Electronic Notes in Theoretical Computer Science 75 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume75.html 22 pages

Rank 2 Types for Term Graph Rewriting

Extended Abstract

Steffen van Bakel

Department of Computing, Imperial College,
80 Queen's Gate, London SW7 2BZ, U.K

Abstract

We define a notion of type assignment with polymorphic intersection types of rank 2 for a term graph rewriting language that expresses sharing and cycles. We show that type assignment is decidable through defining, using the extended notion of unification from [5], a notion of principal pair which generalizes ML's principal type property.

Introduction

This paper presents a decidable notion of type assignment systems for a term-graph rewriting language that uses polymorphic types of rank 2, so allows for more than just the standard *shallow* polymorphism. In order to obtain principal typings, intersection types of rank 2 are added to the system.

In the past, many notions of type assignment have been studied for (functional) programming languages, all based on (extensions of) the Hindley-Milner type assignment system [22,32]. Moreover, almost all notions of type assignment as proposed for use in functional programming, in reality are developed on (enriched) lambda calculi, and little work is available that discusses and studies types directly *on the level of the programming language*. However, to be able to study the role of types in practice, it is arguably important that type assignment is formally defined as close to the actual language as possible.

Furthermore, many aspects of those languages are not easily dealt with in the Lambda Calculus (LC) [8], or not expressible at all, like patterns, sharing, and cyclic structures. This motivated the investigation of type assignment for Term Rewriting Systems (TRS) [30] and Term Graph Rewriting Systems (TGRS) [10] presented in various papers [7,6,4,13,5], and the system presented in this paper. As an example, take the problem of I/O in the context of functional

[1] Email: svb@doc.ic.ac.uk

programming: only when representing terms as graphs to express the *sharing* that is heavily used at run-time does it become possible to represent the number of different references to an object accurately; only when the reference is *unique* (see [13] for a discussion of *uniqueness types*; note that we do not consider a notion of uniquess typing here) is it possible to do a *destructive update*.

The main point of focus for [6,5] was *normalisation*, which motivated the choice to use intersections types [9]. This implied, however, that type assignment for those systems is undecidable. It is by now well-known that there are decidable restrictions of the intersection type assignment system [17,29,23,4,24,18,16,26,27], making the definition of notions of type assignment using those types feasible. In particular, in [4] a notion of type assignment for TRS was presented that uses intersection types of rank 2.

Another direction in the area of types is that of *quantified* or *polymorphic* types. This field originated in the context of LC with System F [21,34], which provides a general notion of polymorphism, but lacks principal typings. Moreover, type inference in System F is undecidable in general [38], although it is decidable for some sub-systems, in particular if we consider types of rank 2 [28]. The type system of ML [15] uses (shallow) polymorphic types and has principal types. Since its polymorphism is limited, some programs that arise naturally cannot be typed, and it does *not* have principal *typings* [24], a property that is important for separate compilation, incremental type inference, and accurate type error messages.

Intersection type systems are somewhere in the middle with respect to polymorphism, and have principal typings.

The system of [4] was in [5] extended to a system for a combination of LC and Curryfied TRS ($\mathcal{CA}$TRS) –a notion of first order TRS extended with application– by adding '$\forall$' as an extra type-constructor (i.e. explicit polymorphism). Although the Rank 2 intersection system and the Rank 2 polymorphic system for LC type exactly the same set of terms [39], their combination results in a system with more expressive power: the set of assignable types increases, and types can better express the behaviour of terms [14]. Also, polymorphism can be expressed directly (using the universal quantifier) and, moreover, every typeable expression in [5] has a principal typing. This principal typing property does not hold in a system without intersection.

The decidability of a notion of unification on polymorphic intersection types of rank 2 as shown in [5] could be used in many different contexts. Since intersection types are the natural tool to type nodes that are shared in a notion of type assignment on graphs, in this paper, we adapt the notion of type assignment of [5] to one for (a kind of) TGRS. (Intersection types also provide a good formalism to express overloading.) We will show that the notion of type assignment as presented here has the principal typing property.

We will study type assignment on a class of graphs that can be defined via an abstract syntax definition, which makes an inductive approach to type

assingment possible. Graphs will be written as terms, and type assignment will be treated on the level of terms. A first treatment of types for graph rewriting systems that uses this approach can be found in [13], which itself is based on the approach of [7] as far as the definition of type assignment is concerned. A draw-back of that system is that it uses the standard Curry types to type graphs, so that the types assignable to a graph are fewer than those assignable to the corresponding tree (obtained by unraveling the graph), since there a node shared in the graph would appear as two separate nodes, that can be typed with different types. Using intersection types, the concept of sharing in graphs causes no difficulties, since a shared node can now be typed with more than one type.

The only problem arises when the graph is allowed to have a cyclic structure, which causes the unraveling to generate an infinite tree. Then it is possible that the (infinite number of) copies of a node are all typed with different types, thus creating an intersection over an infinite number of types for the type assignment to the term graph. The solution for this problem used in this paper is to type a cyclic node with *one* Curry type only, similar to the standard way of dealing with recursion.

In our Rank 2 system each typeable term has a principal typing; this is the case also in the Rank 2 *intersection* system of [4], but not in the Rank 2 *polymorphic* system of [28]. For the latter, a type inference algorithm of the same complexity of that of ML was given in [29], where the problems that occur due to the lack of principal types are discussed in detail. Our Rank 2 system (without the share and the cycle) generalizes also Jim's system P_2 [24], which is a combination of ML-types and Rank 2 intersection types. Having Rank 2 quantified types in the system allows us to type, for instance, the constant `runST` used in [31], which cannot be typed in P_2. Our system also generalises the system of [16] that combines rank 2 intersection types and shallow polymorphism, so does not have polymorphic types of rank 2.

The Rank 2 system as used in this paper can be seen as a combination of the systems of [4] and [28]. In [5] an incomplete notion of polymorphic intersection type assignment was presented for a language that is a combination of LC and $\mathscr{A}$TRS; it contains a definition of a Rank 2 system for that combined calculus, and it claimed to show that type assignment in that system is decidable and has principal types; since there were some major flaws to definitions and proofs in that paper, a new correct presentation is necessary [2]. This paper corrects those definitions and extends those result to a calculus with sharing and cycles, by defining a notion of Rank 2 type assignment on @RGRS, inspired by the system that was studied in [5].

We refer to [30,19] for rewrite systems, and to [12,10,11,25,33,37] for def-

[2] The error mainly was in Lemma 5.1.3(i) of that paper, that stated that, if $\sigma \leq \tau$ and $\sigma \in \mathcal{T}_C$, then $\tau \in \mathcal{T}_C$; this should be: $\tau \in \mathcal{T}_2$. The correction of this resulted in, amongst others, a different type assignment rule (Ax), thereby causing a complete overhaul of the paper.

initions of TGRS. The system defined here is aimed to be similar to those, although their relation is not studied here.

We will use a vector notation $\overrightarrow{g}$ for $g_1, \ldots, g_n$, so $\overrightarrow{\langle x_i = t_i \rangle}$ stands for $\langle x_1 = t_1 \rangle, \ldots, \langle x_n = t_n \rangle$, and $\overrightarrow{x_i \mapsto r_i}$ for $x_1 \mapsto r_1, \ \ldots, x_n \mapsto r_n$, etc.

1 Applicative Term Graph Rewriting Systems

In this section, we will present a notion of *Applicative* Term Graph Rewriting (@TGRS) based on an inductive definition of graphs, following essentially a similar system presented in [13]. Term Graph Rewriting distinguishes itself from Term Rewriting in that the objects considered are no longer trees, but allow sharing and cycles; it is different from Generalised Graph Rewriting in that only those rewrites are allowed that can, essentially, be formulated through a term rewrite rule.

Definition 1.1 (i) An *alphabet* or *signature* Σ consists of a countable, infinite set $\mathcal{X}$ of variables x, y, z, ..., a non-empty set $\mathcal{F}$ of *function symbols* F, G, ..., each with a fixed arity *arity*(F), and a special binary operator, called *application* (@, written in in-fix notation).

(ii) The set $T(\mathcal{F}, \mathcal{X})$ of *terms*, ranged over by t, is defined by:

$$t ::= x \mid \mathsf{F} \mid (t_1 \mathbin{@} t_2) \mid (\mathsf{share}\, t_1 \,\mathsf{via}\, x \,\mathsf{in}\, t_2) \mid (\mathsf{cycle} \langle \overrightarrow{x_i = t_i} \rangle \,\mathsf{in}\, t)$$

We write $(t_1\, t_2)$ for $(t_1 \mathbin{@} t_2)$, and omit redundant brackets.

A thing to observe is that function symbols come with an arity, which is relevant when defining rewrite rules (Def. 1.5), and comes into play when translating a 'program' into a graph rewriting system; for details of such a translation, see [13] and below (Def. 1.5ii).

Mainly for readability of proofs, the language of terms we study here differs from the one defined in [13], where *expressions* were defined by:

$$\begin{aligned} E ::= {} & x \mid (\mathsf{F}\,(E_1, \ldots, E_n)) \mid (\mathsf{let}\, x = E_1 \,\mathsf{in}\, E_2) \mid \\ & (\mathsf{letrec}\, \overrightarrow{x = E_1} \,\mathsf{in}\, E_2) \mid (\mathsf{case}\, E \,\mathsf{of}\, \overrightarrow{P} | \overrightarrow{E}) \\ P ::= {} & \mathsf{C}\,(x_1, \ldots, x_n) \end{aligned}$$

Notice that, in Def. 1.1, we do not distinguish between function and constructor symbols, so we do not require a separate treatment of patterns; also, we deal with an *applicative* language. This distinction is cosmetic in that all results obtained here could be reached in a first-order system as that of [13]; it is the presentation of the results that benefits from an applicative syntax by giving less involved and shorter proofs. Using the keywords 'share' and 'cycle' rather than 'let' and 'letrec' serves to highlight the change in syntax and system.

Notice that the language of types (presented below) differs significantly from that considered in [13], in that, as far as assignable types are concerned, the systems are incompatible.

We will now formally introduce term graphs, as done in [10]. Following [13], graphs are written in an *equational style* [10,1], rather than using drawings or 4-tuples (as in [10]).

Definition 1.2 [13,20] A *graph* (over $\mathcal{F}$) is a pair $g = \langle r \mid G\rangle$, where r is a variable and stands for the *root* of the graph, and G is a set of equations of the shape '$x = @(y,z)$' or '$x = \mathsf{F}$', that describe the edges in the graph, where the variables that appear on the left appear there in only *one* equation and should all appear on the right as well.

The *variable set* of graph $g = \langle r \mid G\rangle$, $Var(g)$, is the collection of all variable names appearing in r, G. The set of *free* variables of g, $fv(g)$, contains those variables that do not appear as the left-hand side of an equation in G, and a variable in $Var(g)$ is *bound* if it is not free; we will identify graphs that differ only in the names of their bound variables.

Definition 1.3 (cf. [13]) For each term t, the *graph interpretation of* t, $[\![t]\!]$, is defined by ($[\overrightarrow{x_i \mapsto r_i}]$ stands for the simultaneous replacement of $\overrightarrow{r_i}$ for (the free occurrences of) $\overrightarrow{x_i}$, and different graphs are assumed to share no variable names).

$$\begin{aligned}
[\![x]\!] &= \langle x \mid \emptyset\rangle\\
[\![\mathsf{F}]\!] &= \langle f \mid \{f = \mathsf{F}\}\rangle\\
[\![t_1\, t_2]\!] &= \langle r \mid \{r = @(r_1,r_2)\} \cup G_1 \cup G_2\rangle,\\
&\quad \text{where } [\![t_i]\!] = \langle r_i \mid G_i\rangle, i = 1,2, \text{ and } r \text{ is fresh}\\
[\![\mathsf{share}\, t_1 \,\mathsf{via}\, x \,\mathsf{in}\, t_2]\!] &= \langle r_2 \mid G_1 \cup G_2\rangle\,[x \mapsto r_1],\\
&\quad \text{where } [\![t_i]\!] = \langle r_i \mid G_i\rangle, i = 1,2\\
[\![\mathsf{cycle}\,\langle\, \overrightarrow{x_i = t_i}\,\rangle\,\mathsf{in}\, t']\!] &= \langle r' \mid G_1 \cup \cdots \cup G_n \cup G'\rangle\,[\overrightarrow{x_i \mapsto r_i}],\\
&\quad \text{where } [\![t_i]\!] = \langle r_i \mid G_i\rangle,\ (1 \le i \le n)\\
&\qquad\quad [\![t']\!] = \langle r' \mid G'\rangle,
\end{aligned}$$

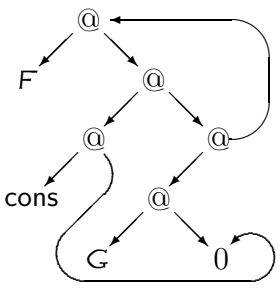
Via this interpretation, the notion of free and bound variables of a graph g induces a notion of free and bound variables on terms; as a result, in the term $(\mathsf{share}\, t_1 \,\mathsf{via}\, x \,\mathsf{in}\, t_2)$, x does not occur free in t_1.

Example 1.4 (cf. [13]) The term

$$(\mathsf{share}\, 0 \,\mathsf{via}\, x \,\mathsf{in}\, (\mathsf{cycle}\,\langle\, z = \mathsf{F}\,(\mathsf{cons}\, x\, (\mathsf{G}\, x\, z))\,\rangle\,\mathsf{in}\, z))$$

translates to the graph

$$\begin{aligned}
\langle z \mid \{&z = @(f,a),\\
&f = \mathsf{F},\\
&a = @(b,c),\\
&b = @(d,x),\\
&c = @(e,z),\\
&d = \mathsf{cons},\\
&e = @(g,x),\\
&g = \mathsf{G},\\
&x = 0\}\rangle
\end{aligned}$$



Reduction on $T(\mathcal{F},\mathcal{X})$ is defined through rewrite rules.

Definition 1.5 (i) A *rewrite rule* is a pair $(\mathit{left}, \mathit{right})$ of terms such that
- $\mathit{left} = \mathsf{F}\, t_1 \cdots t_n$, for some F with $n = \mathit{arity}(\mathsf{F})$, and terms $t_1, \ldots, t_n$, and
- $\mathit{fv}(\mathit{right}) \subseteq \mathit{fv}(\mathit{left})$.

(ii) The translation into graphs of Def. 1.3 is extended to rewrite rules through: Let $\mathit{left} \to \mathit{right}$ be a (recursive) rewrite rule with defined symbol F, then:

$$\begin{aligned} [\![\mathit{left} \to \mathit{right}]\!] &= \langle r_l \mid G_{\mathit{left}} \cup G_{\mathit{right}} \rangle\, [\overrightarrow{x_i \mapsto y_i}], \\ \text{where } [\![\mathsf{F}]\!] &= \langle g \mid \{ g = \mathsf{F} \} \rangle \\ [\![\mathit{left}]\!] &= \langle r_l \mid G_{\mathit{left}} \rangle \\ [\![\mathit{right}]\!] &= \langle r_r \mid G_{\mathit{right}} \rangle \\ \{x_1, \ldots, x_n\} &= \mathit{fv}(\mathit{left}) \end{aligned}$$

and all $y_1, \ldots, y_n$ and g are unused variables.

We take the view that in a rewrite rule a certain symbol is defined. We call a defined symbol F *recursive* if F occurs on a cycle in the dependency-graph, and call every rewrite rule that defines F *recursive*. All function symbols that occur on one cycle in the dependency-graph depend on each other and are, therefore, defined *simultaneously* and are called *mutually recursive*. Since it is always possible to introduce tuples into the language and solve the problem of mutual recursion using only recursive rules, we will assume that rules are *not* mutually recursive.

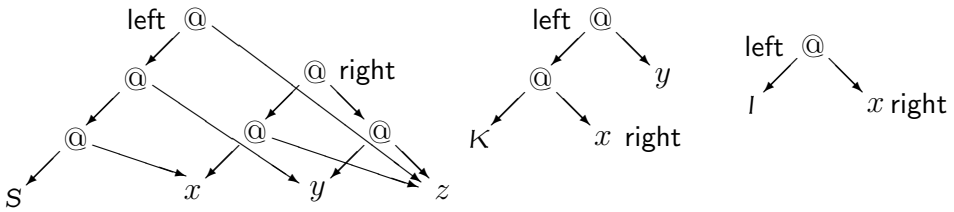
Definition 1.6 We define a rewrite relation on terms by: $t_1 \to t_2$ if and only if there are graphs g_1 and g_2 such that $[\![t_1]\!] = g_1$, $[\![t_2]\!] = g_2$, and $g_1 \to g_2$.

Definition 1.7 An *Applicative Term Graph Rewriting System* (@TGRS) is a pair $(\Sigma, \mathbf{R})$ of an alphabet Σ and a set $\mathbf{R}$ of rewrite rules.

Example 1.8 The rewrite rules that define Combinatory Logic are expressed as a @TGRS by (notice that the rule for S expresses that the variable z is shared):

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

Translated to term graph rewrite rules, these rules look like (using left and right rather than r_l and r_r):



Notice that, if we would have used '$\mathsf{S}\, x\, y\, z \to \mathsf{share}\, z\, \mathsf{via}\, v\, \mathsf{in}\, (x\, v)\, (y\, v)$' instead of the first rule, so would have expressed explicitly that we want the third

parameter to be shared, the resulting graph rewrite rule would have been exactly the same.

The principle of term graph rewriting, presented formally in [10], can be summarised as follows:

- a graphs g contains a *redex* if a left-hand side *left* of a rewrite rule $left \to right$ can be mapped onto a graph, i.e. if there exists a homomorphism from *left* to the graph, which respects the structure of graphs and maps free variables to graphs.
- Reduction (rewriting) of the redex then consists of adding an *instance* of *right* to the graph by adding the right hand side (graph) of the rewrite rule, but by replacing an edge going into a free variable to one going into the image of the variable under the aforementioned homomorphism.
- All edges going into the image of the root of *left* are re-directed into the root of the added instance of *right*.
- Now part of the graph has become *garbage*, in that it is no longer accessible from the root of g; this can be removed.

Example 1.9 As an example of term graph rewriting within the context of this paper, consider Fig. 1.

Since (free) variables in @TGRS may be substituted by function symbols, we obtain the usual functional programming paradigm, extended with definitions of operators and data structures. Notice, however, that we obtain more: in functional programs, the set $\mathcal{F}$ (Def. 1.1) is divided into *function symbols* and (data-type) *constructors*, and, in rewrite rules, function symbols are not allowed to appear in 'constructor position' and vice-versa. This does not hold for @TGRS.

2 Rank 2 types

In Section 4, we will present a decidable notion of type assignment on @TGRS, using polymorphic intersection types of rank 2. The system presented here is a *corrected* version of a similar system presented in [5], and is an extension, by the '$\forall$' type constructor, of the Rank 2 system with intersection types as defined in [4].

We use strict intersection types over a set $V = \Phi \uplus \mathcal{A}$ of *free and bound type-variables* respectively, and a set S of *sorts* or *type constants*. For various reasons (definition of operations on types, definition of unification), we will distinguish syntactically between (names of) *free* type-variables (which belong to Φ) and (names of) *bound* type-variables (in $\mathcal{A}$).

Definition 2.1 [5] We define polymorphic intersection types of Rank 2 in layers: $\mathcal{T}_C$ are Curry types, built out of type variables in Φ (ranged over by φ), sorts (type constants, ranged over by s) and '$\to$', $\mathcal{T}_C^{\forall}$ are quantified Curry

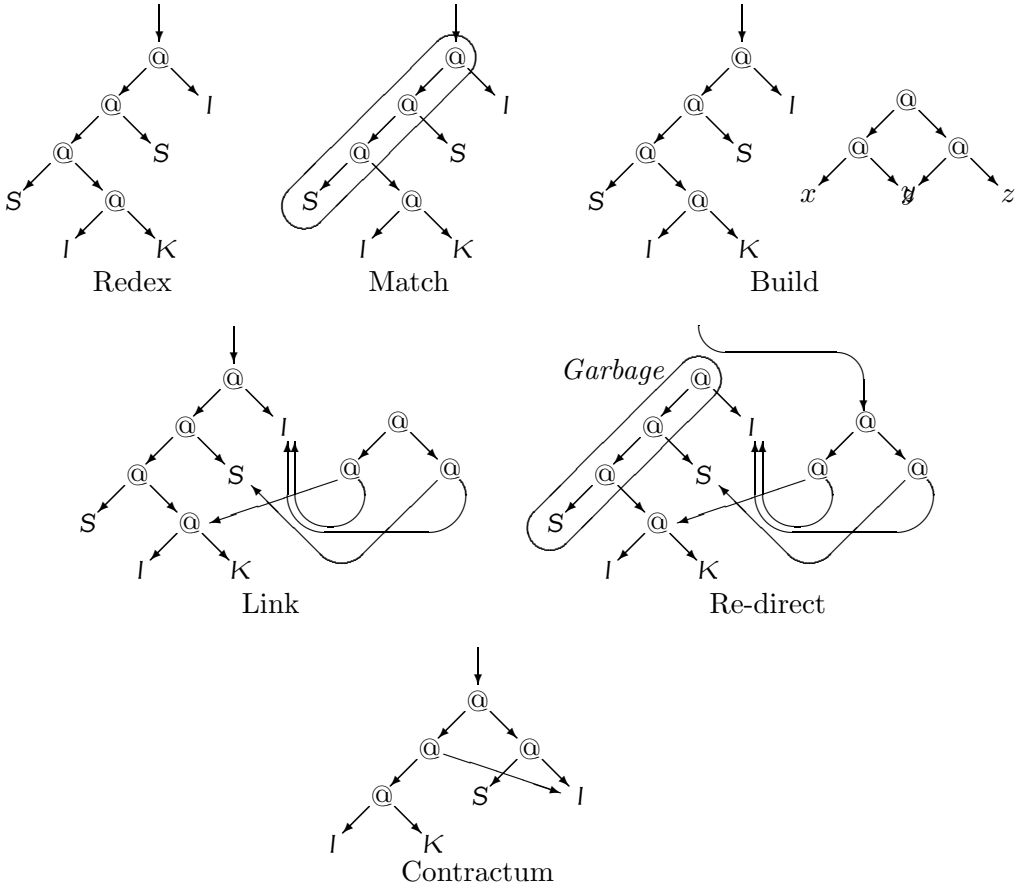


Fig. 1. An example of term graph rewriting

types, $\mathcal{T}_1$, the types of rank 1, are intersections of quantified Curry types, and $\mathcal{T}_2$ are types of Rank 2:

$$\mathcal{T}_{\rm C} ::= \varphi \mid s \mid (\mathcal{T}_{\rm C} \to \mathcal{T}_{\rm C}) \quad \mathcal{T}_{\rm C}^{\forall} ::= \mathcal{T}_{\rm C} \mid (\forall\alpha.\mathcal{T}_{\rm C}^{\forall}[\alpha/\varphi])$$
$$\mathcal{T}_1 ::= (\mathcal{T}_{\rm C}^{\forall} \cap \cdots \cap \mathcal{T}_{\rm C}^{\forall}) \quad \mathcal{T}_2 ::= \varphi \mid s \mid (\mathcal{T}_1 \to \mathcal{T}_2)$$

We use $\mathcal{T}_{\rm R}$ for the union of these sets, and use σ, τ for arbitrary elements of $\mathcal{T}_{\rm R}$. Notice that $\mathcal{T}_{\rm C} \subseteq \mathcal{T}_{\rm C}^{\forall} \subseteq \mathcal{T}_1$ and $\mathcal{T}_{\rm C} \subseteq \mathcal{T}_2$, but that $\mathcal{T}_{\rm C}^{\forall} \not\subseteq \mathcal{T}_2$.

In the notation of types, '$\to$' is assumed to associate to the right, '$\cap$' binds stronger than '$\to$', which binds stronger than '$\forall$'; so $\rho\cap\mu\to(\forall\alpha.\gamma\to\delta)\to\sigma$ stands for $((\rho\cap\mu)\to((\forall\alpha.(\gamma\to\delta))\to\sigma))$. Also, $\forall\vec{\alpha}\,.\sigma$ is used 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.

Definition 2.2 *fv*(σ), the set of *free variables* of a type σ is defined as usual (note that by construction, *fv*(σ) $\subseteq \Phi$). A type is called *closed* if it contains no free variables, and *ground* if it contains no variables at all.

Notice that, because of the distinction between free and bound type vari-

ables, not every syntactic sub-type of $\sigma \in \mathcal{T}_{\rm R}$ is necessarily a type in $\mathcal{T}_{\rm R}$, but ignoring this below will not affect any result.

Definition 2.3 [5] On $\mathcal{T}_{\rm R}$, the pre-order (i.e. reflexive and transitive relation) '$\leq$' is defined by:

$$\begin{array}{rcll} \sigma_1 \cap \cdots \cap \sigma_n & \leq & \sigma_i, & (1 \leq i \leq n) \\ \forall \alpha.(\sigma[\alpha/\varphi]) & \leq & \sigma[\tau/\varphi], & (\tau \in \mathcal{T}_{\rm C}) \\ \forall 1 \leq i \leq n\ [\sigma \leq \sigma_i] & \Rightarrow & \sigma \leq \sigma_1 \cap \cdots \cap \sigma_n & (n \geq 1) \\ \rho \leq \sigma, \tau \leq \mu & \Rightarrow & \sigma \to \tau \leq \rho \to \mu, & (\tau, \mu \in \mathcal{T}_2) \\ \sigma \leq \tau & \Rightarrow & \forall \alpha.\sigma[\alpha/\varphi] \leq \forall \alpha.\tau[\alpha/\varphi]. & \end{array}$$

Definition 2.4 (i) A *statement* is a term of the form $t{:}\sigma$, with $\sigma \in \mathcal{T}_{\rm R}$ and $t \in T(\mathcal{F}, \mathcal{X})$. t is the *subject* and σ the *predicate* of $t{:}\sigma$.

(ii) A *basis* B is a partial mapping from $\mathcal{X}$ to $\mathcal{T}_1$, represented as set of statements with only distinct variables as subjects. By abuse of notation, we write $x \in B$ if there exists a τ such that $x{:}\tau \in B$, $\varphi \in B$ if there is a type in B in which φ occurs, and write $B \backslash x$ for the basis obtained from B by removing the statement that has x as subject.

(iii) For bases B_1, B_2, the basis $B_1 \cap B_2$ is defined by:

$$\begin{array}{rl} B_1 \cap B_2 = & \{x{:}\tau \mid x{:}\tau \in B_1\ \&\ x \notin B_2\} \cup \{x{:}\tau \mid x{:}\tau \in B_2\ \&\ x \notin B_1\} \cup \\ & \{x{:}\tau_1 \cap \tau_2 \mid x{:}\tau_1 \in B_1\ \&\ x{:}\tau_2 \in B_2\} \\ B, x{:}\tau = & B \setminus x \cup \{x{:}\tau\} \end{array}$$

(iv) The relation '$\leq$' is extended to bases by:

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

Notice that if $n = 0$, then $B_1 \cap \ldots \cap B_n = \emptyset$.

3 Operations on types

The Rank 2 versions for the various operations as presented below are defined in much the same way as in [4], with the exception of the operation of closure and lifting, that were not used there, and are taken from [5].

Substitution

We will define substitution as usual in first-order logic, but avoid to go out of the set of polymorphic intersection types of Rank 2. For example, the substitution of φ by $\tau_1 \cap \tau_2$ would transform $\sigma \to \varphi$ into $\sigma \to \tau_1 \cap \tau_2$, which is not in $\mathcal{T}_{\rm R}$. However, since $\mathcal{T}_{\rm C} \subseteq \mathcal{T}_2$, and $\mathcal{T}_{\rm C}$ is closed for (Curry-)substitution, also $\mathcal{T}_2$ is closed for that kind of substitution.

The following definition takes this fact into account.

Definition 3.1 (i) The *substitution* $(\varphi \mapsto \rho)\ : \mathcal{T}_2 \to \mathcal{T}_2$, where φ is a type-

variable in Φ and $\rho \in \mathcal{T}_C$, is defined by:

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

(ii) We use Id_S for the substitution that replaces all type-variables by themselves, write $\mathcal{S}$ for the set of all substitutions, and use S to denote a generic substitution. Substitutions extend to bases in the natural way: $S(B) = \{x{:}S(\rho) \mid x{:}\rho \in B\}$, and the set of substitutions is closed under composition '$\circ$'.

Lifting

The operation of *lifting* replaces basis and type by a smaller basis and a larger type, in the sense of '$\leq$'. This operation allows us to eliminate intersections and universal quantifiers, using the '$\leq$' relation.

Definition 3.2 An operation of *lifting* is $L = <\langle B_1, \tau_1\rangle, \langle B_2, \tau_2\rangle>$ such that $\tau_1 \leq \tau_2$ and $B_2 \leq B_1$, and is defined by $L(\langle B, \sigma\rangle) = \langle B', \sigma'\rangle$ where

$$\begin{array}{ll}
\sigma' = \tau_2, \text{ if } \sigma = \tau_1, & B' = B_2, \text{ if } B = B_1 \\
\sigma' = \sigma, \text{ otherwise} & B' = B, \text{ otherwise}
\end{array}$$

A lifting on types is determined by a pair $L = \langle \tau_1, \tau_2\rangle$ such that $\tau_1 \leq \tau_2$ and is defined by

$$\begin{array}{rl}
L(\sigma) = \tau_2, & \text{if } \sigma = \tau_1 \\
\sigma, & \text{otherwise}
\end{array}$$

Closure

The operation of *closure* introduces quantifiers, taking into account the basis where a type might occur.

Definition 3.3 A *closure* is characterised by a pair $\langle \sigma, \varphi\rangle$ with $\sigma \in \mathcal{T}_C^{\forall}$, and is defined by:

$$\langle \sigma, \varphi\rangle\,(\langle B, \tau_1 \cap \cdots \cap \tau_n\rangle) = \langle B, \tau_1' \cap \cdots \cap \tau_n'\rangle$$

where, for all $1 \leq i \leq n$,

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

Closure is extended to types by: $\langle \varphi\rangle\,(\sigma) = (\tau)$, if $\langle \varphi, \sigma\rangle\,(\langle \emptyset, \sigma\rangle) = \langle \emptyset, \tau\rangle$.

Expansion
The variant of expansion used in the Rank 2 system is quite different from that normally used [2,3,36]. 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 here expansion is only used in very precise situations (within the procedure $unify_2^{\forall}$, and in the proof of Thm. 6.5), the solution is relatively easy: in the context of Rank 2 types, expansion is only called on types in $\mathcal{T}_{\mathrm{C}}^{\forall}$, so it is defined to work well there, by replacing *all* types by an intersection; in particular, intersections are not created at the right of an arrow.

Definition 3.4 Let B be a basis, $\sigma \in \mathcal{T}_{\mathrm{R}}$, and $n \geq 1$. The n-fold *expansion* with respect to the pair $\langle B, \sigma\rangle$, $n_{\langle B,\sigma\rangle} : \mathcal{T}_2 \to \mathcal{T}_2$ is constructed as follows: Suppose $F = \{\varphi_1, \ldots, \varphi_m\}$ 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 F. Let S_i be the substitution that replaces every φ_j by φ_j^i. Then expansion is defined on types, bases, and pairs, respectively, by:

$$\begin{aligned} n_{\langle B,\sigma\rangle}(\tau) &= S_1(\tau) \cap \cdots \cap S_n(\tau), \\ n_{\langle B,\sigma\rangle}(B') &= \{x{:}n_{\langle B,\sigma\rangle}(\rho) \mid x{:}\rho \in B\}, \\ n_{\langle B,\sigma\rangle}(\langle B', \sigma'\rangle) &= \langle n_{\langle B,\sigma\rangle}(B'), n_{\langle B,\sigma\rangle}(\sigma')\rangle. \end{aligned}$$

Notice that, if $\tau \in \mathcal{T}_2$, it can be that $S_1(\tau) \cap \cdots \cap S_n(\tau)$ is not a legal type. However, for the sake of clarity, and since each $S_i(\tau) \in \mathcal{T}_2$, we will not treat this case separately.

Operations will be grouped in chains.

Definition 3.5 (i) A *chain* is an object $[O_1, \ldots, O_n]$, where each O_i is an operation of substitution, expansion, lifting, or closure, and $[O_1, \ldots, O_n](\sigma) = O_n(\cdots(O_1(\sigma))\cdots)$.

(ii) On chains the operation of concatenation is denoted by $*$, and:

$$[O_1, \ldots, O_i] * [O_{i+1}, \ldots, O_n] = [O_1, \ldots, O_n].$$

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

4 Rank 2 Type Assignment

We now come to the definition of Rank 2 type assignment.

Definition 4.1 (i) A *Rank 2 environment* $\mathcal{E}$ is a mapping from $\mathcal{F}$ to $\mathcal{T}_2$.

(ii) *Rank 2 type assignment on terms* is defined by the following natural

deduction system:

$$(Ax): \frac{}{B \vdash_{\mathcal{E}} x{:}\tau} \; (x{:}\sigma \in B \;\&\; \sigma \leq \tau \;\&\; \sigma \in \mathcal{T}_1 \;\&\; \tau \in \mathcal{T}_2)$$

$$(\cap I): \frac{B \vdash_{\mathcal{E}} t{:}\sigma_1 \quad \cdots \quad B \vdash_{\mathcal{E}} t{:}\sigma_n}{B \vdash_{\mathcal{E}} t{:}\sigma_1 \cap \cdots \cap \sigma_n} \; (n \geq 1 \;\&\; \forall 1 \leq i \leq n\,[\sigma_i \in \mathcal{T}_{\mathrm{C}}^{\forall}])$$

$$(\rightarrow E): \frac{B \vdash_{\mathcal{E}} t_1{:}\sigma \rightarrow \tau \quad B \vdash_{\mathcal{E}} t_2{:}\sigma}{B \vdash_{\mathcal{E}} t_1\, t_2{:}\tau}$$

$$(\forall I): \frac{B \vdash_{\mathcal{E}} t{:}\sigma}{B \vdash_{\mathcal{E}} t{:}\forall\alpha.\sigma[\alpha/\varphi]} \; (\varphi \notin B \;\&\; \sigma \in \mathcal{T}_{\mathrm{C}}^{\forall})$$

$$(\mathsf{share}): \frac{B, x{:}\sigma \vdash_{\mathcal{E}} t_2{:}\tau \quad B \vdash_{\mathcal{E}} t_1{:}\sigma}{B \vdash_{\mathcal{E}} (\mathsf{share}\, t_1\, \mathsf{via}\, x\, \mathsf{in}\, t_2){:}\tau}$$

$$(\mathcal{F}): \frac{}{B \vdash_{\mathcal{E}} \mathsf{F}{:}\sigma} \; (\exists\, Ch\,[Ch(\mathcal{E}(\mathsf{F})) = \sigma])$$

$$(\mathsf{cycle}): \frac{B, \overrightarrow{x_i{:}\sigma_i} \vdash_{\mathcal{E}} t_i{:}\sigma_i \quad B, \overrightarrow{x_i{:}\sigma_i} \vdash_{\mathcal{E}} t{:}\tau}{B \vdash_{\mathcal{E}} \mathsf{cycle}\,\langle\, \overrightarrow{x_i = t_i} \,\rangle\, \mathsf{in}\, t{:}\tau} \; (\forall 1 \leq i \leq n\,[\sigma_i \in \mathcal{T}_{\mathrm{C}}])$$

We write $B \vdash_{\mathcal{E}} t{:}\sigma$ if this is derivable using the rules above.

Notice the use of an environment and chain in rule $(\mathcal{F})$; because of this rule, the notion of type assignment defined here is in fact a *partially typed* system: all function symbols are assumed to have a type to begin with, that is 'instantiated' by this rule.

Also, rule $(\mathcal{F})$ formalises the practice of functional languages in that it introduces a notion of polymorphism for 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.

Although these rules express how to type terms, it is straightforward to extend this definition to one that expresses how to type graphs, such that $B \vdash_{\mathcal{E}} t{:}\sigma$ if and only if $B \vdash_{\mathcal{E}} [\![t]\!]{:}\sigma$.

Example 4.2 If we extend the definition of types with the alternative for list types and booleans

$$\mathcal{T}_{\mathrm{C}} ::= \varphi \mid s \mid (\mathcal{T}_{\mathrm{C}} \rightarrow \mathcal{T}_{\mathrm{C}}) \mid [\mathcal{T}_{\mathrm{C}}] \mid \mathsf{Bool}$$

then, using Rank 2 types, we can now express the function 'IsNil', that tests if a list is empty, defined by

$$\mathsf{IsNil}\,[\,] \rightarrow \mathrm{TT}$$

is typeable using the environment

$$\begin{array}{ll} \mathcal{E}(\text{TT}) & = \mathsf{Bool} \\ \mathcal{E}([\,]) & = [\varphi] \\ \mathcal{E}(\mathsf{Cons}) & = \varphi{\rightarrow}[\varphi]{\rightarrow}[\varphi] \\ \mathcal{E}(\mathsf{IsNil}) & = (\forall\alpha.[\alpha]){\rightarrow}\mathsf{Bool} \end{array} \qquad \dfrac{\dfrac{}{\emptyset \vdash_{\mathcal{E}} \mathsf{IsNil} : (\forall\alpha.[\alpha]){\rightarrow}\mathsf{Bool}} \quad \dfrac{\dfrac{}{\emptyset \vdash_{\mathcal{E}} [\,] : [\varphi]}}{\emptyset \vdash_{\mathcal{E}} [\,] : \forall\alpha.[\alpha]}}{\emptyset \vdash_{\mathcal{E}} \mathsf{IsNil}\,[\,] : \mathsf{Bool}}$$

Notice that the type for this function 'IsNil' in the environment prohibits its use against 'concrete' lists that are not empty, since any list with an element is that is of type s is no longer polymorphic. Also, this is not a derivable result in any of the other systems mentioned in the introduction.

Notice that rule $(\mathcal{F})$ models a kind of polymorphism into our system, other than the kind obtained by having quantified types to our disposition. Quantification allows only the replacement of type-variables by Curry types, whereas rule $(\mathcal{F})$ allows any operation to be applied. It allows function symbols to appear in context that require a type that is more specific than the one provided by the environment; the soundness result we show below for the various operations justify the application of chains to the types provided by the environment.

Also, since quantification elimination is implicit in rule (Ax), 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 with a possible rule $(\cap E)$, its use is in a strict system limited to variables, and there its actions are already performed by (Ax).

For this system to be of use in practice, a minimal requirement would be a *subject reduction* result, which expresses that types are preserved by reduction. To achieve this, we define a notion of type assignment on rewrite rules using the notion of principal pair (also called principal typing), that will be developed in Section 6 (see Def. 6.1), and culminates in Thm. 6.5, which states:

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

This property, together with the result that all operations are sound, is used to prove the subject reduction result. (The same method was used in [7,6,5].)

Definition 4.3 (i) We say that *left* $\rightarrow$ *right* $\in \mathbf{R}$ with defined symbol F *is typeable with respect to* $\mathcal{E}$, if there are P, and $\pi \in \mathcal{T}_2$ such that:
(a) $\langle P, \pi\rangle$ is a principal pair (Def. 6.1) for *left* with respect to $\mathcal{E}$.
(b) In $P \vdash_{\mathcal{E}} \mathit{left}{:}\pi$ and $P \vdash_{\mathcal{E}} \mathit{right}{:}\pi$ each occurrenc of F is typed with $\mathcal{E}(\mathsf{F})$.

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

As an aside to part b, remark that, by rule $(\mathcal{E})$, we know that each occurrence of F has a type *generated from* $\mathcal{E}(\mathsf{F})$ by applying a chain of operations. Part b states that, for the derivations involved here, these chains are all empty, i.e. are the identity operation. Since we forced the type of a function symbol

F to be exactly $\mathcal{E}(\mathsf{F})$ in the rules that define F, the typeability of rules ensures consistency with respect to the environment.

Notice that, because in the translation of terms to graphs, the defined node is shared by all occurrences in the rule, when typing the graph rewrite rule the condition 'all occurrences of F are typed with $\mathcal{E}(\mathsf{F})$' becomes 'the occurrence of F is typed with $\mathcal{E}(\mathsf{F})$'.

Before we come to a subject reduction result, first we need to show that all operations defined are sound, which we will show in the next section. The main result there is Lem. 4.7, which states:

If $\sigma \in \mathcal{T}_1$, $B \vdash_{\mathcal{E}} t{:}\sigma$, and Ch is a chain of operations on types such that $Ch(\langle B, \sigma\rangle) = \langle B', \sigma'\rangle$, then $B' \vdash_{\mathcal{E}} t{:}\sigma'$.

We will now take a short-cut, and show that reductions preserve types in our system, using the notion of principal pair and the soundness of operations on types.

The proof of Subject Reduction depends also on the following lemma:

Lemma 4.4 (Replacement) *Let $\mathcal{E}$ be an environment, t a term, and f a mapping from free variables to terms (which extends naturaly to a mapping from terms to terms).*

(i) *If $B \vdash_{\mathcal{E}} t{:}\sigma$ and B' is such that $B' \vdash_{\mathcal{E}} f(x){:}\rho$ for every statement $x{:}\rho \in B$, then $B' \vdash_{\mathcal{E}} f(t){:}\sigma$.*

(ii) *If there are B and σ such that $B \vdash_{\mathcal{E}} f(t){:}\sigma$, then for every x occurring in t there is a type ρ_x such that $\{x{:}\rho_x \mid x \in fv(t)\} \vdash_{\mathcal{E}} t{:}\sigma$, and $B \vdash_{\mathcal{E}} f(x){:}\rho_x$.*

Using this lemma, the following result follows easily.

Theorem 4.5 (Subject reduction) *If $B \vdash_{\mathcal{E}} t{:}\sigma$ and $t \to t'$, then $B \vdash_{\mathcal{E}} t'{:}\sigma$.*

Example 4.6 Let $\sigma, \tau, \rho, \mu, \nu, \gamma$, and δ be (arbitrary) types. Take the rewrite rules that define Combinatory Logic of Ex. 1.8, and the environment $\mathcal{E}$:

$$\begin{array}{lll} \mathcal{E}(\mathsf{S}) & = & (\sigma{\to}\tau{\to}\rho){\to}(\mu{\to}\tau){\to}\sigma{\cap}\mu{\to}\rho \\ \mathcal{E}(\mathsf{K}) & = & \nu{\to}\gamma{\to}\nu \\ \mathcal{E}(\mathsf{I}) & = & \delta{\to}\delta \end{array}$$

Then these rules are typeable with respect to $\mathcal{E}$; we show the derivation for the right-hand side of the first rule in Fig. 2.

It is possible to show that the operations defined in Section 3 are sound; this result is omitted for lack of space.

These soundness results are combined in the following:

Lemma 4.7 (Soundness of chains) *If $\sigma \in \mathcal{T}_1$, $B \vdash_{\mathcal{E}} t{:}\sigma$, and Ch is such that*
$Ch(\langle B, \sigma\rangle) = \langle B', \sigma'\rangle$, then $B' \vdash_{\mathcal{E}} t{:}\sigma'$.

$$\dfrac{\dfrac{\dfrac{\overline{B_1' \vdash_{\mathcal{E}} x : \sigma\to\tau\to\rho} \quad \overline{B_1' \vdash_{\mathcal{E}} v : \sigma}}{B_1' \vdash_{\mathcal{E}} x\, v : \tau\to\rho} \quad \dfrac{\overline{B_1' \vdash_{\mathcal{E}} y : \mu\to\tau} \quad \overline{B_1' \vdash_{\mathcal{E}} v : \mu}}{B_1' \vdash_{\mathcal{E}} y\, v : \tau}}{B_1' \vdash_{\mathcal{E}} (x\, v)\,(y\, v) : \rho} \quad \dfrac{\overline{B_1 \vdash_{\mathcal{E}} z : \sigma} \quad \overline{B_1 \vdash_{\mathcal{E}} z : \mu}}{B_1 \vdash_{\mathcal{E}} z : \sigma\cap\mu}}{B_1 \vdash_{\mathcal{E}} \mathsf{share}\, z\, \mathsf{via}\, v\, \mathsf{in}\, (x\, v)\,(y\, v) : \rho}$$

Fig. 2. A type derivation for Ex. 4.6 (where $B_1 = \{x{:}\sigma\to\tau\to\rho, y{:}\mu\to\tau, z{:}\sigma\cap\mu\}$, and $B_1' = B_1, v{:}\sigma\cap\mu$.

5 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 $\sigma\to\tau$, and t_2 has type ρ, then unification looks for a common instance of the types σ and ρ such that $(t_1\, t_2)$ can be typed properly. The unification algorithm $unify_2^{\forall}$ presented in the next definition (a corrected version of the algorithm presented in [5]) 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* [35], and can be seen as an extension of the notion of unification as presented in [4], in that it deals with quantification as well.

Definition 5.1 [Unification] Unification of Curry types (extended with bound variables and type constants) is defined by:

$$\begin{array}{lll}
unify : \mathcal{T}'_{\mathrm{C}} \times \mathcal{T}'_{\mathrm{C}} \to \mathcal{S} & & \\
unify(\varphi, \varphi') & = (\varphi \mapsto \varphi'), & \\
unify(\varphi, \tau) & = (\varphi \mapsto \tau), \text{ if } \varphi \text{ not in } \tau, & \\
unify(\alpha, \alpha) & = Id_S, & \\
unify(s, s) & = Id_S, & \\
unify(\sigma, \varphi) & = unify(\varphi, \sigma), & \\
unify(\sigma\to\tau, \rho\to\mu) & = S_2 \circ S_1, & \\
 & \quad \text{where } S_1 = unify(\sigma, \rho), & \\
 & \qquad\quad S_2 = unify(S_1(\tau), S_1(\mu)). &
\end{array}$$

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

It is worthwhile to notice that the operation on types returned by *unify* is not really a substitution, since it allows, e.g., $(\varphi \mapsto \alpha)$, without keeping track of the binder for α. This potentially will create wrong results, since unification can now substitute bound variables in unbound places. Therefore, special care has to be taken before applying a substitution, to guarantee its application to the argument acts as a 'real' substitution.

The following property is well-known, and formulates that *unify* returns

the most general unifier for two Curry types, if it exists.

Proposition 5.2 ([35]) *If two types have an instance in common, they have a* highest common instance *which is returned by unify: for all $\sigma, \tau \in \mathcal{T}_C$, substitutions S_1, S_2: if $S_1(\sigma) = S_2(\tau)$, then there are substitutions S_u and S' such that*

$$S_u = \mathit{unify}(\sigma, \tau), \text{ and } S_1(\sigma) = S' \circ S_u(\sigma) = S' \circ S_u(\tau) = S_2(\tau).$$

The unification algorithm $\mathit{unify}_2^\forall$ as defined below gets, typically, called during the computation of the principal pair for an application $t_1\, t_2$. Suppose the algorithm has derived $P_1 \vdash_\mathcal{E} t_1 : \pi_1$ and $P_2 \vdash_\mathcal{E} t_2 : \pi_2$ as principal pairs for t_1 and t_2, respectively, and that $\pi_1 = \sigma \to \tau$. Thus the demanded type σ is in $\mathcal{T}_1$ and the provided type π_2 is in $\mathcal{T}_2$. In order to be consistent, the result of the unification of σ and π_2 – a chain Ch – should always be such that $Ch(\pi_2) \in \mathcal{T}_1$. However, if $\pi_2 \notin \mathcal{T}_C$, then in general $Ch(\pi_2) \notin \mathcal{T}_1$. To overcome this difficulty, an algorithm $to\mathcal{T}_C$ 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, $to\mathcal{T}_C(\rho)$ should fail, since quantifiers that appear before an arrow cannot be removed by any of the operations on types defined above. Finally,

$$\mathit{unify}_2^\forall(\sigma, S_2(\pi_2), S_2(P_2))$$

is called (with $S_2 = to\mathcal{T}_C(\pi_2)$). The basis $S_2(P_2)$ is needed to calculate the expansion of $S_2(\pi_2)$ in case σ is an intersection type.

Definition 5.3 The function $to\mathcal{T}_C : \mathcal{T}_2 \to \mathcal{S}$ is defined by:

$$\begin{aligned} to\mathcal{T}_C(\sigma) &= [Id_S], \quad \text{if } \sigma \in \mathcal{T}_C \\ to\mathcal{T}_C((\sigma_1 \cap \cdots \cap \sigma_n) \to \mu) &= S' \circ S_n, \text{ otherwise,} \end{aligned}$$

where $S_i = \mathit{unify}(S_{i-1}(\sigma_1), S_{i-1}(\sigma_{i+1})) \circ S_{i-1}, (1 \le i \le n-1, \text{ with } S_0 = Id_S)$

$S' = to\mathcal{T}_C(S_n(\mu))$

(Again, notice that $to\mathcal{T}_C(\sigma)$ fails if σ contains '$\forall$'.)

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

The following definition presents the main unification algorithm, $\mathit{unify}_2^\forall$.

Definition 5.4 The function $unify_2^{\forall}$ is defined by:

$$\begin{aligned} unify_2^{\forall}(\varphi, \tau, B) &= [(\varphi \mapsto \tau)], \\ unify_2^{\forall}((\forall\overrightarrow{\alpha_1}.\sigma_1) \cap \ldots \cap (\forall\overrightarrow{\alpha_n}.\sigma_n), \tau, B) &= [Ex, S_n], \quad \text{otherwise} \end{aligned}$$

where
$$\begin{aligned} Ex &= n_{\langle B,\tau\rangle}, \\ \tau_1 \cap \cdots \cap \tau_n &= Ex(\tau), \text{ and} \\ \text{for every } 1 \leq i \leq n,\ S_i &= unify(S_{i-1}(\sigma_i), \tau_i) \circ S_{i-1} \text{ (with } S_0 = Id_S). \end{aligned}$$

The procedure $unify_2^{\forall}$ fails when *unify* fails, and $to\mathcal{T}_C$ fails when either *unify* fails or when the argument contains '$\forall$'. Because of this relation between $unify_2^{\forall}$ and $to\mathcal{T}_C$ on one side, and *unify* on the other, the procedures defined here are terminating and type assignment in the system defined in this paper is decidable.

6 Principal pairs for terms

In this section, the principal pair for a term t with respect to the environment $\mathcal{E}$ – $pp_{\mathcal{E}}(t)$ – is defined, consisting of basis P and type π. In Thm. 6.5 it will be shown that, for every term, this is indeed the principal one.

Definition 6.1 Let t be a term in $T(\mathcal{F}, \mathcal{X})$. $pp_{\mathcal{E}}(t) = \langle P, \pi\rangle$, with $\pi \in \mathcal{T}_2$, is defined, using $unify_2^{\forall}$, by induction to the structure of terms through:

(x) : Then $pp_{\mathcal{E}}(x) = \langle\{x{:}\varphi\}, \varphi\rangle$.

(F) : $pp_{\mathcal{E}}(\mathsf{F}) = \langle\emptyset, \mathcal{E}(\mathsf{F})\rangle$.

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

$(\pi_1 = \varphi)$: $pp_{\mathcal{E}}(t_1\, t_2) = \langle P, \pi\rangle$, where
$$\begin{aligned} \langle P, \pi\rangle &= \langle S_1(P_1 \cap S_2(P_2)), \varphi'\rangle, \\ S_1 &= (\varphi \mapsto S_2(\pi_2) \to \varphi'), \text{ and} \\ &\varphi' \text{ is a fresh variable.} \end{aligned}$$

$(\pi_1 = \sigma \to \tau)$: $pp_{\mathcal{E}}(t_1\, t_2) = \langle P, \pi\rangle$, provided P and π contain no unbound occurrences of αs, where
$$\begin{aligned} \langle P, \pi\rangle &= \langle S(P_1 \cap Ex(S_2(P_2))), S(\tau)\rangle, \\ [Ex, S] &= unify_2^{\forall}(\sigma, S_2(\pi_2), S_2(P_2)). \end{aligned}$$

$(\mathsf{share}\, t_1\, \mathsf{via}\, x\, \mathsf{in}\, t_2)$: Let $pp_{\mathcal{E}}(t_i) = \langle P_i, \pi_i\rangle$, for $i = 1, 2$. Then either:

- (x occurs in t_1). Then there exists $P', \sigma \in \mathcal{T}_1$ such that $P_1 = P', x{:}\sigma$. Let $S_2 = to\mathcal{T}_C(\pi_2)$. Then
$$pp_{\mathcal{E}}(\mathsf{share}\, t_1\, \mathsf{via}\, x\, \mathsf{in}\, t_2) = \langle P, \pi\rangle,$$
provided P and π contain no unbound occurrences of αs, where
$$\begin{aligned} \langle P, \pi\rangle &= \langle S(P' \cap Ex(S_2(P_2))), S(\pi_1)\rangle \\ [Ex, S] &= unify_2^{\forall}(\sigma, S_2(\pi_2), S_2(P_2)). \end{aligned}$$

- (x does not occur in t_1). Then

$$pp_{\mathcal{E}}(\mathsf{share}\, t_1 \,\mathsf{via}\, x \,\mathsf{in}\, t_2) = \langle P_1, \pi_1 \rangle.$$

($\mathsf{cycle}\, \langle \overrightarrow{x_i = t_i} \rangle \,\mathsf{in}\, t'$) : Let, for $1 \leq i \leq n$, $pp_{\mathcal{E}}(t_i) = \langle P_i, \pi_i \rangle$, and $pp_{\mathcal{E}}(t') = \langle P', \pi' \rangle$, and assume, without loss of generality, that these pairs share no type variables. Let

$$P_i = P^i, x_1{:}\rho_1^i, \ldots, x_n{:}\rho_n^i$$

Let S be such that $S(\pi_i) = \tau_i \in \mathcal{T}_{\mathrm{C}}$, and $S(\rho_j^i) = \mu_j^i \in \mathcal{T}_{\mathrm{C}}$, for all $1 \leq i, j \leq n$, and let

$$S_i = \mathit{unify}(S_{i-1}(\mu_i^i), S_{i-1}(\tau_i)) \circ S_{i-1}$$

(with $S_0 = Id_S$). Then

$$pp_{\mathcal{E}}(\mathsf{cycle}\, \langle \overrightarrow{x_i = t_i} \rangle \,\mathsf{in}\, t') = S_n \circ S(\langle P' \cap P_1 \cap \ldots \cap P_n, \pi' \rangle).$$

(Notice that S can be built out of $to\mathcal{T}_C(\pi_i)$, $to\mathcal{T}_C(\rho_j^i)$, and unification.)

Since *unify* or $\mathit{unify}_2^{\forall}$ may fail, not every term has a principal pair.

Notice that, if $pp_{\mathcal{E}}(t) = \langle P, \pi \rangle$, then $\pi \in \mathcal{T}_2$. For example, the principal pair for I with rewrite rule $I\, x \to x$ is $\langle \emptyset, \varphi \to \varphi \rangle$, so, in particular, it is not $\langle \emptyset, \forall\alpha.\alpha \to \alpha \rangle$. 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 $\forall\alpha.\alpha \to \alpha$ can be obtained from $\varphi \to \varphi$ by closure, and $\varphi \to \varphi$ from $\forall\alpha.\alpha \to \alpha$ by lifting.

The following lemma is needed in the proof of Thm. 6.5. It states that if a chain maps the principal pairs of terms t_1, t_2 in an application $t_1\, t_2$ to pairs that allow the application itself to be typed, then these pairs can also be obtained by first performing a unification.

Lemma 6.2 [5] *Let $\sigma \in \mathcal{T}_2$, and $pp_{\mathcal{E}}(t_i) = \langle P_i, \pi_i \rangle$, for $i = 1, 2$, such that these pairs are disjoint. Let Ch_1, Ch_2 be chains such that $Ch_1(pp_{\mathcal{E}}(t_1)) = \langle B, \sigma \to \tau \rangle$ and $Ch_2(pp_{\mathcal{E}}(t_2)) = \langle B, \sigma \rangle$. Then there are chains Ch_u and Ch_p, and type $\rho \in \mathcal{T}_2$ such that*

$$\begin{aligned} pp_{\mathcal{E}}(t_1\, t_2) &= Ch_u(\langle P_1 \cap P_2, \rho \rangle), \text{ and} \\ Ch_p(pp_{\mathcal{E}}(t_1\, t_2)) &= \langle B, \tau \rangle. \end{aligned}$$

Similarly, we can show the following property

Lemma 6.3 *Let $\sigma \in \mathcal{T}_2$, and $pp_{\mathcal{E}}(t_1) = \langle P_1 \cup \{x{:}\rho\}, \pi_1 \rangle$, and $pp_{\mathcal{E}}(t_2) = \langle P_2, \pi_2 \rangle$, such that these pairs are disjoint. Let Ch_1, Ch_2 be chains such that*

$$Ch_1(pp_{\mathcal{E}}(t_1)) = \langle B \cap \{x{:}\sigma\}, \tau \rangle \;\&\; Ch_2(pp_{\mathcal{E}}(t_2)) = \langle B, \sigma \rangle.$$

Then there are chains Ch_u and Ch_p such that

$$pp_{\mathcal{E}}(\mathsf{share}\, x \,\mathsf{via}\, t_1 \,\mathsf{in}\, t_2) = Ch_u(\langle P_1 \cap P_2, \pi_1 \rangle), \text{ and}$$
$$Ch_p(pp_{\mathcal{E}}(\mathsf{share}\, x \,\mathsf{via}\, t_1 \,\mathsf{in}\, t_2)) = \langle B_1 \cap B_2, \tau \rangle.$$

The main result of this section then becomes the soundness and completeness result for $pp_{\mathcal{E}}$.

Theorem 6.4 (Soundness of $pp_{\mathcal{E}}$) *If $pp_{\mathcal{E}}(t) = \langle P, \pi \rangle$, then $P \vdash_{\mathcal{E}} t : \pi$.*

Theorem 6.5 (Completeness of $pp_{\mathcal{E}}$) *If $B \vdash_{\mathcal{E}} t : \sigma$, then there are a basis P and type π such that $pp_{\mathcal{E}}(t) = \langle P, \pi \rangle$, and there is a chain Ch such that $Ch(\langle P, \pi \rangle) = \langle B, \sigma \rangle$.*

References

[1] Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Fundamentae Informaticae*, 26(3,4):207–240, 1996. Extended version: CWI Report CS-R9552.

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

[3] S. van Bakel. Intersection Type Assignment Systems. *Theoretical Computer Science*, 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. Polymorphic Intersection Type Assignment for Rewrite Systems with Abstraction and β-rule. In Thierry Coquand, Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs. International Workshop, TYPES'99,* Lökeberg, Sweden, Sected Papers, volume 1956 of *Lecture Notes in Computer Science*, pages 41–60. Springer-Verlag, 2000.

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

[7] 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.

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

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

[10] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In *Proceedings of PARLE, Parallel Architectures and Languages Europe,* Eindhoven, The Netherlands,

volume 259-II of *Lecture Notes in Computer Science*, pages 141–158. Springer-Verlag, 1987.

[11] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Towards an Intermediate Language based on Graph Rewriting. In *Proceedings of PARLE, Parallel Architectures and Languages Europe,* Eindhoven, The Netherlands, volume 259-II of *Lecture Notes in Computer Science*, pages 159–175. Springer-Verlag, 1987.

[12] E. Barendsen and S. Smetsers. Extending Graph Rewriting with Copying. In Hans Jürgen Schneider and Hartmut Ehrig, editors, *Proceedings of International Workshop 'Graph Transformations in Computer Science',* Daghstuhl, Germany, January 1993, volume 776 of *Lecture Notes in Computer Science*, pages 51–70. Springer-Verlag, 1994.

[13] E. Barendsen and S. Smetsers. Conventional and Uniqueness Typing in Graph Rewrite Systems. *Mathematical Structures of Computer Science*, 1996.

[14] A. Bucciarelli, S. De Lorenzis, A. Piperno, and I. Salvo. Some computational properties of intersection types. In *Proc. Symposium on Logic in Computer Science (LICS'99)*, pages 109–118, 1996.

[15] L.M.M. Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, Department of Computer Science, Edinburgh, 1985. Thesis CST-33-85.

[16] F. Damiani. Typing local definitions and conditional expressions with rank 2 intersection. In *Proceedings of FOSSACS'00*, volume 1784 of *Lecture Notes in Computer Science*, pages 82–97. Springer-Verlag, 2000.

[17] F. Damiani and P. Giannini. A Decidable Intersection Type System based on Relevance. In M. Hagiya and J.C. Mitchell, editors, *Proceedings of TACS '94. International Symposium on Theoretical Aspects of Computer Software,* Sendai, Japan, volume 789 of *Lecture Notes in Computer Science*, pages 707–725. Springer-Verlag, 1994.

[18] F. Damiani and F. Prost. Detecting and Removing Dead-Code using Rank 2 Intersection. In *Proceedings of International Workshop TYPES'96, Selected Papers*, volume 1512 of *Lecture Notes in Computer Science*, pages 66–87. Springer-Verlag, 1998.

[19] 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.

[20] M. Eekelen, S. Smetsers, , and R. Plasmeijer. Graph Rewriting Semantics for Functional Programming Languages. In Dirk van Dalen, editor, *Proceedings of CSL '96, Fifth Annual conference of the European Association for Computer Science Logic (EACSL)*, volume 1258 of *Lecture Notes in Computer Science*, pages 106–128. Springer-Verlag, 1996.

[21] J.Y. Girard. The System F of Variable Types, Fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

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

[23] T. Jim. Rank 2 type systems and recursive definitions. Technical Memorandum MIT/LCS/TM-531, Laboratory for Computer Science Massachusetts Institute of Technology, 1995.

[24] T. Jim. What are principal typings and what are they good for? In *Proceedings of POPL'96. ACM Symposium on Principles of Programming Languages*, 1996.

[25] J. R. Kennaway, J. W. Klop, M. R. Sleep, and F. J. de Vries. The adequacy of term graph rewriting for simulating term rewriting. In M.R. Sleep, M.J. Plasmeijer, and M.C.D.J. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, pages 157–168. John Wiley & Sons, 1993.

[26] A. Kfoury and J. Wells. Principality and decidable type inference for finite-rank intersection types. In *Proceedings of POPL '99:* 26^{th} *ACM Symposium on the Principles of Programming. Languages*, pages 161–174, 1999.

[27] A.J. Kfoury, H.G. Mairson, F.A. Turbak, and J.B. Wells. Relating Typability and Expressibility in Finite-Rank Intersection Type Systems. In *Proceedings of ICFP '99, International Conference on Functional Programming*, pages 90–101, 1999.

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

[29] A.J. Kfoury and J. Wells. A Direct Algorithm for Type Inference in the Rank-2 Fragment of the Second-Order λ-Calculus. In *Proceedings of LFP'94: ACM Conference of LISP Functional Programming*, pages 196–207, 1994.

[30] 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.

[31] J. Launchbury and S.L. Peyton Jones. Lazy functional state threads. In PLDI, 1994.

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

[33] E.G.J.M.H. Nöcker, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J. Plasmeijer. Concurrent Clean. In *Proceedings of PARLE '91, Parallel Architectures and Languages Europe,* Eindhoven, The Netherlands, volume 506-II of *Lecture Notes in Computer Science*, pages 202–219. Springer-Verlag, 1991.

[34] J.C. Reynolds. Towards a Theory of Type Structures. In B. Robinet, editor, *Proceedings of Programming Symposium,* Paris, France, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

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

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

[37] R. Sleep, M.J. Plasmeijer, and M.C.J.C van Eekelen, editors. *Term Graph Rewriting. Theory and Practice.* Wiley, 1993.

[38] J. Wells. Typeability and type checking in Second order λ-calculus are equal and undecidable. In *Proceedings of the ninth Annual IEEE Symposium on Logic in Computer Science,* Paris, France, 1994.

[39] H. Yokohuchi. Embedding a Second-Order Type System into an Intersection Type System. *Information and Computation*, 117:206–220, 1995.