To appear in ECOOP '93 Proceedings, Springer Verlag Lecture Notes on Computer Science.

Type Inference of SELF

Analysis of Objects with Dynamic and Multiple Inheritance

Ole Agesen*[1], Jens Palsberg[2], Michael I. Schwartzbach**[2]

[1] Dept. of Computer Science, Stanford University, Stanford, CA 94305, USA,
agesen@self.stanford.edu
[2] Computer Science Dept., Aarhus University, Ny Munkegade, DK-8000 Århus C,
Denmark, {palsberg,mis}@daimi.aau.dk

Abstract. We have designed and implemented a type inference algorithm for the full SELF language. The algorithm can guarantee the safety and disambiguity of message sends, and provide useful information for browsers and optimizing compilers.

SELF features objects with dynamic inheritance. This construct has until now been considered incompatible with type inference because it allows the inheritance graph to change dynamically. Our algorithm handles this by deriving and solving type constraints that simultaneously define supersets of both the possible values of expressions and of the possible inheritance graphs. The apparent circularity is resolved by computing a global fixed-point, in polynomial time.

The algorithm has been implemented and can successfully handle the SELF benchmark programs, which exist in the "standard SELF world" of more than 40,000 lines of code.

Keywords: Languages and their implementation, tools and environments.

1 Introduction

The choice between static and dynamic typing involves a choice between safety and flexibility. The flexibility offered by dynamically typed object-oriented languages is useful in exploratory programming but may also be a hindrance to safety checking and optimization when delivering products.

Henry Lieberman [8] and Alan Borning [2] developed the notion of object-oriented languages based on prototypes. The absence of classes and types in these

* Generously supported by National Science Foundation Presidential Young Investigator Grant #CCR-8657631, by Sun Microsystems, IBM, Apple Computer, Cray Laboratories, Tandem Computers, NCR, Texas Instruments, DEC, by a research fellowship from the Natural Science Faculty of Aarhus University, and by the Danish Research Academy.

** Partially supported by the Danish Research Council, DART Project (5.21.08.03).

languages yields a considerable flexibility which may be significantly increased by the notions of dynamic and multiple inheritance. These language constructs, however, make safety checking more difficult than for class-based languages.

This paper presents a type inference algorithm for the SELF language [13]. SELF is a prototype-based dynamically typed object-oriented language featuring dynamic and multiple inheritance. Our algorithm can guarantee the safety and disambiguity of message sends, and provide useful information for tools such as browsers and optimizing compilers. Although we focus on SELF our work applies to other languages as well.

Our approach to type inference is based on constraints, like in our previous papers on an idealized subset of SMALLTALK. In [10] we defined the basic type inference framework, and in [9] we demonstrated an efficient implementation.

Dynamic inheritance has until now been considered incompatible with type inference because it allows the inheritance graph to change dynamically. Our algorithm handles this by deriving and solving type constraints that simultaneously define supersets of both the possible values of expressions and of the possible inheritance graphs. The apparent circularity is resolved by computing a global fixed-point, in polynomial time.

In most other type inference algorithms for object-oriented languages, for example that of Graver and Johnson [7, 6] and our own [10, 9], inheritance is expanded away rather than dealt with directly. Using prototypes, however, expansion is impossible, since a parent may have an independent state. This paper demonstrates how to handle inheritance without expansion. It also shows how to handle blocks with non-local return.

In the following Section we give an overview of the required type constraints. In Section 3 we present example runs of our implementation, and in Section 4 we discuss details of our algorithm and implementation. Finally, in Section 5 we summarize our results and conclusions.

2 Type Constraints

SELF [13] resembles SMALLTALK [5] on the surface. However, there are major differences that make SELF a particularly interesting language to analyze: SELF has no classes, instantiation is object cloning, inheritance is between objects having independent state and identity, and the inheritance may be both dynamic and multiple. Dynamic inheritance allows the inheritance graph to change during program execution. The dynamic nature of SELF makes it harder to obtain nontrivial type information for SELF programs than for, say, SMALLTALK programs. It also makes such information immediately useful in browsers and optimizing compilers.

Below we describe our approach to type inference for SELF. We will use the SELF-terminology without explanation.

2.1 Constraint-Based Analysis

Our approach to type inference is based on constraints, like in our previous papers on an idealized subset of SMALLTALK. The main idea in constraint-based analysis [15, 12, 11] is as follows. First, define type variables for the unknown type information. Second, derive constraints on these variables from the given program. Third, solve the resulting constraints to obtain the desired information.

The algorithms in [10, 9] had safety checking as an integral part of the type constraints. As a result, type information could not be inferred for incorrect programs that could provoke a msgNotUnderstood error. The approach in this paper is more liberal, so type information can be computed for *all* programs. This may be useful during program development and debugging, since information about a partially correct program can be obtained. When a guarantee against msgNotUnderstood or ambiguousSend errors is desired, it can be provided by a tool that inspects the computed type information (although it is straightforward to implement these tools, we haven't done it yet).

2.2 Types and Type Variables

Any given SELF program contains a fixed number of *objects* (some of which are "block objects") and *methods* (which are either "normal methods" or "block methods"). We introduce a unique *token* for each of these: $\omega_1, \ldots, \omega_n$ for the objects and $\mu_1, \ldots, \mu_m$ for the methods. We use τ to denote the token for any object or method.

For every expression in a program we want to infer its *type*. The type of an expression is a set of tokens indicating the objects to which the expression may evaluate in any execution of the program. Since exact information is uncomputable in general, we will be satisfied with a (hopefully small) superset.

We now assign every expression a *type variable* $[\![\mathrm{E}]\!]_\tau$. Here E is the syntax of the expression and τ is the token for the nearest enclosing object or method. The intuition is that $[\![\mathrm{E}]\!]_\tau$ denotes the type of the expression E in the object or method τ. In our previous papers, constraint variables simply looked like $[\![\mathrm{E}]\!]$ (without the index); this was possible since we, unlike in this approach, were able to expand away inheritance. We have a similar type variable $[\![\mathrm{x}]\!]_\tau$ for every argument, variable, or parent slot x. There is also an auxiliary type variable $[\![\mu]\!]$ for each method μ. $[\![\mu]\!]$ denotes the type of the values that μ may return. The auxiliary type variables are needed to handle non-local returns. All type variables range over sets of tokens.

2.3 Constraints for SELF

From the syntax of the given program we generate and solve a finite collection of *constraints*. These constraints, which are presented by means of a *trace graph* [10, 9], are all conditional set inclusions. Using the trace graph technique, we need only define constraints for local situations; the corresponding global constraints will then automatically be derived, as described below.

We have a trace graph *node* for each object and method in the program. The MAIN node of the trace graph is the node corresponding to the initial method in the program being analyzed (in a C program this would be the main function). Each trace graph node contains *local constraints* which are generated from the syntax; some examples are shown in Figure 1. The local constraints are quite straightforward. They directly reflect the semantics of the corresponding constructs, constraining the types of expressions in a bottom-up fashion. For slots, **1)**, **2)**, and **3)**, ω is an object and μ is a method. The first constraint is associated with a dynamic parent slot. The constraint says that the initial object in the slot is included in the slot's type. The second constraint is analogous, but for a variable slot. The third constraint is associated with a method slot and it lifts the type of the method, $[\![\mu]\!]$, to the type of the slot, $[\![\mathrm{Id}]\!]$. Constraint **4)** specifies that the type of a sequence of expressions is determined by the type of the last expression in the sequence. **5)** is for a primitive, _Clone. The constraint says that a clone of an object has the same type as the object. There are of course many more primitives—a few hundred in fact—so the type inference implementation has a database of primitive local constraints currently covering the 100 most important primitives. Constraints **6)** and **7)** reflect the fact that an object literal evaluates to itself.

	Slots:	Constraint:
1)	$\mathrm{Id}{*}\leftarrow \omega$	$[\![\mathrm{Id}]\!]_\tau \supseteq \{\omega\}$
2)	$\mathrm{Id} \leftarrow \omega$	$[\![\mathrm{Id}]\!]_\tau \supseteq \{\omega\}$
3)	$\mathrm{Id} = \mu = (\ \mid \mathrm{S} \mid \mathrm{E}\ )$	$[\![\mathrm{Id}]\!]_\tau \supseteq [\![\mu]\!] \supseteq [\![\mathrm{E}]\!]_\mu$
	Expression:	Constraint:
4)	$\mathrm{E}_1\ .\ \mathrm{E}_2$	$[\![\mathrm{E}_1\ .\ \mathrm{E}_2]\!]_\tau \supseteq [\![\mathrm{E}_2]\!]_\tau$
5)	$\mathrm{E}\ \mathsf{\_Clone}$	$[\![\mathrm{E}\ \mathsf{\_Clone}]\!]_\tau \supseteq [\![\mathrm{E}]\!]_\tau$
6)	$(\ \mid \mathrm{S} \mid\ )$	$[\![\ (\ \mid \mathrm{S} \mid\ )\ ]\!]_\tau \supseteq \{\text{the token for this object}\}$
7)	$[\ \mid \mathrm{S} \mid \ldots]$	$[\![\ [\ \mid \mathrm{S} \mid \ldots]\ ]\!]_\tau \supseteq \{\text{the token for this block}\}$

Fig. 1. Some local constraints for SELF.

When defining the local constraints we are fortunate that SELF is a minimal language in which most mechanisms are coded in the language itself, starting with only a small core. For example, control structures such as ifTrue:False:, do:, and whileTrue: are implemented by normal SELF methods and objects. However, we pay a daunting price for this simplicity, since *any* program being analyzed is likely to use several of these control structures which are unseparable from the standard SELF world of more than 40,000 lines of code. Our experiences with this are detailed in Section 3.

Trace graph *edges* describe possible message sends. There is an edge from node A to node B, if A may invoke B. Each edge is decorated with *connecting constraints*, which reflect parameter passing during a message send. The crucial part in setting up this picture is to associate a *condition* with each edge. If all possible edges were taken seriously, then we would obtain only very pessimistic results from our type inference. It would correspond to the assumption that every object could send every possible message to every other object. However, if the condition of an edge is false, then it can safely be ignored.

Edge conditions must be *sound*, i.e., if some condition is false in the type analysis, then the corresponding message send must be impossible at run-time. There is a circularity between conditions and local constraints, in that conditions depend on type variables which are determined by local constraints, the relevance of which depends on conditions. We resolve this circularity by a global fixed-point computation.

The trace graph technique derives *global* constraints by considering all paths without repeating edges from the MAIN node; they correspond to abstract traces of a possible execution. Each such path yields a conditional constraint. The condition is the conjunction of the individual conditions on the edges, and the constraint is the conjunction of both the local constraints in the node to which the path leads and the connecting constraints of the final edge of the path. Such a global constraint means intuitively that if this trace is possible during execution, then these constraints must hold. For further details about the trace graph technique and how to derive global constraints, see [10, 9].

The circularity between conditions and constraints is fairly simple for a language like SMALLTALK, which has *dynamic* dispatch but *static* inheritance. When we include dynamic and multiple inheritance, several further complications arise. The conditions now have to soundly reflect possible searches through parent chains existing only at run-time, and they should of course be as restrictive as possible. This development is detailed below.

For SELF, there are several different kinds of edges, reflecting that the uniform message send syntax is (intentionally) overloaded to do several semantically different things. An edge kind is determined by two orthogonal choices:

- The kind of send. In SELF there are four kinds of sends: send to implicit self, send to an explicit receiver, undirected resend (super), and directed resend (delegation).
- The kind of slot that is "invoked" by the send. In SELF, message sends are used to invoke normal methods, invoke block methods, read variables, and write variables. Furthermore block methods come in two flavors: with and without non-local return.

These 4 × 5 choices have many things in common, but no two of them have the exact same semantics. Hence, our approach involves 20 different kinds of edges.

In the following we will restrict our attention to a send of the form "E_1 id: E_2", i.e., a send to an explicit receiver. Furthermore, we will only look at the cases where a normal method or a block method (with and without non-local return) is invoked. The last simplification we have made is to consider only a send with a single argument; the situations trivially generalize to an arbitrary number of arguments. We first present the edges for the case where the send invokes a normal method. Then we present the edges for invoking block methods. The remaining 17 cases are all quite analogous to the three we show.

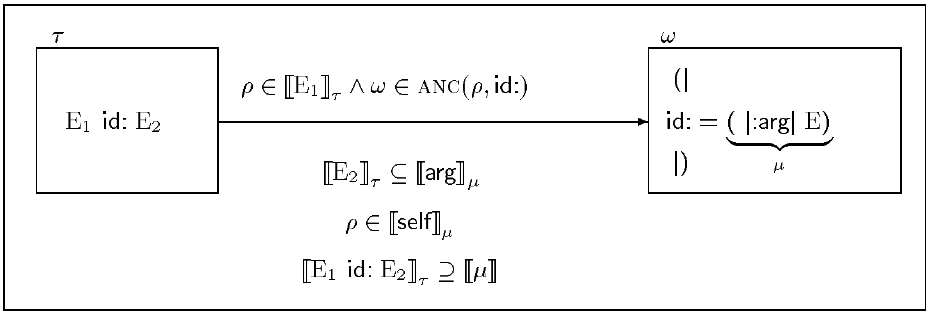

τ
E_1 id: E_2
$\rho \in [\![\mathrm{E}_1]\!]_\tau \wedge \omega \in \textsc{anc}(\rho, \mathsf{id:})$
ω
(|
id: = $\underbrace{(\ |\mathsf{:arg}|\ \mathrm{E})}_{\mu}$
|)
$[\![\mathrm{E}_2]\!]_\tau \subseteq [\![\mathsf{arg}]\!]_\mu$
$\rho \in [\![\mathsf{self}]\!]_\mu$
$[\![\mathrm{E}_1\ \mathsf{id:}\ \mathrm{E}_2]\!]_\tau \supseteq [\![\mu]\!]$

Fig. 2. Edge for invoking a normal method.

Normal method invocation. The edge in Figure 2 describes invocation of a normal method. The expression above the arrow is the condition; the subset relations below the arrow are the constraints that must be respected if the condition becomes satisfied. The method being invoked is μ. It is found in a slot named "id:" in the object ω. The sender is τ, a block or normal method that contains the send "E_1 id: E_2". Finally, ρ is an object in the type of the receiver expression $[\![\mathrm{E}_1]\!]_\tau$. We have the obvious connecting constraints for the argument and the result, plus one for maintaining the self parent in the invoked method μ. Because of dynamic and multiple inheritance, the condition involves the function ANC(ρ,id:) which computes the ancestors of ρ that the message "id:" can reach. This is done by directly expressing the lookup algorithm of SELF, but performing the search in the domain of type variables rather than objects (which do not exist until run-time), see Figure 3. We obtain a great deal of modularity in this manner, since changes in the definition of the lookup algorithm need only be reflected in the ANC function; in all other respects the constraints can be left unchanged. This has already been verified in practice since parent priorities were recently eliminated from SELF. The code in Figure 3, though, applies to the old semantics of SELF as found in release 2.0.1 [1]. For simplicity, the code in Figure 3 ignores privacy of methods and detection of cycles (SELF allows cyclic inheritance graphs).

Block method invocation. Block methods are different from normal methods in two major aspects. First, instead of a self parent they have an anonymous parent that refers to the lexically enclosing method. Second, they may have a non-local return. It is of minor importance for our work that block methods are only found in block objects, and that they are always in slots whose names start with value.

```
Algorithm LOOKUP(obj: object; id: string)
   var m0, m: object;

   if obj has id slot then
      return obj
   end;
   for i := 1 to MaxPriority do
      m0 := nil;
      for all parent slots p
            of priority i in obj do
         m := LOOKUP(contents(p),id);
         if m = ambiguousSend then
            return m
         end;
         if $m \neq$ msgNotUnderstood then
            if $m0 \neq nil \wedge m0 \neq m$ then
               return ambiguousSend
            else
               m0 := m
            end
         end
      end;
      if $m0 \neq nil$ then return m0 end;
   end;
   return msgNotUnderstood
end LOOKUP.
```

```
Algorithm ANC(obj: token; id: string)
   var found, f: boolean;
   var res: set of token;
   if obj has id slot then
      return {obj}
   end;
   for i := 1 to MaxPriority do
      found := false; res := {};
      for all parent slots p
            of priority i in obj do
         f := true;
         for all a in $[\![p]\!]$ do
            f := $f \wedge ANC(a,id) \neq \{\}$;
            res := $res \cup ANC(a,id)$
         end;
         found := $found \vee f$
      end;
      if found then return res end
   end;
   return res
end ANC.
```

Fig. 3. Method lookup in Self release 2.0.1 and the derived ANC function.

The edge for invoking a block method without a non-local return is shown in Figure 4. We have renamed the send to "E_1 value: E_2". The condition reflects that the send is subject to the full lookup, e.g. in the "worst" case the block method may be inherited through dynamic parents. Comparing with the edge for invoking a normal method, we note that there is no constraint involving self. This is because block methods have no self slot of their own; rather they *inherit* the lexically enclosing method's self slot through their lexical parent, and the invocation of the block method does not affect self. Otherwise the two kinds of edges have the same connecting constraints and conditions.

The edge for invoking a block method with a non-local return is shown in Figure 5. The block method being invoked is again labeled μ. The "↑" designates that the following expression, E', is returned non-locally. A non-local return in Self, as in Smalltalk, does not return to the point of the send that invoked the block method, but to the point of the send that invoked the method in which the block is contained. The edge in Figure 5 differs by a single constraint from the edge in Figure 4. The constraint involving the type of the send expression is missing, because invocation of a block method with non-local return does not

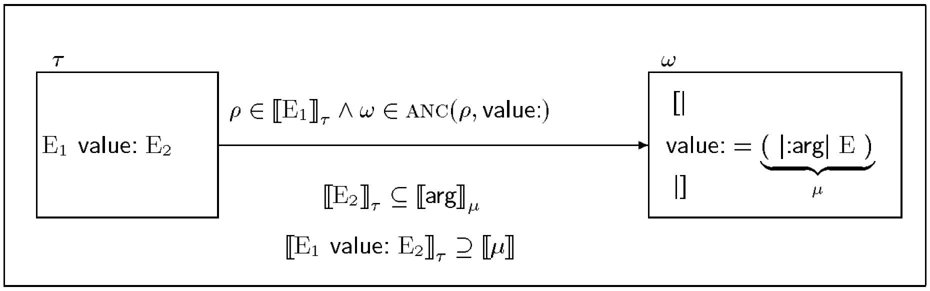

τ
ω
E_1 value: E_2
$\rho \in [\![E_1]\!]_\tau \wedge \omega \in \textsc{anc}(\rho, \text{value:})$
[|
value: = $\underbrace{(\ |\text{:arg}|\ E\ )}_{\mu}$
|]
$[\![E_2]\!]_\tau \subseteq [\![\text{arg}]\!]_\mu$
$[\![E_1 \text{ value: } E_2]\!]_\tau \supseteq [\![\mu]\!]$

Fig. 4. Edge for invoking a block method.

return to the send point that invoked the block method.

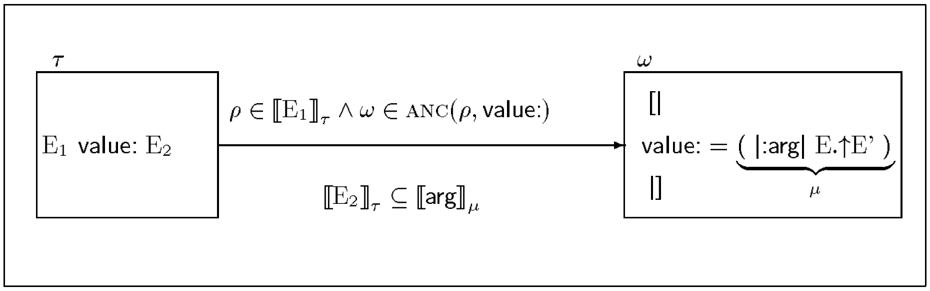

τ
ω
E_1 value: E_2
$\rho \in [\![E_1]\!]_\tau \wedge \omega \in \textsc{anc}(\rho, \text{value:})$
[|
value: = $\underbrace{(\ |\text{:arg}|\ E.\uparrow E'\ )}_{\mu}$
|]
$[\![E_2]\!]_\tau \subseteq [\![\text{arg}]\!]_\mu$

Fig. 5. Edge for invoking a block method with non-local return.

$$\begin{array}{ll} \mu_{\text{normal}} : (& \vdots \\ & \mu_{\text{block}} : [\ \ldots \uparrow \ldots\ ] \\ & \vdots \\ ) & \end{array}$$

$$[\![\mu_{\text{block}}]\!] \subseteq [\![\mu_{\text{normal}}]\!]$$

Fig. 6. Non-local connecting constraint.

Non-local return. Independently of these edges, we have some unconditional non-local connecting constraints, which reflect the possible control flow of non-local returns. One such is shown in Figure 6: the result of a block method, μ_{block}, with non-local return can become the result of the enclosing normal method object, μ_{normal}.

2.4 Code Duplication

As suggested in [10, 9], it is often necessary to duplicate code to obtain sufficiently precise types. The idea is for each method to make individual copies for every syntactic invocation. These copies can then be separately analyzed, so that unwanted “cross-constraints” can be avoided.

This process can be iterated to yield ever more precise type information, but at a huge cost since a single copying of all methods may square the size of the program. The need for code duplication may vary greatly from method to method; thus, a good implementation must include sensible heuristics for deciding when to duplicate the code for a method. We have been quite successful in finding such heuristics for the SELF analysis; see Section 4 for further details.

2.5 Constraint Solving

All the global type constraints, derived from the trace graph, can be written as:

$$c_1 \wedge \ldots \wedge c_n \implies X \subseteq V$$

where V is a type variable, X is either a type variable or a constant set of tokens, and the c_i's is are *monotonic* conditions. Monotonicity of a condition simply means that it can never be true for a particular assignment of types to variables and false for a strictly larger one. In our case, assignments are ordered by variable-wise set inclusion.

In [10] it is shown that if the type constraints are monotonic, they have a unique minimal solution, which is also computable. To see that our constraints for SELF are monotonic, we look at a typical condition c_i which has the form

$$\rho \in [\![\mathrm{E}]\!]_\tau \wedge \omega \in \textsc{anc}(\rho, id).$$

Conjunction preserves monotonicity, so it is enough to ensure that each conjunct is monotonic. The first conjunct obviously is. The second conjunct is monotonic if a larger assignment of types to variables will result in a larger set of ancestors being returned by ANC. This property of ANC is of course dependent on the particular lookup strategy that ANC is derived from. However, any reasonable strategy will likely share this property and indeed the one we study does. This can be informally seen as follows. Imagine first executing ANC for one type assignment and then later executing it for a larger type assignment, see Figure 3. The second execution will perform more iterations in the innermost loop, since $[\![\mathsf{p}]\!]$ will be larger, (by induction) causing a larger res set to be accumulated. Furthermore, searching more parents in the inner loop will never cause the search to be terminated at a higher parent priority (i.e., earlier), since there are more chances for the control variable f to become false, hence found will become true no sooner.

In Section 4 we describe how to obtain an efficient, polynomial-time algorithm for solving monotonic type constraints. It is similar to the one presented in [9], in using a lazy, incremental strategy; however, the new algorithm is a drastic improvement. Early experiments with the old algorithm, adapted to the SELF language, showed an unacceptable performance.

3 Examples

A good illustration of *both* the strengths and weaknesses of our type inference technique is obtained by looking at examples. Illustrating the capabilities of the technique has been our first goal when choosing the examples. Our second goal has been to show that our approach is realistic. This has had three consequences:

- All our examples are "real" in the sense that they are part of the standard SELF system. The code has not been (re)written or modified in any way whatsoever for the purpose of being analyzed. If desired, the test programs can be studied in context, by obtaining a copy of the SELF system by anonymous ftp from `self.stanford.edu`. The only pieces of code that we have written are a few message sends to invoke the main body of the code.
- The code being analyzed is not self-contained: it is an integrated part of a large body of code, the SELF world. Previous work on type inference has mainly analyzed self-contained code. Analyzing a 200 line program of which 100 lines implement a unary representation of natural numbers is not as interesting, or challenging, as analyzing 200 lines of code that just assume and use a fully developed implementation of numbers, collection classes, and other data structures.
- Previous articles [10, 9] have listed small programs, the derived constraints, and their minimal solution as found by a particular constraint solving algorithm. Since the constraints are hard to read and we want to scale towards analyzing realistically sized programs which produce *thousands* of constraints, we do not list any constraints in this section.

We present three concrete examples, each focusing on different aspects of both the type inference and the SELF system. The first example shows how tightly coupled much of the SELF code is. The second example illustrates the capabilities of a browser based on type inference. The third example deals with dynamic inheritance.

3.1 The Tangled Web: Simple Arithmetic

Our first example is conceptually the simplest, yet it illustrates very well just how tightly integrated the code in the SELF world is. We start with almost the simplest possible expressions and observe how a plethora of methods and objects come into play.

SELF has a hierarchy of number objects, the leaves of which are: smallInt which are limited to 30 bits precision, bigInt which have unlimited precision, and float. Mixed representation arithmetic is supported via dynamic dispatching and implicit coercions. For example, if an operation on smallInt overflows, the objects will transparently be coerced to bigInt and the operation will be retried. The consequence is that understanding how a simple expression such as 3+4 is computed is not trivial.

We have defined a small object with messages performing various arithmetic operations, some of which will fail if executed; the object is shown in Figure 7. We

```
example1Object = ( |
  parent* = traits oddball.             "Inherit default behavior."
  test1 = ( 3   +   4 ).                "Result: 7"
  test2 = ( 3.5 +   4 ).                "Result: 7.5"
  test3 = ( 3   + 4.5 ).                "Result: 7.5"
  test4 = ( 3.5 + 4.5 ).                "Result: 8.0"
  test5 = ( nil + 4.5 ).                "Result: error"
  test6 = ( 3   + 'random string' ).    "Result: error"
| ).
```

Fig. 7. Arithmetic operations.

have performed type inference on this object in several different configurations; observations about this are listed in Figure 8, where, e.g., the column labeled "standard system" shows the results of inferring types in the standard SELF system. Each cell in the table shows the inferred type of the expression analyzed, the number of trace graph edges involved, the number of methods analyzed, and the time it took to infer the types on a SPARCstation 2 (we will comment briefly on the execution times in Section 4).

First look at the "standard system" column. Several interesting points should be noted:

- This is the exact output as produced by the type inference implementation, except for a nicer write-up.
- Our type inference does no range analysis or constant folding, so it cannot determine if overflows occur. This is why bigInt shows up e.g. in test1, even though 3+4 will never overflow.
- Why does bigInt show up in test3? Inspection of the code reveals that adding a float to a smallInt can never result in a bigInt. The reason that the type inference cannot determine this is related to a single primitive message send, see Figure 9. The algorithm infers that the primitive send may fail, hence it analyzes the fail block. The fail block handles two failure modes: overflow (adding too big a smallInt) and badTypeError (e.g. adding a float to a smallInt). The two failure modes cannot be distinguished by looking at type information only. The fail block distinguishes them by comparing *values*, not types (specifically string prefix tests are used). Being limited to types, the inference algorithm must assume that both failure modes are possible, hence it cannot determine that the bigInt case never occurs.
- The empty types, inferred for the last two tests, indicate that an error will occur with certainty. The trace graph can be analyzed, and the exact send(s) that will fail can be identified.
- Of the two empty types inferred, one is a lot harder to infer than the other. In test5 there is only a single message send that has a matching slot: this is the send of nil to implicit self that is matched by a slot in an ancestor. The next send, +, finds no matching slot in nil or any of nil's ancestors and hence the type inference is completed. The contrast is test6: to determine that a

Method	Standard system	No bigInt	No bigInt or isPrefixOf:
`test1 = ( 3 + 4 )`	{ smallInt, bigInt } 16,757 edges 4,969 nodes 60 seconds	{ smallInt } 2,444 edges 874 nodes 8 seconds	{ smallInt } 91 edges 41 nodes 0 seconds
`test2 = ( 3.5 + 4 )`	{ float } 16,763 edges 4,972 nodes 60 seconds	{ float } 2,490 edges 894 nodes 9 seconds	{ float } 76 edges 34 nodes 0 seconds
`test3 = ( 3 + 4.5 )`	{ float, bigInt } 16,782 edges 4,979 nodes 60 seconds	{ float } 2,509 edges 904 nodes 8 seconds	{ float } 123 edges 58 nodes 0 seconds
`test4 = ( 3.5 + 4.5 )`	{ float } 16,742 edges 4,964 nodes 60 seconds	{ float } 2,468 edges 885 nodes 8 seconds	{ float } 54 edges 25 nodes 0 seconds
`test5 = ( nil + 4.5 )`	{ } 1 edge 1 node 0 seconds	{ } 1 edge 1 node 0 seconds	{ } 1 edge 1 node 0 seconds
`test6 = ( 3 + 'str' )`	{ } 16,755 edges 4,969 nodes 61 seconds	{ } 2,520 edges 905 nodes 9 seconds	{ } 2,434 edges 880 nodes 9 seconds

Fig. 8. Type inference of arithmetic operations.

string is not a good argument to give the + method of smallInt requires a detailed analysis of this and many other methods.

- The number of edges and nodes is very large; each edge corresponds to a possible message send, each node corresponds to a method that was analyzed (of course, the large numbers are partially caused by the code duplication being done internally by the type inference algorithm, see sections 2.4 and 4).

In order to explain how a simple addition can potentially result in so much code being executed, the type inference was repeated in a world without bigInt. The result is shown in the "no bigInt" column of Figure 8.

The inferred types are not surprising, but the number of edges, although lower, is still high. The explanation is found in the fail block; Figure 9 shows a fragment of code which is the core of the smallInt addition (file `smallInt.self`, line 18). Virtual machine primitives in SELF have the same syntax as "real" message sends, but have names starting with "_" such as _IntAdd:IfFail:. The last argument is a "fail block". It is invoked to produce the result of the primitive send, if the virtual machine is unable to complete the primitive operation, e.g.

```
^ + a = (asSmallInteger _IntAdd: a IfFail: [| :error. :name. |
          ('badTypeError' isPrefixOf: error) ifTrue: [
              " use double dispatching "
              a addSmallInteger: asSmallInteger ] False: [
          ('overflowError' isPrefixOf: error) ifTrue: [
              " retry after coercing to bigInts "
              asBigInteger + a asBigInteger ] False: [
          primitiveFailedError: error Name: name ]]]).
```

Fig. 9. Core of smallInt addition.

because it is given an object of type float where it expects a smallInt. The test isPrefixOf: is complex because it uses general collection behavior to analyze if one sequence (here a string) is a prefix of another. The type inference algorithm precisely infers that the result of isPrefixOf: is true or false, but has to do a non-trivial amount of analysis. Short-circuiting the isPrefixOf: method and performing the inference again shows that we have indeed found the correct explanation for the many edges. The data are shown in the last column of Figure 8. We anticipate that the results of this analysis might lead to redesign of the primitive failure blocks in the future.

The latter example shows that the analysis of failure code significantly complicates the task of type inference. Previous type inference algorithms for object-oriented languages either assume that failures such as overflow are impossible, or treat them as fatal, i.e., the effect of failures is not propagated into the following code. We believe that for a type inference technique to be practical, it *must* be able to precisely analyze failures, not just "normal" execution.

For the last two examples we return to analyzing the standard system, i.e., with bigInt defined and no short-circuiting of any message.

3.2 Browsing Programs: Towers of Hanoi

To gather data for our second example we ran the type inference algorithm on a program that solves the well-known "Towers of Hanoi" problem. The program itself has a long history. Originally, it was written in PASCAL and included in the "Stanford Integer Benchmarks" suite collected by John Hennessy. Later the benchmarks were translated to SELF and used to characterize the run-time performance of the SELF system [3].

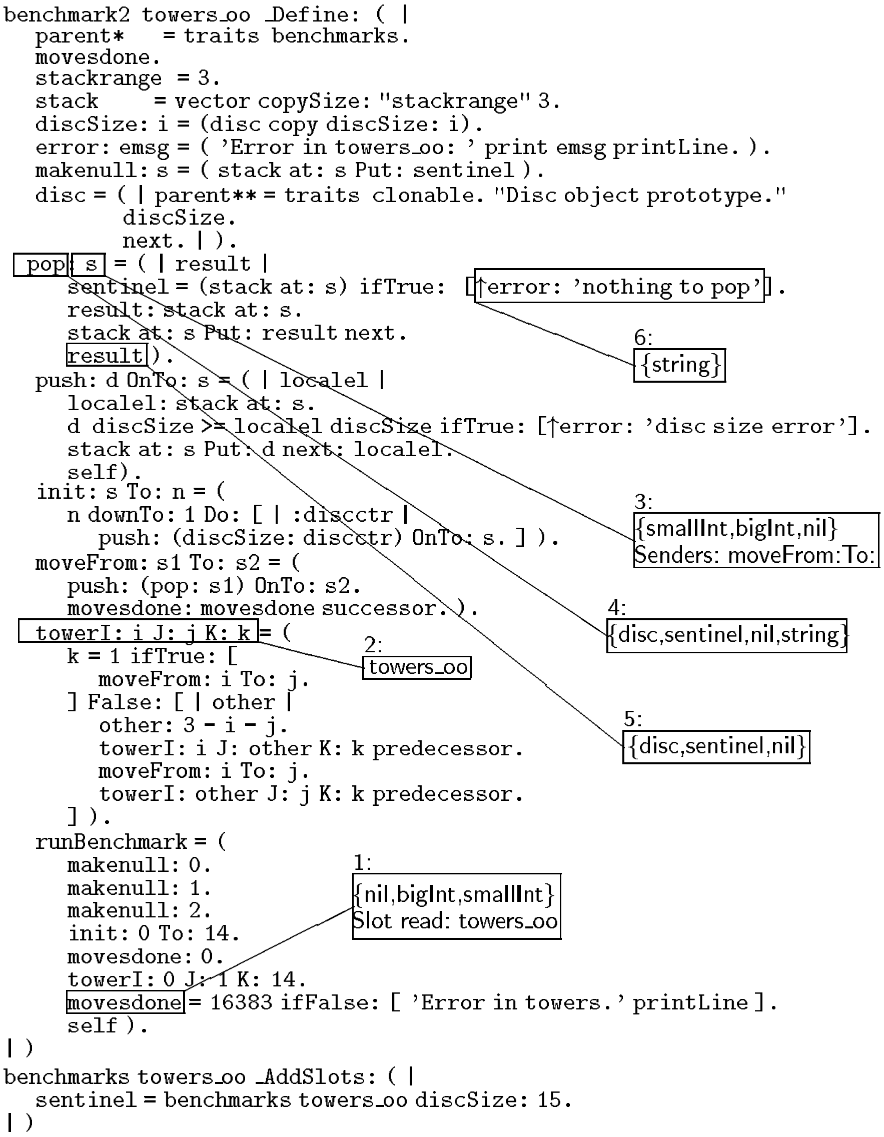
Now we use the towers_oo program to illustrate how a browser may combine program text with inferred types, to make program understanding easier. We call such a browser a "hyperbrowser" and, although we haven't implemented it yet, we believe that the following scenario is realistic, since it is directly based upon information computed by the type inference algorithm.

We use this example to illustrate two things. First, we show how the raw type information computed by the type inference algorithm is useful when a programmer is trying to understand object-oriented programs. Second, we show

how control flow information that can be derived from the type information can be equally useful in the same situations.

The complete program text for the Towers of Hanoi program and selected type annotations produced by the hyperbrowser are shown in Figure 10. Let us look at the annotations one at a time. The paragraph numbers below refer to the numbers next to the annotations in the figure.

1. The runBenchmark method is the "main" method of the program. It is sending various messages to implicit self. Most of the sends ignore the return value and have constant arguments, i.e., their types are manifest from the program text. movesdone is the only exception so we "click" on it to see what information the hyperbrowser can give us. The result is the "balloon" labeled 1: movesdone has type {nil, smallInt, bigInt}. If we want to know which methods the send may invoke (including which variables it may read or write) we can ask the browser for "forward control flow" information. The answer is that the movesdone send will always be answered by reading a variable in a towers_oo object.
   The type of the send includes nil because movesdone is not initialized (see line 3 of the program): by combining type inference with data flow analysis, types can be improved in such situations [14]. The fact that nil shows up in the type could alternatively be attributed to "bad" programming style: prototype objects should have their slots initialized with proper prototypical objects, else they are not prototypical. In the specific case this means that movesdone should be initialized with an integer object. The towers_oo benchmark probably does not follow this style in order to be as similar as possible to the original Pascal program.
2. Next we focus on the tower:I:J:K: method. Again, in such a simple program it is hard to find interesting questions to ask, but at least it is not obvious what the method returns. A click on the selector of the method brings up balloon 2 which shows that the method will always return a towers_oo object, i.e., it returns self.
3. Continuing our exploration we focus on the pop: method. First, what is the type of the argument? This question is easily answered, see balloon 3, but there is the annoying nil again! If we wanted to explore the nil issue further, we could ask the browser for "backward control flow", and be shown all the sends that invoke pop:. We could even ask to see only the sends that invoke pop: with an argument that *may* be nil. This would quickly reveal that nil is here because of another uninitialized variable: other in the towerI:J:K: method.
4. We now look at the return type of pop:. The disc and sentinel objects in balloon 4 seem reasonable, and by now we have learned that the author of the program has a lenient attitude towards nil, so we decide to get an answer to the question: "why can a string be returned?"
5. Our first attempt to answer this question is to "click" on the result send which, being the last in the method, produces the return value. No luck here, though, since there is no string in the resulting balloon 5.



```
benchmark2 towers_oo _Define: ( |
   parent*    = traits benchmarks.
   movesdone.
   stackrange = 3.
   stack      = vector copySize: "stackrange" 3.
   discSize: i = (disc copy discSize: i).
   error: emsg = ( ’Error in towers_oo: ’ print emsg printLine. ).
   makenull: s = ( stack at: s Put: sentinel ).
   disc = ( | parent** = traits clonable. "Disc object prototype."
          discSize.
          next. | ).
 pop: s = ( | result |
      sentinel = (stack at: s) ifTrue: [↑error: ’nothing to pop’].
      result: stack at: s.
      stack at: s Put: result next.
      result ).
   push: d OnTo: s = ( | localel |
      localel: stack at: s.
      d discSize >= localel discSize ifTrue: [↑error: ’disc size error’].
      stack at: s Put: d next: localel.
      self).
   init: s To: n = (
      n downTo: 1 Do: [ | :discctr |
         push: (discSize: discctr) OnTo: s. ] ).
   moveFrom: s1 To: s2 = (
      push: (pop: s1) OnTo: s2.
      movesdone: movesdone successor. ).
   towerI: i J: j K: k = (
      k = 1 ifTrue: [
         moveFrom: i To: j.
      ] False: [ | other |
         other: 3 - i - j.
         towerI: i J: other K: k predecessor.
         moveFrom: i To: j.
         towerI: other J: j K: k predecessor.
      ] ).
   runBenchmark = (
      makenull: 0.
      makenull: 1.
      makenull: 2.
      init: 0 To: 14.
      movesdone: 0.
      towerI: 0 J: 1 K: 14.
      movesdone = 16383 ifFalse: [ ’Error in towers.’ printLine ].
      self ).
| )
benchmarks towers_oo _AddSlots: ( |
   sentinel = benchmarks towers_oo discSize: 15.
| )
```

Fig. 10. Program to solve the Towers of Hanoi problem.

6. Going back to balloon 4 and asking for control information, the browser resolves the mystery: balloon 6 pops up and shows us that string is injected into the return type by the block doing a non-local return. It could be claimed that we have found a bug in the program: error: should not return a string; in fact it should not return at all.

By now it should be clear that the type inference algorithm computes detailed and precise information whose application includes, but goes beyond, simply establishing safety guarantees for programs.

3.3 Mastering Dynamic Inheritance: Binary Search Trees

The previous example illustrated how a hyperbrowser can provide assistance in understanding the control flow of programs that use dynamic dispatching. Dynamic inheritance, while providing the ultimate in expressive power, also provides the ultimate in control flow confusion in the sense that even if the exact receiver type of a send is known, it is still not possible to determine which method is invoked. Fortunately, type inference provides the kind of information that programmers need in order to curb the complexity of dynamic inheritance. Our final example demonstrates this.

One use of dynamic inheritance is to implement objects with modal behavior. The canonical example in the Self system is the implementation of an ordered set data type using binary search trees. A tree is identified by a single node. Nodes are either leaves which contain no elements (e.g., the empty tree is a single leaf node) or they are interior nodes which contain an element and two subtrees. The behavior of any given node is determined by a *dynamic* parent. The parent will switch during execution whenever a node changes status from interior to leaf or vice versa.

Figure 11 shows selected parts of the Self implementation of trees. Due to lack of space, we are unable to list the entire implementation which consists of some 300 lines of code. To simplify further, we have also removed a level of inheritance that is only used because the objects implement both sets and multisets. The figure shows three objects: traits emptyTrees which holds behavior for leaves, traits treeNodes which holds behavior for interior nodes, and treeNode which has a dynamic parent that initially holds traits emptyTrees.

The includesKey: methods search a tree for a given key. Since treeNode inherits this method through the dynamic parent, the result of sending includesKey: to a treeNode depends on the object in the parent slot.

The key to understanding a program that uses dynamic inheritance, is to understand *all* the behavioral modes. This requires knowing all the possible objects that can occur in the dynamic parent slots. Unfortunately, finding these objects is not easy: for example, with incomplete knowledge about the set of parents, the programmer has incomplete knowledge about the assignments that change the dynamic parents. Breaking this circularity is not trivial. Merely executing the program and observing the contents of the dynamic parent slots is not sufficient, since the programmer can never be sure that a new type of parent will

```
traits emptyTrees _Define: ( |
    parent* = traits tree.
    add: x = (parent: nodeProto nodeCopyKey: x Contents: x).
    includesKey: k = ( false ).
    removeAll = (self).
    ...
| )

traits treeNodes _Define: ( |
    parent* = traits tree.
    includesKey: x = (
      findKey: x
         IfHere: true
          IfNot: [ | :subTree | subTree includesKey: x ] ).

    removeAll = ( parent: emptyTreeTraits ).
    left  <- treeSet copy.
    right <- treeSet copy.
    key.
    ...
| )

treeNode _Define: ( |
    parent* <- traits emptyTrees.
| )
```

Fig. 11. Binary search tree implementation using dynamic inheritance.

not show up next time the program is executed. The strong guarantees that the programmer needs to reason correctly about the behavior of the program cannot be provided by the subsets of actual types that are the result of observing dynamic behavior. In contrast, type inference, which computes supersets, directly provides such guarantees: if a type T is inferred for a dynamic parent slot S, the programmer knows that in *any* execution of the program, the contents of S will always be (a clone of) one of the objects found in T. Another way of looking at this is that the fixed-point computation performed during type inference breaks the above-mentioned circularity.

To be concrete, we have inferred types for an example program that uses the (non-simplified) search trees taken from the SELF world. The analysis—as usual—computes a type for every expression and slot in the program, but in this case we focus on a single question: "what are the possible parents of treeNode objects?" The answer is decisive:

$$[\![\mathsf{parent}]\!]_{\mathsf{treeNode}} = \{ \text{ }\mathsf{traits\ emptyTrees},\ \mathsf{traits\ treeNode}\text{ } \}.$$

That is, the type analysis has inferred the precise behavioral modes for tree nodes. Having access to the types of all dynamic parents, the hyperbrowser could also provide control flow information, including information for sends that are looked up through dynamic parents. Furthermore, since the inferred types are precise, so will the control flow information be. We will not go into details with this since it is similar to the Hanoi browsing scenario.

4 Algorithm and Implementation

The problem with a naive implementation of the constraint solver is that an explicit construction of the trace graph will be much too costly for programs of realistic sizes. In [9] this was remedied by an incremental computation of both the trace graph and its minimal solution. Starting with the MAIN node, the general situation would be that a connected part of the graph had been constructed and its solution computed. All outgoing edges from this fragment of the full trace graph were stored in a data structure along with their conditions. If any such condition was satisfied in the current minimal solution, then the local constraints of the targeted node were included, and a new, larger solution was computed. This technique works as long as the conditions are monotonic, as explained in Section 2.5, and it will ensure a polynomial running time.

In the present situation, with thousands of methods, even the collection of outgoing edges is too colossal to manage. Hence we have developed an even more frugal strategy, where we only generate those edges whose conditions are true in the current minimal solution. As the solution increases over time, we may have to go back to earlier processed nodes and include more edges whose conditions have now become true. In particular, we may have to go back and extend previously computed ANC sets, when new tokens are added to type variables associated with assignable parents. Adhering to this strategy leads to an acceptable performance.

As indicated earlier, the quality of the inferred types depends on finding good heuristics for duplicating the code of methods. For example, if no duplication is done, the inferred type of 3+4 degrades to a set of nineteen tokens, rather than the optimal two which our current heuristic can infer.

The problem can be described by looking at the message send in Figure 2, where we must choose whether to create a duplicate of the method μ. If we always duplicated, then the type inference algorithm might never terminate since, for example, a recursive method would result in an infinite number of duplicates being produced during the analysis. In [9] we created one duplicate for every syntactic message send with selector id: in the original program. In the SELF algorithm we apply a "hash function" and create a duplicate if none already exists with the same hash value. Since there are only finitely many different hash values, termination is ensured. The hash value of the situation in Figure 2 is a triple:

$$(\text{parse tree node for the send}, \text{origin}(\tau), \rho).$$

Here, the origin(τ) indicates the original version of the τ method (τ may of course itself be a duplicate). The intuition behind this hash function has two parts. The last component, ρ, ensures that each new receiver type will get its own duplicate, resulting in a duplication strategy akin to customization [3]. The first two components of the hash function refines this strategy to ensure that sends that are different in the original program will invoke different duplicates, even if the sends have the same receiver. This is useful because different sends often supply arguments of different types. The situation is somewhat different for resends, but we will not elaborate this further.

We cannot use type information as part of the hash value, since this has not been computed yet when the hash function must be applied. To compensate for this, a small carefully selected set of methods from the standard SELF world is *always* duplicated, independently of what the hash value recommends. Part of the careful selection is to guarantee termination of the algorithm. The selected methods that are always duplicated include ifTrue:False: and other methods in booleans, some double-dispatching methods (to preserve the type information of the arguments through the second dispatch), and a few "cascading" sends.

The type inference implementation is written in 4,000 lines of SELF. Currently it uses a lot of memory; the examples in Section 3 were running in a 30 Mbyte heap. The main reason for this voracity is that the implementation has not been optimized for space. In practice, run time seems to be proportional to the number of edges and the total size of all type variables. For the execution times of the arithmetic examples, see Figure 8. The measurements were done on a SPARCstation 2. The Hanoi and Search Tree examples have similar execution times as those found in the first column of Figure 8, since in all these cases the dominating factor is the bigInt arithmetic. Specifically, inferring types for the Hanoi example involves 17,380 edges, 5,143 nodes, and takes 93 seconds.

5 Conclusion

We have developed and efficiently implemented a powerful type inference algorithm for SELF. Our algorithm involves a novel way of defining and solving constraints that describe a dynamically changing inheritance graph. To the best of our knowledge, our type inference algorithm is the first algorithm to simultaneously handle dynamic inheritance, multiple inheritance, object based inheritance, and blocks with non-local returns. Furthermore, we have shown that it can handle real programs such as the standard SELF benchmarks, including the traditionally difficult (and often ignored) constructs of primitive failures and user-defined control structures. Our algorithm provides detailed information even for partial and incorrect programs rather than merely rejecting them; for this reason it can be useful as a basis for various advanced tools.

The tools that can be based on the type information include a msgNotUnderstood-checker and an ambiguousSend-checker. Since the computed type information is a precise and conservative approximation, the tools will be correspondingly precise and conservative.

We have also presented a scenario in which a programmer uses an interactive hyperbrowser that draws extensively on the type information inferred by our algorithm to answer queries about types and control flow in a program.

Another possible tool could use the type information to identify unused (dead) code. Dead code detection is important for generating stand-alone applications. Without type inference, one would have to include the entire standard world since it would be hard to determine which parts could not possibly be required at run-time. Using type information, a conservative (but quite precise) approximation to code liveness could be computed, and methods and objects that are deemed dead by this information could safely be omitted from the application.

A further potential gain is particular to the SELF technique of dynamic compilation. The result of type inference gives an upper bound on the methods that must be compiled; thus, these methods could be pre-compiled, obviating the need for dynamic compilation and allowing the compiler to be omitted from stand-alone applications.

The version of SELF that we have described in this paper is the version that was publicly released in the Fall of 1992 (release 2.0.1). It is both a simple and a complex language. Simple, e.g., because it does not have classes and metaclasses, but complex, e.g., because it has complicated inheritance rules [4]. The type inference work has focused attention on many of the complexities, providing input to an ongoing attempt to further simplify SELF. One example is the recent elimination of parent priorities.

The SELF language is less amenable to type inference than many other object-oriented languages, yet we have obtained promising results. We believe that our algorithm is adaptable to other languages, including typed ones like C++. In the latter case, our types would provide more precision than the type declarations written by the programmer. Furthermore, since our algorithm could infer concrete (implementation-level) types for each call site, it could be used as the basis for compiler optimizations such as the inlining of virtual function calls.

Acknowledgement. The authors thank David Ungar, Randall Smith, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, and John Maloney for helpful comments on a draft of the paper. The first author would also like to thank Sun Microsystems Laboratories for its support.

References

1. Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, and David Ungar. The SELF programmer's reference manual, version 2.0. Technical report, Sun Microsystems, Inc, 2550 Garcia Avenue, Mountain View, CA 94043, USA, 1992. SMLI document 93-0056. Available by anonymous ftp from `self.stanford.edu`.
2. Alan H. Borning. Classes versus prototypes in object-oriented languages. In *ACM/IEEE Fall Joint Computer Conference*, pages 36–40, 1986.
3. Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *Proc. OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 1–15, 1991.
4. Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF. In *Lisp and*

Symbolic Computation 4(3), pages 207–222, Kluwer Acadamic Publishers, June 1991.

5. Adele Goldberg and David Robson. *Smalltalk-80—The Language and its Implementation.* Addison-Wesley, 1983.
6. Justin O. Graver and Ralph E. Johnson. A type system for Smalltalk. In *Seventeenth Symposium on Principles of Programming Languages*, pages 136–150. ACM Press, January 1990.
7. 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.
8. Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proc. OOPSLA'86, Object-Oriented Programming Systems, Languages and Applications*, pages 214–223. Sigplan Notices, 21(11), November 1986.
9. Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. In *Proc. ECOOP'92, Sixth European Conference on Object-Oriented Programming*, pages 329–349. Springer-Verlag (*LNCS* 615), Utrecht, The Netherlands, July 1992.
10. Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proc. OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161, Phoenix, Arizona, October 1991.
11. Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference for partial types. *Information Processing Letters*, 43:175–180, 1992.
12. Michael I. Schwartzbach. Type inference with inequalities. In *Proc. TAPSOFT'91*, pages 441–455. Springer-Verlag (*LNCS* 493), 1991.
13. David Ungar and Randall B. Smith. SELF: The power of simplicity. In *Proc. OOPSLA'87, Object-Oriented Programming Systems, Languages and Applications*, pages 227–241, 1987. Also published in Lisp and Symbolic Computation 4(3), Kluwer Acadamic Publishers, June, 1991.
14. Jan Vitek, R. Nigel Horspool, and James S. Uhl. Compile-time analysis of object-oriented programs. In *Proc. CC'92, 4th International Conference on Compiler Construction, Paderborn, Germany*, pages 236–250. Springer-Verlag (*LNCS* 641), 1992.
15. Mitchell Wand. A simple algorithm and proof for type inference. *Fundamentae Informaticae*, X:115–122, 1987.

This article was processed using the LaTeX macro package with LLNCS style