Type inference for nested self types (extended abstract)

Type Inference for Nested Self Types (Extended Abstract)*

Viviana Bono[1]**, Jerzy Tiuryn[2], and Paweł Urzyczyn[2]***

[1] Università di Torino, Dipartimento di Informatica
c. Svizzera 185, 10149 Torino, Italy
bono@di.unito.it
[2] Uniwersytet Warszawski, Instytut Informatyki
Banacha 2, 02-097 Warszawa, Poland
{tiuryn,urzy}@mimuw.edu.pl

Abstract. We address the issue of decidability of the type inference problem for a type system of an object-oriented calculus with general selftypes. The fragment considered in the present paper is obtained by restricting the set of operators to the method invocation only. The resulting system, despite its syntactical simplicity, is sufficiently complicated to merit the study of the intricate constraints emerging in the process of type reconstruction, and it can be considered as the core system with respect to typability for extensions with other operators. The main result of the paper is the decidability of type reconstruction, together with a certain form of a principal type property.

1 Introduction

Object-oriented programming languages enjoy an ever growing popularity, as they are a tool for designing maintainable and expandable code, and are also suited for developing web applications and mobile code. A type discipline is then in order to ensure safety (i.e., absence of message-not-understood run-time errors), but yet this type discipline has to be flexible enough not to restrain reusability. Polymorphic type systems are one answer to that double requirement, see for example [6, 14]. Among the many features that can be included in such systems, there is the use of *selftype*.

The concept of *self* (sometimes called *this*) is of paramount importance in object-oriented languages. Self is a special variable that allows to refer to the object executing the current method, and so to access its fields and to invoke the sibling methods. This concept, while being a very handy feature, influences substantially the problem of static typing for object-oriented languages. Self types have been a subject of foundational studies, both in the object-based and in the class-based setting (see, for example, [1, 8, 12]). The work done has brought up the importance of typing self in a careful way.

* Research work conducted within the framework of Types WG Project IST-1999-29001.
** This work has been partially supported by EU within the FET - Global Computing initiative, project DART IST-2001-33477, and by MIUR project NAPOLI. The funding bodies are not responsible for any use that might be made of the results presented here.
*** Partly supported by KBN Grant 7 T11C 028 20.

The gain of introducing an appropriate type for self is evident when a form of inheritance is present, either a class-based one (via class hierarchies), or an object-based one (via method addition/override). In fact, an appropriate type for self would allow to specialize automatically those inherited/overriden methods that either return the host object, or have some parameters of the same type as the host object (*binary* methods), or both. This specialization (also known as *MyType* specialization) can be seen as an alternative to *typecasts*, which are explicit declarations made by the programmer on the expected actual type of the method result according to the type of the object the method is invoked upon. Typecasts are unsafe—the programmer must be "sure" about the actual type of the returned object, since little or even no static checking is performed on typecasts, as it happens, respectively, in Java or in C++. Moreover typecasts certainly do not improve readability of code, with bad impact on the debugging phase. As hinted above, an alternative to typecasts is the introduction of *selftype*, with the meaning "the type of the current object", i.e., "the type of self", to annotate appropriately binary methods and methods that return the host object. Some type systems including selftype are presented in [1, 8, 12].

In this paper we consider the problem of *type inference* (also called *typability*) in presence of general (arbitrarily nested) selftypes in an object-based setting. Very little is known about type inference with selftypes. At the best of our knowledge, only Palsberg and Jim approached this subject. In [17] they study the type inference problem for one of the Abadi-Cardelli systems [1] extended with the notion of selftype. In [16], Palsberg presents an algorithm for the Abadi-Cardelli's four first-order systems (without any form of selftype), proving that the type inference problem for all four systems is P-complete[3]. The work [17] can be seen as an application of the techniques developed in [16], and it contains a proof that the type-inference problem for an Abadi-Cardelli system with recursive types and width subtyping extended with a simple form of selftype is NP-complete. The Palsberg-Jim "tiny drop of selftype", as the authors themselves point out, consists of:

- the use of the keyword *selftype*, instead of a bound variable, to stand for the selftype in object types;
- the restriction that each occurence of selftype "comes with its context", i.e., more specifically in their type system *selftype* can appear as a component of an object type only, never in isolation.

These two choices imply the following consequences:

- it is not possible to refer to the selftype of enclosing outer objects (i.e., there are no *nested* selftypes);
- it is not possible to override those methods that return the object itself (i.e., of type selftype).

Of the above two restrictions, the first one seems to be very essential. It implies that the access to two or more different *selftype*'s, i.e., two or more different environments, is impossible. Indeed, the "tiny drop of selftype" of Palsberg and Jim can be encoded in a system without selftype, see [9], meaning that it is a rather weak form of selftype.

[3] In [13] this result is improved.

We plan, then, to analyze the decidability of the type inference for a type system that relaxes the above limitations. The system under study is based on the calculus presented in [3] (we will call it C from now on). The calculus C is an untyped version of the calculus introduced in [5], in order to analyze throughout a functional encoding the type system of [4] (hereafter BB), which is, in turn, a simplification of the Lambda Calculus of Object (hereafter LCO) of Fisher, Honsell, Mitchell [12]. The calculus LCO is a functional object-based calculus enriched with object primitives. Operations allowed on objects are *method addition*, *method override*, and *method invocation* (also called *send*). Method bodies are functions, in particular self is modelled using a lambda-abstracted variable. In LCO: (*i*) it is possible to refer to the selves of the enclosing objects; (*ii*) override is as general as possible. Selftype is rendered by using the *row-variables* of [15] to characterize types of methods as type-schemes (i.e., types polymorphic in these variables), and to enforce correct instantiation of the schemes as methods are inherited. The type system of C we base our paper on inherits some of the fundamental ideas from the original system in the modelling of selftype. The main difference is in not using row-variables to model selftype, but exploiting instead Bruce's *matching* and implicit match-bounded quantification over type variables, as it was studied in the BB calculus of [4]. Matching is a relation over recursive object-types that was first introduced by [7] as an alternative to F-bounded subtyping [10] in modelling the subclass relation in class-based languages. The designers of LCO conjectured that type inference for LCO was undecidable, but nobody has proven that yet. Thus, C, having a simpler (yet as expressive as) type system than LCO, seems to be the appropriate system to study type inference in presence of a general form of selftype.

Due to the generality of system C, and the surprising technical difficulties arising in its analysis, we decided to tackle the related type inference problem in steps. First of all, we discarded the method addition operation. This is because we believe that method addition does not add much to the inherent difficulty of the problem of type inference, since method addition is performed on objects but it is forbidden on selves[4]. See Section 8 for an overview on how to deal with it.

In this paper we work with a calculus that has method invocation only, i.e., we also discard method override. The reason for what seems quite a radical choice is that method invocation turns out to create an interesting problem by itself. The possibility of referring to nested selves of enclosing outer objects creates "reference loops" which are difficult to untangle. In the sequel, the formal development of the subject will make it clear what we mean by "loops" and how we solve them in order to prove decidability. Loops may be also created and/or modified by *self-inflicted* overrides, i.e., overrides operating not on proper objects (called *external overrides*), but on selves inside method bodies. Nevertheless, the treatment of external overrides is not as trivial as it might look (see Section 9).

If it is true that the usefulness of selftype shows out when an inheritance mechanism is applied, it is also evident that the problem of type inference we are dealing with is by no means trivial. The restricted system we study in this paper is important because we do believe that it is the core of any other richer systems (i.e., systems including any form

[4] There are calculi that deal with *self-inflicted* method addition such as [11], but they go beyond the goal of this paper, which is about classical object-based calculi.

of method addition and/or override), from the point of view of the type reconstruction. In the sequel of the paper it will be clear how both "detecting bad loops" and "solving good loops", in order to type a term, amount to prove that the term has a correct structure with respect to the way selves are used.

It might seem natural to identify objects with recursive records, so their types with recursive types, but this is misleading. In fact: (*i*) such choice is not adequate already in our setting with method invocation only, because of the generality of our selves (see the examples in Section 2); (*i*) generally, this solution does work in an enlarged setting with method override and/or addition because the meaning of self changes as operations on the host object are performed (see [1], Chapter 6.7.2).

There is a number of simplifications we make in our syntax. For instance we consider objects with exactly two methods, and put "constants" (such as c, d, ...) as *place-holders* wherever we mean "an irrelevant subexpression". These simplifications do not influence the essence of the problem, and make it easier to isolate the basic issue: type assignment in presence of multiple selves. Place-holders are just there to hide whatever is non-important for the current typing and they do not influence typing itself. The presence of two fields only rules out the so-called *message-not-understood* run-time errors, as sends are limited to those two components. Even though catching statically such errors is a primary task of type systems for object-oriented languages, the task of testing a sort of "well-formedness" of objects is essential as well, and this is what our typability algorithm does. Nevertheless, a message-not-understood error treatment may be introduced in our typability algorithm (see Section 8).

This paper might be seen as a first step towards introducing selftypes in real programming languages as an alternative to typecasts. No matter whether the type inference problem in its most general form (including override as well) is decidable or not, the next steps will be tailoring suitable type systems with decidable and tractable forms of selftype.

The main technical contribution of the paper is as follows:

- We prove that type reconstruction is decidable for a language involving nested self-references.
- We show a certain form of a principal quasi type scheme property. The salient feature of the principal quasi type scheme of a term is that it exists iff the term is typable and every instance of the scheme gives a correct typing of the term. Although not every typing of the term is a substitution instance of the scheme, therefore we do not have a complete notion of "principal typing" for the moment being, one can reasonably argue that our principal quasi type scheme provides the most essential information about all typings.

The paper is organized as follows. Section 2 introduces the most basic syntactic categories, terms and types, and explains the motivation of our type assignment. In Section 3 we elaborate the syntactic notions used in our consideration. Important categories of terms are: types, quasi types and stripped terms. We introduce the operation of formal field selection for quasi types (Section 3.1) and evaluation of a stripped term in an environment (Section 3.2). Section 4 introduces a type assignment system. In Section 5 we introduce the concept of a meta scheme – the main tool in establishing the

"principal quasi type scheme" property. This section also contains the notion of scheme equivalence. The latter notion is used for expressing the confluence property of a certain system of reductions.

The main technical part of the paper is Section 6. It is devoted to an algorithm which transforms a given stripped term into a quasi type scheme, or reports a failure. Section 6.1 introduces two fundamental mappings defined on schemes: the projection and reminder maps. It also contains an invariant property maintained by these maps. Sections 6.2 and 6.3 describe two classes of redexes: reducible and cyclic. Among the reducible redexes there are redexes which we call inconsistent. They are the only source of a possible untypability of a term in our setting. Section 6.4 contains the main technical properties of the reduction system: confluence and termination (Theorem 6.1), and recovery of typings (Theorem 6.2).

Section 7 contains the main result of the paper: the principal quasi type theorem (Theorem 7.3). It states that there is a quasi type scheme assigned to every (and only) typable term such that all the instantiations of this scheme give correct typings of the term. This principal quasi type scheme can be effectively obtained if and only if the term is typable. Therefore the typablity problem is decidable.

Section 8 gives an informal account on how to deal with method addition and *message-not-understood* run-time errors.

Due to space limitations we have omitted from this extended abstract all proofs and many auxiliary definitions which are not essential for understanding the presentation of the main results of the paper. The details can be found in the full version of the paper in `http://www.di.unito.it/~bono/Manuscripts`.

2 Terms and types

Assume an infinite set of variables (selves), notation $s, t, \ldots$, and an infinite set of place-holders, notation $c, d, \ldots$ A *term* is either a variable or a place-holder or:

- an *object*, i.e., an expression of the form $\mathbf{pro}\ s\langle\, M_1, M_2 \,\rangle$, where M_1 and M_2 are terms, or:
- a *send*, i.e., an expression of the form $M \Leftarrow i$, where M is a term and $i \in \{1, 2\}$.

The operator $\mathbf{pro}\ s$ binds the self s. Alpha conversion is assumed. The notation $FV(M)$ and $M[N/s]$ is used accordingly (with $c[N/s] = c$).

The intended meaning of "$\mathbf{pro}\ s\langle\, M_1, M_2 \,\rangle$" is an object with two methods M_1 and M_2, which may refer to the whole object via the self variable s. In the notation of [1] this would be written as $\langle\, \varsigma s.M_1, \varsigma s.M_2 \,\rangle$. The meaning of "$M \Leftarrow i$" is to extract the i-th method from the object M, and the operational semantics is given by the following reduction rule:

$$\mathbf{pro}\ s\langle\, M_1, M_2 \,\rangle \Leftarrow i \quad \rightsquigarrow \quad M_i[\mathbf{pro}\ s\langle\, M_1, M_2 \,\rangle / s].$$

We remind the reader that a place-holder c may be seen as replacing and "hiding" a piece of code (i.e., a subexpression) which is irrelevant to the typing of the whole code

in question. Clearly, a message sent to an irrelevant target is irrelevant too, so we do not find anything wrong in postulating this reduction:

$$c \Leftarrow i \quad \rightsquigarrow \quad c,$$

which expresses exactly the idea of ingnoring the "contents" of c. On the other hand, the expression $s \Leftarrow i$ has some meaning, but it can not be evaluated until we substitute an actual object for s.

We want to assign types to expressions of our language. The basic idea is that a type assigned to $\mathbf{pro}\ s\langle\, M_1, M_2 \,\rangle$ should be esssentially a product of types assigned to M_1 and M_2. Thus, we would like to assert something like

$$\mathbf{pro}\ s\langle\, 3, 5 \,\rangle : \langle\langle\, int, int \,\rangle\rangle,$$

provided we know that $3, 5 : int$. In general, the type of a pure object (an expression without sends) should correspond to the shape of the object. If an object refers to a self s, the natural choice is to use a type variable t, corresponding to the self s and assert

$$\mathbf{pro}\ s\langle\, s, 5 \,\rangle : \delta t\, \langle\langle\, t, int \,\rangle\rangle,$$

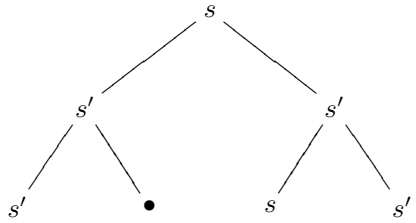
where the operator δt binds t within $\langle\langle \ldots \rangle\rangle$. This can be extended to more complex pure objects, e.g.,

$$\mathbf{pro}\ s\langle\, \mathbf{pro}\ s'\langle\, s', 5 \,\rangle, \mathbf{pro}\ s'\langle\, s, s' \,\rangle\,\rangle$$

is of type

$$\delta t\, \langle\langle\, \delta t'\langle\langle\, t', int \,\rangle\rangle, \delta t'\langle\langle\, t, t' \,\rangle\rangle\,\rangle\rangle.$$

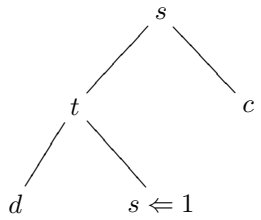
However, as said above, we do not really care about place-holders and their types. This leads us to the following simplification. Assuming $c : c$ for any place-holder c, we can simply identify a pure object with its own type. Indeed, the difference between the term and the type in the last example is just syntactic sugar. The essential part, the structure of self-references, is the same, up to renaming, and can be drawn as the following tree:



This justifies our definition of a type as a term not containing $\Leftarrow$. An assignment of such a type τ to an expresssion M containing occurrences of $\Leftarrow$ means: M is as good as a pure object of type τ.

Our type assignment should have the subject reduction property, i.e., we want $M' : \tau$, whenever $M \rightsquigarrow M'$ and $M : \tau$. This requirement determines what the type assignment rules should be. First of all, observe that $M : \mathbf{pro}\ s\langle\, \tau_1, \tau_2 \,\rangle$ should imply that $M \Leftarrow i$ is of type $\tau_i[s := \mathbf{pro}\ s\langle\, \tau_1, \tau_2 \,\rangle]$. It is less obvious which type should be assigned to a send of the form $s \Leftarrow i$. Clearly, our identification of an object and its type requires a uniform principle $s : s$. A self is of type self. The type of $s \Leftarrow i$

should depend on the context the expression occurs in. Consider as an example the term $M = \mathbf{pro}\, s\langle\, \mathbf{pro}\, t\langle\, d, s \Leftarrow 1\,\rangle, c\,\rangle$, depicted as



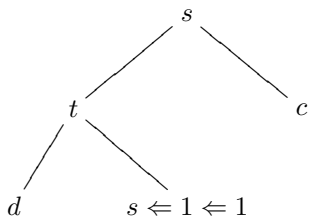
It may be tempting to assert $s \Leftarrow 1 : t$, because $s \Leftarrow 1$ certainly points to the root of the object identified by the self t. This amounts to understanding an object type $\mathbf{pro}\, t\langle \ldots \rangle$ as a recursive type $\mu t\langle \ldots \rangle$, that may freely be replaced by $\langle \ldots \rangle[\mu t\langle \ldots \rangle/t]$. That, however would be wrong: consider the expression $M \Leftarrow 1$. We have

$$M{\Leftarrow}1 \rightsquigarrow \mathbf{pro}\, t\langle\, d, M{\Leftarrow}1\,\rangle \rightsquigarrow \\ \rightsquigarrow \mathbf{pro}\, t\langle\, d, \mathbf{pro}\, t\langle\, d, M{\Leftarrow}1\,\rangle\,\rangle \rightsquigarrow \cdots$$

From this reduction sequence we can see that no finite object type can be assigned to M, as the expression develops into an infinite tree. Thus, M should not be typed at all.

Note that the idea of a recursive type $\mu t\langle\, d, t\,\rangle$ is not adequate here, which can be best seen if we modify M to $M' = \mathbf{pro}\, s\langle\, \mathbf{pro}\, t\langle\, t, s \Leftarrow 1\,\rangle, c\,\rangle$. While M' expands to an infinite tree in reduction, it is not a full binary tree! Another reason why we do not want to use recursive types is that we certainly want to distinguish between $\mathbf{pro}\, s\langle\, 4, \mathbf{pro}\, s\langle\, 2, s\,\rangle\,\rangle$ and $\mathbf{pro}\, s\langle\, 2, s\,\rangle$.

The problem we encountered in the above example does not occur, if we consider the term $N = \mathbf{pro}\, s\langle\, \mathbf{pro}\, t\langle\, c, s \Leftarrow 1 \Leftarrow 1\,\rangle, d\,\rangle$. The picture is now



and the type of $s \Leftarrow 1 \Leftarrow 1$ in this context should undoubtedly be d. So what is the type of $s \Leftarrow 1$? Now we see it must be $\mathbf{pro}\, t\langle\, d, d\,\rangle$. But how can we derive it? For this we need to know the type of N from an *environment* that assigns to s the type of the object s points to. (It is not a type of s as we always have $s : s$.) We arrive at the following rule

$$\frac{t : \mathbf{pro}\, t\langle\, \tau_1, \tau_2\,\rangle \vdash M : t}{t : \mathbf{pro}\, t\langle\, \tau_1, \tau_2\,\rangle \vdash M \Leftarrow i : \tau_i}$$

Thus to derive the type for N we must first guess it, put it into an environment in which we derive types of the components of N, and finally we apply the following rule for typing objects

$$\frac{s : \mathbf{pro}\, s\langle\, \tau_1, \tau_2\,\rangle \vdash N_1 : \tau_1, \quad s : \mathbf{pro}\, s\langle\, \tau_1, \tau_2\,\rangle \vdash N_2 : \tau_2}{\vdash \mathbf{pro}\, s\langle\, N_1, N_2\,\rangle : \mathbf{pro}\, s\langle\, \tau_1, \tau_2\,\rangle}$$

to eliminate the initial guess from the environment.

The need of guessing the final type of a complex expression, before type-checking begins, makes it difficult to apply any structural approach to type inference. The problem becomes even more involved in presence of an interaction between "external" sends to an object expression and "internal" sends occurring within that expression.

2.1 The roadmap of notions

We conclude this informal introduction with a brief description of several syntactic categories used in the course of the proof of the main decidability result. We use the following subsets of the set of all terms, orderd as shown below:

$$\text{types} \subseteq \text{quasi types} \subseteq \text{stripped terms} \subseteq \text{terms}.$$

Stripped terms are terms in which all applications of the send operator $\Leftarrow$ are 'stripped down' to leaves, i.e. $\Leftarrow$ occurs only in the context $s \Leftarrow \Pi$, where s is a self and $\Pi \in \{1,2\}^*$ is a non-empty path. Moreover, if an occurrence of $s \Leftarrow \Pi$ is bound, then the binding $\mathbf{pro}$ is the outermost $\mathbf{pro}$ of the term. The main technical part of the algorithm which decides typability is concerned with stripped terms. The strategy of the algorithm consists in rewriting a given stripped term, trying to eliminate bound occurrences of $s \Leftarrow \Pi$.

In this way we arrive at the next syntactic category of terms: quasi types. A stripped term without bound occurrences of $s \Leftarrow \Pi$ (with $\Pi \neq \varepsilon$) is called a quasi type. Hence a quasi type is a term in which all applications of the send operator are 'stripped down' to the leaves and every such an occurrence is free, i.e. no $\mathbf{pro}$ binds a self s which is in the context $s \Leftarrow \Pi$, with $\Pi \neq \varepsilon$. Quasi types behave in several respects similar to types: a quasi type is always typable and moreover its type is uniquely determined by the environment.

Finally the smallest syntactic category, types, consists of terms in which no send operator occurs.

Since we are interested in a form of principal typing, we have to allow metavariables which range over types. In this way we obtain a class of meta schemes — they are just like ordinary terms, except that they may contain metavariables. Again, meta schemes are stratified syntactically in a similar way as described above. Thus we have:

$$\text{quasi type schemes} \subseteq \text{stripped schemes} \subseteq \text{meta schemes}.$$

Quasi type schemes are produced by the algorithm of the paper for each (and only) typable term (see Theorem 7.3), which is the main result of this paper.

3 Technical background

If $\Pi \in \{1,2\}^+$, then we define $M \Leftarrow \Pi$ by induction: $M \Leftarrow \Pi i := M \Leftarrow \Pi \Leftarrow i$. Occasionally, we use the notation $M \Leftarrow \Pi$, even if Π can be empty, identifying $M \Leftarrow \varepsilon$ with M. We call every send of the form $s \Leftarrow \Pi$, where $\Pi \neq \varepsilon$, an *atomic send*. A *top send* in an object $M = \mathbf{pro}\, s.\langle M_1, M_2 \rangle$ is an atomic send $s \Leftarrow \Pi$ bound by the top

$\mathbf{pro}\ s$ in M. The length of Π is the *length* of the send $s \Leftarrow \Pi$. We say that an atomic send $s \Leftarrow \Pi$ is *free* in M if $s \in FV(M)$.

If a term does not contain non-atomic sends, it is often convenient to think of it as a labeled binary tree. Internal nodes are labeled by selves and leaves are labeled by placeholders, selves or sends. Nodes are identified with paths leading to them. For a string $\Gamma \in \{1,2\}^*$ and a term M, if Γ leads in M to a node we will say that Γ is *contained* in M and write $\Gamma \in M$. For $\Gamma \in M$ we can also refer to a label of Γ meaning the label of the node to which Γ leads in M.

A *type* is a term not containing $\Leftarrow$. In particular an *object type* is a type which is an object, as well. A *quasi type* is a term in which all sends are atomic and free. A *stripped term* is a term of the form $\mathbf{pro}\ s.\langle\, M_1, M_2 \,\rangle$, where M_1 and M_2 are quasi types. Thus in a stripped term all bound sends are top sends.

A *self declaration* is a pair of the form $s : \tau$, where τ is an object type. An *environment* is a sequence of self declarations, such that no declaration in E involves a (free) variable declared later on in E. More precisely, the definition of an environment, its *domain* $\mathrm{Dom}(E)$, and the set $FV(E)$ of *free selves* of E is stated inductively as follows.

- The empty sequence $\emptyset$ is an environment, and $\mathrm{Dom}(\emptyset) = \emptyset = FV(\emptyset)$.
- If E is an environment, s is a self such that $s \notin FV(E)$, and τ is a type, then $E' = E, s : \tau$ is an environment, with $\mathrm{Dom}(E') = \mathrm{Dom}(E) \cup \{s\}$ and $FV(E') = FV(E) \cup FV(\tau)$.

We will use the convention that if $s : \tau$ is a declaration, then τ is of the form $\tau = \mathbf{pro}\ s.\langle\, \tau_1, \tau_2 \,\rangle$. For $s \in \mathrm{Dom}(E)$, we write $E(s) = \tau$ if τ is the type which is assigned to s by the rightmost declaration for s in E.

3.1 Formal field selection

Given a quasi type T and $\Pi \in \{1,2\}^*$, we define a quasi type $T.\Pi$, called a *formal field selection*:

- $T.\varepsilon = T$,
- $c.\Pi = c$,
- $(s \Leftarrow \Gamma).\Pi = s \Leftarrow \Gamma\Pi$, for $\Gamma \in \{1,2\}^*$, in particular $s.\Pi = s \Leftarrow \Pi$,
- if $T = \mathbf{pro}\ s.\langle\, T_1, T_2 \,\rangle$, then $T.i\Pi = (T_i[T/s]).\Pi$

Let us stress that by the above definition the notations $s.\Pi$ and $s \Leftarrow \Pi$ are interchangeable.

In the last clause of the above definition the substitution $T_i[T/s]$ is just the ordinary substitution of T for all free occurrences of s in T_i. Notice that in this case no free occurrence of s in T_i is a free send. Otherwise, the result of the substitution is not necessarily a quasi type. That is, quasi types are not closed with respect to ordinary substitutions. The general case of substitution of quasi types is dealt with in the full version of the paper.

3.2 Evaluation of stripped terms in an environment

Given a stripped term M, we define a stripped term $(M)_E$, called the *value* of M in the environment E, as follows.

- $(s)_E = s$,
- $(c)_E = c$,
- $(s \Leftarrow i\Pi)_E = (\tau_i.\Pi)_E$, whenever $E(s) = \mathbf{pro}\ s.\langle\, \tau_1, \tau_2 \,\rangle$.
- $(s \Leftarrow i\Pi)_E = s \Leftarrow i\Pi$, if $s \notin \mathrm{Dom}(E)$.
- $(\mathbf{pro}\ s.\langle\, M_1, M_2 \,\rangle)_E = \mathbf{pro}\ s\langle\, (M_1)_E, (M_2)_E \,\rangle$, where $s \notin FV(E) \cup s \notin \mathrm{Dom}(E)$.

Note that the above definition is correct, i.e. that the induction is well-founded.

Lemma 1. *Let T be a quasi type and let $\Pi \in \{1,2\}^*$. Then for every environment E, we have*

$$(T_E.\Pi)_E = (T.\Pi)_E.$$

In particular,

$$((s \Leftarrow \Gamma)_E.\Pi)_E = (s \Leftarrow \Gamma\Pi)_E.$$

4 Type assignment

A *type judgement* takes the form $E \vdash M : \tau$, where E is a type environment, M is a term and τ is a type. Here are the rules. In (obj) we use the abbreviations $\tau = \mathbf{pro}\ s\langle\, \tau_1, \tau_2 \,\rangle$ and $M = \mathbf{pro}\ s\langle\, M_1, M_2 \,\rangle$.

$$(\mathbf{const}) \quad \frac{}{E \vdash c : c}$$

$$(\mathbf{var}) \quad \frac{}{E \vdash s : s}$$

$$(\mathbf{obj}) \quad \frac{E, s : \tau \vdash M_1 : \tau_1, \quad E, s : \tau \vdash M_2 : \tau_2}{E \vdash M : \tau}$$

$$(\mathbf{send}) \quad \frac{E \vdash M : \tau}{E \vdash M \Leftarrow i : (\tau.i)_E} \qquad (\text{if } (\tau.i)_E \text{ is a type})$$

First of all, observe that the understanding of $E \vdash M : \tau$ is nonstandard. The environment E does *not* provide types of free variables, as usually, but only "type bindings" used only for typing sends. The type assigned to a free variable is always the variable itself. In particular, one does not need to assume free variables of M to be in the domain of E, as long as there is no (direct or indirect) send involving these variables. For instance we have $\vdash s : s$, but to type $\mathbf{pro}\ s\langle\, s, t \,\rangle \Leftarrow 21$ we need a type binding for t. Furthermore, notice that the type of a place-holder c is the place-holder itself, and this reflects our idea that place-holders stand for ignored sub-expressions.

The reader familiar with [4] will notice that our type bindings are directly inspired by the idea of "matching types". A direct comparison between the present system and C of [3] is possible: our syntax of terms is different than that of C, but if we forget about that, a closer look reveals that our rule (obj) corresponds to (two applications of) rule (Val Method Addition) of C, and rule (send) is essentially the same as C's rule (Val Select).

Below we illustrate the features of the system with some examples.

Example 1. Not every term is typable. Consider the following stripped term

$$M = \mathbf{pro}\ s\langle\, \mathbf{pro}\ t\langle\, s \Leftarrow 1, c\,\rangle, c\,\rangle$$

and show that M is indeed untypable. Assume that $\vdash M : \tau$, for some type τ. It follows that τ must be of the form $\tau = \mathbf{pro}\ s.\langle\, \tau_1, \tau_2\,\rangle$ and we must have a derivation of

$$s : \tau \vdash \mathbf{pro}\ t\langle\, s \Leftarrow 1, c\,\rangle : \tau_1.$$

Now, again τ_1 must be of the form $\tau_1 = \mathbf{pro}\ t\langle\, \tau_{11}, \tau_{12}\,\rangle$ and we must have a derivation

$$s : \tau, t : \tau_1 \vdash s \Leftarrow 1 : \tau_{11}.$$

Thus $\tau_{11} = (\tau_1)_E = \tau_1$, where $E = \{s : \tau, t : \tau_1\}$. This yields a contradiction.

Observe that the type of a term is not uniquely determined by the term and the environment (see Example below). However, it can be shown that the resulting type of a quasi type is uniquely determined by the environment.

Example 2. Consider now a stripped term $M = \mathbf{pro}\ s\langle\, s \Leftarrow 12, s \Leftarrow 112\,\rangle$. The reader will easily check that the following typings are derivable in the system.

$$\vdash M : \mathbf{pro}\ s\langle\, c, c\,\rangle \tag{1}$$

$$\vdash M : \mathbf{pro}\ s\langle\, \mathbf{pro}\ t\langle\, \mathbf{pro}\ x\langle\, y, z\,\rangle, t\,\rangle, z\,\rangle \tag{2}$$

$$\vdash M : \mathbf{pro}\ s\langle\, \mathbf{pro}\ t\langle\, \mathbf{pro}\ x\langle\, y, s\,\rangle, t\,\rangle, s\,\rangle \tag{3}$$

$$x : \mathbf{pro}\ x\langle\, y, z\,\rangle \vdash M : \mathbf{pro}\ s\langle\, \mathbf{pro}\ t\langle\, x, t\,\rangle, z\,\rangle \tag{4}$$

$$t : \mathbf{pro}\ t\langle\, \mathbf{pro}\ x\langle\, y, z\,\rangle, t\,\rangle \vdash M : \mathbf{pro}\ s\langle\, t, z\,\rangle \tag{5}$$

$$x : \mathbf{pro}\ x\langle\, y, z\,\rangle,\ t : \mathbf{pro}\ t\langle\, x, t\,\rangle \vdash M : \mathbf{pro}\ s\langle\, t, z\,\rangle \tag{6}$$

Types assigned to M in (1) and (2) are clearly of completely different nature. Also the types in (2) and (3) are different due to different structure of bindings. Environments in (4)–(6) are used to type atomic sends of M.

We remark on passing that the above type assignment system has the *subject reduction property* (see details in the full version of the paper).

5 Meta schemes

We introduce the meta schemes and their instantiations in order to state the principal quasi type theorem. First we introduce a new category of variables, called *metavariables*. For each path $\Delta \in \{1, 2\}^*$ we have a countable supply of metavariables α^Δ (possibly with subscripts, when necessary). Each α^Δ can be instantiated with a type which has to satisfy a certain property to be stated later. Metavariables play the same role as selves, except they cannot be bound by $\mathbf{pro}$. In particular, the send operation is applicable to a metavariable.

We start with *meta schemes*, $\mathcal{T}$. They are build according to the following grammar

$$\mathcal{T} ::= \ c \mid s \Leftarrow \Pi \mid \alpha^\Delta \Leftarrow \Pi \mid \mathbf{pro}\ s.\langle\, \mathcal{T}_1, \mathcal{T}_2\,\rangle,$$

where Δ and Π range over $\{1,2\}^*$. We identify $s \Leftarrow \varepsilon$ with s and $\alpha^\Delta \Leftarrow \varepsilon$ with α^Δ. Expressions of the form $\alpha^\Delta \Leftarrow \Pi$, where $\Pi \neq \varepsilon$, will be called *meta sends*. Let $TV(\mathcal{T})$ denote the set of all metavariables which occur in $\mathcal{T}$ and let $FS(\mathcal{T})$ denote the set of all sends $s \Leftarrow \Pi$ which occur free in $\mathcal{T}$ (i.e. s is free in $\mathcal{T}$).

A meta scheme in which all sends are free is called a *quasi type scheme*. Observe that a quasi type scheme without metavariables is a quasi type. A *stripped scheme* is a meta scheme in which bindings of sends occur only at the top, i.e. $\mathcal{T}$ is a stripped scheme if it is of the form: $c, s \Leftarrow \Pi, \alpha^\Delta \Leftarrow \Pi$, or $\mathbf{pro}\ s.\langle \mathcal{T}_1, \mathcal{T}_2 \rangle$, where $\mathcal{T}_1$ and $\mathcal{T}_2$ are quasi type schemes. Hence a stripped scheme without metavariables is a stripped term.

Most of the definitions which are applicable to terms are applicable to meta schemes as well . For example, the definition of formal field selection can be extended to quasi type schemes by adding the clause for metavariables:

- $(\alpha^\Delta \Leftarrow \Gamma).\Pi = \alpha^\Delta \Leftarrow \Gamma\Pi$

An *instantiation* of a meta scheme $\mathcal{T}$ is a pair (E, S), where E is an environment and S is a substitution which assigns to every metavariable $\alpha^\Delta \in TV(\mathcal{T})$ a type ρ such that

$$(\rho.\Delta)_E = \rho.$$

For a substitution S which assigns types to metavariables in $TV(\mathcal{T})$, by $\mathcal{T}\{S\}$ we denote the term obtained by substituting types for metavariables in $\mathcal{T}$. The definition of $\mathcal{T}\{S\}$ is by straightforward induction, the only nontrivial clause being this one:

$$(\alpha^\Delta \Leftarrow \Pi)\{S\} = S(\alpha^\Delta).\Pi$$

Of course, we perform α-conversion, when necessary, in order to avoid send capture. Clearly when $\mathcal{T}$ is a quasi type scheme then $\mathcal{T}\{S\}$ is a quasi type. Similarly for stripped schemes.

For a stripped scheme $\mathcal{T}$, the *value* of $\mathcal{T}$ in an instantiation (E, S) for $\mathcal{T}$ is the stripped term $(\mathcal{T}\{S\})_E$.

5.1 Equivalence of meta schemes

Given a stripped scheme $\mathcal{T}$ and a metavariable α^Δ, take any decomposition $\Delta = \Delta_1\Delta_2$, where $\Delta_1, \Delta_2 \in \{1,2\}^*$. Let $\mathcal{T}' = \mathcal{T}[\alpha^{\Delta_2\Delta_1}.\Delta_2/\alpha^\Delta]$, where $\alpha^{\Delta_2\Delta_1}$ is a fresh metavariable. Then $\mathcal{T}'$ is said to be obtained from $\mathcal{T}$ be a *cyclic shift*. We claim that $\mathcal{T}$ and $\mathcal{T}'$ should be considered equivalent. There is one-to-one correspondence between instantiations of $\mathcal{T}$ and $\mathcal{T}'$ which preserves values of $\mathcal{T}$ and $\mathcal{T}'$. More specifically we have:

(P1) If (E, S) is an instantiation of $\mathcal{T}$ and $S(\alpha^\Delta) = \rho$, then for

$$S' = (S - \{(\alpha^\Delta, \rho)\}) \cup \{(\alpha^{\Delta_2\Delta_1}, (\rho.\Delta_1)_E)\}$$

the pair (E, S') is an instantiation of $\mathcal{T}'$ and $(\mathcal{T}\{S\})_E = (\mathcal{T}'\{S'\})_E$.

(P2) And conversely, if (E, S') is an instantiation of $\mathcal{T}'$ and $S'(\alpha^{\Delta_2\Delta_1}) = \rho$, then for

$$S = (S' - \{(\alpha^{\Delta_2\Delta_1}, \rho)\}) \cup \{(\alpha^{\Delta}, (\rho.\Delta_2)_E)\}$$

the pair (E, S) is an instantiation of $\mathcal{T}$ and $(\mathcal{T}\{S\})_E = (\mathcal{T}'\{S'\})_E$.

Observe that (E, S') in (P1) is indeed an instantiation of $\mathcal{T}'$. Using twice Lemma 1 we obtain

$$((\rho.\Delta_1)_E.\Delta_2\Delta_1)_E = (\rho.\Delta_1\Delta_2\Delta_1)_E = ((\rho.\Delta_1\Delta_2)_E.\Delta_1)_E = (\rho.\Delta_1)_E.$$

Also the equality $(\mathcal{T}\{S\})_E = (\mathcal{T}'\{S'\})_E$ holds by a similar argument. An occurrence of ρ in the left side which comes from substituting ρ for α^{Δ} corresponds to an occurrence of $((\rho.\Delta_1)_E.\Delta_2)_E$ in the right side. By Lemma 1 both are equal. Justification of (P2) is similar.

Meta schemes $\mathcal{T}_1$ and $\mathcal{T}_2$ are said to be *equivalent* if

- For every instantiation (E, S) of $\mathcal{T}_1$ there is a substitution S' such that (E, S') is an instantiation of $\mathcal{T}_2$ and $(\mathcal{T}_1\{S\})_E = (\mathcal{T}_2\{S'\})_E$. And
- For every instantiation (E, S) of $\mathcal{T}_2$ there is a substitution S' such that (E, S') is an instantiation of $\mathcal{T}_1$ and $(\mathcal{T}_1\{S'\})_E = (\mathcal{T}_2\{S\})_E$.

The above remarks show that cyclic shift preserves scheme equivalence. Another transformation which preserves equivalence is *cycle contraction*. This consists in replacing one or more occurrences of the expression $\alpha^{\Delta}.\Delta$ by α^{Δ}.

A meta scheme $\mathcal{T}$ is said to be *typable* if there is an instantiation (E, S) of $\mathcal{T}$ and a type τ such that $E \vdash \mathcal{T}\{S\} : \tau$ is derivable.

6 The rewrite system

The aim of this section is to give rewrite rules for transforming a given stripped scheme into a quasi type scheme. The transformation is going to be a partial function, i.e. for some stripped schemes there will be no corresponding quasi type scheme. We are going to describe two kinds of redexes: reducible and cyclic. First we need an auxiliary definition with which we can define the redexes.

6.1 The projection and remainder functions

For a stripped scheme $\mathcal{T}$ we define a pair of functions: a *projection function* $p_{\mathcal{T}} : \{1, 2\}^* \to \mathcal{T}$ and a *remainder function* $r_{\mathcal{T}} : \{1, 2\}^* \to \{1, 2\}^*$. Intuitively $p_{\mathcal{T}}(\Pi)$ is a node of $\mathcal{T}$ which is obtained by traveling in $\mathcal{T}$ along Π, subject to the following conditions. If Π is contained in $\mathcal{T}$ then we terminate at Π. Otherwise we apply the following rules for passing through a leaf Γ:

- if Γ is labeled by a self t which is bound at node Δ, then the next step starts at node Δ.

– if Γ is labeled by a place-holder, then we return to this node in the next step (and thus in all following steps).
– if Γ is labeled by a free send, or a meta send, or a top send, then we terminate at this node, i.e. no next step is possible.

Then $r_{\mathcal{T}}(\Pi)$ is what remains of Π upon the termination of the travel through $\mathcal{T}$. The formal definition follows.

Case A: ($\Pi \in \mathcal{T}$)

$$p_{\mathcal{T}}(\Pi) = \Pi$$
$$r_{\mathcal{T}}(\Pi) = \varepsilon$$

Case B: ($\Pi_1\Pi_2 \in \mathcal{T}$ is a leaf labeled t, Π_1 is labeled t, and $\Pi_2 \neq \varepsilon$ and $\Delta \neq \varepsilon$)

$$p_{\mathcal{T}}(\Pi_1\Pi_2\Delta) = p_{\mathcal{T}}(\Pi_1\Delta)$$
$$r_{\mathcal{T}}(\Pi_1\Pi_2\Delta) = r_{\mathcal{T}}(\Pi_1\Delta)$$

Case C: ($\Delta \neq \varepsilon$ and $\Pi \in \mathcal{T}$ is a leaf labeled by one of the following: a free send, a top send, a meta send)

$$p_{\mathcal{T}}(\Pi\Delta) = \Pi$$
$$r_{\mathcal{T}}(\Pi\Delta) = \Delta$$

Case D: ($\Delta \neq \varepsilon$ and $\Pi \in \mathcal{T}$ is a leaf labeled by a place-holder)

$$p_{\mathcal{T}}(\Pi\Delta) = \Pi$$
$$r_{\mathcal{T}}(\Pi\Delta) = \varepsilon$$

6.2 Reducible top sends

A top send $s \Leftarrow \Pi$ is said to be *reducible* if $p_{\mathcal{T}}(\Pi)$ is not an occurrence of a top send.

Among reducible top sends are those which we call inconsistent. A top send $s \Leftarrow \Pi$ is said to be *inconsistent* if $p_{\mathcal{T}}(\Pi) = i\Delta$, for some i and Δ, and $s \Leftarrow \Pi$ occurs in $\mathcal{T}_i.\Delta$. A reducible send which is not inconsistent is called *consistent*.

Lemma 2. *If a stripped scheme contains an inconsistent top send, then it is not typable.*

Let $s \Leftarrow \Pi$ be a reducible top send in $\mathcal{T}$ and let $p_{\mathcal{T}}(\Pi) = i\Delta$ and $r_{\mathcal{T}}(\Pi) = \xi$. Reduction of $s \Leftarrow \Pi$ consists in replacing every occurrence of $s \Leftarrow \Pi$ in $\mathcal{T}$ by $\mathcal{T}_i.\Delta\xi$.

It follows that $\Delta \in \mathcal{T}_i$ and we have the following two possibilities:

1. Δ is an internal node of $\mathcal{T}_i$. Then $\xi = \varepsilon$.
2. Δ is a leaf in $\mathcal{T}_i$. Then the label of this leaf is one of the following:
   2a. A free send in $\mathcal{T}$.
   2b. A meta send.
   2c. A place-holder.

In each case ((1) or (2)) it follows that $\mathcal{T}_i.\Delta\xi$ does not contain new top sends, i.e. there may be new occurrences of top sends after the reduction, but the set of all different top sends after the reduction is not larger than before. In fact, when the reducible top send is not inconsistent, then the number of top sends after the reduction decreases by one.

The intuitions behind the previous concepts are:

- an inconsistent reducible top send addresses a subtree of the tree representing the term in question which contains the top send itself, meaning that the top send's type should contain properly itself (see the first example in Section 2);
- a consistent reducible top send is one for which we can mimic the evaluation process, by substituting it with the subtree it addresses. This way we make a step towards a send-free term, which will correspond to the quasi type scheme.

6.3 Cyclic top sends

Let $S_{\mathcal{T}}$ be the set of all occurrences of top sends in $\mathcal{T}$. The projection and remainder functions give rise to two mappings $\widehat{p}_{\mathcal{T}} : S_{\mathcal{T}} \to \mathcal{T}$ and $\widehat{r}_{\mathcal{T}} : S_{\mathcal{T}} \to \{1,2\}^*$. For $\Gamma \in S_{\mathcal{T}}$, if the label of Γ is $s \Leftarrow \Pi$, then

$$\widehat{p}_{\mathcal{T}}(\Gamma) = p_{\mathcal{T}}(\Pi), \quad \text{and} \quad \widehat{r}_{\mathcal{T}}(\Gamma) = r_{\mathcal{T}}(\Pi).$$

A top send $s \Leftarrow \Pi$ is said to be *cyclic* if for one of its occurrences $\Gamma \in S_{\mathcal{T}}$ we have

$$\widehat{p}^{\,k}_{\mathcal{T}}(\Gamma) = \Gamma, \tag{7}$$

for some $k \geq 1$. It follows that the occurrence Γ is unique. We call it a *cyclic occurrence* of $s \Leftarrow \Pi$. The least k satisfying (7) will be called the *period* of $s \Leftarrow \Pi$. A *cyclic coefficient* of a cyclic send is the word $\widehat{r}_{\mathcal{T}}(\widehat{p}^{\,k-1}_{\mathcal{T}}(\Gamma)) \cdots \widehat{r}_{\mathcal{T}}(\widehat{p}_{\mathcal{T}}(\Gamma))\widehat{r}_{\mathcal{T}}(\Gamma)$, where Γ is the cyclic occurrence and k is the period of $s \Leftarrow \Pi$.

Let $s \Leftarrow \Pi$ be a cyclic top send in $\mathcal{T}$ and let Δ be its cyclic coefficient. Reduction of $s \Leftarrow \Pi$ consists in replacing every occurrence of $s \Leftarrow \Pi$ in $\mathcal{T}$ by α^{Δ}, where α^{Δ} is a fresh metavariable not occurring in $\mathcal{T}$.

It follows that sends which label the nodes $\widehat{p}_{\mathcal{T}}(\Gamma), \ldots, \widehat{p}^{\,k-1}_{\mathcal{T}}(\Gamma)$ are also cyclic in $\mathcal{T}$. After the reduction the send labeling the node $\widehat{p}^{\,k-1}_{\mathcal{T}}(\Gamma)$ becomes reducible, while the other sends are not subject to immediate reduction in the new scheme.

The intuitions behind a cyclic send is that it represents an infinite computation (infinite computations are universally accepted in object-oriented calculi, see typical examples in [1, 12]). Essentially, it refers to itself within a certain number of computation steps, which is the period.

6.4 Confluence and termination

Let $\mathcal{T}$ be a stripped scheme. Each top send in $\mathcal{T}$ is either cyclic or reducible. Every such top send is called a *redex*. To be more precise, a redex is a term (send) as such, not a single occurrence of that term. Clearly if $\mathcal{T}$ has no redexes then it is a quasi type scheme.

The main properties of the above rewrite system are collected in the next two results.

Theorem 6.1. *Let $\mathcal{T}$ be a stripped scheme.*

1. **(Termination)** *Let n be the number of top sends in $\mathcal{T}$. After n steps of reduction we either arrive at a quasi type scheme, or else we must have earlier detected an inconsistent reducible send.*
2. **(Confluence)** *Let $\mathcal{T}'$ and $\mathcal{T}''$ be two quasi type schemes obtained from $\mathcal{T}$ by a sequence of reductions. Then $\mathcal{T}'$ and $\mathcal{T}''$ are equivalent.*

Theorem 6.2. *Let $\mathcal{T}$ be a stripped scheme. The following are equivalent.*

1. *$\mathcal{T}$ is typable.*
2. *There exists a sequence of reductions which transforms $\mathcal{T}$ into a quasi type scheme.*
3. *Every sequence of reductions transforms $\mathcal{T}$ into a quasi type scheme.*

Moreover, if $\mathcal{T}^{\#}$ is a quasi type scheme obtained from $\mathcal{T}$ by a sequence of reductions and (E, S) is any instantiation of $\mathcal{T}^{\#}$ such that $(\mathcal{T}^{\#}\{S\})_E$ is a type, say τ, then

$$E \vdash \mathcal{T}\{S\} : \tau$$

is derivable.

Every quasi type scheme $\mathcal{T}^{\#}$ obtained from a stripped scheme $\mathcal{T}$ by a sequence of reductions will be called a *normal form* of $\mathcal{T}$. It follows from Theorem 6.1 that every stripped scheme has at most one normal form, up to scheme equivalence.

Example 1. $M = \mathbf{pro}\, s\langle\, \mathbf{pro}\, t\langle\, d, s \Leftarrow 1\,\rangle, c\,\rangle$: M is a stripped term and the atomic send $s \Leftarrow 1$ is reducible and inconsistent, because $p_M(1) = 1$ and $s \Leftarrow 1$ occurs in the sub-tree M_1 (following the definition of Section 6.2). M is not typable by Lemma 2.

Example 2. $M = \mathbf{pro}\, s\langle\, s \Leftarrow 1, c\,\rangle$: M is a stripped term and $s \Leftarrow 1$ is a cyclic send. It can be solved using the technique of Section 6.3. This term is typable and a type is $\mathbf{pro}\, s\langle\, s, c\,\rangle$.

Example 3. $M = \mathbf{pro}\, s\langle\, s \Leftarrow 12, s \Leftarrow 112\,\rangle$: M is a stripped term and first we solve the cyclic send $s \Leftarrow 12$, obtaining the quasi-type schema $\mathbf{pro}\, s\langle\, \alpha^2, s \Leftarrow 112\,\rangle$. Now $s \Leftarrow 112$ becomes reducible and we get $\mathbf{pro}\, s\langle\, \alpha^2, \alpha^2.12\,\rangle$. By giving α^2 a type ρ such that $(\rho.2) = \rho$ (remember that the first send was cyclic, so its type must represent this), we can get as types of M: $\mathbf{pro}\, s\langle\, c, c\,\rangle$, $\mathbf{pro}\, s\langle\, \mathbf{pro}\, t\langle\, \mathbf{pro}\, x\langle\, y, z\,\rangle, t\,\rangle z\,\rangle$, etc.

7 Main Result

We define a partial map which assigns to a term M a quasi type scheme $\mathcal{T}_M$, called a *principal quasi type scheme* of M. It is defined by induction on M.

- $\mathcal{T}_c = c$
- $\mathcal{T}_s = s$

– $\mathcal{T}_{\mathbf{pro}\ s.\langle\, M_1, M_2 \,\rangle} = (\mathbf{pro}\ s\langle\, \mathcal{T}_{M_1}, \mathcal{T}_{M_2} \,\rangle)^{\#}$
– $\mathcal{T}_{M \Leftarrow i} = \mathcal{T}_M.i.$

The above recurrence equations have to be understood in such a way that the left hand side is defined iff the right hand side is defined.

The main result of this paper is the following theorem.

Theorem 7.3. (Principal quasi type theorem)

1. *M is typable iff $\mathcal{T}_M$ is defined.*
2. *If $\mathcal{T}_M$ is defined, then for every instantiation (E, S) of $\mathcal{T}_M$ such that $(\mathcal{T}_M\{S\})_E$ is a type we have*
$$E \vdash M : (\mathcal{T}_M\{S\})_E.$$
3. *The partial mapping $M \mapsto \mathcal{T}_M$ is computable. Therefore the problem of typability is decidable.*

8 Extensions

We have solved the type reconstruction problem for a system containing only the send operator to highlight the essential mathematical content of the problem itself. But the approach can be extended to deal with the method addition operator $\longleftarrow\!\!+$ of the C calculus [3], and with the *message-not-understood* run-time error, without changing the mathematical core of our solution. The override operator, though, remains an open problem (see in the conclusions).

Method addition. Firstly, we must extend the object syntax to include objects with an indefinite number of components. Then, since method addition is permitted only on proper objects, it is enough to extend appropriately the notion of "principal quasi type scheme" $\mathcal{T}_M$, in order to check, in the case of method addition, if the resulting (quasi) type of the object receiving the addition is a $\mathbf{pro}\ \langle \ldots \rangle$ (quasi) type and it does not contain the method to be added, together with checking that the method body is typable.

Message-not-understood. As a consequence of dealing with objects with more than two components plus the method addition, we could lift the constraint on the send operation, by allowing the invocation of whichever method, both on proper objects (external send) and on selves (self-inflicted send). In the external case, the right-hand side of the equation $\mathcal{T}_{M \Leftarrow i} = \mathcal{T}_M.i$ of the principal quasi type scheme would be satisfied (and $\mathcal{T}_{M \Leftarrow i}$ would be typable) if $\mathcal{T}_M$ were a quasi type scheme of the form $\mathbf{pro}\ \langle \ldots \rangle$ containing an i component, and the resulting quasi type scheme would be as in the two-method situation. A more difficult case is when the send is self-inflicted, i.e., if M is a self s: this case must be solved directly during the global process of going from the stripped term containing $M \Leftarrow i$ to its quasi type scheme, because we need to check if the subtree rooted at s has an i branch. In order to do so, for every top send $s \Leftarrow \Pi$ we must check that the branching described by Π exist in the subtree rooted at s.

9 Conclusion and future work

We have shown that decidable type reconstruction is possible for languages with nested selftype references. This is an important conclusion even if the language we solved the problem for contains only the send operator, because we do believe that it is the core of any other richer systems (i.e., systems including method addition and/or override), from the point of view of the type reconstruction.

Our result raises a number of further questions. Obviously, one wants to expand the analysis to the case of object languages with a more reasonable choice of operators.

Adding method addition must be still formalized, but we conjecture that is nothing more than careful bookwork. Dealing with *message-not-understood* appears to be more delicate, because it implies an extension of the algorithm as hinted above, but it does not change the techniques we use to detect and solve "loops", which are the central part of our solution.

Override is, instead, an open question at the moment, so far it can be only shown that adding method override makes the problem PTIME-hard. Intuitively, override, by substituting method bodies, may change the interrelationships among the cyclic top sends, inducing complex equational constraints — a very special case of second-order unification. As mentioned in the introduction, it looks like self-inflicted overrides are the main issue. Nevertheless, also external overrides introduce some difficulties. In order to type a method override on an object, we would need to compare the (quasi) type of the overriden (old) method body with the (quasi) type of the overridding (new) one, and only if they are "equal" the override is typable. Now, the problem lies in the fact that we still do not have a complete notion of principality for our typing, making not possible deciding equality among (quasi) types [5].

Even for the simple language we discussed above, there are still issues to be investigated. The naive algorithm, involving the construction of $\mathcal{T}_M$, is obviously not feasible, as it involves nested substitutions. Although we believe the problem is solvable in polynomial time, a workable implementation is still to be developed, and does not seem to be trivial.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.
2. Baader, F., Nipkow, T., *Term Rewriting and* All That, Cambridge University Press, 1998.
3. V. Bono. Extensible Objects: a Tutorial. In *Global Computing – Trento*, LNCS. Springer, 2003. To appear.
4. V. Bono and M. Bugliesi. Matching for the Lambda Calculus of Objects. *Theoretical Computer Science*, 1999.
5. V. Bono, M. Bugliesi, and S. Crafa. Typed Interpretations of Extensible Objects. ACM Transactions on Computational Logic, 2002.
6. G. Bracha, M. Odersky, D. Stoutamire and P.Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. of OOPSLA'98*.

[5] The equality between the type of the old body and of the overriding one is a typical requirement in object-oriented type systems when no subtyping is present.

7. K.B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.
8. K.B. Bruce. *Foundations of Object-Oriented Languages–Types and Semantics*. The MIT Press, 2002.
9. Michele Bugliesi and Santiago Pericas. Depth subtyping and type inference for object calculi. *Information and Computation*, 177(1):2–27, 2002.
10. W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proc. of ACM Symp. POPL'90*, pages 125–135. ACM Press, 1990.
11. P. Di Gianantonio, F. Honsell, and L. Liquori. A Lambda Calculus of Objects with Self-inflicted Extension. In *Proc. of ACM-SIGPLAN OOPSLA, International Symposium on Object Oriented, Programming, System, Languages and Applications*, pages 166–178. The ACM Press, 1998.
12. K. Fisher, F. Honsell, and J.C. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.
13. F. Henglein. Breaking through the n^3 barrier: Faster object type inference. *Theory and Practice of Object Systems (TAPOS)*, 5(1):57–72, 1999. Invited paper selected from 4th Int'l Workshop on Foundations of Object-Oriented Languages (FOOL 4), 1997.
14. A.V. Hense. *Polymorphic Type Inference for Object-Oriented Programming Languages*. Pirrot Verlag, 1994.
15. J. C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proc. of ACM Symp. POPL*, pages 109–124. ACM Press, 1990.
16. J. Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995.
17. J. Palsberg and T. Jim. Type inference with simple selftypes is NP-complete. *Nordic Journal of Computing*, 4(3):259–286, 1997.