Type Inference with Recursive Type Equations

Mario Coppo

Dipartimento di Informatica, Università di Torino,
Corso Svizzera 185, 10149 Torino, Italy
coppo@di.unito.it
http://www.di.unito.it/∼coppo

Abstract. This paper discusses decidability and existence of principal types in type assignment systems for λ-terms in which types are considered modulo an equivalence relation induced by a set of type equations. There are two interesting ways of defining such equivalence, an initial and a final one. A suitable transformation will allow to treat both in an uniform way.

Keywords: Recursive types, type assignment.

1 Introduction

In this paper we study type inference problems in the basic type inference system for the λ-calculus ([5], [8]) extended with recursive type definitions. For example, assuming the existence of a type constant *nil* for the one element type and of type constructors + for disjoint union and $\times$ for cartesian product, the type of integer lists can be specified by the equation

$$int\text{–}list = nil + (int \times int\text{–}list)$$

An alternative way of denoting the same type is via the use of an explicit operator μ or recursion over types. In this case the former type is denoted by $\mu t.nil + (int \times t)$. Although the use of recursive equations is closer to the way in which recursive types are introduced in programming languages like ML ([9]) most of the technical literature on the subject has been focused on the μ-notation (see e.g. [12]). Although the two notations are equivalent in many contexts, there are interesting aspects in which they are not such. Taking recursive definitions we specify in advance a finite number of recursive types that we can use for typing a term, whereas taking μ-types we are allowed to use arbitrary recursive types. The introduction of recursive types via recursive definitions rises questions about typability of terms of the kind:
- Given a term M and a set $\mathcal{R}$ of recursive definitions can M be typed from R?
- Can the notion of principal type scheme ([8]) be generalized to type assignment with respect to $\mathcal{R}$?

While some general results about typability of terms using μ-types are well established in the literature (see for instance [3]) there seems to be much less about recursive definitions.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 184–198, 2001.

Following [7], we start by considering the notion of "type structure" which is a set T of types equipped with a congruence relation $\simeq$ with respect to the $\rightarrow$ type constructor. Type structures are usually defined starting from a set of equations between types of the shape $B = C$, where B and C are simple type expressions. There are two basic ways of defining an equivalence relation from a set of type equations: an initial one in which equivalence is defined via simple equational reasoning and a final one which amounts to consider equivalent two types if they have the same interpretation as infinite trees. The latter is, for instance, the notion of type equivalence determined by the type checking algorithms of most programming languages, starting from ALGOL60. Tree equivalence is also the notion of equality induced by the interpretations of types in continuous models (see [3]). We will define a transformation to reduce tree equivalence to the equational one. This allows to treat both equivalences in an uniform way.

In this paper we take $\rightarrow$ as the only type constructor, but this is enough to define quite interesting type assignments. For example it is well known that, assuming a type c such that $c = c \rightarrow A$ (where A is any type) one can assign type $(A \rightarrow A) \rightarrow A$ to the fixed point combinator $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, permitting to type recursive functions over values without assuming an "ad-hoc" constant for this. Other type constructors, as those used in the introductory example, can be added easily.

The basic definitions and properties about type structures and type equalities are given in sections 2 while type assignment is introduced in section 3. Section 4 is about some basic properties of type equivalence. The results concerning type assignment are given in section 5. The basic source for this paper is [11] in which some basic properties of recursive definition are established. Type inference, however, is not discussed in that paper.

2 Type Structures and Type Constraints

Following [7], we start by defining the notion of type structure. This notion was first motivated by Scott [10] and formally developed in Breazu Tannen and Meyer [2].

Type Structures

The main feature of recursive types is that one makes identifications between them. In a type structure types are taken modulo a congruence relation.

Definition 1. *Let* T *be a set of syntactic objects (types) closed under the* $\rightarrow$ *type constructor. A* type structure *over* T *is a pair* $\mathcal{T} = \langle \mathsf{T}, \simeq \rangle$ *where* $\simeq$ *is a congruence over* T *(i.e. an equivalence relation such that* $A \simeq A'$ *and* $B \simeq B'$ *implies* $A \rightarrow A' \simeq B \rightarrow B'$*).*

We mainly consider a set T of types built from a denumerable set $\mathbf{A}$ of *atomic* types by the $\rightarrow$ type constructor[1]. We will sometime write $\mathsf{T}(\mathbf{A})$ to make explicit

[1] There are other interesting sets of types that naturally build a type structure like that of μ-types defined in the introduction.

the set of atomic types we have started from. The notion of atomic type here is only a syntactic one: an atomic type can be equivalent via $\simeq$ to a non atomic one.

The set T of simple types is isomorphic to *free* type structure $\langle \mathsf{T}, \equiv \rangle$, that will be identified with T itself.

Definition 2.
(i) *A mapping* $h : \mathsf{T} \to \mathsf{T}$ *is a* type homomorphism *if* $h(A \to B) = h(A) \to h(B)$.
(ii) *Let* $\mathcal{T} = \langle \mathsf{T}, \simeq \rangle$ *and* $\mathcal{T}' = \langle \mathsf{T}, \simeq' \rangle$ *be type structures. A type homomorphism* $h : \mathsf{T} \to \mathsf{T}$ *is a* type structure homomorphism *(written also* $h : \mathcal{T} \to \mathcal{T}'$*) if* $A \simeq B \ \Rightarrow \ h(A) \simeq' h(B)$.

Remark 1. Note that if $\mathcal{T} = \langle \mathsf{T}(\mathbf{A}), \simeq \rangle$, any type homomorphism $h : \mathcal{T} \to \mathcal{T}'$, where $\mathcal{T}'$ is any other type structure, is uniquely determined by its restriction on the atomic types in $\mathbf{A}$. We will then define an homomorphism h by giving only its value on the atomic types.

A substitution (in the usual sense) is a type homomorphism. A substitution is a homomorphism between free type structures but not, in general, between arbitrary ones. We denote with $[a_1 := A_1, \ldots, a_n := A_n]$ the type homomorphism $h : \mathsf{T} \to \mathsf{T}$ determined by $h(a_1) = A_1, h(a_n) = A_n$ and $h(a) = a$ for all other atomic types $a \notin \{a_1, \ldots, a_n\}$. In this case e say that h is a *finite* type homomorphism.

Type structure homomorphisms are closed under composition (indeed type structures with homomorphisms form a category).

Definition 3 (Invertibility). *Let* $\langle \mathsf{T}, \simeq \rangle$ *be a type structure. Then* $\simeq$ *is said to be* invertible *if* $(A \to B) \simeq (A' \to B') \ \Rightarrow \ A \simeq A'$ *and* $B \simeq B'$.
We also say that the relation $\simeq$ *over* T *is* invertible.

Invertibility holds, for instance, in a free type structure.

Type Constraints and Recursive Definitions

A very natural way of generating type structures is to assume a (finite) set of equations between types and to take the congruence generated by it via equational reasoning. A stronger way of generating a congruence between types (interpreting them in the domain of trees) will be considered in the next section.

Definition 4. *A* system of type constraints *over* T *is a set* $\mathcal{C} = \{A_i = B_i \mid 1 \leq i \leq n\}$ *of formal equations between types in* T.

A system of type constraints determines, via standard equational reasoning, a congruence between types.

Definition 5. *Let* $\mathcal{C} = \{A_i = B_i \mid 1 \leq i \leq n\}$*, be a system of type constrains over* T*. The congruence relation determined by* $\mathcal{C}$ *(notation* $=_{\mathcal{C}}$*) is the least relation satisfying the following axioms and rules:*

$$(eq)\quad \vdash A =_{\mathcal{C}} B \ \text{ if } \ A = B \in \mathcal{C} \qquad (ident)\ \vdash A =_{\mathcal{C}} A$$

$$(symm)\ \frac{\vdash A =_{\mathcal{C}} B}{\vdash B =_{\mathcal{C}} A} \qquad (trans)\ \frac{\vdash A =_{\mathcal{C}} B \qquad \vdash B =_{\mathcal{C}} C}{\vdash A =_{\mathcal{C}} C}$$

$$(\rightarrow)\ \frac{\vdash A =_{\mathcal{C}} A' \qquad \vdash B =_{\mathcal{C}} B'}{\vdash A \rightarrow B =_{\mathcal{C}} A' \rightarrow B'}$$

We call $=_{\mathcal{C}}$ *the* equational *equivalence over* T *induced by* $\mathcal{C}$*. Let* $\mathcal{T}_{\mathcal{C}}$ *do denote the type structure* $\langle \mathsf{T}, =_{\mathcal{C}} \rangle$*.*

The relation $=_{\mathcal{C}}$ is the minimal congruence over T generated by the equations in $\mathcal{C}$. A basic related notion is the following.

Definition 6. *A type structure* $\mathcal{T} = \langle \mathsf{T}', \simeq \rangle$ solves *a system of type constraints* $\mathcal{C}$ *over* T *if there is a type structure homomorphism* $h : \mathcal{T}_{\mathcal{C}} \rightarrow \mathcal{T}$*. We also say that* h solves $\mathcal{C}$ *in* $\mathcal{T}$*.*

Note that h solves $\mathcal{C}$ in $\mathcal{T} = \langle \mathsf{T}', \simeq \rangle$ iff $h(A) \simeq h(B)$ for all $A = B \ \in \ \mathcal{C}$.

In the definition of recursive types we are particularly interested in the type structures generated by a system of type constraints of the shape $c = C$ where c is an atom and C is a non atomic type expression. This corresponds to the idea of defining type c as equivalent to a type expression C containing possibly c itself. More formally:

Definition 7 (Simultaneous recursion).

(i) A system of type constraints $\mathcal{R}$ *over* T *is a* simultaneous recursion *(s.r. for short) if it has the form* $\mathcal{R} = \{c_i = C_i \mid 1 \leq i \leq n\}$*. where*

1. *for all* $1 \leq i \leq n$*,* c_i *is atomic and* $C_i \in \mathsf{T}$*;*
2. $c_i \not\equiv c_j$ *for all* $i \neq j$*;*
3. *if* $C_i \equiv c_j$ *for some* $1 \leq i, j \leq n$ *then* $i < j$*.*

We call $\{c_1, \ldots, c_n\}$ *the* domain *of* $\mathcal{R}$*, and denote it by* $dom(\mathcal{R})$*. The atoms* $c_1, \ldots, c_n$ *are called the* indeterminates *of the s.r..*

(ii) A s.r. is proper *if* C_i *is not atomic for all* $1 \leq i \leq n$*.*

Example 1. Let $\mathcal{R}_1 = \{c_1 = t \rightarrow c_1\}$ where t is a type variable is a s.r. defining a type c such that $c_1 =_{\mathcal{R}_1} t \rightarrow c_1 =_{\mathcal{R}_1} t \rightarrow t \rightarrow c_1 \ldots$ and so on. Define now $\mathcal{R}_2 = \{c_2 = t \rightarrow t \rightarrow c_2\}$. Note that $c_2 \neq_{\mathcal{R}_2} t \rightarrow c_2$. Indeed if we take $\mathcal{R}_3 = \mathcal{R}_1 \cup \mathcal{R}_2$ we have that $c_1 \neq_{\mathcal{R}_3} c_2$.

In general an s.r. satisfies point 3. above if it can be defined a total order $<$ on its domain such that if there is an equation $c = c' \in \mathcal{R}$ then $c < c'$. This restriction is introduced to rule out circular equations like $c = c$ or $c_1 = c_2, c_2 =$

c_1 which have no interesting meaning. Notice that in a s.r. we can eliminate any equation of the shape $c_i = c_j$ $(i \neq j)$ simply by replacing c_i by c_j in $\mathcal{R}$ (or vice versa).

We call *unfolding* the operation of replacing c_i by C_i in a type and *folding* the reverse operation. So, two types are weakly equivalent w.r.t. $\mathcal{C}$ if they can be transformed one into the other by a finite number of applications of the operations folding and unfolding.

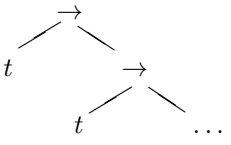
Definition 8. *Let $\mathcal{R}$ be a s.r. over T. The* kernel *of $\mathcal{R}$ is the set $ker(\mathcal{R}) \subseteq \mathcal{R}$ containing all equations $c = C \in \mathcal{R}$ such that $c =_{\mathcal{R}} A$ for some non atomic type A containing at least one occurrence in c.*

It is a simple exercise to design an algorithm to test whether an indeterminate c is in the kernel of $\mathcal{R}$. The kernel of $\mathcal{R}$ determines essentially the set of new types defined via $\mathcal{R}$. An equation $c = C$ which is not in $ker(\mathcal{R})$ is simply a way of giving "name" c to type C, but is uninteresting from our point of view.

The Tree Equivalence of Types

Consider Example 1 above. Types like c_1 and c_2, although not equivalent with respect to $=_{\mathcal{R}_3}$, seem to express the same informal behaviour. Indeed c_1 and c_2 represent to the same (infinite) type when the they are "pushed down" by repeated steps of unfolding. Another notion of equivalence between recursive type expressions can be defined by regarding them as finitary descriptions of infinite trees: this approach is followed systematically in [3] and is indeed the most popular way of considering type equivalence in programming languages ([12]).

For the basic notions about infinite and regular trees we refer to [4]). Recall that an infinite trees is *regular* if it has only a finite number of subtrees. An example of regular tree is the following tree α_0:



Indeed α_0 has only two subtrees, t and α_0 itself.

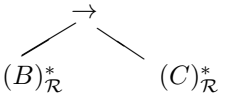
Let $\mathsf{Tr}^R(\mathbf{A})$ denote the set of regular trees defined starting from a set $\mathbf{A}$ of atomic types ($\mathbf{A}$ will be omitted when understood). Let $\mathsf{Tr}^F(\mathbf{A}) \subseteq \mathsf{Tr}^R(\mathbf{A})$ be the set of *finite* trees. $\mathsf{Tr}^F(\mathbf{A})$ and $\mathsf{T}(\mathbf{A})$ are isomorphic and will be identified.

Note that $\langle \mathsf{Tr}^R, \equiv \rangle$ is a type structure, that we identify with Tr^R. It is immediate to verify that Tr^R is invertible.

There is a standard way of interpreting recursive types as infinite trees.

Definition 9. *Let $\mathcal{R}$ be a s.r. over T. Let $(-)^*_{\mathcal{R}} : \mathcal{T}_{\mathcal{R}} \to \mathsf{Tr}^R$ be the type structure homomorphism defined in the following way:*

1. *If $A \equiv a$ for some $a \in \mathbf{A}$ such that $a \notin dom(\mathcal{R})$ then $(A)^*_{\mathcal{R}}$ is a leaf a.*
2. *If $A \equiv B \to C$ for some $B, C \in \mathsf{T}$ then $(A)^*_{\mathcal{R}}$ is*



3. *If $A \equiv c$ for some $c \in dom(\mathcal{R})$ such that $c = C \in \mathcal{R}$ then $(c)^*_{\mathcal{R}} = (C)^*_{\mathcal{R}}$.*

Note that this is not an inductive definition but a co-inductive one since C in case 3. is in general more complex than c. Note that, owing to condition 3. of Definition 7, $(A)^*_{\mathcal{R}}$ is always well defined.

It is well known ([4]) that if $\mathcal{R}$ is finite $(A)^*_{\mathcal{R}}$ is a regular tree for all $A \in \mathsf{T}$. The map $(-)^*_{\mathcal{R}}$ induces the following notion of equivalence between types.

Definition 10. (i) *Let $\mathcal{R}$ be a s.r over T. Let $=^*_{\mathcal{R}}$ be the relation over T defined by $A =^*_{\mathcal{R}} B$ if $(A)^*_{\mathcal{R}} = (B)^*_{\mathcal{R}}$. We call $=^*_{\mathcal{R}}$ the* tree equivalence *over T induced by $\mathcal{R}$.*

(ii) *Let $\mathcal{T}^*_{\mathcal{R}} = \langle \mathsf{T}, =^*_{\mathcal{R}} \rangle$ be the type structure determined by $=^*_{\mathcal{R}}$.*

It is easy to see that $=^*_{\mathcal{R}}$ is a congruence with respect to $\to$ and hence that $\langle \mathsf{T}, =^*_{\mathcal{R}} \rangle$ is well defined and $=_{\mathcal{R}} \subseteq =^*_{\mathcal{R}}$. Then there is a (unique) homomorphism $\mathsf{i}^* : \mathcal{T}_{\mathcal{R}} \to \mathcal{T}^*_{\mathcal{R}}$ which coincides with the identity at the type level.

Referring to example 1 it is easy to see that $c_2 =^*_{\mathcal{R}_2} t \to c_2$ and that $c_1 =^*_{\mathcal{R}_3} c_2$.

Clearly $=^*_{\mathcal{R}}$ has a more semantic nature than $=_{\mathcal{R}}$ and is indeed the type equivalence induced by the interpretation of types in continuous models ([3]). Axiomatizations of $=^*_{\mathcal{R}}$ are usually given using some co-induction principle (see for instance [1]). However $=^*_{\mathcal{R}}$ can be also characterized as the equational theory of a suitable s.r. $\mathcal{R}^*$.

Theorem 1. *Let $\mathcal{R}$ be an s.r. over T. Then there is s.r. $\mathcal{R}^*$ such that:*

(i) *For all $A, B \in \mathsf{T}$ $A =^*_{\mathcal{R}} B$ iff $A =_{\mathcal{R}^*} B$.*

(ii) *$\mathcal{T}^*_{\mathcal{R}}$ is isomorphic to $\mathcal{T}_{\mathcal{R}^*}$.*

We can give here only an example of this construction. The complete proof will be contained in a forthcoming paper. Let $\mathcal{R} = \{c_1 = t \to t \to c_1\}$ we proceed through the following steps:

1. We first transform $\mathcal{R}$ in its *flat* form $\mathsf{f}(\mathcal{R})$, in which every equation is of the shape $c = a$ where $a \in \mathbf{A}$ or $c = d \to e$ where both c and d are indeterminates. This may require the introduction of new indeterminates. In the case of our example we have $\mathsf{f}(\mathcal{R}_1) = \{c_1 = d_1 \to d_2, d_1 = t, d_2 = d_3 \to c_1, d_3 = t\}$, where d_1, d_2, d_3 are new indeterminates.

2. Then define by an iterative construction a relation $\approxeq$ on the indeterminates of $\mathsf{f}(\mathcal{R})$ such that $a \approxeq b$ if and only if a and b have the same infinite unfolding (this is the crucial step of the construction). In our example we get $c_1 \approxeq d_2$ and $d_1 \approxeq d_3$.

3. Now apply to $\mathsf{f}(\mathcal{R})$ a transformation $\mathsf{c}(-)$ that identifies (by adding suitable equations) all the indeterminates that are equated by $\cong$. In our case we get $\mathsf{c}(\mathsf{f}(\mathcal{R})) = \{c_1 = d_1 \to c_1, d_2 = c_1, d_1 = t, d_3 = d_1\}$.

4. Lastly apply to $\mathsf{c}(\mathsf{f}(\mathcal{R}))$ a transformation $\mathsf{s}(-)$ which removes all the indeterminates introduced by $\mathsf{f}(-)$ in step 1. and define $\mathcal{R}^* = \mathsf{s}(\mathsf{c}(\mathsf{f}(\mathcal{R})))$. In our case we get $\mathcal{R}^* = \{c_1 = t \to c_1\}$.

It is easy to see that $\mathcal{R}^*$ has the same indeterminates of $\mathcal{R}$ and satisfies Theorem 1.

3 λ-Calculi with Recursive Types

Type assignment in a generic type structure $\mathcal{T} = \langle \mathsf{T}, \simeq \rangle$ is defined by introducing a rule to consider types modulo the equivalence determined by $\mathcal{T}$.

The judgements of the inference system are of the shape $\Gamma \vdash_{\mathcal{T}} M : A$ where Γ denotes a *type environment*, i.e. a set of assumptions of the shape $x : A$, where x is a term variable and A a type.

Definition 11. *Let $\mathcal{T} = \langle \mathsf{T}, \simeq \rangle$ be a Type structure and $A, B \in \mathsf{T}$. Then the inference rules for assigning types to λ-terms from $\mathcal{T}$ are the following:*

$$(ax)\ \Gamma, x : A \vdash_{\mathcal{T}} x : A \qquad (\to E)\ \frac{\Gamma \vdash_{\mathcal{T}} M : A \to B \qquad \Gamma \vdash_{\mathcal{T}} N : A}{\Gamma \vdash_{\mathcal{T}} (M\ N) : B}$$

$$(\to I)\ \frac{\Gamma, x : A \vdash_{\mathcal{T}} M : B}{\Gamma \vdash_{\mathcal{T}} \lambda x.M : A \to B} \qquad (equiv)\ \frac{\Gamma \vdash_{\mathcal{T}} M : A \quad A \simeq B}{\Gamma \vdash_{\mathcal{T}} M : B}$$

If $\mathcal{C}$ is a system of type constraints, we simply write $\Gamma \vdash_{\mathcal{C}} M : A$ for $\Gamma \vdash_{\mathcal{T}_{\mathcal{C}}} M : A$ and $\Gamma \vdash^*_{\mathcal{C}} M : A$ instead of $\Gamma \vdash_{\mathcal{T}^*_{\mathcal{C}}} M : A$ (in case $\mathcal{C}$ is a s.r.).

Type assignment in different type structures can be related via the notion of type structure homomorphism. The following property can easily be proved by induction on derivations.

Lemma 1. *Let $h : \mathcal{T} \to \mathcal{T}'$ be a type structure homomorphism. Then*

$$\Gamma \vdash_{\mathcal{T}} M : A \quad \Rightarrow \quad h(\Gamma) \vdash_{\mathcal{T}'} M : h(A)$$

where $h(\Gamma)$ is obtained by applying h to all type occurrences in Γ.

For example since there is a homomorphism $\mathsf{i}^* : \langle \mathsf{T}, =_{\mathcal{R}} \rangle \to \langle \mathsf{T}, =^*_{\mathcal{R}} \rangle$ we have that any term which can be typed in $\mathcal{T}_{\mathcal{R}}$ can also be typed (with the same types) in $\mathcal{T}^*_{\mathcal{R}}$.

A remarkable property of type assignment is subject reduction which states that typings of terms are preserved by β-reduction. The following property characterizing the type systems in which subject reduction hold has been proved by R. Statman in [11].

Theorem 2. *A type assignment system $\vdash_{\mathcal{T}}$ has the subject reduction property iff $\mathcal{T}$ is invertible.*

We will see in the next section that if $\mathcal{C}$ is a system of type constraints $=_{\mathcal{C}}$ is invertible iff $\mathcal{C}$ is equivalent to a s.r.. Then it is mostly interesting to study assignment in type structures determined by a s.r..

4 Some Properties of Simultaneous Recursions

In this section we state some properties of the equivalence between types induced by a s.r. or by a systems of type constraints. We are mainly concerned in studying the solvability of a system of type constraints in a s.r.. Many results of this section are essentially due to R. Statman ([11]).

The Term Rewriting System Associated to a S.R.

Definition 12. (i) *Let $\mathcal{C}$ and $\mathcal{C}'$ be two systems of type constraints over the same set $\mathsf{T}_{\mathbf{A}}$ of types. $\vdash_{\mathcal{C}} \mathcal{C}'$ means that $\vdash_{\mathcal{C}} A{=}B$ for all equations $A{=}B \in \mathcal{C}'$.*
(ii) *$\mathcal{C}$ and $\mathcal{C}'$ are* equivalent *(notation $\mathcal{C} \sim \mathcal{C}'$) if $\vdash_{\mathcal{C}} \mathcal{C}'$ and $\vdash_{\mathcal{C}'} \mathcal{C}$.*

Note that $\mathcal{C}$ is equivalent to $\mathcal{C}'$ if $=_{\mathcal{C}}$ and $=_{\mathcal{C}'}$ coincide.

Given a s.r. $\mathcal{R}$ we define now a Church-Rosser and strongly normalizing term rewriting system which generates $=_{\mathcal{R}}$. The construction is a slight modification of the original one given by Statman [11], to which we refer for the proofs. A more structured but less direct approach is given in Marz [7].

Definition 13. *Let $\mathcal{R} = \{c_i = C_i \mid 1 \leq i \leq n\}$ be an s.r. over T. The rewriting system Trs($\mathcal{R}$) contains a rewriting rules $C_i \rightsquigarrow c_i$ for all $c_i = C_i \in \mathcal{R}$.*

Note that $=_{\mathcal{R}}$ is the convertibility relation over $\mathsf{T} \times \mathsf{T}$ generated by Trs($\mathcal{R}$).

It is easy to see that Trs($\mathcal{R}$) is strongly normalizing, because each contraction either decreases the size of the type to which it is applied or is of the shape $c_j \rightsquigarrow c_i$ where $i < j$. However, $\rightsquigarrow$ it is not, in general, Church-Rosser.

Example 2. Let $\mathcal{R} = \{c_0 = c_0 \to c_2,\ c_1 = (c_0 \to c_2) \to c_2,\ c_2 = c_0 \to c_1\}$. Then Trs($\mathcal{R}$) consists of the rules $\{c_0 \to c_2 \rightsquigarrow c_0,\ (c_0 \to c_2) \to c_2 \rightsquigarrow c_1,\ c_0 \to c_1 \rightsquigarrow c_2\}$. Observe that the l.h.s. of the first equation is a subterm of the l.h.s. of the second one. In particular $(c_0 \to c_2) \to c_2$ can be reduced both to c_1 and to $c_0 \to c_2$ which further reduces to c_0: it has then two distinct normal forms c_1 and c_0. Therefore Trs($\mathcal{R}$) is not confluent.

Expressions like $c_0 \to c_2$ and $(c_0 \to c_2) \to c_2$ (called *critical pairs* in the literature on term rewriting systems [6]) destroy confluence. By a well known result of Knuth and Bendix (see [6, Corollary 2.4.14]) a strongly normalizing term rewriting system without critical pairs is Church-Rosser.

The following algorithm, which amounts to Knuth-Bendix completion (see [6]), transforms any s.r. into an equivalent one without critical pairs which has, therefore, the Church-Rosser property.

Definition 14 (Completion of $\mathcal{R}$).
(i) *Let $\mathcal{R}$ be a s.r.. We define by induction on n a sequence of s.r. $\mathcal{R}_n$ ($n \geq 0$). Let $\mathcal{R}_0 = \mathcal{R}$. Define $\mathcal{R}_{n+1}$ from $\mathcal{R}_n$ ($n \geq 0$) in the following way:*

1. if there exists a pair of equations $c_i = C_i, c_j = C_j \in \mathcal{R}_n$ such that C_j is not atomic and is a proper subexpression of C_i take

$$\mathcal{R}_{n+1} = (\mathcal{R}_n - \{c_i = C_i\}) \cup \{c_i = C_i^*\}$$

where C_i^* is the result of replacing all occurrences of C_j in C_i by c_j.

2. If there exist two equations $c_i = C, c_j = C \in \mathcal{R}_n$ then, assuming $i < j$, take

$$\mathcal{R}_{n+1} = \mathcal{R}_n[c_i := c_j] \cup \{c_i = c_j\}.$$

3. otherwise take $\mathcal{R}_{n+1} = \mathcal{R}_n$.

(ii) *Let N be the least n such that $\mathcal{R}_{n+1} = \mathcal{R}_n$ (this value must certainly exist since, in both steps 1 and 2 the total number of symbols in $\mathcal{R}_n$ strictly decreases). Let $Trs^+(\mathcal{R}) = Trs(\mathcal{R}_N)$.*

It is easy to prove by induction on n that for all $n \geq 0$ $\mathcal{R}_n$ is a s.r. equivalent to $\mathcal{R}$. Then $\mathrm{Trs}(\mathcal{R}_N)$ has no critical pairs (by construction) and generates the same equality as $\mathcal{R}$.

Applying this construction to Example 2 get $N = 2$ and the resulting s.r. is $\mathcal{R}_2 = \{c_1 = c_0 \to c_2,\ c_2 = c_0 \to c_0,\ c_0 = c_1\}$ and then we get
$\mathrm{Trs}^+(\mathcal{R}) = \{c_0 \to c_2 \rightsquigarrow c_1,\ c_0 \to c_0 \rightsquigarrow c_2,\ c_1 \rightsquigarrow c_0\}$.

Lemma 2. *Let $\mathcal{R}$ be a s.r.. Then $Trs^+(\mathcal{R})$ is strongly normalizing and Church-Rosser.*

Corollary 1. *Let $\mathcal{R}$ be a s.r.. Then $=_{\mathcal{R}}$ is decidable.* □

Using $\mathrm{Trs}^+(\mathcal{R})$ it can be proved the invertibility of $=_{\mathcal{R}}$ (see [11] for details).

Theorem 3. *Let $\mathcal{R}$ be a s.r.. Then $=_{\mathcal{R}}$ is invertible.*

Solving a S.R. in Another

It is sometimes useful to force a type system generated by a set of type constraints $\mathcal{C}$ to have the invertibility property.

Definition 15. *Let $\mathcal{C}$ be a system of type constraints over $\mathsf{T}_{\mathbf{A}}$. The relation $=_{\mathcal{C}}^{\mathrm{inv}}$ is defined by adding to the axioms and rules of Definition 5 the following rule for invertibility*

$$(inv)\ \frac{\vdash A_1 \to A_2 =_{\mathcal{C}}^{\mathrm{inv}} B_1 \to B_2}{\vdash A_i =_{\mathcal{C}}^{\mathrm{inv}} B_i}\ (i = 1, 2)$$

Then $=_{\mathcal{C}}^{\mathrm{inv}}$ is the least invertible congruence containing $=_{\mathcal{C}}$.

If A is type expression its *length* (denoted $|A|$) is defined as the number of symbols occurring in it. The following construction is essentially taken from ([11]).

Definition 16. *Let $\mathcal{C}$ a system of type constraints. As in Definition 14 we define by induction on n a sequence of sets of equations $\mathcal{C}_n$ ($n \geq 0$). We can assume without less of generality that there is a total order between the atomic types occurring in $\mathcal{C}$, that will be denoted by $c_1, c_2, \ldots$). Let $\mathcal{C}_0 = \mathcal{C}$. Define $\mathcal{C}_{n+1}$ from $\mathcal{C}_n$ ($n \geq 0$) in the following way:*

1. If there is an equation $A \to B = C \to D \in \mathcal{C}_n$ then take

$$\mathcal{C}_{n+1} = \mathcal{C}_n - \{A \to B = A' \to B'\} \cup \{A = A', B = B'\};$$

2. If there are two equations $c = A \to B, c = A' \to B' \in \mathcal{C}_n$, assuming $|A \to B| \leq |A' \to B'|$, then take

$$\mathcal{C}_{n+1} = \mathcal{C}_n - \{c = A' \to B'\} \cup \{A = A', B = B'\};$$

3. Otherwise take $\mathcal{C}_{n+1} = \mathcal{C}_n$.

Let N be the least n such that $\mathcal{C}_{n+1} = \mathcal{C}_n$ (it is easy to prove that value must certainly exist). Note that we can write $\mathcal{C}_N = \mathcal{R} \cup \mathcal{E}$ where $\mathcal{R}$ contains all equations of the shape $c = C$ such that C is not atomic and $\mathcal{E}$ is a set of equations of the shape $c_i = c_j$ such that both c_i and c_j are atomic. Now take the equivalence classes of the atomic types occurring in $\mathcal{E}$ with respect to the $\mathcal{E}$ itself (seen as a set of type constraints). For each equivalence class c with respect to $\mathcal{E}$ with more than one element choose as representative the element $c_j \in \mathsf{c}$ with the greatest index, replace with c_j each occurrence of an element of c in $\mathcal{R}$ and add to $\mathcal{R}$ an equations $c = c_j$ for each element $c \in \mathsf{c}$ different from c_j.

Lastly define $\mathcal{C}^{\mathrm{inv}}$ as the so obtained s.r..

It is immediate to prove, by induction on n, that for all $n \geq 0$ $\mathcal{C}_n \vdash \mathcal{C}$ and $\mathcal{C} \vdash^{\mathrm{inv}} \mathcal{C}_n$. We have immediately the following result.

Lemma 3. *Let $\mathcal{C}$ a system of type constraints. Then $\mathcal{C} \vdash^{\mathrm{inv}} \mathcal{C}^{\mathrm{inv}}$ and $\mathcal{C}^{\mathrm{inv}} \vdash \mathcal{C}$ (i.e. $\mathcal{C}^{\mathrm{inv}}$ is equivalent to $\mathcal{C}$ plus invertibility).*

From Theorem 3 we have immediately that a system of type constraint is invertible iff it is equivalent to a s.r.

Let now $\mathcal{C}$ be a system of type constraints over T. We say that a s.r. $\mathcal{R}$ over T' *solves* $\mathcal{C}$ if $\langle \mathsf{T}', =_{\mathcal{R}} \rangle$ solves $\mathcal{C}$ (see Definition 6). Another immediate consequence of Theorem 3 is the following.

Lemma 4. *Let $\mathcal{C}$ be a system of type constraints and $\mathcal{R}$ a s.r.. Then $h : \langle \mathsf{T}, =_{\mathcal{C}} \rangle \to \langle \mathsf{T}, =_{\mathcal{R}} \rangle$ iff $h : \langle \mathsf{T}, =^{\mathrm{inv}}_{\mathcal{C}} \rangle \to \langle \mathsf{T}, =_{\mathcal{R}} \rangle$, i.e. $\mathcal{R}$ solves $\mathcal{C}$ iff $\mathcal{R}$ solves $\mathcal{C}^{\mathrm{inv}}$.*

Owing to lemma 4 we are mainly interested to study solvability of a s.r. into another one. The following lemma is an extension of a result proved in [11].

Lemma 5. *Let $\mathcal{R} = \{c_i = C_i \mid 1 \leq i \leq n\}$ be a s.r. over T and let $a_1, \ldots, a_p$ ($p \geq 0$) the other variables occurring in $ker(\mathcal{R})$. If $\mathcal{S}$ is another s.r. over T any homomorphism $h : \langle \mathsf{T}, =_{\mathcal{R}} \rangle \to \langle \mathsf{T}, =_{\mathcal{S}} \rangle$ (solving $\mathcal{R}$ in $\mathcal{S}$) can be written as*

$$h = \overline{\overline{h}} \circ \overline{h}$$

where $\overline{h} : \langle \mathsf{T}, =_{\mathcal{R}} \rangle \to \langle \mathsf{T}, =_{\mathcal{S}} \rangle$ and $\overline{\overline{h}} : \langle \mathsf{T}, =_{\mathcal{S}} \rangle \to \langle \mathsf{T}, =_{\mathcal{S}} \rangle$ are such that:

1. $\overline{h} = [c_1 := B_1, \ldots, c_n := B_n, a_1 := A_1, \ldots, a_p := A_p]$ *is a finite homomorphism which solves* $\mathcal{R}$ *in* $\mathcal{S}$ *where both* $B_1, \ldots, B_n$ *and* $A_1, \ldots, A_n$ *are subexpressions of some* $Trs^+(\mathcal{R})-$*redex.*

2. $h' \circ \overline{h}$ *solves* $\mathcal{R}$ *in* $\mathcal{S}$ *for any homomorphism* $h' : \langle \mathsf{T}, =_{\mathcal{S}} \rangle \to \langle \mathsf{T}, =_{\mathcal{S}} \rangle$*;*
We say that $\overline{h}$ *is an* essential *solution of* $\mathcal{R}$ *in* $\mathcal{S}$.

Example 3. (i) Let $\mathcal{R}_0 = \{a = a \to b\}$ be an s.r.. We want to find a solution of $\mathcal{R}_0$ in the s.r. $\mathcal{R}_1 = \{c = c \to c\}$. By point 1. of the Lemma both $h(a)$ and $h(b)$ must be subterms of some redex in $\mathrm{Trs}^+(\mathcal{R}_1)$. Note that the only redex in $\mathcal{R}_1$ is $c \to c$. Indeed choosing $h_k(a) = h_k(b) = (c \to c)$ we have a solution of $\mathcal{R}_0$ in $\mathcal{R}_1$.

(ii) Let now $\mathcal{R}'_1 = \{c = c \to c,\ c' = c' \to c'\}$ and note that $c \neq_{\mathcal{R}'_1} c'$. So, besides h_k, also the homomorphism h'_k defined by $h'_k(a) = h'_k(b) = (c' \to c')$ solves $\mathcal{R}_0$ in $\mathcal{R}'_1$. But note that $h_k \neq h'_k$.

Since there are a finite number of $\mathrm{Trs}^+(\mathcal{R})-$redexes (the l.h.s. of the equations left in $\mathcal{R}_N$) we have immediately the following corollary.

Corollary 2. *It is decidable whether a s.r.* $\mathcal{R}$ *solves a system* $\mathcal{C}$ *of equations over* $\mathsf{T}_{\mathbf{A}}$.

Statman [11] has shown that the solvability of a s.r. in another s.r. is in general a NP-complete problem. A last property will be needed in the following section.

Lemma 6. *Let* $\mathcal{R}$ *be a s.r. over* T *and* $\mathcal{P} = \{\langle A_i, B_i \rangle \mid 1 \leq i \leq n\}$ *be a set of pair of types in* T*. Then it is decidable whether there is a type structure homomorphism* $h : \mathcal{T}_{\mathcal{R}} \to \mathcal{T}_{\mathcal{R}}$ *such that* $h(A_i) = B_i$ *for all* $1 \leq i \leq n$ *(in this case we say that* h solves $\mathcal{P}$*).*

Proof. Let's define a sequence $\mathcal{P}_m$ $(m \geq 0)$ where $\mathcal{P}_m$ is either a set of pair of types or FAIL. Let $\mathcal{P}_0 = \mathcal{P}$. Then define $\mathcal{P}_{m+1}$ from $\mathcal{P}_m$ in the following way:

a. If there is a pair $\langle A_1 \to A_2, B \rangle \in \mathcal{P}_m$ we have the following cases.
   1. If $B \neq_{\mathcal{R}} B_1 \to B_2$ for some types B_1 and B_2 then $\mathcal{P}_{m+1} = \text{FAIL}$.
   2. Otherwise let $\mathcal{P}_{m+1} = \mathcal{P}_m - \{\langle A_1 \to A_2, B \rangle\} \cup \{\langle A_l, B_l \rangle \mid l = 1, 2\}$.

b. Otherwise take all pairs $\langle c, A \rangle \in \mathcal{P}_m$ such that c is an indeterminate of $\mathcal{R}$ and set $\mathcal{P}_{m+1} = \mathcal{P}_m \cup \{\langle C, A \rangle \mid \langle c, A \rangle \in \mathcal{P}_m \text{ and } c = C \in \mathcal{R}\}$.

Let's say that $\mathcal{P}_m$ is *flatten* if it contains only pairs of the shape $\langle a, A \rangle$ where a is atomic. Now it is easy to prove that either $\mathcal{P}_m = \text{FAIL}$ for some $m > 0$ or there is an $N > 0$ such that

1. $\mathcal{P}_N$ is flatten.
2. for all $n > N$ such that $\mathcal{P}_n$ is flatten we have that $\mathcal{P}_n = \mathcal{P}_N$.

The proof of this follows from the observation that all types occurring $\mathcal{P}_m$ for $m \geq 0$ must be subtypes of some type occurring in $\mathcal{P}$ or in some C_i.
It is immediate to prove, by induction on m, that there is a solution of $\mathcal{P}$ iff there is a solution of $\mathcal{P}_m$.

Now for each $a \in \mathbf{A}$ let

$$\mathcal{B}_a = \{B \mid a = B \in \mathcal{P}_N\} \cup \{C \mid a \in dom(\mathcal{R}) \text{ and } a = C \in \mathcal{R}\}.$$

Let $e_1, \ldots, e_m$ $(1 \leq m \leq n)$ denote the atoms $a \in \mathbf{A}$ such that $\mathcal{B}_a \neq \emptyset$. Now for all e_j $(1 \leq j \leq m)$ and for all $B', B'' \in \mathcal{B}_{e_j}$ check that $B' =_{\mathcal{R}} B''$. If this is true then $h = [e_1 := B_1, \ldots, e_m := B_m]$, where $B_i \in \mathcal{B}_{e_i}$ for all $1 \leq i \leq n$, solves $\mathcal{P}$. Otherwise there is no h which solves it.

Example 4. Let $\mathcal{R} = \{c_1 = c_2 \to c_1, c_2 = c_1 \to c_2\}$ and let $\mathcal{P} = \{\langle c_1, c_2 \rangle\}$. Then we have $\mathcal{P}_1 = \{\langle c_2 \to c_1, c_2 \rangle\}$ and $\mathcal{P}_2 = \{\langle c_2, c_1 \rangle, \langle c_1, c_2 \rangle\}$

It is easy to see that $\mathcal{P}_5 = \mathcal{P}_2$ and then $N = 2$. So let h be defined as $h(c_1) = c_2$ and $h(c_2) = c_1$. It is immediate to verify that $h(c_i) = h(C_i)$ for $i = 1, 2$. Then h solves $\mathcal{P}$. Note that setting only $h(c_1) = c_2$ would not give the correct solution. This shows the necessity of step b in the definition of h.

5 Finding Types

We consider now some natural problems about the typability of a λ-term in an inference system with recursive types. The questions that we will consider are the following. Let M be a given untyped λ-term:

1. Does it exist a s.r. $\mathcal{R}$, a type environment Γ and a type A such that $\Gamma \vdash_{\mathcal{R}} M : A$?
2. Given a s.r. $\mathcal{R}$ does there exist a type environment Γ and a type A such that $\Gamma \vdash_{\mathcal{R}} M : A$?
3. Given a s.r. $\mathcal{R}$, a type environment Γ and a type A does $\Gamma \vdash_{\mathcal{R}} M : A$ hold?

The same questions can be stated for the corresponding systems with tree equivalence, i.e. with $\vdash^*_{\mathcal{R}}$ instead of $\vdash_{\mathcal{R}}$. We will prove that all these questions are decidable for both notions of equivalence.

Theorem 1 which reduces tree equivalence to the equational one allows to threat both equivalences in a uniform way. The decidability of question 1. was well known (see for instance [3] or [7]) but there seems to be no published proof of the decidability of 2. and 3..

Note that in this paper recursive types are not considered in polymorphic sense. For instance an equation like $c = t \to c$ represent only itself and not all its possible instances via t. It would be interesting to allow s.r.s to contain schemes like $\forall t.c[t] = t \to c[t]$ rather then simple equations, but this is left for further research.

To keep technical details simple we investigate these problems for pure λ-terms (without term constants). All results however can be generalized in a quite straightforward way to terms including constants (see remark 2).

Using type structure homomorphisms we can define, in a quite natural way, a notion of principal type scheme for type assignments with respect to arbitrary invertible type structures. In the following definition we assume, without loss of generality, that all free and bound variables in a term have distinct names.

Definition 17. *Let M be a pure λ-term. The system of type constraints $\mathcal{C}_M$, the basis Γ_M and type T_M are defined as follows.*
Let $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ where $\mathcal{S}_1$ is set of all bound and free variables of M and $\mathcal{S}_2$

is the set of all occurrences of subterms of M which are not variables. Take a set $\mathbf{I}_M$ of atomic types such that there is a bijection $\pi : \mathcal{S} \to \mathbf{I}_M$. Then define a system of type constraints $\mathcal{C}_M$ over $\mathsf{T}_{\mathbf{I}_M}$ in the following way.

(a) For every $\lambda x.P \in \mathcal{S}_2$ put $\pi(\lambda x.P) = \pi(x) \to \pi(P)$ in $\mathcal{C}_M$;

(b) For every $P = (P_1 P_2)$ put $\pi(P_1) = \pi(P_2) \to \pi(P)$ in $\mathcal{C}_M$.

Now let $\Gamma_M = \{x : \pi(x) \mid x \text{ is free in } M\}$ and $T_M = \pi(M)$.

Lemma 7. *Let M be λ-term. Then $\Gamma_M \vdash_{\mathcal{C}_M} M : T_M$.*

Example 5. Consider the term $\lambda x.(xx)$ and assume $\pi(x) = i_1$, $\pi(xx) = i_2$, $\pi(\lambda x.(xx)) = i_3$. Then we have $\mathcal{C}_{\lambda x.(xx)} = \{i_1 = i_1 \to i_2,\ i_3 = i_1 \to i_2\}$. Moreover we have $\Gamma_{\lambda x.(xx)} = \emptyset$, $T_{\lambda x.(xx)} = i_3$.

Note that $\mathcal{C}_M$ is a system of type constraints but not, in general, a s.r.. In fact in general we can have many equations with the same left hand side.

Indeed to type any pure λ-term it would be enough to take a s.r. with only one equation $\mathcal{R}_0 = \{c = c \to c\}$. However the typings obtained assuming $\mathcal{R}_0$ are trivial and not interesting. $\mathcal{C}_M$, instead, gives the weakest type constraints necessary to type M, and then gives more informations about the structure of M. But there are obviously terms which can be typed only with respect to $\mathcal{R}_0$, like $(\lambda x.(xx))(\lambda y.y)$ (we leave to the reader to show this).
We will remark (see Remark 2) that the problem of typability is not more trivial when term and type constants are considered.

For the proof of this theorem, which was more or less known as folklore of the subject, we refer to [7].

Theorem 4. *Let M be a Λ-term and $\mathcal{T} = \langle \mathsf{T}, \simeq \rangle$ be a type structure. Then*

$$\Gamma \vdash_{\mathcal{T}} M : A$$

iff there is a type structure homomorphism

$$h : \langle \mathsf{T}_{\mathbf{I}_M}, =_{\mathcal{C}_M} \rangle \to \mathcal{T}$$

such that $A \simeq h(T_M)$ and $h(\Gamma_M)$ is a subset (modulo $\simeq$) of Γ.

Theorem 4 is a straightforward generalization of the Principal Type Theorem for Simple Curry's assignment system. The terms for which $\mathcal{C}_M$ can be solved in an empty s.r. (i.e. no equation between types assumed) are exactly these typable in Curry's system. A generalization of Theorem 4 is the following.

Theorem 5. (i) *Let M be a Λ-term and $\mathcal{R}$ a s.r. over T. Then there is a finite set $H = \{h^{\mathcal{R}}_{M,1}, \ldots, h^{\mathcal{R}}_{M,k}\}$ $(k \geq 0)$ of type structure homomorphisms $h^{\mathcal{R}}_{M,i} : \langle \mathsf{T}(\mathbf{I}_M), =_{\mathcal{C}_M} \rangle \to \mathcal{T}_{\mathcal{R}}$ such that:*

1) $h^{\mathcal{R}}_{M,i}(\Gamma_M) \vdash_{\mathcal{R}} M : h^{\mathcal{R}}_{M,i}(T_M)$

2) $\Gamma \vdash_{\mathcal{R}} M : A$ for some environment Γ and type A iff for some $1 \leq i \leq k$ there is a type structure homomorphism $h' : \mathcal{T}_{\mathcal{R}} \to \mathcal{T}_{\mathcal{R}}$ such that $A =_{\mathcal{R}} h' \circ h^{\mathcal{R}}_{M,i}(T_M)$ and $h' \circ h^{\mathcal{R}}_{M,i}(\Gamma_M)$ is a subset (modulo $=_{\mathcal{R}}$) of Γ.

(ii) *The same property holds for $\vdash^*_{\mathcal{R}}$, replacing $\mathcal{T}_{\mathcal{R}}$ with $\mathcal{T}^*_{\mathcal{R}}$ and $=_{\mathcal{R}}$ with $=^*_{\mathcal{R}}$.*

Proof. (i) If $\mathsf{T} = \mathsf{T}(\mathbf{A})$ we can assume without loss of generality that $\mathbf{I}_M \subseteq \mathbf{A}$. The proof follows by Theorem 4 and Lemma 5, observing that any homomorphism $h : \langle \mathsf{T}, =_{\mathcal{C}_M} \rangle \to \mathcal{T}_{\mathcal{R}}$ gives a solution of $\mathcal{C}_M$ in $\mathcal{R}$ and then, by Lemma 4, of ${\mathcal{C}_M}^{\text{inv}}$ in $\mathcal{R}$. So we take H as the set of the essential solutions of ${\mathcal{C}_M}^{\text{inv}}$ in $\mathcal{R}$.
(ii) Apply (i) by replacing $\vdash^*_{\mathcal{R}}$ with $\vdash_{\mathcal{R}^*}$, where $\mathcal{R}^*$ is built as in the proof of Theorem 1.

Some direct consequences of this theorem are the following.

Theorem 6. (i) *Given a pure λ-term M and an s.r. $\mathcal{R}$ it is decidable whether there exist a type environment Γ and a type A such $\Gamma \vdash_{\mathcal{R}} M : A$.*

(ii) *Given a pure λ-term M, an s.r. $\mathcal{R}$, a type context Γ and a type A it is decidable whether $\Gamma \vdash_{\mathcal{R}} M : A$*

(iii) *Both i) and ii) hold also for $\vdash^*_{\mathcal{R}}$ (i.e. replacing $\vdash_{\mathcal{R}}$ by $\vdash^*_{\mathcal{R}}$).*

Proof. (i) By Theorem 5 we have that a term can be types with respect to $\mathcal{R}$ only if $\mathcal{C}_M$ can be solved in $\mathcal{R}$. The property then follows by Lemma 2
(ii) Assume $\Gamma \vdash_{\mathcal{R}} M : A$. We can assume without loss of generality that Γ contains all and only the variables free in M. By Lemma 4 and Theorem 5 there is a type structure homomorphism $h' \circ h : \langle \mathsf{T}, \mathcal{C}_M \rangle \to \mathcal{T}_{\mathcal{R}}$ such that $A \simeq h' \circ h(T_M)$ and $h' \circ h(\Gamma_M) =_{\mathcal{R}} \Gamma$, where $h \in H$, the set of essential solutions of $\mathcal{C}_M$ in $\mathcal{R}$, and $h' : \mathcal{T}_{\mathcal{R}} \to \mathcal{T}_{\mathcal{R}}$. Referring to Definition 17, let $\mathcal{X} = \{i \mid x \in FV(M)$ and $x : A_i \in \Gamma$ and $\pi(x) = i\}$. Then h' must be such that:

- $h'(h(T_M)) = A$;
- $h'(h(i)) = A_i$ for all $i \in \mathcal{X}$.

We get the result from Lemma 6 and the fact that H has a finite number of elements.
(iii) Immediate from Theorem 1.

Remark 2 (About Term Constants). If M can contain occurrences of term constants then we must assume a set K of constants types, like the type **int** of integers. Moreover to each term constants c we associate a type $\tau(\mathsf{c})$ which is usually a constant of first order type (for instance if 3 is a numeral $\tau(3) = \mathbf{int}$). It is standard to assume that constant types can be equivalent only to themselves, and one is not allowed to equate them to other types, for instance **int** to an arrow type. We say that $\mathcal{C}_M$ is *consistent* if it does not imply any equation $\kappa = C$ where κ is a constant type and C is a either a different constant type or a non atomic expression. Then as a consequence of Theorem 4 a term M can be typed (w.r.t. some type structure) iff $\mathcal{C}_M$ is consistent.

We can easily take that into account constants in the construction of Definition 17 by adding to $\mathcal{C}_M$ an equation $\pi(\mathsf{c}) = \tau(\mathsf{c})$ for each constant c occurring in M. To check now that $\mathcal{C}_M$ is consistent it is enough to build ${\mathcal{C}_M}^{\text{inv}}$. We can easily prove that $\mathcal{C}_M$ is consistent iff for each atomic constant type κ there are in ${\mathcal{C}_M}^{\text{inv}}$ equations $\kappa = a_1, a_1 = a_2, \ldots, a_n = C$ where C is either a constant different from κ or a non atomic type expression. It is then decidable if $\mathcal{C}_M$ is consistent.

All the results given this section still hold if we consider terms with constants (in the previous sense). In some cases this makes these results more interesting. For instance Problem 1. (to decide whether a given term has a type w.r.t. some s.r.) is not trivial any more since there are terms, like (3 3), which have no type w.r.t. any s.r.. By the subject reduction theorem, moreover, we still have that a term to which it can be given a type (in any consistent s.r.) can not produce bad applications during its evaluation.

Acknowledgments. The author acknowledge H. Barendregt, W. Dekkers and the anonymous referees for their helpful suggestions.

References

1. M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. In P. de Groote and J. R. Hindley, editors, *Typed Lambda Calculi and Applications*, volume 1210 of *Lecture Notes in Computer Science*, pages 63–81. Springer-Verlag, 1997.
2. V. Breazu-Tannen and A. Meyer. Lambda calculus with constrained types. In R. Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 23–40. Springer-Verlag, 1985.
3. F. Cardone and M. Coppo. Type inference with recursive types. Syntax and Semantics. *Information and Computation*, 92(1):48–80, 1991.
4. B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.
5. J.R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.
6. J.W. Klop. Term rewriting systems. In Dov M. Gabbay S. Abramsky and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 1–116. Oxford University Uress, New York, 1992.
7. M. Marz. An algebraic view on recursive types. *Applied Categorical Structures*, 7(12):147–157, 1999.
8. R. Milner. A Theory of Type Polimorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
9. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
10. D. Scott. Some philosophical issues concerning theories of combinators. In C. Böhm, editor, *Lambda calculus and computer science theory*, volume 37 of *Lecture Notes in Computer Science*, pages 346–366. Springer-Verlag, 1975.
11. R. Statman. Recursive types and the subject reduction theorem. Technical Report 94–164, Carnegie Mellon University, 1994.
12. B. Pierce V. Gapeyev, M. Levin. Recursive subtyping revealed. In *Proceedings fifth ACM SIGPLAN International Conference on Functional Programming*, pages 221–232. ACM press, 2000.