The Design of Program Analysis Services

Robert O'Callahan
June 1999
CMU-CS-99-135

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The difficulty of understanding large programs is a major contribution to the cost of program maintenance and transformation. Tools for static program analysis promise to address this problem by automatically discovering truths about a program from its source text. It is often desirable to have a suite of specialized tools that each address a different task; however, static analyses are difficult and expensive to build, and therefore should be reused across tools wherever possible. Therefore we desire a general interface between analyses that produce information and tools that consume it. It should be easy and cheap to build tools on top of such an interface, with no need to understand the analysis implementation; however, the interface must permit scalability and precision comparable to a monolithic design. It turns out to be desirable, and feasible, for the same interface to cover a wide range of analysis implementations. In this paper we discuss the design and implementation of such an interface in Ajax, a framework and set of tools for the static analysis of Java programs.

This research is sponsored in part by the Defense Advanced Research Projects Agency and the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, F33615-93-1-1330, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0031 and in part by the National Science Foundation under Grant No. CCR-9523972. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The author was supported by a graduate fellowship from Microsoft Corporation.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency Rome Laboratory, the U.S. Government, or Microsoft Corporation.

Keywords: Java, bytecode, Ajax, value-point relation, tools, interface

1 Introduction

We are building a set of semantics-based static analysis tools for program understanding and maintenance. Each tool is specialized to a particular task. To make the development of these tools cost-effective, we must make it easy to reuse static analyses across tools. This requires that as little as possible of the complexity of the analyses be exposed to the tool designer. In our system for analyzing Java code, Ajax, we define an interface that tools use to access program analysis services.

In the remainder of this paper, we discuss the requirements for the Ajax analysis interface and our basic approach to addressing these requirements. We present the "value-point relation", an abstraction that is the basis for communicating semantic information through our interface. Tools request views on this relation; we describe the queries that generate these views, and how tools and analyses cooperate through the interface to obtain optimizations necessary for scalability. We give some preliminary results suggesting that with this new abstraction we can still perform scalable analysis of real Java programs.

2 The Value-Point Relation

Overview

We wish to hide the complexity of static analyses from tools. The interface between tool and analysis should be simple and small, not only in the code, but also conceptually. For example, many alias analyses map the potentially infinite set of run-time memory locations into a finite set of "abstract locations", and compute results as sets or relations over these abstract locations. If these abstract locations are exported to tools, then the tool writer must understand what they are and how they are used. This represents conceptual and implementation overhead. Therefore we specify an interface to analyses simply and directly in terms of the semantics of Java.

We define a static abstraction of all possible program executions, called the "value-point relation". This abstraction is the basis for transmitting information from the analysis to the tool. The relation encodes abstract "flow" information that we have found suitable for many different kinds of tools. It is the core of the Ajax interface; conceptually, an analysis computes a conservative approximation to the true value-point relation, and a tool filters and postprocesses this approximation in response to user requests.

A "value-point" is an expression e at some location l in the program code. The expression need not occur in the program text; thus in general there are an infinite number of value-points. Given a value-point (e, l) and a program state s where the program counter is at l, we can evaluate the expression in that state to yield a runtime value v, in which case we say that (e, l) evaluates to v in state s. If s is a state where the program counter is not at l, then we say that the value-point does not evaluate to any value in state s. In Java, the set of runtime values consists of all object references, booleans, and all the scalar values such as ints, floats, etc. We always evaluate expressions as "rvalues"; e.g. the value of "`a.x`" is the current contents of `a`'s field `x`, not the location of field `x` (as if "`a.x`" appeared on the left hand side of an assignment).

Two value-points VP_1 and VP_2 are *related* if there is a execution of the program that passes through two states s_1 and s_2 such that VP_1 evaluates to a value v at s_1 and VP_2 evaluates to the same value v at s_2. For example, if the values of VP_1 and VP_2 are always pointers (or object references), then VP_1 and VP_2 are related iff they alias, i.e. iff the resulting pointers (or references) are ever the same.

We formally define the value-point relation for Java bytecode programs, but it seems readily adaptable to other languages.

Definition

We write the value-point relation associated with a program p as $\leftrightarrow_p$. Usually the program p is implied and we simply refer to the value-point relation as $\leftrightarrow$.

First we present a preamble, declaring an abstraction of the Java bytecode language sufficient to define $\leftrightarrow$.

Let S be the set of states of Java bytecode programs. Let S_0 be the initial state. Let $\Rrightarrow_p$ be the transition relation that executes a single step of p (usually, an instruction); $\Rrightarrow_p$ relates S to S. ($\Rrightarrow_p$ need not be a function, i.e. nondeterminism is allowed.) Let L be the set of program locations; in Java bytecode, L is the set of pairs each consisting of a fully qualified method name and a bytecode instruction offset. We treat the program p as a partial map from L to bytecode instructions. Let PC be a function from S to L, that returns the "current program location" of each state, the location of the next instruction to be executed. Let V be the universe of Java run time values (object references, integers, floats, etc).

We formally define a trace t of a program p as follows: a trace t is a finite sequence of length $|t|$ of states $\langle s_i \rangle$, such that $s_0 = S_0$ and for all $0 < i \leq |t|$, $s_{i-1} \Rrightarrow_p s_i$.

The bytecode language does not have a built in notion of expressions; it uses a RISC-like instruction format. (See the example in Figure 1.) Therefore we define our own family of "bytecode expressions". Ajax has bytecode expressions to compute the values of the currently executing method's local variables and working stack elements. Ajax also has expressions for global variables (i.e. static fields), and the nonstatic fields of other bytecode expressions. To simplify the presentation, here we only consider expressions that give the values of elements of the working stack.

Therefore let V_{Stack} be a partial function from $S \times \mathbb{Z}$ to V, such that $V_{Stack}(s, n)$ gives the value of working stack element n in state s (V_{Stack} is partial because the stack varies in height over time). We choose the convention that stack element 0 is the top of the stack. We define a bytecode expression e to be of the form "stack-n", where n denotes the index of the stack element to be retrieved. As discussed in the overview, a value-point VP is a pair (e, l) with a program location and a bytecode expression to be evaluated at that location. For example, in Figure 1, (stack-0, #5) is a bytecode expression.

Now we can define the evaluation partial function $\hookrightarrow$, that evaluates a value-point in the context of a program state.

$(s, (\text{stack-}n, l)) \hookrightarrow v$ iff $PC(s) = l$ and $V_{Stack}(s, n) = v$

This implies that the evaluation is only defined if the location of the value-point matches the current location of the program state, as suggested in the overview above.

For example, in Figure 1, $(s, (\text{stack-}0, \#5)) \hookrightarrow v$ iff $PC(s) = \#5$ and $V_{Stack}(s, 0) = v$. The value at the top of the stack when the program counter is at location #5 is always the value pushed on at location #4, i.e. $v = 100$.

Finally we can define the value-point relation $\leftrightarrow_p$:

$b_1 \leftrightarrow_p b_2$ iff there exists a trace $t = \langle s_i \rangle$ of the program p, integers i, j and a value v such that $(s_i, b_1) \hookrightarrow v$ and $(s_j, b_2) \hookrightarrow v$.

Usage example: finding aliases

Consider a tool in which the user specifies an expression that reads from a field of an object, and the tool displays all program points that may assign to the field of the object. The first step is to locate the "getfield" bytecode instruction that the user selected; suppose it has location *r* and retrieves field *f*. Then the tool computes the following set:

{ *w* | (stack-0, *r*) $\leftrightarrow_p$ (stack-1, *w*) and *p*(*w*) = "putfield *f*" }

The locations *w* are mapped back to source code and highlighted.

The idea is that the values corresponding to (*r*, stack-0) are the object operands of the "getfield" instruction at *r*. The values corresponding to (*w*, stack-1) are the object operands of the "putfield" instruction at *w* ("putfield" takes its object operand in stack-1 and the value to be stored in stack-0). We find locations *w* where these object references are the same.

Example

Consider the following program. This is not a real bytecode program, having been edited for brevity. In particular, in a real program, this code would be the body of one method.

#0: #1:	`new Foo` `store x`	x = new Foo;
#2: #3:	`new Foo` `store y`	y = new Foo;

#4:
#5:
#6:
	`load x`
	`push 100`
	`putfield Foo.p`
		x.p = 100;

#7:
#8:
#9:
	`load y`
	`push 42`
	`putfield Foo.p`
		y.p = 42;

#10: #11:	`load x` `getfield Foo.p`	x.p;
#12: #13:	`load y` `getfield Foo.p`	y.p;

Figure 1: Example program

Suppose the user wishes to find the statements that write values into the field `p` to be returned by the statement "x.p;". The tool determines that *r* = location #11 is the getfield instruction associated with the field access `x.p`, and requests a projection of the value-point relation:
{ *w* | (stack-0, #11) $\leftrightarrow_p$ (stack-1, *w*) and *p*(*w*) = "`putfield Foo.p`" }
The values yielded by the value-point (stack-0, #11) are the object references that are dereferenced in executions of the expression.

There are only two "putfield Foo.p" instructions, so the analysis reduces the problem to deciding which of the following hold:

1. (stack-0, #11) $\leftrightarrow_p$ (stack-1, #6)

2. (stack-0, #11) $\leftrightarrow_p$ (stack-1, #9)

In both cases there is only one trace t to consider; the states $<s_i>$ of the trace are simply the states at successive instructions. The questions then reduce to:

1. Are there states s_i, s_j and a value v such that (s_i, (stack-0, #11)) $\hookrightarrow v$ and (s_j, (stack-1, #6)) $\hookrightarrow v$?
   Yes. v = the Foo created at #0.

2. Are there states s_i, s_j and a value v such that (s_i, (stack-0, #11)) $\hookrightarrow v$ and (s_j, (stack-1, #9)) $\hookrightarrow v$?
   No.

Therefore the analysis returns { #6 }.

The tool concludes that the putfield instruction at location #6 may store the value read by instruction #11, but the putfield instruction at location #9 does not. The tool will display the source statement associated with instruction #6, "x.p = 100;".

Another tool example: proving downcasts safe

Consider a tool in which the user specifies a downcast expression and the tool tries to prove that the downcast is safe. This can be reduced to identifying all the possible classes of objects that are subject to the downcast. Suppose the downcast instruction is at location d. Then the tool computes the following set:

{ c | there exists l such that $p(l)$ = "new c" and (stack-0, l + 1) $\leftrightarrow_p$ (stack-0, d) }

Here the idea is that the object at d has class c if and only if there is some instruction l that creates an object of class c and the reference to the created object shows up at d. The values corresponding to (stack-0, l + 1) are the references to the object created at l, pushed onto the working stack by the "new" instruction and available in the state of the successor instruction.

Handling "null"

In practical use, the above definition of the value-point relation has serious problems related to the "null" object reference. It is very common for a given object expression to be able to return "null" — or at least, it is hard to prove with a static analysis that it does not return "null". This means that any two expressions will often be related simply because they could both yield "null". However, tools are usually only interested in the case where the value is non-null; this is true for both of the example tools above. Therefore we modify the definition to read:

$b_1 \leftrightarrow_p b_2$ iff there exists a trace $t = <s_i>$ of p, integers i, j and a value v such that (s_i, b_1) $\hookrightarrow v$ and (s_j, b_2) $\hookrightarrow v$ and $v \neq$ null.

For example, considering the example above, suppose that "new Foo" was returned either a new Foo or else null indicating failure. With the basic definition we would have to conclude that the analysis should return { #6, #9 }, for setting v = null would show that (stack-0, #11) $\leftrightarrow_p$ (stack-1, #9) holds. The extended definition rules this out. The tool is still satisfied with the definition, since reads or writes to the null object raise runtime errors and need not be considered.

Similarly, the downcast checking tool is satisfied with the extended definition because downcasting a null reference is always safe.

Handling scalars

When the values that are being matched by the value-point relation are object references, we can assume in the semantics that each new object has a unique reference, i.e. object references are never reused. However, when the values are scalars, for example integers, then we may have many unrelated occurrences of the same value. For example, the value 0 occurs very frequently in different contexts. We would like to be able to say that two expressions that both yield 0 are unrelated because their values came from different sources. To make this possible, our semantics for Java bytecode are extended to pair every scalar value with a unique "identity tag". Each operation that "creates" a new scalar value, such as an arithmetic operation or the use of a constant, assigns it a fresh identity tag. Assignment of a scalar simply propagates the existing identity tag. In the definition of the value-point relation, the results of the value-point expressions must match in both the scalar value and the tag.

Then, for example, we could write a tool that finds all arithmetic instructions that could generate the value returned by a given value-point expression b:

$\{ l \mid (\text{stack-0}, l + 1) \leftrightarrow_p b$ and $p(l)$ is an arithmetic instruction $\}$

Without the identity tags, this would rarely be useful because most arithmetic instructions could generate any integer value and hence would match the value at b.

Directionality

The value-point relation is symmetric, so some "directional" information has been lost. However, consider the downcast safety query above. It is clear that the object reference flows from the "new" to the downcast instruction. In general, when one of the expressions specifies the creation of a value, the direction of flow is clear.

On the other hand, when neither of the expressions is related to the creation of the value, the direction of flow is usually not relevant, and there may not even be flow between the two value-points in the conventional sense. For example, in the read/write aliasing tool above, the desired "directional" constraint would require w to be executed before r in the trace. The object reference may not "flow from" w to r or vice versa at all; the reference may "flow to" both w and r from some common originating expression. It would be easy to extend the value-point relation with the "b_1 executes before b_2" constraint:

$b_1 \leftrightarrow_p b_2$ iff there exists a trace $t = \langle s_i \rangle$ of p, integers i, j and a value v such that $(s_i, b_1) \hookrightarrow v$ and $(s_j, b_2) \hookrightarrow v$ and $v \neq$ null and $i < j$.

It is not clear whether analyses could produce such fine-grained information in realistic programs.

An alternative "directional" relation would be to define "$b_1 \leftrightarrow_p b_2$" if and only if there is "flow" from b_1 to b_2, in the sense that a slice on b_2 includes b_1. However, this seems to be much more complicated to define semantically, and more difficult to use in tools other than slicing tools. The main complicating factor is the need to describe the intervening states between the execution of b_1 and b_2.

Note that although the value-point relation is symmetric, analyses that manipulate "directional" information (such as inclusion constraints) are still very useful, because they may produce better approximations to the true relation than other analyses.

Dynamic analysis

If the program has been executed, we could conceptually record information about the executions in a set of traces T. Then we could define a "dynamic value-point relation" as follows:

$b_1 \leftrightarrow_{p,T} b_2$ iff there exists a trace $t = \langle s_i \rangle$ **in** $\boldsymbol{T}$ of p, integers i, j and a value v such that $(s_i, b_1) \hookrightarrow v$ and $(s_j, b_2) \hookrightarrow v$ and $v \neq$ null.

In other words, it seems straightforward to implement the Ajax interface with a dynamic analysis, with the slight change in meaning that the results of tools would be true for a finite set of executions, rather than all executions. In particular, the same set of tools will work with either static or dynamic analysis.

3 Ajax architecture

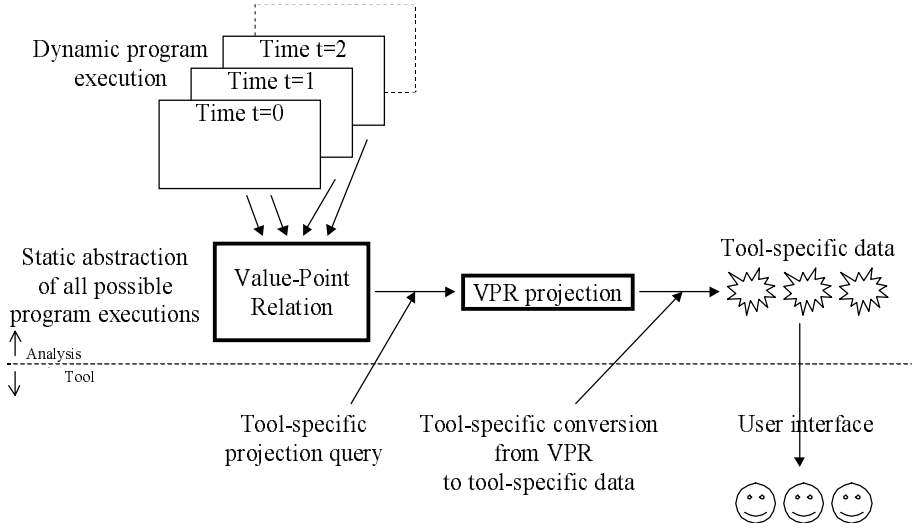


Figure 2: Ajax architecture

Figure 2 shows the architecture of Ajax, focusing on the separation between analysis and tool. The techniques used to reduce all possible dynamic behaviors to the single static abstraction of the value-point relation are beyond the scope of this paper.

A key requirement is that a tool built using Ajax must have scalability and precision comparable to a monolithic system which tightly integrates the tool and the analysis. The figure shows that the interface enables tools to request finite projections of the (infinite) value-point relation, and then specify processing to be performed on the projection. The supported projections and manipulations are described below. The tool's processing is exposed to the analysis so that the analysis can optimize its performance and precision for the task at hand.

A side benefit of using the value-point relation and keeping its definition as simple as possible is that the Ajax interface is not especially biased towards one particular analysis. In fact we have two very different implementations, one based on RTA [B97], and another which is based on type inference with polymorphic recursion [H93, O99]. Some analyses may not be precise enough to be useful in some tools, but the results will be sound.

Another way Ajax reduces complexity is to actually analyze Java bytecode instead of Java source code. The bytecode language contains almost all the relevant information from the source, but it is simpler and evolves more slowly (thanks to the large installed base of Java Virtual Machines). An additional advantage is that Ajax can analyze programs for which source code is not available. When source code is available, we need a mapping between source code and bytecode; this is just the same mapping that debuggers require, and so we use the same techniques.

4 The interface

Conservatism

Clearly the "true" value-point relation is not computable. Any static analysis must compute an approximation. A conservative approximation is any relation $\leftrightarrow$ such that $b_1 \leftrightarrow b_2$ implies $b_1 \leftrightarrow b_2$; in other words, $\leftrightarrow$ is a "super-relation" of $\leftrightarrow$. We expect any implementation of the Ajax interface to be a conservative approximation.

Queries

Tools that use the Ajax interface will not scale if the amount of data passed across the interface is quadratic (or worse) in the size of the program being analyzed. Certainly, returning the entire value-point relation is impossible.

To avoid passing the whole relation, a tool must specify a finite set of "source value-points" S and a finite set of "target value-points" T. The system returns the value-point relation projected onto S×T:
$\{ (s, t) \mid s \leftrightarrow t \wedge s \in S \wedge t \in T \}$

For example, in the alias finding tool above, we use
S = { (stack-0, *r*) }
T = { (stack-1, *w*) | *p*(w) = "putfield *f*" }

For the downcast checking tool:
S = { (stack-0, *l* + 1) | *p*(*l*) = "new *c*" }
T = { (stack-0, *d*) }

In these examples, the relation returned is at worst linear in the size of the program. However, consider a tool for checking **all** downcasts in a program. We would require:

S = { (stack-0, *l* + 1) | *p*(*l*) = "new *c*" }
T = { (stack-0, *d*) | *p*(*d*) = "checkcast *b*" }

In general the size of S × T is would be quadratic in the size of the program, i.e. not scalable.

Exposing tool-specific optimizations

The solution is to avoid returning the projected relation explicitly. Instead, the tool defines a computation to be performed over the projected relation. We hope that the internal structure of the analysis allows the computation to actually be performed in subquadratic time.

Given sources and targets S and T, we consider the analysis to compute a directed graph G with the following properties:

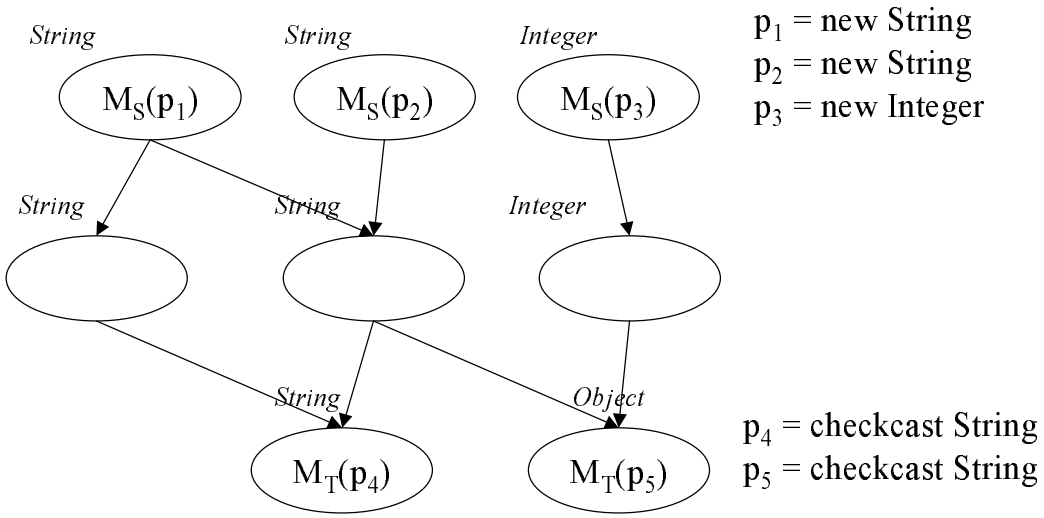
- There are functions M_S and M_T mapping the source and target value-points to graph nodes.

- $s \leftrightarrow t$ iff there is a path from $M_S(s)$ to $M_T(t)$ in G.

Obviously, given any relation $\leftrightarrow$, such a graph can be constructed. More importantly, for a wide variety of analyses it is possible to construct a graph G whose number of nodes and edges is subquadratic in the size of real programs. This is easy to see for traditional interprocedural dataflow analyses based on transfer functions, but it is also true for analyses such as RTA [B97] and type inference based analyses [S96, OJ97] which are not explicitly graph-based.

Example: proving downcasts safe

Consider the following graph computed by some analysis:



This graph encodes the relation { $p_1 \leftrightarrow p_4$, $p_1 \leftrightarrow p_5$, $p_2 \leftrightarrow p_4$, $p_2 \leftrightarrow p_5$, $p_3 \leftrightarrow p_5$ }. For example, $p_3 \leftrightarrow p_5$ is in the relation because $M_T(p_5)$ is reachable from $M_S(p_3)$, but $p_3 \leftrightarrow p_4$ is not in the relation because $M_T(p_4)$ is not reachable from $M_S(p_3)$.

p_1, p_2 and p_3 are value-points which correspond to a newly constructed String, a newly constructed String and a newly constructed Integer. p_4 and p_5 are value-points corresponding to the arguments of downcasts to Strings. $p_3 \leftrightarrow p_5$ means that the object reference that p_3 evaluates to (a newly constructed Integer) can also appear as the result of evaluating p_5, which implies that the downcast to String associated with p_5 can fail.

Now we can check the safety of the p_4 and p_5 downcasts by examining every tuple in the relation and collecting, for each downcast, the set of classes that can reach the downcast. In this case, we find that the p_1 and p_2 are related to p_4, therefore the set of classes that are downcast by p_4 is { String }, hence the downcast at p_4 is safe. On the other hand, p_1, p_2 and p_3 are all related to p_5, hence the set of classes that are downcast by p_5 is { String, Integer }; the downcast to String is unsafe.

Of course, this strategy is quadratic in the size of the program in general. However, if the graph is subquadratic in the program size, then we can do better. Instead of explicitly computing all the relation pairs, we propagate class information along the graph as shown by the italicized labels in the diagram. At each node *n*, we write the most derived common superclass of all the classes associated with M_S nodes from which *n* is reachable. Thus, node $M_S(p_3)$ is labeled "Integer" since it is only reachable from itself, and the class associated with p_3 is

Integer. On the other hand, $M_T(p_5)$ is labeled with Object, which is the most derived common superclass of { String, Integer }, the classes associated with M_S nodes from which $M_T(p_5)$ is reachable. Clearly, if the label at a node satisfies the bound on the associated downcast (as it does for p_4), the downcast is safe. Conversely, if the label at a node does not satisfy the bound on the associated downcast (as for p_5), then the downcast is unsafe (assuming that the class hierarchy is a lattice — for Java, this requires adding some elements to the class hierarchy).

The point is that these labels can be computed efficiently. We first initialize the labels on the M_S nodes to be their associated classes, then for each node *n*, we push *n*'s label along *n*'s outgoing edges. Conflicting labels are merged by taking the most derived common superclass. These updates continue until there are no changes in the labels. Since the class hierarchy is usually shallow, the number of times the label on a node can change is small, so the total number of updates (and hence the total running time of the algorithm) is roughly proportional to the size of the graph. Furthermore the space consumed is proportional to the size of the graph. Note that cyclic graphs are possible, but the algorithm still works.

Caveat

It is extremely tempting to assign an interpretation to the graph, e.g. to assume that the edges represent values "flowing" from definitions to uses. That would be a serious error of understanding. In general, the only important property of the graph is that it encodes the value-point relation correctly in the manner described. For example, in the SEMI analysis based on polymorphic recursion, the edges of the graph correspond to polymorphism constraints and say nothing about the direction of data flow.

Generalization

In general the tool specifies:

- A type D of "intermediate data" (e.g., a Java class)
- Lattice operations on this type, in particular a commutative, associative, idempotent "merge" function $D_M : D \times D \to D$ that is used to update the value at a graph node from the values at its predecessors (in the example, the lattice is the class hierarchy and the "merge" finds the most specific common superclass).
- A "bottom" element $D_\perp$ for the lattice (in the example above, this represents a special "class" that is considered a subclass of all classes, and is used to initialize C at nodes that do not correspond to any source value-points); $D_\perp$ is an identity for D_M.
- "Starting values" (elements of D) for the source value-points ($D_S : S \to D$).

The analysis propagates the intermediate values over its internal graph, and for each value-point *b* in the specified target set T, returns to the tool the D-label on node $M_T(b)$. More precisely, the analysis provides:

$\lambda b.\ D_M \{ D_S(b') \mid b \leftrightarrow b' \}$

(Extending D_M to operate on sets in the natural way.) Note that the graph G and the M_S and M_T functions are entirely private to the analysis and are not visible to the tool. They are mentioned only to explain how this computation can be implemented efficiently.

5 Evaluation

We have implemented these ideas in our Ajax system. We have two very different analyses that implement the interface. One is RTA [B97]; the other is SEMI [O99], a new analysis based on Lackwit-style polymorphic type inference [OJ97], with some extensions [H93]. We do not have space to review the details of these analyses in this paper.

We have only just started to build tools, but we already have tools for a variety of tasks, as shown in Figure 3.

As a very rough estimate of the difficulty of writing these tools, we present the number of lines of tool-specific Java code required for each tool. (Many of the tools share code, such code is counted multiple times here.) The complete Ajax system consists of about 31,000 lines of Java code.

Note that although the first four tools could all be subsumed into a "call graph tool", that tool would not be as efficient or easy to use as the four individual tools. (Just outputting a call graph could be quadratic in the size of the program.)

We have applied the tools to *javac*, Sun's Java source to bytecode compiler from JDK 1.1.7. It contains 1635 method bodies. We also analyze the library code that *javac* uses, so the total number of methods is over 2,200, in over 700 classes. Analysis times varied from less than two minutes (using RTA) to up to several hours using SEMI. SEMI is slow and could be improved, but the results are encouraging and indicate that our interface is not causing fundamental scalability problems.

6 Related work

Our work on Ajax was motivated by observations we made during our previous work on *Lackwit* [OJ97], a semantics-based program understanding tool for C code. It is a single general tool that is very powerful, but requires user expertise to map it onto particular tasks.

There have been many previous analysis frameworks, but none of them provide a simple interface insulating tools from details of analysis implementations, in the way that Ajax does.

An example of one of the most modern and powerful dataflow analysis frameworks is in the Vortex compiler [CDG96]. It is designed to simplify the implementation of new dataflow analyses, their composition, and their use in code optimization — different goals to ours. There is no attempt to

Task	**Lines of code**
Finding the live methods in the program	487
Finding methods that are called from just one call site	504
Resolving indirect call sites with a single possible callee	379
Building a call graph	603
Proving downcasts safe	546
Identifying object fields that are not both read and written	323

Figure 3: Tools

separate the production of information from tools that consume the information.

The BANE toolkit [AFFS98] takes a different approach. It is an engine for working with constraint systems used in static analysis. The client must generate constraints from the source code under analysis and interpret the solutions of the constraints. Although BANE does not provide a way to conceal these details from tools, it could be used to build implementations of the Ajax interface.

Heintze and McAllester's work on subtransitive control flow analysis [HM97] gave me the idea for propagating tool-specific intermediate values over graphs.

7 Conclusions and future work

The Ajax interface provides a useful and novel separation between analyses that produce semantic information and tools that consume it. The interface is simple and highly tool and analysis independent, yet preserves scalability and precision.

We are conducting further experiments in tool design to identify the strengths and weaknesses of the abstractions we have chosen. It seems likely that there are combinations of tools and analyses for which the value-point relation is insufficiently expressive to communicate the required semantic information. The design of more expressive, but still simple, abstractions is an interesting problem.

Of course, Ajax's fundamental purpose is to enable the construction of specialized tools for specific tasks, and we plan to exploit this capability and construct novel tools.

8 Bibliography

[AFFS98] A. Aiken, M. Fähndrich, J. Foster, Z. Su. *A Toolkit for Constructing Type- and Constraint-Based Program Analyses*, in Second International Workshop on Types in Compilation, Kyoto, Japan, March 1998

[B97] D. Bacon, *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*, PhD thesis, UC Berkeley report UCB/CSD-98-1017, 1997

[CDG96] C. Chambers, J. Dean and D. Grove. *Frameworks for Intra- and Interprocedural Dataflow Analysis*, technical report UW-CSE-96-11-02

[H93] F. Henglein. Type inference with polymorphic recursion. *TOPLAS, Volume 15, No. 2, 1993*.

[HM97] N. Heintze and D. McAllester. Linear-time subtransitive control flow analysis. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, June, 1997.

[O99] R. O'Callahan. Optimizing a Solver of Polymorphism Constraints: SEMI. CMU-CS-TR-99-136.

[OJ97] R. O'Callahan and D. Jackson, *Lackwit: A Program Understanding Tool Based on Type Inference*, Proceedings of the International Conference on Software Engeering, May 1997

[S96] B. Steensgaard. Points-to analysis in almost linear time. *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*,

January 1996.