In Proc. *ECOOP'92*, Springer-Verlag (LNCS 615), pages 329–349.

Making Type Inference Practical

Nicholas Oxhøj alf@daimi.aau.dk
Jens Palsberg palsberg@daimi.aau.dk
Michael I. Schwartzbach mis@daimi.aau.dk

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

Abstract

We present the implementation of a type inference algorithm for untyped object-oriented programs with inheritance, assignments, and late binding. The algorithm significantly improves our previous one, presented at OOPSLA'91, since it can handle collection classes, such as List, in a useful way. Also, the complexity has been dramatically improved, from exponential time to low polynomial time. The implementation uses the techniques of incremental graph construction and constraint template instantiation to avoid representing intermediate results, doing superfluous work, and recomputing type information. Experiments indicate that the implementation type checks as much as 100 lines pr. second. This results in a mature product, on which a number of tools can be based, for example a safety tool, an image compression tool, a code optimization tool, and an annotation tool. This may make type inference for object-oriented languages practical.

Keywords: Type inference, implementation, collection classes, tools, Smalltalk.

1 Introduction

The basic purpose of doing type inference for untyped object-oriented programs is to guarantee that all messages are understood [1].

At OOPSLA'91 we presented a type inference algorithm for an untyped object-oriented language with inheritance, assignments, and late binding [17]. The algorithm can type check many common programs, including those with polymorphic and recursive methods. It can *not*, however, infer types in programs that use collection classes. A collection class is used to contain different instances in different parts of the program. For example, a class List may be used as a boolean list in one place, and as an integer list in another place. Our algorithm from OOPSLA'91 confuses these uses of List, leading to rejection of many type-safe programs.

This paper presents:

- An improved algorithm that handles collection classes in a useful way; and

- An implementation of the new algorithm, including two novel techniques for making type inference faster and less space consuming. The complexity has been reduced from exponential time to low polynomial time.

The implementation type checks hundreds of lines in seconds, runs on Sun3, SPARC, and HP9000, and is available by anonymous ftp.

The improved algorithm is similar to the previous one in that types are sets of classes. This is a simple, yet flexible concept that is useful when considering implementation aspects of object-oriented languages. Together with the speed and generality of our current implementation it opens new perspectives on what tools can be provided for untyped object-oriented languages. This includes tools for doing image compression, code optimization, annotation of programs, class hierarchy construction, and insertion of dynamic checks for class memberships. These usages of type inference may turn out to be just as important as the safety guarantee. In our opinion, type inference is a promising basis for such sophisticated tools, and our new implementation techniques may make it practical.

In the following section we outline the example language, which is exactly the one of the OOPSLA'91 paper. In section 3 we survey the previous algorithm and explain how to extend it to handle collection classes. In section 4 we demonstrate how the implementation works, and in section 5 we summarize the tools that we either have built or envision, on the basis of our implementation.

2 The Language

We now outline an example language on which to apply our algorithm. The language resembles SMALLTALK [8], see figure 1, and is taken from our previous paper [17].

(Program)	P	::=	$C_1 \ldots C_n$ E
(Class)	C	::=	**class** ClassId [**inherits** ClassId] **var** $Id_1 \ldots Id_k$ $M_1 \ldots M_n$ **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 | ClassId new | self class new | E instanceOf ClassId | self | super | Id | nil

Figure 1: Syntax of the example language.

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 (m_1_ ... 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.

In the paper [17] we demonstrated how to program the classes True, False, Natural, and List is this basic language.

Suzuki [24] was the first to address the problem of type inference for such a language; his algorithm was not capable of checking most common programs, however. Later, Graver and Johnson [10, 9] provided an algorithm for a simplified problem, where the types of instance variables must be specified by programmer so that only the types of arguments are to be inferred. Recently, Hense [11] addressed the problem of inferring types that are useful in connection with separate compilation. This means that his algorithm is not allowed to reconsider the program text when new classes are added to the program. The comparison in [17] demonstrates that this demand leads to the rejection of more programs than does our algorithm. It remains to be seen if the two approaches can be combined.

The basic concept to be used in this paper is that of *type*:

Terminology:

Type: A type is a finite set of classes.

The idea is to compute type information for all expressions in a concrete program. The information should be a superset of the classes of all possible non-nil values to which it may evaluate in any execution of that particular program. We want the set to be as small as possible; smaller sets are more precise, lead to the acceptance of more program, and yield more efficient code generation. Note that our notion of type, which we also investigated in [18, 17, 19], differs from those used in other theoretical studies of types in object-oriented programming [3, 7, 2] and related record-calculi [21, 26].

In the following section we show an algorithm to infer such type information. The algorithm works even for programs with collection classes.

3 The Algorithm

We now review the type inference algorithm presented in [17]. This section can serve both as a brief summary for those who want to appreciate the refinements presented in the following section, and as a gentle introduction for those who want to read the original paper.

The general idea is to define a *type variable* $[\![E]\!]$ for every expression E occurring in the program [25, 22]. We then generate a collection of *constraints* on these variables. Finally, we attempt to solve these constraints. If they are solvable, then their *minimal solution* corresponds to the inferred typing; if not, then the program is not typable. A summary of the entire algorithm is shown in figure 2.

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 away inheritance.
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 2: The algorithm.

3.1 Expanding Inheritance

The algorithm starts by expanding away all use of inheritance in the current program. Thus, we really only perform type inference on the subset of the example language that does not use inheritance.

This approach is common to all previous algorithms, except the one presented in [11]. The well-known insight is that inheritance is a technique for reusing untyped code, and thus an inherited method can be used in completely unexpected contexts in a subclass. Much better typings can be obtained by considering such a method twice—once for the superclass and once for the subclass. This results in the duplication of type variables; in general, more type variables will improve the chances for typability, and the inferred types will be more precise. The algorithm in [11] pays a price for not expanding inheritance; it can type fewer programs.

The actual expansion is a simple syntactic transformation on program texts. It is illustrated in figure 3. Note that methods are duplicated and renamed, and that care must be taken in connection with `self` and `super`. After being expanded, a program may increase quadratically in size. This worst-case only occurs when the inheritance hierarchy is a narrow sequence. For a well-balanced hierarchy the increase in size is much less.

3.2 The Trace Graph

We use the notion of a *trace graph* as an aid in explaining the constraints that we shall impose on the type variables. The *nodes* of the graph correspond to methods, and the *edges* correspond to message sends. There is an extra node corresponding to the main program.

When the graph has been set up, then the constraints are derived from *paths* in the graph starting in the main node. Each path is associated with the *trace* of an execution of the program. In the following we shall illustrate this technique on the example program in figure 4.

Classes *before* expansion:

```
class A
  var x
  method m
    self class new
  method n
    self class new
end A

class B inherits A
  var y
  method m
    super m
end B

class C inherits B
  method m
    super m
  method n
    super n
end C
```

Classes *after* expansion:

```
class A
  var x
  method m
    A new
  method n
    A new
end A

class B
  var x y
  method m$A
    B new
  method n
    B new
  method m
    self m$A
end B

class C
  var x y
  method m$A
    C new
  method n$B
    C new
  method m$B
    self m$A
  method m
    self m$B
  method n
    self n$B
end C
```

Figure 3: Expanding inheritance.

3.3 Nodes and Local Constraints

Each node of the trace graph corresponds to a particular method which is implemented in some class in the current program. In fact, we shall have several nodes for each method implementation. As explained above, it is an advantage to distinguish between different uses of a method. If possible, they should be analyzed separately; this gives rise to more type variables, leading to typability of more programs and better typings.

A simple criterion for distinguishing between methods comes from syntactic message sends. Suppose that the program contains k different syntactic message sends with selector, say, m. For every method with the corresponding name m we will then generate k copies. A class that previously implemented a method m will now implement k methods $\mathsf{m}_1, \ldots, \mathsf{m}_k$, all with identical bodies. The i'th syntactic message send with selector m will now use selector m_i. Clearly, this does not change the semantics of the program, but it does cause the size of the trace graph to be quadratic in the size of the program, in the worst case. However, this distinction is instrumental in securing typability of polymorphic methods.

For the example program we will have the seven trace graph nodes indicated in figure 5. With each node we associate some local constraints which reflect the semantics of the corresponding method body; these are quite straightforward. The general rules are illustrated in figure 6; the local constraints for the example program are listed in figure 7, organized by trace graph nodes and with references to the general rules. Note that rule

```
class A
  method m: e
    e n
end A

class B
  var temp
  method m: e
    (temp := e) n
  method n
    temp p
  method p
    self p
end B

(A new) m: (B new)
```

Figure 4: Example program.

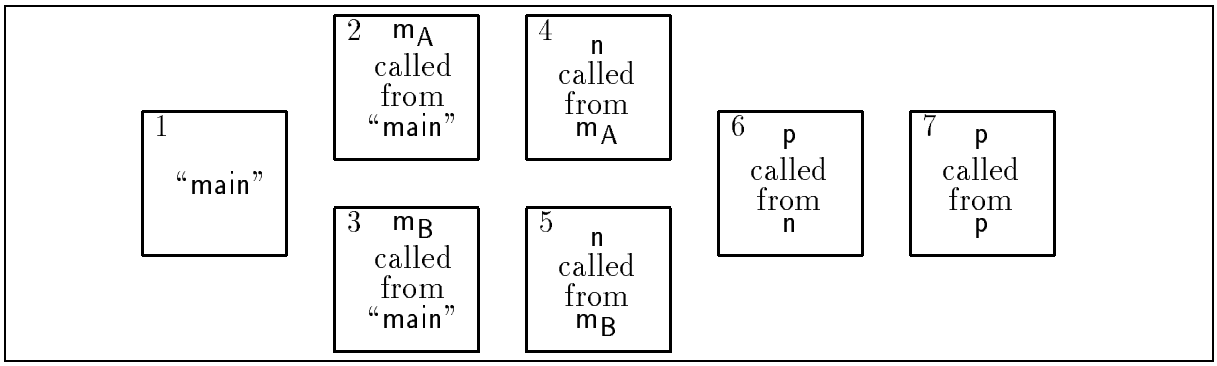


Figure 5: Trace graph nodes for the example program.

number **2)** is responsible for enforcing the safety guarantee.

3.4 Edges, Conditions, and Connecting Constraints

The edges of the trace graph correspond to potential transfers of control. If a method body contains a message send with selector m, then we have an outgoing edge to any node corresponding to a method with the name m. Here the renaming of copies of methods that we mentioned in section 3.3 is to be taken seriously. Thus, the edges of the trace graph for the example program are as illustrated in figure 8.

With each edge we associate a condition. Consider an edge from a node N_1 to a node N_2. The edge will never be traversed during a trace of the program unless the type of the sender in N_1 contains the class implementing the method that N_2 corresponds to. Thus, we can label an edge with a condition that must necessarily hold if the edge, and the local constraints that it leads to, must be taken seriously. A condition always state that

	Expression:	Constraint:
1)	Id := E	$[\![\text{Id}]\!] \supseteq [\![\text{E}]\!] \;\wedge\; [\![\text{Id} := \text{E}]\!] \supseteq [\![\text{E}]\!]$
2)	E m_1 E_1 ... m_n E_n	$[\![\text{E}]\!] \subseteq \{\mathsf{C} \mid \mathsf{C}$ implements $\mathsf{m}_1 \ldots \mathsf{m}_n\}$
3)	E_1 ; E_2	$[\![\text{E}_1 \text{ ; } \text{E}_2]\!] \supseteq [\![\text{E}_2]\!]$
4)	if E_1 then E_2 else E_3	$[\![\text{if } \text{E}_1 \text{ then } \text{E}_2 \text{ else } \text{E}_3]\!] \supseteq [\![\text{E}_2]\!] \cup [\![\text{E}_3]\!]$
5)	C new	$[\![\mathsf{C} \text{ new}]\!] = \{\mathsf{C}\}$
6)	E instanceOf C	$[\![\text{E instanceOf C}]\!] = \{\mathsf{C}\}$
7)	self	$[\![\text{self}]\!] = \{$the enclosing class$\}$
8)	Id	$[\![\text{Id}]\!] = [\![\text{Id}]\!]$
9)	nil	$[\![\text{nil}]\!] = \{\,\}$

Figure 6: Local constraint rules.

Trace graph node	Local constraints	rule
1	$[\![\text{A new}]\!] = \{\text{A}\}$	**5)**
	$[\![\text{B new}]\!] = \{\text{B}\}$	**5)**
	$[\![\text{A new}]\!] \subseteq \{\text{A,B}\}$	**2)**
2	$[\![\text{e}]\!]_1 \subseteq \{\text{B}\}$	**2)**
3	$[\![\text{temp}]\!] \supseteq [\![\text{e}]\!]_2$	**1)**
	$[\![\text{temp := e}]\!] = [\![\text{e}]\!]_2$	**1)**
	$[\![\text{temp := e}]\!] \subseteq \{\text{B}\}$	**2)**

Trace graph node	Local constraints	rule
4	$[\![\text{temp}]\!] \subseteq \{\text{B}\}$	**2)**
5	$[\![\text{temp}]\!] \subseteq \{\text{B}\}$	**2)**
6	$[\![\text{self}]\!]_1 = \{\text{B}\}$	**7)**
	$[\![\text{self}]\!]_1 \subseteq \{\text{B}\}$	**2)**
7	$[\![\text{self}]\!]_2 = \{\text{B}\}$	**7)**
	$[\![\text{self}]\!]_2 \subseteq \{\text{B}\}$	**2)**

Figure 7: Local constraints for the example program.

some class constant is contained in a type variable. An examination of the conditions in figure 8 will show the simplicity of this idea.

With each edge we further associate a collection of connecting constraints, which reflect the semantics of message sends. They simply state that the actual argument is assigned to the formal argument, and that the result of a message send is the result of the body of the invoked method. The connecting constraints for the example program are listed in figure 9.

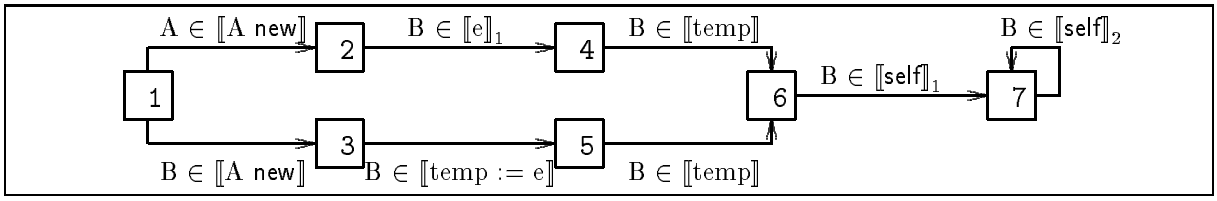


Figure 8: Trace graph edges for the example program.

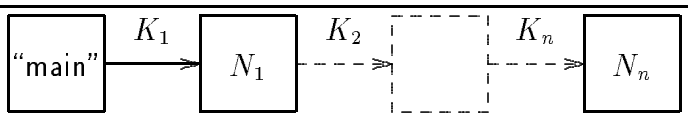
Edge		Connecting constraints
from	to	
1	2	$[\![\text{B } \textsf{new}]\!] \subseteq [\![\text{e}]\!]_1$ $[\![(\text{A } \textsf{new})\text{ m: }(\text{B } \textsf{new})]\!] \supseteq [\![\text{e n}]\!]$
1	3	$[\![\text{B } \textsf{new}]\!] \subseteq [\![\text{e}]\!]_2$ $[\![(\text{A } \textsf{new})\text{ m: }(\text{B } \textsf{new})]\!] \supseteq [\![\text{temp := e}]\!]$
2	4	$[\![\text{e n}]\!] \supseteq [\![\text{temp p}]\!]_1$
3	5	$[\![(\text{temp := e) n}]\!] \supseteq [\![\text{temp p}]\!]_2$
4	6	$[\![\text{temp p}]\!]_1 \supseteq [\![\text{self p}]\!]_1$
5	6	$[\![\text{temp p}]\!]_2 \supseteq [\![\text{self p}]\!]_1$
6	7	$[\![\text{self p}]\!]_1 \supseteq [\![\text{self p}]\!]_2$
7	7	$[\![\text{self p}]\!]_2 \supseteq [\![\text{self p}]\!]_2$

Figure 9: Connecting constraints for the example program.

3.5 Global Constraints

The construction of the trace graph has now been completed. We are left with extracting the global constraints. These will be conditional constraints that arise from paths in the trace graph.

Consider a path from the main node. If the conditions on the edges we encounter all hold, then that path corresponds to a possible execution of the program. Thus, the local and connecting constraints that we arrive at had better hold. We can express this relationship as illustrated in figure 10.



For the above path the conditional constraints will look like

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

where K = a condition on an edge,
L = local constraints of N_n (the final node),
C = connecting constraints of the final edge.

Figure 10: Global constraints derived from a single path.

The total collection of global constraints are derived in this manner from all paths from the main node in the graph. Note that the graph may have infinitely many paths; however, since the program is finite, we will always get a finite set of different conditional constraints—for simple combinatorial reasons.

Note that the global constraints may contain a great amount of recursive dependencies. One cannot decide whether a condition holds without considering the local constraints,

which only come into play if their conditions in turn hold.

For the example program, consider the path originating in node 1, going through node 2 and 4, and ending in node 6. The global constraints derived from this path are

$$\mathrm{A} \in [\![\mathrm{A}\ \mathsf{new}]\!], \mathrm{B} \in [\![\mathrm{e}]\!]_1, \mathrm{B} \in [\![\mathrm{temp}]\!] \Rightarrow \begin{cases} [\![\mathrm{self}]\!]_1 = \{\mathrm{B}\} \\ [\![\mathrm{self}]\!]_1 \subseteq \{\mathrm{B}\} \\ [\![\mathrm{temp\ p}]\!]_1 \supseteq [\![\mathrm{self\ p}]\!]_1 \end{cases}$$

The complete set of global constraints can be found in figure 11.

$$[\![\mathrm{A}\ \mathsf{new}]\!] = \{\mathrm{A}\}$$
$$[\![\mathrm{B}\ \mathsf{new}]\!] = \{\mathrm{B}\}$$
$$[\![\mathrm{A}\ \mathsf{new}]\!] \subseteq \{\mathrm{A},\mathrm{B}\}$$
$$\mathrm{A} \in [\![\mathrm{A}\ \mathsf{new}]\!] \Rightarrow \begin{cases} [\![\mathrm{e}]\!]_1 \subseteq \{\mathrm{B}\} \\ [\![\mathrm{B}\ \mathsf{new}]\!] \subseteq [\![\mathrm{e}]\!]_1 \\ [\![(\mathrm{A}\ \mathsf{new})\ \mathrm{m}\colon (\mathrm{B}\ \mathsf{new})]\!] \supseteq [\![\mathrm{e\ n}]\!] \end{cases}$$
$$\mathrm{B} \in [\![\mathrm{A}\ \mathsf{new}]\!] \Rightarrow \begin{cases} [\![\mathrm{temp}]\!] \supseteq [\![\mathrm{e}]\!]_2 \\ [\![\mathrm{temp} := \mathrm{e}]\!] \supseteq [\![\mathrm{e}]\!]_2 \\ [\![\mathrm{temp} := \mathrm{e}]\!] \subseteq \{\mathrm{B}\} \\ [\![\mathrm{B}\ \mathsf{new}]\!] \subseteq [\![\mathrm{e}]\!]_2 \\ [\![(\mathrm{A}\ \mathsf{new})\ \mathrm{m}\colon (\mathrm{B}\ \mathsf{new})]\!] \supseteq [\![(\mathrm{temp} := \mathrm{e})\ \mathrm{n}]\!] \end{cases}$$
$$\mathrm{A} \in [\![\mathrm{A}\ \mathsf{new}]\!], \mathrm{B} \in [\![\mathrm{e}]\!]_1 \Rightarrow \begin{cases} [\![\mathrm{temp}]\!] \subseteq \{\mathrm{B}\} \\ [\![\mathrm{e\ n}]\!] \supseteq [\![\mathrm{temp\ p}]\!]_1 \end{cases}$$
$$\mathrm{B} \in [\![\mathrm{A}\ \mathsf{new}]\!], \mathrm{B} \in [\![\mathrm{temp} := \mathrm{e}]\!] \Rightarrow \begin{cases} [\![\mathrm{temp}]\!] \subseteq \{\mathrm{B}\} \\ [\![(\mathrm{temp} := \mathrm{e})\ \mathrm{n}]\!] \supseteq [\![\mathrm{temp\ p}]\!]_2 \end{cases}$$
$$\mathrm{A} \in [\![\mathrm{A}\ \mathsf{new}]\!], \mathrm{B} \in [\![\mathrm{e}]\!]_1, \mathrm{B} \in [\![\mathrm{temp}]\!] \Rightarrow \begin{cases} [\![\mathrm{self}]\!]_1 = \{\mathrm{B}\} \\ [\![\mathrm{self}]\!]_1 \subseteq \{\mathrm{B}\} \\ [\![\mathrm{temp\ p}]\!]_1 \supseteq [\![\mathrm{self\ p}]\!]_1 \end{cases}$$
$$\mathrm{B} \in [\![\mathrm{A}\ \mathsf{new}]\!], \mathrm{B} \in [\![\mathrm{temp} := \mathrm{e}]\!], \mathrm{B} \in [\![\mathrm{temp}]\!] \Rightarrow \begin{cases} [\![\mathrm{self}]\!]_1 = \{\mathrm{B}\} \\ [\![\mathrm{self}]\!]_1 \subseteq \{\mathrm{B}\} \\ [\![\mathrm{temp\ p}]\!]_2 \supseteq [\![\mathrm{self\ p}]\!]_1 \end{cases}$$
$$\mathrm{A} \in [\![\mathrm{A}\ \mathsf{new}]\!], \mathrm{B} \in [\![\mathrm{e}]\!]_1, \mathrm{B} \in [\![\mathrm{temp}]\!], \mathrm{B} \in [\![\mathsf{self}]\!]_1 \Rightarrow \begin{cases} [\![\mathrm{self}]\!]_2 = \{\mathrm{B}\} \\ [\![\mathrm{self}]\!]_2 \subseteq \{\mathrm{B}\} \\ [\![\mathrm{self\ p}]\!]_1 \supseteq [\![\mathrm{self\ p}]\!]_2 \end{cases}$$
$$\mathrm{B} \in [\![\mathrm{A}\ \mathsf{new}]\!], \mathrm{B} \in [\![\mathrm{temp} := \mathrm{e}]\!], \mathrm{B} \in [\![\mathrm{temp}]\!], \mathrm{B} \in [\![\mathsf{self}]\!]_1 \Rightarrow \begin{cases} [\![\mathrm{self}]\!]_2 = \{\mathrm{B}\} \\ [\![\mathrm{self}]\!]_2 \subseteq \{\mathrm{B}\} \\ [\![\mathrm{self\ p}]\!]_1 \supseteq [\![\mathrm{self\ p}]\!]_2 \end{cases}$$
$$\mathrm{A} \in [\![\mathrm{A}\ \mathsf{new}]\!], \mathrm{B} \in [\![\mathrm{e}]\!]_1, \mathrm{B} \in [\![\mathrm{temp}]\!], \mathrm{B} \in [\![\mathsf{self}]\!]_1, \mathrm{B} \in [\![\mathsf{self}]\!]_2 \Rightarrow \begin{cases} [\![\mathrm{self}]\!]_2 = \{\mathrm{B}\} \\ [\![\mathrm{self}]\!]_2 \subseteq \{\mathrm{B}\} \\ [\![\mathrm{self\ p}]\!]_1 \supseteq [\![\mathrm{self\ p}]\!]_2 \end{cases}$$
$$\mathrm{B} \in [\![\mathrm{A}\ \mathsf{new}]\!], \mathrm{B} \in [\![\mathrm{temp} := \mathrm{e}]\!], \mathrm{B} \in [\![\mathrm{temp}]\!], \mathrm{B} \in [\![\mathsf{self}]\!]_1, \mathrm{B} \in [\![\mathsf{self}]\!]_2 \Rightarrow \begin{cases} [\![\mathrm{self}]\!]_2 = \{\mathrm{B}\} \\ [\![\mathrm{self}]\!]_2 \subseteq \{\mathrm{B}\} \\ [\![\mathrm{self\ p}]\!]_1 \supseteq [\![\mathrm{self\ p}]\!]_2 \end{cases}$$

Figure 11: Global constraints for the example program.

3.6 Solving the Constraints

A finite set of conditional constraints can be resolved by a single fixed-point computation over an appropriate lattice. The details of this result are presented in [17].

Note how this technique can manage to untangle the recursive dependencies mentioned above. In the limit a condition will either hold or not, and every type variable will attain some value. If no solution exists, then a special error value will be the result of the computation. The time for computing a solution is quadratic in the size of the global constraints.

In [17] we sketched a proof of the soundness of the solution with respect to a dynamic semantics [4, 5, 20, 14, 12].

The global constraints for the example program are solvable; hence, the example program is typable. The minimal solution is shown in figure 12.

$$[\![\mathsf{A\ new}]\!] = \{\mathsf{A}\}$$
$$[\![\mathsf{B\ new}]\!] = [\![\mathsf{e}]\!]_1 = \{\mathsf{B}\}$$
$$[\![\mathsf{(A\ new)\ m\colon\ (B\ new)}]\!] = [\![\mathsf{e\ n}]\!] = [\![\mathsf{temp}]\!] = [\![\mathsf{temp\ p}]\!]_1 = \{\}$$

The remaining type variables have no constraints imposed on them; consequently they attain the value $\{\}$ in the minimal solution.

Figure 12: Minimal solution for the example program.

3.7 Collection Classes

So far, we have described the algorithm in [17] It can find typings for many different kinds of programs: polymorphic methods, recursive methods, and late binding pose no problem. It does, however, have a fatal flaw that renders it next to useless in a practical context: each instance variable has only a single associated type variable. This means that *collection classes* cannot be typed in a useful manner.

Suppose that we have a class List, which has an instance variable head containing the first element of a list. If we want to use both a list of integers and a list of booleans, then we could create two separate instances by means of List new. One instance would only contain integers, and the other only booleans. In the type analysis, however, we only have a single type variable $[\![\mathsf{head}]\!]$ which will attain the type $\{\mathsf{Int}, \mathsf{Bool}\}$. Consequently, the algorithm will surmise that both List instances contains a mixture of integers and booleans, and prevent e.g. the sending of a succ message to the head of the integer list.

Fortunately, we can extend the algorithm to handle such cases, also. The idea is again to obtain more type variables by code duplication. A manual solution to the above problem would be to define two trivial subclasses of List, called IntList and BoolList. Through the expansion of inheritance this would create two new type variables for head. If the rest of the program kept these separate, then one could attain the type $\{\mathsf{Int}\}$ and the other the type $\{\mathsf{Bool}\}$. Our extended algorithm applies this strategy universally throughout the program. For every syntactic occurrence of C new, we create a copy of the entire class C. This will produce many more type variables, and in the worst case cause a further quadratic increase in the size of the program.

This technique allows very liberal typings of collection classes. We have yet to encounter a useful example where the improved algorithm falls short. Furthermore, we are still working with completely untyped programs. In fact, the distinction between syntactic occurrences of new expressions also improves typings of many programs that do not relate to collection classes—or even have instance variables.

4 The Implementation

A naive implementation of the type inference algorithm would construct the possibly large trace graph, then extract an even larger set of type constraints, and finally compute the minimal solution. The size of the intermediate results alone makes this infeasible; the collection of global constraints is in general worst-case exponential in the size of the program.

Our approach to efficiently implementing the algorithm embodies three major ideas, to be explained in this section. First we show how to combine the three steps of the algorithm and incrementally compute the graph, the constraints, and the solution. This avoids representing intermediate results and unreachable parts of the trace graph. The result is a polynomial time algorithm. Then we introduce *naming schemes* and *constraint templates* to avoid costly recomputation and storage of local constraints in different nodes for the same method. Last, we present a data structure that compactly represents constraints and the so far computed minimal solution. In the final subsection, we give some performance measurements of our implementation and state how it can be obtained by anonymous ftp.

4.1 Incremental Graph Construction

We can combine the three steps of the algorithm because of the following observation.

> **Observation:** Suppose we are given a set of *unconditional* inclusions, one by one. Suppose also that we in each step compute the minimal solution to the inclusions given so far. Then these minimal solutions will increase monotonically for each step. This implies that if a condition is satisfied at any point, then it will also be satisfied in the final solution (if such a solution exist).

This observation justifies an *incremental* construction of the trace graph, as follows. Starting in the main node, we only follow edges whose condition is true. Because true conditions remain true, we never need to "undo" the decision of following an edge, and we need never record any conditions. Every inclusion met along the way is inserted into a data structure Solver which always contains the minimal solution to the inclusions contained in it.

> **Terminology:** A *front edge* is an unprocessed, outgoing edge from a visited node.

The computation stops when no front edges are left. This means that possibly large parts of the trace graph need not be constructed, let alone traversed, see figure 13.

This would be especially important in a typical SMALLTALK environment which often has a very large number number of classes, many of which have methods with the same name. As an example we can look at the trace graph for the program (Set new) add: 3 in such a typical SMALLTALK environment. It will resemble the structure in figure 14 because of the many different classes implementing a method add:.

But since the only condition which can be satisfied is Set $\in$ [[Set new]] it would be wasteful to follow all the other edges and generate conditional constraints which will never come

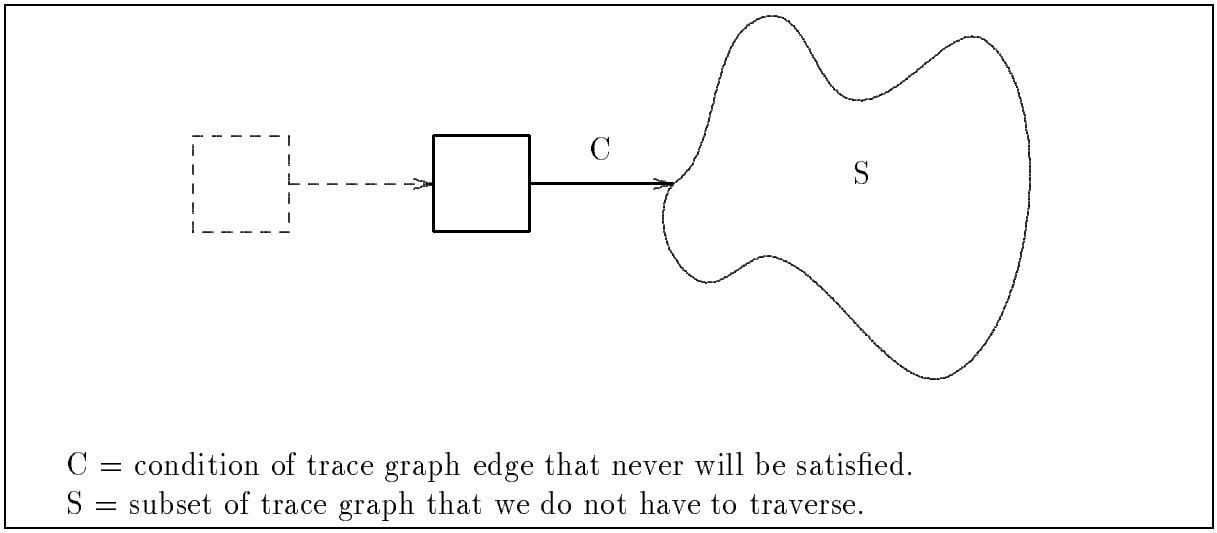


Figure 13: Avoiding unreachable parts of the trace graph.

into play. Furthermore, there is no need to visit a node more than once. Since we no longer generate *conditional* constraints, subsequent visits to a node can add nothing new.

Our main data structure Solver represents front edges, constraints, and their minimal solution which is continuously updated. There are three operations on the data structure, see figure 16.

Note that the trace graph is only represented through its front edges and minimal solution. In a later subsection we describe an efficient implementation of the constraint aspect of this data structure. Using the Solver we can rephrase the algorithm, see figure 15. This is clearly a polynomial time algorithm, since every node and edge will be visited at most once, and every visit takes at most linear time. Recall also that the trace graph is polynomial in the size of the program.

4.2 Naming Schemes and Constraint Templates

We represent a program as a parsetree augmented with:

- A *unique number* for each node, called a PTN (Parse Tree Number); and
- A *constraint template* for each method node.

As explained below, this information yields a unique naming of all type variables and it avoids costly recomputation and storage of local constraints in different nodes for the same method.

To uniquely identify type variables, we use the following naming scheme. Recall that there are as many trace graph nodes representing a method as there are syntactic message sends to it. This means that we can identify a trace graph node by a pair of PTNs:

$$(\text{PTN for message send}, \text{PTN for method})$$

Furthermore, recall that each expression needs a fresh type variable in each copy of the method in which it appears. This means that we can also identify type variables by a

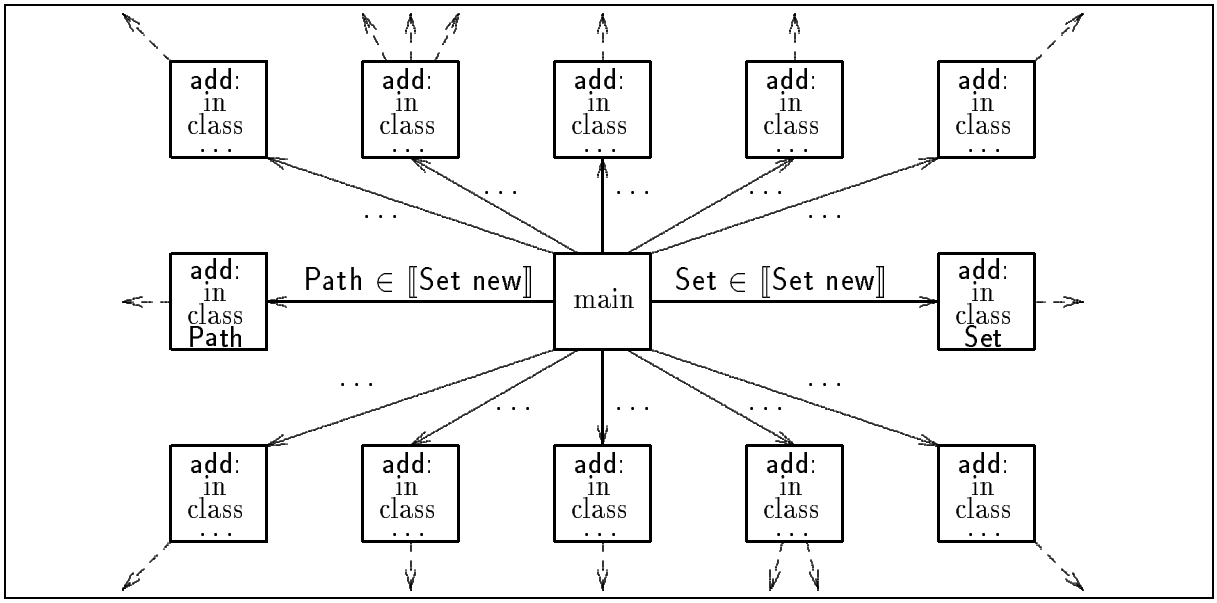


Figure 14: Part of trace graph for (Set new) add: 3.

pair of PTNs:

$$(\text{PTN for message send}, \text{PTN for expression})$$

The only exception is the instance variables which has just one type variable. The extension of the algorithm to handle collection classes is simply that trace graph nodes will now be represented by a quadruple of PTNs:

$$(\text{PTN for message send}, \text{PTN for } \mathsf{new}, \text{PTN for method}, \text{PTN for } \mathsf{new})$$

The two extra PTNs identify the new expressions that generate new versions of the original components. This simplicity of this modification shows the flexibility of the current implementation.

We avoid recomputing the local constraints in different nodes for the same method by using a *constraint template*. The template is computed the first time it is needed, and then merely instantiated later on. Instantiation inserts the appropriate PTN of the message send to the corresponding method.

We have described two data structures used by the implementation: the augmented parse tree and the Solver. The following subsection indicates how to efficiently implement the constraint aspect of the Solver.

4.3 Constraint Representation

As pointed out before, all of the unconditional constraints can be expressed with just three different kinds of inclusions: 1) constant $\subseteq$ variable, 2) variable $\subseteq$ constant, or 3) variable$_1$ $\subseteq$ variable$_2$.

To store these inclusions in an efficient way we give them an ordering. This is achieved by associating the first type of inclusion with the variable on the right hand side, the second

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 away inheritance.

2)
```
Solver add-constraint: <local constraints of the main node>
Solver add-front-edge: <trace graph edges from the main node>
while <more front edges with satisfied condition> do
    e := Solver get-front-edge    "where e goes from node m to node n"
    Solver add-constraint: <connecting constraints of edge e>
    if not <n seen before> then
        Solver add-constraint: <local constraints of node n>
        Solver add-front-edge: <trace graph edges from node n>
    end if
end while
```

3) If a solution to the constraints exists, then output it as the wanted type information, together with a safety guarantee; otherwise, output "unable to type the program".

Figure 15: The algorithm rephrased.

Data structure Solver

add-constraint: Adds a constraint to the constraint set. If there is no solution to the extended constraint set, then an error is returned.

add-front-edge: Adds a front edge to the set of front edges.

get-front-edge: Removes a front-edge whose condition is satisfied with respect to the minimal solution of the current constraint set.

end Solver

Figure 16: The Solver data structure.

and third type of inclusions with the variable on the left hand side. Now we can "store" each inclusion together with its associated type variable.

A type variable is represented by an object of the following form.

```
assignment          : Set of Classes;
constant-constraint : Set of Classes;
variable-constraint : Set of Type Variables;
```

Instead of storing each constraint individually we use the three components of the above object to keep track of the effects of the three different types of inclusions. The relationship

$$C_1 \subseteq V, \ldots, C_n \subseteq V \Leftrightarrow C_1 \cup \ldots \cup C_n \subseteq V$$

shows how to keep track of kind 1) inclusions via the assignment set. That is, for each kind 1) inclusion the assignment set is updated by assignment := assignment ∪ constant.

Likewise

$$V \subseteq C_1, \ldots, V \subseteq C_n \Leftrightarrow V \subseteq C_1 \cap \ldots \cap C_n$$

shows how to keep track of kind 2) inclusions via the constant-constraint set by updating it like constant-constraint := constant-constraint ∩ constant. Finally to deal with kind 3) inclusions we use the variable-constraint set which is updated as variable-constraint := variable-constraint ∪ variable2.

Only two things are left to do.

- If the assignment or constant-constraint change, then make sure that assignment ⊆ constant-constraint. If this is not the case, then the constraint system has no solution and the program is not typable.
- If the assignment changes, then propagate the value of assignment to all the variables in variable-constraint.

The methods of the type variable object consequently look as follows:

```
method subset-of-constant: classset
    constant-constraint := constant-constraint ∩ classset;
    if (assignment ⊈ constant-constraint) then <couldn't type program>

method subset-of-variable: variable
    variable-constraint := variable-constraint ∪ variable;
    variable superset-of-constant: assignment

method superset-of-constant: classset
    if (classset ⊈ assignment) then
        assignment := assignment ∪ classset;
        if (assignment ⊈ constant-constraint) then <couldn't type program>
        foreach variable in variable-constraint do
            variable superset-of-constant: assignment
```

The add-constraint: operation of the Solver can now easily be implemented using these three methods on type variables.

The sets of classes are implemented as bit vectors for efficiency of set operations. The sets of type variables are implemented as sets of pointers to type variable objects. The different type variable objects are stored in a hash table so that when given a type variable name we can quickly locate the corresponding type variable object.

The functionality of this system bares some resemblance to "trigger functions" in databases in the way that a change in the assignment of a type variable "triggers" a chain of events which reestablishes the soundness of the total assignment. In this way the solution to the system of constraints is always up to date.

There are at least two major advantages to this way of implementing the constraint solver. First of all the addition of a constraint to the constraint system and the solution of the expanded constraint set is very simple and requires little work. That is, it requires no

search through lists or other structures but has immediate access to the relevant data. Secondly even though a lot of redundant constraints are created in the first place these have no adverse effects since the individual constraints are not stored.

4.4 Performance Evaluation

With this implementation we have achieved a dramatic speed-up and realistic running times. Figures 17 and 18 contain some experimental data from a G++ implementation on a SPARC. The columns show the name of the program, its size in number of lines, the total number of paths in the trace graph, the much smaller number of edges used by the implementation, and the running time in milliseconds.

PROGRAM	LINES	PATHS TOTAL	EDGES USED	TIME MSEC
Peano integers	125	224247	154	234
Search trees	170	>2000000000	202	434
Container class	30	DOES NOT TYPE CHECK		

Figure 17: The basic algorithm.

PROGRAM	LINES	PATHS TOTAL	EDGES USED	TIME MSEC
Peano integers	125	>>5000000	630	1367
Search trees	170	>>2000000000	321	1134
Container class	30	11	7	17

Figure 18: The extended algorithm.

The implementation is flexible in that the user can selectively specify which classes should be treated as collection classes; this is done by writing Collection Class rather than just Class in the program text. In figure 17 *none* of the classes are treated as collection classes. This corresponds to running the previous algorithm from the OOPSLA'91 paper [17]. Note that one of the programs cannot be type checked with this algorithm. In figure 18 *all* classes are treated as collection classes. This gives slower running times, but better typings; for example, the program "Container class" can now be type checked. All our programs use laborious encodings of integers and booleans; if these are replaced by SMALLTALK-style primitive classes, then running times will be vastly improved.

The implementation is available by anonymous ftp at hyperion.daimi.aau.dk in the directory /pub/palsberg/inference. It contains files checkSparc.Z, checkSun3.Z, and checkHP.Z that are compressed executables for various architectures. Some example programs are also available, together with a README file containing instructions. The appendix shows the analysis of two programs in our example language. The first is an implementation of the Peano integers that we have adopted from a TYPED SMALLTALK program provided

by Ralph Johnson. The second is an adoptation of a program provided by Justin Graver. It uses collection classes and is typable under the extended algorithm, but not under the basic one. The analysis recognizes that the programs are safe, and produces versions with type annotations in two different styles.

5 The Perspectives

An efficient implementation of the present algorithm has perspectives beyond merely type inference in the traditional sense. A host of useful tools for the SMALLTALK programmer can become available. This section describes how some of these with little effort can be obtained from the basic algorithm.

5.1 Safety Tool

As mentioned earlier, if a program is typable, then the error message-not-understood can never occur in any execution. This is an iron-clad guarantee that would be a useful asset for any finished product. Furthermore, this guarantee is obtained in a completely automatic fashion, at no cost to the programmer.

5.2 Image Compression Tool

A SMALLTALK image grows over time, and may contain classes defined for numerous unrelated tasks. This makes it cumbersome for the programmer to create a stand-alone executable version of a program, where unneeded code is left out. However, a by-product of the inferred typing information can show how to discard superfluous code—with a guarantee that nothing essential will be missing. First of all, any class that does not appear in the value of some type variable can be discarded. Furthermore, and less obviously, individual methods in the remaining classes may also be discarded. Consider any message send with selector m. The inferred type of the receiver mentions a set of classes that all implement a method named m. These methods are marked as *live*. When this has been done for all message sends, then some unmarked methods will remain. These can safely be removed, since we are sure that they will never be invoked. Note that this technique of image compression is useful also for typed object-oriented languages such as C++ [23] and Eiffel [16].

5.3 Code Optimization Tool

When a safety guarantee is issued, then the run-time checks for message-not-understood can be left out. This simplifies the code for dynamic method lookups. More significantly, the inferred type of a receiver corresponds closely to the information contained in a polymorphic in-line cache (PIC) employed by Hölzle, Chambers, and Ungar [13] to greatly improve efficiency. This information approximates the set of all possible non-nil values to which the receiver expression may evaluate in any execution of the program. Our

inferred types yield sets that are slightly too large. Using PICs one obtain sets that are slightly too small. Smaller sets will result in smaller target code, but in the case of a cache-miss the PIC technique pays the price of dynamic re-compilation or, alternatively, a dynamic lookup. Our technique avoids cache-misses altogether. Possibly a merge of the two approaches would be optimal.

5.4 Annotation Tool

As seen in the appendix, the inferred type information can be used to annotate the original program. This can improve readability and also serve as a debugging tool. Various styles of annotation can be based on the current algorithm. Conversely, the algorithm could accept annotated programs; the annotation provided by the user would impose further constraints. In this view, an annotation is a rudimentary correctness predicate that is automatically verified by the algorithm.

5.5 Hierarchy Construction Tool

The analysis of collection classes could lead to suggested changes in the class hierarchy, e.g. the introduction of classes such as IntList and BoolList. Furthermore, inferred subtype relationships—or the lack of same—may be used to criticize the class hierarchy suggested by the programmer.

5.6 Check Insertion Tool

If the bounds of a safety guarantee appear too restrictive in some applications, then the local constraint **2)** can be abandonded. Instead the inferred type of a receiver can be examined to determine if the message send is safe. If not, then a dynamic check can be inserted before the method lookup (similarly to the qua checks of SIMULA [6] and BETA [15]). This selective approach may greatly reduce the number of required checks, compared to SIMULA and BETA implementations. Note that a dynamic check can either be inserted implicitly in the compiled code, or explicitly in the program text, in the manner of a program transformation.

6 Conclusion

We have taken two important steps towards making type inference practical. First, our previous algorithm has been improved to handle collection classes in a useful manner; this was formerly the most severe limitation that we suffered. The current version of the algorithm is able to handle realistic programs. Second, the algorithm has been implemented in a quality that promises real-life applicability. A technique of incremental computation has achieved a dramatic speed-up compared to a naive implementation.

Also, we have indicated that a number of practical tools can be based on our type inference algorithm. We believe that such tools can form the basis for a design methodology that

supports rapid prototyping as well as the evolution towards a mature product.

Acknowledgement: This work has been supported in part by the Danish Research Council under the DART Project (5.21.08.03).

Appendix: Examples

This appendix shows the programs "Peano integers" and "Container class", and the output when they are given to the type inference implementation. For the peano integers program we have chosen an option so that only method headings are annotated. For the container class program we have chosen another option so that also message sends are annotated.

Peano Integers

```
class Zero
  method isZero
    True new
  method plus: aNumber
    aNumber
  method minus: aNumber
    aNumber negative
  method negative
    self
  method decrementUntilNonzeroWhileIncrementing: negativeInteger
    negativeInteger
  method plusNegative: negativeInteger
    negativeInteger
  method plusPositive: positiveInteger
    positiveInteger
  method differenceFrom: aNumber
    aNumber
  method incr
    (PositiveInteger new) oneMoreThan: self
  method decr
    (NegativeInteger new) oneLessThan: self
end Zero


class NegativeInteger inherits Zero
  var incr
  method oneLessThan: aNumber
    incr := aNumber;
    self
  method isZero
    False new
  method plus: aNumber
    aNumber plusNegative: self
  method minus: aNumber
    aNumber differenceFrom: self
  method negative
    (PositiveInteger new) oneMoreThan: (incr negative)
  method plusNegative: negativeInteger
    (negativeInteger decr) plus: incr
  method plusPositive: positiveInteger
    (positiveInteger decr) plus: incr
  method differenceFrom: aNumber
    aNumber plus: (self negative)
  method incr
    incr
end NegativeInteger


class PositiveInteger inherits Zero
  var decr tempn1 tempn2 tempp
  method oneMoreThan: aNumber
    decr := aNumber;
    self
  method isZero
    False new
```

```
  method plus: aNumber
    aNumber plusPositive: self
  method minus: aNumber
    aNumber differenceFrom: self
  method negative
    (NegativeInteger new) oneLessThan: (decr negative)
  method decrementUntilNonzeroWhileIncrementing: negativeInteger
    tempn1 := negativeInteger incr;
    if (tempn1 isZero) isTrue
    then decr
    else decr decrementUntilNonzeroWhileIncrementing: tempn1
  method plusPositive: positiveInteger
    (positiveInteger incr) plus: decr
  method plusNegative: negativeInteger
    tempn2 := negativeInteger;
    tempp := self;
    self while1;                            % Unfortunately we have no loops
    if (tempn2 isZero) isTrue then tempp else tempn2
  method while1
     if ((tempn2 isZero) or: (tempp isZero)) isTrue
     then nil
     else ( tempn2 := tempn2 incr; tempp := tempp decr; self while1 )
  method differenceFrom: aNumber
    (aNumber decr) minus: decr
  method decr
    decr
end PositiveInteger

% Booleans

class Object
end Object

class True
  method isTrue
    Object new
  method not
    False new
  method and: aBoolean
    aBoolean
  method or: aBoolean
    self
  method xor: aBoolean
    aBoolean not
end True

class False
  method isTrue
    nil
  method not
    True new
  method and: aBoolean
    self
  method or: aBoolean
    aBoolean
  method xor: aBoolean
    aBoolean
end False

class Main
  var n
  method go
    n := ((Zero new decr) plus: (Zero new incr));
    n minus: n negative;
    n plus: n;
    n decrementUntilNonzeroWhileIncrementing: n
end Main

(Main new) go
```

Annotated Peano Integers

```
Time for initializing parsetree: 49.998msecs
Program is typable.
Time for finding solution: 1366.61 msecs
Number of edges used in the graph: 630
class Zero
  method differenceFrom: aNumber
    {PositiveInteger} -> {PositiveInteger}
    {NegativeInteger} -> {NegativeInteger}
  method negative
    {Zero}
  method incr
    {PositiveInteger}
  method plusNegative: negativeInteger
    {NegativeInteger} -> {NegativeInteger}
  method decrementUntilNonzeroWhileIncrementing: negativeInteger
    {Zero,NegativeInteger,PositiveInteger} -> {Zero,NegativeInteger,PositiveInteger}
  method plus: aNumber
    {Zero,NegativeInteger,PositiveInteger} -> {Zero,NegativeInteger,PositiveInteger}
  method isZero
    {True}
  method decr
    {NegativeInteger}
  method plusPositive: positiveInteger
    {PositiveInteger} -> {PositiveInteger}
  method minus: aNumber
    {Zero,NegativeInteger,PositiveInteger} -> {Zero,NegativeInteger,PositiveInteger}
end Zero

class NegativeInteger
  var incr {Zero,NegativeInteger}
  method differenceFrom: aNumber
    {PositiveInteger} -> {PositiveInteger}
    {NegativeInteger} -> {Zero,NegativeInteger,PositiveInteger}
  method oneLessThan: aNumber
    {Zero,NegativeInteger} -> {NegativeInteger}
  method negative
    {PositiveInteger}
  method incr
    {NegativeInteger}
    {Zero}
  method plusNegative: negativeInteger
    {NegativeInteger} -> {NegativeInteger}
  method decrementUntilNonzeroWhileIncrementing: negativeInteger
    {Zero,NegativeInteger,PositiveInteger} -> {Zero,NegativeInteger,PositiveInteger}
  method plus: aNumber
    {Zero,NegativeInteger,PositiveInteger} -> {Zero,NegativeInteger,PositiveInteger}
  method isZero
    {False}
  method decr
    {NegativeInteger}
  method plusPositive: positiveInteger
    {PositiveInteger} -> {Zero,NegativeInteger,PositiveInteger}
  method minus: aNumber
    {Zero,NegativeInteger,PositiveInteger} -> {Zero,NegativeInteger,PositiveInteger}
end NegativeInteger

class PositiveInteger
  var decr {Zero,PositiveInteger}
      tempn1 {Zero,NegativeInteger,PositiveInteger}
      tempn2 {Zero,NegativeInteger,PositiveInteger}
      tempp {Zero,NegativeInteger,PositiveInteger}
  method differenceFrom: aNumber
    {PositiveInteger} -> {Zero,NegativeInteger,PositiveInteger}
    {NegativeInteger} -> {NegativeInteger}
  method negative
    {NegativeInteger}
  method incr
    {PositiveInteger}
```

```
  method plusNegative: negativeInteger
    {NegativeInteger} -> {Zero,NegativeInteger,PositiveInteger}
  method decrementUntilNonzeroWhileIncrementing: negativeInteger
    {Zero,NegativeInteger,PositiveInteger} -> {Zero,NegativeInteger,PositiveInteger}
  method plus: aNumber
    {Zero,NegativeInteger,PositiveInteger} -> {Zero,NegativeInteger,PositiveInteger}
  method isZero
    {False}
  method decr
    {Zero,PositiveInteger}
  method plusPositive: positiveInteger
    {PositiveInteger} -> {PositiveInteger}
  method minus: aNumber
    {Zero,NegativeInteger,PositiveInteger} -> {Zero,NegativeInteger,PositiveInteger}
  method while1
    {}
  method oneMoreThan: aNumber
    {Zero,PositiveInteger} -> {PositiveInteger}
end PositiveInteger

class Object
end Object

class True
  method and: aBoolean
  method isTrue
    {Object}
  method not
  method or: aBoolean
    {True,False} -> {True}
  method xor: aBoolean
end True

class False
  method and: aBoolean
  method isTrue
    {}
  method not
  method or: aBoolean
    {True,False} -> {True,False}
  method xor: aBoolean
end False

class Main
  var n {Zero,NegativeInteger,PositiveInteger}
  method go
    {Zero,NegativeInteger,PositiveInteger}
end Main

{Zero,NegativeInteger,PositiveInteger}
```

Container Class

```
class Natural
  method isZero
    nil
end Natural

class Boolean
  method isTrue
    nil
end Boolean

class Container
  var x
  method put: val
    x := val
  method get
```

```
    x
end Container

class Main
  var a b
  method go
    a := Container new;
    a put: Natural new;
    (a get) isZero;
    b := Container new;
    b put: Boolean new;
    (b get) isTrue
end Main

(Main new) go
```

Annotated Container Class

```
Time for initializing parsetree: 16.666msecs
Program is typable.
Time for finding solution: 16.666 msecs
Number of edges used in the graph: 7
class Natural
  method isZero
    {}
end Natural

class Boolean
  method isTrue
    {}
end Boolean

class Container
  var x {Natural,Boolean}
  method get
    {Natural}
    {Boolean}
  method put: val
    {Natural} -> {Natural}
    {Boolean} -> {Boolean}
end Container

class Main
  var a {Container}
      b {Container}
  method go
    {}
end Main

{}
```

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] Andreas V. Hense. Wrapper semantics of an object-oriented programming language with state. In T. Ito and A. R. Meyer, editors, *Proc. Theoretical Aspects of Computer Software*, pages 548–568. Springer-Verlag (*LNCS* 526), 1991.

[13] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proc. ECOOP'91, Fifth European Conference on Object-Oriented Programming*, pages 21–38, 1991.

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

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

[17] 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.

[18] 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.

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

[20] 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.

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

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

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

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

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

[26] 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.