A complete type inference algorithm for simple intersection types [1]

M. Coppo - P. Giannini
Dipartimento di Informatica
Universita' di Torino
C. Svizzera 185- 10100 TORINO
ITALY

Abstract
In this paper we present a decidable restriction of the intersection type discipline, obtained by combining intersection and universal quantification over types. The system, which has a notion of principal type, is a proper extension of the ML type system. A sound and complete type checking algorithm is presented and proved correct.

1. Introduction

One of the most interesting notions of type constraint for functional programming languages is the one derived from Curry's *Functionality Theory* [Curry, Feys, 1958], which suggested the type disciplines incorporated in some programming languages of recent design, notably ML [Gordon, Milner, Wadsworth, 1979] and Miranda [Turner, 1985]. In this approach types are assigned to terms of the λ–calculus according to a set of formal rules which can be effectively checked at compile time. Types describe the functional behaviour of terms in such a way that, in general, the same term can be assigned infinitely many types depending on the particular program context in which it occurs. This causes a natural notion of *implicit* polymorphism to be introduced, where *type schemes* are assigned to terms and they may be instantiated to match the types required by the surrounding program context. Several features make this sort of polymorphism particularly attractive both from the practical and the theoretical point of view. The type inference algorithm is complete, due to the existence of *principal type schemes*, [Hindley, 1969] and [Milner, 1978] which fully characterize the set of types assignable to each

[1]Work partially supported by MPI 60% funding.

term. Moreover natural properties of typed terms can be easily proven like the property of strong normalization and the fact that types are preserved by reduction. An important consequence of this property is, for example, that terms having a type cannot produce run time errors during their evaluation.

In spite of that, considerable effort has been done recently to find stronger type systems that, while retaining the basic good properties of the ML type system (decidability, completeness, strong normalization, etc..) could overcome its weakness. In ML, in fact, not all meaningful terms have a proper type, even when they are used in a consistent way. Take for instance the operators **twice**=λf.λx.f(f x) and **K**=λx.λy.x. The type given to **twice** by the ML type checker is such that the term (**twice K**) has no type whreas (**twice K**) β-reduces to λx.λu.λv.x which is well typed. In ML, however, we can assign a type to the term **let** f=K **in** λx.f(f x). Here the body of **twice** must be typed, in the **let** construct, using explicitly the fact that f is **K**. That means that any other use of **twice** requires a different type checking. This is contrary to the functional style of programming in which a function is typechecked once and for all, and all its uses require only its type specification.

Several extensions of the ML type system have been proposed. One of the most interesting is characterized by the introduction (in a type inference context) of the notion of (second-order) universal quantification over types taken from Girard's system F [Girard 1972]. It is not known, however, whether typability in the second-order type assignment system is decidable . Moreover, no notion of principal type seems to exist in this system

A number of restrictions of the second-order type assignment system have been studied. The restriction to rank two has been proved to be decidable [Kfoury, Tyurin, 1990], but it turns out to be essentially equivalent to the ML type system (with the **let** operator). At rank three, instead, the second-order type assignment system is undecidable [Kfoury, Tyurin, 1990]. Another approach to overcome the intrinsic complexity of the second-order type inference is taken in [Giannini, Ronchi, 1991] where a complete stratification of the second-order type discipline is introduced. Such stratification is decidable. In this system, however, type inference is not compositional. The typing properties of a term are represented by a set of equations that can have many uncomparable solutions.

Also related to the second-order type inference system is the recursively polymorphic extension of ML [Mycroft, 1984], which has been proven to be undecidable [Kfoury, et al., 1989]. In this system, moreover, the extension of ML is limited to the use of recursion over polymorphic functions, which is not so usual in the functional programming practice.

In this paper we introduce a type assignment system based on a combination of the notions of rank two polymorphism and intersection . Intersection types, which have been introduced in [Coppo, Dezani, 1980] (see also [Barendregt, Coppo, Dezani, 1983]), allow to assign types to all strongly normalizable terms . As a consequence of this, however, the typechecking in systems with intersection types is undecidable, even if a notion of principal type exists [Ronchi, Venneri, 1984]. Indeed, intersection types in their full generality are more suitable to study semantic properties of λ-terms. This is also supported by the fact that it is possible to define models in which the interpretation of a term is given by the set of its types [Barendregt, Coppo, Dezani, 1983].

In this paper the notion of intersection is used in a limited way, to allow a term variable to have different types in its different occurrences. When a variable is abstracted, instead of unify all the types of its uses, as in ML, we take the intersection of all such types (represented by a finite set of types). In this way we do not commit to a specific representative for all the types. Intersections can be generated also by instantiation of a (universally quantified) type scheme, in the style of the ML type system. So we get an inference system which is substantially stronger than the ML type system (all pure normal form, for instance, have a type). We prove that a notion of principal type exists for such a system and give a complete type inference algorithm. Owing to the finite nature of intersection, it turns out that a term has, in general, a finite set of principal types and not only one as in ML. Each of such types, however, is "more general" than the ML principal type. Terms having more than one principal type, however, are rather pathological (we will show one in section 4). Indeed all terms that arise in the programming practice have only one principal type.

The type assignment system with its properties is introduced in section 2. In section 3 we introduce another system with labelled quantifiers which represents a sort of meta-system in which the type checking properties will be studied. Finally the type checking algorithm is presented in section 4.

2 The type assignment system and its properties

We introduce the type assignment system $\vdash_\wedge$ we will be dealing with. As remarked in the introduction our system is a variant of the intersection type system of [Coppo, et al., 1980]. As in [Coppo, et al., 1980] we avoid the explicit use of intersection by denoting with $\{\alpha_1;...;\alpha_n\}$ the intersection $\alpha_1\wedge...\wedge\alpha_n$ (in the notation of [Barendregt, Coppo, Dezani, 1983]). Moreover we consider also type schemes (in the sense of [Damas, Milner, 1982]) that can generate intersections by (multiple) instantiations.

Therefore, in describing the syntax of types we use three categories of objects : the *types*, the *intersections of types*, and the *schemes*. These categories are kept separate to emphasize their different role in the type assignment system. Types are the ultimate result of our type assignment system. Indeed they express the functionality of the terms we want to characterize. Intersections of types allow to delay the commitment (unification) to a specific type in the multiple use of variables (see rules (VAR) and ($\rightarrow$I)). So they appear in the left-hand-side of an arrow constructor (we do not allow intersections in the right-hand-side of an arrow). Finally schemes are descriptions of (infinite) sets of types. They express, through the use of universal quantification over generic type variables, the polymorphism intrinsic in terms (see rule ($\forall$I) and ($\forall$E)). The fact that schemes are not components of types emphasize their metatheoretical nature. (This is also the view taken in [Damas, Milner, 1982].)

We will study type assignment for a standard core language which consists of the basic λ–calculus with constants, whose terms are defined from a a set of basic constants and a set of terms variables by the operations of application and abstraction.

Definition 2.1. (i) The sets T of *types* and Q of *intersections of types* are defined from a set K of basic types and a set V of type variables by:

- $V, K \subseteq T$
- $\{\alpha_1;...;\alpha_n\} \in Q$ whenever $\alpha_1,...,\alpha_n \in T$. ($\{\} \in Q$ is the empty intersection)
- $\xi \to \beta \in T$ whenever $\xi \in Q$, $\beta \in T$.

(ii) The set S of *schemes* is defined by

- $\forall \underline{t}.\alpha \in S$ whenever $\alpha \in T$ and $\underline{t}$ is a (possibly empty) sequence of type variables. □

α, β, γ, and δ range over types, ξ, χ over intersections and σ, τ over schemes. Variables are ranged over by t, u, v ... $\underline{t}$, $\underline{u}$, ... denote finite sequences of type variables, and $\underline{\alpha}$, $\underline{\beta}$, $\underline{\gamma}$, etc. denote sequences of types. ε denotes the empty sequence.

When explicitly mentioned α can also be used to denote objects that belong to more than one syntactic category.

Note that, for technical reasons, schemes are obtained from types by the use of a unique universal quantification over a sequence of variables. Intersections and sequences of variables may be seen, respectively, as finite sets of type or type variables and are considered equal modulo permutations and repetitions of the same element.

A substitution is a function $s{:}V \to T$. For substitutions we use the standard notations for finite functions. $[t_1{:=}\alpha_1, \ldots, t_n{:=}\alpha_n]$ $(n \geq 0)$ denotes the substitution assigning α_1 to t_1, ..., α_n to t_n. $s[t_1{:=}\alpha_1, \ldots, t_n{:=}\alpha_n]$ is the function that coincides with s on all the variables except t_1, ..., t_n, and is such that $s(t_i)=\alpha_i$ $(1 \leq i \leq n)$. Substitutions can be extended to $T \cup Q \cup S$ in the standard way. We use the same notatin to denote explicit substitution. So $\alpha[t{:=}\beta]$ is the same as $[t{:=}\beta](\alpha)$.

Types, intersections and schemes are closed under substitution, i.e. if α is a type (intersection, scheme) $s(\alpha)$ is a type (intersection, scheme).

The type assignment system $\vdash_\wedge$ is given as a set of formal rules in natural deduction.

Definition 2.2. (i) A basis B is a set of statements $x{:}\alpha$ where α is a type. Define $B/x = B - \{x{:}\alpha \mid x{:}\alpha \in B\}$.

(ii) A type assignment statement is an expression $B \vdash_\wedge M{:}\alpha$ where B is a basis and α is a type, an intersection or a scheme. The rules for type assignment are the following.

(VAR) $\{x{:}\alpha\} \vdash_\wedge x{:}\alpha$

$$(\to I)\ \frac{B/x \cup \{x{:}\alpha_1,...,x{:}\alpha_n\} \vdash_\wedge M{:}\beta}{B/x \vdash_\wedge \lambda x.M{:}\{\alpha_1;...;\alpha_n\} \to \beta}$$

$$(\to E)\ \frac{B_1 \vdash_\wedge M{:}\xi \to \beta \quad B_2 \vdash_\wedge N{:}\xi}{B_1 \cup B_2 \vdash_\wedge (MN){:}\beta}$$

$$(\forall I)\ \frac{B \vdash_\wedge M{:}\alpha}{B \vdash_\wedge M{:}\forall \underline{t}.\alpha}$$ (the variables in $\underline{t}$ are not free for M in B)

$$(\forall E)\ \frac{B\vdash_{\wedge} M:\forall\underline{t}\,\alpha}{B\vdash_{\wedge} M:\{\alpha[\underline{t}:=\underline{\beta}_1];\ldots;\alpha[\underline{t}:=\underline{\beta}_n]\}}\ (n\geq 0)\ \ (\text{where } \underline{\beta}_1,\ldots,\underline{\beta}_n\in T^*)\qquad\square$$

It is immediate to see that deductions are closed under substitution, i.e. if $B\vdash_{\wedge}M:\alpha$, then $s(B)\vdash_{\wedge}M:s(\alpha)$, where α is a type, an intersection or a schema.

Remark 2.3. (i) If $B\vdash_{\wedge}M:\alpha$ then B contains assumptions on all and only the free variables of M, and for each variable x there are at most as many different assumptions as occurrences of x in M. In the conclusion of ($\rightarrow$I) if x is not a free variable of M we have $B\vdash_{\wedge}\lambda x.M:\{\}\rightarrow\beta$.
(ii) The only way we can derive an intersection for M is by an application of ($\forall$E). Therefore only intersections whose elements are instances of a same type are derivable for a term M.
(iii) From $B\vdash_{\wedge}M:\alpha$ we can always derive, by ($\forall$I), $B\vdash_{\wedge}M:\forall\varepsilon.\alpha$ (where ε is the empty string) and, by ($\forall$E), then $B\vdash_{\wedge}M:\{\alpha\}$. So the following rule is derivable.

$$(\text{INT})\ \frac{B\vdash_{\wedge}M:\alpha}{B\vdash_{\wedge}M:\{\alpha\}}$$

This rule is useful since the minor premise of ($\rightarrow$E) must be an intersection.
(iv) An instance of ($\forall$I) followed by an instance of ($\forall$E) with n=0 produces the following derived rule

$$(\{\}\text{-I})\ \frac{B\vdash_{\wedge}M:\alpha}{B\vdash_{\wedge}M:\{\}}$$

That is, we can derive the empty intersection for M if and only if we can derive a type for M. Note the difference between $\{\}$ and ω of [Coppo, et. al., 1980]. In that case ω is always derivable for a term. $\square$

Example 2.4. (i) We show a derivation of type $\{v\}\rightarrow v$ for $(\lambda x.xx)(\lambda y.y)$.
Let Δ be $\lambda x.xx$, $\mathbf{D}_\Delta$ is the following derivation of a type for Δ

$$(\rightarrow\text{I})\ \dfrac{(\rightarrow\text{E})\ \dfrac{\{x:\{t\}\rightarrow u\}\vdash_{\wedge}x:\{t\}\rightarrow u\qquad (\text{INT})\ \dfrac{\{x:t\}\vdash_{\wedge}x:t}{\{x:t\}\vdash_{\wedge}x:\{t\}}}{\{x:\{t\}\rightarrow u,\ x:t\}\vdash_{\wedge}(xx):u}}{\emptyset\vdash_{\wedge}\lambda x.(xx):\{t;\{t\}\rightarrow u\}\rightarrow u}$$

Observe that two different types are assigned to the two occurrences of x. Note that $\lambda x.(xx)$ has not type in the ML type system.
(ii) Let $\mathbf{D}_1$ be the following derivation of an intersection for the identity :

$$(\forall\text{E})\ \dfrac{(\forall\text{I})\ \dfrac{(\rightarrow\text{I})\ \dfrac{\{y:t\}\vdash_{\wedge}y:t}{\emptyset\vdash_{\wedge}\lambda y.y:\{t\}\rightarrow t}}{\emptyset\vdash_{\wedge}\lambda y.y:\forall t.\{t\}\rightarrow t}}{\emptyset\vdash_{\wedge}\lambda y.y:\{\{v\}\rightarrow v;\{\{v\}\rightarrow v\}\rightarrow\{v\}\rightarrow v\}}$$

The intersection assigned to $\lambda y.y$ is generated by two different instances of $\{t\}\rightarrow t$ obtained by ($\forall$E).

(iii) Let $\mathbf{D}_2 = \mathbf{D}_\Delta[t,u:=\{v\}\to v]$ be a proof of

$\varnothing\vdash_\wedge \lambda x.(xx):\{\{v\}\to v;\{\{v\}\to v\}\to\{v\}\to v\}\to\{v\}\to v\}.$

Combinig $\mathbf{D}_2$ and $\mathbf{D}_1$ by $(\to E)$ we get a derivation of $\varnothing\vdash_\wedge(\lambda x.(xx))(\lambda y.y):\{v\}\to v$. □

Also terms which have a type in ML can be assigned "more informative" types in $\vdash_\wedge$. Take, for instance, the term **twice** = $\lambda f.\lambda x.\ f\ (f\ x)$ whose principal type scheme in ML is $(t\to t)\to t\to t$. **twice** represents a combinator which applies twice a function f to an argument x. In $\vdash_\wedge$ we can assign to **twice** type $\{\{t\}\to u;\{u\}\to v\}\to\{t\}\to v$ (where t, u, v are type variables). The meaning of this type can be informally explained saying that, if f is a function that maps both t into u and u into v then $f\circ f$ maps t into v . Now, referring to a more extended set of types containing a "list" type constructor (as ML, but lists can be represented in the pure λ-calculus as well), let F be a term of type $\{t\}\to$t-list (for example, F could be a polymorphic function mapping an object x of an arbitrary type t into the list containing x as the unique element). Then , using (∀I) and (∀E) we can deduce

$\vdash_\wedge$**twice**: $\{\{t\}\to$t-list ;{t-list}$\to$t-list-list$\}\to\{t\ \}\to$t-list-list and

$\vdash_\wedge$(**twice** F):$\{t\ \}\to$t-list-list

while no type could be assigned to (**twice** F) in ML.

In the following we establish the property of subject reduction for the system $\vdash_\wedge$. Subject reduction tells us that types are preserved by computations. An immediate consequence of subject reduction is the well-known property that "typed terms cannot go wrong", i.e. the fact that no type-incorrect application (like, for instance, (3 3)) can appear reducing a well-typed term.

Theorem 2.5 (*Subject reduction*). If $B\vdash_\wedge M:\alpha$ and $M\to^*_{\beta\eta}N$, then $B'\vdash_\wedge N:\alpha$ for some $B'\subseteq B$, where α is a type, an intersection or a scheme. □

The reason for which, in general, we have only $B'\subseteq B$ is that, in a reduction step $(\lambda x.M)N\to_\beta M[x:=N]$, if x does not occur in M, M[x:=N] can have less free variables than $(\lambda x.M)N$.

Two main properties of the system are the strong normalization property and the fact that each term in normal form without occurrences of constants has a type. The condition that there be no occurrences of constants is necessary since terms like (3 3), although being in normal form, cannot be typed. Indeed any term in normal form that has not a type in $\vdash_\wedge$ contains some incorrect application and is semantically meaningless.

Strong normalization follows from the strong normalization theorem for the system with unrestricted intersection and universal quantification proved in [Leivant, 1986] of which $\vdash_\wedge$ is a proper subsystem.

Property 2.6. If If $B\vdash_\wedge M:\alpha$ for some B and α then M is strongly normalizable. □

Property 2.7. If M is a term in β-normal form then there exists a basis B and a type α such that $B \vdash_\wedge M{:}\alpha$.

Proof. By induction on M. We have four possible cases.

If $M \equiv x$ is a variable take $\{x{:}t\} \vdash_\wedge x{:}t$ where t is a type variable.

If $M \equiv \lambda x.M'$ then, by induction hypothesis, we have $B' \vdash_\wedge M'{:}\alpha$ for some B', α. There are two possible cases:

- If x occurs in M' then let $B' = B/x \cup \{x{:}\beta_1,\ldots,x{:}\beta_n\}$. By an application of (→I), we get $B/x \vdash_\wedge \lambda x.M'{:}\{\beta_1;\ldots;\beta_n\} \to \alpha$.
- If x does not occur in M' then $B = B/x$ and we get $B \vdash_\wedge \lambda x.M'{:}\{\,\} \to \alpha$.

If $M \equiv (xM_1\ldots M_n)$ $(n \geq 1)$ then by induction hypotheses there are B_i, α_i $(1 \leq i \leq n)$ such that $B_i \vdash_\wedge M_i{:}\alpha_i$. Now let $B = B_1 \cup \ldots \cup B_n$ and $\sigma \equiv \{\alpha_1\} \to \ldots \to \{\alpha_n\} \to t$ where t is a new type variable. By applying n times (→E) we get $B \cup \{x{:}\sigma\} \vdash_\wedge M{:}t$. □

The type assigned to the normal form M in the proof of 2.7 corresponds in some sense to its principal type, as it will be remarked in section 4.

We end this section with a result about the relations between $\vdash_\wedge$ and the ML type assignment system. Such a system is an extension of the basic Curry's type assignment system for λ-terms [Curry, Feys, 1958] in which the new syntactic form **let** has been added to the pure λ-calculus. The form **let** is the only operator that allows to exploit the notion of scheme.

The ML terms are defined by the same syntax of the λ-calculus with constants with the addition of the clause that **let** x = N **in** M is a term whenever M and N are terms. Operationally this term is equivalent to (λx.M)N, but not from the point of view of typechecking. The set T_{ML} of *ML-types* is defined from K (the set of basic types) and V (the set of type variables) by the rule

- $\alpha \to \beta$ is a type whenever α, β are types

The set S_{ML} of ML-*type schemes* is defined from T_{ML} as in Def. 2.1(ii). Then T_{ML} and S_{ML} are, respectively, isomorphic to a subset of T and S. The type assignment rules are given following [Damas, Milner, 1982], but keeping our syntax.

Definition 2.8. (i) An ML-basis B is a set $\{x_i{:}\alpha_i \mid i \in I\}$, where x_i are term variables and α_i are ML-types or ML-schemes, such that for each variable x_i there is at most one premise in B.

(ii) The type assignment rules are the following, where α, β denote ML-types and σ a ML-scheme:

(VAR) $B \vdash_{ML} x{:}\alpha$ if $x{:}\alpha \in B$ where α is a type or a scheme.

$$(\to I)\quad \frac{B/x \cup \{x{:}\alpha\} \vdash_{ML} M{:}\beta}{B/x \vdash_{ML} \lambda x.M{:}\alpha \to \beta}$$

$$(\to E)\quad \frac{B \vdash_{ML} M{:}\alpha \to \beta \qquad B \vdash_{ML} N{:}\alpha}{B \vdash_{ML} (MN){:}\beta}$$

$$(\forall I)\ \frac{B \vdash_{ML} M:\alpha}{B \vdash_{ML} M:\forall \underline{t}.\alpha} \quad \text{(if all variables in } \underline{t} \text{ do not occur free in B)}$$

$$(\forall E)\ \frac{B \vdash_{ML} M:\forall \underline{t}.\alpha}{B \vdash_{ML} M:\alpha[\underline{t}:=\underline{\beta}]} \quad \text{(where } \underline{\beta} \in T_{ML})$$

$$(LET)\ \frac{B/x \cup \{x:\sigma\} \vdash_{ML} M:\alpha \quad B/x \vdash_{ML} N:\sigma}{B \vdash_{ML} (\textbf{let } x=N \textbf{ in } M):\alpha} \qquad \Box$$

Note that in rule ($\rightarrow$E) we are forced to consider the same basis in both premises, since any variable x must occur at most once in B, and the assumption for x must be the same in both deductions. The (LET) rule is essential in ML since a term **let** x = N **in** M can have a type while (λx.M)N, although operationally equivalent, may not. We shall see that the **let** form is useless (as far as typechecking is concerned) in $\vdash_\wedge$. Moreover $\vdash_\wedge$ is more powerful than $\vdash_{ML}$, since there are terms that have a type in $\vdash_\wedge$ and have none in $\vdash_{ML}$, even using **let** instead of application whenever possible (like, for instance, λx.xx).

To make more formal the relation between $\vdash_{ML}$ and $\vdash_\wedge$ we define a formal translation from the set of ML types and schemes into our language. Let $()^*:T_{ML}\cup S_{ML}\rightarrow T\cup S$ be defined by:

- $t^*=t$ - $c^*=c$ where t is type variable and c a basic type.
- $(\alpha\rightarrow\beta)^*=\{\alpha^*\}\rightarrow\beta^*$
- $(\forall\underline{t}.\alpha)^*=\forall\underline{t}.\alpha^*$

Since schemes cannot be assigned to variables in $\vdash_\wedge$ we must translate an assumption x:σ in $\vdash_{ML}$ to a set of assumptions $\{x:\alpha_1,\ldots,x:\alpha_n\}$. So if B is an ML basis we define

$$B^*=\left\{ B' \;\middle|\; \begin{array}{l} x:\sigma\in B \Rightarrow x:\alpha_1\ldots x:\alpha_n\in B' \text{ for some instances } \alpha_1\ldots\alpha_n \text{ of } \sigma^* \ (n\geq 0) \\ x:\alpha\in B \Rightarrow \text{ either } x:\alpha^*\in B' \text{ or x does not occur in } B' \end{array} \right\}.$$

B^* is the set of all bases that could correspond to B translating a deduction in $\vdash_{ML}$ to one in $\vdash_\wedge$.

Finally let M be an ML-term. Define M^* as the pure λ-term obtained by replacing in M each occurrence of **let** x=N **in** M by (λx.M)N. The following property can be easily proved by induction on derivations in $\vdash_{ML}$.

Property 2.9. If $B\vdash_{ML}M:\alpha$ then there exist $B'\in B^*$ such that $B'\vdash_\wedge M^*:\alpha^*$. $\Box$

In [Kfoury, Tyurin, 1990] it has been proved that $\vdash_{ML}$ is essentially equivalent to the (type inference version of the) polymorphic (or second-order) typed λ-calculus limited at rank 2, i.e., in which the $\forall$ operator may occur only in the left scope of not more than one arrow. Then by the previous property in $\vdash_\wedge$ we can assign a type to more terms that in the polymorphic typed λ-calculus of rank 2 even though, as for ML, the types can be different.

In $\vdash_\wedge$ we can give a type also to terms which cannot be typed in the full polymorphic λ-calculus, like the term ((λy.λx.y(x(λz.z))(x(λu.λv.u))) Δ) (see [Giannini, Ronchi, 1988] for the proof that this term cannot be typed in the second-order calculus). We leave as an exercise to find a type for this term in $\vdash_\wedge$. On the other side there are terms which have a type in the second-order λ-calculus (even limited at rank 3) for which we can not infer a type in our system. An example is given by ((λx.xΔx)(Δ **twice**)) which in the second-order type assignment system at rank three has type $\forall t.(t\to t)\to t\to t$ while it has no type in $\vdash_\wedge$.

As already remarked the present system is a restriction of the intersection type system $\vdash_{int}$ of [Coppo, Dezani, 1980] (see also [Van Bakel, 1991] for a more detailed description and properties). In particular in $\vdash_{int}$ there is a rule of intersection introduction of the kind

$$(\wedge I)\ \frac{B\vdash_{int}M:\alpha_1\ \dots\ B\vdash_{int}M:\alpha_n}{B\vdash_{int}M:\{\alpha_1;\dots;\alpha_n\}}$$

which allows to assign to a term M an intersection of types whose deductions can have a quite different structure. In our system, instead, we can assign to M only intersection obtained by generic instance of a unique type. The system $\vdash_{int}$ is very strong (it allows to give a type to all strongly normalizable terms) but its type assignment is undecidable [van Bakel, 1991], as for the case of the intersection type system of [Barendregt, Coppo, Dezani, 1983] (see also [Ronchi, Venneri, 1984]).

An interesting restriction of $\vdash_{int}$ is that of allowing only intersections at rank two. That is intersections may occur either at top level or in the left scope of only one arrow. Also in this system (let call it $\vdash_{int,2}$) we can give type to (the translation in the pure λ-calculus of) all ML-terms which have a type in the ML type system [Leivant, 1983]. $\vdash_{int,2}$, moreover, has the principal type scheme property and its typechecking is decidable. The set of terms that can be typed in $\vdash_{int,2}$ and $\vdash_\wedge$ are, however, incomparable. There are terms that can be typed in $\vdash_\wedge$ but not in $\vdash_{int,2}$. Not all normal forms, for instance, can be typed in $\vdash_{int,2}$ and conversely, there are terms that can be typed in $\vdash_{int,2}$ but not in $\vdash_\wedge$. The main reason for this is that in $\vdash_\wedge$ we cannot assign an intersection to a variable in a basis, while this would be necessary to type terms like $\Delta'=\lambda y.\Delta\ y$. In fact we should assign $\{t;\{t\}\to u\}$ to y (see example 2.4) to give a type to Δ', and this can be done in $\vdash_{int,2}$ but not in $\vdash_\wedge$.

More generally the terms that can be typed in $\vdash_{int,2}$ and not in $\vdash_\wedge$ have at least a subterm of the shape (M x) where x is a variable and M is a term that would not have a type in ML. Instead terms of the kind (f M) where M is a term that can have intersections in its type are typed in $\vdash_\wedge$ but not in $\vdash_{int,2}$.

3. The system $\vdash_L$.

Now we turn to the problrm of authomatic inference of types for the system $\vdash_\wedge$. In particular we want to establish a dependency between all different derivations for the same term via a notion of principal type and principal deduction [Hindley, Seldin, 1986]. To this aim we will introduce a "more informative" system $\vdash_L$ in which deductions are (informally speaking) schemes which represent a class of deductions in the original system $\vdash_\wedge$. In fact $\vdash_L$ turns out to be equivalent to $\vdash_\wedge$. The main feature of $\vdash_L$ is the introduction of a notion of labelled universal quantifier. A labelled quantifier is a syntactic tool used to represent a class of intersections, namely all these that could be obtained by (∀E) from a given scheme. Let us consider the following examples.

Let $\mathbf{D}_I$ be the following deduction, where I is $\lambda x.x$.

$$\mathbf{D}_I: \quad (\to I)\,\frac{\{x:t\}\vdash_\wedge x:t}{\emptyset\vdash_\wedge \lambda x.x:\{t\}\to t}$$

If we want to assign a type to a term $\lambda x.xI$ the most obvious deduction seems to be the following:

$$\mathbf{D}_2: \quad (\to I)\,\frac{(\to E)\,\dfrac{\{x:\{\{t\}\to t\}\to u\}\vdash_\wedge x:\{\{t\}\to t\}\to u \qquad (INT)\,\dfrac{\mathbf{D}_I}{\emptyset\vdash_\wedge \lambda x.x:\{\{t\}\to t\}}}{\{x:\{\{t\}\to t\}\to u\}\vdash_\wedge (xI):u}}{\emptyset\vdash_\wedge \lambda x.(xI):\{\{t\}\to t\}\to u\}\to u}$$

This deduction scheme is good if we want to apply, for instance, $\lambda x.xI$ to I (we have just to make an instance of it replacing u by $\{t\}\to t$, observing that $\{\{t\}\to t\}\to\{t\}\to t$ is a type for I.

However, if we want to apply $\lambda x.xI$ to Δ, which has type $\{t;\{t\}\to u\}\to u$ (and, obviously, all its substitution instances), we cannot use any deduction for $\lambda x.xI$ which is a substitution instance of $\mathbf{D}_1$. Instead we have to use the following deduction for $\lambda x.xI$.

$$\mathbf{D}_2: \quad (\to I)\,\frac{(\to E)\,\dfrac{\{x:\alpha\}\vdash_\wedge x:\alpha \qquad (\forall E)\,\dfrac{(\forall I)\,\dfrac{\mathbf{D}_I}{\emptyset\vdash_\wedge \lambda x.x:\forall t.\{t\}\to t}}{\emptyset\vdash_\wedge I:\{\{t\}\to t;\{\{t\}\to t\}\to\{t\}\to t\}}}{\{x:\alpha\}\vdash_\wedge xI:\{t\}\to t}}{\emptyset\vdash_\wedge \lambda x.xI:\{\alpha\}\to\{t\}\to t}$$

where $\alpha=\{\{t\}\to t;\{\{t\}\to t\}\to\{t\}\to t\}\to\{t\}\to t$. In fact α is a type for Δ, being an instance of $\{t;\{t\}\to u\}\to u$ (replace simultaneously t and u by $\{t\}\to t$).

Note that $\mathbf{D}_2$ (and any of its substitution instances) whould not be useful to type $((\lambda x.xI)\ I)$ since no type of the form $\{\alpha;\beta\}\to\gamma$ can be assigned to I.

We can see, on the contrary, that any derivation of a type for I (or Δ) can be obtained by substitution from $\mathbf{D}_I$ (or $\mathbf{D}_\Delta$) (adding, possibly, an application of $(\forall I)$-$(\forall E)$ at the end), so we could claim that $\{t\}\to t$ and $\{\{t\}\to u;t\}\to u$ are, respectively, the principal types for I and Δ. However, in the case of $\lambda x.xI$ we have not, in $\vdash_\wedge$ any good candidate to be its principal type.

We now introduce a new system in which we can represent a potential intersection using the notion of "labelled" universal quantifier. In this system, for instance, the principal type of $\lambda x.xI$ is a type $\forall u.\ \{(\overset{\downarrow}{\forall} t.\{t\}\to t)\to u\}\to u$ where the internal quantified type $\overset{\downarrow}{\forall} t.\{t\}\to t$ does indeed represent all the possible intersections that can be obtained from it by $(\forall E)$, thus allowing a unified treatement of deductions like $\mathbf{D}_1$ and $\mathbf{D}_2$. We remark, however, that a labelled quantifier is not intended to be a universal quantifier as in the second-order polymorphic lambda-calculus. In fact a second-order type $\forall t.\alpha$ can be instantiated, using $(\forall E)$, in different ways in each of its occurrences in a deduction while a labelled quantifier $\overset{\downarrow}{\forall} t.\alpha$ is a sort of intersection scheme that must be instantiated in the same way in all its possible occurrences in a deduction. This will force us to introduce a somewhat more complicated rule for $\forall$ elimination.

We now introduce formally the system $\vdash_L$ with labelled quantifiers. Labelled quantifiers can be considered both as a particular kind of intersections or as schemes. So they can occurr either in the left-hand-side of $\to$ or as the final result of a deduction.

Definition 3.1. (i) The sets T_L, Q_L of *labelled types* and *intersections* are defined inductively from K and V and a set L of *labels* in the following way:

- $V, K \subseteq T_L$
- $\{\alpha_1;...;\alpha_n\} \in Q_L$ whenever $\alpha_1,...,\alpha_n \in T_L$
- $\overset{l}{\forall}\underline{t}.\alpha \in Q_L$ whenever $\alpha \in T_L$ and $l \in L$ (l is *the label* of $\overset{l}{\forall}\underline{t}.\alpha$)
- $\xi \rightarrow \beta \in T_L$ whenever $\xi \in Q_L$, $\beta \in T_L$

with the constraint that all types with the same label must be α-equivalent (differ at most in the name of bound variables).

(ii) A *type context* $\alpha[-]$ is a type α with a missing subterm. $\square$

Each labelled $\forall$ identifies indeed a type rather than simply a type operator. There are no two types $\overset{l}{\forall}\underline{t}.\alpha$, $\overset{l}{\forall}\underline{u}.\beta$ with the same label such that α is different from β (modulo a re-naming of bound type variables). In particular there cannot be two nested labelled $\forall$ with the same label.

A type context just represents the part of a type surrounding a given subtype. Owing to the previous condition there is in general no freedom in choosing the term that has to fill the hole, if this occurs in the scope of a labelled quantifier.

In order to handle labelled quantifiers we need the operation of expansion, which is crucial in the typechecking algorithm. This operation is defined relatively to a context represented by a set of types or type contexts.

Definition 3.2 (i) The *scope* of a label l occurring in $\overset{l}{\forall}\underline{t}.\alpha$ is α (by the above condition on labelled types α is unique).

(ii) A *environment* is a finite set of types, intersections and type contexts.

(iii) Let A be a environment. We say that a label l *directly dominates* another label m in A if:

1. m occurs in A only in the scope of l.
2. there is no other label l' which occur in the scope of l and such that m occurs in the scope of l'.

(iv) A label l *dominates* m in A if there are $m_1,...,m_k$ $(k>2)$ such that $l=m_1$, $m=m_k$ and each m_i directly dominates m_{i+1}.

As remarked before the instantiation of a labelled quantifier must be performed simultaneously on all its occurrences in a given environment. This is achieved by the operation of expansion.

Definition 3.3. (i) Let A be a environment. A *simple expansion* in A is a partial function denoted $[\overset{l}{\forall}\underline{t}.\alpha:=\xi]$ where ξ can be

-1. an intersection $\{\alpha_1[\underline{t}:=\underline{\beta}_1]);...;\alpha_n[\underline{t}:=\underline{\beta}_n])\}$ $(n>1)$ where $\underline{\beta}_1,...,\underline{\beta}_n \in T_L^*$ and α_i are obtained from α, respectively, by choosing an independent name (say $\overset{m_i}{\forall}$) for each labelled $\overset{m}{\forall}$ in α which is dominated by l in A. In this case we say that m_i is the label *corresponding* to m in α_i.

- 2. $\overset{m}{\forall}\underline{u}.\beta[\underline{y}/\underline{t}]$ where $\underline{u}$ are variables which occur free in $\underline{y}$ and not in β or A, and m is a new label not occurring in A.

(ii) A simple expansion $r = [\overset{l}{\forall}\underline{t}.\alpha:=\xi]$ can be extended to the set of all types and sequences by defining:

$r(\overset{l}{\forall}\underline{t}.\alpha) = \xi$

$r(\alpha\to\beta)= r(\alpha)\to r(\beta)$

$r(\{\alpha_1;...;\alpha_n\}) = \{r(\alpha_1);...;r(\alpha_n)\}$.

(iii) An *expansion* is a composition of simple expansions. □

Example 3.4. Let $A = \{(\overset{m}{\forall}u.\{(\overset{l}{\forall}\underline{t}.t\to t)\to u\}\to u\ )\to v,\ v\}$. An example of simple expansion in A is

$r=[\overset{m}{\forall}u.\{(\overset{l}{\forall}\underline{t}.t\to t)\to u\}\to u := \{\{(\overset{l_1}{\forall}\underline{t}.t\to t)\to\alpha\}\to\alpha\ ;\{(\overset{l_2}{\forall}\underline{t}.t\to t)\to\beta\}\to\beta\}]$

where α,β are types. The result of applying r to A is the set

$\{\{\{(\overset{l_1}{\forall}\underline{t}.t\to t)\to\alpha\}\to\alpha;\ \{(\overset{l_2}{\forall}\underline{t}.t\to t)\to\beta\}\to\beta\}\to v,\ v\}$.

If we take instead

$r'=[\overset{l}{\forall}\underline{t}.t\to t:=\{\{v\}\to v;\{\{v\}\to v\}\to\{v\}\to v\}]$

the result of applying r' to A is

$\{(\overset{m}{\forall}u.\{\{\{v\}\to v\ ;\ \{\{v\}\to v\}\to\{v\}\to v\}\to u\}\to u)\to v,\ v\}$. □

If B is a basis then $A_B = \{\beta \mid x:\beta\in B\}$. In the following we will often use B as a environment, identifying B and A_B.

We now introduce the type assignment system $\vdash_L$. The main differences over $\vdash_\wedge$ are the introduction of two new rules to handle labelled quantifiers.

Definition 3.5.(i) The type assignment system $\vdash_L$ is defined as $\vdash_\wedge$ by replacing ($\forall$I) and ($\forall$E) by the following rules ($\forall^L$I) and (EXP) .

$$(\forall^L I)\ \frac{B\vdash_L M:\alpha}{B\vdash_L M:\overset{l}{\forall}\underline{t}.\alpha}$$

where α is type, l is any label and the variables in $\underline{t}$ do not occur in B.

$$(EXP)\ \frac{B\vdash_L M:\alpha[\overset{l}{\forall}\underline{t}.\beta]}{B\vdash_L M:\alpha[\xi]}$$

where α is either a type or an intersection, l does not occur in B, $\alpha[]$ and ξ has been obtained by a simple expansion of $\overset{l}{\forall}\underline{t}.\beta$ in the environment $B\cup\{\alpha[]\}$. □

Note that a corresponding of rule ($\forall$E) can be obtained as a particular case of rule (EXP) where $\alpha[]$ is the empty context.

Example 3.6. We show a deduction of an intersection for $\lambda x.x(\lambda y.y)$ (indeed its principal deduction) in $\vdash_L$

$$(\forall^L I)\ \dfrac{(\to I)\ \dfrac{(\to E)\ \dfrac{(VAR)\ \{x:(\overset{m}{\forall}t.\{t\}\to t)\to u\}\vdash_L x:(\overset{m}{\forall}t.\{t\}\to t)\to u \qquad (\forall I)\ \dfrac{\mathbf{D}_I}{\emptyset\vdash_L \lambda y.y:(\overset{m}{\forall}t.\{t\}\to t)}}{\{x:(\overset{m}{\forall}t.\{t\}\to t)\to u\}\vdash_L x(\lambda y.y):u}}{\emptyset\vdash_L \lambda x.xI:((\overset{m}{\forall}t.\{t\}\to t)\to u)\to u}}{\emptyset\vdash_L \lambda x.xI:\overset{l}{\forall}u.\{(\overset{m}{\forall}t.\{t\}\to t)\to u\}\to u}$$

The present notion of expansion (with respect, in particular, to the relabelling of the internal labels in Definition 3.3(i)-1.) is needed in order to keep the maximum level of generality in the potential intersections represented by labelled $\forall$. Take for instance the statement

$\emptyset\vdash_L \lambda x.xI:\overset{l}{\forall}u.\{(\overset{m}{\forall}t.\{t\}\to t)\to u\}\to u.$

The informal meaning of $\overset{m}{\forall}t.\{t\}\to t$ is that we can deduce $\emptyset\vdash_L \lambda x.xI:\overset{l}{\forall}u.\{\xi\to u\}\to u$ where ξ is any intersection that can be obtained by (simple) expansion from $\overset{m}{\forall}t.\{t\}\to t$. Now observe that we are indeed able to deduce, for instance,

$\emptyset\vdash_\wedge \lambda x.xI:\{\{\xi_1\to u_1\}\to u_1;\ \{\xi_2\to u_2\}\to u_2\}$

where ξ_1, ξ_2 are any two expansions of $\overset{m}{\forall}t.\{t\}\to t$. In fact let $\xi_1=\{\alpha_1;...;\alpha_h\}$ and $\xi_2=\{\beta_1;...;\beta_k\}$ (where α_i and β_i are instances of $\{t\}\to t$) and let p=max(h,k). We can deduce

$\emptyset\vdash_\wedge \lambda x.xI:\{\{\{t_1\}\to t_1;...;\{t_p\}\to t_p\}\to u\}\to u$

applying ($\forall$I) and ($\forall$E) to the endstatement of $\mathbf{D}_I$ and assigning a suitable type to x. Then, assuming that $t_1,...,t_p$ do not occur elsewhere in the deduction, we have

$\emptyset\vdash_\wedge \lambda x.xI:\forall u t_1...t_p.\{\{\{t_1\}\to t_1;...;\{t_p\}\to t_p\}\to u\}\to u.$

Now we can get

$\emptyset\vdash_\wedge \lambda x.xI:\{\{\xi_1\to u_1\}\to u_1;\ \{\xi_2\to u_2\}\to u_2\}$

by ($\forall$E) replacing the sequence u $t_1...t_p$ in the two copies generated by ($\forall$E), respectively, by $\underline{\alpha}=u_1\ \alpha_1...\alpha_p$ and $\underline{\beta}=u_2\ \beta_1...\beta_p$. If $\underline{\alpha}$ or $\underline{\beta}$ have less than p+1 types it is enough to identify some of the types replaced for the variables in $\underline{t} = t_1...t_p$ in them (intersections are seen modulo repetitions of the same type). By expanding $\overset{l}{\forall}u.\{(\overset{m}{\forall}t.\{t\}\to t)\to u\}\to u$ in

$\{\{(\overset{m_1}{\forall}t.\{t\}\to t)\to u_1\}\to u_1;\{(\overset{m_2}{\forall}t.\{t\}\to t)\to u_2\}\to u_2\}$

we keep track of this possibility, that would be lost if we identify the labels m_1 and m_2 (recall that expansion is performed uniformly on all label occurrences). Obviously this construction is possible only if the variables $\underline{t}$ can be quantified, i.e. if they (or equivalently $\overset{m}{\forall}$) do not occur elsewhere outside the range of $\overset{l}{\forall}$.

The need for a rule (EXP) more powerful than ($\forall$E) is a consequence of the possibility of renaming internal labels allowed by expansion. Since we want deduction in $\vdash_L$ to be closed under expansion (this is essential to prove the correctness of the typechecking algorithm), we must be able to expand also the internal labels like m_1 and m_2. This would not be possible without (EXP) (m_1 and m_2 are created only after the expansion of the label l).

Lemma 3.7. Deductions in $\vdash_L$ are closed by expansion, i.e. if $B\vdash_L M:\alpha$ and B', α' are obtained from B, α by expansion in the environment $B\cup\{\alpha\}$ then $B'\vdash_L M:\alpha'$. □

It is obvious that if $B\vdash_{\wedge}M:\alpha$ then $B\vdash_{L}M:\alpha$ (the rules of $\vdash_{\wedge}$ are weaker than the rules of $\vdash_{L}$). To prove the converse we need to establish a correspondence between types and intersections in $\vdash_{\wedge}$ (sets T and Q) and the (possibly) labelled ones in $\vdash_{L}$ (sets T_L and Q_L). In particular a labelled quantification in $\vdash_{L}$ can always be replaced by an intersection in $\vdash_{\wedge}$.

Definition 3.8. Let A be a environment. An expansion r in A is a *terminal expansion* if, for all $\alpha\in A$, $r(\alpha)\in T\cup Q$, i.e. no labelled quantifier is left in α. $\square$

The correspondence between types and derivations in $\vdash_{\wedge}$ and in $\vdash_{L}$ is given by the following result, which states that $\vdash_{L}$ is conservative with respect to $\vdash_{\wedge}$.

Lemma 3.9. Let $B\vdash_{L}M:\alpha$ $(\alpha\in T_L\cup Q_L)$. Then $B'\vdash_{\wedge}M:\alpha'$ where B', α' is any terminal expansion of B, α. $\square$

4. Typechecking

In this section we will prove the existence of principal type schemes and of a complete type inference algorithm for the system $\vdash_{L}$. This implies also the existence of a complete type inference algorithm for $\vdash_{\wedge}$ by Lemma 3.9. As remarked in the introduction the finitary nature of intersections determines the fact that we will have a set of principal types instead of a unique principal type as in the case of the ML type system. However each of them will be "more general" than the ML principal type scheme.

The crucial step in the type inference algorithm which is defined by induction on the structure of the term is obviously the case of application of two terms. For this case a sort of unification procedure is defined via the notion of reduction of a system of equations. It is useful, to this purpose, to introduce a notion of transformation on types which is a combination of substitutions and expansions.

Let us define, following Definition 3.3, a *simple substitution* $[t:=\alpha]$ to be a partial map between type variables and types in T_L. Simple substitutions can be extended to $T_L\cup Q_L$ in the obvious way, assuming that $[t:=\alpha](v)=v$ if $v\neq t$. A *substitution* is then a composition of simple substitutions.

Definition 4.1. (i) A *transformation* in a environment A is a composition of substitutions and expansions in A. We will write down explicitly a transformation as

$$r = [t_1:=\alpha_1,\dots,t_n:=\alpha_n,\forall^{l_1}\underline{t}_1.\beta_1:=\xi_1,\dots,\forall^{l_m}\underline{t}_m.\beta_m:=\xi_m]$$

$\{t_1,\dots,t_n,\forall^{l_1}\underline{t}_1.\beta_1,\dots,\forall^{l_m}\underline{t}_m.\beta_m\}$ is the *domain* of r and $\{\alpha_1,\dots,\alpha_n,\xi_1,\dots,\xi_m\}$ its *range*. We assume, without loss of generality, that the set of free variables of the domain and the set of free variables of the range of a transformations are disjoint.

(ii) $r\leq r'$ if $r'=r''\circ r$ for some transformation r''. $\square$

Note that if

$$r = [t_1{:=}\alpha_1,\dots,t_n{:=}\alpha_n, {}^{l_1}\forall \underline{t}_1.\beta_1{:=}\xi_1,\dots,{}^{l_m}\forall \underline{t}_m.\beta_m{:=}\xi_m]$$

and $s = [u{:=}\gamma]$ is, for instance, a simple substitution, then

$$s\circ r = [u{:=}\gamma, t_1{:=}s(\alpha_1),\dots,t_n{:=}s(\alpha_n), {}^{l_1}\forall \underline{t}_1.\beta_1{:=}s(\xi_1),\dots,{}^{l_m}\forall \underline{t}_m.\beta_m{:=}s(\xi_m)]$$

(where we assume that u is different from $t_1,\dots,t_m$). The same property holds if s is a simple transformation.

We consider the identity Id as the "empty" transformation, i.e such that $Id(\alpha)=\alpha$ for all types and sequences α. Note that $Id\leq r$ for all transformations r. If $r\leq r'$ then each label occurring in the domain of r must occur in the domain of r'.

We can now formally define what we mean by system of equations.

Definition 4.2 Let A be a environment.
(i) A *system* S in contex A is a set $S = \{(\alpha_i,\beta_i) \mid i\in I\}$ where α_i, and β_i are either both labelled types or both labelled intersections. Let FV(S) be the set of all variables free in S. A system S is *solved* if $\alpha_i=\beta_i$ for all $i\in I$.
(ii) A *solution* of a system S in a environment A is a transformation r in A such that r(S) is solved.
(iii) A set of solutions $\{r_i \mid i\in I\}$ of a system S is a *principal set of solutions* if for all solutions r' of S there is $i\in I$ such that $r_i\leq r'$. □

It is clear that the empty system is solved and any transformation r is a solution for it.

We now give a constructive proof that any system S which has a solution has a principal set of solutions. This is obtained by reducing the complexity of S by a collection of formal reduction rules which generate, in the process of reducing S, a set of transformations. We then show that S has a solution if and only if it can be reduced to the empty set. In this case the set of transformations produced by the reduction process is a principal set of solutions for S. Since the reduction process does always terminate, the reduction rules define an algorithm to find the principal set of solutions for S.

In particular we introduce a relation $\Rightarrow_A$ which reduces, in a contex A, a system S producing a transformation r and a set of (simpler) systems $\{S_1,\dots,S_n\}$. We then show that S has a solution r' if and only if $r'=r''\circ r$ for some r'', where r'' is a solution of S_i for some i ($1\leq i\leq n$). That is the transformation r is really needed in the solution of S, and the result of applying r to S is S_i. In all cases except one we have n=1 (i.e. S is reduced to a unique system S_1). But in one case (corresponding to the attempting the unification of two intersections) we can make different choices which determine a set of (possibly) incomparable solutions. The solution algorithm can thus be described, as for Huet's higher order unification [Huet, 1976] , as the construction of an OR tree whose nodes and brances are labelled, respectively, by systems and by transformations. The leaves are either the empty system, corresponding to a success in the solution of the system, or a non empty irreducible system, with respect to $\Rightarrow_A$, corresponding to a failure. A solution of S can be found composing the transformations on a path from the root to a success node. In this way we find the set of principal solutions. Finally we prove that such set is always finite (the unification tree does not have infinite paths).

In defining the reduction $\Rightarrow_A$ some care must be taken in case the reduction is determined by the attempt of matching two quantified types (this may happen if there is a pair $(\overset{l}{\forall}\underline{t}.\alpha, \overset{m}{\forall}\underline{u}.\beta)$ in S). For this case we introduce an extension of transformations in which a labelled quantified type $\overset{l}{\forall}\underline{t}.\alpha$ can be associated with a type in T_L indexed by a sequence of variables,denoted $\gamma|_{\underline{w}}$. The variables that index a type are not affected by transformations, i.e. $[v:=\delta](\gamma|_{\underline{w}})=\gamma[v:=\delta]|_{\underline{w}}$ even if v occurs in $\underline{w}$. The indexed type $\gamma|_{\underline{w}}$ is temporarily used in the system reduction to indicate that γ in the final transformation must be quantified with respect to the variables $\underline{w}$. Such types will be replaced in the final solution of the system by the corresponding labelled quantified types.

Definition 4.3 Let S be a system and α, β types or intersections.
(i) I_S is the set $\{\alpha \mid \alpha \in T_L \cup Q_L$, and either $(\alpha,\beta) \in S$ or $(\beta,\alpha) \in S\}$ of all types or intersections occurring in pairs of S.
(ii) For $\alpha \in I_S$, $\mathfrak{I}_S(\alpha)$ is the equivalence class of α in S . I.e.

$\mathfrak{I}_S(\alpha)=\{\beta \mid \beta \equiv \alpha$ *or* for some $\beta' \in \mathfrak{I}_S(\alpha)$ either $(\beta',\beta) \in S$ or $(\beta,\beta') \in S\}$. □

Therefore $\mathfrak{I}_S = \{\mathfrak{I}_S(\alpha) \mid \alpha \in I_S\}$ is a partition of I_S.

We introduce a more technical definition to formally handle the matching of two intersections.

Definition 4.4. Let R be a relation on A×B. R is *surjective* if and only if the *domain* of R (the set $\{a \mid (a,b) \in R$ for some b$\}$) is A and the *range* of R (The set $\{b \mid (a,b) \in R$ for some a$\}$, is B. R is *minimally surjective* if for no $(a,b) \in R$, $R-\{(a,b)\}$ is a surjective relation. □

In the following definition we write S+S' to indicate the union of two disjoint sets of equations S (i.e. no pair of S' is in S and vice-versa).

Definition 4.5 (*System reduction*). (i) Let A be a environment. The relation $\Rightarrow_A$ between a system S and a pair consisting of a transformation r and a finite set of systems $\{S_1,\dots,S_n\}$ in A, denoted by $S \Rightarrow_A \langle r, \{S_1,\dots,S_n\} \rangle$), is defined by the following clauses.
1. $S+\{(\xi \to \alpha, \chi \to \beta)\} \Rightarrow_A \langle Id, \{S+\{(\xi, \chi), (\alpha, \beta)\}\} \rangle$;
2. $S+\{(\{\alpha_1;\dots;\alpha_n\}, \{\beta_1;\dots;\beta_m\}))\} \Rightarrow_A$
   $\langle Id, \{S+R \mid R$ is a minimally surjective relation on $\{\alpha_1,\dots,\alpha_n\} \times \{\beta_1,\dots,\beta_m\}\} \rangle$;
3. $S+\{(\alpha, \alpha)\} \Rightarrow_A \langle Id, \{S\} \rangle$;
4. Let $S = S'+\{(t, \alpha_1),\dots,(t, \alpha_n)\}$ where t does occur neither in α_i (for $1 \le i \le n$) nor in S. Then
   $S \Rightarrow_A \langle [t:=\alpha_1], \{S'+\{(\alpha_1, \alpha_2),\dots,(\alpha_1,\alpha_n)\}\} \rangle$
5. Let $S = S'+\{(\overset{l}{\forall}\underline{t}.\beta, \{\alpha_1;\dots;\alpha_n\}), (\overset{l}{\forall}\underline{t}.\beta, \xi_1),\dots,(\overset{l}{\forall}\underline{t}.\beta, \xi_m)\}$ where $\overset{l}{\forall}$ does not occur in S', α_i $(1 \le i \le n)$ and ξ_j $(1 \le i \le m)$. Then
   $S \Rightarrow_A \langle [\overset{l}{\forall}\underline{t}.\beta := \{\beta_1[\underline{u}_1/\underline{t}]),\dots,\beta_n[\underline{u}_n/\underline{t}]\}]$,
   $\{S+\{(\alpha_1, \beta_1[\underline{u}_1/\underline{t}]),\dots,(\alpha_n, \beta_n[\underline{u}_n/\underline{t}]), (\{\alpha_1;\dots;\alpha_n\}, \xi_1),\dots,(\{\alpha_1;\dots;\alpha_n\}, \xi_m)\}\} \rangle$

where $\underline{u}_1,\dots,\underline{u}_n$ are fresh variables and $\beta_1,\dots,\beta_n$ are obtained by renaming internal labels of β in the environment A according to Def. 3.3(i).

6. Let $S=S'+\{(\overset{m}{\forall}^{i}\,\underline{u}_i.\alpha_i,\overset{m}{\forall}^{j}\underline{u}_j.\alpha_j) \mid <i,j>\in P\}$ where there is a finite set I of indices such that $\{\overset{m}{\forall}^{i}\underline{u}_i.\alpha_i \mid i\in I\}\in \mathfrak{I}_S$ (i.e. is an equivalence class), $P\subseteq I\times I$ and for all $i\in I$ $\overset{m}{\forall}^{i}$ does not occur neither in S' nor in any α_j (for $j\in I$, $j\neq i$). Then

$S\Rightarrow_A<[\overset{m}{\forall}^{i}\underline{u}_i.\alpha_i:=\alpha_i[\underline{u}'_i/\underline{u}_i]|_{\underline{w}} \mid i\in I], \{S'+\{(\alpha_i[\underline{u}'_i/\underline{u}_i],\alpha_j[\underline{u}'_j/\underline{u}_j]) \mid <i,j>\in P\}\}>$

where $\underline{w}=\bigcup_{i\in I}\underline{u}'_i$ and $\underline{u}'_i$ (for $i\in I$) are fresh variables.

(ii) A system is *irreducible* if neither of the previous rules can be applied (i.e. if $\Rightarrow_A$ is not defined on S). □

Note that the six cases exhaust all the possible kinds of pairs (type,type) or (sequence,sequence). Rule 2 is the only one in which a set of systems containing more than one element is generated. This introduces an OR branch on the solution tree. All the other rules generate only one system . Rules 1, 2, and 3, moreover, generate a trivial transformation (the identity). They only simplify the system whereas the other rules solve one or more pairs.

Even though at any step there are many reductions that can be applied, the order in which the reductions are applied does not influence the final transformation produced by the system. The fact that many independent solutions can be found is due uniquely to rule 2.

In rule 2 we formalize the fact that a transformation, r, that solves $(\{\alpha_1;\dots;\alpha_n\},\{\beta_1;\dots;\beta_m\})$ $(n,m\geq 1)$ must be such that $r(\{\alpha_1;\dots;\alpha_n\})=r(\{\beta_1;\dots;\beta_m\})=\{\gamma_1;\dots;\gamma_j\}$ for some types $\gamma_1,\dots,\gamma_j$ $(1\leq j\leq \min(n,m))$. So some types in $r(\{\alpha_1;\dots;\alpha_n\})$ must be equal to some types in $r(\{\beta_1;\dots;\beta_m\})$ and must be equal to γ_1 and so on. There could be more than one independent solution of this pair. That is an r_1 and r_2 solving the pair and such that neither $r_1\leq r_2$ nor $r_2\leq r_1$. This is why a system can have a finite set (of cardinality bigger than one) of independent solutions.

A particular case of rule 2 (one intersection is empty) is

$S+\{(\{\},\{\beta_1;\dots;\beta_m\}))\}\Rightarrow_A<Id,\{S\}>$.

Rules 5 and 6 are applied when we need to unify pairs containing a labelled type $\overset{m}{\forall}\underline{t}.\alpha$. There are two possibilities. If $\mathfrak{I}(\overset{m}{\forall}\underline{t}.\alpha)$ contains at least one explicit intersection $\{\alpha_1;\dots;\alpha_n\}$) there is a pair $(\overset{1}{\forall}\underline{t}.\beta,\{\alpha_1;\dots;\alpha_n\})$ in the system (case 5) where $\overset{1}{\forall}\underline{t}.\beta\in\mathfrak{I}(\overset{m}{\forall}\underline{t}.\alpha)$. Then the number of instances that must be obtained from $\overset{1}{\forall}\underline{t}.\beta$ must be (not more than) n and we are reduced to matching n instances of $\overset{1}{\forall}\underline{t}.\beta$ with $\alpha_1,\dots,\alpha_n$. Moreover we replace all other occurrences of $\overset{1}{\forall}\underline{t}.\beta$ with $\{\alpha_1;\dots;\alpha_n\}$, thus eliminating the label 1 from the system. On the other hand if $\mathfrak{I}(\overset{m}{\forall}\underline{t}.\alpha)$ contains only intersections which are quantified types (case 6) then there is no reason to fix the number of instances we have to do. We simply force the body of all the equivalent types to match. In the final transformation we will restore the labelled quantification with respect to all generic type variables which occur in the body. This corresponds to apply an expansion of the kind of Definition 3.3(i)-2.

A nonempty system cannot be reduced only if it has no solutions

Lemma 4.6 If $S\neq\varnothing$ and S is irreducible then S has no solution. □

The set of solutions of a system can be obtained by iterated reductions. The notion of (iterated) system reduction is based on the one-step reduction of Def. 4.5, but is defined as a relation between a system and a pair of a system and a tranformation.

Definition 4.7. Let S be a system. Define $S \Longrightarrow_A \langle r', S' \rangle$ (S produces S' and r) if $S \Rightarrow_A \langle r, \{S_1, \ldots, S_n\} \rangle$ and
either $r = r'$ and $S' = S_i$ for some $1 \leq i \leq n$
or $S_i \Longrightarrow_{r(A)} \langle r'', S' \rangle$ for some $1 \leq i \leq n$ and $r' = r'' \circ r$. □

Indexed types can be removed by transformations by means of the operation of closure. The *closure* of of a transformation r , Cl(r), is the transformation obtained by replacing each occurrence of an indexed type $\alpha|_{\underline{u}}$ in the range of r by $\overset{l}{\forall}\underline{u}'.\alpha$ where l is a fresh label and $\underline{u}'$ is the subset of the variables in $\underline{u}$ that are free in α. A transformation r is *closed* if it has no occurrences of indexed types.

By a simlpe transfinite induction on a suitable measure of the complexity of the system we can prove that the relation ==> is strongly terminating, i.e. that each possible reduction of a system S ends producing either $\varnothing$ or an irreducible system.

Lemma 4.8. $\Longrightarrow_A$ is strongly terminating, i.e. there cannot be an infinite chain of reductions. □

Definition 4.9. Let S be a system in a environment A. Define $\mathfrak{R}(S,A)$ (the set of solutions of S in A) as

$\mathfrak{R}(S,A) = \{Cl(r) \mid S \Longrightarrow_A \langle r, \varnothing \rangle\}$. □

The correctness and completeness of the reduction procedure can then be proved by induction on the number of steps using the following Theorem. This also proves that any system S of equations that has a solution has a finite principal set of solutions.

Theorem 4.10 (i) Let $S \Rightarrow_A \langle r, \{S_1, \ldots, S_n\} \rangle$, r' is a solution of S if there is an integer i ($1 \leq i \leq n$) and a solution r'' of S_i such that $r' = Cl(r'' \circ r)$.
(ii) $\mathfrak{R}(S,A)$ is a principal set of solutions of S in the environment A. □

The set $\mathfrak{R}(S,A)$ can then be systematically produced using the following algorithm.

Definition 4.11 (*Solution Procedure*) Let S be the system to be solved in the environment A. Start up a tree with the root labelled by the pair <S,A>. Given any node, labelled by <S',A'>, there are two possibilities

1. the node is terminal in which case S' is
   either empty (indicating success),
   or an irreducile system (indicating failure):

2. let $S' \Rightarrow_{A'} <r,\{S_1,\dots,S_n\}>$. S' has n children labelled by $<S_1,r(A')>,\dots,<S_n,r(A')>$, and the branches from $<S',A'>$ to the $<S_i,A_i>$ are labelled by r.

By lemma 4.8 this construction produces a finite tree. The solutions of the system can be found composing the transformations on paths from the root to success nodes. □

Example 4.12 Consider the solution of the following system $S_0=\{(\overset{1}{\forall}t.(\{t\}\to t)\to u, \{z;\{z\}\to v\}\to v)\}$ in the environment $\emptyset$. This system must be solved to find the principal type of $((\lambda x.xI)\Delta)$. The tree we obtain is as follows (the environment is always empty and has been omitted):

$$\begin{tikzcd}
\boxed{\{(\overset{1}{\forall}t.(\{t\}\to t)\to u,\ \{z;\{z\}\to v\}\to v)\}} \arrow[d, "Id"] \\
\boxed{\{(\overset{1}{\forall}t.\{t\}\to t,\{z;\{z\}\to v\}),(u,v)\}} \arrow[d, "{[\overset{1}{\forall}t.\{t\}\to t:=\{\{t_1\}\to t_1,\{t_2\}\to t_2\}]\ =r_1}"] \\
\boxed{\{(\{t_1\}\to t_1,z),\ (\{t_2\}\to t_2,\{z\}\to v),\ (u,v)\}} \arrow[d, "Id"] \\
\boxed{\{(\{t_1\}\to t_1,z),\ (\{t_2\},\{z\}),\ (t_2,v),\ (u,v)\}} \arrow[d, "{[v:=u]\ =r_2}"] \\
\boxed{\{(\{t_1\}\to t_1,z),\ (t_2,z),\ (t_2,u)\}} \arrow[d, "{[t_2:=z]\ =r_3}"] \\
\boxed{\{(\{t_1\}\to t_1,z),\ (z,u)\}} \arrow[d, "{[z:=u]\ =r_4}"] \\
\boxed{\{(\{t_1\}\to t_1,u)\}} \arrow[d, "{[u:=\{t_1\}\to t_1]\ =r_5}"] \\
\boxed{\emptyset}
\end{tikzcd}$$

Hence the principal set of solutions for S contains only one element, i.e.

$\mathfrak{R}(S_0,\emptyset) = \{r_5\circ r_4\circ r_3\circ r_2\circ r_1\}$

$= \{[u:=\{t_1\}\to t_1, z:=\{t_1\}\to t_1, t_2:=\{t_1\}\to t_1, v:=\{t_1\}\to t_1,$
$\overset{1}{\forall}t.\{t\}\to t:=\{\{t_1\}\to t_1;\{\{t_1\}\to t_1\}\to\{t_1\}\to t_1]\}$. □

Property 4.8(i) assure that the set of solutions found with the previous procedure is principal.

We now give the type-inference algorithm for the system $\vdash_L$. Let **B** denote the set of bases. We will define a function

$T:\Lambda\text{->}\mathbf{P}_F(\mathbf{B}\times T_L)$ (where $\mathbf{P}_F$ is the finite power domain) which applied to a term M gives a finite set of pairs basis-type which is intended to be its set of principal pairs.

Definition 4.13. (*Type-inference Algorithm*). The function $\mathbf{T}:\Lambda\text{->}\mathbf{B}\times\mathrm{T_L}$ is defined inductively on Λ in the following way.

```
- T(x) = {<{x:t},t>    where t is a fresh type variable
- T(λx.M) = {<B,{α_1;...;α_n}→β> | {<B∪{x:α_1,...,x:α_n},β>}∈T(M)}.
- T(MN) = for all <B_M, α>∈T(M)
          for all <B_N, β>∈T(N)
          let ξ=∀^l t..β where t is the set of variables free in β and not in B_N
                                                  and l is a new label
          in
          if α≡t then {<[t:=ξ→u](B_M)∪B_N , u >} where u is a fresh variable
          if α≡χ→γ then {<r(B_M∪B_N), r{γ}> | r∈ℜ({(ξ,χ)},B_M∪B_N)}
                                      where r(B) = {x:r(α)| x:α∈B}. □
```

If $\langle B,\alpha\rangle\in \mathbf{T}(M)$ we say that $\langle B,\alpha\rangle$ is a *principal pair* for M, α is a *principal type* and B a *pricipal basis* for it.

For terms in normal form, for instance, the principal type is the one defined in the proof of Lemma 2.7 with the only difference that sequences of the shape $\{\alpha\}$ are replaced by $\overset{l}{\forall}\underline{t}.\alpha$ where l is a new label and $\underline{t}$ are all the type variables generic in α. The type checking algorithm is sound and complete for the system $\vdash_L$, in the following sense.

Lemma 4.14 (i) (*Soundness*) For all $\langle B,\alpha\rangle\in \mathbf{T}(M)$, $B\vdash_L M:\alpha$.
(ii) (*Completeness*) If $B\vdash_L M:\gamma$ for some type γ, then there exists $\langle B_p,\alpha\rangle\in \mathbf{T}(M)$ such that for some transformation r, $r(B_p)=B$ and $r(\alpha)=\gamma$. □

The previous soundness and completeness with respect to the system $\vdash_L$ immediately imply the result we are ultimately interested in, i.e. the existence of a sound a complete type inference algorithm for the system $\vdash_\wedge$. Each deduction $\vdash_\wedge$ in fact is also a deduction in $\vdash_L$. Indeed the system $\vdash_L$ has only been a tool to this end. But remark that principal types (and principal bases) for M are indeed types in $\mathrm{T_L}$ rather than in T.

Main Theorem. (i) (*Soundness*) For all $\langle B,\alpha\rangle\in \mathbf{T}(M)$, $B'\vdash_\wedge M:\alpha'$ where B' and α' are any terminal expansion of B and α.
(ii) (*Completeness*) If $B\vdash_\wedge M:\gamma$ for some type γ, then there exists $\langle B_p,\alpha\rangle\in \mathbf{T}(M)$ such that for some terminal transformation r, $r(B_p)=B$ and $r(\alpha)=\gamma$. □

As already mentioned when matching an intersection with an intersection we can get a finite number of principal solutions. This case however does not seem to occur often in the usual programming practice. An example of a term which has three principal types is the following.

Take **twice** = $\lambda f.\lambda x.f(f\ x)$ and **four** = $\lambda f.\lambda x.f(f(f(f\ x)))$ whose principal types are, respectvely,

$\tau_{twice}=\{a_1\rightarrow b_1;b_1\rightarrow c_1\}\rightarrow\{a_1\}\rightarrow c_1$ and

$\tau_{four}=\{a_2\rightarrow b_2;b_2\rightarrow c_2;c_2\rightarrow d_2;d_2\rightarrow e_2\}\rightarrow\{a_2\}\rightarrow e_2$.

Consider the term $T=\lambda z.\lambda y.((\lambda x.y\ (x\ \textbf{twice})\ (x\ \textbf{four}))\ (N\ z))$ where $N=(\Delta\ \textbf{twice})$.
The principal type of N is $\{\{t\}\rightarrow t\}\rightarrow\{t\}\rightarrow t$ while the (unique) type assigned to $(\lambda x.y\ (x\ \textbf{twice})\ (x\ \textbf{four}))$ by the type inference algorithm is $\{\xi_{twice}\rightarrow a;\xi_{four}\rightarrow b\}\rightarrow d$ (where $\xi_{twice}=\forall a_1b_1c_1.\tau_{twice}$ and $\xi_{four}=\forall a_2b_2c_2d_2e_2.\tau_{four}$) with respect to the basis $\{y:a\rightarrow b\rightarrow c,\ z:t\rightarrow t\}$. Now to apply $\lambda x.y\ (x\ \textbf{twice})\ (x\ \textbf{four})$ to $(N\ z)$ (whose type is $\{t\}\rightarrow t$) we have to solve the system $\{(\{\xi_{twice}\rightarrow a;\ \xi_{four}\rightarrow b\},\ \{t\rightarrow t\})\}$. This requires the unification of τ_{twice} and τ_{four} which, in its turn, requires the unification of the pair $(\{a_1\rightarrow b_1;b_1\rightarrow c_1\},\{a_2\rightarrow b_2;b_2\rightarrow c_2;c_2\rightarrow d_2;d_2\rightarrow e_2\})$ which has three different unifiers (none of the intersections, in fact, has been obtained by $(\forall E)$). The three unifiers are
-$[c_1:=b_1,a_2:=a_1,b_2:=b_1,c_2:=b_1,d_2:=b_1,e_2:=b_1]$, which yields $\{a_1\rightarrow b_1;b_1\rightarrow b_1\}$,
-$[b_1:=a_1,a_2:=a_1,b_2:=a_1,c_2:=a_1,d_2:=a_1,e_2:=c_1]$, which yields $\{a_1\rightarrow a_1;a_1\rightarrow c_1\}$ and
-$[c_1:=a_1,a_2:=a_1,b_2:=b_1,c_2:=a_1,d_2:=b_1,e_2:=a_1]$ which yields $\{a_1\rightarrow b_1;b_1\rightarrow a_1\}$.
Note that none of the solutions is more general than the other.
The principal types of T are then $\{\{\tau_i\}\rightarrow\tau_i\}\rightarrow\{\{\tau_i\}\rightarrow\{\tau_i\}\rightarrow c\}\rightarrow c$ for i=1,2,3 where

– $\tau_1=\{a_1\rightarrow b_1;b_1\rightarrow b_1\}\rightarrow\{a_1\}\rightarrow b_1$

– $\tau_2=\{a_1\rightarrow a_1;a_1\rightarrow c_1\}\rightarrow\{a_1\}\rightarrow c_1$

– $\tau_i=\{a_1\rightarrow b_1;b_1\rightarrow b_1\}\rightarrow\{a_1\}\rightarrow b_1$

References

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

[Coppo, Dezani, 1980] Coppo M., Dezani-Ciancaglini M.: An extension of basic functionality theory for lambda-calculus, Notre Dame J. Formal Logic **21**(4), 685-693.

[Coppo et. al., 1980] Coppo M., Dezani-Ciancaglini M., Venneri B.: Principal type schemes and lambda calculus semantics, in To H.B.Curry: essays on combinatory logic, lambda-calculus and formalism, J. Seldin and R. Hindley eds., Academic Press 1980, 536-560.

[Coppo et al., 1981] Coppo M., Dezani-Ciancaglini M., Venneri B.: Functional characters of solvable terms, Zeit. Math. Logik und Grund. Math. **27**(1981) , 45-58.

[Curry, Feys, 1958] Curry, H.B.; Feys, R.: Combinatory Logic, I, North-Holland, Amsterdam, 1958.

[Damas, Milner, 1982] Damas,L.M.M.; Milner,R.: Principal type schemes for functional programs, 9th ACM Symposium on Principles of Programming Languages, ACM, 1982, pp. 207 - 212.

[Giannini, Ronchi, 1988] Giannini P., Ronchi della Rocca S. Characterization of typings in polymorphic type discipline. In Logic in Computer Science, 1988, pp.61 - 70.

[Giannini, Ronchi, 1991] Giannini P., Ronchi della Rocca S. Type inference in polymorphic type discipline. In Theoretical aspects of Computer Science 1991, LNCS 526, 18-37.

[Girard,1971] Girard, J.Y.: Interpretation fonctionelle et elimination des coupures dans l'arithmetique d'ordre superieur, These de Doctorat d'Etat, Paris VII 1971.

[Gordon,Milner,Wadsworth,1979] Gordon M.J.C.; Milner, R.; Wadsworth, C.P.: Edinburgh LCF, Springer LNCS **78**, 1979

[Hindley, 1969] Hindley,J.R.: The principal type scheme of an object in combinatory logic, Trans. American Math. Soc., **146**,1969, pp. 29 - 60.

[Hindley, Seldin (eds.),1980] Hindley R., Seldin, J. (eds.): To H.B. Curry: Essays in Combinatory Logic, Lambda calculus and formalism, Academic Press, 1980.

[Hindley,Seldin, 1986] Hindley,J.R.;Seldin,J.P.: Introduction to Combinators and λ-Calculus, London Mathematical Society Student Texts 1, Cambridge University Press, London,1986

[Huet, 1976] Huet, G.: Resolution d'equations dans les langages d' ordre 1, 2, . . .,ω, These d'Etat, Universite' Paris VII, 1976.

[Kfoury, et.al., 1989] Kfoury A. J., Tyurin J., Urzyczyn P.: Computational consequences and partial solutions of a generalized unification problem. In Logic in Computer Science, pp.98 - 105., 1989.

[Kfoury, Tyurin, 1990] Kfoury A. J., Tyurin J.: Type Reconstruction in finite-rank fragments of the polymorphic λ-calculus . In Proc. of Logic in Computer Science '90, pp. 2-11.

[Leivant, 1986] Leivant D. Typing and computational properties of lambda-expressions, Theoretical Computer Science, **44** (1986), pp. 51-68.

[Mycroft, 1984] Mycroft A.: Polymorphic type schemes and recursive defintions. In International Symposium on Programming, LNCS 167, pp. 217-228.

[Milner, 1978] Milner, R.: A theory of type polimorphism in programming, J. Comput. System Sci., **17**, 1978, pp. 348 - 375

[Ronchi , Venneri, 1984] Ronchi della Rocca S., Venneri B. Principal Type Schemes for an extended type Theory Theoretical Computer Science **28** (1984), pp. 151-169.

[Turner, 1985] Turner, D.A.: Miranda: a non-strict functional language with polymorphic types, Proceedings of the IFIP International Conference on Functional Programming Languages and Computer Architecture, Springer LNCS **201**, 1985, pp. 1-16.

[van Bakel, 1991] van Bakel, S.: Complete restrictions of the intersection type discipline, to appear in Theoretical Computer Science 99 (1992).