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

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

[1] Department of Computing, Imperial College,
180 Queen's Gate, London SW7 2BZ. svb@doc.ic.ac.uk
[2] Dipartimento di Matematica e Informatica, Università degli Studi di Catania,
Viale A. Doria 6, 95125 Catania, Italia. barba@dipmat.unict.it
[3] LIENS (CNRS UMR 8548), École Normale Supérieure,
45, rue d'Ulm, 75005 Paris, France. maribel@ens.fr

Abstract. We define two type assignment systems for first-order rewriting extended with application, λ-abstraction, and β-reduction, using a combination of (ω-free) intersection types and second-order polymorphic types. The first system is the general one, for which we prove subject reduction, and strong normalisation of typeable terms. The second is a decidable subsystem of the first, by restricting to Rank 2 (intersection and quantified) types. For this system we define, using an extended notion of unification, a notion of principal typing which is more general than ML's principal type property, since also the types for the free variables of terms are inferred.

Introduction

Since the first investigations on combinations of λ-calculus (LC) and term rewriting systems (TRS) [11, 17, 12, 24], this topic has drawn attention from the theoretical computer science community. At first is was mainly the area of programming language design to be seen as the typical ground in which the theoretical results about the combinations of the two computational paradigms could better exploit their potentialities. Later on, the evolution of interactive proof development tools with inductive types, and theorem provers in general, disclosed a number of possible applications.

Apart from the practical outcome, most of the theoretical investigations in this particular field have shown that type disciplines provide an optimal environment in which rewrite rules and β-reduction can interact without loss of their useful properties [12, 13, 24, 7, 8, 5]. Type disciplines come in two main flavours: *explicitly typed* and *type inference systems* (also called à la Curry). Systems based on the latter sort of type discipline are of great interest from the programming language design point of view. In fact, they save the programmer from specifying a type for each variable (i.e. no type annotation is required). Most of the results about combinations of LC and TRS, however, concern systems which are explicitly typed.

* Partially supported by NATO Collaborative Research Grant CRG 970285 'Extended Rewriting and Types'.
** Partially supported by MURST COFIN'99 TOSCA Project.

In the context of the LC alone, type inference disciplines have been widely studied, in particular with intersection types, and some work has also been done for TRS alone, more precisely, for *Curryfied* TRS (@TRS) [6] which are first-order TRS with application, that correspond to the TRS underlying the programming language Clean [32]. The interactions between LC and TRS for type systems à la Curry, instead, has not been extensively investigated. They were first studied in [5], where @TRS extended with λ-abstraction and β-reduction were defined, together with a notion of intersection type assignment for both the LC and the TRS fragments. In this paper we carry on this study by taking *explicit polymorphism* into account, namely the possibility, from the programming language point of view, of using the same program with arguments of different types.

We take into account also another important feature of type systems for programming languages: the notion of *principal type*, that is, a type from which all the other types of a term can be derived. As a matter of fact we consider an even stronger property than principal types: *the principal typing property*. This means that any *typing judgement* for a term can be obtained from the principal one, that is, not only the type but also the basis (containing the type assumptions for the free variables) is obtained. The pragmatic value of this property is demonstrated in [23], where it is shown that, unlike principal types, principal typings provide support for separate compilation, incremental type inference, and for accurate type error messages.

The type system of ML is polymorphic and has principal types, but its polymorphism is limited (some programs that arise naturally cannot be typed), and it does *not* have principal typings (see [15, 23]). System F [21] provides a much more general notion of polymorphism, but lacks principal types, and type inference is undecidable in general (although it is decidable for some subsystems, in particular if we consider types of rank 2 [25]). Intersection type systems [10] are somewhere in the middle with respect to polymorphism, and have principal typings. But type assignment is again undecidable; decidability is recovered if we restrict ourselves to intersection types of finite rank [27]. In view of the above results, two questions arise naturally:

1. Is the rank 2 combination of System F and the Intersection System also decidable?
2. Does it have the principal typing property?

Concerning the latter, there is a system for LC combining intersection types and System F, which has principal typings (see [22, 30]). In this paper we extend the approach of that system to a combination of LC and @TRS. In other words, we extend the type system of [5] further, adding '$\forall$' as an extra type-constructor (i.e. explicit polymorphism). Although extending the set of types by adding '$\forall$' does not extend the expressivity of the system in terms of typeable terms, the set of assignable types increases, and types can better express the behaviour of terms (see [14]). The resulting system has the expected properties: Subject Reduction, and Strong Normalization when the rewrite rules use a limited form of recursion (inspired by the General Schema of Jouannaud and Okada [24]). The proof of the latter follows the method of Tait-Girard's reducibility candidates, extended in order to take the presence of (higher-order) algebraic rewriting into account.

We also answer the first question in the affirmative. The restriction to types of rank 2 of the combined system of polymorphic and intersection types is decidable. This re-

stricted system can be seen as a combination of the systems considered in [4] and [25]. The combination is two-fold: not only the type systems of those two papers are combined (resp. intersection and polymorphic types of rank 2), but also their calculi are combined (resp. @TRS and LC). 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 [25]. For the latter, a type inference algorithm of the same complexity of that of ML was given in [26], where the problems that occur due to the lack of principal types are discussed in detail. Our Rank 2 system generalizes also Jim's system P_2 [23], 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 [29], which cannot be typed in P_2.

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

1 Term Rewriting Systems with β-reduction

We present a combination of Lambda Calculus with Algebraic Rewriting, obtained by extending TRS with application, abstraction, and β-reduction. We can also look at such calculi as extensions of Curryfied Term Rewriting Systems (@TRS) [6], by adding λ-abstraction and β-reduction. We assume the reader to be familiar with LC [9] and refer to [28, 16] for rewrite systems.

Definition 1 (Terms). Given a countable infinite set $\mathcal{X}$ of variables $x_1, x_2, x_3, \ldots$ (or $x, y, z, x', y', \ldots$), a non-empty set $\mathcal{F}$ of *function symbols* F, G, $\ldots$, each with a fixed arity, and a special binary operator, called *application* (Ap), the set $T(\mathcal{F}, \mathcal{X})$ of *terms* is defined inductively:

1. $\mathcal{X} \subseteq T(\mathcal{F}, \mathcal{X})$.
2. If $F \in \mathcal{F} \cup \{Ap\}$ is an n-ary symbol ($n \geq 0$), and $t_1, \ldots, t_n \in T(\mathcal{F}, \mathcal{X})$, then $F\,(t_1, \ldots, t_n) \in T(\mathcal{F}, \mathcal{X})$.
3. If $t \in T(\mathcal{F}, \mathcal{X})$, and $x \in \mathcal{X}$, then $\lambda x.t \in T(\mathcal{F}, \mathcal{X})$.

Terms are considered modulo α-conversion. A *context* C[ ] is a term with a hole. A *neutral* term is a term not of the form $\lambda x.t$. A *lambda term* is a term not containing function symbols. An *algebraic term* is a term containing neither λ nor Ap. The set of *free variables* of a term t is defined as usual, and denoted by $FV(t)$.

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

Definition 2 (Reduction). A *rewrite rule* is a pair (l, r) of terms, written $l \to r$. Three conditions are imposed: $l \notin \mathcal{X}$, l is an algebraic term, and $FV(r) \subseteq FV(l)$. The β-rule is defined by: $Ap(\lambda x.t, u) \;\to_\beta\; t^{\{x \mapsto u\}}$. A rewrite rule $l \to r$ determines a set of

rewrite steps $\mathrm{C}[l^{\mathrm{R}}] \to \mathrm{C}[r^{\mathrm{R}}]$ for all term-substitutions R and contexts C[ ]. Likewise, for any t and u, $\mathrm{C}[Ap(\lambda x.t, u)] \to_\beta \mathrm{C}[t^{\{x \mapsto u\}}]$. Concatenating rewrite steps we have *rewrite sequences* $t_0 \to t_1 \to t_2 \to \cdots$. If $t_0 \to \cdots \to t_n$ ($n \geq 0$) we also write $t_0 \to^* t_n$, and $t_0 \to^+ t_n$ if $t_0 \to^* t_n$ in one step or more.
A *Term Rewriting System with β-reduction rule* (TRS + β) is defined by a set **R** of rewrite rules.

Note that the rewrite rules considered in this paper may contain λ-abstractions in the right-hand sides. A TRS + β is strongly normalizing, or terminating, if all the rewrite sequences are finite. It is confluent if for all t such that $t \to^* u$ and $t \to^* v$, there exists s such that $u \to^* s$ and $v \to^* s$. We take the view that in a rewrite rule a certain symbol is defined: F is *a defined symbol* if there is a rewrite rule $F(t_1, \ldots, t_n) \to r$. $Q \in \mathcal{F}$ is called a *constructor* if Q is not a defined symbol. Notice that Ap cannot be a defined symbol since it cannot appear in the left-hand side of a rewrite rule.

Example 3. The following TRS + β defines the functions append and map on lists and establishes the associativity of append; nil and cons are constructors.

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

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

One could argue that the map function of the above example works properly only when f is replaced by a λ-expression. What if we would like to substitute a defined symbol, say G (with arity 1), for the variable f? This is not allowed by our syntax, since Curryfication is not used on algebraic terms, i.e. an element of the signature is not a term by itself. This makes sense; otherwise we could reduce $\mathsf{map}(G, \mathsf{cons}(y, l))$ to $\mathsf{cons}(Ap(G, y), \mathsf{map}(f, l))$ and this would force us to add a further rule stating that $Ap(G, y)$ can be reduced to $G(y)$, so adding extra rules and allowing also Ap in left-hand sides of reduction rules. In our approach, instead, using λ-abstraction, the desired result is obtained simply by instantiating f with the correct term $\lambda x.G(x)$. It will be the β-rule which will produce $G(x)$ out of $Ap(\lambda x.G(x), y)$. This means that we can treat the "mapping" of either a defined symbol or of a lambda-calculus function uniformly.

Notice how, by allowing λ-abstractions in the right-hand sides, it is possible to define rules like $\mathsf{maptwice}(f, l) \to \mathsf{map}(\lambda x.Ap(f, Ap(f, x)), l)$.

2 A Polymorphic Intersection System for TRS + β

We define a type assignment system for TRS+β that contains sorts (constant types), arrow-, intersection- and universally quantified types. We assume the reader to be familiar with intersection type assignment systems, and refer to [10, 1, 3] for more details.

Several systems can be considered as fragments of our type assignment system:

1. The system of [5]: $\forall$-free fragment
2. The system of [6]: λ-free, $\forall$-free fragment
3. The system of [3]: sort-free, $\forall$-free, LC-fragment
4. The type assignment version of System F [21] : intersection-free LC-fragment
5. The system of [30]: sort-free LC-fragment

Indeed, the latter system is not a proper fragment, since we use *strict* intersection types (i.e. an intersection type cannot be the right-hand side of an arrow type). However this is not an actual difference, since the use of strict intersection types, while simplifying the typing procedures, does not affect the typing power. Any term typeable in the full intersection type discipline can be given a strict type and vice-versa [1].

For what concerns System F, its type assignment version can be seen as a fragment of our system, but our system is not a pure intersection-extension: we cannot quantify intersection types. Again, this is not a real problem: for any universally quantified intersection type we have an equivalent type of the form $(\forall\alpha.\sigma_1)\cap(\forall\alpha.\sigma_2)$.

Definition 4 (Types). Let $\Phi \uplus \mathcal{A}$ be a set of *type-variables*, where $\Phi = \{\varphi_0, \varphi_1, \ldots\}$ is the set of *free* variables, and $\mathcal{A} = \{\alpha_0, \alpha_1, \ldots\}$ the set of *bound* variables. Let $\mathcal{S} = \{s_0, s_1, \ldots\}$ be a set of sorts. $\mathcal{T}_s$, the set of *polymorphic strict types*, and $\mathcal{T}$, the set of *polymorphic strict intersection types*, are defined by:

$$\mathcal{T}_s ::= \varphi \mid s \mid (\mathcal{T} \to \mathcal{T}_s) \mid \forall\alpha.\mathcal{T}_s[\alpha/\varphi],\ \mathcal{T} ::= (\mathcal{T}_s \cap \cdots \cap \mathcal{T}_s)$$

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

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

Definition 5 (Relations on types). The relation $\leq$ is defined as the least preorder (i.e. reflexive and transitive relation) on $\mathcal{T}$ such that:

$$\forall n \geq 1, \forall 1 \leq i \leq n [\sigma_1\cap\cdots\cap\sigma_n \leq \sigma_i] \qquad \forall\alpha.\sigma\to\tau \leq \sigma\to\forall\alpha.\tau, (\alpha \text{ not in } \sigma)$$
$$\sigma \leq \tau \ \Rightarrow\ \forall\alpha.\sigma[\alpha/\varphi] \leq \forall\alpha.\tau[\alpha/\varphi] \qquad \forall\alpha.\sigma \leq \sigma[\tau/\alpha]$$
$$\rho \leq \sigma \& \tau \leq \mu \ \Rightarrow\ \sigma\to\tau \leq \rho\to\mu \qquad \sigma \leq \forall\alpha.\sigma, (\alpha \text{ is fresh})$$
$$\forall n \geq 1, \forall 1 \leq i \leq n\ [\sigma \leq \sigma_i] \Rightarrow \sigma \leq \sigma_1\cap\cdots\cap\sigma_n$$

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

2.1 Type assignment

Definition 6 (Statement and Basis).

1. A *statement* is an expression of the form $t{:}\sigma$, where $\sigma \in \mathcal{T}$. The term t is the *subject* and σ the *predicate* of $t{:}\sigma$.

2. A *basis* is a set of statements with only distinct variables as subjects.
3. For bases $B_1, \ldots, B_n$, $\Pi\{B_1, \ldots, B_n\}$ is the basis defined by: $x{:}\sigma_1 \cap \cdots \cap \sigma_m \in \Pi\{B_1, \ldots, B_n\}$ if and only if $\{x{:}\sigma_1, \ldots, x{:}\sigma_m\}$ is the (non-empty) set of all statements about x that occur in $B_1 \cup \cdots \cup B_n$.
4. We extend $\leq$ and $\sim$ to bases by: $B \leq B'$ if and only if for every $x{:}\sigma' \in B'$ there is an $x{:}\sigma \in B$ such that $\sigma \leq \sigma'$, and $B \sim B'$ if and only if $B \leq B' \leq B$.

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

One of the main features of type assignment systems, and intersection systems in particular, is to provide *flexibility* of typing. This feature seems in contrast with the type *rigidity* implicitly possessed by function symbols. They have precise arities and a precise functional behaviour, as expressed by the rewriting rules.

Developing a type assignment containing function symbols necessarily implies a sort of mediation, for what concerns algebraic terms, between *flexibility* and *rigidity*. We achieve this by using an *environment* providing a type for each function symbol. Such a type, however, is not rigid: from it we can derive many types to be used for different occurrences of the symbol in a term, all of them 'consistent' with the type provided by the environment.

Definition 7 (Environment). An *environment* is a mapping $\mathcal{E} : \mathcal{F} \to \mathcal{T}_s$.

In order to obtain valid instances of the type provided by an environment for a function symbol we will use operations which are standard in type systems with intersection types, suitably modified in order to take into account the presence of universal quantifiers. These operations are: *substitution*, *expansion*, *lifting* and *closure*.

In type systems based on arrow types with type-variables, the operation of *substitution* generates all valid instances of a given type by replacing types for type variables. In presence of intersection types, valid instances could also be the result of replacing (sub)types of substitution instances by the intersection of a number of renamed copies of that (sub)type. This is (roughly) what is performed by the operation of *expansion*. The operation of *lifting*, instead, generates instances of types using the $\leq$ relation. The last operation we consider, *closure*, is not present in other type systems with intersection types and has been devised to deal in particular with universal quantification.

In the following we shall have to consider the notion of *principal typing* for a term, that is the typing from which all the possible typings for the term can be derived. This can be achieved by means of the above discussed operations. This implies that we shall have to define the above operations not only on types, but on type derivations as well. We shall use a triple $\langle B, \tau, E\rangle$ – where B is a basis, τ a type, and E a set of types – as an abstraction of a derivation for a term. Theorem 19 will guarantee that a triple is a sound representation of a derivation from the point of view of operations. Indeed for any operation *op*, $B \vdash_{\mathcal{E}} t{:}\sigma$ implies $B' \vdash_{\mathcal{E}} t{:}\sigma'$, where the triple representing the second derivation is obtained from the one for the first derivation by means of *op*.

In the following, we shall define two versions of any of our four operations, one to be applied on types, the other one on triples representing typing derivations. For any

operation we shall use the same symbols to denote the two versions, since the context will always clarify any possible ambiguity.

For substitution and lifting the versions on triples will be straightforward extensions of those on types, whereas for the other operations, expansion in particular, even if strongly related, the two versions will not be simple extensions of each other.

Definition 8. The *substitution* $(\varphi \mapsto \rho) : \mathcal{T} \to \mathcal{T}$, where φ is a type-variable in Φ and $\rho \in \mathcal{T}_s$, applied to σ replaces all the occurrences of φ in σ by ρ, formally:

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

We will use S to denote a generic substitution. Substitutions extend to triples in the natural way: $S(\langle B, \sigma, E\rangle) = \langle \{x{:}S(\rho) \mid x{:}\rho \in B\}, S(\sigma), \{S(\rho) \mid \rho \in E\}\rangle$.

Two different definitions of expansion appear in the literature for LC, depending on whether one uses a set of types (see e.g. [34]) or a set of type variables (see e.g. [3]) to characterise the set of types affected by the expansion. We adapt to our system the definition given in [34]. The extension to deal with types containing sorts has already been done in [18], here quantifiers are also taken into account.

As mentioned above, the operation of expansion deals with the replacement of a subtype of a type by an intersection of a number of renamed copies of that subtype. Hence an expansion is determined by a pair $\langle \mu, n\rangle$ which indicates the subtype to be expanded and the number of copies that have to be generated. When a subtype is expanded, new type variables are generated, and other subtypes might be affected (e.g. the expansion of τ in $\sigma \to \tau$ might affect also σ: intuitively, each renamed copy of τ will have an associated copy of σ; see [34] for a detailed explanation). Ground types are not affected by expansions since all renamed copies coincide (and $\sigma \cap \sigma \sim \sigma$).

Also, we need to define the notion of expansion not only for types, but for derivations as well, represented by triples. An expansion applied to a derivation should "produce" another well-formed derivation. This however cannot be the case unless, when defining the expansion on triples, the set of types that might be affected by the expansion includes also the types in the basis and those assigned to the function symbols in the derivation (represented by the set E of the triple). This means that the definition of expansion on triples cannot, like for substitution, be a simple extension of that on types. However, instead of defining two quite different versions of expansion (for types and for derivations), we shall define first the notion of expansion on triples, depending on a pair $\langle \mu, n\rangle$, and later the notion of expansion on types based on the one on triples.

In the definition of expansion, given a pair $\langle \mu, n\rangle$, to compute the triple obtained by applying the expansion determined by the given pair to a triple $\langle B, \sigma, E\rangle$, we first compute a set $\mathcal{L}_\mu(\langle B, \sigma, E\rangle)$ of affected types. The types modified by the expansion will be those that 'end' with a type in this set. The notion of *last subtypes* in a strict type plays an important rôle in this operation.

Definition 9. The set of *last subtypes* of a type $\tau \in \mathcal{T}_s$, *last* (τ), is defined by:

$$\begin{array}{llll} \mathit{last}\,(\varphi) & = \{\varphi\} & \mathit{last}\,(\sigma \to \rho) & = \{\sigma \to \rho\} \cup \mathit{last}\,(\rho) \\ \mathit{last}\,(s) & = \{s\} & \mathit{last}(\forall \alpha.\sigma) & = \{\forall \alpha.\sigma\} \cup \mathit{last}(\sigma[\varphi_\alpha/\alpha]) \end{array}$$

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

The definition of expansion, already non-trivial in the intersection system, becomes quite involved in the presence of universal quantifiers. We define it in four steps.

Definition 10 (Expansion on triples). Let μ be a type in $\mathcal{T}$, $n \geq 2$. The pair $\langle \mu, n\rangle$ determines an *expansion* $E_{\langle \mu,n\rangle}$, that is a function from triples to triples in the following way: Let $\langle B, \sigma, E\rangle$ be a triple (B a basis, $\sigma \in \mathcal{T}$ and E a set of types in $\mathcal{T}$; we assume that each variable is bound at most once in μ, B, σ, E). The triple $E_{\langle \mu,n\rangle}(\langle B, \sigma, E\rangle)$ is computed applying the following steps.

1. Affected types: Let $\mathcal{L}_\mu(\langle B, \sigma, E\rangle)$ be the set of types defined by:
   (a) Any non-closed strict subtype of μ is in $\mathcal{L}_\mu(\langle B, \sigma, E\rangle)$.
   (b) Let τ be a non-closed strict (sub)type occurring in $\langle B, \sigma, E\rangle$. If τ' is a most general instance (with respect to the universal quantifiers) of τ such that $\mathit{last}(\tau') \cap \mathcal{L}_\mu(\langle B, \sigma, E\rangle) \neq \emptyset$, then $\tau' \in \mathcal{L}_\mu(\langle B, \sigma, E\rangle)$.
   (c) Any non-closed strict subtype of $\tau \in \mathcal{L}_\mu(\langle B, \sigma, E\rangle)$ is in $\mathcal{L}_\mu(\langle B, \sigma, E\rangle)$.
2. Renamings: Let $\mathcal{V}_\mu(\langle B, \sigma, E\rangle) = \{\varphi_1, \ldots, \varphi_m\}$ be the set of free type variables occurring in B, σ, μ, E that appear in $\mathcal{L}_\mu(\langle B, \sigma, E\rangle)$, and let S_i ($1 \leq i \leq n$) be the substitution that replaces every φ_j by a fresh variable φ_j^i, and every α_j and φ_{α_j} by α_j^i (actually, S_i is just a renaming).
3. Definition of the function $\mathit{Ex}_{\mathit{aux}}$ from types to types: for any $\tau \in \mathcal{T}$ (without loss of generality we assume that its bound variables are disjoint with those of μ, B, σ, E) the type $\mathit{Ex}_{\mathit{aux}}(\tau)$ is obtained out of τ by traversing τ top-down and replacing in τ a maximal non-closed subtype β such that there exists a most general instance (w.r.t. the universal quantifiers) β' of β with $\mathit{last}(\beta') \cap \mathcal{L}_\mu(\langle B, \sigma, E\rangle) \neq \emptyset$
   (a) by $S_1(\beta) \cap \cdots \cap S_n(\beta)$ if $\beta' = \beta$,
   (b) otherwise by
   $$\bigcap_{1\leq j\leq p} (S_1(\beta'_j) \cap \cdots \cap S_n(\beta'_j) \cap \forall\overline{S_i(\overline{\alpha})}\mathit{Ex}_{\mathit{aux}}(\rho[\overline{c_j}/\overline{\alpha}])[\overline{\alpha}/\overline{c_j}])$$
   if $\beta = \forall\overline{\alpha}.\rho$, β'_j ($1 \leq j \leq p$) are all the most general instances of β satisfying the condition, and $\overline{c_j}$ are fresh constants replacing the variables instantiated in the instance β'_j of β.
4. $E_{\langle \mu,n\rangle}(\langle B, \sigma, E\rangle)$ is the triple
   $$\langle \{x : \mathit{Ex}_{\mathit{aux}}(\rho) \mid x : \rho \in B\}, \mathit{Ex}_{\mathit{aux}}(\sigma), \{\mathit{Ex}_{\mathit{aux}}(\rho) \mid \rho \in E\}\rangle$$

Definition 11 (Expansion on types). Given $\langle \mu, n\rangle$ and $\langle B, \sigma, E\rangle$, an *expansion* $E_{\langle \mu,n\rangle}$ on types is defined by
$$E_{\langle \mu,n\rangle}(\tau) = \mathit{Ex}_{\mathit{aux}}(\tau)$$
where $\mathit{Ex}_{\mathit{aux}}$ is defined as in the previous definition.

Some explanations are in order. The result of an operation of expansion on triples, and hence on types, is not unique because it depends on the choice of new variables

in part 2 of the definition; but it is unique modulo renaming of variables (and this is sufficient for our purpose). It is always a type in $\mathcal{T}$: we never introduce an intersection at the right-hand side of an arrow type, and never quantify an intersection type (see part 3). A type might be affected by an expansion even if its free variables are disjoint with those of the subtype to be expanded. The reason is that universally quantified variables represent an infinite set of terms (their instances), so if one instance is affected, the whole type is affected. If we are applying an expansion operation to a universally quantified type, some instances may be expanded (if their last subtypes are in the set of computed types) whereas others are not (if their last subtypes are not in the set of computed types). In this case the expansion of the universally quantified type will be the intersection of the expansions of each class of instances. Since there is only a finite set of computed types, the operation is well defined.

Example 12. Let γ be $(\varphi_1 \to \varphi_2) \to (\varphi_3 \to \varphi_1) \to \varphi_3 \to \varphi_2$, and E be the expansion determined by $\langle \varphi_1, 2 \rangle$ with respect to $\langle \emptyset, \gamma, \emptyset \rangle$. Then
$\mathcal{L}_{\varphi_1}(\langle \emptyset, \gamma, \emptyset \rangle) = \{\varphi_1, \varphi_3 \to \varphi_1, \varphi_3\}$, $\mathcal{V}_{\varphi_1}(\langle \emptyset, \gamma \rangle) = \{\varphi_1, \varphi_3\}$, and (to save space, we write $\overset{j}{i}$ instead of φ_i^j)

$$E(\gamma) = ((\overset{1}{1} \cap \overset{2}{1}) \to \overset{2}{}) \to ((\overset{1}{3} \to \overset{1}{1}) \cap (\overset{2}{3} \to \overset{2}{1})) \to (\overset{1}{3} \cap \overset{2}{3}) \to \overset{2}{}.$$

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

$$E(\gamma) = (\forall\alpha_3.(\overset{1}{1} \to \overset{1}{1}) \to (\alpha_3 \to \overset{1}{1}) \to \alpha_3 \to \overset{1}{1}) \cap \forall\alpha_3.(\overset{2}{1} \to \overset{2}{1}) \to (\alpha_3 \to \overset{2}{1}) \to \alpha_3 \to \overset{2}{1}) \cap$$
$$(\forall\alpha_2 \forall\alpha_3^1 \forall\alpha_3^2.(\overset{1}{1} \cap \overset{2}{1} \to \alpha_2) \to ((\alpha_3^1 \to \overset{1}{1}) \cap (\alpha_3^2 \to \overset{2}{1})) \to \alpha_3^1 \cap \alpha_3^2 \to \alpha_2).$$

For types in $\mathcal{T}$ without sorts and $\forall$, the operation of expansion defined by Ronchi della Rocca and Venneri [34] gives the same results as ours, modulo the relation $\sim$ defined for full intersection types (but the representatives of equivalence classes chosen in [34] are not always types in $\mathcal{T}$).

The operation of *lifting* replaces basis and type by a smaller basis and a larger type, in the sense of $\leq$ (see [2] for details). This operation allows us to eliminate intersections and universal quantifiers, using the $\leq$ relation.

Definition 13. An operation of *lifting* on triples is determined by a pair $L = <\langle B_0, \tau_0 \rangle, \langle B_1, \tau_1 \rangle>$ such that $\tau_0 \leq \tau_1$ and $B_1 \leq B_0$.

$$L(\langle B, \sigma, E \rangle) = \langle B', \sigma', E \rangle$$

where $\sigma' = \tau_1$, if $\sigma = \tau_0$, otherwise $\sigma' = \sigma$; and $B' = B_1$, if $B = B_0$, $B' = B$, otherwise. A lifting on types is determined by a pair $L = \langle \tau_0, \tau_1 \rangle$ such that $\tau_0 \leq \tau_1$ and defined by $L(\sigma) = \sigma'$, where $\sigma' = \tau_1$, if $\sigma = \tau_0$, and $\sigma' = \sigma$, otherwise.

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

Definition 14. A *closure* is an operation characterised by a type-variable $\langle \varphi \rangle$. It is defined by:

$$Cl_\varphi(\langle B, \sigma, E \rangle) = \langle B, \tau, E \rangle$$

where $\tau = \forall\alpha.\sigma[\alpha/\varphi]$, if φ does not appear in B (α is a fresh variable), and $\tau = \sigma$, otherwise. It is extended to types by: $Cl_\varphi(\sigma) = (\tau)$, if $Cl_\varphi(\langle \emptyset, \sigma, \emptyset \rangle) = \langle \emptyset, \tau, \emptyset \rangle$

Definition 15. The *set $\mathcal{C}h$ of chains* for types/triples is defined as the smallest set containing expansions, substitutions, liftings, and closures on types/triples, that is closed under composition. Chains are denoted as $[O_1, \ldots, O_n]$.

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

Definition 16 (Type Assignment Rules).

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

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

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

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

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

   (a) If there exists a chain *Ch* on types such that $\sigma_1 \to \cdots \to \sigma_n \to \sigma = Ch(\mathcal{E}(F))$.
   (b) If φ does not occur (free) in B.
2. We write $B \vdash_{\mathcal{E}} t{:}\sigma$, and say that t is *typeable*, if and only if this judgement is derivable using the above rules.
3. If E is the set of types assigned to function symbols in such a derivation, we represent it by $\langle B, \sigma, E\rangle$.

As said before, the use of an environment in rule ($\mathcal{F}$) introduces a notion of polymorphism for our function symbols, which is an extension (with intersection types and general quantification) of the ML-style of polymorphism. The environment returns the 'principal type' for a function symbol; this symbol can be used with types that are 'instances' of its principal type, obtained by applying chains of operations.

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

Type Assignment for Rewrite Rules Being able to infer a type for a term does not give any guarantee about the typing of the terms in any reduction path out of it. Indeed we need to make sure that the rewrite rules respect the intended functional behaviour for

the function symbols of the signature expressed by the environment. The environment, however, does not express a strict condition on the type of function symbols, leaving room to flexibility by letting us use different consistent instances of the type of a symbol for different occurrences of it. So, we would like to have a certain degree of flexibility also in the use of rewriting rules, without loosing the property of subject reduction, which is essential in type systems. In order to achieve this we define a notion of type assignment on rewrite rules, as done in [6], using the notion of principal triple.

Definition 17. A triple $\langle P, \pi, E\rangle$ is called *a principal triple for t with respect to $\mathcal{E}$*, if the following conditions hold:

(a) $P \vdash_{\mathcal{E}} t{:}\pi$, with associated triple $\langle P, \pi, E\rangle$;
(b) $B \vdash_{\mathcal{E}} t{:}\sigma$, with associated triple $\langle B, \sigma, E'\rangle$, implies that there is a chain on triples *Ch* such that $Ch(\langle P, \pi, E\rangle) = \langle B, \sigma, E'\rangle$.

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

Definition 18. 1. We say that $l \to r \in \mathbf{R}$ with defined symbol F *is typeable with respect to $\mathcal{E}$*, if there are P, π and E such that:
   (a) $\langle P, \pi, E\rangle$ is a principal triple for l with respect to $\mathcal{E}$, and $P \vdash_{\mathcal{E}} r{:}\pi$.
   (b) In $P \vdash_{\mathcal{E}} l{:}\pi$ and $P \vdash_{\mathcal{E}} r{:}\pi$ all occurrences of F are typed with $\mathcal{E}(F)$.
2. We say that a TRS $+ \beta$ *is typeable with respect to $\mathcal{E}$*, if all $\mathbf{r} \in \mathbf{R}$ are.

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

Subject Reduction We will show that reductions preserve types in our system. To obtain this result, we first prove that the operations (substitution, expansion, lifting, and closure) are sound on typeable terms, that is if $\langle B, \sigma, E\rangle$ represents $B \vdash_{\mathcal{E}} t{:}\sigma$, then, for any *Ch*, $Ch(\langle B, \sigma, E\rangle)$ represents a typing derivation for t.

Theorem 19 (Soundness of Operations). *Let Op be a substitution, an expansion, a lifting or a closure. If $B \vdash_{\mathcal{E}} t{:}\sigma$ (with associated triple $\langle B, \sigma, E\rangle$) and $Op(\langle B, \sigma, E\rangle) = \langle B', \sigma', E'\rangle$, then $B' \vdash_{\mathcal{E}} t{:}\sigma'$ (with associated triple $\langle B', \sigma', E'\rangle$).*

Proof. For Substitution and Lifting: by induction on the structure of derivations. For expansion: by induction on the structure of derivations, using the result for lifting in the case of $(\forall I)$. For Closure: Direct by definition of closure, using rule $(\forall I)$.

A direct consequence of the above theorem for Lifting is that the following derivation rule is admissible.

$$\frac{B \vdash_{\mathcal{E}} t{:}\sigma \quad \sigma \leq \tau}{B \vdash_{\mathcal{E}} t{:}\tau}$$

Also, the following is immediate.

Corollary 20. $B \vdash_{\mathcal{E}} t{:}\sigma_1 \cap \cdots \cap \sigma_n$, if and only if, $B \vdash_{\mathcal{E}} t{:}\sigma_i$, for all $1 \leq i \leq n$.

Combining the above results for the different operations, we have:

Theorem 21 (Soundness of Chains). *Let Ch be a chain. Then, if* $B \vdash_{\mathcal{E}} t{:}\sigma$ *(with associated triple* $\langle B, \sigma, E\rangle$*) and* $Ch(\langle B, \sigma, E\rangle) = \langle B', \sigma', E'\rangle$ *then* $B' \vdash_{\mathcal{E}} t{:}\sigma'$ *(with associated triple* $\langle B', \sigma', E'\rangle$*).*

In the proof of Subject Reduction we will use one more lemma:

Lemma 22. *Let* $\mathcal{E}$ *be an environment,* t *a term, and* R *a term-substitution.*

1. *If* $B \vdash_{\mathcal{E}} t{:}\sigma$ *and* B' *is a basis such that* $B' \vdash_{\mathcal{E}} x^{\mathrm{R}}{:}\rho$ *for every statement* $x{:}\rho \in B$*, then* $B' \vdash_{\mathcal{E}} t^{\mathrm{R}}{:}\sigma$*.*
2. *If there are* B *and* σ *such that* $B \vdash_{\mathcal{E}} t^{\mathrm{R}}{:}\sigma$*, then for every* x *occurring in* t *there is a type* ρ_x *such that* $\{x{:}\rho_x \mid x \text{ occurs in } t\} \vdash_{\mathcal{E}} t{:}\sigma$*, and* $B \vdash_{\mathcal{E}} x^{\mathrm{R}}{:}\rho_x$*.*

Theorem 23 (Subject Reduction). *If* $B \vdash_{\mathcal{E}} t{:}\sigma$ *and* $t \to t'$*, then* $B \vdash_{\mathcal{E}} t'{:}\sigma$*.*

Proof. For β-reduction the proof is standard. Let $l \to_{\mathbf{r}} r$ be the typeable rewrite rule applied in the rewrite step $t \to t'$. We will prove that for every term-substitution R and type μ, if $B \vdash_{\mathcal{E}} l^{\mathrm{R}}{:}\mu$, then $B \vdash_{\mathcal{E}} r^{\mathrm{R}}{:}\mu$, which proves the theorem.

Since **r** is typeable, there are P, π and E such that $\langle P, \pi, E\rangle$ is a principal triple for l with respect to $\mathcal{E}$, and $P \vdash_{\mathcal{E}} r{:}\pi$. Suppose R is a term-substitution such that $B \vdash_{\mathcal{E}} l^{\mathrm{R}}{:}\mu$. By Lemma 22(2) there is a B' such that for every $x{:}\rho \in B'$, $B \vdash_{\mathcal{E}} x^{\mathrm{R}}{:}\rho$, and $B' \vdash_{\mathcal{E}} l{:}\mu$, represented by $\langle B', \mu, E'\rangle$. Since $\langle P, \pi, E\rangle$ is a principal triple for l with respect to $\mathcal{E}$, by Definition 17 there is a chain *Ch* such that $Ch(\langle P, \pi, E\rangle) = \langle B', \mu, E'\rangle$. Since $P \vdash_{\mathcal{E}} r{:}\pi$, by Theorem 21 also $B' \vdash_{\mathcal{E}} r{:}\mu$. Then by Lemma 22(1) $B \vdash_{\mathcal{E}} r^{\mathrm{R}}{:}\mu$.

3 Strong Normalisation

Often in type systems types serve not only as partial specifications of the behaviour of terms, but also to ensure that reduction sequences terminate. In fact, this is a well-known property of the intersection system for LC, and of System F. The situation is different in TRS: a rule $t \to t$ may be typeable, although it obviously leads to non-termination. Inspired by the work of Jouannaud and Okada [24], who defined a general scheme of recursion that ensures termination of higher-order rewrite rules combined with LC, we define a general scheme for TRS + β, such that typeability of the rewrite rules in the polymorphic intersection system defined in this paper implies strong normalisation of all typeable terms.

Definition 24 (General Scheme of Recursion). Let $\mathcal{F}_n = \mathcal{Q} \cup \{F^1, \ldots, F^n\}$, where $F^1, \ldots, F^n$ will be the defined symbols, and $\mathcal{Q}$ the set of constructors. We will assume that $F^1, \ldots, F^n$ are defined incrementally (i.e. there is no mutual recursion), by typeable rules that satisfy the *general scheme*:

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

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

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

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

Theorem 25 (Strong Normalisation). *If the rewrite rules satisfy the general schema, any typeable term is strongly normalisable.*

The proof, which we omit for lack of space, can be carried on by using Tait-Girard's method [20] and the techniques devised in [24] in order to cope with some of the difficulties that arise because of the presence of algebraic rewriting.

It is possible to show that if we assume the rules to satisfy the general schema, a typeable TRS $+ \beta$ without critical pairs in $\mathbf{R}$ is locally confluent on typeable terms (we omit also this proof for lack of space), and hence, by Newman's Lemma [31], we can deduce confluence from strong normalisation and local confluence.

4 Restriction to Rank2

In this section, we will present a decidable restriction of the type system as presented above, based on types of rank 2. Although the Rank 2 intersection system and the Rank 2 polymorphic system for LC type exactly the same set of terms [35], their combination results in a system with more expressive power: polymorphism can be expressed directly (using the universal quantifier) and every typeable term has a principal type, as we will show below. The latter property does not hold in a system without intersection.

4.1 Rank 2 type assignment

The polymorphic intersection types of Rank 2, $\mathcal{T}_2$, are a true subset of the set of polymorphic intersection types as defined in Definition 4.

Definition 26. We define polymorphic intersection types of Rank 2 in layers:

$$\mathcal{T}_C ::= \varphi \mid s \mid (\mathcal{T}_C \to \mathcal{T}_C) \qquad \mathcal{T}_I ::= (\mathcal{T}_C^{\forall} \cap \cdots \cap \mathcal{T}_C^{\forall})$$
$$\mathcal{T}_C^{\forall} ::= \mathcal{T}_C \mid (\forall\alpha.\mathcal{T}_C^{\forall}[\alpha/\varphi]) \qquad \mathcal{T}_2 ::= \mathcal{T}_C \mid (\mathcal{T}_I \to \mathcal{T}_2)$$

Below, we will define a unification procedure that will recursively go through types. However, using the sets defined above, not every subtype of a type is a legal type. This will cause a few problems, that will be dealt with.

As for $\mathcal{T}$, we will consider a relation on types, $\leq_2$, that is the restriction to $\mathcal{T}_2$ of the relation $\leq$ defined in Definition 5. For the sake of clarity, we will give the formal definition here.

Definition 27 (Relations on types). On $\mathcal{T}_2$, the pre-order (i.e. reflexive and transitive relation) $\leq_2$ is defined by:

$$\begin{array}{rcll} \sigma_1 \cap \cdots \cap \sigma_n & \leq_2 & \sigma_i, & (1 \leq i \leq n) \\ \forall 1 \leq i \leq n . \sigma \leq_2 \sigma_i & \Rightarrow & \sigma \leq_2 \sigma_1 \cap \cdots \cap \sigma_n & (n \geq 1) \\ \rho \leq_2 \sigma, \tau \leq_2 \mu & \Rightarrow & \sigma \to \tau \leq_2 \rho \to \mu, & (\tau, \mu \in \mathcal{T}_2) \\ \forall \alpha . \sigma[\alpha/\varphi] & \leq_2 & \sigma[\tau/\varphi], & (\tau \in \mathcal{T}_C) \\ \sigma & \leq_2 & \forall \alpha . \sigma[\alpha/\varphi], & (\varphi \text{ not in } \sigma, \text{ and } \alpha \text{ fresh}), \\ \sigma \leq_2 \tau & \Rightarrow & \forall \alpha . \sigma[\alpha/\varphi] \leq_2 \forall \alpha . \tau[\alpha/\varphi]. & \end{array}$$

The equivalence relation $\sim_2$ is defined by: $\sigma \sim_2 \tau \iff \sigma \leq_2 \tau \leq_2 \sigma$, and we extend $\leq_2$ to bases in the same way as done for $\leq$.

Notice that part '$\forall \alpha . \sigma \to \tau \leq_2 \sigma \to \forall \alpha . \tau$, *if* α *does not occur in* σ' is omitted, since $\sigma \to \forall \alpha . \tau$ is not a type of Rank 2.

The Rank 2 versions for the various operations 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. Notice that, because the operation of expansion will need to be defined in a different way than before, now we can use a pair basis-type (and no more a triple), as an abstraction for a derivation. The first three operations used for the Rank 2 system are straightforward variants of operations defined for the full system.

Definition 28.

1. *Substitution* $(\varphi \mapsto \rho) : \mathcal{T}_2 \to \mathcal{T}_2$ is defined as in Definition 8, but with the restriction that $\rho \in \mathcal{T}_C$. We use Id_S for the substitution that replaces all variables by themselves, and write $\mathcal{S}$ for the set of all substitutions.
   For the sake of clarity, and in order to avoid writing $[S_1, \ldots, S_n]$ for a chain of single type-variable substitutions, we will close the set of substitutions under composition '$\circ$'.
2. *Lifting* is defined as in Definition 13, but with the restriction that $\leq$ is taken to be $\leq_2$ of Definition 27.
3. *Closure* is defined as a pair of types $\langle \sigma, \varphi \rangle$, with $\sigma \in \mathcal{T}_C^{\forall}$, by:
   $$\langle \sigma, \varphi \rangle \left(\langle B, \tau_1 \cap \cdots \cap \tau_n \rangle \right) = \langle B, \tau_1' \cap \cdots \cap \tau_n' \rangle$$
   where, for all $1 \leq i \leq n$, $\tau_i' = \forall \alpha . \sigma[\alpha/\varphi]$, if $\tau_i = \sigma$ and φ does not appear in B (α fresh), and $\tau_i' = \tau_i$, otherwise.

Below, we will also use $\langle \sigma, \overline{\alpha} \rangle$ for a closure, with the intention that $\langle \sigma[\overline{\alpha}/\overline{\varphi}], \overline{\alpha} \rangle = \langle \sigma, \overline{\varphi} \rangle$, and $\langle \sigma, \overline{\varphi} \rangle = [Cl_1, \ldots, Cl_n]$ where, for $0 \leq i \leq n$, $\sigma_{i+1} = Cl_{i+1}(\sigma_i)$ and $Cl_{i+1} = \langle \sigma_i, \varphi_{i+1} \rangle$, and $\sigma_0 = \sigma$.

The variant of expansion as used in the Rank 2 system is quite different from that of Definition 10. The reason for this is that expansion, normally, increases the rank of a type:

$$\langle \varphi_1, 2 \rangle (\langle \{x{:}\varphi_1 \to \varphi_2\}, \varphi_1, \emptyset \rangle)(\varphi_1 \to \varphi_2) = (\varphi_1^1 \cap \varphi_1^2) \to \varphi_2,$$

a feature that is of course not allowed within a system that limits the rank of types. Since below expansion is only used in very precise situations (within the procedure *unify*$_2^\forall$, and in the proof of Theorem 40), the solution is relatively easy: in the context of Rank 2 types, expansion is only called on types in $\mathcal{T}_C^\forall$, so it is defined to work well there, by replacing *all* types by an intersection; in particular, intersections are not created at the left of an arrow.

Definition 29. Let B be a Rank 2 basis, $\sigma \in \mathcal{T}_2$, and $n \geq 1$. The n-fold *Rank 2 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 $V = \{\varphi_1, \ldots, \varphi_m\}$ is the set of all (free) variables occurring in $\langle B, \sigma\rangle$. Choose $m \times n$ different variables $\varphi_1^1, \ldots, \varphi_1^n, \ldots, \varphi_m^1, \ldots, \varphi_m^n$, such that each φ_j^i ($1 \leq i \leq n$, $1 \leq j \leq m$) does not occur in V. Let S_i be the substitution that replaces every φ_j by φ_j^i. Then Rank 2 expansion is defined on types, bases, and pairs, respectively, by:

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

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, since each $S_i(\tau) \in \mathcal{T}_2$, for $1 \leq i \leq n$, for the sake of clarity, we will not treat it separately.

Notice that we have no need for the third parameter 'E' in this notion of expansion.

Since all results in this section regard the Rank 2 system, we will use 'expansion' rather than 'Rank 2 expansion.'

As before, operations will be grouped in chains.

Definition 30. A *Rank 2 chain* (or R2-chain for short) is a chain *Ch* of operations, composed of at most one expansion, at most one substitution, at most one lifting, and a number (≥ 0) of closures: $Ch = [E, S, L, Cl_1, \ldots Cl_m] = [E, S, L, \overline{Cl}]$.

Lemma 31. *Let Ch be an R2-chain.*

1. *If $\sigma \in \mathcal{T}_C$, and $Ch(\sigma) \in \mathcal{T}_2$, then $Ch(\sigma) \in \mathcal{T}_C$, and there is a substitution S such that $Ch(\sigma) = S(\sigma)$.*
2. *If $\sigma \in \mathcal{T}_2$, and $Ch(\sigma) \in \mathcal{T}_C^\forall$, then there are a substitution S, and closures $Cl_1, \ldots, Cl_n$, such that $Ch(\sigma) = [S, Cl_1, \ldots, Cl_n](\sigma)$.*
3. *If $\sigma \in \mathcal{T}_C$, and $Ch(\sigma) \in \mathcal{T}_l$, then there exists a lifting-free R2-chain Ch' such that $Ch(\sigma) = Ch'(\sigma)$.*
4. *If $\sigma \in \mathcal{T}_2$, and $Ch(\sigma) \in \mathcal{T}_C$, then there is a substitution S such that $Ch(\sigma) = S(\sigma)$.*
5. *If $\sigma \in \mathcal{T}_2$, and $Ch(\sigma) \in \mathcal{T}_2$, then there are substitution S, and lifting L such that $Ch(\sigma) = [S, L](\sigma)$.*

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

Definition 32. 1. A *Rank 2 environment* $\mathcal{E}$ is a mapping from $\mathcal{F}$ to $\mathcal{T}_2$.
2. *Rank 2 type assignment* is defined by the following natural deduction system:

$$(\leq_2)\text{:}\ \frac{x{:}\sigma\ \in\ B \quad \sigma \leq_2 \tau}{B \vdash^2_{\mathcal{E}} x{:}\tau}\ \text{(a)} \qquad (\mathcal{F})\text{:}\ \frac{B \vdash^2_{\mathcal{E}} t_1{:}\sigma_1 \quad \ldots \quad B \vdash^2_{\mathcal{E}} t_n{:}\sigma_n}{B \vdash^2_{\mathcal{E}} F(t_1,\ldots,t_n){:}\sigma}\ \text{(b)}$$

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

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

(a) $\sigma\ \in\ \mathcal{T}_I$, and $\tau\ \in\ \mathcal{T}_C$.
(b) If there exists a chain *Ch* such that $\sigma_1\to\cdots\to\sigma_n\to\sigma = Ch(\mathcal{E}(F))$.
(c) If φ does not occur in B, and $\sigma\ \in\ T^{\forall}_C$.
(d) If $n \geq 1$, and $\sigma_i\ \in\ \mathcal{T}^{\forall}_C$, for every $1 \leq i \leq n$.

Notice that, since quantification elimination is implicit in rule $(\leq_2)$ when restricting the use of the quantifier to the left of arrows only, there is no longer need for a general $(\forall E)$ rule; as rule $(\cap E)$, its use is in a strict system limited to variables, and there its actions are already performed by $(\leq_2)$.

We have the following soundness result for chains of operations:

Lemma 33. *If* $\sigma\ \in\ \mathcal{T}_I$, $B \vdash^2_{\mathcal{E}} t{:}\sigma$, *and* $Ch(\langle B,\sigma\rangle) = \langle B',\sigma'\rangle$, *then* $B' \vdash^2_{\mathcal{E}} t{:}\sigma'$.

4.2 Unification of Rank 2 Types

In the context of types, unification is a procedure normally used to find a common instance for demanded and provided type for applications, i.e: if t_1 has type $\sigma\to\tau$, and t_2 has type ρ, then unification looks for a common instance of the types σ and ρ such that $Ap(t_1,t_2)$ can be typed properly. The unification algorithm $unify^{\forall}_2$ presented in this section 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* [33].

Definition 34. Unification of Curry types (extended with bound variables and type constants), $unify : \mathcal{T}'_C \times \mathcal{T}'_C \to \mathcal{S}$, is defined by:

$$unify(\varphi,\tau) = unify(\tau,\varphi) = (\varphi\mapsto\tau),\ \text{if } \varphi \text{ does not occur in } \tau \text{ or } \varphi = \tau$$
$$unify(\alpha,\alpha) = unify(s,s) = Id_S,$$
$$unify(\sigma\to\tau,\rho\to\mu) = S_2\circ S_1,\ \text{where } S_1 = unify(\sigma,\rho), S_2 = unify(S_1(\tau), S_1(\mu)).$$

(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.

Property 35. ([33]) 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 $S_u = \mathit{unify}(\sigma, \tau)$, 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 $Ap(t_1, t_2)$. Suppose the algorithm has derived $P_1 \vdash_{\mathcal{E}}^2 t_1{:}\pi_1$ and $P_2 \vdash_{\mathcal{E}}^2 t_2{:}\pi_2$ as principal derivations for t_1 and t_2, respectively, and that $\pi_1 = \sigma \to \tau$. Thus the demanded type σ is in $\mathcal{T}_I$ 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 $Ch(\pi_2) \in \mathcal{T}_I$. However, if $\pi_2 \notin \mathcal{T}_C$, then in general $Ch(\pi_2) \notin \mathcal{T}_I$. To overcome this difficulty an algorithm $\mathit{to}\mathcal{T}_C$ will be inserted that, when applied to the type ρ, returns a chain of operations that removes, if possible, intersections in ρ. Note that if quantifiers appear in ρ, $\mathit{to}\mathcal{T}_C(\rho)$ should fail, since quantifiers that appear before an arrow cannot be removed by any of the operations on types. Finally, $\mathit{unify}_2^{\forall}(\sigma, S_2(\pi_2), S_2(P_2))$ is called (with $S_2 = \mathit{to}\mathcal{T}_C(\pi_2)$). 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 36. 1. The function $\mathit{to}\mathcal{T}_C : \mathcal{T}_2 \to \mathcal{S}$ is defined by:

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

where $S_i = \mathit{unify}(S_{i-1}(\sigma_1), S_{i-1}(\sigma_{i+1})) \circ S_{i-1}$, $(1 \leq i \leq n-1$, with $S_0 = Id_S)$, and $S' = \mathit{to}\mathcal{T}_C(S_n(\mu))$.

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

$$\begin{array}{ll} \mathit{unify}_2^{\forall}(\sigma, \tau, B) & = \mathit{unify}(\sigma, \tau), \text{ if } \sigma \in \mathcal{T}_C \\ \mathit{unify}_2^{\forall}((\forall\overline{\alpha_1}.\sigma_1) \cap \ldots \cap (\forall\overline{\alpha_n}.\sigma_n), \tau, B) & = [E, S_n], \qquad \text{otherwise} \end{array}$$

where $E = n_{\langle B, \tau \rangle}$, $\tau_1 \cap \cdots \cap \tau_n = E(\tau)$, and $S_i = \mathit{unify}(S_{i-1}(\sigma_i), \tau_i) \circ S_{i-1}$ (with $S_0 = Id_S$), provided S_{i-1} does not introduce αs when applied to σ_i , for $1 \leq i \leq n$.

Notice that *unify*, $\mathit{to}\mathcal{T}_C$, and $\mathit{unify}_2^{\forall}$ all return lifting-free R2-chains. Moreover, both *unify* and $\mathit{to}\mathcal{T}_C$ return a substitution, and the R2-chain returned by $\mathit{unify}_2^{\forall}(\sigma, \tau)$ acts on σ as a substitution: the expansion in the chain is defined for the sake of τ only. Notice also that $\mathit{unify}_2^{\forall}$ does not really return a unifying chain for its first two arguments; to achieve this, also closures would have to be inserted. They are not needed for the present purpose.

The procedure $\mathit{unify}_2^{\forall}$ fails when *unify* fails, and $\mathit{to}\mathcal{T}_C$ fails when either *unify* fails or when the argument contains $\forall$. Because of this relation between $\mathit{unify}_2^{\forall}$ and $\mathit{to}\mathcal{T}_C$ on one side, and *unify* on the other, the procedures defined here are terminating and type assignment in the system defined in this section is decidable.

Lemma 37. *Let Ch be an R2-chain.*

1. *If $\sigma \in \mathcal{T}_2$, and $Ch(\sigma) = \tau \in \mathcal{T}_C$, then there is a S' such that $S' \circ \mathit{to}\mathcal{T}_C(\sigma)(\sigma) = \tau$.*
2. *If $\sigma \in \mathcal{T}_2$, and $Ch(\sigma) = \tau \in \mathcal{T}_I$, then $[\mathit{to}\mathcal{T}_C(\sigma)] * Ch'(\sigma) = \tau$, for some R2-chain Ch'.*

Proof. Easy, using Property 35, and Lemma 31.3-4. ■

4.3 Principal pairs for terms

In this subsection, 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 Theorem 40 it will be shown that, for every term, this is indeed the principal one.

Notice that, in the definition below, if $pp_{\mathcal{E}}(t) = \langle P, \pi \rangle$, then $\pi \in \mathcal{T}_2$. For example, the principal pair for the term $\lambda x.x$ is $\langle \emptyset, \varphi \to \varphi \rangle$, 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.

Definition 38. Let t be a term in $T(\mathcal{F}, \mathcal{X})$. Using $unify_2^{\forall}$, $pp_{\mathcal{E}}(t)$ is defined by:

1. $t \equiv x$. Then $pp_{\mathcal{E}}(x) = \langle \{x{:}\varphi\}, \varphi \rangle$.
2. $t \equiv \lambda x.t'$. Let $pp_{\mathcal{E}}(t') = \langle P, \pi \rangle$, then:
   (a) If x occurs free in t', and $x{:}\sigma \in P$, then $pp_{\mathcal{E}}(\lambda x.t') = \langle P \backslash x, \sigma \to \pi \rangle$.
   (b) Otherwise, let φ be a fresh variable, and $pp_{\mathcal{E}}(\lambda x.t') = \langle P, \varphi \to \pi \rangle$.
3. $t \equiv Ap(t_1, t_2)$. Let $pp_{\mathcal{E}}(t_1) = \langle P_1, \pi_1 \rangle$, $pp_{\mathcal{E}}(t_2) = \langle P_2, \pi_2 \rangle$ (choose, if necessary, trivial variants such that these pairs are disjoint), and $S_2 = to\mathcal{T}_C(\pi_2)$, then
   $\pi_1 \in \mathcal{T}_C$ **:** $pp_{\mathcal{E}}(Ap(t_1, t_2)) = \langle P, \pi \rangle$, where $P = S_1(\Pi\{P_1, S_2(P_2)\})$, $\pi = S_1(\varphi)$, $S_1 = unify(\pi_1, S_2(\pi_2) \to \varphi)$, and φ is a fresh variable.
   $\pi_1 \notin \mathcal{T}_C$ **:** Assume $\pi_1 = \sigma \to \tau$. Then $pp_{\mathcal{E}}(Ap(t_1, t_2)) = \langle P, \pi \rangle$, provided P and π contain no unbound occurrences of αs, where $P = S(\Pi\{P_1, E(S_2(P_2))\})$, $\pi = S(\tau)$, and $[E, S] = unify_2^{\forall}(\sigma, S_2(\pi_2), S_2(P_2))$.
4. $t \equiv F(t_1, \ldots, t_n)$. Let, for every $1 \leq i \leq n$, $pp_{\mathcal{E}}(t_i) = \langle P_i, \pi_i \rangle$ (assume that the $\langle P_i, \pi_i \rangle$ are pairwise disjoint), then $pp_{\mathcal{E}}(F(t_1, \ldots, t_n)) = \langle P, \pi \rangle$, provided P and π contain no unbound occurrences of αs, where

$$
\begin{aligned}
P &= S^n \circ \cdots \circ S^1(\Pi\{E_1(S_1(P_1)), \ldots, E_n(S_n(P_n))\}),\\
\pi &= S^n \circ \cdots \circ S^1(\gamma),\\
\gamma_1 \to \cdots \to \gamma_n \to \gamma &\text{ is a fresh instance of } \mathcal{E}(F), \text{ and}\\
\text{for every } 1 \leq i \leq n,\ S_i &= to\mathcal{T}_C(\pi_i), \text{ and}\\
[E_i, S^i] &= unify_2^{\forall}(S^{i-1} \circ \cdots \circ S^1(\gamma_i), S_i(\pi_i), S_i(P_i)).
\end{aligned}
$$

Since *unify* or $unify_2^{\forall}$ may fail, not every term has a principal pair.

Notice that closures are not needed when calculating the new basis and type.

The treatment of $F(t_1, \ldots, t_n)$ in Definition 38 is in fact very much the same as a repeated treatment of $Ap(t_1, t_2)$. This can be understood by observing that the terms

$$F(t_1, \ldots, t_n) \text{ and } Ap(\cdots Ap(\lambda x_1 \cdots x_n.F(x_1, \ldots, x_n), t_1) \cdots, t_n)$$

should be treated in the same way.

The following lemma is needed in the proof of Theorem 40.2. It states that if a chain maps the principal pairs of terms t_1, t_2 in an application $Ap(t_1, t_2)$ to pairs that allow the application itself to be typed, then these pairs can also be obtained by first performing a unification.

Lemma 39. *Let $\sigma \in \mathcal{T}_2$, and for $i = 1, 2$, $pp_{\mathcal{E}}(t_i) = \langle P_i, \pi_i \rangle$, such that these pairs are disjoint. Let Ch_1, Ch_2 be R2-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 R2-chains Ch_u and Ch_p, and type $\rho \in \mathcal{T}_2$ such that*

$$pp_{\mathcal{E}}(Ap(t_1, t_2)) = Ch_u(\langle \Pi\{P_1, P_2\}, \rho \rangle), \text{ and}$$
$$Ch_p(pp_{\mathcal{E}}(Ap(t_1, t_2))) = \langle \Pi\{B_1, B_2\}, \tau \rangle.$$

Proof. Using Property 35 and Lemmas 31 and 37. ■

A similar result holds for terms of the shape $F(t_1, \ldots, t_n)$.

The main result of this section then becomes:

Theorem 40. *1.* Soundness of $pp_{\mathcal{E}}$. *If $pp_{\mathcal{E}}(t) = \langle P, \pi \rangle$, then $P \vdash^2_{\mathcal{E}} t{:}\pi$.*

2. Completeness of $pp_{\mathcal{E}}$. *If $B \vdash^2_{\mathcal{E}} t{:}\sigma$, then there are a basis P and type π such that $pp_{\mathcal{E}}(t) = \langle P, \pi \rangle$, and there is an R2-chain Ch such that $Ch(\langle P, \pi \rangle) = \langle B, \sigma \rangle$.*

Proof. By induction on the structure of derivations. Lemma 39 is used for $(\to E)$. ■

Acknowledgements The second author would like to thank Mariangiola Dezani and Carmelo Dell'Acqua for all they taught him in different fields.

References

1. S. van Bakel. Complete restrictions of the Intersection Type Discipline. *TCS*, 102(1):135–163, 1992.
2. S. van Bakel. Principal type schemes for the Strict Type Assignment System. *Logic and Computation*, 3(6):643–670, 1993.
3. S. van Bakel. Intersection Type Assignment Systems. *TCS*, 151(2):385–435, 1995.
4. S. van Bakel. Rank 2 Intersection Type Assignment in Term Rewriting Systems. *Fundamenta Informaticae*, 2(26):141–166, 1996.
5. S. van Bakel, F. Barbanera, and M. Fernández. Rewrite Systems with Abstraction and β-rule: Types, Approximants and Normalization. In *ESOP'96*, *LNCS* 1058, pages 387–403. Springer-Verlag, 1996.
6. S. van Bakel and M. Fernández. Normalization Results for Typeable Rewrite Systems. *I&C* 133(2):73–116, 1997.
7. F. Barbanera and M. Fernández. Intersection Type Assignment Systems with Higher-Order Algebraic Rewriting. *TCS* 170:173–207, 1996.
8. F. Barbanera, M. Fernández, and H. Geuvers. Modularity of Strong Normalization and Confluence in the Algebraic λ-cube. In *LICS '94*, 1994.
9. H. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, Amsterdam, revised edition, 1984.
10. H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. of Symbolic Logic*, 48(4):931–940, 1983.
11. V. Breazu-Tannen. Combining algebra and higher-order types. In *LICS '88*, 1988.
12. V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization. *TCS* 83(1):3–28, 1991.
13. V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic confluence. *I&C* 82:3–28, 1992.

14. A. Bucciarelli, S. De Lorenzis, A. Piperno, I. Salvo. Some Computational Properties of Intersection Types (Extended Abstract). In *LICS '99*, pages 109–118, 1999.
15. L.M.M. Damas. *Type Assignment in Programming Languages*. PhD Thesis, University of Edinburgh, 1984.
16. 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.
17. D. J. Dougherty. Adding algebraic rewriting to the untyped lambda calculus. Information and Computation, 101, 251-267 (1992).
18. M.Fernández. Type Assignment and Termination of Interaction Nets. *Mathematical Structures in Computer Science*, 8:593–636, 1998.
19. M. Fernández and J.P. Jouannaud. Modular termination of term rewriting systems revisited. In *LNCS* 906, pages 255–272. Springer-Verlag, 1994.
20. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.
21. J.Y. Girard. The System F of Variable Types, Fifteen years later. *TCS* 45:159–192, 1986.
22. B. Jacobs, I. Margaria, and M. Zacchi. Filter models with polymorphic types. *TCS* 95:143–158, 1992.
23. T. Jim. What are principal typings and what are they good for? In *Proc. POPL'96, ACM Symposium on Principles of Programming Language*, 1996.
24. J.P. Jouannaud and M. Okada. Executable higher-order algebraic specification languages. In LICS '91, pages 350–361, 1991.
25. A.J. Kfoury and J. Tiuryn. Type reconstruction in finite-rank fragments of the second-order λ-calculus. *I&C* 98(2):228–257, 1992.
26. A.J. Kfoury and J.B. Wells. A Direct Algorithm for Type Inference in the Rank-2 Fragment of the Second-Order λ-Calculus. In *LFP'94*, 1994.
27. A.J. Kfoury and J.B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Proc. 26th ACM Symp. Principles of Prog. Languages*, POPL '99, pages 161–174, 1999.
28. 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.
29. J. Launchbury and S.L. Peyton Jones. Lazy functional state threads. In *Proc. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, 1994.
30. I. Margaria and M. Zacchi. Principal Typing in a $\forall\cap$-Discipline. *Logic and Computation*, 5(3):367–381, 1995.
31. M.H.A. Newman. On theories with a combinatorial definition of 'equivalence'. *Ann. Math.*, 43:223–243, 1942.
32. E.G.J.M.H. Nöcker, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J. Plasmeijer. Concurrent Clean. In *PARLE '91*, *LNCS* 506-II, pages 202–219. Springer-Verlag, 1991.
33. J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
34. S. Ronchi Della Rocca and B. Venneri. Principal type schemes for an extended type theory. *TCS* 28:151–169, 1984.
35. H. Yokohuchi. Embedding a Second-Order Type System into an Intersection Type System. *I&C* 117:206–220, 1995.