Rank 2 Types for Term Graph Rewriting

(Types in Programming (TIP’02); ENTCS, Volume 75, 2002)

Steffen van Bakel
Department of Computing, Imperial College,
80 Queen’s Gate, London SW7 2BZ, U.K
Email: svb@doc.ic.ac.uk

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 avail-
able that discusses and studies types directly on the level of the programming language. How-
ever, 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 (\(\lambda c\)) [8], or not expressible at all, like patterns, sharing, and cyclic structures. This
motivated the investigation of type assignment for Term Rewriting Systems (\(t\!r\!s\)) [30] and
Term Graph Rewriting Systems (\(t\!g\!r\!s\)) [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 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 uniqueness 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 \(t\!r\!s\) 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 \(\lambda c\) 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 (Grtrs) —a notion of first order TRS extended with application— by adding ‘∀’ 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 assignment 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 P2 [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 P2. 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 \( \mathsf{LC} \) and \( \mathsf{GTRS} \); 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\(^1\). This paper corrects those definitions and extends those results to a calculus with sharing and cycles, by defining a notion of Rank 2 type assignment on \( \mathsf{GTRS} \), 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 definitions of \( \mathsf{GTRS} \). The system defined here is aimed to be similar to those, although their relation is not studied here.

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

1 \ Application Term Graph Rewriting Systems

In this section, we will present a notion of Application Term Graph Rewriting (\( \mathsf{ATGRS} \)) 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 \( \Sigma \) consists of a countable, infinite set \( X \) of variables \( x, y, z, \ldots \), a non-empty set \( \mathcal{F} \) of function symbols \( F, G, \ldots \), each with a fixed arity \( \mathsf{arity}(F) \), and a special binary operator, called application \( (@, \text{written in in-fix notation}) \).

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

\[
t := x \mid F \mid (t_1 @ t_2) \mid (\text{share } t_1 \text{ via } x \text{ in } t_2) \mid (\text{cycle } \langle x_i = t_i \rangle \text{ in } t)
\]

We write \( (t_1 \cdot t_2) \) for \( (t_1 @ t_2) \), 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. 1.5(ii)).

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

\[
E ::= x \mid (F(E_1, \ldots, E_n)) \mid (\text{let } x = E_1 \text{ in } E_2) \mid (\text{letrec } x = E_1 \text{ in } E_2) \mid (\text{case } E \text{ of } \vec{P} \mid \vec{E})
\]

\[
P ::= C(x_1, \ldots, x_n)
\]

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.

---

\(^1\) The error mainly was in Lemma 5.1.3(ii) 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.
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 \(G\) is a pair \(g = (r \mid G)\), 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 ‘\(x = F\)’, 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 \(g = (r \mid G)\), \(Var(g)\), 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 \(t\), \(\llbracket t \rrbracket\), is defined by \((\vec{x_i} \mapsto \vec{r_i})\) stands for the simultaneous replacement of \(\vec{r_i}\) for \((\text{the free occurrences of}) \vec{x_i}\).

\[
\begin{align*}
\llbracket x \rrbracket &= \{x \mid \emptyset \\
\llbracket F \rrbracket &= \{f \mid \{f = F\}\} \\
\llbracket t_1 t_2 \rrbracket &= \{r \mid \{r = @(r_1, r_2)\} \cup G_1 \cup G_2\}, \text{ where } \llbracket t_i \rrbracket = \{r_i \mid G_i\}, i = 1, 2, \text{ and } r \text{ is fresh} \\
\llbracket \text{share } t_1 \text{ via } x \text{ in } t_2 \rrbracket &= \{r_2 \mid G_1 \cup G_2 \} \{x \mapsto r_1\}, \text{ where } \llbracket t_i \rrbracket = \{r_i \mid G_i\}, \text{ and } r \text{ is fresh} \\
\llbracket \text{cycle } (x_i = t_i) \text{ in } t' \rrbracket &= \{r' \mid G_1 \cup \cdots \cup G_n \cup G'\} \{x_i \mapsto r_i\}, \text{ where } \llbracket t_i \rrbracket = \{r_i \mid G_i\}, (1 \leq i \leq n) \text{ and } \llbracket t' \rrbracket = \{r' \mid G'\},
\end{align*}
\]

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 \((\text{share } t_1 \text{ via } x \text{ in } t_2\), \(x\) does not occur free in \(t_1\).

**Example 1.4** (cf. [13]) The term

\[(\text{share } 0 \text{ via } x \text{ in } (\text{cycle } (z = F (\text{cons } x (G x z))) \text{ in } z))\]

translates to the graph

\[
\begin{align*}
\llbracket x \rrbracket &= \{z = @ (f, a)\}, \\
f &= F, \\
a &= @ (b, c), \\
b &= @ (d, x), \\
c &= @ (e, z), \\
d &= \text{cons}, \\
e &= @ (g, x), \\
g &= G, \\
x &= 0\}
\end{align*}
\]

Reduction on \(T(F, X)\) is defined through rewrite rules.

**Definition 1.5** i) A rewrite rule is a pair \((\text{left}, \text{right})\) of terms such that

- \(\text{left} = F t_1 \cdots t_n\), for some \(F\) with \(n = \text{arity}(F)\), and terms \(t_1, \ldots, t_n\,\), and
- \(\text{fo(right)} \subseteq \text{fo(left)}\).
The translation into graphs of Def. 1.3 is extended to rewrite rules through: Let \( \text{left} \to \text{right} \) be a (recursive) rewrite rule with defined symbol \( F \), then:

\[
\llbracket \text{left} \to \text{right} \rrbracket = \langle r_l \mid G_{\text{left}} \cup G_{\text{right}} \rangle [x_i \mapsto y_i],
\]

where

\[
\llbracket F \rrbracket = \langle g \mid \{ g = F \} \rangle.
\]

\[
\llbracket \text{left} \rrbracket = \langle r_l \mid G_{\text{left}} \rangle.
\]

\[
\llbracket \text{right} \rrbracket = \langle r_r \mid G_{\text{right}} \rangle.
\]

\[
\{ x_1, \ldots, x_n \} = \text{fv} (\text{left})
\]

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 \( \llbracket t_1 \rrbracket = g_1, \llbracket t_2 \rrbracket = g_2, \) and \( g_1 \to g_2 \).

**Definition 1.7** An Applicative Term Graph Rewriting System (atgrs) is a pair \((\Sigma, R)\) of an alphabet \( \Sigma \) and a set \( R \) of rewrite rules.

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

\[
S \ x \ y \ z \ \rightarrow \ x \ z \ (y \ z)
\]

\[
K \ x \ y \ \rightarrow \ x
\]

\[
I \ x \ \rightarrow \ x
\]

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

![Diagram of term graph rewrite rules](image)

Notice that, if we would have used ‘\( S \ x \ y \ z \rightarrow \text{share} \ z \ via \ v \ in \ (x \ v) \ (y \ v) \)’ 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 graph \( g \) contains a redex if a left-hand side \( \text{left} \) of a rewrite rule \( \text{left} \to \text{right} \) can be mapped onto a graph, i.e. if there exists a homomorphism from \( \text{left} \) 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 \( \text{right} \) 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.

![Diagram of term graph rewriting](image)

**Figure 1:** An example of term graph rewriting

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 $V = \Phi \uplus A$ 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 $\Phi$) and (names of) bound type-variables (in $A$).

**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 $\Phi$ (ranged over by $\varphi$), sorts (type constants, ranged
over by s) and ‘→’, \( \mathcal{T}_C^i \) are quantified Curry types, \( \mathcal{T}_I \), the types of rank 1, are intersections of quantified Curry types, and \( \mathcal{T}_2 \) are types of Rank 2:

\[
\begin{align*}
\mathcal{T}_C & ::= \varphi \mid s \mid (\mathcal{T}_C \rightarrow \mathcal{T}_C) \quad \mathcal{T}_C^i ::= \varphi \mid (\forall \alpha . \mathcal{T}_C^i[\alpha/\varphi]) \\
\mathcal{T}_1 & ::= (\mathcal{T}_C^i \cap \cdots \cap \mathcal{T}_C^i) \quad \mathcal{T}_2 ::= \varphi \mid s \mid (\mathcal{T}_1 \rightarrow \mathcal{T}_2)
\end{align*}
\]

We use \( \mathcal{T}_R \) for the union of these sets, and use \( \sigma, \tau \) for arbitrary elements of \( \mathcal{T}_R \). Notice that \( \mathcal{T}_C \subseteq \mathcal{T}_C^i \subseteq \mathcal{T}_1 \) and \( \mathcal{T}_C \subseteq \mathcal{T}_2 \), but that \( \mathcal{T}_C^i \not\subseteq \mathcal{T}_2 \).

In the notation of types, ‘→’ is assumed to associate to the right, ‘∩’ binds stronger than ‘→’, which binds stronger than ‘∀’; so \( \rho \cap \mu \rightarrow (\forall \alpha \gamma \rightarrow \delta) \rightarrow \sigma \) stands for \( ((\rho \cap \mu) \rightarrow ((\forall \alpha \gamma) \rightarrow \delta)) \rightarrow \sigma \).

Also, \( \forall \alpha . \sigma \) 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 \( \sigma[\tau/\alpha] \) (resp. \( \sigma[\sigma[\tau/\alpha]] \)) the substitution of the type-variable \( \varphi \) (resp. \( \alpha \)) by \( \tau \) in \( \sigma \).

**Definition 2.2** \( \text{fv}(\sigma) \), the set of free variables of a type \( \sigma \) is defined as usual (note that by construction, \( \text{fv}(\sigma) \subseteq \Phi \)). 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 variables, not every syntactic sub-type of \( \sigma \in \mathcal{T}_R \) is necessarily a type in \( \mathcal{T}_R \), but ignoring this below will not affect any result.

**Definition 2.3** [5] On \( \mathcal{T}_R \), the pre-order (i.e. reflexive and transitive relation) ‘\( \leq \)’ is defined by:

\[
\begin{align*}
\sigma_1 \cap \cdots \cap \sigma_n \leq \sigma_i \quad & \quad (1 \leq i \leq n) \\
\forall \alpha . (\sigma[\alpha/\varphi]) \leq \sigma[\tau/\varphi], \quad & \quad (\tau \in \mathcal{T}_C) \\
\forall 1 \leq i \leq n \quad [\sigma \leq \sigma_i] \Rightarrow \sigma \leq \sigma_1 \cap \cdots \cap \sigma_n \quad & \quad (n \geq 1) \\
\rho \leq \sigma, \tau \leq \mu \Rightarrow \sigma \rightarrow \tau \leq \rho \rightarrow \mu, \quad (\tau, \mu \in \mathcal{T}_2) \\
\sigma \leq \tau \Rightarrow \forall \alpha . \sigma[\alpha/\varphi] \leq \forall \alpha . \tau[\alpha/\varphi].
\end{align*}
\]

**Definition 2.4** i) A statement is a term of the form \( t:\sigma \), with \( \sigma \in \mathcal{T}_R \) and \( t \in T(\mathcal{F}, \mathcal{X}) \). \( t \) is the subject and \( \sigma \) 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 \( \tau \) such that \( x:\tau \in B \), \( \varphi \in B \) if there is a type in \( B \) in which \( \varphi \) occurs, and write \( B \setminus x \) 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:

\[
B_1 \cap B_2 = \{ x:\tau : x:\tau \in B_1 \& x \not\in B_2 \} \cup \{ x:\tau : x:\tau \in B_2 \& x \not\in B_1 \} \cup \{ x:\tau_1 \cap \tau_2 : x:\tau_1 \in B_1 \& x:\tau_2 \in B_2 \}
\]

\[
B_1 \cup B_2 = B \setminus x \cup \{ x:\tau \}
\]

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 $\phi$ by $\tau_1 \cap \tau_2$ would transform $\sigma \rightarrow \phi$ into $\sigma \rightarrow \tau_1 \cap \tau_2$, which is not in $T_R$. However, since $T_C \subseteq T_2$, and $T_C$ is closed for (Curry-)substitution, also $T_2$ is closed for that kind of substitution.

The following definition takes this fact into account.

Definition 3.1  

(i) The substitution $(\phi \mapsto \rho) : T_2 \rightarrow T_2$, where $\phi$ is a type-variable in $\Phi$ and $\rho \in T_C$, is defined by:

\[
\begin{align*}
(\phi \mapsto \rho)(\phi) &= \rho \\
(\phi \mapsto \rho)(\phi') &= \phi', \text{if } \phi' \neq \phi \\
(\phi \mapsto \rho)(s) &= s \\
(\phi \mapsto \rho)(\alpha) &= \alpha \\
(\phi \mapsto \rho)(s \rightarrow \tau) &= (\phi \mapsto \rho)(s) \rightarrow (\phi \mapsto \rho)(\tau) \\
(\phi \mapsto \rho)(\sigma_1 \cap \cdots \cap \sigma_n) &= (\phi \mapsto \rho)(\sigma_1) \cap \cdots \cap (\phi \mapsto \rho)(\sigma_n) \\
(\phi \mapsto \rho)(\forall \alpha. \sigma) &= \forall \alpha.(\phi \mapsto \rho)(\sigma)
\end{align*}
\]

(ii) We use $Id_S$ for the substitution that replaces all type-variables by themselves, write $S$ 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 $L = \langle (B_1, \tau_1), (B_2, \tau_2) \rangle$ such that $\tau_1 \leq \tau_2$ and $B_2 \preceq B_1$, and is defined by $L(\langle B, \sigma \rangle) = \langle B', \sigma' \rangle$ where

\[
\begin{align*}
\sigma' &= \tau_2, \text{ if } \sigma = \tau_1, & B' &= B_2, \text{ if } B = B_1 \\
\sigma' &= \sigma, \text{ otherwise} & B' &= B, \text{ otherwise}
\end{align*}
\]

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

\[
L(\sigma) = \tau_2, \text{ if } \sigma = \tau_1 \\
\sigma, \text{ otherwise}
\]

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 $\langle \sigma, \phi \rangle$ with $\sigma \in T_C^{\phi}$, and is defined by:

\[
\langle \sigma, \phi \rangle (\langle B, \tau_1 \cap \cdots \cap \tau_n \rangle) = \langle B, \tau'_1 \cap \cdots \cap \tau'_n \rangle
\]

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

\[
\begin{align*}
\tau'_i &= \forall \alpha. \sigma[\alpha / \phi], \text{ if } \tau_i = \sigma, \text{ and } \phi \text{ does not appear in } B \\
\tau'_i &= \tau_i, \quad \text{otherwise}
\end{align*}
\]
Closure is extended to types by: \(\langle \varphi \rangle (\sigma) = (\tau)\), if \(\langle \varphi, \sigma \rangle (\langle \emptyset, \sigma \rangle) = (\emptyset, \tau)\).

**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\(_2^\emptyset\), 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 \(T_2^\emptyset\), 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, \(\sigma \in T_R\), and \(n \geq 1\). The \(n\)-fold expansion with respect to the pair \(\langle B, \sigma \rangle\), \(n_{\langle B, \sigma \rangle} : T_2 \rightarrow T_2\) 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 \(\varphi_i^j\) \((1 \leq i \leq m, 1 \leq j \leq n)\) does not occur in \(F\). Let \(S_i\) be the substitution that replaces every \(\varphi_i\) by \(\varphi_i^j\). Then expansion is defined on types, bases, and pairs, respectively, by:

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

Notice that, if \(\tau \in T_2\), 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 \(S_i(\tau) \in T_2\), 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 \(\sigma\), \(Ch_1(\sigma) = Ch_2(\sigma)\).

**4 Rank 2 Type Assignment**

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

**Definition 4.1** i) A Rank 2 environment \(E\) is a mapping from \(F\) to \(T_2\).

ii) Rank 2 type assignment on terms is defined by the following natural deduction system:

\[
\begin{align*}
(Ax) : \quad & B \vdash \varepsilon x : \tau \\
\quad & (x: \sigma \in B \& \sigma \leq \tau \& \sigma \in T_1 \& \tau \in T_2)
\end{align*}
\]

\[
\begin{align*}
(\land) : \quad & B \vdash \varepsilon t : \sigma_1 \quad \ldots \quad B \vdash \varepsilon t : \sigma_n \\
\quad & B \vdash \varepsilon t : \sigma_1 \cap \cdots \cap \sigma_n (n \geq 1 \& \forall 1 \leq i \leq n [\sigma_i \in T_2^\emptyset])
\end{align*}
\]

\[
\begin{align*}
(\rightarrow) : \quad & B \vdash \varepsilon t_1 : \sigma \rightarrow \tau \\
& B \vdash \varepsilon t_2 : \sigma \\
\quad & B \vdash \varepsilon t_1 t_2 : \tau
\end{align*}
\]
We write $B \vdash_\varepsilon t : \sigma$ if this is derivable using the rules above.

Notice the use of an environment and chain in rule $(F)$; 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 $(F)$ 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 $\text{ml}$-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 $B \vdash_\varepsilon t : \sigma$ if and only if $B \vdash_\varepsilon \exists t_1 : \sigma$.  

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

$$\mathcal{T}_C ::= \varphi \mid s \mid (\mathcal{T}_C \to \mathcal{T}_C) \mid [\mathcal{T}_C] \mid \text{Bool}$$

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

$$\text{IsNil } [\ ] \to \text{TT}$$

is typeable using the environment

$$\begin{align*}
\mathcal{E} (\text{tt}) &= \text{Bool} \\
\mathcal{E} ([\ ] ) &= [\varphi] \\
\mathcal{E} (\text{Cons}) &= \varphi \to [\varphi] \to [\varphi] \\
\mathcal{E} (\text{IsNil}) &= (\forall a. [a]) \to \text{Bool} \\
\end{align*}$$

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 $(F)$ 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 $(F)$ 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 \((\forall \forall)\), its use is in a strict system limited to variables, and there its actions are already performed by \((\text{Ax})\).

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 \(B \vdash \varepsilon \ 1 : \sigma\), then there are a basis \(P\) and type \(\pi\) such that \(\text{pp}_{\varepsilon} (t) = \langle P, \pi \rangle\), and there is a chain \(\text{Ch}\) such that \(\text{Ch} (\langle P, \pi \rangle) = \langle B, \sigma \rangle\).

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**

1. We say that \(\text{left} \rightarrow \text{right} \in \mathbb{R}\) with defined symbol \(F\) is typeable with respect to \(\varepsilon\), 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 \(\text{left} \rightarrow \varepsilon\) with respect to \(\varepsilon\).

   b) In \(P \vdash \varepsilon \ \text{left} : \pi\) and \(P \vdash \varepsilon \ \text{right} : \pi\) each occurrence of \(F\) is typed with \(\varepsilon (F)\).

2. We say that \((\Sigma, \mathbb{R})\) is typeable with respect to \(\varepsilon\), if all rules in \(\mathbb{R}\) are.

As an aside to part (i.b), remark that, by rule \((\varepsilon)\), we know that each occurrence of \(F\) has a type generated from \(\varepsilon (F)\) by applying a chain of operations. Part (i.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 \(\varepsilon (F)\) 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 \(\varepsilon (F)'\)’ becomes ‘the occurrence of \(F\) is typed with \(\varepsilon (F)'\).

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

If \(\sigma \in \mathcal{T}_1\), \(B \vdash \varepsilon \ 1 : \sigma\), and \(\text{Ch}\) is a chain of operations on types such that \(\text{Ch} (\langle B, \sigma \rangle) = \langle B', \sigma' \rangle\), then \(B' \vdash \varepsilon \ 1 : \sigma'\).

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 \(\varepsilon\) be an environment, \(t\) a term, and \(f\) a mapping from free variables to terms (which extends naturally to a mapping from terms to terms).

1. If \(B \vdash \varepsilon \ t : \sigma\) and \(B'\) is such that \(B' \vdash \varepsilon\ f (x) : \rho\) for every statement \(x : \rho \in B\), then \(B' \vdash \varepsilon\ f (t) : \sigma\).

2. If there are \(B\) and \(\sigma\) such that \(B \vdash \varepsilon \ f (t) : \sigma\), then for every \(x\) occurring in \(t\) there is a type \(\rho_x\) such that \(\{x : \rho_x \mid x \in \text{fv} (t)\} \vdash \varepsilon \ t : \sigma\), and \(B \vdash \varepsilon \ f (x) : \rho_x\).

Using this lemma, the following result follows easily.

**Theorem 4.6 (Subject Reduction)** If \(B \vdash \varepsilon \ t : \sigma\) and \(t \rightarrow t'\), then \(B \vdash \varepsilon \ t' : \sigma\).

**Example 4.6** Let \(\sigma, \tau, \rho, \mu, \nu, \gamma,\) and \(\delta\) be arbitrary types. Take the rewrite rules that define Combinatory Logic of Ex. 1.8, and the environment \(\varepsilon\):

\[
\varepsilon (S) = (\sigma \rightarrow \tau \rightarrow \rho) \rightarrow (\mu \rightarrow \tau) \rightarrow \sigma \cap \mu \rightarrow \rho \\
\varepsilon (K) = \nu \rightarrow \gamma \rightarrow \nu \\
\varepsilon (I) = \delta \rightarrow \delta
\]
Then these rules are typeable with respect to $E$; we show the derivation for the right-hand side of the first rule in Fig. 2.

$$
\begin{array}{llll}
B_1 \vdash x: \sigma \rightarrow \tau \rightarrow \rho & B_1 \vdash v: \sigma & B_1 \vdash y: \mu \rightarrow \tau & B_1 \vdash z: \mu \\
B_1 \vdash x \cdot v: \tau \rightarrow \rho & B_1 \vdash y \cdot v: \tau & B_1 \vdash z: \sigma \rightarrow \tau & B_1 \vdash z: \mu \\
\end{array}
$$

$B_1 \vdash (x \cdot v)(y \cdot v): \rho$

$B_1 \vdash \text{share z via } v \text{ in } (x \cdot v)(y \cdot v): \rho$

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

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 $\sigma \in T_1, B \vdash t: \sigma$, and Ch is such that $Ch(B, \sigma) = \langle B', \sigma' \rangle$, then $B' \vdash t: \sigma'$.

## 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 $\sigma \rightarrow \tau$, and $t_2$ has type $\rho$, then unification looks for a common instance of the types $\sigma$ and $\rho$ such that $(t_1, t_2)$ can be typed properly. The unification algorithm $\text{unify}_2$ 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 $\text{unify}$ [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:

$$
\begin{align*}
\text{unify}: T_C' \times T_C' & \rightarrow S \\
\text{unify}(\varphi, \varphi') & = (\varphi \mapsto \varphi'), \\
\text{unify}(\varphi, \tau) & = (\varphi \mapsto \tau), \text{ if } \varphi \text{ not in } \tau, \\
\text{unify}(\alpha, \alpha) & = \text{Id}_S, \\
\text{unify}(s, s) & = \text{Id}_S, \\
\text{unify}(\sigma, \varphi) & = \text{unify}(\varphi, \sigma), \\
\text{unify}(\sigma \rightarrow \tau, \rho \rightarrow \mu) & = S_2 \circ S_1, \\
\end{align*}
$$

where $S_1 = \text{unify}(\sigma, \rho)$,

$$
S_2 = \text{unify}(S_1(\tau), S_1(\mu)).
$$

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

It is worthwhile to notice that the operation on types returned by $\text{unify}$ is not really a substitution, since it allows, e.g., $(\varphi \mapsto \alpha)$, without keeping track of the binder for $\alpha$. 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 \textit{unify} returns the most general unifier for two Curry types, if it exists.

\textbf{Property 5.1} (\cite{35}) \textit{If two types have an instance in common, they have a highest common instance which is returned by \textit{unify}: for all } \sigma, \tau \in \mathcal{T}_C, \text{ substitutions } S_1, S_2: \text{ if } S_1(\sigma) = S_2(\tau), \text{ then there are substitutions } S_u \text{ and } S' \text{ such that}

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

The unification algorithm \textit{unify}_2 as defined below gets, typically, called during the computation of the principal pair for an application \( t_1 \circ t_2 \). Suppose the algorithm has derived \( P_1 \vdash \epsilon \ t_1 : \pi_1 \) and \( P_2 \vdash \epsilon \ t_2 : \pi_2 \) as principal pairs for \( t_1 \) and \( t_2 \), respectively, and that \( \pi_1 = \sigma \rightarrow \tau \).

Thus the demanded type \( \sigma \) is in \( \mathcal{T}_1 \) and the provided type \( \pi_2 \) is in \( \mathcal{T}_2 \). In order to be consistent, the result of the unification of \( \sigma \) and \( \pi_2 \) – a chain \( \text{Ch} \) – should always be such that \( \text{Ch}(\pi_2) \in \mathcal{T}_1 \). However, if \( \pi_2 \notin \mathcal{T}_C \), then in general \( \text{Ch}(\pi_2) \notin \mathcal{T}_1 \). To overcome this difficulty, an algorithm \( \text{to}\mathcal{T}_C \) will be inserted that, when applied to the type \( \rho \), returns a chain of operations that removes, if possible, intersections in \( \rho \). This can be understood by observing that, for example, \( ((\sigma \rightarrow \sigma) \rightarrow \sigma) \rightarrow \sigma \) is a substitution instance of \( ((\varphi_1 \rightarrow \varphi_1) \rightarrow \varphi_2) \cap (\varphi_3 \rightarrow \varphi_4 \rightarrow \varphi_4) \rightarrow \varphi_5 \). Note that if quantifiers appear in \( \rho \), \( \text{to}\mathcal{T}_C(\rho) \) should fail, since quantifiers that appear before an arrow cannot be removed by any of the operations on types defined above. Finally,

\[ \text{unify}_2^\gamma(\sigma, S_2(\pi_2), S_2(P_2)) \]

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

\textbf{Definition 5.2} The function \( \text{to}\mathcal{T}_C : \mathcal{T}_2 \rightarrow \mathcal{S} \) is defined by:

\[ \begin{align*}
\text{to}\mathcal{T}_C(\sigma) &= [\text{Id}_\mathcal{S}], \quad \text{if } \sigma \in \mathcal{T}_C \\
\text{to}\mathcal{T}_C((\sigma_1 \cap \ldots \cap \sigma_n) \rightarrow \mu) &= S' \circ S_u, \quad \text{otherwise,}
\end{align*} \]

where \( S_i = \text{unify}(S_{i-1}(\sigma_1), S_{i-1}(\sigma_{i+1})) \circ S_{i-1}, (1 \leq i \leq n-1, \text{ with } S_0 = \text{Id}_\mathcal{S}) \)

\[ S' = \text{to}\mathcal{T}_C(S_n(\mu)) \]

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

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

The following definition presents the main unification algorithm, \textit{unify}_2^\gamma.

\textbf{Definition 5.3} The function \textit{unify}_2 is defined by:

\[ \begin{align*}
\text{unify}_2^\gamma(\varphi, \tau, B) &= [(\varphi \rightarrow \tau)], \\
\text{unify}_2^\gamma((\forall \alpha_1. \sigma_1) \cap \ldots \cap (\forall \alpha_n. \sigma_n), \tau, B) &= [\text{Ex}, S_n], \quad \text{otherwise,}
\end{align*} \]

where \( \text{Ex} = n_{(B, \tau)} \),

\[ \tau_1 \cap \ldots \cap \tau_n = \text{Ex}(\tau), \text{ and} \]

for every \( 1 \leq i \leq n \), \( S_i = \text{unify}(S_{i-1}(\sigma_i), \tau_i) \circ S_{i-1} \) (with \( S_0 = \text{Id}_\mathcal{S} \)).
The procedure \( \text{unify}^T \) fails when \( \text{unify} \) fails, and \( \to T_C \) fails when either \( \text{unify} \) fails or when the argument contains ‘\( \forall \)’. Because of this relation between \( \text{unify}^T \) and \( \to T_C \) on one side, and \( \text{unify} \) 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 \( E - pp_C (t) - \) is defined, consisting of basis \( P \) and type \( \pi \). 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 \( T (E, X) \). \( pp_C (t) = \langle P, \pi \rangle \), with \( \pi \in T_2 \), is defined, using \( \text{unify}^T \), by induction to the structure of terms through:

- \( (x) \): Then \( pp_C (x) = \langle \{ x: \phi \}, \phi \rangle \).
- \( (F) \): \( pp_C (F) = \langle \emptyset, E (F) \rangle \).
- \( (t_1 t_2) \): Let \( pp_C (t_1) = \langle P_1, \pi_1 \rangle \), \( pp_C (t_2) = \langle P_2, \pi_2 \rangle \) (choose, if necessary, trivial variants such that these pairs share no type variables, and \( S_2 = \to T_C (\pi_2) \), then
  
  \( (\pi_1 = \phi) \): \( pp_C (t_1 t_2) = \langle P, \pi \rangle \), where
  
  \[ \langle P, \pi \rangle = \langle S_1 (P_1 \cap S_2 (P_2)), \phi' \rangle, \]
  
  \[ S_1 = (\phi \mapsto S_2 (\pi_2) \mapsto \phi'), \text{ and } \phi' \text{ is a fresh variable.} \]

- \( (\pi_1 \rightarrow \tau) \): \( pp_C (t_1 t_2) = \langle P, \pi \rangle \), provided \( P \) and \( \pi \) contain no unbound occurrences of \( \alpha \), where
  
  \[ \langle P, \pi \rangle = \langle S (P_1 \cap Ex (S_2 (P_2))), S (\tau) \rangle, \]
  
  \[ [Ex, S] = \text{unify}^T (\pi_2, S_2 (\pi_2), S_2 (P_2)). \]

- \( (\text{share } t_1 \text{ via } x \text{ in } t_2) \): Let \( pp_C (t_i) = \langle P_i, \pi_i \rangle \), for \( i = 1, 2 \). Then either:
  
  - \( (x \text{ occurs in } t_1) \). Then there exists \( P', \sigma \in T_1 \) such that \( P_1 = P', x: \sigma \). Let \( S_2 = \to T_C (\pi_2) \). Then
    
    \[ pp_C (\text{share } t_1 \text{ via } x \text{ in } t_2) = \langle P, \pi \rangle, \]
    
    provided \( P \) and \( \pi \) contain no unbound occurrences of \( \alpha \), where
    
    \[ \langle P, \pi \rangle = \langle S (P' \cap Ex (S_2 (P_2))), S (\pi_1) \rangle, \]
    
    \[ [Ex, S] = \text{unify}^T (\sigma, S_2 (\pi_2), S_2 (P_2)). \]
  
  - \( (x \text{ does not occur in } t_1) \). Then
    
    \[ pp_C (\text{share } t_1 \text{ via } x \text{ in } t_2) = \langle P_1, \pi_1 \rangle. \]

- \( (\text{cycle } (\overline{x_i = t_i}) \text{ in } t') \): Let, for \( 1 \leq i \leq n \), \( pp_C (t_i) = \langle P_i, \pi_i \rangle \), and \( pp_C (t') = \langle P', \pi' \rangle \), and assume, without loss of generality, that these pairs share no type variables. Let
  
  \[ P_i = P^i, x_1: p^i_1, \ldots, x_n: p^i_n, \]

  Let \( S \) be such that \( S (\pi_i) = \tau_i \in T_C \), and \( S (\rho^j_i) = \mu^j_i \in T_C \), for all \( 1 \leq i, j \leq n \), and let
  
  \[ S_i = \text{unify} (S_{i-1} (\mu^j_i), S_{i-1} (\tau_i)) \circ S_{i-1} \](with \( S_0 = Id_S \)). Then
  
  \[ pp_C (\text{cycle } (\overline{x_i = t_i}) \text{ in } t') = S_n \circ S (\langle P' \cap P_1 \cap \ldots \cap P_n, \pi' \rangle). \]

(Notice that \( S \) can be built out of \( \to T_C (\pi_i) \), \( \to T_C (\rho^j_i) \), and unification.)
Since \textit{unify} or \textit{unify}_t^\forall \textit{may fail, not every term has a principal pair.}

Notice that, if \(ppE(t) = \langle P, \pi \rangle\), then \(\pi \in T_2\). For example, the principal pair for \(l\) with rewrite rule \(lx \rightarrow x\) 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.

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.

\textbf{Lemma 6.2} [5] Let \(\sigma \in T_2\), and \(ppE(t_i) = \langle P_i, \pi_i \rangle\), for \(i = 1, 2\), such that these pairs are disjoint. Let \(Ch_1, Ch_2\) be chains such that \(Ch_1(ppE(t_1)) = \langle B, \sigma \rightarrow \tau \rangle\) and \(Ch_2(ppE(t_2)) = \langle B, \sigma \rangle\). Then there are chains \(Ch_u\) and \(Ch_p\), and type \(\rho \in T_2\) such that

\[
ppE(t_1, t_2) = Ch_u(\langle P_1 \cap P_2, \rho \rangle), \text{ and } Ch_p(ppE(t_1, t_2)) = \langle B, \tau \rangle.
\]

Similarly, we can show the following property

\textbf{Lemma 6.3} Let \(\sigma \in T_2\), and \(ppE(t_1) = \langle P_1 \cup \{x: \rho\}, \pi_1 \rangle\), and \(ppE(t_2) = \langle P_2, \pi_2 \rangle\), such that these pairs are disjoint. Let \(Ch_1, Ch_2\) be chains such that

\[
Ch_1(ppE(t_1)) = \langle B \cap \{x: \sigma\}, \tau \rangle \land Ch_2(ppE(t_2)) = \langle B, \sigma \rangle.
\]

Then there are chains \(Ch_u\) and \(Ch_p\) such that

\[
ppE(\text{share } x \text{ via } t_1 \text{ in } t_2) = Ch_u(\langle P_1 \cap P_2, \pi_1 \rangle), \text{ and } Ch_p(ppE(\text{share } x \text{ via } t_1 \text{ in } t_2)) = \langle B_1 \cap B_2, \tau \rangle.
\]

The main result of this section then becomes the soundness and completeness result for \(ppE\).

\textbf{Theorem 6.4 (Soundness of \(ppE\))} If \(ppE(t) = \langle P, \pi \rangle\), then \(P \vdash E t : \pi\).

\textbf{Theorem 6.5 (Completeness of \(ppE\))} If \(B \vdash E t : \sigma\), then there are a basis \(P\) and type \(\pi\) such that \(ppE(t) = \langle P, \pi \rangle\), and there is a chain \(Ch\) such that \(Ch(\langle P, \pi \rangle) = \langle B, \sigma \rangle\).

\textbf{References}


