigma{\to}\sigma){\to}\sigma$ is a substitution instance of $((\varphi_1{\to}\varphi_1){\to}\varphi_2) \cap (\varphi_3{\to}\varphi_4{\to}\varphi_4){\to}\varphi_5$. Note that if quantifiers appear in ρ, the $\mathit{to}\mathcal{T}_C^{\forall}\,()$ procedure fails; so quantifiers that appear before an arrow cannot be removed.

The algorithm $\mathit{unify}_2^{\forall}\,()$ is called with the types σ and ρ', the latter being ρ in which the intersections are removed by $\mathit{to}\mathcal{T}_C^{\forall}\,()$. Since none of the derivation rules, nor one of the operations, allows for the removal of a quantifier that occurs *inside* a type, if $\sigma = \forall\overline{\alpha}.\sigma'$, the unification of σ with ρ' will not remove the $\forall\overline{\alpha}$ part.

It is possible that $\sigma \notin \mathcal{T}_C$, so it can be that ρ must be expanded. Since such an operation affects also the basis, the third argument of $\mathit{unify}_2^{\forall}\,()$ is a basis.

Definition 4.6 Let $\mathcal{C}h$ be the set of all chains. $\mathit{to}\mathcal{T}_C^{\forall}\,()$: $\mathcal{T}_2 \to \mathcal{C}h$ is defined by:

$$\begin{array}{rcll}
\mathit{to}\mathcal{T}_C^{\forall}\,(\sigma) & = & [Id_S], & \text{if } \sigma \in \mathcal{T}_C \\
\mathit{to}\mathcal{T}_C^{\forall}\,((\sigma_1\cap\cdots\cap\sigma_n){\to}\mu) & = & [S_1,\ldots,S_{n-1}] * Ch, & \text{otherwise, where,} \\
\text{for every } 1 \le i \le n{-}1, S_i & = & \multicolumn{2}{l}{\mathit{unify}\,([S_1,\ldots,S_{i-1}]\,(\sigma_1),[S_1,\ldots,S_{i-1}]\,(\sigma_{i+1})), \text{ and}} \\
Ch & = & \mathit{to}\mathcal{T}_C^{\forall}\,([S_1,\ldots,S_{n-1}]\,(\mu)) &
\end{array}$$

(Again, notice that $\mathit{to}\mathcal{T}_C^{\forall}\,(\sigma)$ fails if σ contains $\forall$.)

Definition 4.7 Let $\mathcal{B}$ be the set of all bases, and $\mathcal{C}h$ the set of all chains. $\mathit{unify}_2^{\forall} : \mathcal{T}_1 \times \mathcal{T}_C \times \mathcal{B} \to \mathcal{C}h$ is defined by:

$$\begin{array}{l}
\mathit{unify}_2^{\forall}\,((\forall\overline{\alpha_1}.\sigma_1) \cap \ldots \cap (\forall\overline{\alpha_n}.\sigma_n),\tau,B) \;=\; [Ex] * Ch_1 * \cdots * Ch_n, \text{ where} \\
Ex \;=\; n_{\langle B,\tau\rangle}, \qquad \tau_1\cap\cdots\cap\tau_n \;=\; Ex\,(\tau), \text{ and, for every } 1 \le i \le n, \\
S_i \;=\; \mathit{unify}\,([S_1,\ldots,S_{i-1}]\,(\sigma_i),\tau_i) \quad Cl_i \;=\; \langle S_i\,(B),S_i\,(\sigma_i),\overline{\alpha_i}\rangle, \text{ and } Ch_i = [S_i,Cl_i].
\end{array}$$

Notice that $\mathit{unify}\,()$, $\mathit{to}\mathcal{T}_C^{\forall}\,()$, and $\mathit{unify}_2^{\forall}\,()$ all return lifting-free chains.

The closure operations in the definition of $\mathit{unify}_2^{\forall}\,()$ are correct because the variables $\overline{\alpha_i}$ do not appear in B. Notice also that $\mathit{unify}_2^{\forall}\,()$ only fails when $\mathit{unify}\,()$ fails, and that $\mathit{to}\mathcal{T}_C^{\forall}\,()$ fails when either $\mathit{unify}\,()$ fails or when the argument contains $\forall$, and that the other operations never fail. Because of this relation between $\mathit{unify}_2^{\forall}\,()$ and $\mathit{to}\mathcal{T}_C^{\forall}\,()$ on one side, and $\mathit{unify}\,()$ on the other, the procedures defined here are terminating and type assignment in the system defined in this section is decidable.

4.3 Principal pairs for terms

In this subsection, the principal pair for a term t with respect to $\mathcal{E}$ – $PP_{\mathcal{E}}(t)$ – is defined, consisting of basis P and type π. In Theorem 4.9 it will be shown that, for every term, this is indeed the principal one.

Notice that, in the definition below, if $PP_{\mathcal{E}}(t) = \langle P,\pi \rangle$, then $\pi \in \mathcal{T}_2$. For example, the principal pair for the term $\lambda x.x$ is $\langle \emptyset, \varphi \rightarrow \varphi \rangle$, so, in particular, it is not $\langle \emptyset, \forall \alpha.\alpha \rightarrow \alpha \rangle$. Although one could argue that the latter type is more 'principal' in the sense that it expresses the generic character the principal type is supposed to have, we have chosen to use the former instead. This is mainly for technical reasons; because unification is used in the definition below, using the latter type, we would often be forced to remove the external quantifiers. Both types can be seen as 'principal' though, since $\forall \alpha.\alpha \rightarrow \alpha$ can be obtained from $\varphi \rightarrow \varphi$ by closure, and $\varphi \rightarrow \varphi$ from $\forall \alpha.\alpha \rightarrow \alpha$ by lifting.

Definition 4.8 Let t be a term in $T(\mathcal{F},\mathcal{X})$. Using $\mathit{unify}_2^{\forall}()$, $PP_{\mathcal{E}}(t)$ is defined by:

i) $t \equiv x$. Then $PP_{\mathcal{E}}(x) = \langle \{x{:}\varphi\}, \varphi \rangle$.

ii) $t \equiv \lambda x.t'$. Let $PP_{\mathcal{E}}(t') = \langle P,\pi \rangle$, then:

- *a)* If x occurs free in t', and $x{:}\sigma \in P$, then $PP_{\mathcal{E}}(\lambda x.t') = \langle P \backslash x, \sigma \rightarrow \pi \rangle$.
- *b)* Otherwise $PP_{\mathcal{E}}(\lambda x.t') = \langle P, \varphi \rightarrow \pi \rangle$ where φ is a fresh variable.

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

- *a)* If $\pi_1 \in \mathcal{T}_C$ then: $PP_{\mathcal{E}}(Ap(t_1,t_2)) = \langle P,\pi \rangle$, where $P = \langle S_2,S_1 \rangle (\Pi\{P_1,P_2\})$, $S_1 = \mathit{unify}(\pi_1, S_2(\pi_2) \rightarrow \varphi)$, and $\pi = \langle S_2,S_1 \rangle(\varphi)$, and φ is a fresh variable.
- *b)* If $\pi_1 = \sigma \rightarrow \tau$, with $\sigma \in \mathcal{T}_1$, $\tau \in \mathcal{T}_2$, then $PP_{\mathcal{E}}(Ap(t_1,t_2)) = \langle P,\pi \rangle$, where $P = \langle S_2 \rangle * Ch(\Pi\{P_1,P_2\})$, $Ch = \mathit{unify}_2^{\forall}(\sigma, S_2(\pi_2), S_2(P_2))$, and $\pi = \langle S_2 \rangle * Ch(\tau)$.

iv) $t \equiv F(t_1,\ldots,t_n)$. Assume $\mathcal{E}(F) = \gamma_1 \rightarrow \cdots \rightarrow \gamma_n \rightarrow \gamma$, and let, for every $1 \leq i \leq n$, $PP_{\mathcal{E}}(t_i) = \langle P_i,\pi_i \rangle$ (choose, if necessary, trivial variants such that the $\langle P_i,\pi_i \rangle$ are disjoint in pairs), then:

$$\begin{array}{rcl} PP_{\mathcal{E}}(F(t_1,\ldots,t_n)) & = & \langle P,\pi \rangle, \\ \text{where } P & = & [S_1,\ldots,S_n] * Ch_1 * \cdots * Ch_n(\Pi\{P_1,\ldots,P_n\}) \\ S_i & = & \mathit{to}\mathcal{T}_C^{\forall}(\pi_i) \\ Ch_i & = & \mathit{unify}_2^{\forall}(Ch_1 * \cdots * Ch_{i-1}(\gamma_i'), S_i(\pi_i), S_i(P_i)), \\ \pi & = & [S_1,\ldots,S_n] * Ch_1 * \cdots * Ch_n(\gamma') \\ \gamma_1' \rightarrow \cdots \rightarrow \gamma_n' \rightarrow \gamma' & \text{is} & \text{a fresh instance of } \gamma_1 \rightarrow \cdots \rightarrow \gamma_n \rightarrow \gamma. \end{array}$$

Note that, since $\mathit{unify}()$ or $\mathit{unify}_2^{\forall}()$ may fail, not every term has a principal pair.

The main result for the Rank 2 system is the following:

Theorem 4.9 (Soundness and Completeness of $PP_{\mathcal{E}}$) • *If $PP_{\mathcal{E}}(t) = \langle P,\pi \rangle$, then $P \vdash_{\mathcal{E}}^2 t{:}\pi$.*

- *If $B \vdash_{\mathcal{E}}^2 t{:}\sigma$, then there are a basis P and type π such that $PP_{\mathcal{E}}(t) = \langle P,\pi \rangle$, and there is a chain Ch such that $Ch(\langle P,\pi \rangle) = \langle B,\sigma \rangle$.*

References

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

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

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

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

[5] S. van Bakel, F. Barbanera, and M. Fernández. Rewrite Systems with Abstraction and β-rule: Types, Approximants and Normalization. In *ESOP'96*, *LNCS* 1058, pages 387–403, 1996.

[6] S. van Bakel and M. Fernández. Normalization Results for Typeable Rewrite Systems. *I&C* 133(2):73–116, 1997.

[7] F. Barbanera and M. Fernández. Intersection Type Assignment Systems with Higher-Order Algebraic Rewriting. *TCS* 170:173–207, 1996.

[8] F. Barbanera, M. Fernández, and H. Geuvers. Modularity of Strong Normalization and Confluence in the Algebraic λ-cube. In *LICS '94*, 1994.

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

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

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

[12] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic confluence. *I&C* 82:3–28, 1992.

[13] A.Bucciarelli, S. De Lorenzis, A. Piperno, I. Salvo. Some Computational Properties of Intersection Types (Extended Abstract). In *LICS '99*, pages 109–118, 1999.

[14] 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.

[15] M.Fernández. Type Assignment and Termination of Interaction Nets. *Mathematical Structures of Computer Science*, 1998.

[16] M. Fernández and J.P. Jouannaud. Modular termination of term rewriting systems revisited. In *LNCS* 906, pages 255–272, 1994.

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

[18] J.Y. Girard. The System F of Variable Types, Fifteen years later. *TCS* 45:159–192, 1986.

[19] B. Jacobs, I. Margaria, and M. Zacchi. Filter models with polymorphic types. *TCS* 95:143–158, 1992.

[20] J.P. Jouannaud and M. Okada. Executable higher-order algebraic specification languages. In LICS '92, pages 350–361, 1991.

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

[22] A.J. Kfoury and J.B. Wells. A Direct Algorithm for Type Inference in the Rank-2 Fragment of the Second-Order λ-Calculus. In *LFP'94*, 1994.

[23] 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.

[24] I. Margaria and M. Zacchi. Principal Typing in a $\forall\cap$-Discipline. *Logic and Computation*, 5(3):367–381, 1995.

[25] M.H.A. Newman. On theories with a combinatorial definition of 'equivalence'. *Ann. Math.*, 43:223–243, 1942.

[26] E.G.J.M.H. Nöcker, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J. Plasmeijer. Concurrent Clean. In *PARLE '91*, *LNCS* 506-II, pages 202–219, 1991.

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

[28] S. Ronchi Della Rocca and B. Venneri. Principal type schemes for an extended type theory. *TCS* 28:151–169, 1984.

[29] H. Yokohuchi. Embedding a Second-Order Type System into an Intersection Type System. *I&C* 117:206–220, 1995.