Compile-Time Analysis of Object-Oriented Programs

Jan Vitek, R. Nigel Horspool and James S. Uhl

Depatrment of Computer Science
University of Victoria *

Abstract. Generation of efficient code for object-oriented programs requires knowledge of object lifetimes and method bindings. For object-oriented languages that have automatic storage management and dynamic look-up of methods, the compiler must obtain such knowledge by performing static analysis of the source code. We present an analysis algorithm which discovers the potential classes of each object in an object-oriented program as well as a safe approximation of their lifetimes. These results are obtained using abstract domains that approximate memory configurations and interprocedural call patterns of the program. We present several alternatives for these abstract domains that permit a trade-off between accuracy and complexity of the overall analysis.

1 Introduction

The object-oriented approach to programming has become an accepted programming paradigm, joining other paradigms such as imperative, functional and relational programming. This new paradigm is normally associated with the concepts of *class, method* and *inheritance*. Different object-oriented languages implement these concepts with varying degrees of dynamic behaviour.

At one extreme, SMALLTALK makes every feature as dynamic as possible. Instances of a class are created dynamically which, when no longer referenced, are garbage collected automatically. Messages are implemented by dynamic binding of call site to method implementation. In general this means that a run-time search through the inheritance hierarchy to find the appropriate method is required for every message send. Finally, SMALLTALK is dynamically typed, there are no type declarations and methods are type-checked at run-time.

At the other extreme, C++ implements as much as possible in a static manner so that C++ compilers can generate efficient code. In particular, memory is allocated and de-allocated explicitly by the programmer, methods are statically typed, and message sends are bound statically under programmer control.

In an ideal world, an object-oriented language would be as dynamic as SMALLTALK and as efficient as C++. The language would provide dynamic features, but a compiler would analyze the code and determine whether or not these features are used. The user would then only pay a run-time penalty when, and where, the language is used in a truly dynamic manner.

* P.O. Box 3055, Victoria, BC, Canada V8W 3P6. {jvitek, nigelh, juhl}@csr.uvic.ca

Short of the ideal, we believe there are ways to improve over existing compiler technology. For instance, *static program analysis* provides a compiler with the information needed to distinguish between static and dynamic portions of a program. In particular, we have been investigating compile-time analysis techniques to determine the class(es) of each object in an object-oriented program. This knowledge should permit the following compiler optimizations.

- Static binding of message sends to particular method implementations.
- Compile-time type checking of some method parameters.
- In-line expansion of method bodies.

The in-lining optimization is particularly relevant to object-oriented languages where programs tend to be composed of many classes containing very short method definitions. In practice, in-lining wins big. Experiments have shown that with no in-lining SELF programs would run between 4 and 160 times slower [2]. With minor changes, our analysis can be used to infer lifetimes of objects and therefore shift some of the garbage collection overhead from run-time to compile-time.

For our research we have invented a "typical" language with (single) inheritance, no type declarations of variables, untyped dynamically bound methods and automatic memory management. The use of this language is not a restriction. We expect our techniques to be widely applicable to compilers for existing languages.

The remaining sections of this paper review past work in the area and give a general overview of our approach, briefly introduce the small object-oriented language, and then describe the analysis algorithm and abstract domains.

2 Related Work and Overview

The difficulty in optimizing object-oriented programs lies in the lack of type information. Without precise information on the *class* of the object to which a message is sent, it is difficult to determine the effect of the message and to bind it statically. (Note that, static typing, as implemented in EIFFEL, helps little since a type in EIFFEL consists of a class along with *all* of its subclasses.)

Previous research specific to optimizing object-oriented programs has focused on providing some form of useful type information to the compiler [13, 10, 11, 7]. This research was pioneered by Suzuki [13] who first separated the concepts of type and inheritance. In his framework any set of classes defines a type. Smaller sets are more informative: a singleton set represents the case when the object has a unique class, and an empty set indicates an error—there is no possible class for the object. Types are then inferred by a unification algorithm.

Several extensions to this framework were proposed to correct some of the shortcomings of the original algorithm [10, 11, 7]. Johnson proposed parameterized types so that many common programs using objects with polymorphic instance variables could be typed [10]. Palsberg and Schwartzbach developed a type inference algorithm that takes the context of each message send into account to derive sharper types [11]. Although the algorithm is limited to the analysis of complete programs, their results seem to be the most accurate to date.

These algorithms have a number of shortcomings which cause unsatisfactory results for many simple programs. One source of inaccuracy is that all variables are treated as global, so that an assignment to the instance variable of one object affects all other objects of the same class. Another problem is that the `nil` object (empty reference) is essentially swept under the rug, by either ignoring it entirely or treating it as a special case. Finally, all of the above algorithms enforce a type-discipline on the programs to ensure a degree of static type-safety. (This approach is inherently limited by the possibility of sending a message to `nil`.)

We propose a data flow analysis algorithm [1], parameterized with analysis domains, that provides abstract representations for program state and interprocedural call patterns. The technique is similar to abstract interpretation [4]. In our case, we draw on work done in the field of compiler optimization for our state abstraction [3, 9] and call pattern abstraction [12, 6, 2]. Specifically, our algorithm differentiates among instances of the same class and therefore maintains greater precision with respect to the type of their instance variables. Empty references can be detected; the algorithm can provide *must/may* information for `nil` objects. Aliasing is implicitly taken into account by the state abstraction (as in [3]).

As with most abstract interpretation or data flow analysis frameworks, the accuracy of the results depends on the complexity of the abstract domains. Therefore, we present two related domains for abstracting program state. The first is relatively unsophisticated but should produce results rapidly. The second extends the first and yields more accurate results. With both domains, any approximations are conservative in nature. When our algorithm, with either domain, derives a set of classes for an object at a particular point in the program, that set is guaranteed to either be exact or be a superset of the exact answer. At worst, the compiler will be unable to optimize away all unnecessary method look-ups and run-time type-checks.

There are similarities between our static analysis technique and the SELF technology [2]. But the ultimate goals are different. In SELF compilation is done at run-time, so efficiency of the compiler is of paramount importance. For this reason the SELF compiler does not perform interprocedural analysis, nor does it keep track of program state.

3 An Object-Oriented Language

The simple untyped object-oriented language defined in this section is used to illustrate the analysis presented in subsequent sections. It should be noted that this language is biased towards compilation. In particular, run-time modification of classes (as in [5]) and of the inheritance hierarchy (as in [2]) is not allowed, and neither messages nor class identifiers may be manipulated at run-time.

The syntax of the language is shown in Figure 1, using the following notational conventions: curly brackets represent optional constructs, superscript plus signs indicate one or more occurrences of the preceding construct, and italicized text represents non-terminals.

The semantics are typical of class-based object-oriented languages in which dynamic object creation, automatic storage reclamation and single inheritance play a major role. A *class* is similar to a record or structure type in an imperative language

$$
\begin{array}{lll}
\mathit{program} & ::= & \{\, \mathit{class}^{+} \,\}\ \mathit{main} \\
\mathit{main} & ::= & \textbf{main}\ \{\, \textbf{var}\ \mathit{id}^{+} \,\}\ \mathit{stat}^{+} \\
\mathit{class} & ::= & \textbf{class}\ \mathit{classid}\ \{\, \textbf{inherits}\ \mathit{classid} \,\}\ \{\, \textbf{var}\ \mathit{id}^{+} \,\}\ \{\, \textbf{methods}\ \mathit{method}^{+} \,\} \\
\mathit{method} & ::= & \mathit{methodid}\ \{\, (\mathit{id}) \,\}\ \{\, \textbf{var}\ \mathit{id}^{+} \,\}\ \textbf{is}\ \mathit{stat}^{+} \\
\mathit{stat} & ::= & \mathit{id}\ :=\ \mathit{exp}\ \mid\ \textbf{if}\ \mathit{exp}\ \textbf{then}\ \mathit{stat}^{+}\ \textbf{else}\ \mathit{stat}^{+}\ \mid \\
 & & \quad \textbf{while}\ \mathit{exp}\ \textbf{do}\ \mathit{stat}^{+}\ \mid\ \textbf{return}\ \mathit{exp}\ \mid\ \mathit{exp} \\
\mathit{exp} & ::= & \mathit{id}\ \mathit{methodid}\{\, (\mathit{id}) \,\}\ \mid\ \textbf{self}\ \mid\ \mathit{classid}\ \textbf{new}\ \mid\ \mathit{id}\ \mid\ \mathit{constant} \\
\mathit{constant} & ::= & \textbf{nil}\ \mid\ \textbf{true}\ \mid\ \textbf{false}\ \mid\ \textbf{0}\ \mid\ \textbf{1}\ \mid\ \ldots
\end{array}
$$

Fig. 1: Syntax of a simple object-oriented language

where the fields consist of: data fields called *instance variables*, and named function pointers, called *methods*. The instance variables of an object may be accessed only through the object's methods. A class `Y` may *inherit* from another class `X`, using the **`inherits`** clause, which includes all of `X`'s data fields in `Y`, and a pointer to a list of `X`'s methods. `Y` is said to be a *subclass* of `X`, and, similarly, `X` is said to be a *superclass* of `Y`. Not including `X`'s methods directly allows them to be *over-ridden* in `Y`, which is the essence of inheritance.

An object is manipulated by sending it a *message*, which consists of a name and an argument, if appropriate. A message send causes a search of the object's methods for a matching name. In the case of failure, superclass method tables are searched in order, starting at the immediate superclass, until either a match is found or no such method exists. If the search is successful, the corresponding function is called with the message's argument, along with a hidden argument named *self* which is bound to the object receiving the message. If the search fails, the program is erroneous and a run-time error is signalled.

A new object, or instance, of class `C` is created by `C` **`new`**. All instance variables of a new object are initialized to the value **`nil`**. Predefined constants in the language include numbers, truth values, **`nil`**, along with binary arithmetic and logic primitives such as $+$ and $\geq$. The language does not include the **`super`** construct, which allows direct access to inherited methods, because such method invocations can always be bound statically at compile time.

4 A Model of Programs and State

In our toy language a program is defined as a set of class definitions and a main procedure. Before beginning analysis, two transformations are applied to the program. The inheritance hierarchy is flattened, as in [11], which "fills" in all inherited methods with the appropriate superclass methods. Also, all variables and formal arguments are given unique names. To preserve semantics, instance variables inherited from a superclass retain the same name.

When a program is executed, its state consists of a set of constants and active objects referring to each other through their instance variables. We find it convenient to view this structure as a graph of unbounded size in which nodes stand for objects or constants and arcs represent the value of instance variables. Labels distinguish the arcs originating from a node; there is one label per instance variable of the

corresponding object. Constants are nodes with no outgoing arcs. Activation records for methods are regarded as objects and therefore appear as nodes in the object graph. Their local variables are treated as instance variables.

Definition 1. Let p be a program and L (labels) a set of variable names. An *object graph* is a pair $\langle N, A \rangle$, where N is a finite set of objects (nodes) and $A \subseteq N \times N \times L$ is a finite set of instance variables (labeled arcs).

During static analysis a program, is represented by a *control flow graph* and an *abstract object graph*. Building the control flow graph for our analysis differs only slightly from building the interprocedural control flow graph in an imperative language [12]. In particular, a message send has arcs leading to and returning from all methods whose selector matches the message. Message sends are split into two nodes, corresponding to the call and return portions, the former binding arguments, the latter binding the return result if the statement is part of an assignment. Finally, an extra exit node is added to every method and to the main program.

5 Analysis with Abstract Objects

The state of an object-oriented program at any point in its computation is captured by the instantaneous states of all the objects in the computer's memory. At some point in a particular computation, this state is unique and can be represented by an object graph. Static analysis, however, examines textual points in the program and there may be many different computations that cause control to reach a particular textual point. Thus, each textual point is associated with a set of possible execution states and, accordingly, we consider the type of an object at a textual point to be the set of classes that the object can assume over all possible executions.

For each textual point in the program, we construct an *abstract object graph* (AOG) summarizing all memory states occurring at this point during some program execution. Since the AOG models the memory structure of the program it keeps track of the value of instance variables, local variables, and possible aliasing relations among them.

Definition 2. An AOG is a pair $\langle N, A \rangle$ where N is a finite set of abstract objects, $A \subseteq N \times N \times L$ is a finite set of arcs and L is a set of labels. The summary nature of the AOG entails that multiple, identically labeled, arcs can originate from the same node.

An AOG typically contains *may* information. An arc labeled **xy** originating at node **x** and ending at node **y** means that at run-time the value of instance variable **xy** of the object corresponding to **x** *may* refer to **y**. We later modify this representation to include *must* information with the addition of creation counts.

Definition 3. The type of a variable is a set of AOG nodes. Let $n \in N$, $l \in L$ and $A \subseteq N \times N \times L$. The type of variable l of node n in $\langle N, A \rangle$ is $\{ m \mid \langle n, m, l \rangle \in A \}$

The set of abstract objects, N, is syntactically derived from the program being analysed. This set, a program wide constant, remains static over the analysis whereas the set of arcs, A, is dynamic.

Definition 4. Let p be a program, N its set of nodes and L its variable names. The abstract object graphs of p, $AOG_p = \{\langle N, A\rangle \mid A \subseteq N \times N \times L\}$, form a complete lattice. Let $g_1, g_2 \in AOG_p$ and $g_1 = \langle N, A_1\rangle, g_2 = \langle N, A_2\rangle$. The partial order relation, $g_1 \sqsubseteq g_2$, is defined as set inclusion of the set of arcs, $A_1 \subseteq A_2$. The lattice meet operation, $g_1 \sqcup g_2 = \langle N, A_1 \cup A_2\rangle$. The least element, $\bot$ is the disconnected graph, $\langle N, \{\}\rangle$, and the greatest element, $\top$, is the fully connected graph, $\langle N, N \times N \times L\rangle$.

Note that we require N to be finite, but that the object graphs summarized by an AOG are unbounded. This apparent contradiction is solved by allowing nodes to summarize many objects *of the same class.* With this approximation we can limit the size of the graphs without unnecessary loss of type information. The choice of mapping from object graphs to AOGs is crucial to the efficiency and precision of the analysis. We now present two abstract domains effecting different mappings.

5.1 Class Object Graphs

Objects of the same class are likely to be used in the same way in a program, i.e. their instance variables are likely to refer to objects of the same type. Based on this observation it seems reasonable to define a domain where objects of the same class are represented by a single node in the graph.

Definition 5. A *class object graph* (COG) maps all dynamically created instances of a class onto a single graph node.

In a COG, finite sets of constant objects, such as the logical values `true` and `false` are represented by nodes in the graph, one for each constant. Infinite, or large, sets of constants are abstracted so that one graph node represents all values in the set. For our toy object-oriented language the constant nodes are: `true, false, nil` and `Int`.

The following program is used to illustrate the analysis with COGs. For the sake of clarity, we omit the abstract object nodes for methods in the discussion.

```
class Point
  var x, y
  methods
    set-x(v) is
      x := v
    set-y(v) is
      y := v
```

```
main
  var p, p' is
    p := Point new
    p' := Point new          -G1
    if read-int = 0
      then p set-x(1)        -G2
      else p set-y(p')
           p' set-y(0)       -G3
                             -G4
```

For this program, $N = \{Point, nil, Int\}$ with *Point* being the COG node representing objects created at the two textual instances of `Point new`. The sequence of graphs inferred by static analysis is shown in Figure 2. G_1 describes the state after creation of the two `Points`, their instance variables refer to *nil*. G_2 portrays the

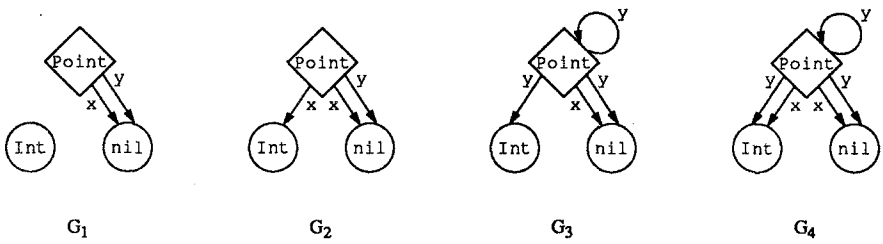

Fig. 2: A sequence of class object graphs

state if the **then**-branch of the **if** is traversed. The arc x from *Point* to the integer node *Int* appears as a result of the assignment to instance variable **x** of **p**. G_3 shows the effect of taking the **else**-branch. Now **y** refers to an integer or to a point; the self-loop appeared because we have mapped both **Points** onto the same node and thus lost all distinction between them. The same graph could represent a recursive structure, e.g. a linked list, where **y** is the next pointer.

The result of the analysis is the COG G_4, obtained by taking the meet of G_2 and G_3. At this point we know that **x** has type $\{Int, nil\}$, and that **y** has type $\{Int, Point, nil\}$.

COGs have been used extensively to analyze object-oriented programs [10, 11, 13]. Unfortunately, the polymorphism inherent to the object-oriented style of programming proves to be a serious problem since objects of the same class often are used with instance variables of different classes[10].

5.2 Textual Object Graphs

To improve the accuracy of static analysis we alter the analysis domain so that different objects of the same class do not necessarily share the same node in the graph. Only objects created at the same textual point share abstract representations.

Definition 6. A *textual object graph* (TOG) maps all instances of a class created at the same textual point in a program onto the same graph node.

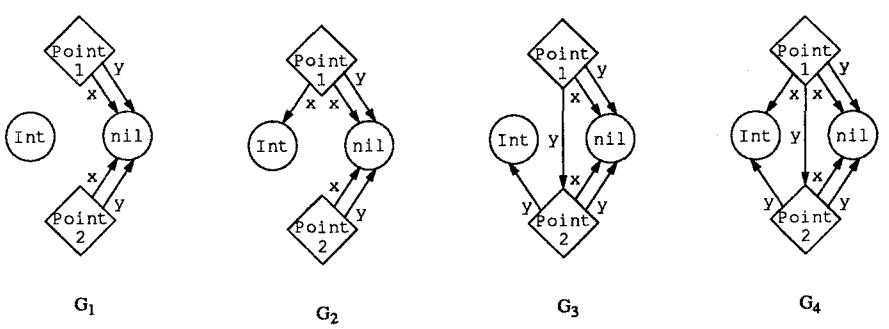

Fig. 3: Textual object graphs

In our previous example, there were two textual creations of `Point`, one for `p` and one for `p'`; the TOG associates a graph node to each: $Point_1$ to `p` and $Point_2$ to `p'`. Therefore $N = \{Point_1, Point_2, nil, Int\}$ and the analysis infers the sequence of graphs of Figure 3. G_4 clearly shows that there are no recursive structures in this program and that the type of `y` is different for `p` and `p'`.

In general, this abstract representation yields better results than those obtained with COGs. Also, the types given to instances of "container" classes, e.g. lists, is more accurate and does not require special treatment as in [10].

5.3 Creation Counts

A problem common to both domains is that assignment is defined as an additive operation—arcs are never removed. For example, the arcs connecting each node to `nil` never disappear from the graph. A compiler using the results of the analysis will have to assume that every instance variable of an object can always refer to `nil`. Due to this "sticky" behaviour, the quality of the analysis will only deteriorate as the number of possible values increases. This problem, and specifically `nil`-valued instance variables, has marred previous attempts to provide a type system for object-oriented programs [10, 11, 13]. (Some authors have defined type-safety so that programs may legally send messages to `nil` [11].)

The work of Chase *et al.* [3] points to a simple solution. The key observation is: if it can be determined that an assignment will affect all the objects represented by a single node in the graph (*must* information), then it is possible to update the graph destructively. In the analysis of the example program with the TOGs, all assignments have this property since each node represents only one object. This is the static criterion for destructive update. (A node may represent many objects if the creation point is in a loop, or if it occurs in a method that is called more than once along some control flow path.) We note in passing that a node with no incoming arcs represents objects which are not referenced and, therefore, can be statically de-allocated. The graph model is extended in the following way.

Definition 7. An *extended abstract object graph* is a triple $\langle N, A, C\rangle$, where $\langle N, A\rangle$ is an AOG and $C = N \times \{0, 1, \infty\}$ maps nodes to their creation counts. $\{0, 1, \infty\}$ is totally ordered by the relation $0 < 1 < \infty$. We define the commutative addition operator $\oplus$ to be

$$[add] \qquad 0 \oplus x = x \qquad\qquad 1 \oplus 1 = \infty \qquad\qquad \infty \oplus x = \infty$$

and the substraction operator $\ominus$ to be

$$[sub] \qquad 0 \ominus x = 0 \qquad\qquad 1 \ominus 1 = 0 \qquad\qquad \infty \ominus x = \infty$$

Every node starts with a count of 0. Each time a node's creation point is encountered along a path in the program, the count is incremented.

When two control flow paths meet, the greatest creation count is retained for each node in the graph. Note that although creation counts can be applied to COGs, the gains are negligible since it is likely that the counts for all classes will quickly converge to ∞.

5.4 AOG Operations

We now define abstract operations corresponding to assignment, object creation and control flow merge. These definitions apply to both textual object graphs and class object graphs.

Merging of graphs, used for confluence of control flow paths, is implemented by the lattice meet operation which is simply the union of the arcs and the pointwise maximum of creation counts. For a given N, and graphs $g_1 = \langle N, A_1, C_1 \rangle$ and $g_2 = \langle N, A_2, C_2 \rangle$, the merge of g_1 and g_2 is:

$$[merge] \quad g_1 \sqcup g_2 = \langle N, A_1 \cup A_2, C' \rangle \quad \textbf{where } C' = \{ \langle n, max(c_1, c_2) \rangle \mid \langle n, c_1 \rangle \in C_1, \langle n, c_2 \rangle \in C_2 \}$$

Assigning a set of objects S to an instance variable l of n updates the arcs of the corresponding graph node, and is written $(g|\langle n, l \rangle \mapsto S)$. If the creation count of n is not ∞ then the update replaces the current set of arcs labeled l with S, otherwise S is simply added to the existing arcs. Thus, for a given N, and graph $g = \langle N, A, C \rangle$ the operation is:

$$[assign] \quad (g|\langle n, l \rangle \mapsto S) = \textbf{if } \langle n, \infty \rangle \in C \textbf{ then } \langle N, A \cup \mathcal{N}ew, C \rangle \textbf{ else } \langle N, \mathcal{U}pd \cup \mathcal{N}ew, C \rangle$$
$$\textbf{where } \mathcal{N}ew = \{ \langle n, m, l \rangle \mid m \in S \},$$
$$\mathcal{U}pd = \{ \langle n', m', l' \rangle \in A \mid n \neq n', l \neq l' \}$$

Creation of an object is modelled by adding arcs pointing to `nil` to the corresponding node and increasing the creation count of n. The function `labels` returns the set of instance variable names of a node. For a given N, and graph $g = \langle N, A, C \rangle$ the operation is:

$$[create] \quad create(g, n) = (\ldots(\langle N, A, C' \rangle | \langle n, l_1 \rangle \mapsto \{\texttt{nil}\}) \ldots | \langle n, l_k \rangle \mapsto \{\texttt{nil}\})$$
$$\textbf{where } C' = \{ \langle n, c \oplus 1 \rangle \mid \langle n, c \rangle \in C \} \cup \{ \langle n', c' \rangle \in C \mid n' \neq n \},$$
$$\{l_1, \ldots, l_k\} = \texttt{labels}(n)$$

6 A Flow Analysis Algorithm

Programs are analyzed using conventional flow analysis [1]. The first step is to construct a control flow graph representation of the program. Each arc in this graph is assigned a flow function which computes the effect of executing the code in its source node, assuming that its target node is to be executed next. Finally, all control flow nodes are assigned initial values in the lattice, and the flow functions are applied to the values until a fixpoint is reached.

In the abstract object graph, methods and the main program are treated as abstract objects with outgoing arcs for each variable; in the case of methods, the graph contains arcs for `self`, the argument, and the return result. These objects correspond to activation records that may be created at run-time. The creation count associated with such nodes is initially set to zero, incremented on entry to the method and decremented on exit.

We now describe the flow functions associated with a given control flow arc a. Assume that the source node of a is in the control flow subgraph corresponding to method $\mathtt{m_0}$. The flow functions take the abstract object graph $\langle N, A, C\rangle$ as argument and return an updated AOG.

6.1 Flow Functions for Assignment Statements

Assigning a value to a variable involves finding all abstract objects that may be affected and updating their arcs accordingly. Here we consider assignments to instance variables; the treatment of local variables is similar.

When `value` is assigned to an instance variable, `ivar`, it potentially affects all objects that may be bound to `self`. Let $\mathcal{S}elves = \{\, n \mid \langle m_0, n, \mathtt{self}\rangle \in A \,\}$ be this set. The general form of a flow function for an arc leaving such an assignment is:

$$\lambda\langle N, A, C\rangle.\ \bigsqcup_{n \in \mathcal{S}elves} (\langle N, A, C\rangle | \langle n, \mathtt{ivar}\rangle \mapsto \mathcal{V}alues)$$

where $\mathcal{V}alues$ is the set of abstract objects corresponding to `value`. The functions for the various types of assignable values are as follows:

`value`	$\mathcal{V}alues$
`constant`	$\{const\}$, where *const* is the AOG corresponding to `value`
`local variable`	$\{\, n \mid \langle m_0, n, \mathtt{value}\rangle \in A \,\}$
`instance variable`	$\{\, n \mid \langle s, n, \mathtt{value}\rangle \in A, s \in \mathcal{S}elves \,\}$

If the value is a newly created object, the creation count for the creation point must be updated prior to updating the arcs. Let cp be the AOG node corresponding to the creation point, the flow function is:

$$\lambda\langle N, A, C\rangle.\ \bigsqcup_{n \in \mathcal{S}elves} (\mathit{create}(\langle N, A, C\rangle, cp) | \langle n, \mathtt{ivar}\rangle \mapsto \{cp\})$$

6.2 Flow Functions for Message Sends

Message sends involve creation of AOG arcs representing the binding of parameters and locals on entry to, and removal of those arcs on exit from, the called method. The implicit argument `self` is bound to the set of possible receivers that match the class of the called method. An extra local variable, `result`, is used to hold the return result; its value is required if the message send is an assignment statement.

Let $\mathcal{A}rguments$ and $\mathcal{PR}eceivers$ be the sets of arguments and possible receivers, respectively. They are determined in the same manner as $\mathcal{V}alues$ above. Now consider a flow graph arc corresponding to a call from m_0 to some method implementation m_1, the latter having a single local variable `local`. Let $\mathit{classof}(o)$ be the function returning the class of the abstract object o. The flow function for this arc is:

$$\begin{aligned}[call]\quad \lambda\langle N, A, C\rangle.((((\langle N, A, C'\rangle,\ &|\langle m_1, \mathtt{self}\rangle \mapsto \mathcal{R}eceivers) \\ &|\langle m_1, \mathtt{formal}\rangle \mapsto \mathcal{A}rguments) \\ &|\langle m_1, \mathtt{local}\rangle \mapsto \{\mathtt{nil}\}) \\ &|\langle m_1, \mathtt{result}\rangle \mapsto \{\mathtt{nil}\})\end{aligned}$$

$$\begin{aligned}&\textbf{where } C' = \{\, \langle m_1, c \oplus 1\rangle \mid \langle m_1, c\rangle \in C \,\} \cup \{\, \langle n', c'\rangle \in C \mid n' \neq m_1 \,\}, \\ &\mathcal{R}eceivers = \{\, r \mid r \in \mathcal{PR}eceivers, \mathit{classof}(r) = \mathit{classof}(m_1) \,\}\end{aligned}$$

For the return flow arc, assume the result is assigned to the instance variable **ivar**. The resulting flow function is:

$$
\begin{aligned}
[return]\ \lambda\langle N, A, C\rangle.\ & \bigsqcup_{n \in Selves} (g' | \langle n, \mathtt{ivar}_1\rangle \mapsto \mathcal{R}esult) \\
\textbf{where } g' = (\langle N, A, C'\rangle & | \langle m_1, \mathtt{self}\rangle \mapsto \{\}) \\
& | \langle m_1, \mathtt{formal}\rangle \mapsto \{\}) \\
& | \langle m_1, \mathtt{local}\rangle \mapsto \{\}) \\
& | \langle m_1, \mathtt{result}\rangle \mapsto \{\}), \\
C' = \{\, \langle m_1, c \ominus 1\rangle \mid \langle m_1, c\rangle \in C \,\} & \cup \{\, \langle n', c'\rangle \in C \mid n' \neq m_1 \,\}, \\
\mathcal{R}esult = \{\, r \mid \langle m_1, r, \mathtt{result}\rangle \in A \,\} &
\end{aligned}
$$

6.3 Flow Analysis

Performing the analysis is now straightforward. The flow value stored at each node in the control flow graph represents what is known about the abstract objects and variables at entry to the node. All are initially assigned $\perp$. These values are then iterated through the functions on the arcs, with the meet operator applied when control flow paths merge. This process is continued until a fixpoint is reached. In the algorithm below, the function **pred** returns the set of predecessors of n and $f_{i,n}$ is the flow function on the arc from i to n.

FOREACH *node n in the control flow graph* **DO**
  $n := \perp$
REPEAT
  FOREACH *node n in the control flow graph* **DO**
    $n := n \sqcup (\bigsqcup_{i \in \mathtt{pred}(I)} f_{i,n}(i))$
UNTIL *a fixpoint is reached.*

7 More Precise Analysis with Call Strings

Because the effect of a method depends strongly on its calling environment (*i.e.* the type of the receiver and of the arguments), analysis information from different call sites should be kept separate during the analysis. We propose tagging the abstract object graphs with the sequence of call-points (control flow graph call nodes) through which they were propagated. The abstract state of the program at a point inside method m becomes a set of pairs $\langle$*call-path*, $g\rangle$, where g represents an instance of the abstract object graph and *call-path* is the sequence of method call points that brought control to m.

Since static analysis requires finite representations and call chain sequences for methods cannot, in general, be bounded, we represent the call-path by a finite approximation. One possibility is a regular expression whose elements are method call points, similar to the abstract interpretation described in [8]. The alternative described here approximates a call path by the last k calls in the sequence [12], for

some small value of k. All paths which are identical in the last k calls will have their state information merged thus creating some approximation in the results. The choice of k determines a trade-off between accuracy and the computational effort of performing the analysis. Most traditional interprocedural data flow analysis algorithms [1] implicitly use $k = 0$, and [11] use essentially the same method with $k = 1$.

The extended analysis algorithm is a generalization of the one given earlier. At any textual point p in the program, the algorithm derives a set of $\langle \textit{call-path}, g \rangle$ pairs that describes the possible object states and corresponding call strings that may hold at that point in the program. If the statement at that point is any operation other than a method invocation, the g component of each $\langle \textit{call-path}, g \rangle$ pair is updated as in the previous analysis method.

Assume that we are currently analyzing method m_0, the program point p is a message send and m_1 is an implementation of that message. The state description set at this point is S and f_{p,m_1} is the flow function defined in the previous section. The flow function on the arc from p to m_1 appends p to all call-strings and applies the abstract object graphs to f_{p,m_1}.[1]

$$[\textit{call}] \qquad \lambda S.\{\, \langle s +\!\!+ p, f_{p,m_1}(g) \rangle \mid \langle s, g \rangle \in S \,\}$$

At the end of analysis of a method, the state description set must be propagated back to each applicable invocation point. If p' is the next node in m_0 and m_1' is the last point in m_1, then the flow function on the arc from m_1' to p' prefixes the symbol '?', the unknown call point, to each call string and shortens them to length k by dropping their suffix.

$$[\textit{return}] \qquad \lambda S.\{\, \langle (? +\!\!+ s) \uparrow k, f_{m_1',p'}(g) \rangle \mid s[k] = p, \langle s, g \rangle \in S \,\}$$

The start node of m_1 is the target of multiple control flow arcs. Let $\mathcal{I}n$ be the set of all incoming $\langle \textit{call-path}, g \rangle$ pairs. The call strings in this set are shortened to length k by dropping the earliest call point in the sequence and the TOG values in pairs with identical call strings are merged.

$$[\textit{entry}] \qquad \{\, \langle s, g \rangle \mid g = \bigsqcup \{\, g' \mid s = s' \downarrow k, \langle s', g' \rangle \in \mathcal{I}n \,\} \,\}$$

At p', there will in general be several description sets propagated back from different implementations of the message. The resulting description sets, $\mathcal{R}es$, must be combined with description sets that describe the program state immediately before the call, S to remove '?' from the call strings, and all pairs with identical call-strings are merged.

$$[\textit{resume}] \qquad \lambda S.\{\, \langle a +\!\!+ s, g \rangle \mid g = \bigsqcup \{\, g' \mid \langle ? +\!\!+ s, g' \rangle \in \mathcal{R}es, \langle a +\!\!+ s, g'' \rangle \in S \,\} \,\}$$

[1] The ith element of *string* s is written as $s[i]$. The concatenation of strings s and t is denoted by $s +\!\!+ t$. A prefix of length n of a string x is denoted by $x \uparrow n$ while a suffix of length n is denoted by $x \downarrow n$.

```
class Point                    main
  var x, y                  1.  var p is
  methods                   2.    p := Point new
    set-x(v) is             3.    p set-x(true)
A.    x := v                4.  if read-int = 0
    set-y(v) is             5.    then p set-x(1)
B.    y := v                6.    else nil
```

Fig. 4: A Sample Program for Analysis

We now present a trivial example to clarify the analysis algorithm and the use of call strings. The sample program for analysis is shown in Figure 4. To simplify the description, the statements in the main body are numbered 1 through 6 and the two executable statements inside the Point class are lettered A and B. We use the name p_i to represent a program point immediately before statement i, and p'_i to represent a program point immediately after statement i. With this naming convention, several program points coincide (e.g. $p'_2 = p_3$) and we will pass over such duplicates in the explanation.To keep the example as simple as possible, we will limit the length of call strings to 1. The analysis algorithm starts with an initial state description at statement 1 of $\langle ?, \perp \rangle$ where '?' is a call-string of length 1 that indicates that the caller is unknown (the caller is actually the operating system), and $\perp$ represents the bottom graph in the TOG lattice. Pictorially, $\perp$ is the disconnected graph drawn as graph G_0 in Figure 6. The analysis algorithm might produce the sequence of state description sets shown in Figure 5.

make a pass over main program:

$Point\ p_1 : \{\langle ?, G_0 \rangle\}$
$Point\ p_2 : \{\langle ?, G_1 \rangle\}$
$Point\ p_3 : \{\langle ?, G_2 \rangle\}$
$Point\ p_4 : \{\langle ?, G_2 \rangle\}$
$Point\ p_5 : \{\langle ?, G_2 \rangle\}$
$Point\ p'_5 : \{\langle ?, G_2 \rangle\}$
$Point\ p_6 : \{\langle ?, G_2 \rangle\}$
$Point\ p'_6 : \{\langle ?, G_2 \rangle\}$

make a pass over method set-x:

$Point\ p_A : \{\langle p_3, G_3 \rangle, \langle p_5, G_5 \rangle\}$
$Point\ p'_A : \{\langle p_3, G_4 \rangle, \langle p_5, G_6 \rangle\}$

propagate results back to call points and repeat analysis as required:

$Point\ p_4 : \{\langle ?, G_7 \rangle\}$
$Point\ p_5 : \{\langle ?, G_7 \rangle\}$
$Point\ p'_5 : \{\langle ?, G_8 \rangle\}$
$Point\ p_6 : \{\langle ?, G_7 \rangle\}$
$Point\ p'_6 : \{\langle ?, G_9 \rangle\}$

repeat pass over method set-x

$Point\ p_A : \{\langle p_3, G_3 \rangle, \langle p_5, G_5 \rangle\}$

... and nothing more changes

Fig. 5: Analysing the program of Fig. 4

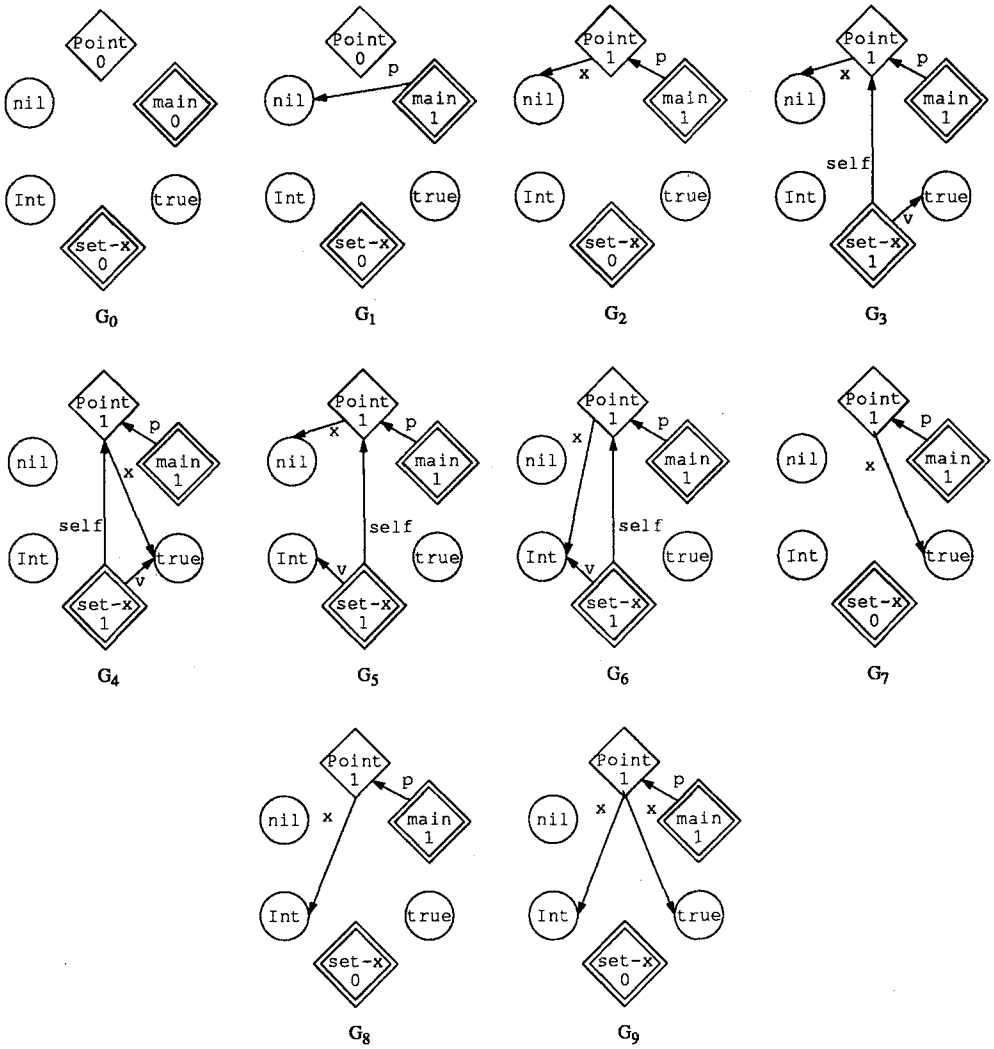

Fig. 6: Abstract Object Graphs. Creation counts are displayed under node names.

8 Discussion

Previous work in analysis of object-oriented languages has produced some relatively imprecise analysis techniques. We have therefore developed better techniques. Furthermore, we provide a means for the compiler implementer to choose an appropriate trade-off between precision and the cost of the analysis.

Separate compilation remains a challenge for any form of interprocedural static analysis. On one hand we would like to keep program units separate and minimize the re-compilation effort. On the other hand we need to be able to analyze as much as possible of the source program to generate efficient code. Research towards a *modus vivendi* continues.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.
2. C. D. Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Languages.* Ph. D. Thesis, Stanford University, 1992.
3. D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of Pointers and Structures. In *SIGPLAN'90 Conf. on Programming Language Design and Implementation*, 1990.
4. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Program by Construction or Approximation of Fixpoints. In *Conf. Rec. of the 4th ACM Symp. on Principles of Programming Languages*, 1977.
5. A. Goldberg and G. Robson. *Smalltalk-80 – The Language and its Implementation.* Addison-Wesley, 1983.
6. L. Harrison. The Interprocedural Analysis and Automatic Parallelization of Scheme Programs. Technical Report CSRD Rpt. 860, University of Illinois, Urbana, Ill., 1989.
7. A. V. Hense. Type Inference for O'small. Technischer Bericht A 06/91, Universität des Saarlandes, 1991.
8. R. N. Horspool and J. Vitek. Static Analysis of PostScript. In *Proceeding of the International Conference on Computer Languages*, 1992.
9. P. Hudak. A Semantic Model of Reference Counting and its Abstraction. In *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.
10. R. E. Johnson. Type-checking Smalltalk. In *OOPSLA'86 Conf. Proc.*, 1986.
11. J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *OOPSLA'91 Conf. Proc.*, 1991.
12. M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In S. S. Munchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.
13. N. Suzuki. Inferring Types in Smalltalk. In *Conf. Rec. of the 8th ACM Symp. on Principles of Programming Languages*, 1981.