# Object-Oriented Type Inference

**Jens Palsberg and Michael I. Schwartzbach**
palsberg@daimi.aau.dk and mis@daimi.aau.dk

Computer Science Department, Aarhus University
Ny Munkegade, DK-8000 Århus C, Denmark

## Abstract

We present a new approach to inferring types in untyped object-oriented programs with inheritance, assignments, and late binding. It guarantees that all messages are understood, annotates the program with type information, allows polymorphic methods, and can be used as the basis of an optimizing compiler. Types are finite sets of classes and subtyping is set inclusion. Using a *trace graph*, our algorithm constructs a set of conditional type constraints and computes the least solution by least fixed-point derivation.

## 1 Introduction

Untyped object-oriented languages with assignments and late binding allow rapid prototyping because classes inherit implementation and not specification. Late binding, however, can cause programs to be unreliable, unreadable, and inefficient [27]. Type inference may help solve these problems, but so far no proposed inference algorithm has been capable of checking most common, completely untyped programs [9].

We present a new type inference algorithm for a basic object-oriented language with inheritance, assignments, and late binding.

In Proc. ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) October 6-11, 1991, pages 146–161.


The algorithm guarantees that all messages are understood, annotates the program with type information, allows polymorphic methods, and can be used as the basis of an optimizing compiler. Types are finite sets of classes and subtyping is set inclusion. Given a concrete program, the algorithm constructs a finite graph of type constraints. The program is *typable* if these constraints are solvable. The algorithm then computes the least solution in worst-case exponential time. The graph contains all type information that can be derived from the program without keeping track of nil values or flow analyzing the contents of instance variables. This makes the algorithm capable of checking most common programs; in particular, it allows for polymorphic methods. The algorithm is similar to previous work on type inference [18, 14, 27, 1, 2, 19, 12, 10, 9] in using type constraints, but it differs in handling late binding by *conditional* constraints and in resolving the constraints by least fixed-point derivation rather than unification.

The example language resembles Smalltalk [8] but avoids metaclasses, blocks, and primitive methods. Instead, it provides explicit new and if-then-else expressions; classes like Natural can be programmed in the language itself.

In the following section we discuss the impacts of late binding on type inference and examine previous work. In later sections we briefly outline the example language, present the type inference algorithm, and show some examples of its capabilities.

## 2 Late Binding

Late binding means that a message send is dynamically bound to an implementation depending on the class of the receiver. This allows a form of polymorphism which is fundamental in object-oriented programming. It also, however, involves the danger that the class of the receiver does *not* implement a method for the message—the receiver may even be nil. Furthermore, late binding can make the control flow of a program hard to follow and may cause a time-consuming run-time search for an implementation.

It would significantly help an optimizing compiler if, for each message send in a program text, it could infer the following information.

- Can the receiver be nil?
- Can the receiver be an instance of a class which does not implement a method for the message?
- What are the classes of all possible non-nil receivers in any execution of the program?

Note that the available set of classes is induced by the particular program. These observations lead us to the following terminology.

**Terminology:**

**Type:** A type is a finite set of classes.

**Induced Type:** The induced type of an expression in a concrete program is the set of classes of all possible non-nil values to which it may evaluate in any execution of that particular program.

**Sound approximation:** A sound approximation of the induced type of an expression in a concrete program is a *superset* of the induced type.

Note that a sound approximation tells "the whole truth", but not always "nothing but the truth" about an induced type. Since induced types are generally uncomputable, a compiler must make do with sound approximations. An induced type is a *subtype* of any sound approximation; subtyping is set inclusion. Note also that our notion of type, which we also investigated in [22], differs from those usually used in theoretical studies of types in object-oriented programming [3, 7]; these theories have difficulties with late binding and assignments.

The goals of type inference can now be phrased as follows.

**Goals of type inference:**

**Safety guarantee:** A guarantee that any message is sent to either nil or an instance of a class which implements a method for the message; and, given that, also

**Type information:** A sound approximation of the induced type of any receiver.

Note that we ignore checking whether the receiver is nil; this is a standard data flow analysis problem which can be treated separately.

If a type inference is successful, then the program is *typable*; the error messageNotUnderstood will not occur. A compiler can use this to avoid inserting some checks in the code. Furthermore, if the type information of a receiver is a *singleton* set, then the compiler can do early binding of the message to the only possible method; it can even do in-line substitution. Similarly, if the type information is an *empty* set, then the receiver is known to always be nil. Finally, type information obtained about variables and arguments may be used to annotate the program for the benefit of the programmer.

SMALLTALK and other untyped object-oriented languages are traditionally implemented by interpreters. This is ideal for prototyping and exploratory development but often too inefficient and space demanding for real-time applications and embedded systems. What is needed is an optimizing compiler that can be used near the end of the programming phase, to get the required efficiency and a safety guarantee. A compiler which produces good code

can be tolerated even it is slow because it will be used much less often than the usual programming environment. Our type inference algorithm can be used as the basis of such an optimizing compiler. Note, though, that both the safety guarantee and the induced types are sensitive to small changes in the program. Hence, separate compilation of classes seems impossible. Typed object-oriented languages such as SIMULA [6]/BETA [15], C++ [26], and EIFFEL [17] allow separate compilation but sacrifice flexibility. The relations between types and implementation are summarized in figure 1.

| When programs are: | Their implementation is: |
|---|---|
| Untyped | Interpretation |
| Typable | Compilation |
| Typed | Separate Compilation |

Figure 1: Types and implementation.

Graver and Johnson [10, 9], in their type system for SMALLTALK, take an intermediate approach between "untyped" and "typed" in requiring the programmer to specify types for instance variables whereas types of arguments are inferred. Suzuki [27], in his pioneering work on inferring types in SMALLTALK, handles late binding by assuming that each message send may invoke all methods for that message. It turned out, however, that this yields an algorithm which is not capable of checking most common programs.

Both these approaches include a notion of *method type*. Our new type inference algorithm abandons this idea and uses instead the concept of *conditional constraints*, derived from a finite graph. Recently, Hense [11] addressed type inference for a language O'SMALL which is almost identical to our example language. He uses a radically different technique, with type schemes and unification based on work of Rémy [24] and Wand [29]. His paper lists four programs of which his algorithm can type-check only the first three. Our algorithm can type-check all four, in particular the fourth which is shown in figure 11 in appendix B. Hense uses record types which can be extendible and recursive. This seems to produce less precise typings than our approach, and it is not clear whether the typings would be useful in an optimizing compiler. One problem is that type schemes always correspond to either singletons or infinite sets of monotypes; our finite sets can be more precise. Hense's and ours approaches are similar in neither keeping track of nil values nor flow analyzing the contents of variables. We are currently investigating other possible relations.

Before going into the details of our type inference algorithm we first outline an example language on which to apply it.

# 3 The Language

Our example language resembles SMALLTALK, see figure 2.

A *program* is a set of classes followed by an expression whose value is the result of executing the program. A *class* can be defined using inheritance and contains instance variables and methods; a *method* is a message selector ($\mathsf{m}_1\_ \ldots \mathsf{m}_n\_$) with formal parameters and an expression. The language avoids metaclasses, blocks, and primitive methods. Instead, it provides explicit new and if-then-else expressions (the latter tests if the condition is non-nil). The result of a sequence is the result of the last expression in that sequence. The expression "self class new" yields an instance of the class of self. The expression "E instanceOf ClassId" yields a run-time check for class membership. If the check fails, then the expression evaluates to nil.

The SMALLTALK system is based on some primitive methods, written in assembly language. This dependency on primitives is not necessary, at least not in this theoretical study, because classes such as True, False, Natural, and List can be programmed in the language itself, as shown in appendix A.

| | | | |
|---|---|---|---|
| (Program) | P | ::= | $C_1 \ldots C_n$ E |
| (Class) | C | ::= | **class** ClassId [**inherits** ClassId]<br>**var** $Id_1 \ldots Id_k$ $M_1 \ldots M_n$<br>**end** ClassId |
| (Method) | M | ::= | **method** $m_1$ $Id_1 \ldots m_n$ $Id_n$ E |
| (Expression) | E | ::= | Id := E \| E $m_1$ $E_1 \ldots m_n$ $E_n$\| E ; E \| if E then E else E \|<br>ClassId new \| self class new \| E instanceOf ClassId \|<br>self \| super \| Id \| nil |

Figure 2: Syntax of the example language.

# 4 Type Inference

Our type inference algorithm is based on three fundamental observations.

**Observations:**

**Inheritance:** Classes inherit implementation and not specification.

**Classes:** There are finitely many classes in a program.

**Message sends:** There are finitely many syntactic message sends in a program.

The first observation leads to separate type inference for a class and its subclasses. Notionally, this is achieved by expanding all classes before doing type inference. This expansion means removing all inheritance by

- Copying the text of a class to its subclasses
- Replacing each message send to super by a message send to a renamed version of the inherited method
- Replacing each "self class new" expression by a "ClassId new" expression where ClassId is the enclosing class in the expanded program.

This idea of expansion is inspired by Graver and Johnson [10, 9]; note that the size of the expanded program is at most quadratic in the size of the original.

The second and third observation lead to a finite representation of type information about all executions of the expanded program; this representation is called the *trace graph*. From this graph a finite set of type constraints will be generated. Typability of the program is then solvability of these constraints. Appendix B contains seven example programs which illustrate different aspects of the type inference algorithm, see the overview in figure 3. The program texts are listed together with the corresponding constraints and their least solution, if it exists. Hense's program in figure 11 is the one he gives as a typical example of what he cannot type-check [11]. We invite the reader to consult the appendix while reading this section.

A trace graph contains three kinds of type information.

**Three kinds of type information:**

**Local constraints:** Generated from method bodies; contained in nodes.

**Connecting constraints:** Reflect message sends; attached to edges.

**Conditions:** Discriminate receivers; attached to edges.

| Example program in: | Illustrates: | Can we type it? |
|---|---|---|
| Figure 10 | Basic type inference | Yes |
| Figure 11 | Hense's program | Yes |
| Figure 12 | A polymorphic method | Yes |
| Figure 13 | A recursive method | Yes |
| Figure 14 | Lack of flow analysis | No |
| Figure 15 | Lack of nil detection | No |
| Figure 16 | A realistic program | Yes |

Figure 3: An overview of the example programs.

### 4.1 Trace Graph Nodes

The *nodes* of the trace graph are obtained from the various methods implemented in the program. Each method yields a number of different nodes: one for each syntactic message send with the corresponding selector. The situation is illustrated in figure 4, where we see the nodes for a method m that is implemented in each of the classes $C_1$,$C_2$,...,$C_n$. Thus, the number of nodes in the trace graph will at most be quadratic in the size of the program. There is also a single node for the main expression of the program, which we may think of as a special method without parameters.

Methods do not have types, but they can be provided with type annotations, based on the types of their formal parameters and result. A particular method implementation may be represented by several nodes in the trace graph. This enables it to be assigned several different type annotations—one for each syntactic call. This allows us effectively to obtain method polymorphism through a finite set of method "monotypes".

### 4.2 Local Constraints

Each node contains a collection of *local constraints* that the types of expressions must satisfy. For each syntactic occurrence of an expression E in the implementation of the method, we regard its type as an unknown variable [[E]]. Exact type information is, of course, uncomputable. In our approach, we will ignore the following two aspects of program executions.

**Approximations:**

**Nil values:** It does not keep track of nil values.

**Instance variables:** It does not flow analyze the contents of instance variables.

The first approximation stems from our discussion of the goals of type inference; the second corresponds to viewing an instance variable as having a single possibly large type, thus leading us to identify the type variables of different occurrences of the same instance variable. In figures 14 and 15 we present two program fragments that are typical for what we cannot type because of these approximations. In both cases the constraints demand the false inclusion {True} ⊆ {Natural}. Suzuki [27] and Hense [11] make the same approximations.

For an expression E, the local constraints are generated from all the phrases in its derivation, according to the rules in figure 5. The idea of generating constraints on type variables from the program syntax is also exploited in [28, 25].

The constraints guarantee safety; only in the cases **4)** and **8)** do the approximations manifest themselves. Notice that the constraints can all be ex-

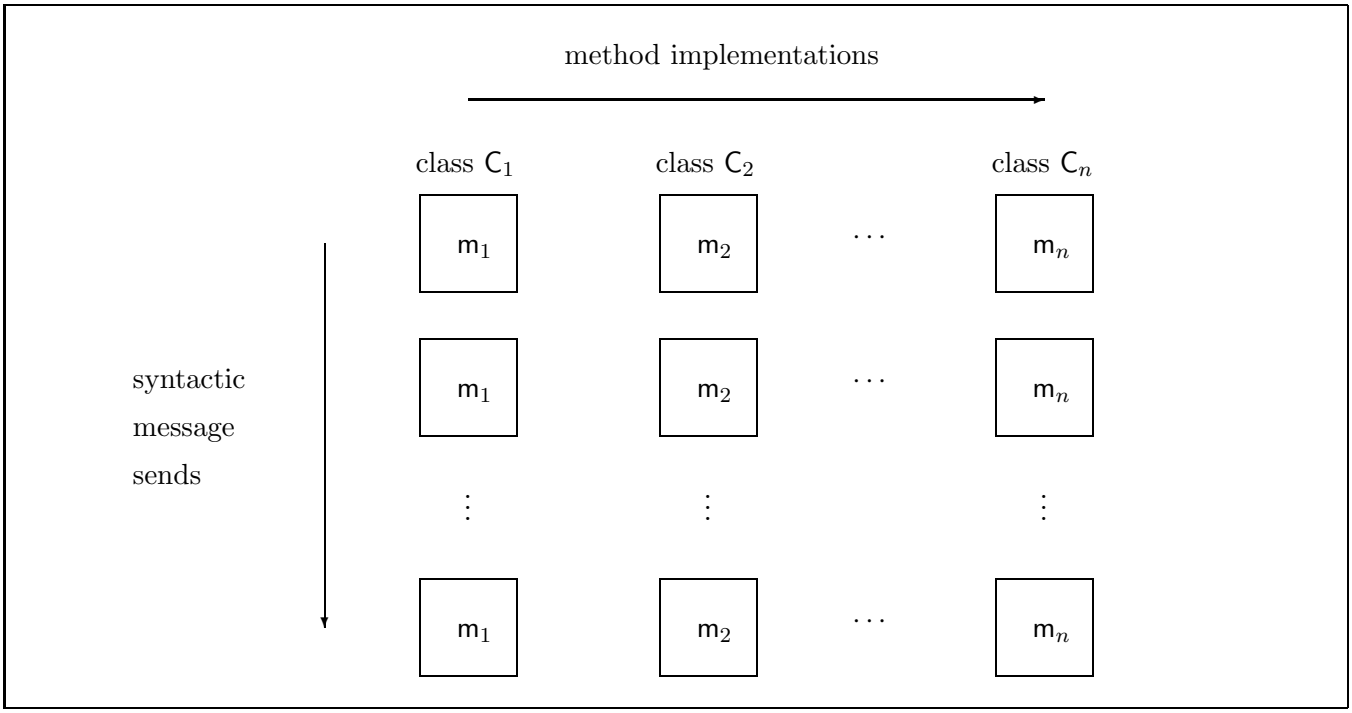



Figure 4: Trace graph nodes.

pressed as inequalities of one of the three forms: "constant $\subseteq$ variable", "variable $\subseteq$ constant", or "variable $\subseteq$ variable"; this will be exploited later.

Each different node employs unique type variables, except that the types of instance variables are common to all nodes corresponding to methods implemented in the same class. A similar idea is used by Graver and Johnson [10, 9].

### 4.3 Trace Graph Edges

The *edges* of the trace graph will reflect the possible connections between a message send and a method that may implement it. The situation is illustrated in figure 6.

If a node corresponds to a method which contains a message send of the form $\mathsf{X}$ m: $\mathsf{A}$, then we have an edge from that sender node to any other receiver node which corresponds to an implementation of a method m. We label this edge with the condition that the message send may be executed, namely $\mathsf{C} \in [\![\mathsf{X}]\!]$ where $\mathsf{C}$ is the class in which the particular method m is implemented. With the edge we associate the *connecting constraints*, which reflect the relationship between formal and actual parameters and results. This situation generalizes trivially to methods with several parameters. Note that the number of edges is again quadratic in the size of the program.

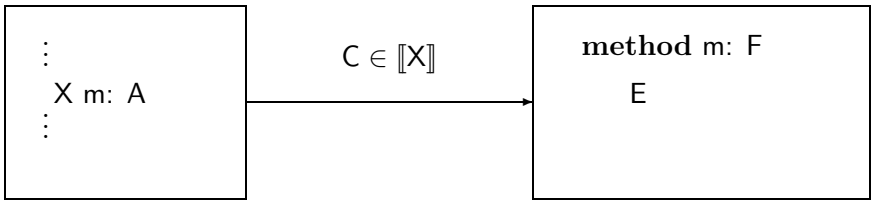
### 4.4 Global Constraints

To obtain the *global constraints* for the entire program we combine local and connecting constraints in the manner illustrated in figure 7. This produces *conditional constraints*, where the inequalities need only hold if all the conditions hold. The global constraints are simply the union of the conditional constraints generated by all paths in the graph, originating in the node corresponding to the main expression of the program. This is a finite set, because the graph is finite; as shown later in this section, the size of the constraint set may in (extreme) worst-cases become exponential.

If the set of global constraints has a solution, then

| | Expression: | Constraint: |
|---|---|---|
| **1)** | Id := E | $[\![\mathrm{Id}]\!] \supseteq [\![\mathrm{E}]\!] \ \wedge \ [\![\mathrm{Id} := \mathrm{E}]\!] = [\![\mathrm{E}]\!]$ |
| **2)** | E $\mathsf{m}_1$ $\mathrm{E}_1$ ... $\mathsf{m}_n$ $\mathrm{E}_n$ | $[\![\mathrm{E}]\!] \subseteq \{\mathsf{C} \mid \mathsf{C}$ implements $\mathsf{m}_1 \ldots \mathsf{m}_n\}$ |
| **3)** | $\mathrm{E}_1$ ; $\mathrm{E}_2$ | $[\![\mathrm{E}_1 \ ; \ \mathrm{E}_2]\!] = [\![\mathrm{E}_2]\!]$ |
| **4)** | if $\mathrm{E}_1$ then $\mathrm{E}_2$ else $\mathrm{E}_3$ | $[\![$if $\mathrm{E}_1$ then $\mathrm{E}_2$ else $\mathrm{E}_3]\!] \supseteq [\![\mathrm{E}_2]\!] \cup [\![\mathrm{E}_3]\!]$ |
| **5)** | C new | $[\![\mathsf{C}$ new$]\!] = \{\mathsf{C}\}$ |
| **6)** | E instanceOf C | $[\![$E instanceOf C$]\!] = \{\mathsf{C}\}$ |
| **7)** | self | $[\![$self$]\!] =$ {the enclosing class} |
| **8)** | Id | $[\![\mathrm{Id}]\!] = [\![\mathrm{Id}]\!]$ |
| **9)** | nil | $[\![$nil$]\!] = \{\ \}$ |

Figure 5: The local constraints.




**Connecting constraints:**

| | |
|---|---|
| $[\![\mathsf{A}]\!] = [\![\mathsf{F}]\!]$ | ("actual equals formal") |
| $[\![\mathsf{E}]\!] = [\![\mathsf{X}$ m: A$]\!]$ | ("formal result equals actual result") |

Figure 6: Trace graph edges.

this provides approximate information about the dynamic behavior of the program.

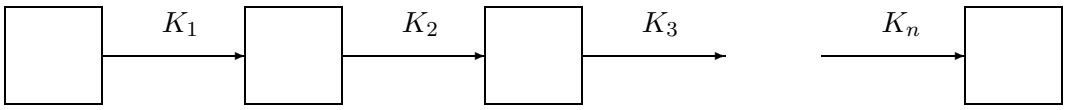
Consider any execution of the program. While observing this, we can trace the pattern of method executions in the trace graph. Let E be some expression that is evaluated at some point, let VAL(E) be its value, and let CLASS($b$) be the class of an object $b$. If $L$ is some solution to the global constraints, then the following result holds.

**Soundness Theorem:**

If VAL(E) $\neq$ nil then CLASS(VAL(E)) $\in L([\![\mathrm{E}]\!])$

It is quite easy to see that this must be true. We sketch a proof by induction in the number of message sends performed during the trace. If this is zero, then we rely on the local constraints alone; given a dynamic semantics [4, 5, 23, 13] one can easily verify that their satisfaction implies the above property. If we extend a trace with a message send X m: A implemented by a method in a class C, then we can inductively assume that $\mathsf{C} \in L([\![\mathsf{X}]\!])$. But this implies that the local constraints in the node corresponding to the invoked method must hold, since all their conditions now hold and $L$ is a solution. Since the relationship between actual and formal parameters and results is soundly represented by the connecting constraints, which also must hold, the result follows.

Note that an expression E occurring in a method that appears $k$ times in the trace graph has $k$ type variables $[\![\mathrm{E}]\!]_1, [\![\mathrm{E}]\!]_2, \ldots, [\![\mathrm{E}]\!]_k$ in the global constraints. A sound approximation to the induced




**Conditional constraint:**

$K_1, K_2, K_3, \ldots, K_n \Rightarrow L \cup C$

$L$ = local constraints of the final node

$C$ = connecting constraints of the final edge

Figure 7: Conditional constraints from a path.

type of E is obtained as

$$\bigcup_i L([\![\mathrm{E}]\!]_i)$$

Appendix C gives an efficient algorithm to compute the smallest solution of the extracted constraints, or to decide that none exists. The algorithm is at worst quadratic in the size of the constraint set.

The complete type inference algorithm is summarized in figure 8.

### 4.5 Type Annotations

Finally, we will consider how a solution $L$ of the type constraints can produce a *type annotation* of the program. Such annotations could be provided for the benefit of the programmer.

An instance variable $\mathsf{x}$ has only a single associated type variable. The type annotation is simply $L([\![\mathsf{x}]\!])$. The programmer then knows an upper bound of the set of classes whose instances may reside in $\mathsf{x}$.

A method has finitely many type annotations, each of which is obtained from a corresponding node in the trace graph. If the method, implemented in the class $\mathsf{C}$, is

**Input:** A program in the example language.

**Output:** Either: a safety guarantee and type information about all expressions; or: "unable to type the program".

**1)** Expand all classes.

**2)** Construct the trace graph of the expanded program.

**3)** Extract a set of type constraints from the trace graph.

**4)** Compute the least solution of the set of type constraints. If such a solution exists, then output it as the wanted type information, together with a safety guarantee; otherwise, output "unable to type the program".

Figure 8: Summary of the type inference algorithm.

**method** $\mathsf{m}_1$: $\mathsf{F}_1$ $\mathsf{m}_2$: $\mathsf{F}_2$ $\ldots$ $\mathsf{m}_n$: $\mathsf{F}_n$
$\mathsf{E}$

then each type annotation is of the form

$$\{\mathsf{C}\} \times L([\![\mathsf{F}_1]\!]) \times \cdots \times L([\![\mathsf{F}_n]\!]) \to L([\![\mathsf{E}]\!])$$

The programmer then knows the various manners in which this method may be used.

A constraint solution contains more type informa-

tion about methods than the method types used by Suzuki. Consider for example the polymorphic identity function in figure 12. Our technique yields both of the method type annotations

$$\mathsf{id} : \{\mathsf{C}\} \times \{\mathsf{True}\} \to \{\mathsf{True}\}$$
$$\mathsf{id} : \{\mathsf{C}\} \times \{\mathsf{Natural}\} \to \{\mathsf{Natural}\}$$

whereas the method type using Suzuki's framework is

$$\mathsf{id} : \{\mathsf{C}\} \times \{\mathsf{True}, \mathsf{Natural}\} \to \{\mathsf{True}, \mathsf{Natural}\}$$

which would allow neither the succ nor the isTrue message send, and, hence, would lead to rejection of the program.

### 4.6 An Exponential Worst-Case

The examples in appendix B show several cases where the constraint set is quite small, in fact linear in the size of the program. While this will often be the situation, the theoretical worst-case allows the constraint set to become exponential in the size of the program. The running time of the inference algorithm depends primarily on the topology of the trace graph.

In figure 9 is shown a program and a sketch of its trace graph. The induced constraint set will be exponential since the graph has exponentially many different paths. Among the constraints will be a family whose conditions are similar to the words of the regular language

$$(\mathsf{CCC} + \mathsf{DCC})^{\frac{n}{3}}$$

the size of which is clearly exponential in $n$.

Note that this situation is similar to that of type inference in ML, which is also worst-case exponential but very useful in practice. The above scenario is in fact not unlike the one presented in [16] to illustrate exponential running times in ML. Another similarity is that both algorithms generate a potentially exponential constraint set that is always solved in polynomial time.

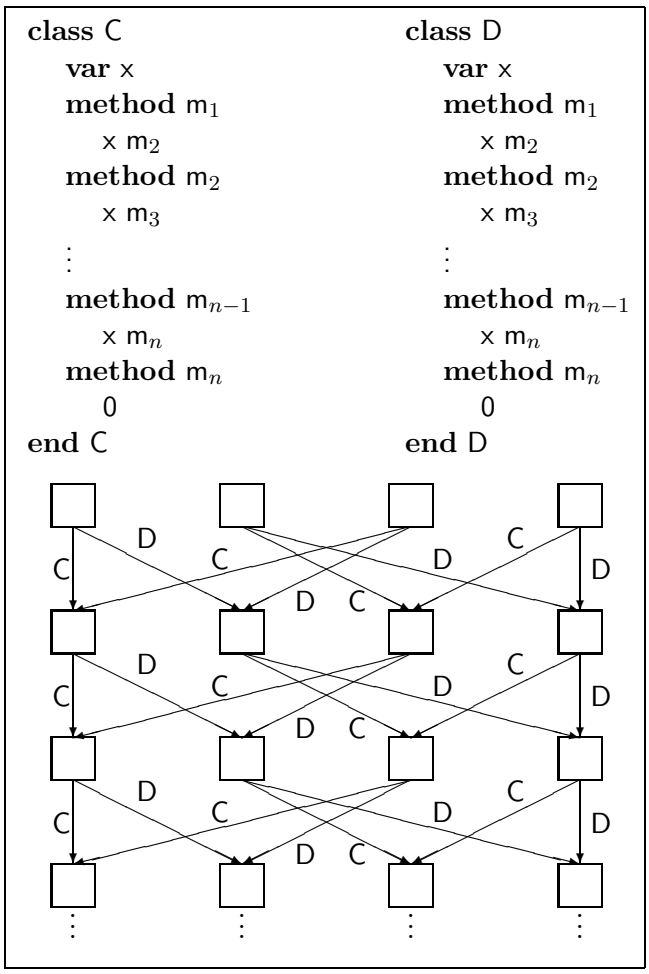



Figure 9: A worst-case program.

## 5 Conclusion

Our type inference algorithm is sound and can handle most common programs. It is also conceptually simple: a set of uniform type constraints is constructed and solved by fixed-point derivation. It can be further improved by an orthogonal effort in data flow analysis.

The underlying type system is simple: types are finite sets of classes and subtyping is set inclusion.

An implementation of the type inference algorithm is currently being undertaken. Future work includes extending this into an optimizing compiler. The inference algorithm should be easy to modify to work for full SMALLTALK, because metaclasses

are simply classes, blocks can be treated as objects with a single method, and primitive methods can be handled by stating the constraints that the machine code must satisfy. Another challenge is to extend the algorithm to produce type annotations together with type substitution, see [20, 21, 22].

## Appendix A: Basic classes

```
class Object
end Object
```

```
class True
    method isTrue
        Object new
end True
```

```
class False
    method isTrue
        nil
end False
```

Henceforth, we abbreviate "True **new**" as "true", and "False **new**" as "false".

```
class Natural
    var rep
    method isZero
        if rep then false else true
    method succ
        (Natural new) update: self
    method update: x
        rep := x; self
    method pred
        if (self isZero) isTrue then self else rep
    method less: i
        if (i isZero) isTrue
        then false
        else if (self isZero) isTrue then true
        else (self pred) less: (i pred)
end Natural
```

Henceforth, we abbreviate "Natural **new**" as "0", and, recursively, "$n$ succ" as "$n+1$".

```
class List
    var head, tail
    method setHead: h setTail: t
        head := h; tail := t
    method cons: x
        (self class new) setHead: x setTail: self
    method isEmpty
        if head then false else true
    method car
        head
    method cdr
        tail
    method append: aList
        if (self isEmpty) isTrue
        then aList
        else (tail append: aList) cons: head
    method insert: x
        if (self isEmpty) isTrue
        then self cons: x
        else
            if (head less: x) isTrue
            then self cons: x
            else (tail insert: x) cons: head
    method sort
        if (self isEmpty) isTrue then self
        else (tail sort) insert: head
    method merge: aList
        if (self isEmpty) isTrue
        then aList
        else
            if (head less: (aList car)) isTrue
            then (tail merge: aList) cons: head
            else (self merge: (aList cdr)) cons: (aList car)
end List
```

```
class Comparable
    var key
    method getKey
        key
    method setKey: k
        key := k
    method less: c
        key less: (c getKey)
end Comparable
```

## Appendix B: Example Programs

**class** A
  **method** f
    7
**end** A
**class** B
  **method** f
    true
**end** B
x := A **new**; (x f) succ

**Constraints:**

$[\![\text{A } \textbf{new}]\!] = \{\text{A}\}$
$[\![\text{x}]\!] \supseteq [\![\text{A } \textbf{new}]\!]$
$[\![\text{x} := \text{A } \textbf{new}]\!] = [\![\text{A } \textbf{new}]\!]$
$[\![\text{x}]\!] \subseteq \{\text{A}, \text{B}\}$
$\text{A} \in [\![\text{x}]\!] \Rightarrow [\![\text{x f}]\!] = [\![7]\!]$
$\text{A} \in [\![\text{x}]\!] \Rightarrow [\![7]\!] = \{\text{Natural}\}$
$\text{B} \in [\![\text{x}]\!] \Rightarrow [\![\text{x f}]\!] = [\![\text{true}]\!]$
$\text{B} \in [\![\text{x}]\!] \Rightarrow [\![\text{true}]\!] = \{\text{True}\}$
$[\![\text{x f}]\!] \subseteq \{\text{Natural}\}$
$\text{Natural} \in [\![\text{x f}]\!] \Rightarrow [\![(\text{x f}) \text{ succ}]\!] = \{\text{Natural}\}$
$[\![\text{x} := \text{A } \textbf{new}; (\text{x f}) \text{ succ}]\!] = [\![(\text{x f}) \text{ succ}]\!]$

**Smallest Solution:**

$[\![\text{x}]\!] = [\![\text{A } \textbf{new}]\!] = [\![\text{x} := \text{A } \textbf{new}]\!] = \{\text{A}\}$
$[\![\text{x f}]\!] = [\![(\text{x f}) \text{ succ}]\!] =$
$[\![\text{x} := \text{A } \textbf{new}; (\text{x f}) \text{ succ}]\!] = [\![7]\!] = \{\text{Natural}\}$
$[\![\text{true}]\!] = \{\text{True}\}$

**Trace graph sketch:**

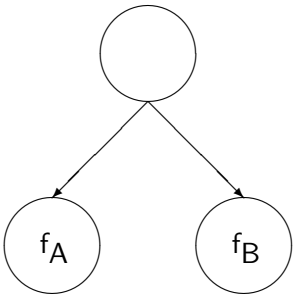



Figure 10: Conditions at work.

**class** A
  **method** m
    0
**end** A
**class** B **inherits** A
  **method** n
    0
**end** B
a := A **new**;
b := B **new**;
a := b;
a m

**Constraints:**

$[\![\text{A } \textbf{new}]\!] = \{\text{A}\}$
$[\![\text{a}]\!] \supseteq [\![\text{A } \textbf{new}]\!]$
$[\![\text{B } \textbf{new}]\!] = \{\text{B}\}$
$[\![\text{b}]\!] \supseteq [\![\text{B } \textbf{new}]\!]$
$[\![\text{a}]\!] \supseteq [\![\text{b}]\!]$
$[\![\text{a}]\!] \subseteq \{\text{A,B}\}$
$\text{A} \in [\![\text{a}]\!] \Rightarrow [\![\text{a m}]\!] = [\![0]\!]$
$\text{B} \in [\![\text{a}]\!] \Rightarrow [\![\text{a m}]\!] = [\![0]\!]$
$[\![0]\!] = \{\text{Natural}\}$
$\vdots$

**Smallest Solution:**

$[\![\text{a}]\!] = \{\text{A,B}\}$
$[\![\text{b}]\!] = \{\text{B}\}$
$[\![\text{a m}]\!] = \{\text{Natural}\}$
$[\![\text{A } \textbf{new}]\!] = \{\text{A}\}$
$[\![\text{B } \textbf{new}]\!] = \{\text{B}\}$
$\vdots$

**Trace graph sketch:**

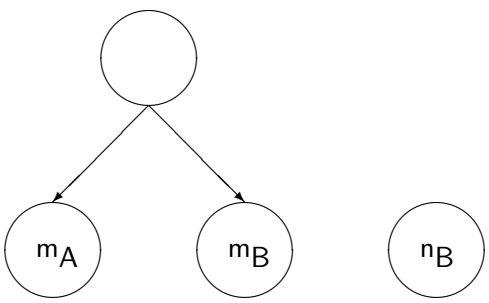



Figure 11: Hense's program.

**class** C
  **method** id: x
    x
**end** C
((C **new**) id: 7) succ;
((C **new**) id: true) isTrue

**Constraints:**

$[\![\text{C } \mathbf{new}]\!]_1 = \{\text{C}\}$
$[\![\text{C } \mathbf{new}]\!]_1 \subseteq \{\text{C}\}$
$\text{C} \in [\![\text{C } \mathbf{new}]\!]_1 \Rightarrow [\![7]\!] = [\![\text{x}]\!]_1$
$\text{C} \in [\![\text{C } \mathbf{new}]\!]_1 \Rightarrow [\![\text{x}]\!]_1 = [\![(\text{C } \mathbf{new}) \text{ id: } 7]\!]$
$[\![7]\!] = \{\text{Natural}\}$
$[\![(\text{C } \mathbf{new}) \text{ id: } 7]\!] \subseteq \{\text{Natural}\}$
$\text{Natural} \in [\![(\text{C } \mathbf{new}) \text{ id: } 7]\!] \Rightarrow \{\text{Natural}\} = [\![((\text{C } \mathbf{new}) \text{ id: } 7) \text{ succ}]\!]$
$[\![\text{C } \mathbf{new}]\!]_2 = \{\text{C}\}$
$[\![\text{C } \mathbf{new}]\!]_2 \subseteq \{\text{C}\}$
$\text{C} \in [\![\text{C } \mathbf{new}]\!]_2 \Rightarrow [\![\text{true}]\!] = [\![\text{x}]\!]_2$
$\text{C} \in [\![\text{C } \mathbf{new}]\!]_2 \Rightarrow [\![\text{x}]\!]_2 = [\![(\text{C } \mathbf{new}) \text{ id: true}]\!]$
$[\![\text{true}]\!] = \{\text{True}\}$
$[\![(\text{C } \mathbf{new}) \text{ id: true}]\!] \subseteq \{\text{True,False}\}$
$\text{True} \in [\![(\text{C } \mathbf{new}) \text{ id: true}]\!] \Rightarrow \{\text{Object}\} = [\![((\text{C } \mathbf{new}) \text{ id: true}) \text{ isTrue}]\!]$
$\text{False} \in [\![(\text{C } \mathbf{new}) \text{ id: true}]\!] \Rightarrow \{\} = [\![((\text{C } \mathbf{new}) \text{ id: true}) \text{ isTrue}]\!]$

**Smallest Solution:**

$[\![\text{C } \mathbf{new}]\!]_1 = [\![\text{C } \mathbf{new}]\!]_2 = \{\text{C}\}$
$[\![7]\!] = [\![\text{x}]\!]_1 = [\![(\text{C } \mathbf{new}) \text{ id: } 7]\!] = [\![((\text{C } \mathbf{new}) \text{ id: } 7) \text{ succ}]\!] = \{\text{Natural}\}$
$[\![\text{true}]\!] = [\![\text{x}]\!]_2 = [\![(\text{C } \mathbf{new}) \text{ id: true}]\!] = \{\text{True}\}$
$[\![((\text{C } \mathbf{new}) \text{ id: true}) \text{ isTrue}]\!] = \{\text{Object}\}$

**Trace graph sketch:**

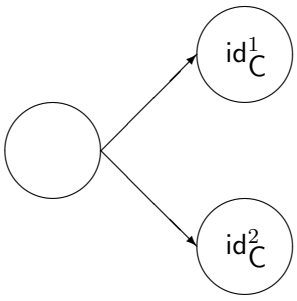



Figure 12: A polymorphic method.

```
class D
   method f: x
      if x then self f: x else nil
end D
(D new) f: nil
```

**Constraints:**

$[\![\text{D new}]\!] = \{\text{D}\}$
$[\![\text{D new}]\!] \subseteq \{\text{D}\}$
$\text{D} \in [\![\text{D new}]\!] \Rightarrow [\![\text{nil}]\!] = [\![\text{x}]\!]_1$
$\text{D} \in [\![\text{D new}]\!] \Rightarrow [\![\textbf{if x then self f: x else nil}]\!]_1 = [\![(\text{D new})\ \text{f: nil}]\!]$
$\text{D} \in [\![\text{D new}]\!] \Rightarrow [\![\textbf{if x then self f: x else nil}]\!]_1 \supseteq [\![\text{self f: x}]\!]_1 \cup [\![\text{nil}]\!]_1$
$\text{D} \in [\![\text{D new}]\!] \Rightarrow [\![\text{nil}]\!]_1 = \{\}$
$\text{D} \in [\![\text{D new}]\!] \Rightarrow [\![\text{self}]\!]_1 = \{\text{D}\}$
$\text{D} \in [\![\text{D new}]\!] \Rightarrow [\![\text{self}]\!]_1 \subseteq \{\text{D}\}$
$\text{D} \in [\![\text{D new}]\!], \text{D} \in [\![\text{self}]\!]_1 \Rightarrow [\![\text{x}]\!]_1 = [\![\text{x}]\!]_2$
$\text{D} \in [\![\text{D new}]\!], \text{D} \in [\![\text{self}]\!]_1 \Rightarrow [\![\textbf{if x then self f: x else nil}]\!]_2 = [\![\text{self f: x}]\!]_1$
$\text{D} \in [\![\text{D new}]\!], \text{D} \in [\![\text{self}]\!]_1 \Rightarrow [\![\textbf{if x then self f: x else nil}]\!]_2 \supseteq [\![\text{self f: x}]\!]_2 \cup [\![\text{nil}]\!]_2$
$\text{D} \in [\![\text{D new}]\!], \text{D} \in [\![\text{self}]\!]_1 \Rightarrow [\![\text{nil}]\!]_2 = \{\}$
$\text{D} \in [\![\text{D new}]\!], \text{D} \in [\![\text{self}]\!]_1 \Rightarrow [\![\text{self}]\!]_2 = \{\text{D}\}$
$\text{D} \in [\![\text{D new}]\!], \text{D} \in [\![\text{self}]\!]_1 \Rightarrow [\![\text{self}]\!]_2 \subseteq \{\text{D}\}$
$\text{D} \in [\![\text{D new}]\!], \text{D} \in [\![\text{self}]\!]_1, \text{D} \in [\![\text{self}]\!]_2 \Rightarrow [\![\text{x}]\!]_2 = [\![\text{x}]\!]_2$
$\text{D} \in [\![\text{D new}]\!], \text{D} \in [\![\text{self}]\!]_1, \text{D} \in [\![\text{self}]\!]_2 \Rightarrow [\![\textbf{if x then self f: x else nil}]\!]_2 = [\![\text{self f: x}]\!]_2$
$[\![\text{nil}]\!] = \{\}$

**Smallest Solution:**

$[\![\text{D new}]\!] = [\![\text{self}]\!]_1 = [\![\text{self}]\!]_2 = \{\text{D}\}$
$[\![\text{nil}]\!] = [\![\text{x}]\!]_1 = [\![\text{nil}]\!]_1 = [\![\textbf{if x then self f: x else nil}]\!]_1 = [\![\text{self f: x}]\!]_1 =$
$[\![(\text{D new})\ \text{f: nil}]\!]_1 = [\![\text{x}]\!]_2 = [\![\text{nil}]\!]_2 = [\![\textbf{if x then self f: x else nil}]\!]_2 =$
$[\![\text{self f: x}]\!]_2 = [\![(\text{D new})\ \text{f: nil}]\!]_2 = \{\}$

**Trace graph sketch:**

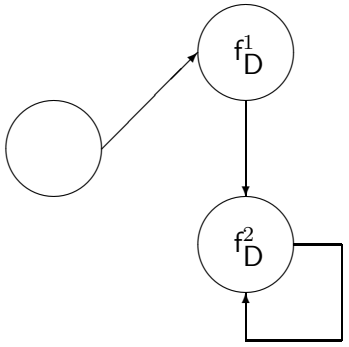



Figure 13: A recursive method.

```
x := 7;
x succ;
x := true;
x isTrue
```
**Constraints:**

$[\![x]\!] \supseteq [\![7]\!]$
$[\![7]\!] = \{\text{Natural}\}$
$[\![x]\!] \subseteq \{\text{Natural}\}$
$[\![x]\!] \supseteq [\![\text{true}]\!]$
$[\![\text{true}]\!] = \{\text{True}\}$
$[\![x]\!] \subseteq \{\text{True,False}\}$
$\vdots$

Figure 14: A safe program rejected.

```
(if nil then true else 7) succ
```
**Constraints:**

$[\![\textbf{if nil then true else 7}]\!] \subseteq \{\text{Natural}\}$
$[\![\textbf{if nil then true else 7}]\!] \supseteq [\![\text{true}]\!] \cup [\![7]\!]$
$[\![\text{true}]\!] = \{\text{True}\}$
$[\![7]\!] = \{\text{Natural}\}$
$\vdots$

Figure 15: Another safe program rejected.

```
class Student inherits Comparable
   ...
end Student
class ComparableList inherits List
    method studentCount
        if (self isEmpty) isTrue
        then 0
        else
            if (self car) instanceOf Student
            then ((self cdr) studentCount) succ
            else (self cdr) studentCount
end ComparableList
```

Figure 16: An example program.

# Appendix C: Solving Systems of Conditional Inequalities

This appendix shows how to solve a finite system of conditional inequalities in quadratic time.

**Definition C.1:** A *CI-system* consists of

- a finite set $\mathcal{A}$ of *atoms*.
- a finite set $\{\alpha_i\}$ of *variables*.
- a finite set of *conditional inequalities* of the form
$$C_1, C_2, \ldots, C_k \Rightarrow Q$$
Each $C_i$ is a *condition* of the form $a \in \alpha_j$, where $a \in \mathcal{A}$ is an atom, and $Q$ is an *inequality* of one of the following forms
$$\begin{array}{rcl} A & \subseteq & \alpha_i \\ \alpha_i & \subseteq & A \\ \alpha_i & \subseteq & \alpha_j \end{array}$$
where $A \subseteq \mathcal{A}$ is a set of atoms.

A *solution* $L$ of the system assigns to each variable $\alpha_i$ a set $L(\alpha_i) \subseteq \mathcal{A}$ such that all the conditional inequalities are satisfied. □

In our application, $\mathcal{A}$ models the set of classes occurring in a concrete program.

**Lemma C.2:** Solutions are closed under intersection. Hence, if a CI-system has solutions, then it has a unique minimal one.

**Proof:** Consider any conditional inequality of the form $C_1, C_2, \ldots, C_k \Rightarrow Q$, and let $\{L_i\}$ be all solutions. We shall show that $\cap_i L_i$ is a solution. If a condition $a \in \cap_i L_i(\alpha_j)$ is true, then so is all of $a \in L_i(\alpha_j)$. Hence, if all the conditions of $Q$ are true in $\cap_i L_i$, then they are true in each $L_i$; furthermore, since they are solutions, $Q$ is also true in each $L_i$. Since, in general, $A_k \subseteq B_k$ implies $\cap_k A_k \subseteq \cap_k B_k$, it follows that $\cap_i L_i$ is a solution. Hence, if there are any solutions, then $\cap_i L_i$ is the unique smallest one. □

**Definition C.3:** Let $\mathcal{C}$ be a CI-system with atoms

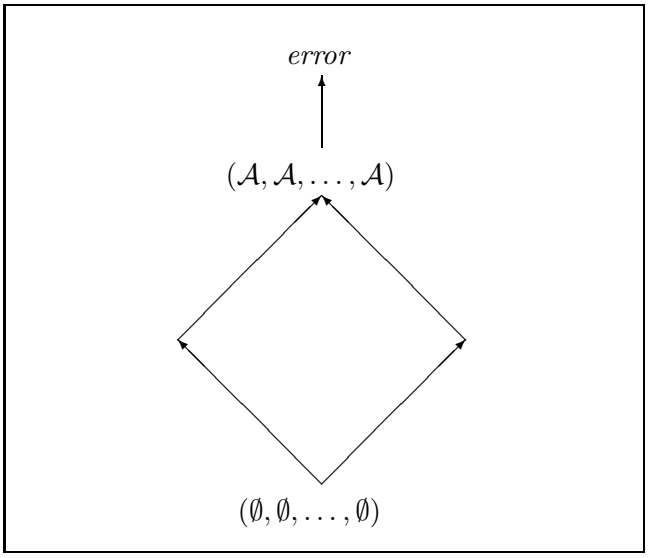



Figure 17: The lattice of assignments.

$\mathcal{A}$ and $n$ distinct variables. An *assignment* is an element of $(2^{\mathcal{A}})^n \cup \{error\}$ ordered as a lattice, see figure 17. If different from *error*, then it assigns a set of atoms to each variable. If $V$ is an assignment, then $\tilde{\mathcal{C}}(V)$ is a new assignment, defined as follows. If $V = error$, then $\tilde{\mathcal{C}}(V) = error$. An inequality is *enabled* if all of its conditions are true under $V$. If for any enabled inequality of the form $\alpha_i \subseteq A$ we do *not* have $V(\alpha_i) \subseteq A$, then $\tilde{\mathcal{C}}(V) = error$; otherwise, $\tilde{\mathcal{C}}(V)$ is the smallest pointwise extension of $V$ such that

- for every enabled inequality of the form $A \subseteq \alpha_j$ we have $A \subseteq \tilde{\mathcal{C}}(V)(\alpha_j)$.
- for every enabled inequality of the form $\alpha_i \subseteq \alpha_j$ we have $V(\alpha_i) \subseteq \tilde{\mathcal{C}}(V)(\alpha_j)$.

Clearly, $\tilde{\mathcal{C}}$ is monotonic in the above lattice. □

**Lemma C.4:** An assignment $L \neq error$ is a solution of a CI-system $\mathcal{C}$ *iff* $L = \tilde{\mathcal{C}}(L)$. If $\mathcal{C}$ has no solutions, then *error* is the smallest fixed-point of $\tilde{\mathcal{C}}$.

**Proof:** If $L$ is a solution of $\mathcal{C}$, then clearly $\tilde{\mathcal{C}}$ will not equal *error* and cannot extend $L$; hence, $L$ is a fixed-point. Conversely, if $L$ is a fixed-point of $\tilde{\mathcal{C}}$, then all the enabled inequalities must hold. If there are no solutions, then there can be no fixed-point below *error*. Since *error* is by definition a fixed-point, the result follows. □

This means that to find the smallest solution, or to decide that none exists, we need only compute the least fixed-point of $\tilde{\mathcal{C}}$.

**Lemma C.5:** For any CI-system $\mathcal{C}$, the least fixed-point of $\tilde{\mathcal{C}}$ is equal to

$$\lim_{k \to \infty} \tilde{\mathcal{C}}^k(\emptyset, \emptyset, \ldots, \emptyset)$$

**Proof:** This is a standard result about monotonic functions on complete lattices. □

**Lemma C.6:** Let $n$ be the number of *different* conditions in a CI-system $\mathcal{C}$. Then

$$\lim_{k \to \infty} \tilde{\mathcal{C}}^k(\emptyset, \emptyset, \ldots, \emptyset) = \tilde{\mathcal{C}}^{n+1}(\emptyset, \emptyset, \ldots, \emptyset)$$

**Proof:** When no more conditions are enabled, then the fixed-point is obtained by a single application. Once a condition is enabled in an assignment, it will remain enabled in all larger assignments. It follows that after $n$ iterations no new conditions can be enabled; hence, the fixed-point is obtained in at most $n + 1$ iterations. □

**Lemma C.7:** The smallest solution to any CI-system, or the decision that none exists, can be obtained in quadratic time.

**Proof:** This follows from the previous lemmas. □

# References

[1] Alan H. Borning and Daniel H. H. Ingalls. A type declaration and inference system for Smalltalk. In *Ninth Symposium on Principles of Programming Languages*, pages 133–141, 1982.

[2] Luca Cardelli. A semantics of multiple inheritance. In Gilles Kahn, David MacQueen, and Gordon Plotkin, editors, *Semantics of Data Types*, pages 51–68. Springer-Verlag (*LNCS* 173), 1984.

[3] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[4] 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.

[5] William R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[6] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. Simula 67 common base language. Technical report, Norwegian Computing Center, Oslo, Norway, 1968.

[7] Scott Danforth and Chris Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–72, March 1988.

[8] Adele Goldberg and David Robson. *Smalltalk-80—The Language and its Implementation*. Addison-Wesley, 1983.

[9] Justin O. Graver and Ralph E. Johnson. A type system for Smalltalk. In *Seventeenth Symposium on Principles of Programming Languages*, pages 136–150, 1990.

[10] Justin Owen Graver. *Type-Checking and Type-Inference for Object-Oriented Programming Languages*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, August 1989. UIUCD-R-89-1539.

[11] Andreas V. Hense. Polymorphic type inference for a simple object oriented programming language with state. Technical Report No. A 20/90, Fachbericht 14, Universität des Saarlandes, December 1990.

[12] Ralph E. Johnson. Type-checking Smalltalk. In *Proc. OOPSLA'86, Object-Oriented Programming Systems, Languages and Applications*, pages 315–321. Sigplan Notices, 21(11), November 1986.

[13] Samuel Kamin. Inheritance in Smalltalk–80: A denotational definition. In *Fifteenth Symposium on Principles of Programming Languages*, pages 80–87, 1988.

[14] Marc A. Kaplan and Jeffrey D. Ullman. A general scheme for the automatic inference of variable types. In *Fifth Symposium on Principles of Programming Languages*, pages 60–75, 1978.

[15] 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.

[16] Harry G. Mairson. Decidability of ML typing is complete for deterministic exponential time. In *Seventeenth Symposium on Principles of Programming Languages*, pages 382–401, 1990.

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

[18] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[19] Prateek Mishra and Uday S. Reddy. Declaration-free type checking. In *Twelfth Symposium on Principles of Programming Languages*, pages 7–21, 1985.

[20] Jens Palsberg and Michael I. Schwartzbach. Type substitution for object-oriented programming. In *Proc. OOPSLA/ECOOP'90, ACM SIGPLAN Fifth Annual Conference on Object-Oriented Programming Systems, Languages and Applications; European Conference on Object-Oriented Programming*, pages 151–160, Ottawa, Canada, October 1990.

[21] Jens Palsberg and Michael I. Schwartzbach. What is type-safe code reuse? In *Proc. ECOOP'91, Fifth European Conference on Object-Oriented Programming*, pages 325–341. Springer-Verlag (*LNCS* 512), Geneva, Switzerland, July 1991.

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

[23] Uday S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 289–297, 1988.

[24] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Sixteenth Symposium on Principles of Programming Languages*, pages 77–88, 1989.

[25] Michael I. Schwartzbach. Type inference with inequalities. In *Proc. TAPSOFT'91*, pages 441–455. Springer-Verlag (*LNCS* 493), 1991.

[26] Bjarne Stroustrup. *The* C++ *Programming Language*. Addison-Wesley, 1986.

[27] Norihisa Suzuki. Inferring types in Smalltalk. In *Eighth Symposium on Principles of Programming Languages*, pages 187–199, 1981.

[28] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamentae Informaticae*, X:115–122, 1987.

[29] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *LICS'89, Fourth Annual Symposium on Logic in Computer Science*, pages 92–97, 1989.