# Type Inference with Simple Selftypes is NP-complete

Jens Palsberg* Trevor Jim†

November 27, 2004

**Abstract**

The metavariable *self* is fundamental in object-oriented languages. Typing self in the presence of inheritance has been studied by Abadi and Cardelli, Bruce, and others. A key concept in these developments is the notion of *selftype*, which enables flexible type annotations that are impossible with recursive types and subtyping. Bruce et al. demonstrated that, for the language TOOPLE, type checking is decidable. Open until now is the problem of type inference with selftype.

In this paper we present a simple type system with selftype, recursive types, and subtyping, and we prove that type inference is *NP*-complete. With recursive types and subtyping alone, type inference is known to be P-complete. Our example language is the object calculus of Abadi and Cardelli. Both our type inference algorithm and our lower bound are the first such results for a type system with selftype.

Categories and Subject Descriptors: D.3.2 [Programming Languages]: Languages Classifications—object-oriented languages; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—type structure.

General Terms: Languages, Theory.

Additional Key Words and Phrases: type inference, constraints.

*Purdue University, Dept of Computer Science, W Lafayette, IN 47907, palsberg@cs.purdue.edu.

†Department of Computer and Information Science, University of Pennsylvania, 200 S. 33rd Street, Philadelphia, PA 19104–6389, tjim@saul.cis.upenn.edu.

# 1 Introduction

The metavariable *self* is fundamental in object-oriented languages. It may be used in a method to refer to the object executing the method. Since methods can be inherited, the meaning of self cannot be determined statically. This phenomenon is a key reason why static typing for object-oriented languages is a challenging problem. For a denotational semantics of inheritance and self, see for example [8].

Typing self in the presence of inheritance has been studied by Abadi and Cardelli [3, 2, 1, 4], Bruce [6, 7], Mitchell, Honsell, and Fisher [12, 13], Palsberg and Schwartzbach [15, 16], and others. These developments all identify a need to give self a special treatment, as illustrated by the following standard example.

```
object Point
  ...
  method move
    ...
    return self
  end
end

object ColorPoint extends Point
  ...
  method setcolor
    ...
  end
end

-- Main program:

ColorPoint.move.setcolor
```

The object `ColorPoint` is defined by inheritance from `Point`: it extends `Point` with the method `setcolor`. The only significant aspect of the objects is that the `move` method returns self. Now consider the main program. It executes without errors, but is it typable? With most conventional type systems, the answer is: no! For example, suppose we use a C++ style of types such that we can annotate the method `move` with the return type `Point`. Then the expression `ColorPoint.move` has the type `Point`, and thus `ColorPoint.move.setcolor` is not type-correct, since `Point` does not have a `setcolor` method. In C++, we would have to insert an unsafe type cast to make the program type check.

One way of typing the example *without* using type casts is to introduce *selftype*, that is, a special notation for "the type of self." If we annotate the `move` method with "selftype" as the return type, `ColorPoint.move` will have the *same* type as `ColorPoint`, so `ColorPoint.move.setcolor` is type-correct.

Type systems with selftype have been presented by Abadi and Cardelli [4], Bruce et al. [6, 7], Mitchell, Honsell, and Fisher [12, 13], and others. A type system with selftype is used in the language Eiffel [11].

In this paper, we address the following fundamental question:

**Fundamental question.** Is type inference with selftype feasible?

Of course, the answer will depend on the exact details of the type system. And there is no common agreement on the "right" type system with selftype. One of the design issues is the notation for selftype. Bruce et al. use the keyword `Mytype` to refer to the type of self, and similarly, Eiffel uses the notation `like Current`. Both `Mytype` and `like Current` refer to the "selftype" of the *innermost* enclosing object; in these systems there is no way of referring to the "selftype" of other enclosing objects. The systems of Abadi and Cardelli [4], and Mitchell, Honsell, and Fisher [12, 13], are more expressive, binding a name for "selftype" in each object type.

Another design issue is the choice of type rules. For example, when comparing the type rules of Abadi and Cardelli [4] with those of Bruce et al. [6, 7], we find both striking similarities, such as the rules for message send, and significant differences. Both of these type systems have been proved sound, and Bruce et al. have shown that type checking is decidable in their language, TOOPLE [7]. However, we know of no type inference algorithm for any system with selftype.

Our approach to type inference with selftype is to begin with a system of object types where type inference is well understood, and then consider the simplest possible extension with selftype. We use Abadi and Cardelli's system of recursive object types and subtyping as our starting point; type inference in this system is P-complete [14]. The only type constructor in this type system is the one for object types. The type of an object is of the form $[l_i : B_i \ ^{i \in 1..n}]$, where each $l_i$ is a method name and each $B_i$ is a type. The form of subtyping is the "width" subtyping of Abadi and Cardelli, that is, if $A$ is a subtype of $B$, then $A$ has at least the fields of $B$, and for common fields, $A$ and $B$ have the same field type. The type system does not contain function types, base types, etc. Moreover, there are no contravariance or atomic subtyping in the type system, so the complexity results on type inference of Tiuryn [17] and Hoang and Mitchell [9] do not apply. Of course, if we introduce more constructs, then the upper and lower complexity bounds for type inference may change.

Our extension of the Abadi/Cardelli type system is based on two design decisions, both aiming for the simplest possible extension. The first decision is to use the syntax selftype, rather than binding a name for selftype in each object type. The second decision is related to the observation that the meaning of selftype is context dependent. We decree that each occurrence of selftype "comes with its context," so that its meaning can be recovered. Specifically, in our type system selftype can only appear as a component of an object type, and never in isolation. Thus in a typing judgment $E \vdash a : A$, we prohibit the environment $E$ from mapping any variable to selftype, and our typing rules will guarantee that the derived type $A$ is not selftype.

Our rule for typing a message send, $a.l$, is as follows.

$$\frac{E \vdash a : A}{E \vdash a.l : B\{A\}} \quad (\text{where } A \leq [l : B]) \ .$$

Here $a$ is an object, $l$ is a method name, $A$ and $B$ are types, and the notation $B\{A\}$ is defined

$$B\{A\} = \begin{cases} A & \text{if } B = \mathsf{selftype} \\ B & \text{otherwise.} \end{cases}$$

The use of the notation $B\{A\}$ guarantees that selftype cannot be derived as the type of $a.l$. Two instances of the rule are

$$\frac{E \vdash a : [l : [\,]]}{E \vdash a.l : [\,]}, \qquad \text{and} \qquad \frac{E \vdash a : [l : \mathsf{selftype}]}{E \vdash a.l : [l : \mathsf{selftype}]}.$$

In the first instance, $B = [\,]$ is not selftype, so we conclude $a.l : B$. In the second instance, $B = \mathsf{selftype}$, so we instead conclude $a.l : A$, where $a : A$

and $A = [l : \mathsf{selftype}]$; notice that we use the side condition

$$A \leq [l : \mathsf{selftype}]$$

to determine that the meaning of selftype is $A$.

Let us write the type of `Point` as $[\mathit{move} : \mathsf{selftype}]$ and the type of `ColorPoint` as $[\mathit{move} : \mathsf{selftype},\ \mathit{setcolor} : \mathit{Void}]$. With the rule above, we can type the expression `ColorPoint.move.setcolor` as follows:

$$\cfrac{\cfrac{\emptyset \vdash \texttt{ColorPoint} : [\mathit{move} : \mathsf{selftype},\ \mathit{setcolor} : \mathit{Void}]}{\emptyset \vdash \texttt{ColorPoint.move} : [\mathit{move} : \mathsf{selftype},\ \mathit{setcolor} : \mathit{Void}]}}{\emptyset \vdash \texttt{ColorPoint.move.setcolor} : \mathit{Void}}\ .$$

**Our Result** We prove that type inference for our type system with selftype, recursive types, and subtyping is *NP*-complete. With recursive types and subtyping alone, type inference is known to be P-complete [14]. Intuitively, the type inference problem is in *NP* because we can first guess which methods should be annotated with selftype as the return type, and then solve the remaining type inference problem in polynomial time. The *NP*-hardness emphasizes that there is no efficient way of finding a successful such guess. Both our type inference algorithm and our lower bound are the first such results for a type system with selftype. Our *NP*-hardness result directly contradicts the intuition that "use selftype whenever possible; it only makes typing easier." Certainly, selftype increases expressiveness, but it must be used judiciously. In particular, a feasible greedy algorithm for placing selftypes does not exist for our type system. See Section 6 for an illustration of this.

**Implications** In slogan-form, our result reads:

polynomial time type inference + a tiny drop of selftype = *NP*-complete

This suggests that the answer to our fundamental question is: "no, type inference with selftype is not feasible." In contrast to ML where type inference, in spite of being *EXPTIME*-complete, is fast for the programs that are written in practice, the type inference problem for our type system seems to require exhaustive search regardless of the form of the input program. Type *checking* in our system of simple selftypes is in polynomial time. Thus, selftype seems to be a construct which in practice should be used in languages with *explicit* typing.

**Future Work** We have been unable to establish a connection between our type system and the seemingly more expressive type systems of Abadi and Cardelli [4], and Bruce et al [6, 7]. This means that although a "tiny drop of selftype" gives *NP*-hardness, one can imagine that "a bigger drop of selftype" may or may not give *NP*-hardness. In general, one may ask if there is any sound type system at all which types at least what can be typed by the type system in this paper, and which has type inference in polynomial time. (For an investigation of tractable extensions of $F_{\leq}$, see [18].) Moreover, one may test the robustness of the NP-completeness result and the proof technique by extending the calculus with, say, object extension. We are so far unable to provide an intuition which directly explains why the types rules are able to code SAT. Our proof of NP-hardness begins by reducing SAT to a particularly simple form of constraint system (Definition 5.1 and Lemma 5.6) in which Boolean variables and a restricted form of conditional constraints can be encoded. Afterwards we show (Lemma 5.7) that the type rules can code such constraints.

**Paper Outline** In the next section we briefly recall Abadi and Cardelli's calculus, and in Section 3 we present our new type system. In Section 4 we prove that the type inference problem is log-space reducible to a constraint problem which can be solved in *NP* time. In Section 5 we prove that the type inference problem is *NP*-complete. Finally, in Section 6 we illustrate some of the constructions in the paper. We use an example program which is typable with selftype but not without.

# 2 Abadi and Cardelli's Object Calculus

We now present Abadi and Cardelli's untyped object calculus, called the $\varsigma$-calculus. We use $a$, $b$, $c$, $o$ to range over $\varsigma$-terms, which are defined by the following grammar.

| | | | |
|---|---|---|---|
| $a$ ::= | $x$ | | variable |
| $\mid$ | $[l_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$ | ($l_i$ distinct) | object |
| $\mid$ | $a.l$ | | field selection / method invocation |
| $\mid$ | $(a.l \Leftarrow \varsigma(x)b)$ | | field update / method override |

An object $[l_i = \varsigma(x_i)b_i \ ^{i\in 1..n}]$ has method names $l_i$ and methods $\varsigma(x_i)b_i$. The order of the methods does not matter. Each method binds a name that means self. Thus, in a method $\varsigma(x)b$, $x$ is self and $b$ is the body. Since the names for self can be chosen to be different and since objects can be nested, one can refer to any enclosing object, as in the Beta language [10].

Abadi and Cardelli define a term rewriting operational semantics by the following rules.

- If $o \equiv [l_i = \varsigma(x_i)b_i \ ^{i\in 1..n}]$, then, for $j \in 1..n$,
  - $o.l_j \ \rightsquigarrow \ b_j[x_j := o]$, and
  - $(o.l_j \Leftarrow \varsigma(y)b) \ \rightsquigarrow \ o[l_j \leftarrow \varsigma(y)b]$.
- If $b \rightsquigarrow b'$ then $a[b] \rightsquigarrow a[b']$.

Here, $b_j[x_j := o]$ denotes the $\varsigma$-term $b_j$ with $o$ substituted for free occurrences of $x_j$ (renaming bound variables to avoid capture); and $o[l_j \leftarrow \varsigma(y)b]$ denotes the $\varsigma$-term $o$ with the $l_j$ field replaced by $\varsigma(y)b$. A *context* is an expression with one hole, and $a[b]$ denotes the term formed by replacing the hole of the context $a[\cdot]$ by the term $b$ (possibly capturing free variables in $b$).

A $\varsigma$-term is said to be an *error* if it is irreducible and it contains either $o.l_j$ or $(o.l_j \Leftarrow \varsigma(y)b)$, where $o \equiv [l_i = \varsigma(x_i)b_i \ ^{i\in 1..n}]$, and $o$ does *not* contain an $l_j$ method.

For example, if $o \equiv [l = \varsigma(x)x.l]$, then the expression $o.l$ yields the infinite computation

$$o.l \rightsquigarrow (x.l)[x := o] \equiv o.l \rightsquigarrow \cdots,$$

the expression $o.m$ is an error, and the expression $o.l.m$ yields the infinite computation

$$o.l.m \rightsquigarrow ((x.l)[x := o]).m \equiv o.l.m \rightsquigarrow \cdots.$$

The rewrite system is confluent.

# 3 The Type System

The following type system for the $\varsigma$-calculus catches errors statically, that is, rejects all programs that may yield errors.

We use $U$, $V$ to range over type variables drawn from some possibly infinite set $\mathcal{U}$; $l$, $m$, $\ldots$ to range over labels drawn from some possibly infinite set $\mathcal{N}$ of method names; and $A$, $B$ to range over types defined by the grammar

$$B ::= \mathsf{selftype} \mid [l_i : B_i \;^{i \in 1..n}] \mid V \mid \mu(V)B.$$

For reasons explained momentarily, strings of the form

$$\cdots (\mu(V_1) \cdots \mu(V_n) V_1) \cdots$$

are not considered to be types; all other strings generated from the grammar are valid object types.

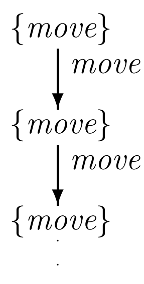
We identify types with their infinite unfoldings under the rule

$$\mu(V)B \to B[V := \mu(V)B] \; .$$

Because we do not allow types like $\mu(V_1) \cdots \mu(V_n) V_1$, the rule eliminates all uses of $\mu$ in types, so that types are a class of regular trees over the alphabet

$$\Sigma = \{\mathsf{selftype}\} \cup \mathcal{U} \cup \{N \subseteq \mathcal{N} \mid N \text{ is finite}\},$$

with edges labeled by method names. For example, the type $\mu(X)[\mathit{move} : X]$ is identified with the following tree.




The set of types over $\Sigma$ is denoted $T_\Sigma$. We use strings over $\mathcal{N}^*$ to identify subtrees of types, writing $A \downarrow \alpha$ for the subtree of $A$ identified by $\alpha$, if any. Thus a type $A$ can be considered a partial function from $\mathcal{N}^*$ to $\Sigma$: $A(\alpha)$ is the symbol at the root of the tree $A \downarrow \alpha$. We write $\mathcal{D}(A)$ for the domain of $A$ when it is thought of as a function in this way.

The set of object types is ordered by the subtyping relation $\leq$ as follows. First,

$$U \leq U \quad \text{for } U \in \mathcal{U}$$
$$\mathsf{selftype} \leq \mathsf{selftype}$$

and second, if $A$ and $B$ both are of the form $[l_i : B_i \;^{i \in 1..n}]$, then

$$A \leq B \quad \text{if and only if} \quad \forall l \in \mathcal{N}: \quad l \in \mathcal{D}(B) \Rightarrow (l \in \mathcal{D}(A) \;\wedge\; A{\downarrow}l = B{\downarrow}l) \;.$$

Intuitively, if $A \leq B$, then $A$ may contain more fields than $B$, and for common fields, $A$ and $B$ must have the same type. For example, $[l : A, m : B] \leq [l : A]$, but $[l : [m : A]] \not\leq [l : [\,]]$. Thus subtyping reduces to equivalence of recursive types, which in turn reduces to equivalence of finite state automata. Notice that $\leq$ is a partial order, and if $A \leq B$, then $\mathcal{D}(B) \subseteq \mathcal{D}(A)$.

As an aside, one might wonder why we do not relax the definition of $A \leq$B to allow $A \downarrow l \leq B \downarrow l$, instead of $A \downarrow l = B \downarrow l$. Intuitively, this relaxation would allow "depth" subtyping. Unfortunately, this would make the type rules below unsound [5].

If $A$ and $B$ are object types, then $B\{A\}$ is defined

$$B\{A\} = \begin{cases} A & \text{if } B = \mathsf{selftype} \\ B & \text{otherwise.} \end{cases}$$

The typing rules below allow us to derive judgments of the form $E \vdash a : A$, where $E$ is a type environment, $a$ is a $\varsigma$-term, and $A$ is an object type. We do not allow $E$ to assign any variable the type $\mathsf{selftype}$, as this would be a use of $\mathsf{selftype}$ "out of context." Similarly, our rules will insure that $A$ is not $\mathsf{selftype}$ in any derivable judgment $E \vdash a : A$.

$$E \vdash x : A \quad (\text{provided } E(x) = A) \tag{1}$$

$$\frac{E \vdash a : A}{E \vdash a.l : B\{A\}} \quad (\text{where } A \leq [l : B]) \tag{2}$$

$$\frac{E[x_i \leftarrow A] \vdash b_i : B_i\{A\} \quad \forall i \in 1..n}{E \vdash [l_i = \varsigma(x_i)b_i \;^{i \in 1..n}] : A} \quad (\text{where } A = [l_i : B_i \;^{i \in 1..n}]) \tag{3}$$

$$\frac{E \vdash a : A \quad E[x \leftarrow A] \vdash b : B}{E \vdash a.l \Leftarrow \varsigma(x)b : A} \quad (\text{where } A \leq [l : B]) \tag{4}$$

$$\frac{E \vdash a : A}{E \vdash a : B} \quad (\text{where } A \leq B) \tag{5}$$

The first four rules express the typing of each of the four constructs in the object calculus and the last rule is the rule of subsumption. The type rules may be understood as a generalization of those introduced by Abadi and

Cardelli in [3] and studied further by Palsberg in [14]. Specifically, if selftype is never used, then $B\{A\} = B$ and the rules take the form used in [14]. Thus, the type rules above type more terms than the rules in [14].

**Theorem 3.1 (Subject Reduction)** *If $E \vdash a : A$ and $a \rightsquigarrow a'$, then $E \vdash a' : A$.*

*Proof.* By induction on the structure of the derivation of $E \vdash a : A$. □

We say that a term $a$ is *well-typed* if $E \vdash a : A$ is derivable for some $E$ and $A$. Along with the observation that no error is well-typed, Subject Reduction implies that *a well-typed term cannot go wrong.*

Note in the method override rule (4) that the type $B$ cannot be selftype; it appears in the antecedent as the derived type of the judgment $E[x \leftarrow A] \vdash b : B$, and no derived type is selftype in our system. Therefore, it is not possible for a method returning selftype to be overridden in our type system. At first, this seems like an overly severe restriction, and we might be tempted instead to use the rule

$$\frac{E \vdash a : A \quad E[x \leftarrow A] \vdash b : B\{A\}}{E \vdash a.l \Leftarrow \varsigma(x)b : A} \quad \text{(where } A \leq [l : B]\text{)}.$$

However, this rule is not sound (Subject Reduction fails), as noted by Abadi (personal communication). Both Abadi and Cardelli, and Bruce et al., define sound extensions of our rule, but their extensions require that typing judgments include syntactic assumptions that resolve the meaning of selftype. While not as expressive, our system is considerably more simple.

By a simple induction on typing derivations, we obtain the following syntax-directed characterization of typings. The characterization will be used in the next section, where we reduce type inference to the solution of a particular system of constraints.

**Lemma 3.2 (Characterization of Typings)** *$E \vdash c : A$ if and only if one of the following cases holds:*

- *$c = x$ and $E(x) \leq A$;*
- *$c = a.l$, and for some $B$ and $C$, $E \vdash a : B$, $B \leq [l : C]$ and $C\{B\} \leq A$;*

- $c = [l_i = \varsigma(x_i)b_i \ ^{i\in 1..n}]$, *and for some* $C$ *and* $B_i$ *for* $i \in 1..n$, $E[x_i \leftarrow C] \vdash b_i : B_i\{C\}$, *and* $C = [l_i : B_i \ ^{i\in 1..n}] \leq A$; *or*

- $c = a.l \Leftarrow \varsigma(x)b$, *and for some* $B$ *and* $C$, $E \vdash a : B$, $E[x \leftarrow B] \vdash b : C$, $B \leq [l : C]$ *and* $B \leq A$.

# 4 Type Inference in *NP* time

In this section we prove that the following *type inference problem* is computable in *NP* time.

> **Type inference**: given a $\varsigma$-term $c$, either produce an environment $E$ and type $A$ such that $E \vdash c : A$, or halt and fail if no such $E$ and $A$ exist.

We do this by first reducing the type inference problem to solving a finite system of type constraints (Lemma 4.2), and then showing that the constraints can be solved in *NP* time (Corollary 4.4).

We work with *constraints* of the form $W_1 \leq W_2$, where $W$'s are defined by the grammar

$$W ::= U \mid [l_i : U_i \ ^{i\in 1..n}] \mid \mathsf{selftype} \mid U\{U'\}$$

We use $U\{U'\}$ here in a syntactic way, in contrast with its use in the typing rules of the last section. This syntax is eliminated as follows: for any function $L : \mathcal{U} \rightarrow T_\Sigma$, we define $\tilde{L}$ by

$$\tilde{L}(W) = \begin{cases} L(U) & \text{if } W = U\{U'\} \text{ and } L(U) \neq \mathsf{selftype} \\ L(U') & \text{if } W = U\{U'\} \text{ and } L(U) = \mathsf{selftype} \\ L(U) & \text{if } W = U \\ [l_i : L(U_i) \ ^{i\in 1..n}] & \text{if } W = [l_i : U_i \ ^{i\in 1..n}] \\ \mathsf{selftype} & \text{if } W = \mathsf{selftype} \end{cases}$$

**Definition 4.1** For any denumerable set $\mathcal{U}$ of variables and subset $\mathcal{U}_0 \subseteq \mathcal{U}$, an *S-system (selftype-system) over* $\mathcal{U}$ *and* $\mathcal{U}_0$ is a finite set of constraints whose variables are drawn from $\mathcal{U}$. A *solution* to an S-system $\mathcal{C}$ is a function $L : \mathcal{U} \rightarrow T_\Sigma$ such that for all $W \leq W'$ in $\mathcal{C}$, $\tilde{L}(W) \leq \tilde{L}(W')$, and such that for all $U \in \mathcal{U}_0$, $L(U) \neq \mathsf{selftype}$. □

For an example of an S-system, see Section 6. In comparison with the AC-systems of [14], the novel aspect of S-systems is the use of selftype and the notation $U\{U'\}$.

We now show how to reduce type inference for a term $c$ to the solution of an S-system $\mathcal{C}(c)$.

Given a $\varsigma$-term $c$, assume that it has been $\alpha$-converted so that all free and bound variables are pairwise distinct. We will now generate an S-system where the variables of $c$ are a subset of the variables used in the constraint system. This will be convenient in the proof of Lemma 4.2 below.

We define $\mathcal{U}(c)$, $\mathcal{U}_0(c)$, and $\mathcal{C}(c)$ as follows.

- $\mathcal{U}(c)$ is a set of variables. It consists of: every variable $x$ that appears in $c$; a variable $[\![b]\!]$ for each occurrence of a subterm of $b$ of $c$; a variable $\langle a.l \rangle$ for each occurrence of a subterm $a.l$ of $c$; and a variable $\langle\!\langle b_i \rangle\!\rangle$ for each occurrence of a subterm $[l_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$ of $c$ and for each $i \in 1..n$.
- $\mathcal{U}_0(c)$ is the subset of $\mathcal{U}(c)$ consisting of the variables $x$ where $x$ appears in $c$ and variables $[\![b]\!]$ where $b$ is a subterm of $c$.
- $\mathcal{C}(c)$ is the S-system over $\mathcal{U}(c)$ and $\mathcal{U}_0(c)$ consisting of the following constraints:
  - for every occurrence in $c$ of a variable $x$, the constraint
  $$x \leq [\![x]\!] \tag{6}$$
  - for every occurrence in $c$ of a subterm of the form $a.l$, the two constraints
  $$[\![a]\!] \leq [l : \langle a.l \rangle] \tag{7}$$
  $$\langle a.l \rangle \{[\![a]\!]\} \leq [\![a.l]\!] \tag{8}$$
  - for every occurrence in $c$ of a subterm of the form $[l_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$, the constraint
  $$[l_i : \langle\!\langle b_i \rangle\!\rangle \ ^{i \in 1..n}] \leq [\![ [l_i = \varsigma(x_i)b_i \ ^{i \in 1..n}] ]\!] \tag{9}$$
  and for every $j \in 1..n$, the two constraints
  $$x_j = [l_i : \langle\!\langle b_i \rangle\!\rangle \ ^{i \in 1..n}] \tag{10}$$
  $$[\![b_j]\!] \leq \langle\!\langle b_j \rangle\!\rangle \{x_j\} \tag{11}$$

- for every occurrence in $c$ of a subterm of the form $a.l \Leftarrow \varsigma(x)b$, the three constraints

$$[\![a]\!] \leq [\![a.l \Leftarrow \varsigma(x)b]\!] \tag{12}$$

$$[\![a]\!] = x \tag{13}$$

$$[\![a]\!] \leq [l : [\![b]\!]] \ . \tag{14}$$

In the definition of $\mathcal{U}(c)$, the notations $[\![b]\!]$, $\langle a.l \rangle$, and $\langle\!\langle b_i \rangle\!\rangle$ are ambiguous because there may be more than one occurrence of the terms $b$, $a.l$, or $b_i$ in $c$. However, it will always be clear from context which occurrence is meant. In the definition of $\mathcal{C}(c)$, each equality $A = B$ denotes the two inequalities $A \leq B$ and $B \leq A$.

For a $\varsigma$-term of size $n$, the S-system $\mathcal{C}(c)$ is of size $O(n)$, and it is generated using polynomial time. We show below that the solutions of $\mathcal{C}(c)$ correspond to the possible type annotations of $c$ in a sense made precise by Lemma 4.2. For an example of an S-system generated from a $\varsigma$-term, see Section 6.

For any term $c$, type environment $E$, and function $L : \mathcal{U} \to T_\Sigma$, we say that $L$ *and* $E$ *agree on (the free variables of)* $c$ iff $L(x) = E(x)$ for all $x$ free in $c$.

**Lemma 4.2** *The judgment $E \vdash c : A$ is derivable if and only if there exists a solution $L$ of $\mathcal{C}(c)$ such that $L([\![c]\!]) = A$, and $L$ and $E$ agree on $c$.*

*Proof.* ($\Leftarrow$) We prove the following stronger statement:

*If $L$ is a solution to $\mathcal{C}(c)$, and $L'$ is the restriction of $L$ to the variables appearing in $c$, then $L' \vdash c_0 : L([\![c_0]\!])$ for every subterm $c_0$ of $c$.*

The proof is by induction on $c_0$.

- If $c_0 = x$, then $L' \vdash x : L'(x)$ by rule (1). And by (6), $L'(x) = L(x) \leq L([\![x]\!])$, so $L' \vdash x : L([\![x]\!])$ by rule (5).

- If $c_0 = a.l$, then by induction, $L' \vdash a : L([\![a]\!])$. By (7), $L([\![a]\!]) \leq \tilde{L}([l : \langle a.l \rangle])$, so by rule (2), $L' \vdash a.l : \tilde{L}(\langle a.l \rangle\{[\![a]\!]\})$. Finally by (8), $\tilde{L}(\langle a.l \rangle\{[\![a]\!]\}) \leq L([\![a.l]\!])$, so by rule (5), $L' \vdash a.l : L([\![a.l]\!])$.

- If $c_0 = [l_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$, then by induction, for $j \in 1..n$ we have $L' \vdash b_j : L([\![b_j]\!])$.

Note that $L' = L'[x_j \leftarrow L'(x_j)]$, and by (11), $L([\![b_j]\!]) \leq \tilde{L}(\langle\!\langle b_j \rangle\!\rangle\{x_j\})$; then by rule (5), $L'[x_j \leftarrow L'(x_j)] \vdash b_j : \tilde{L}(\langle\!\langle b_j \rangle\!\rangle\{x_j\})$.

So by rule (3), $L' \vdash c_0 : L(x_j)$ for any $j \in 1..n$.

By (10), $L(x_j) = \tilde{L}([l_i : \langle\!\langle b_i \rangle\!\rangle \ ^{i \in 1..n}])$, and by (9), $\tilde{L}([l_i : \langle\!\langle b_i \rangle\!\rangle \ ^{i \in 1..n}]) \leq L([\![c_0]\!])$, so by rule (5), $L' \vdash c_0 : L([\![c_0]\!])$.

- If $c_0 = (a.l \Leftarrow \varsigma(x)b)$, by induction we have $L' \vdash a : L([\![a]\!])$ and $L' \vdash b : L([\![b]\!])$.

  By (13), $L'(x) = L(x) = L([\![a]\!])$, so $L'[x \leftarrow L([\![a]\!])] \vdash b : L([\![b]\!])$.

  By (14), $L([\![a]\!]) \leq \tilde{L}([l : [\![b]\!]])$, so by rule (4) we have $L' \vdash c_0 : L([\![a]\!])$.

  Finally by (12), $L([\![a]\!]) \leq L([\![c_0]\!])$ so $L' \vdash c_0 : L([\![c_0]\!])$ by rule (5).

($\Rightarrow$) First we introduce some convenient notation. We say the *domain* of a function $L : \mathcal{U} \to T_\Sigma$ is the set $\{U \mid L(U) \neq U\}$. We write $\{U_1 := A_1, \ldots, U_n := A_n\}$ for the function with domain $\{U_1, \ldots, U_n\}$ mapping $U_i$ to $A_i$ for $i \in 1..n$. If $L_1 : \mathcal{U} \to T_\Sigma$ and $L_2 : \mathcal{U} \to T_\Sigma$, and $L_1(U) = L_2(U)$ for every $U$ in the domain of both $L_1$ and $L_2$, then $L_1 \cup L_2$ is the function from $\mathcal{U}$ to $T_\Sigma$ defined by

$$(L_1 \cup L_2)(U) = \begin{cases} L_1(U) & \text{if } U \text{ is in the domain of } L_1 \\ L_2(U) & \text{if } U \text{ is in the domain of } L_2 \\ U & \text{otherwise} \end{cases}$$

We now prove the following statement by induction on $c$, using Lemma 3.2:

*If $E \vdash c : A$ is derivable then there exists a solution $L$ of $\mathcal{C}(c)$ such that $L([\![c]\!]) = A$, $L$ has domain $\mathcal{U}(c)$, and $L$ and $E$ agree on $c$.*

- If $c = x$, then $E(x) \leq A$ and $\mathcal{C}(c) = \{x \leq [\![x]\!]\}$. Then let $L = \{x := E(x), [\![x]\!] := A\}$; clearly $L$ solves $\mathcal{C}(c)$, $L([\![c]\!]) = A$, $L$ has domain $\mathcal{U}(c)$, and $L$ and $E$ agree on $c$.

- If $c = a.l$, then for some $B$ and $C$, $E \vdash a : B$, $B \leq [l : C]$, $C\{B\} \leq A$, and $\mathcal{C}(c) = \mathcal{C}(a) \cup \{[\![a]\!] \leq [l : \langle a.l \rangle], \langle a.l \rangle\{[\![a]\!]\} \leq [\![a.l]\!]\}$.

  By induction there is a solution $L'$ of $\mathcal{C}(a)$ such that $L'([\![a]\!]) = B$, $L'$ has domain $\mathcal{U}(a)$, and $L'$ agrees with $E$ on $a$.

  Define $L = L' \cup \{[\![a]\!] := B, \langle a.l \rangle := C, [\![a.l]\!] := A\}$. Clearly $L$ solves $\mathcal{C}(c)$, $L([\![c]\!]) = A$, $L$ has domain $\mathcal{U}(c)$, and $L$ and $E$ agree on $c$.

- If $c = [l_i = \varsigma(x_i)b_i\ ^{i \in 1..n}]$, then for some $C$ and $B_i$ for $i \in 1..n$, $E[x_i \leftarrow C] \vdash b_i : B_i\{C\}$, $C = [l_i : B_i\ ^{i \in 1..n}] \leq A$, and

$$\begin{aligned} \mathcal{C}(c) = \ & \{[l_i : \langle\!\langle b_i \rangle\!\rangle\ ^{i \in 1..n}] \leq [\![c]\!]\} \\ & \cup \{x_j = [l_i : \langle\!\langle b_i \rangle\!\rangle\ ^{i \in 1..n}] \mid j \in 1..n\} \\ & \cup \{[\![b_j]\!] \leq \langle\!\langle b_j \rangle\!\rangle\{x_j\}\} \\ & \cup (\textstyle\bigcup_{i \in 1..n} \mathcal{C}(b_i)). \end{aligned}$$

  By induction, for $i \in 1..n$ there is a solution $L_i$ for $\mathcal{C}(b_i)$ such that $L_i([\![b_i]\!]) = B_i\{C\}$, $L_i$ has domain $\mathcal{U}(b_i)$, and $L_i$ agrees with $E[x_i \leftarrow C]$ on $b_i$.

  Define $L = \{[\![c]\!] := A\} \cup (\bigcup_{i \in 1..n}(L_i \cup \{\langle\!\langle b_i \rangle\!\rangle := B_i, x_i := C\}))$. This is well-defined because a variable $x$ is in the domain of both $L_i$ and $L_j$ iff $x$ is free in both $b_i$ and $b_j$, in which case $L_i(x) = E(x) = L_j(x)$.

  Then $L$ solves $\mathcal{C}(c)$, $L([\![c]\!]) = A$, $L$ has domain $\mathcal{U}(c)$, and $L$ and $E$ agree on $c$.

- If $c = (a.l \Leftarrow \varsigma(x)b)$, then for some $B$ and $C$, $E \vdash a : B$, $E[x \leftarrow B] \vdash b : C$, $B \leq [l : C]$ $B \leq A$, and

$$\mathcal{C}(c) = \mathcal{C}(a) \cup \mathcal{C}(b) \cup \{[\![a]\!] \leq [\![c]\!], [\![a]\!] = x, [\![a]\!] \leq [l : [\![b]\!]]\}.$$

  By induction there is a solution $L_1$ of $\mathcal{C}(a)$ such that $L_1([\![a]\!]) = B$, $L_1$ has domain $\mathcal{U}(a)$, and $L_1$ agrees with $E$ on $a$; and a solution $L_2$ of $\mathcal{C}(b)$ such that $L_2([\![b]\!]) = C$, $L_2$ has domain $\mathcal{U}(b)$, and $L_2$ agrees with $E[x \leftarrow B]$ on $b$.

  Let $L = \{[\![c]\!] := A\} \cup L_1 \cup L_2$; this is well-defined because a variable $x$ is in the domain of both $L_1$ and $L_2$ iff $x$ is free in both $a$ and $b$, in which case $L_1(x) = E(x) = L_2(x)$.

  Then $L$ solves $\mathcal{C}(c)$, $L([\![c]\!]) = A$, $L$ has domain $\mathcal{U}(c)$, and $L$ and $E$ agree on $c$. □

To solve an arbitrary S-system, we proceed in two steps. First we define a family of transformations. Each transformation eliminates the components of the form $U\{U'\}$ in an S-system. Given these transformations, it is straightforward that an S-system can be solved in *NP* time.

The family of mappings $F_S$ from S-systems to S-systems is defined as follows. Let $\mathcal{C}$ be an S-system over $\mathcal{U}$ and $\mathcal{U}_0$, and let $S \subseteq (\mathcal{U} \setminus \mathcal{U}_0)$. Intuitively, $S$ is a guess on the set of variables that some solution of $\mathcal{C}$ would map to $\mathsf{selftype}$. Define $F_S(\mathcal{C})$ to be the S-system over $\mathcal{U}$ and $(\mathcal{U} \setminus S)$ where

- For each $U \in S$, the constraint $U \leq \mathsf{selftype}$ is in $F_S(\mathcal{C})$.
- If a constraint of the form $W \leq W'$ is in $\mathcal{C}$, then $f_S(W) \leq f_S(W')$ is in $F_S(\mathcal{C})$, where:

$$f_S(W) = \begin{cases} U' & \text{if } W = U\{U'\} \text{ and } U \in S, \\ U & \text{if } W = U\{U'\} \text{ and } U \notin S, \\ W & \text{otherwise.} \end{cases}$$

We can now characterize solvability of S-systems in terms of the mappings $F_S$.

**Lemma 4.3** *Suppose $\mathcal{C}$ is an S-system over $\mathcal{U}$ and $\mathcal{U}_0$. Then $\mathcal{C}$ is solvable if and only if there exists $S \subseteq (\mathcal{U} \setminus \mathcal{U}_0)$ such that $F_S(\mathcal{C})$ is solvable over $\mathcal{U}$ and $\mathcal{U} \setminus S$.*

*Proof.* Suppose first that $\mathcal{C}$ has solution $L$. Define $S = \{U \in \mathcal{U} \mid L(U) = \mathsf{selftype}\}$. It is straightforward to show that $F_S(\mathcal{C})$ has solution $L$.

Suppose then that we have $S \subseteq \mathcal{U}$ such that $F_S(\mathcal{C})$ has solution $L$. It is straightforward to show that $\mathcal{C}$ has solution $L$. □

**Corollary 4.4** *Solvability of S-systems is in NP time.*

*Proof.* Suppose $\mathcal{C}$ is an S-system over $\mathcal{U}$ and $\mathcal{U}_0$. Guess $S \subseteq (\mathcal{U} \setminus \mathcal{U}_0)$. Transform $\mathcal{C}$ into $F_S(\mathcal{C})$, using polynomial time. It follows from Lemma 4.3 that it is sufficient to decide if $F_S(\mathcal{C})$ is solvable. This is in turn equivalent to deciding if $F_S(\mathcal{C})$ has a solution $L$ where no $L(U)$ contains free variables. This can be done in $O(n^3)$ time using a slightly modified version of the algorithm in [14]. (The algorithm in [14] handles so-called AC-systems, that is, S-systems without $U\{U'\}$ and without $\mathsf{selftype}$. In the journal version of [14], it is indicated how to extend the constraint solving algorithm for AC-systems to handle functions and records. It is equally easy to extend the algorithm to handle a constant such as $\mathsf{selftype}$ and the condition that variables in $\mathcal{U} \setminus S$ cannot be mapped to $\mathsf{selftype}$.) □

Lemma 4.2 and Corollary 4.4 imply the main result of this section.

**Theorem 4.5** *The type inference problem for the type system with selftype, recursive types, and subtyping can be decided in nondeterministic polynomial time.*

# 5 Type Inference is *NP*-hard

In this section we prove that the type inference problem is *NP*-hard. We do this by first proving that a certain constraint problem is *NP*-hard (Lemma 5.6), and then showing that the constraint problem reduces to the type inference problem (Lemma 5.7).

**Definition 5.1** Given a denumerable set $\mathcal{U}$ of variables, a *CS-system (core selftype system) over* $\mathcal{U}$ is a finite set of constraints of the forms:

(i) $V \leq [l : [\,]]$

(ii) $V \leq [l : [m : [\,]]]$

(iii) $V \leq U\{[l : U]\}$

(iv) $V \leq [l : U] \;\wedge\; U\{V\} \leq V'$

where $U, V, V' \in \mathcal{U}$. If $[l : U]$ appears in a CS-system, then there is exactly one constraint of form (iii) which involves $[l : U]$. If $[l : U]$ and $[l' : U']$ appear in a CS-system, then either $l = l'$ and $U = U'$, or $l \neq l'$ and $U \neq U'$.

A *solution* for a CS-system is a function $L : \mathcal{U} \to T_\Sigma$ such that all constraints are satisfied when elements of $\mathcal{U}$ are mapped to types by $L$.

For a given CS-system $\mathcal{C}$, denote by $\mathcal{V}(\mathcal{C})$ the set of variables of $\mathcal{C}$ which do not occur as part of any $[l : U]$. □

Notice that if the CS-system $\mathcal{C}$ is solvable, then no variable in $\mathcal{V}(\mathcal{C})$ is mapped to selftype; in each case (i–iv) the only possible member of $\mathcal{V}(\mathcal{C})$ is $V$ or $V'$, and each $V$ or $V'$ is related directly or indirectly to a record type.

We now define a CS-system which will be used to encode Boolean variables. The construction is based on the observation that the types $[m : [\,]]$ and $[m : [m : [\,]]]$ do not have a common lower bound, since $[\,] \neq [m : [\,]]$.

**Definition 5.2** Let $x$ be a Boolean variable. Define

$$\mathcal{U}_x = \{U_x, U_{\overline{x}}, V_x, V'_x, V''_x, V'''_x, V^A_x, V^B_x\} \ .$$

The CS-system $\mathcal{C}_x$ over $\mathcal{U}_x$ consists of the following seven constraints:

(i) $V_x \leq [l_x : U_x] \ \wedge \ U_x\{V_x\} \leq V^A_x$

(ii) $V_x \leq [l_{\overline{x}} : U_{\overline{x}}] \ \wedge \ U_{\overline{x}}\{V_x\} \leq V^B_x$

(iii) $V'_x \leq [l_x : U_x] \ \wedge \ U_x\{V'_x\} \leq V''_x$

(iv) $V''_x \leq U_{\overline{x}}\{[l_{\overline{x}} : U_{\overline{x}}]\}$

(v) $V'''_x \leq U_x\{[l_x : U_x]\}$

(vi) $V^A_x \leq [m_x : [\ ]]$

(vii) $V^B_x \leq [m_x : [m_x : [\ ]]]$

□

**Lemma 5.3** *If $\mathcal{C}_x$ has solution $L$, then $L(V^A_x)$ and $L(V^B_x)$ have no common lower bound.*

*Proof.* Suppose $\mathcal{C}_x$ has solution $L$, and suppose that $P$ is a common lower bound for $L(V^A_x)$ and $L(V^B_x)$. Then $P \leq L(V^A_x) \leq [m_x : [\ ]]$ and $P \leq L(V^B_x) \leq [m_x : [m_x : [\ ]]]$. Thus we have both $P \downarrow m_x = [\ ]$ and $P \downarrow m_x = [m_x : [\ ]]$, a contradiction. □

The next two lemmas show that $U_x$ and $U_{\overline{x}}$ can be used as boolean variables.

**Lemma 5.4** *There is no solution $L$ of $\mathcal{C}_x$ such that $L(U_x) = \mathsf{selftype} = L(U_{\overline{x}})$ or $L(U_x) \neq \mathsf{selftype} \neq L(U_{\overline{x}})$.*

*Proof.* If $L(U_x) = L(U_{\overline{x}}) = \mathsf{selftype}$, then by 5.2(i) and (ii), $L(V_x) \leq L(V^A_x)$ and $L(V_x) \leq L(V^B_x)$, contradicting Lemma 5.3. If $L(U_x) \neq \mathsf{selftype}$ and $L(U_{\overline{x}}) \neq \mathsf{selftype}$, then by 5.2(i), $L(U_x) \leq L(V^A_x)$, and by 5.2(iii), (iv), and (ii), $L(U_x) \leq L(V''_x) \leq L(U_{\overline{x}}) \leq L(V^B_x)$, contradicting Lemma 5.3. □

**Lemma 5.5** *$\mathcal{C}_x$ has both a solution $L$ where $L(U_x) = \mathsf{selftype} \neq L(U_{\overline{x}})$, and a solution $L'$ where $L'(U_x) \neq \mathsf{selftype} = L'(U_{\overline{x}})$.*

*Proof.* First choose two types $A$ and $B$ such that $A \leq [m_x : [\ ]]$ and $B \leq [m_x : [m_x : [\ ]]]$, and such that $l_x$ and $l_{\overline{x}}$ are not fields of either of $A$ and $B$. For types $A_1$ and $A_2$, define $A_1 \oplus A_2$ to be union of $A_1$ and $A_2$ thought of as functions, assuming that the domains are disjoint. We now list a solution $L$ of $\mathcal{C}_x$ with $L(U_x) = \mathsf{selftype}$, and a solution $L'$ of $\mathcal{C}_x$ with $L'(U_x) \neq \mathsf{selftype}$.

| | $L$ | $L'$ |
|---|---|---|
| $U_x$ | $\mathsf{selftype}$ | $[l_{\overline{x}} : \mathsf{selftype}] \oplus A$ |
| $U_{\overline{x}}$ | $B$ | $\mathsf{selftype}$ |
| $V_x$ | $[l_x : \mathsf{selftype}, l_{\overline{x}} : B] \oplus A$ | $[l_x : [l_{\overline{x}} : \mathsf{selftype}] \oplus A, l_{\overline{x}} : \mathsf{selftype}] \oplus B$ |
| $V'_x$ | $[l_x : \mathsf{selftype}] \oplus B$ | $[l_x : [l_{\overline{x}} : \mathsf{selftype}] \oplus A]$ |
| $V''_x$ | $B$ | $[l_{\overline{x}} : \mathsf{selftype}]$ |
| $V'''_x$ | $[l_x : \mathsf{selftype}]$ | $[l_{\overline{x}} : \mathsf{selftype}] \oplus A$ |
| $V^A_x$ | $A$ | $A$ |
| $V^B_x$ | $B$ | $B$ |

□

**Lemma 5.6** *Solvability of CS-systems is NP-hard.*

*Proof.* We reduce SAT to solvability of CS-systems. Given a CNF formula $\varphi \equiv \prod_{i=1}^{n} \sum_{j=1}^{n_i} e^i_j$ where each $e^i_j$ is either a variable or its negation, and where for each $i$, $e^i_1, \ldots, e^i_{n_i}$ are different. Define

$$\mathcal{U}_\varphi = (\bigcup_x \mathcal{U}_x) \cup (\bigcup_{i=1}^{n} \bigcup_{j=1}^{n_i} \{V^i_j\}) \ .$$

The CS-system $\mathcal{C}(\varphi)$ over $\mathcal{U}_\varphi$ consists of the constraints:

- For each variable $x$ in $\varphi$: $\mathcal{C}_x$
- For all $i \in 1..n$:
  - $V^i_1 \leq [l_i : [\ ]]$
  - $V^i_{n_i} \leq [l_i : [l_i : [\ ]]]$
  - For $j \in 1..n_i$: $V^i_j \leq [l_{e^i_j} : U_{e^i_j}] \ \wedge \ U_{e^i_j}\{V^i_j\} \leq V^i_{j+1}$.

We will prove that $\varphi$ is satisfiable if and only if $\mathcal{C}(\varphi)$ is solvable.

Suppose first that $\varphi$ is satisfiable. Let $s$ be a satisfying assignment. We can assume, without loss of generality, that there is a total ordering $\sqsubseteq$ of the literals such that for each $i$, if $j \leq j'$, then $e^i_j \sqsupseteq e^i_{j'}$ and $s(e^i_j) \Leftarrow s(e^i_{j'})$. (This can be obtained by, for each $i$, reordering the literals.) We now define a solution $L$ of $C(\varphi)$. For a literal $\rho$ where $s(\rho) = \mathbf{false}$, define $L(U_\rho) = \mathsf{selftype}$, and for each occurrence $e^i_j$ of $\rho$, define $L(V^i_j) = [l_{e^i_j} : \mathsf{selftype}, \ldots, l_{e^i_{n_i}} : \mathsf{selftype}, l_i : [l_i : [\,]]]$. We will process the literals $\rho$ for which $s(\rho) = \mathbf{true}$ in the order $\sqsubseteq$, beginning with the smallest. Let $\rho$ be a literal for which $s(\rho) = \mathbf{true}$. Define

$$
\begin{aligned}
L(U_\rho) \quad = \quad & \bigoplus_{occ.\ e^i_j\ of\ \rho} (L(V^i_{j+1}) \quad \oplus \quad [l_i : [\,] \ \ (\text{only if } j = 1)]) \\
& \oplus \quad [m_x : [\,], \quad l_{\overline{x}} : \mathsf{selftype} \ \ (\text{only if } \rho \equiv x) \\
& \qquad\ m_x : [m_x : [\,]] \ \ (\text{only if } \rho \equiv \overline{x}) \ \ ]
\end{aligned}
$$

For each occurrence $e^i_j$ of $\rho$, define $L(V^i_j) = [l_{e^i_j} : L(U_{e^i_j})]$. Finally, define, for each $i$, $L(V^i_{n_i}) = [l_i : [l_i : [\,]]]$. It is straightforward to show that $C(\varphi)$ has solution $L$.

Conversely, suppose $C(\varphi)$ is solvable. Let $L$ be a solution. Suppose we have $i$ such that all $U_{e^i_j}$ are $\mathsf{selftype}$ under $L$. Then $L$ is a solution of

$$
\begin{aligned}
V^i_1 &\leq [l_i : [\,]] \\
V^i_1 &\leq \ldots \leq V^i_{n_i} \leq [l_i : [l_i : [\,]]] \ ,
\end{aligned}
$$

a contradiction. Thus, for each $i$, at least on of $U_{e^i_j}$ is not $\mathsf{selftype}$ under $L$. Define

$$
s(x) = \begin{cases} \mathbf{false} & \text{if } L(U_x) = \mathsf{selftype} \\ \mathbf{true} & \text{otherwise .} \end{cases}
$$

It is straightforward to show that $s$ satisfies $\varphi$. □

**Lemma 5.7** *Solvability of CS-systems is reducible to the type inference problem.*

*Proof.* Let $\mathcal{C}$ be a CS-system. Let $a^{\mathcal{C}}$ denote the following $\varsigma$-term:

$$
\begin{array}{lll}
[ & k_V & = \varsigma(x)((x.k_V \Leftarrow \varsigma(y)(x.k_V)).k_V) \\
& & \quad \text{for each variable } V \in \mathcal{V}(\mathcal{C}) \\
& k_R & = \varsigma(x)[l = \varsigma(y)[\,]] \\
& & \quad \text{for each constraint } V \leq R \text{ in } \mathcal{C}, \\
& & \quad \text{where } R \equiv [l : [\,]]. \\
& k_R & = \varsigma(x)[l = \varsigma(y)[m = \varsigma(z)[\,]]] \\
& & \quad \text{for each constraint } V \leq R \text{ in } \mathcal{C}, \\
& & \quad \text{where } R \equiv [l : [m : [\,]]]. \\
& k_P & = \varsigma(x)((x.k_R \Leftarrow \varsigma(y)(x.k_V)).k_R.l) \\
& & \quad \text{for each constraint } P \equiv (V \leq R) \text{ in } \mathcal{C}, \\
& & \quad \text{where } R \equiv [l : [\,]]. \\
& k_P & = \varsigma(x)((x.k_R \Leftarrow \varsigma(y)(x.k_V)).k_R.l.m) \\
& & \quad \text{for each constraint } P \equiv (V \leq R) \text{ in } \mathcal{C}, \\
& & \quad \text{where } R \equiv [l : [m : [\,]]]. \\
& k_{[l:U]} & = \varsigma(x)[l = \varsigma(y)(x.k_V)] \\
& & \quad \text{for each constraint } V \leq U\{[l : U]\} \text{ in } \mathcal{C}. \\
& k'_{[l:U]} & = \varsigma(x)(x.k_{[l:U]}.l) \\
& & \quad \text{for each constraint } V \leq U\{[l : U]\} \text{ in } \mathcal{C}. \\
& k_P & = \varsigma(x)((x.k_{[l:U]} \Leftarrow \varsigma(y)(x.k_V)).k_V) \\
& & \quad \text{for each constraint } P \equiv (V \leq [l : U] \ \wedge \ U\{V\} \leq V') \text{ in } \mathcal{C}. \\
& k'_P & = \varsigma(x)((x.k_{V'} \Leftarrow \varsigma(y)(x.k_V.l)).k_V) \\
& & \quad \text{for each constraint } P \equiv (V \leq [l : U] \ \wedge \ U\{V\} \leq V') \text{ in } \mathcal{C}. \\
]
\end{array}
$$

We first prove that if $\mathcal{C}$ is solvable, then $a^{\mathcal{C}}$ is typable. Suppose $\mathcal{C}$ has solution $L$. Let $A$ denote the following type:

$$
\begin{array}{lll}
[ & k_V & : L(V) \\
& & \quad \text{for each variable } V \in \mathcal{V}(\mathcal{C}) \\
& k_R & : [l : [\,]] \\
& & \quad \text{for each constraint } V \leq R \text{ in } \mathcal{C}, \\
& & \quad \text{where } R \equiv [l : [\,]]. \\
& k_R & : [l : [m : [\,]]] \\
& & \quad \text{for each constraint } V \leq R \text{ in } \mathcal{C}, \\
& & \quad \text{where } R \equiv [l : [m : [\,]]]. \\
& k_P & : [\,] \\
& & \quad \text{for each constraint } P \equiv (V \leq R) \text{ in } \mathcal{C}, \\
& & \quad \text{where } R \equiv [l : [\,]]. \\
& k_P & : [\,] \\
& & \quad \text{for each constraint } P \equiv (V \leq R) \text{ in } \mathcal{C}, \\
& & \quad \text{where } R \equiv [l : [m : [\,]]]. \\
& k_{[l:U]} & : [l : L(U)] \\
& & \quad \text{for each constraint } V \leq U\{[l : U]\} \text{ in } \mathcal{C}. \\
& k'_{[l:U]} & : L(U)\{[l : L(U)]\} \\
& & \quad \text{for each constraint } V \leq U\{[l : U]\} \text{ in } \mathcal{C}. \\
& k_P & : L(V) \\
& & \quad \text{for each constraint } P \equiv (V \leq [l : U] \ \wedge \ U\{V\} \leq V') \text{ in } \mathcal{C}. \\
& k'_P & : L(V) \\
& & \quad \text{for each constraint } P \equiv (V \leq [l : U] \ \wedge \ U\{V\} \leq V') \text{ in } \mathcal{C}. \\
] & &
\end{array}
$$

It is straightforward to show that $\emptyset \vdash a^{\mathcal{C}} : A$ is derivable.

Now we prove that if $a^{\mathcal{C}}$ is typable, then $C$ is solvable. Suppose $a^{\mathcal{C}}$ is typable. From Lemma 4.2 we get a solution $M$ of $\mathcal{C}(a^{\mathcal{C}})$. Notice that each method in $a^{\mathcal{C}}$ binds a variable $x$. Each of these variables corresponds to a distinct type variable in $\mathcal{C}(a^{\mathcal{C}})$. Since $M$ is a solution of $\mathcal{C}(a^{\mathcal{C}})$, and $\mathcal{C}(a^{\mathcal{C}})$ contains constraints of the form $x = [\ldots]$ for each method in $a^{\mathcal{C}}$ (from rule (10)), all those type variables are mapped by $M$ to the same type. Thus, we can think of all the bound variables in $a^{\mathcal{C}}$ as being related to the same type variable, which we will write as $x$.

Let $L : \mathcal{U} \to T_\Sigma$ be defined by

$$
\begin{array}{lcll}
L(V) & = & M(x)\downarrow k_V & \text{for } V \in \mathcal{V}(\mathcal{C}) \\
L(U) & = & M(x)\downarrow k_{[l:U]}\downarrow l & \text{for } U \in \mathcal{U} \setminus \mathcal{V}(\mathcal{C}) \ .
\end{array}
$$

The definition is justified by the two properties listed below. We will proceed by first showing the two properties and then showing that $\mathcal{C}$ has solution $L$.

- **Property 1** If $V \in \mathcal{V}(\mathcal{C})$, then $M(x)\downarrow k_V$ is defined and $M(x)\downarrow k_V \neq \mathsf{selftype}$.
- **Property 2** For each constraint $V \leq U\{[l : U]\}$ in $\mathcal{C}$,
  (i) $M(x)\downarrow k_{[l:U]} = [l : A]$ for some $A \in T_\Sigma$, and
  (ii) $M(x)\downarrow k_V \leq A\{[l : A]\}$ where $A = M(x)\downarrow k_{[l:U]}\downarrow l$.

To see Property 1, notice that in the body of the method $k_V$ we have the expression $x.k_V \Leftarrow \varsigma(y)(x.k_V)$. Since $M$ is a solution of $\mathcal{C}(a^{\mathcal{C}})$, we have from the rules (6) and (14) that $M$ satisfies

$$x \leq [\![x]\!] \leq [k_V : [\![x.k_V]\!]] \ .$$

Thus, $M(x)\downarrow k_V = M([\![x.k_V]\!])$ is defined, and since $[\![x.k_V]\!] \in \mathcal{U}_0(a^{\mathcal{C}})$, we have $M([\![x.k_V]\!]) \neq \mathsf{selftype}$, so $M(x)\downarrow k_V \neq \mathsf{selftype}$.

To see Property 2, notice that in the body of the method $k'_{[l:U]}$ we have the expression $x.k_{[l:U]}.l$. Since $M$ is a solution of $\mathcal{C}(a^{\mathcal{C}})$, we have from the rules (6), (7), (8), and (7) that $M$ satisfies

$$\begin{aligned} x \leq [\![x]\!] &\leq [k_{[l:U]} : \langle x.k_{[l:U]}\rangle] && (15)\\ \langle x.k_{[l:U]}\rangle\{[\![x]\!]\} &\leq [\![x.k_{[l:U]}]\!] && (16)\\ [\![x.k_{[l:U]}]\!] &\leq [l : \langle x.k_{[l:U]}.l\rangle] \ . && (17) \end{aligned}$$

Moreover, in the body of the method $k_{[l:U]}$ we have the expression $[l = \varsigma(y)(x.k_V)]$. Since $M$ is a solution of $\mathcal{C}(a^{\mathcal{C}})$, we have from the rules (9), (10), (11), (6), (7), and (8), that $M$ satisfies

$$\begin{aligned} [l : \langle\!\langle x.k_V\rangle\!\rangle] &\leq [\![[l = \varsigma(y)(x.k_V)]]\!] && (18)\\ y &= [l : \langle\!\langle x.k_V\rangle\!\rangle] && (19)\\ [\![x.k_V]\!] &\leq \langle\!\langle x.k_V\rangle\!\rangle\{y\} && (20)\\ x \leq [\![x]\!] &\leq [k_V : \langle x.k_V\rangle] && (21)\\ \langle x.k_V\rangle\{[\![x]\!]\} &\leq [\![x.k_V]\!] \ . && (22) \end{aligned}$$

Finally, from $a^{\mathcal{C}}$ itself and the rules (10) and (11), we get that $M$ satisfies

$$\begin{aligned} x &= [\ldots k_{[l:U]} : \langle\!\langle [l = \varsigma(y)(x.k_V)] \rangle\!\rangle \ldots] && (23)\\ [\![ [l = \varsigma(y)(x.k_V)] ]\!] &\leq \langle\!\langle [l = \varsigma(y)(x.k_V)] \rangle\!\rangle\{x\}\ . && (24) \end{aligned}$$

From (15) we get $M(x)\!\downarrow\! k_{[l:U]} = M(\langle x.k_{[l:U]}\rangle)$, and from (23) we get $M(x)\!\downarrow\! k_{[l:U]} = M(\langle\!\langle [l = \varsigma(y)(x.k_V)] \rangle\!\rangle)$. Hence,

$$M(\langle\!\langle [l = \varsigma(y)(x.k_V)] \rangle\!\rangle) = M(\langle x.k_{[l:U]}\rangle) \qquad (25)$$

We get

$$\begin{aligned} [l : M(\langle\!\langle x.k_V \rangle\!\rangle)] &\leq M([\![ [l = \varsigma(y)(x.k_V)] ]\!]) && \text{from (18)}\\ &\leq (M(\langle\!\langle [l = \varsigma(y)(x.k_V)] \rangle\!\rangle))\{M(x)\} && \text{from (24)}\\ &= (M(\langle x.k_{[l:U]}\rangle))\{M(x)\} && \text{from (25)}\\ &\leq (M(\langle x.k_{[l:U]}\rangle))\{M([\![x]\!])\} && \text{from (6)}\\ &\leq M([\![x.k_{[l:U]}]\!]) && \text{from (16)}\\ &\leq [l : M(\langle x.k_{[l:U]}.l\rangle)] && \text{from (17).} \end{aligned}$$

From this calculation we get that $M(\langle\!\langle x.k_V \rangle\!\rangle) = M(\langle x.k_{[l:U]}.l\rangle)$, so

$$(M(\langle x.k_{[l:U]}\rangle))\{M(x)\} = [l : M(\langle\!\langle x.k_V \rangle\!\rangle)]\ .$$

Hence, $M(\langle x.k_{[l:U]}\rangle) \neq \mathsf{selftype}$, since $M(\langle x.k_{[l:U]}\rangle) = \mathsf{selftype}$ would imply $M(x) = [l : M(\langle\!\langle x.k_V \rangle\!\rangle)]$ which clearly is false. Thus,

$$M(\langle x.k_{[l:U]}\rangle) = [l : M(\langle\!\langle x.k_V \rangle\!\rangle)]$$

so $M(x)\!\downarrow\! k_{[l:U]}\!\downarrow\! l = M(\langle x.k_{[l:U]}\rangle)\!\downarrow\! l = [l : M(\langle\!\langle x.k_V \rangle\!\rangle)]\!\downarrow\! l = M(\langle\!\langle x.k_V \rangle\!\rangle)$ is defined, and by defining $A = M(\langle\!\langle x.k_V \rangle\!\rangle)$ we get $M(x)\!\downarrow\! k_{[l:U]} = [l : A]$. From Property 1 we get that $M(x)\!\downarrow\! k_V \neq \mathsf{selftype}$, so

$$\begin{aligned} M(x)\!\downarrow\! k_V &= M(\langle x.k_V\rangle) && \text{from (21)}\\ &\leq M([\![x.k_V]\!]) && \text{from (22)}\\ &\leq (M(\langle\!\langle x.k_V \rangle\!\rangle))\{M(y)\} && \text{from (20)}\\ &= (M(\langle\!\langle x.k_V \rangle\!\rangle))\{[l : M(\langle\!\langle x.k_V \rangle\!\rangle)]\} && \text{from (19)}\\ &= A\{[l : A]\} && \text{by definition.} \end{aligned}$$

This establishes Property 2. We now show that $\mathcal{C}$ has solution $L$.

Consider first a constraint $V \leq U\{[l : U]\}$ in $\mathcal{C}$. From Property 2 we get that $L(V) = M(x) \downarrow k_V \leq (L(U))\{[l : L(U)]\}$. Thus, $L$ satisfies the constraint.

Consider then a constraint $P \equiv (V \leq [l : U] \wedge U\{V\} \leq V')$ in $\mathcal{C}$. In the body of the method $k_P$ we have the expression $x'.k_{[l:U]} \Leftarrow \varsigma(y)(x.k_V)$ where we, for clarity, have written the first occurrence of $x$ as $x'$. Since $M$ is a solution of $\mathcal{C}(a^{\mathcal{C}})$, we have from the rules (6), (14), (6), (7), and (8), that $M$ satisfies

$$\begin{aligned} x \leq [\![x']\!] &\leq [k_{[l:U]} : [\![x.k_V]\!]] & (26) \\ x \leq [\![x]\!] &\leq [k_V : \langle x.k_V \rangle] & (27) \\ \langle x.k_V \rangle \{[\![x]\!]\} &\leq [\![x.k_V]\!] \ . & (28) \end{aligned}$$

From (27) we get $M(x) \downarrow k_V = M(\langle x.k_V \rangle)$. By Property 1, $M(x) \downarrow k_V \neq$ $\mathsf{selftype}$, so from (28) and (26) we get $M(\langle x.k_V \rangle) \leq M([\![x.k_V]\!]) = M(x) \downarrow k_{[l:U]}$. We conclude $L(V) = M(x) \downarrow k_V \leq M(x) \downarrow k_{[l:U]} = [l : M(x) \downarrow k_{[l:U]} \downarrow l] = [l : L(U)]$, using Property 2. It follows that

$$M(x) \downarrow k_{[l:U]} \downarrow l = M(x) \downarrow k_V \downarrow l \qquad (29)$$

In the body of the method $k'_P$ we have the expression $x'.k_{V'} \Leftarrow \varsigma(y)(x.k_V.l)$ where we, for clarity, have written the first occurrence of $x$ as $x'$. Since $M$ is a solution of $\mathcal{C}(a^{\mathcal{C}})$, we have from the rules (6), (14), (6), (7), (8), (7), and (8), that $M$ satisfies

$$\begin{aligned} x \leq [\![x']\!] &\leq [k_{V'} : [\![x.k_V.l]\!]] & (30) \\ x \leq [\![x]\!] &\leq [k_V : \langle x.k_V \rangle] & (31) \\ \langle x.k_V \rangle \{[\![x]\!]\} &\leq [\![x.k_V]\!] & (32) \\ [\![x.k_V]\!] &\leq [l : \langle x.k_V.l \rangle] & (33) \\ \langle x.k_V.l \rangle \{[\![x.k_V]\!]\} &\leq [\![x.k_V.l]\!] \ . & (34) \end{aligned}$$

From (31) we get $M(x) \downarrow k_V = M(\langle x.k_V \rangle)$. By Property 1, $M(x) \downarrow k_V \neq$ $\mathsf{selftype}$, so from (32) and (33) we get $M(\langle x.k_V \rangle) \leq M([\![x.k_V]\!]) \leq [l : M(\langle x.k_V.l \rangle)]$. Thus,

$$\begin{aligned} M(x) \downarrow k_V &\leq M([\![x.k_V]\!]) & (35) \\ M(x) \downarrow k_V \downarrow l &= M(\langle x.k_V.l \rangle) \ . & (36) \end{aligned}$$

We conclude

$$
\begin{array}{rcll}
(L(U))\{L(V)\} & = & (M(x)\downarrow k_{[l:U]}\downarrow l)\{M(x)\downarrow k_V\} & \text{by definition} \\
 & = & (M(x)\downarrow k_V\downarrow l)\{M(x)\downarrow k_V\} & \text{from (29)} \\
 & = & (M(\langle x.k_V.l\rangle))\{M(x)\downarrow k_V\} & \text{from (36)} \\
 & \leq & (M(\langle x.k_V.l\rangle))\{M([\![x.k_V]\!])\} & \text{from (35)} \\
 & \leq & M([\![x.k_V.l]\!]) & \text{from (34)} \\
 & = & M(x)\downarrow k_{V'} & \text{from (30)} \\
 & = & L(V') & \text{by definition.}
\end{array}
$$

Thus, $L$ satisfies the constraint $P$.

The remaining two cases of constraints of the forms $V \leq [l : [\ ]]$ and $V \leq [l : [m : [\ ]]]$ are handled similarly. We omit the details. □

By combining Theorem 4.5, Lemma 5.6, and Lemma 5.7 we obtain our main theorem.

**Theorem 5.8** *The type inference problem for the type system with selftype, recursive types, and subtyping is NP-complete.*

It also follows that solvability of S-systems and solvability of CS-systems are NP-complete.

# 6 Example

We now illustrate some of the constructions in the paper. Consider the following program skeleton.

```
object Point
  ...
  method move
    ...
    return self
  end
end

object ColorPoint
  ...
  method move
    ...
    return self
  end
  method setcolor
    ...
    return self
  end
end
```

```
object Circle
  ...
  method center
    return Point
  end
  ...
end

object ColorCircle overrides Circle
  ...
  method center
    return ColorPoint.move.setcolor
  end
end

-- Main program:

ColorCircle.center.move
```

The only significant aspect of the `Point` and `ColorPoint` objects is that their methods return self. The object `Circle` returns the `Point` object when asked for its center. The object `ColorCircle` is defined by inheritance from `Circle`: it overrides the center method. When asked for its center, the `ColorCircle` first slightly changes the coordinates and color of the `ColorPoint`, and then it returns the resulting object. The main program executes without errors.

The key aspects of the example can be directly represented in the object calculus of Abadi and Cardelli as follows.

$$\begin{array}{rcl} Point & \equiv & [move = \varsigma(x)x] \\ ColorPoint & \equiv & [move = \varsigma(y)y \ \ , \ \ setcolor = \varsigma(z)z] \\ Circle & \equiv & [center = \varsigma(d)Point] \\ ColorCircle & \equiv & Circle.center \Leftarrow \varsigma(e)(ColorPoint.move.setcolor) \\ Main & \equiv & ColorCircle.center.move\ . \end{array}$$

We may then ask: can the program be typed in Abadi and Cardelli's first-order type system with recursive types and subtyping? The answer is, perhaps surprisingly: no! This answer can be obtained by running the type inference algorithm of Palsberg [14]. The key reason for the untypability is that the body of the $ColorCircle$'s center method forces $ColorPoint$ to have a type which is *not* a subtype of the type of $Point$, intuitively as follows.

$$\begin{array}{rcl} Point & : & \mu(X)[move : X] \\ ColorPoint & : & \mu(X)[move, setcolor : X] \\ \mu(X)[move, setcolor : X] & \not\leq & \mu(X)[move : X] \\ \text{Moreover} & : & ColorCircle.center.move \text{ is } \textit{not} \text{ typable}\ . \end{array}$$

In our type system with selftype, however, $ColorPoint$ can be given a type that is a subtype of the type of $Point$, and the program is typable:

$$\begin{array}{rcl} Point & : & [move : \mathsf{selftype}] \\ ColorPoint & : & [move, setcolor : \mathsf{selftype}] \\ [move, setcolor : \mathsf{selftype}] & \leq & [move : \mathsf{selftype}] \\ \text{Moreover} & : & ColorCircle.center.move\ \textit{is} \text{ typable}\ . \end{array}$$

More specifically, define

$$\begin{array}{rcl} P & \equiv & [move : \mathsf{selftype}] \\ Q & \equiv & [move, setcolor : \mathsf{selftype}] \\ E & \equiv & \emptyset[d \leftarrow [center : P]] \\ F & \equiv & \emptyset[e \leftarrow P]\ . \end{array}$$

We can then derive $\emptyset \vdash ColorCircle.center.move : P$ as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{E[x \leftarrow P] \vdash x : P}{E \vdash Point : P}}{\emptyset \vdash Circle : [center : P]} \qquad \dfrac{\dfrac{\dfrac{\dfrac{F[y \leftarrow Q] \vdash y : Q \qquad F[z \leftarrow Q] \vdash z : Q}{F \vdash ColorPoint : Q}}{F \vdash ColorPoint.move : Q}}{F \vdash ColorPoint.move.setcolor : Q}}{F \vdash ColorPoint.move.setcolor : P}}{\emptyset \vdash ColorCircle : [center : P]}}{\emptyset \vdash ColorCircle.center : P}}{\emptyset \vdash ColorCircle.center.move : P}$$

Notice the use of subsumption with $Q \leq P$.

We now show how the *NP*-time type inference algorithm works when given the above program. The expression

$$ColorCircle.center.move$$

yields the following S-system.

| Occurrence | Constraints |
|---|---|
| $x$ | $x \leq [\![x]\!]$ |
| $Point$ | $[move : \langle\!\langle x \rangle\!\rangle] \leq [\![Point]\!]$ |
| | $x = [move : \langle\!\langle x \rangle\!\rangle]$ |
| | $[\![x]\!] \leq \langle\!\langle x \rangle\!\rangle\{x\}$ |
| $y$ | $y \leq [\![y]\!]$ |
| $z$ | $z \leq [\![z]\!]$ |
| $ColorPoint$ | $[move : \langle\!\langle y \rangle\!\rangle \quad setcolor : \langle\!\langle z \rangle\!\rangle] \leq [\![ColorPoint]\!]$ |
| | $y = [move : \langle\!\langle y \rangle\!\rangle \quad setcolor : \langle\!\langle z \rangle\!\rangle]$ |
| | $z = [move : \langle\!\langle y \rangle\!\rangle \quad setcolor : \langle\!\langle z \rangle\!\rangle]$ |
| | $[\![y]\!] \leq \langle\!\langle y \rangle\!\rangle\{y\}$ |
| | $[\![z]\!] \leq \langle\!\langle z \rangle\!\rangle\{z\}$ |
| $Circle$ | $[center : \langle\!\langle Point \rangle\!\rangle] \leq [\![Circle]\!]$ |
| | $d = [center : \langle\!\langle Point \rangle\!\rangle]$ |
| | $[\![Point]\!] \leq \langle\!\langle Point \rangle\!\rangle\{d\}$ |
| $ColorCircle$ | $[\![Circle]\!] \leq [\![ColorCircle]\!]$ |
| | $[\![Circle]\!] = e$ |
| | $[\![Circle]\!] \leq [center : [\![ColorPoint.move.setcolor]\!]]$ |
| $ColorPoint.move$ | $[\![ColorPoint]\!] \leq [move : \langle ColorPoint.move \rangle]$ |
| | $\langle ColorPoint.move \rangle\{[\![ColorPoint]\!]\} \leq [\![ColorPoint.move]\!]$ |
| $ColorPoint.move.setcolor$ | $[\![ColorPoint.move]\!] \leq [setcolor : \langle ColorPoint.move.setcolor \rangle]$ |
| | $\langle ColorPoint.move.setcolor \rangle\{[\![ColorPoint.move]\!]\} \leq [\![ColorPoint.move.setcolor]\!]$ |

| | |
|---|---|
| $ColorCircle.center$ | $\llbracket ColorCircle \rrbracket \leq [center : \langle ColorCircle.center \rangle]$ |
| | $\langle ColorCircle.center \rangle \{\llbracket ColorCircle \rrbracket\} \leq \llbracket ColorCircle.center \rrbracket$ |
| $ColorCircle.center.move$ | $\llbracket ColorCircle.center \rrbracket \leq [move : \langle ColorCircle.center.move \rangle]$ |
| | $\langle ColorCircle.center.move \rangle \{\llbracket ColorCircle.center \rrbracket\} \leq \llbracket ColorCircle.center.move \rrbracket$ |

In the left column are all occurrences of subterms of $ColorCircle.center.move$. In the right column we show the constraints that are generated for each occurrence, according to the rules (6)–(14) in Section 4.

We denote this S-system by $\mathcal{C}$. Choose

$$\begin{aligned} S \; = \{ \;\; & \langle\!\langle x \rangle\!\rangle, \langle\!\langle y \rangle\!\rangle, \langle\!\langle z \rangle\!\rangle, \\ & \langle ColorPoint.move \rangle, \langle ColorPoint.move.setcolor \rangle, \\ & \langle ColorCircle.center.move \rangle \;\; \} \; . \end{aligned}$$

The S-system $F_S(\mathcal{C})$ looks as follows.

| | |
|---|---|
| $\langle\!\langle x \rangle\!\rangle = \mathsf{selftype}$ | $x \leq \llbracket x \rrbracket$ |
| $\langle\!\langle y \rangle\!\rangle = \mathsf{selftype}$ | $[move : \langle\!\langle x \rangle\!\rangle] \leq \llbracket Point \rrbracket$ |
| $\langle\!\langle z \rangle\!\rangle = \mathsf{selftype}$ | $x = [move : \langle\!\langle x \rangle\!\rangle]$ |
| $\langle ColorPoint.move \rangle = \mathsf{selftype}$ | $\llbracket x \rrbracket \leq x$ |
| $\langle ColorPoint.move.setcolor \rangle = \mathsf{selftype}$ | $y \leq \llbracket y \rrbracket$ |
| $\langle ColorCircle.center.move \rangle = \mathsf{selftype}$ | $z \leq \llbracket z \rrbracket$ |
| $\langle\!\langle Point \rangle\!\rangle \leq [\,]$ | $[move : \langle\!\langle y \rangle\!\rangle \;\; setcolor : \langle\!\langle z \rangle\!\rangle] \leq \llbracket ColorPoint \rrbracket$ |
| $\langle ColorCircle.center \rangle \leq [\,]$ | $y = [move : \langle\!\langle y \rangle\!\rangle \;\; setcolor : \langle\!\langle z \rangle\!\rangle]$ |
| $x \leq [\,]$ | $z = [move : \langle\!\langle y \rangle\!\rangle \;\; setcolor : \langle\!\langle z \rangle\!\rangle]$ |
| $y \leq [\,]$ | $\llbracket y \rrbracket \leq y$ |
| $z \leq [\,]$ | $\llbracket z \rrbracket \leq z$ |
| $d \leq [\,]$ | $[center : \langle\!\langle Point \rangle\!\rangle] \leq \llbracket Circle \rrbracket$ |
| $e \leq [\,]$ | $d = [center : \langle\!\langle Point \rangle\!\rangle]$ |
| $\llbracket x \rrbracket \leq [\,]$ | $\llbracket Point \rrbracket \leq \langle\!\langle Point \rangle\!\rangle$ |
| $\llbracket y \rrbracket \leq [\,]$ | $\llbracket Circle \rrbracket \leq \llbracket ColorCircle \rrbracket$ |
| $\llbracket z \rrbracket \leq [\,]$ | $\llbracket Circle \rrbracket = e$ |
| $\llbracket Point \rrbracket \leq [\,]$ | $\llbracket Circle \rrbracket \leq [center : \llbracket ColorPoint.move.setcolor \rrbracket]$ |
| $\llbracket ColorPoint.move.setcolor \rrbracket \leq [\,]$ | $\llbracket ColorPoint \rrbracket \leq [move : \langle ColorPoint.move \rangle]$ |
| $\llbracket ColorCircle.center \rrbracket \leq [\,]$ | $\llbracket ColorPoint \rrbracket \leq \llbracket ColorPoint.move \rrbracket$ |
| $\llbracket ColorCircle.center.move \rrbracket \leq [\,]$ | $\llbracket ColorPoint.move \rrbracket \leq [setcolor : \langle ColorPoint.move.setcolor \rangle]$ |
| $\llbracket ColorPoint \rrbracket \leq [\,]$ | $\llbracket ColorPoint.move \rrbracket \leq \llbracket ColorPoint.move.setcolor \rrbracket$ |
| $\llbracket ColorPoint.move \rrbracket \leq [\,]$ | $\llbracket ColorCircle \rrbracket \leq [center : \langle ColorCircle.center \rangle]$ |
| $\llbracket Circle \rrbracket \leq [\,]$ | $\langle ColorCircle.center \rangle \leq \llbracket ColorCircle.center \rrbracket$ |
| $\llbracket ColorCircle \rrbracket \leq [\,]$ | $\llbracket ColorCircle.center \rrbracket \leq [move : \langle ColorCircle.center.move \rangle]$ |
| | $\llbracket ColorCircle.center \rrbracket \leq \llbracket ColorCircle.center.move \rrbracket$ |

The constraint system $F_S(\mathcal{C})$ has the solution $L$ where:

$$L(W) = \begin{cases} \textsf{selftype} & \text{if } W \in S \\ [move : \textsf{selftype}] & \text{if } W \in \{ \quad x, [\![x]\!], [\![Point]\!], \langle\!\langle Point \rangle\!\rangle, \\ & \quad [\![ColorPoint.move.setcolor]\!], \\ & \quad [\![ColorCircle.center]\!], \\ & \quad \langle ColorCircle.center \rangle, \\ & \quad [\![ColorCircle.center.move]\!] \ \} \\ [move : \textsf{selftype} \quad setcolor : \textsf{selftype}] & \text{if } W \in \{ \quad y, [\![y]\!], z, [\![z]\!], [\![ColorPoint]\!], \\ & \quad [\![ColorPoint.move]\!] \ \} \\ [center : [move : \textsf{selftype}]] & \text{if } W \in \{ \quad d, e, [\![Circle]\!], [\![ColorCircle]\!] \ \} \end{cases}$$

In conclusion, if we annotate the two move methods and the setcolor method with selftype as the return type, then the program is typable.

Notice that $L$ does not assign selftype to $\langle\!\langle Point \rangle\!\rangle$ and $\langle\!\langle ColorCircle.center \rangle\!\rangle$. If we define $L'$ such that it agrees with $L$ except

$$L'(\langle\!\langle Point \rangle\!\rangle) = L'(\langle\!\langle ColorCircle.center \rangle\!\rangle) = \textsf{selftype} \ ,$$

then $L'$ is not a solution of $\mathcal{C}$. To see this, notice the constraints

$$\begin{aligned} &[center : \langle\!\langle Point \rangle\!\rangle] \leq [\![Circle]\!] \\ &[\![Circle]\!] \leq [\![ColorCircle]\!] \\ &\langle ColorCircle.center \rangle \{ [\![ColorCircle]\!] \} \leq [\![ColorCircle.center]\!] \\ &[\![ColorCircle.center]\!] \leq [move : \langle ColorCircle.center.move \rangle] \ . \end{aligned}$$

From $L'(\langle ColorCircle.center \rangle) = \textsf{selftype}$ and the transitivity of $\leq$ we have that $L'$ should satisfy

$$[center : \textsf{selftype}] \leq [move : \textsf{selftype}] \ ,$$

which is impossible.

# 7 Conclusion

Throughout, we have considered a type system with recursive types. Our constructions also work without recursive types (the details of checking this are left to the reader). We have thus completed the following table.

| Selftype | Recursive types | Subtyping | Type inference |
|---|---|---|---|
| | | $\surd$ | $O(n^3)$ time, P-complete [14] |
| | $\surd$ | $\surd$ | $O(n^3)$ time, P-complete [14] |
| $\surd$ | | $\surd$ | $NP$-complete [this paper] |
| $\surd$ | $\surd$ | $\surd$ | $NP$-complete [this paper] |

**Acknowledgments** We thank Martín Abadi, Kim Bruce, and Luca Cardelli for helpful discussions. We also thank the anonymous referees for many insightful suggestions and comments. The first author was supported by BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation). The second author was supported by NSF grant CCR–9417382 and ONR Contract N00014–92–J–1310.

# References

[1] Martín Abadi and Luca Cardelli. A semantics of object types. In *Proc. LICS'94, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 332–341, 1994.

[2] Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. In *Proc. ESOP'94, European Symposium on Programming*, pages 1–25. Springer-Verlag (*LNCS* 788), 1994.

[3] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *Proc. TACS'94, Theoretical Aspects of Computing Software*, pages 296–320. Springer-Verlag (*LNCS* 789), 1994.

[4] Martín Abadi and Luca Cardelli. An imperative object calculus. In *Proc. TAPSOFT'95, Theory and Practice of Software Development*, pages 471–485. Springer-Verlag (*LNCS* 915), Aarhus, Denmark, May 1995.

[5] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[6] Kim B. Bruce. Safe type checking in a statically typed object-oriented programming language. In *Proc. POPL'93, Twentieth Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 285–298, 1993.

[7] Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, Robert van Gent, Allyn Dimock, and Robert Muller. Safe and decidable type checking in an object-oriented language. In *Proc. OOPSLA'93, ACM SIGPLAN Eighth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 29–46, 1993.

[8] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, 1994. Preliminary version in Proc. OOPSLA'89, ACM SIGPLAN Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications, pages 433–443, New Orleans, Louisiana, October 1989.

[9] My Hoang and John C. Mitchell. Lower bounds on type inference with subtypes. In *Proc. POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 176–185, 1995.

[10] Bent B. Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The BETA programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. MIT Press, 1987.

[11] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[12] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Seventeenth Symposium on Principles of Programming Languages*, pages 109–124, 1990.

[13] John C. Mitchell, Furio Honsell, and Kathleen Fisher. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994. Also in Proc. LICS'93, pp.26–38.

[14] Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995. Preliminary version in Proc. LICS'94, Ninth Annual IEEE Symposium on Logic in Computer Science, pages 186–195, Paris, France, July 1994.

[15] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.

[16] Jens Palsberg and Michael I. Schwartzbach. Static typing for object-oriented programming. *Science of Computer Programming*, 23(1):19–53, 1994.

[17] Jerzy Tiuryn. Subtype inequalities. In *LICS'92, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 308–315, 1992.

[18] Sergei Vorobyov. Structural decidable extensions of bounded quantification. In *Proc. POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, 1995.