BRICS

Basic Research in Computer Science

BRICS RS-95-34 J. Palsberg: Type Inference with Selftype

Type Inference with Selftype

Jens Palsberg

BRICS Report Series **RS-95-34**

ISSN 0909-0878 **June 1995**

Copyright © 1995, BRICS, Department of Computer Science
University of Aarhus. All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

Type Inference with Selftype

Jens Palsberg

BRICS[*]
Department of Computer Science
University of Aarhus
Ny Munkegade
DK-8000 Aarhus C, Denmark

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 type inference algorithm for a type system with selftype, recursive types, and subtyping. The example language is the object calculus of Abadi and Cardelli, and the type inference algorithm runs in nondeterministic polynomial time.

[*]Basic Research in Computer Science, Centre of the Danish National Research Foundation. E-mail: `palsberg@daimi.aau.dk`.

1 Introduction

1.1 Background

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. For a denotational semantics of inheritance, see for example [7].

Typing self in the presence of inheritance has been studied by Abadi and Cardelli [3, 2, 1, 4], Bruce [5, 6], Palsberg and Schwartzbach [11, 12], 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. Consider now 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.

One approach to giving self a special treatment is the use of *selftype*, "the type of self", which enables flexible type annotations that are impossible with recursive types and subtyping. Selftype has been studied by Abadi and Cardelli [4], Bruce [5, 6], and others, and used in for example Eiffel [9] (Eiffel uses the syntax `like Current` for selftype). In the example with `Point` and `ColorPoint`, we can annotate the `move` method with selftype as the return type. This has the effect that the type of `ColorPoint.move` has the *same* type as `ColorPoint`, and thus `ColorPoint.move.setcolor` is type-correct.

Although the object `ColorPoint` extends the object `Point`, this use of inheritance is *not* essential for creating examples that demonstrates the usefulness of selftype. In Section 1.3 we present an example where an object overrides a method in its parent object. That example is typable with selftype, but not with recursive types and subtyping.

There is no common agreement on the "right" type system with selftype. For example, when comparing the type rules of Abadi and Cardelli [4] with those of Bruce et al. [5, 6], we find both striking similarities, such as in the rules for message send, and significant differences, such as Bruce's use of the $\leq_{meth}$ relation on types. Both these type systems have been proved sound, and for Bruce's language TOOPLE, type checking is decidable [6].

Open until now is the problem of type inference with selftype. Of course, the complexity of such a type inference problem depends on the exact details of the type system. In this paper, we address the following fundamental question:

> **Fundamental question.** Can we design a useful type system with selftype such that type inference is decidable?

In other words: are selftype and type inference compatible?

1.2 Our Results

We present a type inference algorithm for a type system with selftype, recursive types, and subtyping. The example language is the object calculus of Abadi and Cardelli, and the type inference algorithm runs in nondeterministic polynomial time. Intuitively, our algorithm works by first guessing which methods should be annotated with selftype as the return type, and then solving the remaining type inference problem in polynomial time. It remains open if type inference is NP-complete or in polynomial time.

Type inference in the smaller type system without selftype is computable in $O(n^3)$ time and it is P-complete [10]. In Section 1.3 we present a program which is typable with selftype but not without. Thus, selftype makes the type system more powerful and type inference remains decidable.

Our type system with selftype is essentially a subsystem of the one of Abadi and Cardelli in [4]. The key restriction is that a method cannot both have selftype as return type and also be overridable. It remains open if type inference in the full version of Abadi and Cardelli's type system is decidable.

1.3 Example

We now present an example program which uses overriding of methods but not extension of objects.

```
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. This behavior may of course seem odd, but from a typing perspective, we would prefer that it does *not* make a difference if the center method returns `ColorPoint` or `ColorPoint.move.setcolor`. In both cases, the main program executes without errors.

The key aspects of the example can be directly represented in the object calculus of Abadi and Cardelli [3, 2, 1, 4], as follows.

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

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 [10]. 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.

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

If we change the body of the *ColorCircle*'s center method to return simply *ColorPoint*, then the program *is* typable with recursive types and subtyping (actually with subtyping alone). This state of affairs is not satisfactory and calls for something to supplement or replace recursive types and subtyping.

This call is answered by our type system with selftype. With that type system the type of *ColorPoint* is a subtype of the type of *Point*, and the program is typable:

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

Note that our type system can type this program even though it is strictly less powerful than the one suggested by Abadi and Cardelli in [4].

In the following section we briefly present 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, and in Section 5 we prove that the constraint problem is solvable in nondeterministic polynomial time. Finally, in Section 6 we give an example of how the algorithm works.

2 Abadi and Cardelli's Calculus

Abadi and Cardelli has presented an untyped object calculus, called the ς-calculus. The ς-terms are generated by the following grammar:

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

We use a, b, c to range over ς-terms. 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 components does not matter. In a method $\varsigma(x)b$, we have that x is the self variable and b is the body. Thus, in the body of a method we can refer to any enclosing object, like in the Beta language [8].

The reduction rules for ς-terms are as follows. If $o \equiv [l_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$, then, for $j \in 1..n$,

- $o.l_j \ \rightsquigarrow \ b_j[o/x_j]$
- $o.l_j \Leftarrow \varsigma(y)b \ \rightsquigarrow \ o[l_j \leftarrow \varsigma(y)b]$

Here, $a[o/x]$ denotes the ς-term a with o substituted for free occurrences of x (after renaming bound variables if necessary); and $o[l_j \leftarrow \varsigma(y)b]$ denotes the ς-term o with the l_j field replaced by $\varsigma(y)b$. An evaluation context is an expression with one hole. For an evaluation context $a[.]$, if $b \rightsquigarrow b'$, then $a[b] \rightsquigarrow a[b']$.

A ς-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 field.

For an example of a reduction, consider the object $o \equiv [l = \varsigma(x)x.l]$. The expression $o.l$ yields the infinite computation: $o.l \rightsquigarrow x.l[o/x] \equiv o.l \rightsquigarrow \ldots$

3 The Type System

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

The concrete syntax of object types is presented by the following grammar:

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

The labels l_i are drawn from some possibly infinite set $\mathcal{N}$ of method names. We denote by $\mathcal{C}$ the powerset of $\mathcal{N}$.

Define $\Sigma = \{\mathsf{Selftype}\} \cup \mathcal{C}$. Each type denotes a regular tree over Σ. Intuitively, such a tree can be obtained from a type by infinite unfolding of the type.

Given a type, we represent the corresponding regular tree by a *term* over Σ, that is, a partial function

$$t : \mathcal{N}^* \ \rightarrow \ \Sigma$$

with domain $\mathcal{D}(t)$ satisfying the following properties:

- $\mathcal{D}(t)$ is nonempty and prefix-closed;
- if $t(\alpha) = \mathsf{Selftype}$, then $\{l \mid \alpha l \in \mathcal{D}(t)\} = \emptyset$; and
- if $t(\alpha) = \{l_i \in \mathcal{N} \mid i \in 1..n\}$, then $\{l \mid \alpha l \in \mathcal{D}(t)\} = \{l_i \mid i \in 1..n\}$.

Intuitively, $\mathcal{D}(t)$ is the set of paths from the root in the tree, and t maps each such path to the symbol at the end of the path. In the remainder of the paper, we always work with the term representation of types.

Let t be a term and $\alpha \in \mathcal{N}^*$. Define the partial function $t\!\downarrow\!\alpha : \mathcal{N}^* \to \Sigma$ by

$$t\!\downarrow\!\alpha(\beta) \quad = \quad t(\alpha\beta) \; .$$

If $t\!\downarrow\!\alpha$ has nonempty domain, then it is a term, and is called the *subterm of t at position α*.

A term t is said to be *regular* if it has only finitely many distinct subterms; *i.e.*, if $\{t\!\downarrow\!\alpha \mid \alpha \in \mathcal{N}^*\}$ is a finite set. The terms denoted by object types are regular terms. The set of all regular terms over Σ is denoted T_Σ.

We now define operators $\mathsf{selftype}^{T_\Sigma}$ and $[l_i : A_i\ ^{i\in 1..n}]^{T_\Sigma}$ on terms that correspond to the type constructs $\mathsf{selftype}$ and $[l_i : A_i\ ^{i\in 1..n}]$. For $l_1, \ldots, l_n \in \mathcal{N}$, $A_1, \ldots, A_n \in T_\Sigma$, $j \in 1..n$, and $\alpha \in \mathcal{N}^*$, define

$$\begin{array}{rcl}
\mathcal{D}(\mathsf{selftype}^{T_\Sigma}) & = & \{\epsilon\} \\
\mathsf{selftype}^{T_\Sigma}(\epsilon) & = & \mathsf{Selftype} \\
\mathcal{D}([l_i : A_i\ ^{i\in 1..n}]^{T_\Sigma}) & = & \{\epsilon\} \cup \bigcup_{i=1}^{n} \{l_i\alpha \mid \alpha \in \mathcal{D}(A_i)\} \\
{[l_i : A_i\ ^{i\in 1..n}]}^{T_\Sigma}(\epsilon) & = & \{l_i \mid i \in 1..n\} \\
{[l_i : A_i\ ^{i\in 1..n}]}^{T_\Sigma}(l_j\alpha) & = & A_j(\alpha) \; .
\end{array}$$

The set $T_\Sigma \setminus \{\mathsf{selftype}^{T_\Sigma}\}$ is denoted P_Σ. At the risk of ambiguity, we omit the superscript T_Σ on the operators $\mathsf{selftype}^{T_\Sigma}$ and $[l_i : A_i\ ^{i\in 1..n}]^{T_\Sigma}$.

The following properties are immediate from the definitions:

(i) $[l_i : A_i\ ^{i\in 1..n}]\!\downarrow\! l_i = A_i$

(ii) $(A\!\downarrow\!\alpha)\!\downarrow\!\beta = A\!\downarrow\!\alpha\beta$

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

$$\mathsf{selftype} \leq \mathsf{selftype}$$

and second, if $A \neq \mathsf{selftype}$ and $B \neq \mathsf{selftype}$, then

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

Clearly, $\leq$ is a partial order. 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 $[l : [m : A]] \not\leq [l : [\ ]]$. Notice that if $A \leq B$, then $\mathcal{D}(B) \subseteq \mathcal{D}(A)$.

If A, B are object types, define

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

We now present the typing rules. If a is a ς-term, A is an object type, and E is a type environment, that is, a partial function assigning elements of P_Σ to variables, then the judgement $E \vdash a : A$ means that a has the type A in the environment E. This holds when the judgement is derivable using the following five rules:

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

$$\frac{E[x_i \leftarrow A] \vdash b_i : B_i\{A\} \quad \forall i \in 1..n}{E \vdash [l_i = \varsigma(x_i)b_i \ ^{i \in 1..n}] : A} \quad (\text{where } A = [l_i : B_i \ ^{i \in 1..n}]) \tag{2}$$

$$\frac{E \vdash a : A}{E \vdash a.l : B\{A\}} \quad (\text{where } A \leq [l : B]) \tag{3}$$

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

$$\frac{E \vdash a : A \quad A \leq B}{E \vdash a : B} \tag{5}$$

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 [10]. Specifically, if selftype is never used, then $B\{A\} = B$ and the rules take the form used in [10]. The type rules may also be understood as a simplification of those introduced by Abadi and Cardelli in [4]. The key restriction is found in rule (4) where the condition $B \neq \textsf{selftype}$ ensures that a method cannot both have selftype as return type and also be overridable.

If $E \vdash a : A$ is derivable, we say that a is *well-typed* with type A.

Theorem 3.1 (Subject Reduction) *If $E \vdash a : t$ and $a \leadsto a'$, then $E \vdash a' : t$.*

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

For an example of a type derivation, let us consider the example term from Section 1.3. Define

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

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

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

Notice the use of subsumption with $Q \leq P$ which was also mentioned in Section 1.3.

4 From Rules to Constraints

In this section we prove that the type inference problem is log space reducible to solving a finite system of type constraints. The constraints isolate the essential combinatorial structure of the type inference problem.

Definition 4.1 Given two denumerable and disjoint sets $\mathcal{U}$ and $\mathcal{V}$ of variables, an S-system (selftype-system) over $\mathcal{U}$ and $\mathcal{V}$ is a finite set of constraints of the forms:

$$W \leq W'$$
$$\textsf{if } U = \textsf{selftype then } W \leq W' \textsf{ else } W'' \leq U$$
$$\textsf{if } U = \textsf{selftype then } W \leq W' \textsf{ else } U \leq W''$$

where W, W', W'' are of the forms V, $[l_i : U_i \ ^{i \in 1..n}]$, or $[l_i : V_i \ ^{i \in 1..n}]$, and where $U, U_1, \ldots, U_n \in \mathcal{U}$ and $V, V_1, \ldots, V_n \in \mathcal{V}$.

A *solution* for an S-system is a pair of maps (L, M), where $L : \mathcal{U} \to T_\Sigma$ and $M : \mathcal{V} \to P_\Sigma$, such that all constraints are satisfied when elements of $\mathcal{U}$ are mapped to types by L, and elements of $\mathcal{V}$ are mapped to types by M. □

For an example of an S-system, see Section 6. In comparison with the AC-systems of [10], the novel aspect of S-systems is the use of conditional constraints.

Given a ς-term c, assume that it has been α-converted so that all bound variables are distinct. We will now generate an S-system where the bound variables of c are a subset of the variables used in the constraint system. This will be convenient in the statement and proof of Theorem 4.2 below. Let X be the set of bound variables in c, and let Y be a set of variables disjoint from X consisting of one variable $[\![b]\!]$ for each occurrence of a subterm of b of c. Define $\mathcal{V} = X \cup Y$. Moreover, let $\mathcal{U}$ be a set of variables disjoint from $\mathcal{V}$ consisting of one variable $\langle a.l \rangle$ for each occurrence of a subterm $a.l$ of c, and consisting of one variable $\langle b_i \rangle$ for each occurrence of a subterm $[l_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$ of c and for each $i \in 1..n$. (The notations $[\![b]\!]$, $\langle a.l \rangle$, and $\langle b_i \rangle$ 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 the context which occurrence is meant.)

We generate from c the following S-system over $\mathcal{U}$ and $\mathcal{V}$:

- for every occurrence in c of a bound variable x, the constraint

$$x \leq [\![x]\!] \tag{6}$$

- for every occurrence in c of a subterm of the form $[l_i = \varsigma(x_i)b_i \;^{i \in 1..n}]$, the constraint

$$[l_i : \langle b_i \rangle \;^{i \in 1..n}] \leq [\![[l_i = \varsigma(x_i)b_i \;^{i \in 1..n}]]\!] \tag{7}$$

and for every $j \in 1..n$, the two constraints

$$x_j = [l_i : \langle b_i \rangle \;^{i \in 1..n}] \tag{8}$$

$$\textsf{if } \langle b_j \rangle = \textsf{selftype then } x_j = [\![b_j]\!] \textsf{ else } \langle b_j \rangle = [\![b_j]\!] \tag{9}$$

- for every occurrence in c of a subterm of the form $a.l$, the two constraints

$$[\![a]\!] \leq [l : \langle a.l \rangle] \tag{10}$$

$$\textsf{if } \langle a.l \rangle = \textsf{selftype then } [\![a]\!] \leq [\![a.l]\!] \textsf{ else } \langle a.l \rangle \leq [\![a.l]\!] \tag{11}$$

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

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

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

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

Each equality $A = B$ denotes the two inequalities $A \leq B$ and $B \leq A$. Moreover, each constraint of the form

$$\textsf{if } U = \textsf{selftype then } V = V' \textsf{ else } U = V''$$

denotes the two constraints

$$\textsf{if } U = \textsf{selftype then } V \leq V' \textsf{ else } U \leq V''$$
$$\textsf{if } U = \textsf{selftype then } V' \leq V \textsf{ else } V'' \leq U \;.$$

Denote by $C(c)$ the S-system of constraints generated from c in this fashion. For a ς-term of size n, the S-system $C(c)$ is of size $O(n)$, and it is generated using $O(\log n)$ space. We show below that the solutions of $C(c)$ correspond to the possible type annotations of c in a sense made precise by Theorem 4.2. For an example of an S-system generated from a ς-term, see Section 6.

Let E be a type environment assigning a type in P_Σ to each variable occurring freely in c. If $M : \mathcal{V} \to P_\Sigma$, we say the M *extends* E if E and M agree on the domain of E.

Theorem 4.2 *The judgement $E \vdash c : A$ is derivable if and only if there exists a solution (L, M) of $C(c)$ such that M extends E and $M([\![c]\!]) = A$. In particular, if c is closed, then c is well-typed with type A if and only if there exists a solution (L, M) of $C(c)$ such that $M([\![c]\!]) = A$.*

Proof. The proof uses the same technique as the proof of Lemma 4.2 in [10].

We first prove that if $C(c)$ has a solution (L, M), then $M \vdash c : M([\![c]\!])$ is derivable. We proceed by induction on the structure of c. For the base case, $M \vdash x : M([\![x]\!])$ is derivable using rules (1) and (5), since $M(x) \leq M([\![x]\!])$. For the induction step, consider first $[l_i = \varsigma(x_i)b_i \; ^{i\in 1..n}]$. Let $A = [l_i : L(\langle b_i \rangle) \; ^{i\in 1..n}]$. To derive $M \vdash [l_i = \varsigma(x_i)b_i \; ^{i\in 1..n}] : M([\![[l_i = \varsigma(x_i)b_i \; ^{i\in 1..n}]]\!])$, by rule (5) and the fact that $A \leq M([\![[l_i = \varsigma(x_i)b_i \; ^{i\in 1..n}]]\!])$, it suffices to derive $M \vdash [l_i = \varsigma(x_i)b_i \; ^{i\in 1..n}] : A$. The side condition of (2) is clearly satisfied, so it suffices to derive, for each $i \in 1..n$, $M[x_i \leftarrow A] \vdash b_i : (L(\langle b_i \rangle))\{A\}$. Since $M(x_i) = A$ for each $i \in 1..n$, it suffices to derive $M \vdash b_i : (L(\langle b_i \rangle))\{A\}$. For each $i \in 1..n$, there are two cases. If $L(\langle b_i \rangle) = \mathsf{selftype}$, then $M(x_i) = M([\![b_i]\!])$ and $(L(\langle b_i \rangle))\{A\} = A$, so since $M(x_i) = A$, we get $(L(\langle b_i \rangle))\{A\} = M([\![b_i]\!])$, and hence the desired derivation is provided by the induction hypothesis. If $L(\langle b_i \rangle) \neq \mathsf{selftype}$, then $L(\langle b_i \rangle) = M([\![b_i]\!])$ and $(L(\langle b_i \rangle))\{A\} = L(\langle b_i \rangle)$, so we get $(L(\langle b_i \rangle))\{A\} = M([\![b_i]\!])$, and again the desired derivation is provided by the induction hypothesis.

Now consider $a.l$. Let $A = M([\![a]\!])$. From the induction hypothesis, we obtain a derivation of $M \vdash a : A$. By rule (3) and the fact that $A \leq [l : L(\langle a.l \rangle)]$, we obtain a derivation of $M \vdash a.l : (L(\langle a.l \rangle))\{A\}$. There are two cases. If $L(\langle a.l \rangle) = \mathsf{selftype}$, then $A \leq M([\![a.l]\!])$ and $(L(\langle a.l \rangle))\{A\} = A$, so $(L(\langle a.l \rangle))\{A\} \leq M([\![a.l]\!])$, and hence $M \vdash a.l : M([\![a.l]\!])$ can be derived using rule (5). If $L(\langle a.l \rangle \neq \mathsf{selftype}$, then $L(\langle a.l \rangle) \leq M([\![a.l]\!])$ and $(L(\langle a.l \rangle))\{A\} = L(\langle a.l \rangle)$, so $(L(\langle a.l \rangle))\{A\} \leq M([\![a.l]\!])$, and again $M \vdash a.l : M([\![a.l]\!])$ can be derived using rule (5).

Finally, consider $a.l \Leftarrow \varsigma(x)b$. Let $A = M([\![a]\!])$. To derive $M \vdash a.l \Leftarrow \varsigma(x)b : M([\![a.l \Leftarrow \varsigma(x)b]\!])$, by rule (5) and the fact that $A \leq M([\![a.l \Leftarrow \varsigma(x)b]\!])$, it suffices to derive $M \vdash a.l \Leftarrow \varsigma(x)b : A$. From the facts that $A \leq [l : M([\![b]\!])]$ and $M([\![b]\!]) \in P_\Sigma$, we get that the side conditions of rule (4) are satisfied and that it suffices to derive $M \vdash a : A$ and $M[x \leftarrow A] \vdash b : M([\![b]\!])$. Since $A = M(x)$, the desired derivations are provided by the induction hypothesis.

We then prove that if $E \vdash c : A$ is derivable, then there exists a solution (L, M) of $C(c)$ such that M extends E and $M([\![c]\!]) = A$.

Suppose $E \vdash c : A$ is derivable, and consider a derivation of minimal length. Since the derivation is minimal, there is exactly one application of the rule (1) involving a particular occurrence of a variable x, exactly one application of the rule (2) involving a particular occurrence of a subterm $[l_i = \varsigma(x_i)b_i \; ^{i \in 1..n}]$, exactly one application of the rule (3) involving a particular occurrence of a subterm $a.l$, and exactly one application of the rule (4) involving a particular occurrence of a subterm $a.l \Leftarrow \varsigma(x)b$. In the case of a bound variable x, there is a unique type B_x such that $F(x) = B_x$ for any F such that a judgement $F \vdash a : B'$ appears in the derivation for some occurrence of a subterm a of $\varsigma(x)b$; this can be proved by induction on the structure of the derivation of $F \vdash a : B'$. Finally, there can be at most one application of the rule (5) involving a particular occurrence of any subterm; if there were more than one, they could be combined using the transitivity of $\leq$ to give a shorter derivation.

Now construct (L, M) as follows. For every free variable x of c, define $M(x) = E(x)$. For every bound variable x of c, define $M(x) = B_x$. For every occurrence of a subterm a of c, find the last judgement in the derivation of the form $F \vdash a : B$ involving that occurrence of a, and define $M([\![a]\!]) = B$. Intuitively, the *last* judgement of the form $F \vdash a : B$ means the judgement *after* the use of subsumption. For each occurrence of a subterm $a.l$ of c, find the unique application of the rule (3) deriving the judgement $F \vdash a.l : B\{A'\}$ from the premise $F \vdash a : A'$ where $A' \leq [l : B]$, and define $L(\langle a.l \rangle) = B$. For each occurrence of a subterm $[l_i = \varsigma(x_i)b_i \; ^{i \in 1..n}]$ of c, find the unique application of the rule (2) deriving the judgement $F \vdash [l_i = \varsigma(x_i)b_i \; ^{i \in 1..n}] : A'$ from the premises $F[x_i \leftarrow A'] \vdash b_i : B_i\{A'\}$ for $i \in 1..n$ where $A' = [l_i : B_i \; ^{i \in 1..n}]$, and define $L(\langle b_i \rangle) = B_i$ for $i \in 1..n$.

Certainly M extends E and $M([\![c]\!]) = A$. We now show that (L, M) is a solution of $C(c)$.

For an occurrence of a bound variable x, there are two cases. Suppose first that the variable is bound in a method that occurs in an object declaration. Find the unique application of the rule (2) deriving the judgement $F \vdash [l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}] : A$ from a family of premises where one of them is $F[x \leftarrow A] \vdash b : B_i$. Then $L(x) = A$. The rule (1) must have been applied to obtain a judgement of the form $G \vdash x : L(x)$ and only rule (5) applied to that occurrence of x thereafter, thus $L(x) \leq L([\![x]\!])$. Suppose then that the variable is bound in a method that occurs in a method override. Find the unique application of the rule (4) deriving the judgement $F \vdash a.l \Leftarrow \varsigma(x)b : A$ from two premises where one of them is $F[x \leftarrow A] \vdash b : B$. As before, we get that $L(x) \leq L([\![x]\!])$.

For an occurrence of a subterm of the form $[l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]$, find the unique application of the rule (2) deriving the judgement $F \vdash [l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}] : A'$ from the premises $F[x_i \leftarrow A'] \vdash b_i : B_i\{A'\}$ where $A' = [l_i : B_i\ ^{i\in 1..n}]$. Then $B_j = L(\langle b_j\rangle)$ and $M([\![b_j]\!]) = B_j\{A'\}$ for each $j \in 1..n$. Hence, $[l_i : L(\langle b_i\rangle)\ ^{i\in 1..n}] \leq M([\![[l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]]\!])$ and $M(x_j) = [l_i : L(\langle b_i\rangle)\ ^{i\in 1..n}]$ for each $j \in 1..n$. Moreover, for each $j \in 1..n$, if $L(\langle b_j\rangle) = \mathsf{selftype}$, then $B_j\{A'\} = A'$, so $M(x_j) = A' = B_j\{A'\} = M([\![b_j]\!])$, and if $L(\langle b_j\rangle) \neq \mathsf{selftype}$, then $B_j\{A'\} = B_j$, so $L(\langle b_j\rangle) = B_j = B_j\{A'\} = M([\![b_j]\!])$.

For an occurrence of a subterm of the form $a.l$, find the unique application of the rule (3) deriving the judgement $F \vdash a.l : B\{A'\}$ from the premise $F \vdash a : A$ where $A \leq [l : B]$. Then $B = L(\langle a.l\rangle)$ and $A = M([\![a]\!])$. Hence, $M([\![a]\!]) = A \leq [l : B] = [l : L(\langle a.l\rangle)]$. Moreover, if $L(\langle a.l\rangle) = \mathsf{selftype}$, then $B\{A'\} = A'$, so $M([\![a]\!]) = A' = B\{A'\} \leq M([\![a.l]\!])$, and if $L(\langle a.l\rangle) \neq \mathsf{selftype}$, then $B\{A'\} = B$, so $L(\langle a.l\rangle) = B = B\{A'\} \leq M([\![a.l]\!])$.

Finally, for an occurrence of a subterm of the form $a.l \Leftarrow \varsigma(x)b$, find the unique application of the rule (4) deriving the judgement $F \vdash a.l \Leftarrow \varsigma(x)b : A'$ from the premise $F \vdash a : A'$ and $F[x \leftarrow A'] \vdash b : B$ where $A' \leq [l : B]$ and $B \neq \mathsf{selftype}$. Then $M([\![a]\!]) = A' \leq M([\![a.l \Leftarrow \varsigma(x)b]\!])$, and $A' = M(x)$. Moreover, $M([\![b]\!]) = B$, so $M([\![a]\!]) = A' \leq [l : B] = [l : M([\![b]\!])]$. □

5 Solving Constraints

To solve an arbitrary S-system, we will use a use a non-deterministic algorithm to transform it into a so-called ACS-system which then can be solved in polynomial time.

The notion of an ACS-system is a slight extension of that of an AC-system that was studied by Palsberg [10]. The extension is the constant selftype. Intuitively, selftype enjoys a special status in an S-system because of the conditional constraints. In contrast, selftype is an "ordinary" constant in an ACS-system.

Definition 5.1 Given a denumerable set of variables $\mathcal{W}$, an *ACS-system* over $\mathcal{W}$ is a finite set of constraints of the forms:

$$V = \mathsf{selftype}$$
$$W \leq W'$$

where W, W' are of the forms V or $[l_i : V_i \ ^{i \in 1..n}]$, and where $V, V_1, \ldots, V_n \in \mathcal{W}$.

A *solution* for an ACS-system is a map $\psi : \mathcal{W} \rightarrow T_\Sigma$, such that all constraints are satisfied when elements of $\mathcal{W}$ are mapped to types by ψ. □

If we disallow the use of selftype in the constraints and in the solutions, then we get an AC-system. Type inference with recursive types and subtyping is log-space equivalent to solving AC-systems. Since the constant selftype has no special status in an ACS-system, it could be replaced by any other constant, e.g., Integer, Real, without changing the problem of solving constraints. If we extend the object calculus with constructs for computing with for example integers, then we can in log-space reduce the type inference problem to solving ACS-systems with Integer in the place of selftype.

In the journal version of [10], 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 selftype. Thus, solvability of an ACS-system is decidable in $O(n^3)$ time.

We now define a family of mappings F_S from S-systems to ACS-systems. Let C be an S-system over $\mathcal{U}$ and $\mathcal{V}$, and let $S \subseteq \mathcal{U}$. Intuitively, S is a guess on the set of variables that some solution of C would map to selftype. Define $F_S(C)$ to be the ACS-system over $\mathcal{U} \cup \mathcal{V}$ where

- For each $U \in S$, the constraint $U = \mathsf{selftype}$ is in $F_S(C)$.
- For each $V \in (\mathcal{U} \setminus S) \cup \mathcal{V}$, the constraint $V \leq [\,]$ is in $F_S(C)$.
- If a constraint of the form $W \leq W'$ is in C, then it is also in $F_S(C)$.

- If a constraint of the form $\textsf{if } U = \textsf{selftype then } W \leq W' \textsf{ else } W'' \leq W'''$ is in C, then

  (i) If $U \in S$, then $W \leq W'$ is in $F_S(C)$; and

  (ii) If $U \notin S$, then $W'' \leq W'''$ is in $F_S(C)$.

We can now prove our main result which relates solvability of S-systems to solvability of ACS-systems.

Theorem 5.2 (Main Result) *Suppose C is an S-system over $\mathcal{U}$ and $\mathcal{V}$. Then C is solvable if and only if there exist $S \subseteq \mathcal{U}$ such that $F_S(C)$ is solvable.*

Proof. Suppose first that C has solution (L, M). Define

$$\begin{array}{rcl} S & = & \{U \in \mathcal{U} \mid L(U) = \textsf{selftype}\} \\ \psi & : & \mathcal{U} \cup \mathcal{V} \rightarrow T_\Sigma \\ \psi(W) & = & \left\{ \begin{array}{ll} L(W) & \text{if } W \in \mathcal{U} \\ M(W) & \text{if } W \in \mathcal{V} \end{array} \right. \end{array}$$

Clearly, ψ is a solution of $F_S(C)$.

Suppose then that we have $S \subseteq \mathcal{U}$ such that $F_S(C)$ has solution ψ. Define

$$\begin{array}{rcl} L & : & \mathcal{U} \rightarrow T_\Sigma \\ M & : & \mathcal{V} \rightarrow P_\Sigma \\ L(U) & = & \psi(U) \text{ if } U \in \mathcal{U} \\ M(V) & = & \psi(V) \text{ if } V \in \mathcal{V} \end{array}$$

Clearly, (L, M) is a solution of C. □

Corollary 5.3 *We can decide in nondeterministic polynomial time if an S-system has a solution.*

Proof. Suppose C is an S-system over $\mathcal{U}$ and $\mathcal{V}$. Guess $S \subseteq \mathcal{U}$. Transform C into $F_S(C)$, using log-space. Decide whether $F_S(C)$ is solvable, using $O(n^3)$ time. The conclusion then follows from Theorem 5.2. □

By combining Theorem 4.2 and Corollary 5.3, we obtain the following result.

Corollary 5.4 *The type inference problem for the type system with selftype, recursive types, and subtyping can be decided in nondeterministic polynomial time.*

Suppose we drop either or both of recursive types and subtyping. In each case, the type inference problem can be decided in nondeterministic polynomial time. by small modifications of the algorithm above, as follows. For the case of dropping recursive types, there is a slightly different algorithm for solving the generated ACS-system in $O(n^3)$ time, see [10]. For the case of dropping subtyping, the only change is that when generating the S-system, the inequalities in (6), (7), (11), and (12) should be changed to equalities. For the case of dropping both recursive types as subtyping, one should combine the changed mentioned in the two previous cases.

We have thus completed the following table.

Selftype	Recursive types	Subtyping	Type inference
			$O(n^3)$ time, P-complete [10]
		√	$O(n^3)$ time, P-complete [10]
	√		$O(n^3)$ time, P-complete [10]
	√	√	$O(n^3)$ time, P-complete [10]
√			*NP* [this paper]
√		√	*NP* [this paper]
√	√		*NP* [this paper]
√	√	√	*NP* [this paper]

6 Example of Type Inference

We now give an example of how the type inference algorithm works. The example program is the one from Section 1.3. The expression

$$ColorCircle.center.move$$

yields the following S-system.

Occurrence	Constraints
x	$x \leq [\![x]\!]$
$Point$	$[move : \langle x \rangle] \leq [\![Point]\!]$
	$x = [move : \langle x \rangle]$
	if $\langle x \rangle =$ selftype then $x = [\![x]\!]$ else $\langle x \rangle = [\![x]\!]$
y	$y \leq [\![y]\!]$
z	$z \leq [\![z]\!]$
$ColorPoint$	$[move : \langle y \rangle \quad setcolor : \langle z \rangle] \leq [\![ColorPoint]\!]$
	$y = [move : \langle y \rangle \quad setcolor : \langle z \rangle]$
	$z = [move : \langle y \rangle \quad setcolor : \langle z \rangle]$
	if $\langle y \rangle =$ selftype then $y = [\![y]\!]$ else $\langle y \rangle = [\![y]\!]$
	if $\langle z \rangle =$ selftype then $z = [\![z]\!]$ else $\langle z \rangle = [\![z]\!]$
$Circle$	$[center : \langle Point \rangle] \leq [\![Circle]\!]$
	$d = [center : \langle Point \rangle]$
	if $\langle Point \rangle =$ selftype then $d = [\![Point]\!]$ else $\langle Point \rangle = [\![Point]\!]$
$ColorCircle$	$[\![Circle]\!] \leq [\![ColorCircle]\!]$
	$[\![Circle]\!] = e$
	$[\![Circle]\!] \leq [center : [\![ColorPoint.move.setcolor]\!]]$
$ColorPoint.move$	$[\![ColorPoint]\!] \leq [move : \langle ColorPoint.move \rangle]$
	if $\langle ColorPoint.move \rangle =$ selftype
	then $[\![ColorPoint]\!] \leq [\![ColorPoint.move]\!]$
	else $\langle ColorPoint.move \rangle \leq [\![ColorPoint.move]\!]$
$ColorPoint.move.setcolor$	$[\![ColorPoint.move]\!] \leq [setcolor : \langle ColorPoint.move.setcolor \rangle]$
	if $\langle ColorPoint.move.setcolor \rangle =$ selftype
	then $[\![ColorPoint.move]\!] \leq [\![ColorPoint.move.setcolor]\!]$
	else $\langle ColorPoint.move.setcolor \rangle \leq [\![ColorPoint.move.setcolor]\!]$
$ColorCircle.center$	$[\![ColorCircle]\!] \leq [center : \langle ColorCircle.center \rangle]$
	if $\langle ColorCircle.center \rangle =$ selftype
	then $[\![ColorCircle]\!] \leq [\![ColorCircle.center]\!]$
	else $\langle ColorCircle.center \rangle \leq [\![ColorCircle.center]\!]$
$ColorCircle.center.move$	$[\![ColorCircle.center]\!] \leq [move : \langle ColorCircle.center.move \rangle]$
	if $\langle ColorCircle.center.move \rangle =$ selftype
	then $[\![ColorCircle.center]\!] \leq [\![ColorCircle.center.move]\!]$
	else $\langle ColorCircle.center.move \rangle \leq [\![ColorCircle.center.move]\!]$

We denote this S-system by C. Choose

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

Notice that

$$\mathcal{U} \setminus S = \{ \; \langle Point\rangle, \langle ColorCircle.center\rangle \; \} \;.$$

The ACS-system $F_S(C)$ looks as follows.

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

The constraint system $F_S(C)$ has the solution ψ where:

$$\psi(W) = \begin{cases} \mathsf{selftype} & \text{if } W \in S \\ [move : \mathsf{selftype}] & \text{if } W \in \{ \;\; x, \llbracket x\rrbracket, \llbracket Point\rrbracket, \langle Point\rangle, \\ & \qquad \llbracket ColorPoint.move.setcolor\rrbracket, \\ & \qquad \llbracket ColorCircle.center\rrbracket, \\ & \qquad \langle ColorCircle.center\rangle, \\ & \qquad \llbracket ColorCircle.center.move\rrbracket \; \} \\ [move : \mathsf{selftype} \;\; setcolor : \mathsf{selftype}] & \text{if } W \in \{ \;\; y, \llbracket y\rrbracket, z, \llbracket z\rrbracket, \llbracket ColorPoint\rrbracket, \\ & \qquad \llbracket ColorPoint.move\rrbracket \; \} \\ [center : [move : \mathsf{selftype}]] & \text{if } W \in \{ \;\; d, e, \llbracket Circle\rrbracket, \llbracket ColorCircle\rrbracket \; \} \end{cases}$$

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

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 Sofware*, 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] Kim B. Bruce. Safe type cheching 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.

[6] 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.

[7] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, 1994. Also 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.

[8] 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.

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

[10] Jens Palsberg. Efficient inference of object types. *Information and Computation*. To appear. Also in Proc. LICS'94, Ninth Annual IEEE Symposium on Logic in Computer Science, pages 186–195, Paris, France, July 1994.

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

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

Recent Publications in the BRICS Report Series

RS-95-34 **Jens Palsberg.** ***Type Inference with Selftype*****.** **June 1995. 22 pp.**

RS-95-33 **Jens Palsberg, Mitchell Wand, and Patrick O'Keefe.** ***Type Inference with Non-structural Subtyping*****.** **June 1995. 22 pp.**

RS-95-32 **Jens Palsberg.** ***Efficient Inference of Object Types*****.** **June 1995. 32 pp. To appear in** ***Information and Computation*****. Preliminary version appears in** ***Ninth Annual IEEE Symposium on Logic in Computer Science*****, LICS '94 Proceedings, pages 186–195.**

RS-95-31 **Jens Palsberg and Peter Ørbæk.** ***Trust in the λ-calculus*****.** **June 1995. 32 pp. To appear in** ***Static Analysis: 2nd International Symposium*****, SAS '95 Proceedings, 1995.**

RS-95-30 **Franck van Breugel.** ***From Branching to Linear Metric Domains (and back)*****.** **June 1995. 30 pp. Abstract appeared in Engberg, Larsen, and Mosses, editors,** ***6th Nordic Workshop on Programming Theory*****, NWPT '6 Proceedings, 1994, pages 444-447.**

RS-95-29 **Nils Klarlund.** ***An $n \log n$ Algorithm for Online BDD Refinement*****.** **May 1995. 20 pp.**

RS-95-28 **Luca Aceto and Jan Friso Groote.** ***A Complete Equational Axiomatization for MPA with String Iteration*****.** **May 1995. 39 pp.**

RS-95-27 **David Janin and Igor Walukiewicz.** ***Automata for the μ-calculus and Related Results*****.** **May 1995. 11 pp. To appear in** ***Mathematical Foundations of Computer Science: 20th Int. Symposium*****, MFCS '95 Proceedings, LNCS, 1995.**

RS-95-26 **Faith Fich and Peter Bro Miltersen.** ***Tables should be sorted (on random access machines)*****.** **May 1995. 11 pp. To appear in** ***Algorithms and Data Structures: 4th Workshop*****, WADS '95 Proceedings, LNCS, 1995.**

RS-95-25 **Søren B. Lassen.** ***Basic Action Theory*****.** **May 1995. 47 pp.**