Generalized Aliasing as a Basis for Program Analysis Tools

Robert O'Callahan

November 2000

CMU-CS-01-124

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

This research was sponsored by the National Science Foundation (NSF) under grant no. CCR9523972, the Defense Advance Research Projects Agency (DARPA) and Air Force (USAF) agreement no. F33615-93-1-1330, the Air Force Research Laboratory (AFRL) under agreement no. F306029720031, and a Microsoft Fellowship.

Keywords: alias analysis, Java, program analysis, software engineering tools, program understanding, scalability, polymorphic type inference, polymorphic recursion, object models, object oriented program analysis, SEMI, VPR

Abstract

Tools for automatic program analysis promise to improve programmer productivity by searching and summarizing large bodies of code. However, the phenomenon of aliasing — different names being used to refer to the same data — reduces the effectiveness of simple textual analyses. This dissertation describes the design of a system, Ajax, that addresses this problem by using semantics-based program analysis as the basis for a number of different tools to aid Java programmers.

To enable the construction of many tools, Ajax imposes a clean separation between analysis engines that produce alias information and tools that consume it. Analyses are treated as "black boxes" satisfying a simple, formal specification given in terms of the semantics of Java bytecode. Knowing only this specification, one can build many different tools with only a small amount of code. The thesis explores the flexibility and efficiency of the design by describing the construction and evaluation of several different tools: tools to find dead code, resolve Java virtual method calls, statically check Java downcasts, search for accesses to objects, and build object models.

To support these tools, Ajax includes a novel static analysis engine for Java called SEMI, based on type inference with polymorphic recursion. SEMI provides fully context sensitive analysis of large programs. Using SEMI with the downcast checking tool, Ajax can prove the safety of more than 50% of the downcast instructions in some real-life Java programs, such as Sun's bytecode disassembler and the JavaCC parser generator. Ajax is the first system to address this particular task.

One of the key goals of this thesis is to study issues bearing on the practical utility of static analysis tools for programmers. This document describes some of the challenges involved in building an analysis system for off-the-shelf Java applications, and suggests some possible avenues for future research.

Acknowledgements

It almost goes without saying that I could not have completed this thesis without the support and tireless efforts of my advisors, Daniel Jackson and Jeannette Wing. With their help, I have learned far more during my graduate studies than I ever expected. Not only are they excellent supervisors and colleagues, but they are also marvellous people with whom I am fortunate to be acquainted. Thank you!

I am extraordinarily grateful to all my friends and colleagues in the Carnegie Mellon School of Computer Science. They have created an environment that is friendly, well-organized, incredibly stimulating, and designed to allow students to focus on learning and getting their work done rather than dealing with secondary issues. I can honestly say I do not expect ever again to work in such a wonderful setting.

In the two and a half years since our marriage, my wife Janet has consistently supported me in my work and indulged me when it interfered with our lives together. Fortunately such interference was not too frequent, and her love and companionship have been truly delightful.

My parents tolerated my obsession with computers from a young age, and have also supported me wholeheartedly during my interminable studenthood. Thanks Mum and Dad!

Much of the joy and support in the lives of Janet and I has come from our walk with God in the fellowship of the Pittsburgh Chinese Church. I would like to especially thank Yuan Chou and the other brothers and sisters who provided us with a spiritual home and great examples of servanthood for Janet and I to follow.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Setting

1.1.1 Software Engineering and Alias Analysis

Building large, complex software systems is difficult. Human beings have limited capacity to understand and recall the details of such systems. Since computers are adept at handling large quantities of data, one would expect automatic tools to be useful for helping programmers to understand large programs.

Indeed, many such tools do exist. Program code is partitioned into files and organized using file systems. Data about programs are stored in bug databases [88] and design documents [70].

In my thesis, I focus on tools that work directly with program code. A key phenomenon that makes program code difficult to understand is **aliasing**: the use of multiple names to refer to the same entity. For example, consider the fragment of Java code shown in Figure 1-1. In this code, a reference to the string object “Hello” is stored in `s1` and inserted into the `Vector`, and then extracted into `s`. Therefore the variables `s` and `s1` are aliased. Likewise `s` and `s2` are aliased.

```
static void main() {
    String s1 = "Hello";
    String s2 = "Kitty";
    Vector v = new Vector();    // Create a new Vector containing
    v.addElement(s1);           // s1 and s2, and print out its
    v.addElement(s2);           // elements

    Integer i1 = new Integer(7);
    Vector v2 = new Vector();
    v2.addElement(i1);

    for (Enumeration e = v.elements(); e.hasMoreElements();) {
        String s = (String)e.nextElement();
        System.out.println(s.length());
    }
}
```

Figure 1-1. Example of Java code exhibiting aliasing

Suppose the programmer wants to find out information about the object referred to by `s1` — for example, what methods are called on it, and where in the program those calls occur. It is insufficient to search the text for the name “`s1`”. The programmer must also examine `s1`’s aliases — in this case, `s`. In general, whenever the programmer is interested in properties of data which may be accessed through different names, alias information is required.

Most tools for understanding code make no attempt to handle aliasing. The programmer must manually peruse the source code to discover aliasing relationships and to gather information about the referenced data. This thesis describes the design of a practical alias analysis system for a modern programming language (Java), and code understanding tools based on it.

1.1.2 The Need For Alias Information

Many different questions which arise during programming involve alias information. Consider these questions that a programmer might ask:[1]

1. "What kind of objects can be in the container X?"
2. "What does the structure of object X and its contents look like?"
3. "Which methods of object X are invoked, and where are they called?"
4. "Is this line of code ever executed or not?"

The programmer might specify "object X" by giving, for example, a program location and the name of a variable in scope at that location.

All of these questions require alias information. Questions 1, 2 and 3 clearly require information about objects; collecting this information will require knowledge of which names refer to the objects of interest. In an object-oriented setting, question 4 also requires alias information because tracing the flow of control requires information about objects that are targets of method invocations.

This thesis demonstrates that not only do these questions require alias information, but once alias information is available in a convenient format, these questions are relatively easy to answer.

1.1.3 Shortcomings of Existing Tools

Existing practical tools use very simple approximations whenever they need alias information. A common and useful approximation is to compare the declared types of variables to see whether they may be aliases [23]. For example, in Figure 1-1, the `Vector v` and the `String s` cannot be aliases because the Java class hierarchy does not permit any object to be simultaneously a `String` and a `Vector`.

However, code reuse frequently leads to different instances of the same type being used in different ways. For example, in Figure 1-1 `v` and `v2` are `Vectors`, a generic container type frequently used in Java. Suppose the programmer wishes to prove that the `Vector` in Figure 1-1 contains only `Strings`. She must find all aliases to `v` and show that the objects inserted into those `Vectors` are `Strings`. An alias analysis based on declared types

1. These questions are all phrased in terms of object-oriented programs, but similar questions and observations apply to programs written in C, or any modern programming language.

alone will imply that `v` and `v2` are aliases, and therefore `v`'s `Vector` might contain `Integers` as well as `Strings`. Such an analysis will inaccurately conclude that the downcast to `String` might fail.

Researchers have devised much more sophisticated alias analyses. However, the fruits of this research are not being used by production-line programmers. The motivation for this thesis is to attack this adoption barrier.

Therefore I have constructed a program analysis system called **Ajax**. The design goals of Ajax reflect perceived limitations of previous attempts at implementing analysis tools.

- **Scalability**
  An analysis that produces wonderfully detailed information will be useless if it is unable to handle large programs. If a program is small enough to be easily understood by a programmer, then the programmer does not need an analysis tool.
- **Applicability**
  Many analyses are not useful because they do not deal well with features of modern programming languages and modern programs, such as
  - Higher order control flow and dynamic method dispatch;
  - Ubiquitous dynamic memory allocation;
  - Large, complex dynamic data structures;
  - Multiple levels of data encapsulation;
  - Class library code used in multiple contexts

  Ajax is designed to handle programs written in a modern language with all these features — Java — and is specifically designed to handle these features well.
- **Usability**
  Previous work such as Lackwit [54] erred by exposing the results of analysis very directly to the user, with little summarization or interpretation. It was often unclear to a normal programmer how the results should be interpreted. Therefore, instead of building a single monolithic tool, Ajax is designed to be a platform upon which a variety of tools can be built, each addressing a particular kind of task or question that the programmer may pose. The user interface to each tool is customized for its particular function.

An additional implied design goal is that Ajax must be powerful enough to be worth using while meeting the above requirements. At the least, it must discover useful information that could not be obtained by simple methods based on local reasoning. This thesis shows how Ajax achieves all these goals simultaneously.

1.1.4 Assumptions

Apart from the requirements above, the design of Ajax was constrained by assumptions about the nature of the solution. These assumptions stemmed from the background of this work, and have some independent justification, but are not fundamental.

- **Sound Static Analysis**
  Ajax is designed to produce static guarantees: results that are valid for all possible inputs and executions of the program. Therefore it must use conservative analysis. For example, when finding the sites of all method invocations on a particular object or set of objects, it only promises to return a superset of the true sites. One justification for using sound analysis is that the meaning of the results is easier to define; the results do not need to be qualified by the limits of a test suite or the nature of heuristics used by the system. Also, for some applications, such as compilation or automatic transformation, it is intrinsically important that the results be sound. However, an analysis need not be sound to be useful, so the choice to explore this part of the design space was not a necessary decision.
- **Global Analysis**
  Ajax analyzes whole programs. The behavior of any unavailable parts must be represented by specifications. This is desirable because behaviors due to component interactions are often the most difficult to understand, and therefore the most useful to be able to analyze automatically. Also, sound analysis of partial programs requires some sort of description of the missing parts, or else one must make “worst case” assumptions about those parts. The quality of the analysis results is likely to be severely degraded by such pessimistic assumptions.

1.1.5 Goal

The goal of this thesis is to demonstrate that sound, static, global alias analysis can be the basis for tools that accurately answer programmers’ questions about real, large object-oriented programs.

By “accurately”, I mean that the results are significantly more accurate than those provided by existing tools.

1.2 Approach

Ajax incorporates several key features to achieve the above goal.

1.2.1 Support For Multiple Tools and Analyses

The key to the design of Ajax is its division into *tools* and *analyses*. In Ajax, a *tool* is a component presenting a single interface to the user (typically, a programmer), designed to aid the user in a specific task by providing specific information in a specific way. An *analysis* is a component that produces alias information to be consumed by tools. Each analysis implements a simple, fixed, and rigorously defined interface, which presents aliasing information to tools in the form of an abstraction called the *value-point relation* (or VPR). This is illustrated in Figure 1-2.

This design has major benefits:

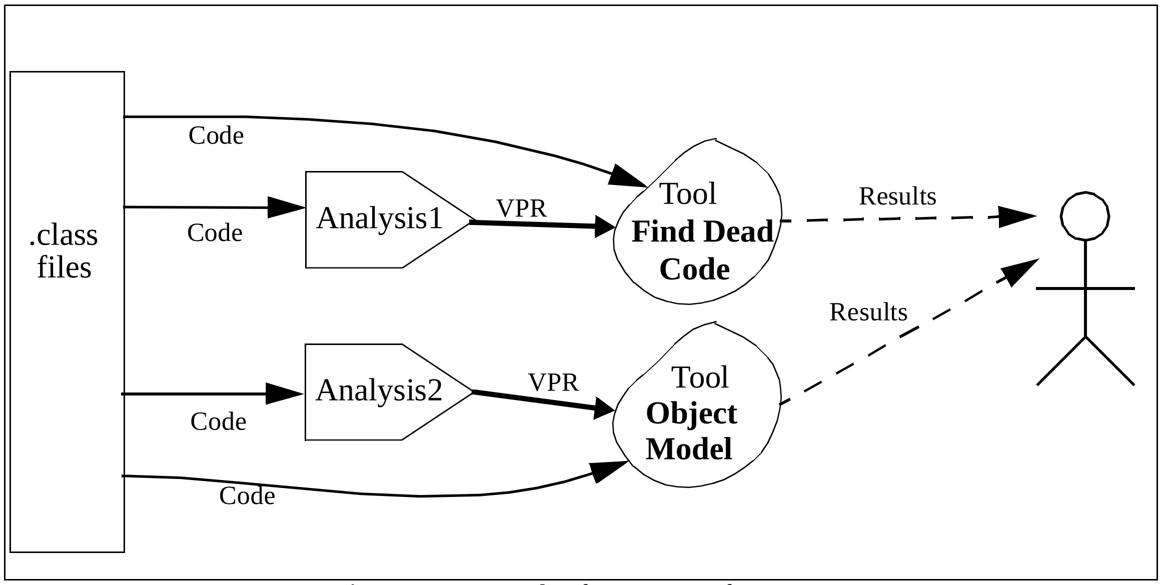


Figure 1-2. Example of an Ajax configuration

- One can use Ajax to construct one tool for each specific task that requires alias information. Ajax is carefully organised so that each tool requires little effort to implement. In particular, unlike some other analysis toolkits such as BANE [28], knowledge of the semantics of the target language is built into Ajax's analyses and does not have to be provided by the tool.
- Ajax offers a suite of different analysis engines. One can select an engine for a given problem to achieve an appropriate tradeoff between accuracy and resource consumption. Results show that the appropriate analysis configuration varies significantly according to the task being addressed. Because the VPR interface is fixed and fully defined, there are no fundamental restrictions on combining analyses with tools; any tool will operate correctly with any analysis. A given combination may or may not give good quality results, but it will give correct results.
- Ajax allows composition of analyses. Two analyses can be "intersected" to combine the best results of both to solve a particular problem. Alternatively, one analysis can be used as a "preprocessing step" to provide information that will speed up or improve the accuracy of another analysis. These capabilities are both crucial to good performance and accuracy in Ajax. To implement composition, an analysis simply uses the VPR interface to consume alias information produced by one or more other analyses. One such configuration is illustrated in Figure 1-3 below.

Conceptually, the value-point relation is simply the aliasing relation between program variables (and expressions). The difficult part of the design is defining a concrete interface connecting tools to analyses that allows efficient, simple implementations of both. The VPR also generalizes alias analysis to provide information about values which are not object references — e.g., integers. The details are explained in Chapter 3 and Chapter 4.

The design is exercised by constructing multiple analysis engines (see Section 1.2.3 below), and tools for the following tasks:

- Proving the safety of Java downcasts

- Identifying dead code
- Resolving virtual method calls
- Computing object models
- Scanning the program for accesses to objects satisfying certain criteria

1.2.2 Support For Java Programs

As mentioned above, Ajax is designed to handle general Java programs. Java programs exhibit a variety of “modern” language features that are becoming common:

- Objects — that is, inheritance, dynamic method dispatch, and data abstraction
- Extensive use of class libraries, such as the Java standard library and the Abstract Window Toolkit user-interface and graphics library
- Well-defined semantics; the language specification defines the behavior of all Java code
- Reflection and dynamic loading; Java programs can dynamically load new code at runtime, and metadata describing and providing access to loaded code and data is exported to the running program
- Exceptions
- Thread-based concurrency

To simplify the presentation and implementation, Ajax actually processes Java bytecode programs. This also makes it possible for Ajax to process programs whose source code is not available.

1.2.3 Simple Context Sensitive Analysis

To give significantly more accurate results than local analyses such as those based on declared types, an alias analysis must be able to distinguish between different data accessed with the same variable/type names. In complex programs, the interesting data are often constructed and accessed through one or more levels of indirection. For example, in object oriented programs, patterns such as constructors, abstract factories, and field access methods are ubiquitous. For these programs, some context sensitive analysis is required.

The goal is not to have the most sophisticated analysis, but rather one that significantly improves on existing fast analyses by providing context sensitivity. Therefore I chose to base Ajax’s primary analysis on the simplest analysis with a high degree of context sensitivity: Hindley-Milner style polymorphic type inference [49].

Hindley-Milner type inference is the basis for type inference in Standard ML [50]. The basic idea of applying this procedure to analyze aliasing in Java programs is to erase the declared types of variables, and perform type inference based only on the type constraints induced by operators used in the program code. The inferred type information is used to resolve aliasing questions in a similar way to which declared type information is used. However, inferred types give more precise information than declared types, because the inferred types can be finer and their type system richer, by virtue of polymorphism. For example, in Figure 1-1 Ajax can automatically prove that the `Vector v` contains only `Strings`, and therefore the downcast cannot fail. This example requires *context sensitive* analysis (see Section 2.2.2); no other comparable system provides it.

Based on experiences with Lackwit [54], a similar system for analyzing C programs, I extended the analysis in several ways:

- The addition of polymorphic recursion [42] prevents loss of polymorphism in the presence of mutually recursive declarations.
- To better handle Java objects, the analysis treats "extensible records" [65] in a clean way.
- I changed some details of the theory and implementation to improve performance and better fit Java programs.

These features are extensively discussed and evaluated in this thesis. The general problem of type inference with polymorphic recursion can be reduced to the formal problem of *semiunification* [42]; for this reason I call this alias analysis engine "SEMI".

I also implemented a variant of Rapid Type Analysis [9], an analysis based on reasoning about the declared types of variables. Figure 1-3 shows an example Ajax configuration using one instance of SEMI and two instances of RTA. This configuration is explained further in Section 4.4.5 and Section 9.6.

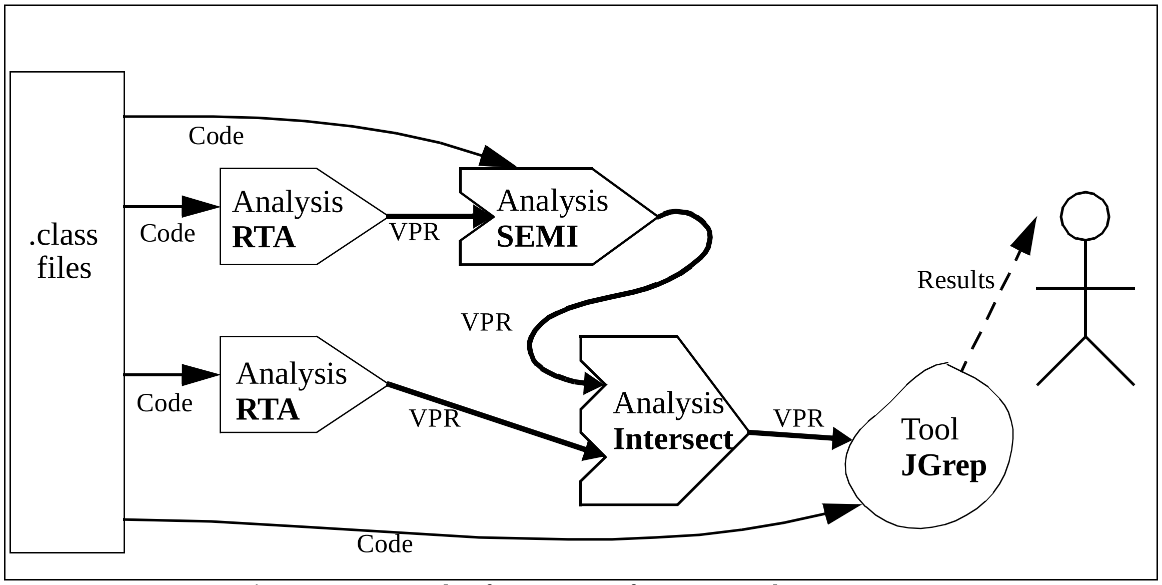


Figure 1-3. Example of an Ajax configuration with composition

1.2.4 Distinguishing Features

Some unique features distinguish Ajax from all prior work:

- The SEMI analysis engine is the only engine combining full support for the Java language, context sensitivity, and higher-order control flow analysis.
- SEMI is the only analysis engine for a real programming language that provides polymorphic recursion and also distinguishes different fields of structures.

- Ajax is the only analysis toolkit able to provide aliasing information directly to tools in a clean, efficient and analysis-independent way.
- Ajax is the only system able to prove the safety of Java downcasts related to generic data structures (effectively reverse engineering the type parametricity of those structures).
- Ajax has the only object modelling tool able to automatically and soundly “split” classes in the model.

1.3 Contributions

This thesis makes the following technical contributions:

- It introduces and evaluates new techniques for performing generalized context-sensitive alias analysis of Java code. These techniques extend previously published work in several directions.
- It defines the value-point relation, and uses it to describe a flexible and general interface for efficiently transmitting generalized alias information from analyses to tools and other analyses. The ideas behind the value-point relation are not new, but the relation has not previously been formally specified and used as the basis for an implementation. Similarly, the interface between tools and analyses formalizes and generalizes some existing ideas.
- It demonstrates a variety of tools that programmers can use to analyze Java programs, including a tool for building object models and a tool that proves the safety of downcasts associated with the use of Java generic containers.
- It shows how all the above contributions are achieved in the context of the full Java language and realistic Java programs. This context imposes some fundamental difficulties that must be faced by any system for global static analysis. The thesis explains the difficulties and how they are addressed by Ajax.

1.4 Thesis Overview

The thesis comprises five major sections.

The first section of the thesis introduces my work and places it in the context of other work on program analysis and software engineering. Chapter 2 surveys the related work and discusses its relationship to Ajax.

The second section of the thesis explains the architecture of Ajax, in particular the “value-point relation” interface that separates tools from analyses. In Chapter 3, I introduce the VPR abstraction and describe how it is used to communicate alias information. It takes some thought to actually realize this abstraction in a way that permits efficient implementation; the resulting interface is described in Chapter 4. In Chapter 5, I present an extension of RTA as an example of how an analysis can implement the VPR interface.

The third section of the thesis describes Ajax’s SEMI analysis. Chapter 6 formally defines the analysis over a subset of the Java bytecode language, and proves that the analysis is sound. Perhaps surprisingly, the proof reveals that the soundness of SEMI does not depend on any static type safety properties of the analyzed program; if the class file can be parsed, then the code can be correctly analyzed. Chapter 7 describes some of the actual implementation details, in particular those that aim to improve performance. Unfortunately Java has some features that are hard to treat with global static analysis; these features are discussed in Chapter 8.

The fourth section of the thesis is a description of five tools built using Ajax, along with quantitative and qualitative evaluations of those tools using a suite of example programs. The example programs — which include "real-life" programs such as `javac` and some large GUI applications, along with the standard Java library — are described in Chapter 9. Chapter 9 also presents quantitative results for two tools: one for resolving dynamic method invocations, and one for finding dead code. This chapter focuses on comparing the effectiveness of different analysis engines in different configurations. In Chapter 10 I present and evaluate a tool for checking the validity of downcasts. Chapter 11 describes the implementation and results of a tool for producing object models (similar to storage shape graphs), which requires the use of multiple VPR queries and some amount of post-processing. In Chapter 12, I present "JGrep," a simple tool with a variety of uses, that simply scans for certain kinds of aliases to expressions specified by the user.

Chapter 13 contains the conclusions of the thesis. In brief, I have achieved the main goal of the thesis: Ajax performs sound, static, global alias analysis; provides tools to answer programmers' questions using this information; gives results significantly more useful than those obtainable using previous systems; and is practically applicable to real programs and problems. However, I have identified some major barriers to adoption for general purpose, large scale programming. One problem is that the analysis is still not scalable enough; SEMI consumes too many resources and seems less accurate as programs get larger. More importantly, most real Java programs use language features — such as reflection and dynamic loading — that are inherently inimical to sound global static analysis.

2 Related Work

2.1 Introduction

Much work has been done in areas related to this thesis. The Ajax analysis engines are related to work on global flow and closure analysis, alias analysis, and type inference systems. The Ajax tools are similar to previous systems for program understanding.

As discussed in Section 1.2.1, Ajax separates analyses from tools. Analyses compute generalized alias information about a program, and tools consume the information. Ajax is the only toolkit able to provide alias information directly to tools in a clean, efficient and analysis-independent way.

The SEMI analysis engine also has unique properties. It is designed to handle real programs using modern features such as objects and many levels of indirection. No other alias analysis engine combines context sensitivity and higher-order control flow analysis with full support for a modern programming language and the ability to handle realistically large programs. SEMI is also the only engine for any language which uses polymorphic recursion and also distinguishes different fields of structures.

Ajax provides some unique tools to demonstrate its power. Its downcast checking tool is the only system able to prove the safety of Java downcasts related to generic data structures (effectively reverse engineering the type parametricity of those structures). Ajax also provides the only object modelling tool able to “split” classes in the model both automatically and soundly; see Chapter 11 for details.

2.2 Program Analyses

This section describes related work in program analysis. Section 2.2.1 explains why it is important to distinguish fundamental analysis techniques from the particular problems to which they are applied. Sections 2.2.2 and 2.2.3 define some terms useful for classifying analyses, and give some general comments about interpreting the results of work in this area. The following sections describe the actual related work, clustered according to the characteristics of each analysis technique.

The final sections deal with work that is not about specific program analysis techniques. Section 2.2.8 covers type inference for type checking in programming languages. Section 2.2.9 presents work on composing analyses, and Section 2.2.10 compares program analysis toolkits.

2.2.1 Distinguishing Analysis Techniques from Analysis Problems

The problems of “flow analysis,” “closure analysis,” “higher-order control-flow analysis,” “alias analysis,” and “concrete type inference” are all closely related, being attempts to automatically and statically characterize the values of program variables. They differ only in the types of the values they characterize and in the kinds of characterizations they make.

The same basic analysis techniques are often applied to different problems to yield apparently different solutions. For example, a closure analysis is so called because it determines which function bodies may be evaluated to by an expression denoting a higher-order function. Alias analysis is so called because it determines which abstract memory locations may be evaluated to by an expression denoting a pointer value. However, despite the different contexts, and often radically different presentation styles, the same techniques can be used to solve both problems. (Some alias analysis techniques are applicable only to first-order code, limiting their utility for closure analysis.)

Prior to Ajax, applying an existing analysis technique to a new problem domain often required significant effort. For example, researchers first described how to use declared type information to resolve higher-order control flow [22] and then later showed how to use the same techniques to perform general alias analysis [23]. As discussed in Section 1.2.1, Ajax completely separates analyses from problem contexts. In Ajax, matching an analysis to a problem context is a simple runtime configuration decision. No prior work has this property.

As well as adding useful implementation flexibility, the decoupling of analysis techniques from problem contexts makes for easier comparison of the underlying techniques. For example, in Chapter 5 I show that the two analyses mentioned above, both based on declared types and superficially similar, are actually subtly different in precision.

In this discussion, I deemphasize the original context in which work was presented and focus on underlying techniques.

2.2.2 Classifying Analyses

It is helpful to classify analyses according to whether they possess “flow sensitivity” and/or “context sensitivity”. These terms are used informally and inconsistently in the literature. I adopt the following definitions:

- An analysis is *flow sensitive* if, when expressed in the form of constraints, it uses inclusion (subtype or subset) constraints.

The intuition behind flow sensitivity is that, considering the program fragment “if *x* then *y* else *z*”, a flow sensitive analysis can determine that the result is either *y* or *z* while still distinguishing *y* and *z*.

Many authors use “flow sensitive” to mean that the analysis may produce different results depending on the ordering of statements within a method or function. However, with this definition, any analysis can trivially be made flow-sensitive simply by converting the program to single static assignment form (for local variables) as the first phase of the analysis. Therefore, such a definition does not usefully characterize the analysis technique itself.

- An analysis is *context sensitive* if, when expressed in the form of constraints, it is possible for two occurrences of the same program variable to induce equality or inclusion constraints whose sets of free variables are disjoint.

The intuition behind context sensitivity is that the information obtained by a context sensitive algorithm will not necessarily be improved by duplicating code that is used multiple times in the analyzed programs. This includes analyses described as “polyvariant” or “polymorphic,” and also some uses of intersection types [59].

Both of these definitions refer to data flow sensitivity, i.e., they describe the kinds of constraints used to approximate data flow in the program. I am not concerned with control flow sensitivity.

These crude definitions can be usefully applied to most of the related work. They are used inconsistently in the literature, and therefore other authors may apply them differently.

2.2.3 Describing Results

I deliberately emphasize performance demonstrated in practice over asymptotic worst-case complexity. Complexity results can be very misleading because real programs almost always have characteristic properties that prevent them from triggering the worst-case behavior of many algorithms (ML type inference is the classic example). Unfortunately, published benchmark results can also be misleading, because real programs almost always have properties (such as internal code reuse) that are not exhibited by most small benchmark programs.

Many authors report results in terms of the number of abstract locations associated with load or store operations in the program (i.e., sizes of points-to sets). Unfortunately, this metric is not very useful, because the domain of abstract locations often varies from analysis to analysis. Indeed, type inference analyses do not directly define a domain of abstract locations. Furthermore, it is not clear how the sizes of the sets relate to the utility of the results. An analysis that maps the result of every C `malloc` operation to the same abstract location could easily produce very small points-to sets but be absolutely useless in practice. Measurements that relate the dynamic behavior of a program to its static approximation, such as the work of Grove et al. [37], are much more useful.

Many of the alias analyses presented below assume that pointed-to memory locations can have only one outgoing pointer, or in other words, every structure can have only one field. For structures with more than one field, the fields are treated as one and not distinguished. This can drastically change the performance characteristics of an analysis, because it effectively reduces program data structure shape graphs from branching trees to linear sequences, and ensures that all recursive structures become pure cycles. This approximation is so common that it is not always clearly stated.

2.2.4 Flow Sensitive, Context Insensitive Analyses

One area of analysis where scalability is often an explicit goal is alias analysis and related problems, such as side effect estimation.

Andersen [5] gives a simple flow-sensitive algorithm based on inclusion constraints for alias analysis of C programs. It is often thought of as context-sensitive, because passing a parameter to a called procedure is treated as assignment of the actual parameter to the formal parameter; flow sensitivity ensures that different actual parameters at different call sites can be distinguished even when they map onto the same formal parameter. Unfortunately the result of a called procedure is never handled context sensitively; a returned pointer always maps to the same set of abstract locations regardless of the calling context. Thus, if access to object fields is consistently performed through accessor methods of the object (as is often the case in Java programs), Andersen’s algorithm is equivalent to requiring, for each declared field of a class, a single abstract storage location that summarizes the contents of every runtime instance of that field.

In a series of reports [30] [75], Aiken and his collaborators describe methods for improving the performance of inclusion-based analyses such as Andersen’s algorithm. This work is almost exclusively aimed at analyzing large C programs and does not consider context sensitivity. Their work makes Andersen’s algorithm practically applicable to large programs. Note however that even their most recent results make the “one field per structure” approximation; this is especially significant because their “projection merging” technique relies on type constructors having small arity.

Rountev, Milanova and Ryder [66] extend the improved algorithm to model multiple fields per object, and apply it to Java programs. Their method effectively transforms programs to first-order code before analysis, using declared type information and analysis of the class hierarchy to determine possible callees of indirect method calls. They do not attempt to handle reflection and completely ignore the effects of library code; therefore it is difficult to interpret their results. In particular, the numbers of methods they find to be dead in their test programs are suspiciously large.

A classic approach to “higher order control flow analysis” (“CFA”) was presented by Shivers [71]. Heintze [39] introduced set-based analysis. Both of these techniques can be thought of as methods for higher-order control flow analysis using inclusion constraints. Since then, much work has been done to decrease the time and space requirements of these techniques, especially when some kind of context sensitivity is required.

Heintze and McAllester [41] describe an implementation of CFA that answers certain questions in linear time for programs that have types that are bounded in size. Unfortunately this approach cannot be directly applied to C and Java programs because its treatment of recursive types is based on ML datatypes. If the entire Object type were treated as one datatype, there would be a great loss of accuracy: it would be impossible to distinguish different fields of the same object (other than scalar fields). This is because an ML datatype has a fixed pattern of type recursion, so modelling Object with a datatype requires all fields holding object references to have the same type as the containing object. Heintze and McAllester's analysis uses type information to guide its approximations for dealing with recursive types, and in this case it will resort to the gross approximation mentioned above. Another problem with their method is that extending it with some kind of polyvariance or polymorphism could lead to serious performance problems.

Flanagan and Felleisen [33] describe an implementation of set-based analysis designed to handle large programs. It analyzes each component separately, generating a collection of set constraints that approximate the behavior of the component, then simplifying the constraints. Finally the sets of simplified constraints are combined and solved. This reduces the amount of space required to analyze an entire program. The improvement over the basic algorithm is very impressive, but the largest program analyzed is 18,000 lines of Scheme, so it is difficult to draw conclusions about scalability, or about its behavior on object oriented programs.

DeFouw, Grove and Chambers [21] consider a framework of “fast” algorithms posessing varying degrees of flow sensitivity and ranging from linear to cubic time complexity in the size of the program. Sudaresan et al. [76] present new algorithms in this class, as do Tip and Palsberg [80]. All these algorithms could easily and profitably be implemented to produce VPR approximations in Ajax.

2.2.5 Flow Sensitive, Context Sensitive Analyses

Ruf [67] compares two flow-sensitive algorithms, one context-sensitive and the other context-insensitive. The sets of possible locations at each load or store were almost identical, leading him to conclude that for those benchmarks, context sensitivity was worthless. However, he suggests in the paper that those results may not generalize to larger programs. (The largest program considered was less than 7,000 lines of C.)

A similar study was done by Foster et al. [34]; they conclude that adding context sensitivity improves the accuracy of a flow insensitive analysis, but not a flow sensitive analysis (Andersen’s algorithm). Unfortunately their context-sensitive analyses do not distinguish memory objects created by the same textual occurrence of “malloc”, and therefore may be failing to exploit some of the power of context sensitivity (for example, by failing to distinguish instances of heap-allocated abstract data types, which Lackwit and Ajax are able to do). They observe that the main advantage a true context-sensitive algorithm has over a flow-sensitive algorithm (such as Andersen’s algorithm) is that results or “out parameters” of function calls can be distinguished in different contexts, and that their C programs do not exhibit much of this kind of polymorphism, functions being mostly executed for their side effects. However, Java and C++ encourage reads of object state to be encapsulated in accessor methods, so “result polymorphism” is much more common in programs for these languages.

Ryder and her collaborators [74] [14] developed a series of algorithms for large-scale flow-sensitive alias analysis, and embodied them in a toolkit. Their approach is based on the propagation of “points-to sets” encoding the aliasing relationships that hold at each program point. Each points-to set is a set of abstract locations that a pointer may be referring to. This basic method is extended to handle higher-order code (by dynamically updating a call graph and incrementally propagating information between new callees and callers); other extensions are introduced to handle structures, exceptions and other modern language features. Their most sophisticated general-purpose algorithm which is also context-sensitive [14] is only demonstrated on programs with less than 7,000 lines of C++ code. (It does not explicitly handle higher-order code; the programs are first reduced to first-order by applying class hierarchy analysis.) Also, they have one abstract location for each occurrence of a call to “malloc” in the source code. Therefore this analysis can never treat memory allocation context-sensitively, and can never distinguish instances of abstract data types which are allocated by a common constructor function.

Wilson and Lam [84] give an algorithm for context-sensitive, flow-sensitive alias analysis for C programs that computes abstractions of procedures, called “partial transfer functions”, that depend on the calling context but can often be reused between calling contexts (often, only one PTF is ever computed for a procedure). Unfortunately, they only report results for small, mostly numeric applications (no larger than 5,000 lines), though their results are excellent. Because their PTFs depend on the alias patterns in the calling context, and in particular depend on the actual values of function pointers passed in by the caller, it is not clear how much expensive reanalysis would be required for larger programs with complex data structures and/or use of function pointers (object oriented programs fall into this category). They give no measurements of the quality of the results of their algorithm. Also, they only analyzed C programs with mostly first-order code.

Cheng and Hwu [16] describe another PTF-based technique that trades off accuracy in exchange for better scalability. Their system has been successfully used as part of an optimizing compiler for the C SPEC benchmarks. According to my definitions, it is both flow sensitive and context sensitive, but it does make a number of approximations that make it hard to compare with other algorithms. It is unclear how it would fare on object-oriented programs.

Plevyak's analysis [63] for object-oriented programs is based on "adaptive splitting," which dynamically adds context and flow sensitivity when needed to improve the accuracy of the analysis on some particular task. The analysis is used as the basis for a number of optimizations in an optimizing compiler for a Java-like language, ICC++. The analysis looks promising but, as is often the case, only relative small programs are targeted (up to 25,000 lines in later work [24], which does not report absolute performance results) and direct comparisons with other systems are difficult.

Grove, Dean, DeFouw and Chambers [37] survey a number of algorithms for "call graph construction" for object oriented languages. The algorithms studied include those of Palsberg and Schwartzbach [60], Oxhøj, Palsberg and Schwartzbach [56], and Agesen [1]. The call graph construction problem is essentially the same as higher-order control flow analysis: identify the possible targets of an indirect function (or procedure, or method) invocation. They conclude "our experiments demonstrated that scalability problems prevent the flow-sensitive algorithms from being applied beyond the domain of small benchmark [Cecil] programs." All of the context-sensitive algorithms they consider are also flow-sensitive. The algorithms performed much better on Java programs, presumably because Java is not as "pure" an object-oriented language as Cecil and therefore method dispatches are less ubiquitous.

Their results show that for resolving dispatches, adding flow sensitivity makes more difference than adding context-sensitivity, if the context-sensitive analysis is also flow sensitive. Unfortunately it is hard to compare their results to mine, because our systems make different assumptions. For example, we handle library code differently — see Chapter 8.

Fähndrich and Aiken [29] describe how to construct an interesting analysis framework that incorporates inclusion constraints and polymorphism, but uses equational (i.e., flow insensitive) constraints judiciously to improve the efficiency of the algorithm, where loss of information is not as important. They apply the framework to the problem of inferring uncaught exceptions in ML programs, but provide very little information on the actual performance of their algorithm.

2.2.6 Simpler Analyses

In response to the expense of applying known flow-sensitive or context-sensitive analyses, researchers have developed fast, but somewhat crude algorithms for answering various program analysis questions, mostly in the context of compilation and optimization.

A classic algorithm for determining the possible targets of a method call is "class hierarchy analysis." In a statically typed language, it examines the class that the source program declared for the object reference in a method call; the run-time class of the object must be a subclass of the declared class, and so the possible targets of the dispatch are the method in the declared class (if there is one), and any overriding method declarations in those subclasses [32, 20, both cited in 9]. Even languages such as Smalltalk that lack a static type discipline can use similar approaches, by computing the set of classes which declare or inherit a method implementation compatible with the call.

Diwan, Moss and McKinley [22] [23] extend this basic method with intraprocedural flow analysis and some very simple (context insensitive) interprocedural propagation and handling of data structures, resulting in an analysis that is still linear in practice. Their algorithms are quite effective for their benchmarks, but the benchmarks are mostly small. In their system for resolving dynamic method invocations [22], the only program ("Trestle") that consists of more than 20,000 lines of code gives their second-poorest result, resolving almost none of the 20% or so dynamic method invocations that are invoked at monomorphic call sites (i.e., call sites observed always to call the same method implementation at run-time). Interestingly, they comment that this program is the only one of their benchmarks that might benefit significantly from context sensitivity.

Bacon and Sweeney [9] extend class hierarchy analysis with "Rapid Type Analysis," which essentially eliminates dead code and classes in C++ programs, by starting with the assumption that only "main" is called and adding in classes, procedures and methods as necessary until a safe approximation is reached. The analysis runs in linear time and gives good results for many programs, particularly because stripping out entire unused classes can often improve the results of class hierarchy analysis. However, most of their benchmarks do not exploit subclass polymorphism, and the benchmarks are mostly small (only one has more than 20,000 lines of code). An interesting lesson from their work is that it is highly desirable for an analysis to ignore code shown to be dead. RTA achieves this by approximating the set of live methods from below; Ajax generalizes this strategy and uses it for all its analyses. Also, because of RTA's simplicity, efficiency and effectiveness, I have used it as the basis for one of the Ajax analysis engines.

Steensgaard [72] applied a very simple type inference scheme to analyze aliasing for C programs. In its original incarnation, it did not distinguish members of the same record, and it was context and flow insensitive. The ability to distinguish record members was added in later work [73]. In practice, these schemes scale to very large programs with millions of lines of code. Other variations have been created which introduce carefully limited flow sensitivity while retaining scalability [19].

Heintze [40] describes extensions of the equivalence results of Palsberg and O'Keefe [58] that, among other things, show the equivalence of unification-based type inference (i.e., without subtyping) to a simple closure analysis. There are no empirical results, and polymorphic type systems are not treated. The type system obtained is very similar to that used for binding time analysis by Bondorf and Jørgensen [8]. The analysis is more powerful than Steensgaard's [72], but less powerful than Wright and Cartwright's [85] (see below).

2.2.7 Flow Insensitive, Context Sensitive Analyses

Several researchers have produced flow insensitive, context sensitive program analyses based on the Hindley-Milner algorithm for inferring polymorphic types in languages based on lambda calculi [49]. This algorithm is attractive because of its exceptional simplicity, its elegant handling of higher-order code and complex data structures, and its proven scalability in some contexts, such as type inference for ML [50].

Tofte and Talpin's region inference [81] is somewhat similar to the SEMI algorithm used in Ajax, partly because it uses polymorphic recursion [42]. There are significant differences, however. Their system is unnecessarily complex (for my purposes) because it includes effect inference, which I do not need. On the other hand, their treatment of recursive types is insufficient for my needs because they analyze ML programs which have explicit datatype declarations describing the recursive types. Their use of polymorphic recursion is also limited to the region variables, but my usage is much more general. Also, my work is in totally different application domains from theirs, so the results are incomparable.

Wright and Cartwright's soft typing system for Scheme [85] handles recursive types, records, and polymorphism, but it does not distinguish different instances of the same basic type, which is a fundamental requirement for many of my applications. For example, if two variables both refer to lists of integers, Soft Scheme must assume that the references are aliased.

Lackwit [54] [55] is a system using polymorphic type inference to perform alias analyis of large C programs. It was the direct predecessor to Ajax. Lackwit’s analysis worked well — analyzing more than 100,000 lines of code in less than 64MB of RAM — and handled recursive types, structures, and some uses of type casting. However SEMI improves on it in several ways, as discussed in Section 1.2.3. Also, the design of Ajax as a “tool suite” stems directly from the shortcomings of Lackwit as an “all in one” tool.

Liang and Harrold [62] constructed a similar analysis for C programs by extending Steensgaard’s algorithm. They do not distinguish structure fields or handle higher-order code. Their test programs have less than 25,000 lines of code.

Fähndrich et al. [31] built an analysis similar to Lackwit, adding polymorphic recursion and “polarity” information to instantiation constraints. The polarity information improves accuracy without much effect on performance. They achieve good scalability results on C programs, but their system is not discriminating between the fields of structures, which avoids some of the performance problems which I had to address in SEMI. My SEMI analysis could exploit polarity information in the same way to improve its accuracy.

Pessaux and Leroy [61] created an analysis for finding uncaught exceptions in O’Caml programs. Previous approaches had used inclusion constraints; they abandoned these in favor of unification-based type inference and polymorphic recursion. They have some interesting comments about the tradeoffs involved; they saw little degradation in accuracy, and were actually able to increase precision because the simpler technology allowed them to build a more complete analysis. Their analysis is impressive; they can analyze nearly

20,000 lines of (non-object-oriented) O'Caml code. Because they are interested in recovering only the concrete types of exceptions which can be thrown, their analysis and results are not directly comparable with systems such as Ajax.

There has been much recent work on specialised alias analyses for Java for tasks such as escape analysis and synchronization removal [17] [10] [11] [83] [4]. The analysis most similar to SEMI is Ruf's [69]. It computes similar information to Ajax, partitioning object references into equivalence classes and propagating information from callees to callers in a context-sensitive manner. His analysis is much faster than SEMI. This is partly because it is applied to programs that have already been transformed to be first-order, and it does not support polymorphic recursion. He also uses several tricks to improve performance for his particular task. Even when SEMI is configured to reduce the program to first-order before analysis, and full polymorphic recursion is disabled, Ruf's analysis is still much faster. This indicates that when polymorphic recursion or incremental analysis are not required, deterministic propagation of summaries along the call graph is much more efficient than using a general incremental constraint system like SEMI. Lackwit used a similar single-pass deterministic algorithm to propagate type information from the leaves of the graph of program declarations up to the root, and it also seems to be much faster than SEMI.

2.2.8 Type Inference for Object Oriented Languages

Many researchers have developed sophisticated type inference systems, and there has been much recent work on integrating object-oriented features into languages with type inference. These systems mostly rely on introducing inclusion (subtyping) constraints, and their performance is usually not evaluated. Furthermore, as for the soft typing system discussed above, these inference systems are oriented towards finding type errors and do not attempt to distinguish values with the same concrete type (e.g., two integers, or two objects with identical structure).

Although not for object oriented programs, Henglein's exposition of type inference for polymorphic recursion [42] was a major influence on my work and the work of others.

Eifrig, Smith and Trifonov [27] give a rich type inference system for languages with object oriented features (with support for state and records). There is no mention at all of any implementation or its performance.

Palsberg and O'Keefe [58] prove that a certain simple type inference system with recursive types and subtyping is equivalent to a standard closure analysis. Obviously performance problems exhibited by flow analyses will carry over to the equivalent type inference systems, unless we relinquish some expressive power. Context sensitive closure analyses or polymorphic type systems are not treated.

Palsberg [57] describes a type inference algorithm for Abadi and Cardelli's object calculus. The algorithm incorporates subtype constraints, and requires O(n^3) time in the worst case because it computes a transitive closure; empirical results are not reported. It does not incorporate parametric polymorphism. Because the subtyping rule is based on record extension (requiring common fields to have the same type), parametric polymorphism would be required to ensure true context sensitivity.

Rémy and Vouillon [65] describe the type system of Objective Caml, which provides type inference for an object-oriented extension of ML, without the use of subtype constraints. They use polymorphic row variable types to write functions that are polymorphic over object types. (Row variables range over a set of unknown fields and their types.) They require explicit coercions in other situations (e.g., heterogeneous containers). They can infer recursive types in function and method signatures. This type system is very close to the type system used by SEMI, except that because their source programs have properly block-structured declarations, they have no need for polymorphic recursion. Furthermore, like Wright and Cartwright's Soft Scheme, the system is designed to prove type safety, and has none of the extensions required to collect other information. Also, the language is intended to be class-based, but class types are not suitable for my purposes. In my system, the type inferred for an object of class A may encode information about the subclasses of A as well, since the object could be one of those subclasses. This information is neither needed nor allowed in O'Caml, since it breaks modularity and is not useful for typechecking.

Duggan [25] proposes a type inference procedure for reverse engineering parameterized types from Java code. His system is significantly more complex than SEMI and Ajax's downcast checker, because it is construed as a source-to-source translation from Java to "PolyJava", an extension of Java with bounded parameteric polymorphism. Therefore he is concerned with ensuring that the translated code typechecks and has the same semantics as the original code. Most importantly, he has not implemented the analysis, so its behavior in practice is unknown.

2.2.9 Composing Analyses

Hybrid approaches to closure analysis and alias analysis have been proposed, that combine traditional flow analysis of abstract values with type inference. Ruf [68] and Zhang, Ryder and Landi [86] [87] suggest similar schemes for alias analysis that first apply a fast type inference analysis, and then use the results to select a subset of the program to be analyzed with a more expensive flow analysis to obtain more precise information for a certain set of values. In fact, this approach can actually improve the accuracy of the results because analyses are often precise or imprecise in different ways, and taking the intersection of the results can be better than any single set of results. The Ajax framework explicitly supports this kind of composition; see Section 4.4.5.

2.2.10 Analysis Toolkits

One of the strengths of Ajax is its modular design, enabling tools for different tasks to be quickly and easily built using a simple, powerful abstraction of alias information. Two "state of the art" toolkits for global static analysis are BANE [2] and PAF [74].

BANE [2] provides an engine for solving term equality and set inclusion constraints. It also supports Hindley-Milner style polymorphism (but not polymorphic recursion). To implement a task-specific tool using BANE, the implementor must create a front end to traverse program code and build a set of constraints to be solved. The implementor must also create a "back end" to interpret the solved constraints in order to solve the problem at hand. In particular, the implementor must determine how to express the problem in the form of constraints, and prove that the constraint problem corresponds to the real problem. In contrast, an Ajax tool implementor is provided with the VPR abstraction of semantic information, without having to write any front end code, and without having to worry about how the information was produced. In most cases the implementor’s desired information can be extracted directly from the VPR. The price is that Ajax can only provide aliasing information; BANE could be reused in other contexts.

Like Ajax, PAF [74] computes alias analyses of programs. However, it does not provide an abstract interface comparable to the VPR. Instead, the analyses produce “points-to sets” listing, for each pointer dereference in the program, the abstract locations the pointer could be pointing to. For a tool to use this information, it must encode the meaning of the abstract locations; this is undesirable because the domain of abstract locations could change depending on the analysis method being used. It is also undesirable because it places an unnecessary burden of understanding on the tool implementor. Also, it is not always efficient to explicitly convert analysis results into points-to sets and then interpret those sets; the points-to sets can be very large. The VPR is designed to avoid this bottleneck.

2.3 Software Engineering Tools

2.3.1 Software Engineering Tools for Program Understanding

There are many tools that address aspects of the program understanding task, some built as research projects and some as commercial products. Almost exclusively, such tools that aim to be scalable do not rely on semantics-based analyses, but operate at the lexical or syntactic level. For example, the products of Imagix Corporation [90] provide a number of different views and summaries of program source code, all of which rely on lexical and syntactic information, or on profile information gathered by running the program. The C Information Abstraction system [15], and its successors and many other similar systems, essentially treat a program as an abstract syntax tree without assigning meaning to the syntax elements. In CIA, this information is imported into a database, and various relational queries can then be used to extract useful information. For example, the tool could rapidly locate all mentions of a particular field of a given structure type. My work extends these ideas by providing much richer information about the semantics of the program.

Murphy and Notkin developed some lexical analyses that are particularly efficient and easy to customize [51]. Due to its lexical nature, their tool can be more flexible (for example, it can analyze programs written in multiple languages), and will be more efficient in most cases. Its strength is also its weakness. By operating purely at the lexical level, it cannot address semantic queries with the precision or soundness of semantics-based analysis.

The same researchers' Reflection Model Tool (“RMT”) [52] allows the results of a static analysis to be presented at a more abstract level than the code, such as an architecture diagram, and to be compared to the expectations that the user has for that level. It assumes that the result of the source code analysis is a graph, and produces diagrams to show how the abstracted graph differs from that expected. RMT is independent of the tool used to analyze the source code, and my tools could be used in that role.

Bowdidge and Griswold's “Star Diagram” tool [7] and its successors aid in encapsulating abstract data types, by presenting a special view of the program that focuses on a particular variable. They assume that there is a single variable to be abstracted, but they discuss extending their method to operate on data structures with multiple instances. They consider operating on all data structures of a certain type, but comment “The potential shortcoming of this approach is that two data structures of the same representation type, particularly two arrays, might be used for sufficiently different purposes that they are not really instances of the same type abstraction.” Ajax and SEMI solve this problem.

The Womble object modelling tool [46] uses syntactic analysis, intraprocedural analysis, heuristics and built-in knowledge of the Java class library to produce object models [70] of Java programs. It is not sound; its object models can fail to reveal class relationships that actually exist in the program. In contrast, the Ajax object modelling tool is sound, and can accurately “split” classes without being given any special information other than the code. See Chapter 11 for more details.

2.3.2 Semantics-based Tools For Program Understanding

The majority of work from the software engineering community that tries to capture truly semantic information is concerned with slicing [82] [78] — that is, the identification of a subset of a program that completely determines the value of a given variable at a given program point. This kind of information may be useful for testing, debugging and other applications. Unfortunately, most efforts to date have failed to achieve any kind of scalability or to operate on realistic languages and programs. The most realistic slicing tool available is Grammatech’s CodeSurfer product [89]. CodeSurfer analyzes C programs and relies on Andersen’s algorithm to resolve aliasing in order to compute more accurate dataflow graphs. My work shows that alias information itself can be used to solve several problems of interest to the software engineering community.

The Anno Domini tool [26] uses monomorphic, unification-based type inference to compute “Y2K” type information for data in COBOL programs. Anno Domini is a tool designed to support one task very well. Ajax is designed to enable cheap construction of many such “domain specific” tools.

2.4 Language Semantics

This thesis presents a soundness proof for SEMI, which requires specification of the semantics of the source language — in this case, a large subset of Java bytecode. The semantics presented here are a correction and simplification of the work of Qian [64]. In contrast with other semantics for Java bytecode, my semantics are completely dynamic and rather “lax”. There are no static checks, and the only run-time checks are those necessary to ensure deterministic and sensible execution. This is because Ajax is not concerned with verifying the static safety of Java bytecode; in fact, the soundness proofs demonstrate that SEMI can soundly analyze bytecode which violates any and all static safety constraints.

However, it is also true that the techniques that underly Ajax, and SEMI in particular, can be useful in performing static typechecking of bytecode. I have done some work in this area [53], but it is beyond the scope of this thesis.

3 The Value-Point Relation: Separating Analyses from Tools

3.1 Overview

The design of Ajax separates analyses, which produce alias information, from tools, which consume the information. This chapter presents a high level functional specification of the interface between tools and analyses. Chapter 4 describes details of the interface which allow analyses and tools to work together efficiently.

3.1.1 Desirability of Simple Semantics

In previous systems, alias information is encoded in formats specific to the analysis used. For example, many analyses compute “points-to” sets. For a pointer variable or expression in a program, such an algorithm computes a static set of abstract locations; each abstract location represents one or more real memory locations that the variable may point to at run time. A tool that interprets points-to sets requires knowledge of the abstraction mapping, which varies from analysis to analysis. Furthermore, in practice, an analysis will compute points-to information for some subset of the pointer variables and expressions in the program; tools need to know exactly which subset, or be able to specify it in advance. If the analysis treats the program in some intermediate form, tools need to understand the same format.

This dependence on details of specific analyses prevents arbitrary combination of analyses with tools. More importantly, it also increases the cost of tool construction even if only one analysis is provided. Tool designers must understand details of the analysis, and this knowledge must be encoded in the tool code.

Therefore, I propose that an interface between tools and analyses should reveal as little as possible of the mechanism of the analysis. The specification of the interface presented to a tool, written out purely in terms of the semantics of the programming language, should be as simple as possible.

3.1.2 The Value-Point Relation

The value-point relation (VPR) is a well-defined abstract property of Java bytecode programs, encoding generalized alias information. The VPR for a given program is static; it summarizes all possible executions of the program. An analysis is required to compute a conservative approximation to the VPR, that is, any relation that includes the VPR.

The VPR is defined directly in terms of the Java bytecode language (“JBC”). A full formal definition would require complete semantics for JBC, the definition of which is beyond the scope of this thesis. Instead, the VPR is defined in terms of a subset language, “Micro” Java bytecode (“MJBC”), for which I provide complete semantics.

3.2 Semantics of the Micro Java Bytecode Language

This section formally defines the semantics of MJBC. Both natural (untagged) and tagged semantics are given. The style is small-step operational semantics.

3.2.1 Preamble

The MJBC language was originally based on Qian’s formalization of a JBC subset [64].

There is no single syntactic entity corresponding to a “JBC program”. At any given moment at run time, there is a set of class files that have been loaded into the virtual machine. New class files could be added at any time, for example, from a user-specified location in the Internet. To avoid issues of unknown code and dynamic loading, the MJBC semantics assume that the set of class files is fixed and that this set constitutes the entire program. I abstract away the class file format and the linkage process, and consider a program to be a tuple of sets and functions representing the information in the class files after parsing and linking.

These sets and functions are described in terms of some basic types:

- *ClassIdentifier*, the type of abstract names for classes.
- *MethodIdentifier*, the type of abstract names for methods.
- *FieldIdentifier*, the type of abstract names for fields.

In the Java Virtual Machine, a ClassIdentifier corresponds to a fully qualified class name paired with a reference to the class loader that loaded it. A MethodIdentifier corresponds to a method signature including a method name, a return type and a list of parameter types (because overloading is resolved at compile time). A FieldIdentifier corresponds to the name of a field paired with the class in which it was declared — an object can have multiple fields of the same name, inherited from different classes.

ClassIdentifier has a distinguished subset ErrorClassIDs, representing the classes of exceptions thrown by the runtime system (e.g. `OutOfMemoryError` or `NullPointerException`).

There are also some frequently used compound types:

- *MethodImpl* = ClassIdentifier $\times$ MethodIdentifier
  Values of this type identify method implementations. The ClassIdentifier is the class that implements the method, and the MethodIdentifier names the implemented method. The following projection functions are useful:

  MethodImplClass(*classID*, *methodID*) = *classID*
  MethodImplName(*classID*, *methodID*) = *methodID*

- *CodeLoc* = MethodImpl $\times$ $\mathbb{Z}$
  This is the type of code locations. The MethodImpl identifies the method body, and the integer is an offset within the method's code. Only non-negative offsets are actually used. The following projection functions are useful:

  CodeLocMethod(*method*, *offset*) = *method*
  CodeLocOffset(*method*, *offset*) = *offset*

The addition operator is overloaded at +: *CodeLoc* $\times$ $\mathbb{Z} \to$ *CodeLoc* as follows:

(*method*, *offset*) + *disp* = (*method*, *offset* + *disp*)

Some of the runtime structures use lists. The empty list is written as "ε" and list consing is written as "::". For example, 3::2::1::ε denotes a list of the first three positive integers.

The empty finite map is written as "[]". The extension of a finite map *M* with a mapping from *k* to *v* is written "$M[k \to v]$".

3.2.2 Programs

A program is a tuple of several components:

- *Main* : MethodImpl
  This is the identifier of the method that starts the program; it is the static method `main` of some class.
- *InitFields* : ClassIdentifier $\mapsto$ (FieldIdentifier $\mapsto$ InitValue)
  This maps each class in the program to the initial values of the fields when an object of that class is created. Thus it encodes which fields are present in any given class as well as their default values (zero for scalars, null for object references). InitFields is not defined for classes which cannot be instantiated (i.e., interfaces or abstract classes). *InitValue* is simply either "0" or "null"; complicated initialization expressions are actually executed in each object's constructor.
- *InitStaticFields* : FieldIdentifier $\mapsto$ InitValue
  This finite map assigns an initial value to each static field in the program.
- *SubclassesOf* : ClassIdentifier $\mapsto$ $\mathbb{P}$(ClassIdentifier)
  This returns the set of subclasses of the class. If the class is actually an interface, its subinterfaces and the classes implementing it are included. The subclass relation is reflexively and transitively closed.
- *Dispatch* : ClassIdentifier $\times$ MethodIdentifier $\mapsto$ MethodImpl
  This partial function maps a class and a method signature to the implementation called when the method is invoked on an object of the given class.
- *Instruction* : CodeLoc $\mapsto$ Inst
  This maps code locations to the instructions at those locations. The set of instructions Inst is described in Figure 3-1. Except as noted, the names of the instructions are the same as the names of their counterparts in the official Java Virtual Machine specification.

Inst ::=`aconst_null`
    | `bipush` *byte*
    | `iadd`
    | `load` *index* (stands for `aload`*, `iload`* forms)
    | `store` *index* (stands for `astore`*, `istore`* forms)
    | `if_cmpeq` *offset* (stands for `if_icmpeq`, `if_acmpeq`)
    | `goto` *offset*
    | `return` (stands for `ireturn`, `areturn`)
    | `new` *classID*
    | `getfield` *fieldID*
    | `putfield` *fieldID*
    | `getstatic` *fieldID*
    | `putstatic` *fieldID*
    | `invokevirtual` *methodID*
    | `invokestatic` *methodImpl*
    | `checkcast` *classID*
    | `instanceof` *classID*
    | `athrow`

Figure 3-1. The Micro Java Bytecode instruction set

- $\mathit{CatchBlockOffset} : \mathrm{CodeLoc} \times \mathrm{ClassIdentifier} \mapsto \mathbb{Z}$
  This partial function gives the code offset of the handler invoked when an exception of a given class is thrown at a specified program point. It is undefined if the exception should be propagated to the calling method. This function is computed from "catch region" information stored in the class files.

The instruction `aconst_null` pushes a null reference onto the working stack. The `bipush` instruction pushes an integer constant onto the stack. The `iadd` instruction pops to integers off the working stack, adds them, and pushes the result back onto the stack. The `load` and `store` instructions are used to move values between the local variable file and the working stack. The instruction `if_cmpeq` branches if the top of the stack is zero. The `goto` instruction transfers control to another instruction within a method. Programs use the `return` instruction to terminate the invocation of the current method and return a value to the caller. The `new` instruction creates a new object instance of the given class. The `getfield` and `putfield` instructions read and write the given field of the object indicated by the reference on top of the working stack. Similar instructions `getstatic` and `putstatic` read and write static fields; no object reference is required. The `invokevirtual` instruction performs a dynamic method call to the method with signature *methodID* as implemented by the object whose reference is the first method parameter. The `invokestatic` instruction performs a static function call to the given method. Both of the method invocation instructions take the top two elements of the working stack as the parameters to the callee method. The `checkcast` instruction tests whether the object referred to by the top of the working stack is a subclass of the class specified in the instruction (or null); if it is, then no action is taken and the object reference remains on the working stack, but if it is not a valid subclass, an exception is thrown. Alternatively, `instanceof` performs a similar check and then stores the result in a boolean value on top of the stack. The check is different because `instanceof` returns false if the argument is null. The `athrow` instruction raises an exception; on entry to the instruction, the top of stack holds a reference to the exception object to be raised.

The instruction set was designed to be an expressive subset of the JVM instructions, with some streamlining, e.g., there are no per-datatype variants of `load`/`store` instructions, and all methods take exactly two parameters. (I chose two parameters because the first parameter is usually the `this` parameter used for dispatch, and for completeness it seems helpful to have another parameter that is not used for dispatch.) Almost all the interesting behaviors of Java bytecode instructions are captured in this instruction set, with the notable omission of bytecode subroutines, which are of no importance in practice.

MJBC does not define any static constraints on the program beyond the syntactic constraints imposed by the above definitions. In this respect it is much more lenient than the JVM. This is useful because it shows that the definitions and proofs presented in this thesis are independent of any particular static type discipline for JVM bytecode.

3.2.3 State

The description of state requires some additional basic types:

- *ObjectReference*, the type of heap locations.
- *NullRef*, the type of the null reference. There is just one value of this type, “null”.

The type of values is defined as:

$$\mathit{Value} = \mathbb{Z} + \text{ObjectReference} + \text{NullRef}$$

There is a natural embedding of InitValue into Value that maps 0 to the 0 in $\mathbb{Z}$, and maps null to the null in NullRef.

The semantic rules require some additional compound types:

- *HeapObj* = ClassIdentifier $\times$ (FieldIdentifier $\mapsto$ Value)
  A heap maps object references to values of this type. Heap objects retain their dynamic class (used to dispatch virtual methods), and the current values of their fields. The following projection functions are useful:
  - HeapObjClass(*classID, fields*) = *classID*
  - HeapObjFields(*classID, fields*) = *fields*
- *StackFrame* = CodeLoc $\times$ Value list $\times$ ($\mathbb{Z} \mapsto$ Value)
  A tuple of the form (pc, $\mathcal{S}$, $\mathcal{L}$) represents the saved state of a calling method.
  - pc is the location of the method call instruction that transferred control to the callee.
  - $\mathcal{L}$ is the saved local variables of the calling method, defined below.
  - $\mathcal{S}$ is the saved working stack of the calling method, defined below.

A program state Ξ is a record of the form

$$[\text{mode: } \mathit{mode}, \text{ pc: } \mathit{pc}, \text{ wstack: } \mathcal{S}, \text{ locals: } \mathcal{L}, \text{ mstack: } \mathcal{J}, \text{ heap: } \mathcal{H}, \text{ globals: } \mathcal{G}]$$

where

- $mode \in \{$ RUNNING, THROWING $\}$
  THROWING indicates that the program is in the process of throwing an exception.
- pc : CodeLoc
  This is the location of the next instruction to be executed.
- $\mathcal{S}$: Value list
  The working stack is used to evaluate expressions, and is local to the currently executing method. When an exception is being thrown, the stack contains a single element — a reference to the exception object being thrown.
- $\mathcal{L} : \mathsf{Z} \mapsto$ Value
  The local variable file is a finite map recording the state of the local variables. In JBC and MJBC, local variables are numbered, not named. In MJBC all methods take two parameters, so on entry to a method, $\mathcal{L}$ has mappings for local variables 0 and 1, holding the actual values of the parameters.
- $\mathcal{J}$: StackFrame list
  This is the method invocation stack, recording the saved state of the methods above the currently executing method in the call stack.
- $\mathcal{H}$: ObjectReference $\mapsto$ HeapObj
  The heap is a finite partial map from object references to the stored objects.
- $\mathcal{G}$: FieldIdentifier $\mapsto$ Value
  The globals are a finite map from each static field (i.e., global variable) to its value.

To make semantic rules shorter and more readable, state records are written in the form

$[elem_1 \rightarrow value_1, ..., elem_n \rightarrow value_n, \rho]$

where ρ is a variable denoting arbitrary values for the additional elements. However, whenever the element mode is given a value by ρ, then the value is required to be RUNNING; this is convenient because most patterns matching a state record are only applicable when the machine is in the RUNNING state.

3.2.4 Initial State

The initial state is

[mode: RUNNING, pc: (Main, 0), wstack: ε, locals: [], mstack: ε, heap: [],
globals: InitStaticFields]

MJBC does not define any notion of termination; it is not needed for the purposes of this thesis.

3.2.5 Transition Rules

The transition relation is a relation over states. It contains an element $\Xi_1 \Rightarrow \Xi_2$ if and only if in one step, the program in state Ξ_1 can progress to state Ξ_2.

In general a given state Ξ_1 can transition to more than one possible Ξ_2, because certain exceptions can be "spontaneously" raised at any time, by transition rule (21). (In the Java Virtual Machine, such exceptions can occur when the virtual machine runs out of memory or encounters some other kind of critical error.) When a program encounters a runtime error (e.g., it tries to pop an empty stack), no normal transition is possible. However, the program is never "stuck" because it can always make a transition by raising a spontaneous exception. This models the raising of exceptions in response to runtime errors — both errors that would normally caught by static checks, and errors that cannot be caught statically such as failed `checkcast` instructions throwing a `ClassCastException`.

The transition rules are given in Figure 3-2.

The exception throwing and handling mechanism requires some explanation. When an exception is thrown (rules (20) and (21)), the current working stack is cleared and a reference to the exception object is pushed onto it. The state switches to THROWING mode. In THROWING mode, at each step, control either transfers to an exception handler within the current method (rule (22)), or leaves the current method to continue exception throwing at the caller (rule (23)). In the latter case, the new *pc* is the location of the method call instruction, rather than its successor as in the case of a normal return. This is necessary for a catch block enclosing the method call instruction to correctly catch the exception. The state switches back to RUNNING mode when the exception is caught by a handler.

3.2.6 Differences between JBC and MJBC

The following features of full JBC have been omitted or abstracted away in MJBC: threads and their associated synchronization operations, arrays, scalar types other than `int`, finite precision/finite bit-width arithmetic, access control (via packages, `public`, `private` and `protected`), native methods, the fact that instructions have variable lengths, complex control instructions such as `lookupswitch` and `tableswitch`, variations on simple instructions such as `wide`, instructions with the same semantics that vary only in the types of their arguments (which exist to aid the Java bytecode verifier), convenience instructions for manipulating the stack such as `dup`, the full suite of arithmetic operators, the specialized method invocation instructions `invokespecial` and `invokeinterface`, methods that return `void`, methods that take more or less than two parameters, bytecode subroutines, the runtime error exceptions thrown by various instructions (e.g., `NullPointerException`), garbage collection and finalization, multiple classloaders, details of the class file format, and dynamic loading.

However, it does have the stack-based instruction set, local variables, integer and object types (with classes and interfaces), exceptions (both explicitly and implicitly thrown) and exception handling, dynamic type checks, and virtual and static methods and fields. The JBC does not have constructors, since these are reduced to method calls at the bytecode level; therefore MJBC does not have constructors either.

The features abstracted away in MJBC to simplify the formal presentation are still handled by the Ajax implementation. Most of the features are straightforward. Chapter 8 discusses issues related to native code and dynamic loading.

The Java Virtual Machine calls the `finalize()` methods on objects as they are garbage collected. This can happen at any time after the object becomes garbage. Ajax models this as a call to `finalize()` on every object that can happen at any time. This is slightly more general than the actual behavior, but none of the implemented or contemplated analyses would be sensitive enough to detect the difference.

$$\frac{\text{Instruction}(pc) = \texttt{aconst\_null}}{[\text{pc: } pc,\ \text{wstack: } \mathcal{S},\ \rho] \Rightarrow [\text{pc: } pc + 1,\ \text{wstack: null} :: \mathcal{S},\ \rho]} \tag{1}$$

$$\frac{\text{Instruction}(pc) = \texttt{bipush}\ byte}{[\text{pc: } pc,\ \text{wstack: } \mathcal{S},\ \rho] \Rightarrow [\text{pc: } pc + 1,\ \text{wstack: } byte :: \mathcal{S},\ \rho]} \tag{2}$$

$$\frac{\text{Instruction}(pc) = \texttt{iadd}}{[\text{pc: } pc,\ \text{wstack: } v_1 :: v_2 :: \mathcal{S},\ \rho] \Rightarrow [\text{pc: } pc + 1,\ \text{wstack: } (v_1 + v_2) :: \mathcal{S},\ \rho]} \tag{3}$$

$$\frac{\text{Instruction}(pc) = \texttt{load}\ index}{[\text{pc: } pc,\ \text{wstack: } \mathcal{S},\ \text{locals: } \mathcal{L},\ \rho] \Rightarrow [\text{pc: } pc + 1,\ \text{wstack: } \mathcal{L}(index) :: \mathcal{S},\ \text{locals: } \mathcal{L},\ \rho]} \tag{4}$$

$$\frac{\text{Instruction}(pc) = \texttt{store}\ index}{[\text{pc: } pc,\ \text{wstack: } v :: \mathcal{S},\ \text{locals: } \mathcal{L},\ \rho] \Rightarrow [\text{pc: } pc + 1,\ \text{wstack: } \mathcal{S},\ \text{locals: } \mathcal{L}[index \to v],\ \rho]} \tag{5}$$

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{if\_cmpeq}\ offset \\ v \neq 0\end{array}}{[\text{pc: } pc,\ \text{wstack: } v :: \mathcal{S},\ \rho] \Rightarrow [\text{pc: } pc + 1,\ \text{wstack: } \mathcal{S},\ \rho]} \tag{6}$$

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{if\_cmpeq}\ offset \\ v = 0\end{array}}{[\text{pc: } pc,\ \text{wstack: } v :: \mathcal{S},\ \rho] \Rightarrow [\text{pc: } pc + offset,\ \text{wstack: } \mathcal{S},\ \rho]} \tag{7}$$

$$\frac{\text{Instruction}(pc) = \texttt{goto}\ offset}{[\text{pc: } pc,\ \rho] \Rightarrow [\text{pc: } pc + offset,\ \rho]} \tag{8}$$

$$\frac{\text{Instruction}(pc) = \texttt{return}}{\begin{array}{c}[\text{pc: } pc,\ \text{wstack: } v :: \mathcal{S},\ \text{locals: } \mathcal{L},\ \text{mstack: } (pc', \mathcal{S}', \mathcal{L}') :: \mathcal{J},\ \rho] \\ \Rightarrow [\text{pc: } pc' + 1,\ \text{wstack: } v :: \mathcal{S}',\ \text{locals: } \mathcal{L}',\ \text{mstack: } \mathcal{J},\ \rho]\end{array}} \tag{9}$$

Figure 3-2. Rules defining the transition relation

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{new}\ \mathit{classID} \\ \mathit{ref} \notin \text{dom}\ \mathcal{H}\end{array}}{\begin{array}{c}[\text{pc: } pc,\ \text{wstack: } \mathcal{S},\ \text{heap: } \mathcal{H},\ \rho] \\ \Rightarrow [\text{pc: } pc + 1,\ \text{wstack: } \mathit{ref} :: \mathcal{S},\ \text{heap: } \mathcal{H}[\mathit{ref} \to (\mathit{classID}, \text{InitFields}(\mathit{classID}))],\ \rho]\end{array}} \tag{10}$$

$$\frac{\text{Instruction}(pc) = \texttt{getfield}\ \mathit{fieldID}}{\begin{array}{c}[\text{pc: } pc,\ \text{wstack: } \mathit{ref} :: \mathcal{S},\ \text{heap: } \mathcal{H},\ \rho] \\ \Rightarrow [\text{pc: } pc + 1,\ \text{wstack: } \text{HeapObjFields}(\mathcal{H}(\mathit{ref}))(\mathit{fieldID}) :: \mathcal{S},\ \text{heap: } \mathcal{H},\ \rho]\end{array}} \tag{11}$$

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{putfield}\ \mathit{fieldID} \\ \mathit{classID} = \text{HeapObjClass}(\mathcal{H}(\mathit{ref})) \\ \mathit{fields} = \text{HeapObjFields}(\mathcal{H}(\mathit{ref})) \\ \mathit{fieldID} \in \text{dom InitFields}(\mathit{classID})\end{array}}{\begin{array}{c}[\text{pc: } pc,\ \text{wstack: } v :: \mathit{ref} :: \mathcal{S},\ \text{heap: } \mathcal{H},\ \rho] \\ \Rightarrow [\text{pc: } pc + 1,\ \text{wstack: } \mathcal{S},\ \text{heap: } \mathcal{H}[\mathit{ref} \to (\mathit{classID}, \mathit{fields}[\mathit{fieldID} \to v])],\ \rho]\end{array}} \tag{12}$$

$$\frac{\text{Instruction}(pc) = \texttt{getstatic}\ \mathit{fieldID}}{[\text{pc: } pc,\ \text{wstack: } \mathcal{S},\ \text{globals: } \mathcal{G},\ \rho] \Rightarrow [\text{pc: } pc + 1,\ \text{wstack: } \mathcal{G}(\mathit{fieldID}) :: \mathcal{S},\ \text{globals: } \mathcal{G},\ \rho]} \tag{13}$$

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{putstatic}\ \mathit{fieldID} \\ \mathit{fieldID} \in \text{dom}\ \mathcal{G}\end{array}}{\begin{array}{c}[\text{pc: } pc,\ \text{wstack: } v :: \mathcal{S},\ \text{globals: } \mathcal{G},\ \rho] \\ \Rightarrow [\text{pc: } pc + 1,\ \text{wstack: } \mathcal{S},\ \text{globals: } \mathcal{G}[\mathit{fieldID} \to v],\ \rho]\end{array}} \tag{14}$$

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{invokevirtual}\ \mathit{methodID} \\ pc' = (\text{Dispatch}(\text{HeapObjClass}(\mathcal{H}(v_0)), \mathit{methodID}), 0)\end{array}}{\begin{array}{c}[\text{pc: } pc,\ \text{wstack: } v_1 :: v_0 :: \mathcal{S},\ \text{locals: } \mathcal{L},\ \text{mstack: } \mathcal{F},\ \text{heap: } \mathcal{H},\ \rho] \\ \Rightarrow [\text{pc: } pc',\ \text{wstack: } \varepsilon,\ \text{locals: } [0 \to v_0,\ 1 \to v_1],\ \text{mstack: } (pc, \mathcal{S}, \mathcal{L}) :: \mathcal{F},\ \text{heap: } \mathcal{H},\ \rho]\end{array}} \tag{15}$$

Figure 3-2. Rules defining the transition relation

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{invokestatic}\ \mathit{methodImpl} \\ pc' = (\mathit{methodImpl}, 0)\end{array}}{\begin{array}{c}[\text{pc: } pc, \text{ wstack: } v_1 :: v_0 :: \mathcal{S}, \text{ locals: } \mathcal{L}, \text{ mstack: } \mathcal{J}, \rho] \\ \Rightarrow [\text{pc: } pc', \text{ wstack: } \varepsilon, \text{ locals: } [0 \to v_0, 1 \to v_1], \text{ mstack: } (pc, \mathcal{S}, \mathcal{L}) :: \mathcal{J}, \rho]\end{array}} \tag{16}$$

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{checkcast}\ \mathit{classID} \\ \mathit{ref} = \text{null} \vee \text{HeapObjClass}(\mathcal{H}(\mathit{ref})) \in \text{SubclassesOf}(\mathit{classID})\end{array}}{[\text{pc: } pc, \text{ wstack: } \mathit{ref} :: \mathcal{S}, \text{ heap: } \mathcal{H}, \rho] \Rightarrow [\text{pc: } pc + 1, \text{ wstack: } \mathit{ref} :: \mathcal{S}, \text{ heap: } \mathcal{H}, \rho]} \tag{17}$$

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{instanceof}\ \mathit{classID} \\ \text{HeapObjClass}(\mathcal{H}(\mathit{ref})) \in \text{SubclassesOf}(\mathit{classID})\end{array}}{[\text{pc: } pc, \text{ wstack: } \mathit{ref} :: \mathcal{S}, \text{ heap: } \mathcal{H}, \rho] \Rightarrow [\text{pc: } pc + 1, \text{ wstack: } 1 :: \mathcal{S}, \text{ heap: } \mathcal{H}, \rho]} \tag{18}$$

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{instanceof}\ \mathit{classID} \\ \mathit{ref} = \text{null} \vee \text{HeapObjClass}(\mathcal{H}(\mathit{ref})) \notin \text{SubclassesOf}(\mathit{classID})\end{array}}{[\text{pc: } pc, \text{ wstack: } \mathit{ref} :: \mathcal{S}, \text{ heap: } \mathcal{H}, \rho] \Rightarrow [\text{pc: } pc + 1, \text{ wstack: } 0 :: \mathcal{S}, \text{ heap: } \mathcal{H}, \rho]} \tag{19}$$

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{athrow} \\ \mathit{ref} \neq \text{null}\end{array}}{\begin{array}{c}[\text{mode: R}\small{\text{UNNING}}, \text{ pc: } pc, \text{ wstack: } \mathit{ref} :: \mathcal{S}, \rho] \\ \Rightarrow [\text{mode: T}\small{\text{HROWING}}, \text{ pc: } pc, \text{ wstack: } \mathit{ref} :: \varepsilon, \rho]\end{array}} \tag{20}$$

$$\frac{\begin{array}{c}\mathit{classID} \in \text{ErrorClassIDs} \\ \mathit{ref} \notin \text{dom}\ \mathcal{H} \\ \mathit{obj} = (\mathit{classID}, \text{InitFields}(\mathit{classID}))\end{array}}{\begin{array}{c}[\text{mode: R}\small{\text{UNNING}}, \text{ pc: } pc, \text{ wstack: } \mathcal{S}, \text{ heap: } \mathcal{H}, \rho] \\ \Rightarrow [\text{mode: T}\small{\text{HROWING}}, \text{ pc: } pc, \text{ wstack: } \mathit{ref} :: \varepsilon, \text{ heap: } \mathcal{H}[\mathit{ref} \to \mathit{obj}], \rho]\end{array}} \tag{21}$$

$$\frac{\mathit{handler} = \text{CatchBlockOffset}((\mathit{method}, \mathit{offset}), \text{HeapObjClass}(\mathcal{H}(\mathit{ref})))}{\begin{array}{c}[\text{mode: T}\small{\text{HROWING}}, \text{ pc: } (\mathit{method}, \mathit{offset}), \text{ wstack: } \mathit{ref} :: \varepsilon, \text{ heap: } \mathcal{H}, \rho] \\ \Rightarrow [\text{mode: R}\small{\text{UNNING}}, \text{ pc: } (\mathit{method}, \mathit{handler}), \text{ wstack: } \mathit{ref} :: \varepsilon, \text{ heap: } \mathcal{H}, \rho]\end{array}} \tag{22}$$

Figure 3-2. Rules defining the transition relation

$$\frac{((\mathit{method}, \mathit{offset}), \text{HeapObjClass}(\mathcal{H}(\mathit{ref}))) \notin \text{dom CatchBlockOffset}}{\begin{array}{c}[\text{mode: THROWING, pc: } \mathit{pc}, \text{ wstack: } \mathit{ref} :: \varepsilon, \text{ locals: } \mathcal{L}, \text{ mstack: } (\mathit{pc}', \mathcal{S}', \mathcal{L}') :: \mathcal{J}, \text{ heap: } \mathcal{H}, \rho] \\ \Rightarrow [\text{mode: THROWING, pc: } \mathit{pc}', \text{ wstack: } \mathit{ref} :: \varepsilon, \text{ locals: } \mathcal{L}', \text{ mstack: } \mathcal{J}, \text{ heap: } \mathcal{H}, \rho]\end{array}} \quad (23)$$

Figure 3-2. Rules defining the transition relation

The most significant issue is threads. Ajax uses the definition of the VPR presented here, but assumes that a program state includes a list of thread stacks, and that the semantics of JBC include non-deterministic context switching transitions. Handling threads has no practical consequences for the implementation of Ajax, because the analyses implemented in Ajax to date are oblivious to the order in which statements are executed (as far as the heap is concerned, which is where all inter-thread interference occurs).

3.3 The Value-Point Relation

3.3.1 Bytecode Expressions

To describe the properties of a program, it is useful to be able to name values such as stack elements and local variables at particular program points. Thus I define a small language of “bytecode expressions”, shown in Figure 3-3.

BExp	::=	*pc* : *BExpPath*
BExpPath	::=	*BExpRoot BExpFields*
BExpRoot	::=	`stack-`*n*
	|	`local-`*n*
	|	*FieldID*
	|	`exn`
BExpFields	::=	`.`*FieldID BExpFields*
	|	ε

Figure 3-3. The language of bytecode expressions

A bytecode expression includes a code location for context; a *BExpRoot* designating a stack element, local variable, static field or currently-throwing exception; and an optional list of fields to be dereferenced. Each *FieldID* is fully qualified by the name of the class the field is declared in.

Given a program state, a bytecode expression can be evaluated to a value. An expression may not evaluate to any value if an object does not have an appropriate field, or a stack or local variable does not exist, or the state’s program counter is not at the location specified in the expression. The rules for evaluating an expression B in state Ξ, giving a partial judgement of the form $(\Xi, B) \twoheadrightarrow v$, are given in Figure 3-4.

$$\frac{}{([\text{mode: }\textsc{Running}, \text{ pc: } pc, \text{ wstack: } v_0 :: \ldots :: v_n :: \mathcal{S}, \rho], pc : \texttt{stack-}n) \dashrightarrow v_n} \qquad (24)$$

$$\frac{\mathcal{L}(n) = v}{([\text{mode: } \mathit{mode}, \text{ pc: } pc, \text{ locals: } \mathcal{L}, \rho], pc : \texttt{local-}n) \dashrightarrow v} \qquad (25)$$

$$\frac{}{([\text{mode: }\textsc{Throwing}, \text{ pc: } pc, \text{ wstack: } v :: \varepsilon, \rho], pc : \texttt{exn}) \dashrightarrow v} \qquad (26)$$

$$\frac{\mathcal{G}(\mathit{staticField}) = v}{([\text{mode: } \mathit{mode}, \text{ pc: } pc, \text{ globals: } \mathcal{G}, \rho], pc : \mathit{staticField}) \dashrightarrow v} \qquad (27)$$

$$\frac{\begin{array}{c}([\text{mode: } \mathit{mode}, \text{ pc: } pc, \text{ heap: } \mathcal{H}, \rho], pc : \mathit{exp}) \dashrightarrow u \\ \text{HeapObjFields}(\mathcal{H}(u))(\mathit{field}) = v\end{array}}{([\text{mode: } \mathit{mode}, \text{ pc: } pc, \text{ heap: } \mathcal{H}, \rho], pc : \mathit{exp} \,.\, \mathit{field}) \dashrightarrow v} \qquad (28)$$

Figure 3-4. Rules defining the evaluation of bytecode expressions

The rule for `stack-`n extracts the n-th element of the stack, if the program is not throwing an exception. The rule for `local-`n extracts the n-th local variable; local variables are available whether or not the program is throwing an exception. The `exn` expression is available only when the program is throwing an exception; the currently throwing exception is stored on the top of the stack. The values of static fields are extracted from the static field map. Field dereference expressions first evaluate the dereferenced expression; if that returns a value, then it is looked up in the heap and the field of the resulting object is extracted.

3.3.2 The Value-Point Relation

A *trace* T of a program P is a sequence of states $<\Xi_0, \ldots, \Xi_n>$ such that Ξ_0 is the initial program state for program P, and $\forall 0 < i \le n.\ \Xi_{i-1} \Rightarrow \Xi_i$.

Let e_1 and e_2 be bytecode expressions. Define the *value-point relation* $\leftrightarrow_P$ of a program P as follows:

$e_1 \leftrightarrow_P e_2$ iff
$\exists$ a trace T of P and states Ξ_i and Ξ_j in T, such that $(\Xi_i, e_1) \dashrightarrow v$ and $(\Xi_j, e_2) \dashrightarrow v$ for some value v, where v is not equal to null.

Informally, two bytecode expressions are related if there is a common value v that both expressions evaluate to. If v is an object reference, then the two expressions are aliased. Such a v is called a *witness value*.

Null values are not permitted as witnesses because aliasing is only induced when the two expressions refer to actual objects.

3.4 Generalizing Alias Analysis Using Tagging

3.4.1 Overview

The VPR as defined above does not only relate expressions yielding object references. It can also relate expressions yielding scalar values (integers, in MJBC). However, computing a sound approximation to the definition above would require analysis of arithmetic, which is difficult to do efficiently. The definition would also not be very useful, because most pairs of expressions take on overlapping ranges of values (including, e.g., zero).

A more useful definition distinguishes expressions having the same value by an accident of arithmetic from expressions yielding values copied from some common source. Conceptually, scalar values can be treated as “boxed” and alias analysis performed on the box objects. This enables tracking of the propagation and use of scalar values as well as objects.

Formally, we construct an “instrumented” semantics for MJBC associating labels with values. The labels, called *tags*, are similar to object references. When a scalar value is “created” by using a constant or performing arithmetic, a fresh tag is generated and associated with the value to form a *tagged value*. Two tagged values may have the same actual value but different tags. For example, two expressions may both evaluate to tagged values of zero, but with different tags, indicating that the values were not obtained from a common source.

Tags on non-null object references are superfluous, because two equal object references must have the same tag; the MJBC semantics never reuse a heap location once it has been allocated. However, all values are tagged for the sake of uniformity.

3.4.2 Tagged State

Tags are drawn from an infinite uninterpreted set, *Tag*.

Tagged values are defined as

- $\underline{\textit{Value}}$ = Value × Tag

The following projection function is useful:

- Val(*value*, *tag*) = *value*

The following derived types follow immediately:

- $\underline{\textit{HeapObj}}$ = ClassIdentifier × (FieldIdentifier $\mapsto$ $\underline{\text{Value}}$)
- $\underline{\textit{StackFrame}}$ = CodeLoc × $\underline{\text{Value}}$ list × ($\mathsf{Z} \mapsto \underline{\text{Value}}$)

A tagged program state is a record of the form

[mode: *mode*, pc: *pc*, wstack: $\underline{\mathcal{S}}$, locals: $\underline{\mathcal{L}}$, mstack: $\underline{\mathcal{J}}$, heap: $\underline{\mathcal{H}}$, globals: $\underline{\mathcal{G}}$, used: *used*]

where

- *mode* : { Running, Throwing }
- *pc* : CodeLoc

- $\underline{\mathcal{S}}$: Value list
- $\underline{\mathcal{L}}$: Z ↦ Value
- $\underline{\mathcal{J}}$: StackFrame list
- $\underline{\mathcal{H}}$: ObjectReference ↦ HeapObj
- $\underline{\mathcal{G}}$: FieldIdentifier ↦ Value
- *used* : P(Tag)
  This part of the state records all the tags that have been allocated so far in the execution. This is used to help generate unique fresh tags. This set is always finite.

I define the projection functions *Mode, PC*, *WStack*, *Locals*, *Globals*, *MStack*, *Heap* and *Used* to return the corresponding component of a tagged state.

The initial tagged state is

[mode: RUNNING, pc: (Main, 0), wstack: ε, locals: [], mstack: ε, heap: [],
globals: InitStaticFields, used: range InitialTags]

where *InitialTags* is any bijection from the domain of InitStaticFields (the static fields used by the program) to some subset of Tag. *InitStaticFields* is defined to have the same domain as InitStaticFields, and

InitStaticFields(*f*) = (InitStaticFields(*f*), InitialTag(*f*))

In other words, in the initial state, every global variable is initialized to zero or null, each with a unique tag.

3.4.3 Tagged Transition Rules

The inference rules defining the tagged transition relation are given in Figure 3-5.

These rules are almost identical to the untagged transition rules. There are two sets of differences. Whenever a new value is created (by `aconst_null`, `bipush`, `iadd`, `new`, `instanceof`, or a runtime exception throw), a fresh tag *t* is chosen nondeterministically and associated with the new value. Also, whenever the actual value of a tagged value is required, a Val projection is inserted.

3.4.4 Correspondence Between Tagged Semantics and Untagged Semantics

Define the function *Untag* from tagged states to untagged states as follows:

Untag([mode: *mode*, pc: *pc*, wstack: $\underline{\mathcal{S}}$, locals: $\underline{\mathcal{L}}$, mstack: $\underline{\mathcal{J}}$, heap: $\underline{\mathcal{H}}$, globals: $\underline{\mathcal{G}}$, used: *used*])
= [mode: *mode*, pc: *pc*, wstack: $\text{Untag}_S(\underline{\mathcal{S}})$, locals: $\text{Untag}_L(\underline{\mathcal{L}})$, mstack: $\text{Untag}_J(\underline{\mathcal{J}})$,
heap: $\text{Untag}_H(\underline{\mathcal{H}})$, globals: $\text{Untag}_G(\underline{\mathcal{G}})$]

In other words, Untag just strips off all the tags from the state.

It is also useful to define $\mathit{Untag}_\rho(\rho)$ to untag partial records ρ.

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{aconst\_null} \\ t \notin used\end{array}}{\begin{array}{c}[\text{pc: } pc, \text{ wstack: } \underline{\mathcal{S}}, \text{ used: } used, \rho] \\ \underline{\Rightarrow}\ [\text{pc: } pc + 1, \text{ wstack: } (\text{null}, t) :: \underline{\mathcal{S}}, \text{ used: } used \cup \{t\}, \rho]\end{array}} \tag{29}$$

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{bipush}\ byte \\ t \notin used\end{array}}{\begin{array}{c}[\text{pc: } pc, \text{ wstack: } \underline{\mathcal{S}}, \text{ used: } used, \rho] \\ \underline{\Rightarrow}\ [\text{pc: } pc + 1, \text{ wstack: } (byte, t) :: \underline{\mathcal{S}}, \text{ used: } used \cup \{t\}, \rho]\end{array}} \tag{30}$$

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{iadd} \\ t \notin used\end{array}}{\begin{array}{c}[\text{pc: } pc, \text{ wstack: } v_1 :: v_2 :: \underline{\mathcal{S}}, \text{ used: } used, \rho] \\ \underline{\Rightarrow}\ [\text{pc: } pc + 1, \text{ wstack: } (\text{Val}(v_1) + \text{Val}(v_2), t) :: \underline{\mathcal{S}}, \text{ used: } used \cup \{t\}, \rho]\end{array}} \tag{31}$$

$$\frac{\text{Instruction}(pc) = \texttt{load}\ index}{[\text{pc: } pc, \text{ wstack: } \underline{\mathcal{S}}, \text{ locals: } \underline{\mathcal{L}}, \rho]\ \underline{\Rightarrow}\ [\text{pc: } pc + 1, \text{ wstack: } \underline{\mathcal{L}}(index) :: \underline{\mathcal{S}}, \text{ locals: } \underline{\mathcal{L}}, \rho]} \tag{32}$$

$$\frac{\text{Instruction}(pc) = \texttt{store}\ index}{[\text{pc: } pc, \text{ wstack: } v :: \underline{\mathcal{S}}, \text{ locals: } \underline{\mathcal{L}}, \rho]\ \underline{\Rightarrow}\ [\text{pc: } pc + 1, \text{ wstack: } \underline{\mathcal{S}}, \text{ locals: } \underline{\mathcal{L}}[index \rightarrow v], \rho]} \tag{33}$$

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{if\_cmpeq}\ offset \\ \text{Val}(v) \neq 0\end{array}}{[\text{pc: } pc, \text{ wstack: } v :: \underline{\mathcal{S}}, \rho]\ \underline{\Rightarrow}\ [\text{pc: } pc + 1, \text{ wstack: } \underline{\mathcal{S}}, \rho]} \tag{34}$$

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{if\_cmpeq}\ offset \\ \text{Val}(v) = 0\end{array}}{[\text{pc: } pc, \text{ wstack: } v :: \underline{\mathcal{S}}, \rho]\ \underline{\Rightarrow}\ [\text{pc: } pc + offset, \text{ wstack: } \underline{\mathcal{S}}, \rho]} \tag{35}$$

$$\frac{\text{Instruction}(pc) = \texttt{goto}\ offset}{[\text{pc: } pc, \rho]\ \underline{\Rightarrow}\ [\text{pc: } pc + offset, \rho]} \tag{36}$$

Figure 3-5. Rules defining the tagged transition relation

$$\frac{\text{Instruction}(pc) = \texttt{return}}{\begin{array}{c}[\text{pc: } pc, \text{ wstack: } v :: \underline{\mathcal{S}}, \text{ locals: } \underline{\mathcal{L}}, \text{ mstack: } (pc', \underline{\mathcal{S}}', \underline{\mathcal{L}}') :: \underline{\mathcal{Q}}, \rho] \\ \Rightarrow [\text{pc: } pc' + 1, \text{ wstack: } v :: \underline{\mathcal{S}}', \text{ locals: } \underline{\mathcal{L}}', \text{ mstack: } \underline{\mathcal{Q}}, \rho]\end{array}} \quad (37)$$

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{new } \mathit{classID} \\ r \notin \text{dom } \underline{\mathcal{H}} \\ \text{dom } \mathit{fields} = \text{dom } \mathit{tags} = \text{dom InitFields}(\mathit{classID}) \\ \forall f \in \text{dom } \mathit{fields} . \mathit{fields}(f) = (\text{InitFields}(\mathit{classID})(f), \mathit{tags}(f)) \\ \underline{\mathcal{H}}' = \underline{\mathcal{H}}[r \to (\mathit{classID}, \mathit{fields})] \\ (\{t\} \cup \text{range } \mathit{tags}) \cap \mathit{used} = \emptyset \\ t \notin \text{range } \mathit{tags} \\ \mathit{tags} \text{ is a bijection}\end{array}}{\begin{array}{c}[\text{pc: } pc, \text{ wstack: } \underline{\mathcal{S}}, \text{ heap: } \underline{\mathcal{H}}, \text{ used: } \mathit{used}, \rho] \\ \Rightarrow [\text{pc: } pc + 1, \text{ wstack: } (r, t) :: \underline{\mathcal{S}}, \text{ heap: } \underline{\mathcal{H}}', \text{ used: } \mathit{used} \cup \{t\} \cup \text{range } \mathit{tags}, \rho]\end{array}} \quad (38)$$

$$\frac{\text{Instruction}(pc) = \texttt{getfield } \mathit{fieldID}}{\begin{array}{c}[\text{pc: } pc, \text{ wstack: } \mathit{ref} :: \underline{\mathcal{S}}, \text{ heap: } \underline{\mathcal{H}}, \rho] \\ \Rightarrow [\text{pc: } pc + 1, \text{ wstack: HeapObjFields}(\underline{\mathcal{H}}(\text{Val}(\mathit{ref})))(\mathit{fieldID}) :: \underline{\mathcal{S}}, \text{ heap: } \underline{\mathcal{H}}, \rho]\end{array}} \quad (39)$$

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{putfield } \mathit{fieldID} \\ \mathit{classID} = \text{HeapObjClass}(\underline{\mathcal{H}}(\text{Val}(\mathit{ref}))) \\ \mathit{fields} = \text{HeapObjFields}(\underline{\mathcal{H}}(\text{Val}(\mathit{ref}))) \\ \mathit{fieldID} \in \text{dom InitFields}(\mathit{classID})\end{array}}{\begin{array}{c}[\text{pc: } pc, \text{ wstack: } v :: \mathit{ref} :: \underline{\mathcal{S}}, \text{ heap: } \underline{\mathcal{H}}, \rho] \\ \Rightarrow [\text{pc: } pc + 1, \text{ wstack: } \underline{\mathcal{S}}, \text{ heap: } \underline{\mathcal{H}}[\text{Val}(\mathit{ref}) \to (\mathit{classID}, \mathit{fields}[\mathit{fieldID} \to v])], \rho]\end{array}} \quad (40)$$

$$\frac{\text{Instruction}(pc) = \texttt{getstatic } \mathit{fieldID}}{[\text{pc: } pc, \text{ wstack: } \underline{\mathcal{S}}, \text{ globals: } \underline{\mathcal{G}}, \rho] \Rightarrow [\text{pc: } pc + 1, \text{ wstack: } \underline{\mathcal{G}}(\mathit{fieldID}) :: \underline{\mathcal{S}}, \text{ globals: } \underline{\mathcal{G}}, \rho]} \quad (41)$$

$$\frac{\begin{array}{c}\text{Instruction}(pc) = \texttt{putstatic } \mathit{fieldID} \\ \mathit{fieldID} \in \text{dom } \underline{\mathcal{G}}\end{array}}{\begin{array}{c}[\text{pc: } pc, \text{ wstack: } v :: \underline{\mathcal{S}}, \text{ globals: } \underline{\mathcal{G}}, \rho] \\ \Rightarrow [\text{pc: } pc + 1, \text{ wstack: } \underline{\mathcal{S}}, \text{ globals: } \underline{\mathcal{G}}[\mathit{fieldID} \to v], \rho]\end{array}} \quad (42)$$

Figure 3-5. Rules defining the tagged transition relation

$$\frac{\begin{array}{c}\mathrm{Instruction}(pc) = \texttt{invokevirtual}\ \mathit{methodID} \\ pc' = (\mathrm{Dispatch}(\mathrm{HeapObjClass}(\underline{\mathcal{H}}(\mathrm{Val}(v_0))), \mathit{methodID}), 0)\end{array}}{\begin{array}{c}[\mathrm{pc}\colon pc,\ \mathrm{wstack}\colon v_1 :: v_0 :: \underline{\mathcal{S}},\ \mathrm{locals}\colon \underline{\mathcal{L}},\ \mathrm{mstack}\colon \underline{\mathcal{Q}},\ \mathrm{heap}\colon \underline{\mathcal{H}},\ \rho] \\ \underline{\Rightarrow}\ [\mathrm{pc}\colon pc',\ \mathrm{wstack}\colon \varepsilon,\ \mathrm{locals}\colon [0 \to v_0,\ 1 \to v_1],\ \mathrm{mstack}\colon (pc, \underline{\mathcal{S}}, \underline{\mathcal{L}}) :: \underline{\mathcal{Q}},\ \mathrm{heap}\colon \underline{\mathcal{H}},\ \rho]\end{array}} \qquad (43)$$

$$\frac{\begin{array}{c}\mathrm{Instruction}(pc) = \texttt{invokestatic}\ \mathit{methodImpl} \\ pc' = (\mathit{methodImpl}, 0)\end{array}}{\begin{array}{c}[\mathrm{pc}\colon pc,\ \mathrm{wstack}\colon v_1 :: v_0 :: \underline{\mathcal{S}},\ \mathrm{locals}\colon \underline{\mathcal{L}},\ \mathrm{mstack}\colon \underline{\mathcal{Q}},\ \rho] \\ \underline{\Rightarrow}\ [\mathrm{pc}\colon pc',\ \mathrm{wstack}\colon \varepsilon,\ \mathrm{locals}\colon [0 \to v_0,\ 1 \to v_1],\ \mathrm{mstack}\colon (pc, \underline{\mathcal{S}}, \underline{\mathcal{L}}) :: \underline{\mathcal{Q}},\ \rho]\end{array}} \qquad (44)$$

$$\frac{\begin{array}{c}\mathrm{Instruction}(pc) = \texttt{checkcast}\ \mathit{classID} \\ \mathrm{Val}(\mathit{ref}) = \mathrm{null} \vee \mathrm{HeapObjClass}(\underline{\mathcal{H}}(\mathrm{Val}(\mathit{ref}))) \in \mathrm{SubclassesOf}(\mathit{classID})\end{array}}{[\mathrm{pc}\colon pc,\ \mathrm{wstack}\colon \mathit{ref} :: \underline{\mathcal{S}},\ \mathrm{heap}\colon \underline{\mathcal{H}},\ \rho]\ \underline{\Rightarrow}\ [\mathrm{pc}\colon pc + 1,\ \mathrm{wstack}\colon \mathit{ref} :: \underline{\mathcal{S}},\ \mathrm{heap}\colon \underline{\mathcal{H}},\ \rho]} \qquad (45)$$

$$\frac{\begin{array}{c}\mathrm{Instruction}(pc) = \texttt{instanceof}\ \mathit{classID} \\ \mathrm{HeapObjClass}(\underline{\mathcal{H}}(\mathrm{Val}(\mathit{ref}))) \in \mathrm{SubclassesOf}(\mathit{classID}) \\ t \notin \mathit{used}\end{array}}{\begin{array}{c}[\mathrm{pc}\colon pc,\ \mathrm{wstack}\colon \mathit{ref} :: \underline{\mathcal{S}},\ \mathrm{heap}\colon \underline{\mathcal{H}},\ \mathit{used},\ \rho] \\ \underline{\Rightarrow}\ [\mathrm{pc}\colon pc + 1,\ \mathrm{wstack}\colon (1, t) :: \underline{\mathcal{S}},\ \mathrm{heap}\colon \underline{\mathcal{H}},\ \mathit{used} \cup \{t\},\ \rho]\end{array}} \qquad (46)$$

$$\frac{\begin{array}{c}\mathrm{Instruction}(pc) = \texttt{instanceof}\ \mathit{classID} \\ \mathrm{Val}(\mathit{ref}) = \mathrm{null} \vee \mathrm{HeapObjClass}(\underline{\mathcal{H}}(\mathrm{Val}(\mathit{ref}))) \notin \mathrm{SubclassesOf}(\mathit{classID}) \\ t \notin \mathit{used}\end{array}}{\begin{array}{c}[\mathrm{pc}\colon pc,\ \mathrm{wstack}\colon \mathit{ref} :: \underline{\mathcal{S}},\ \mathrm{heap}\colon \underline{\mathcal{H}},\ \mathit{used},\ \rho] \\ \underline{\Rightarrow}\ [\mathrm{pc}\colon pc + 1,\ \mathrm{wstack}\colon (0, t) :: \underline{\mathcal{S}},\ \mathrm{heap}\colon \underline{\mathcal{H}},\ \mathit{used} \cup \{t\},\ \rho]\end{array}} \qquad (47)$$

$$\frac{\begin{array}{c}\mathrm{Instruction}(pc) = \texttt{athrow} \\ \mathrm{Val}(\mathit{ref}) \neq \mathrm{null}\end{array}}{\begin{array}{c}[\mathrm{mode}\colon \textsc{Running},\ \mathrm{pc}\colon pc,\ \mathrm{wstack}\colon \mathit{ref} :: \underline{\mathcal{S}},\ \rho] \\ \underline{\Rightarrow}\ [\mathrm{mode}\colon \textsc{Throwing},\ \mathrm{pc}\colon pc,\ \mathrm{wstack}\colon \mathit{ref} :: \varepsilon,\ \rho]\end{array}} \qquad (48)$$

Figure 3-5. Rules defining the tagged transition relation

$$\frac{\begin{array}{c} \mathit{classID} \in \text{ErrorClassIDs} \\ r \notin \text{dom}\,\underline{\mathcal{H}} \\ \text{dom}\,\mathit{fields} = \text{dom}\,\mathit{tags} = \text{dom InitFields}(\mathit{classID}) \\ \forall f \in \text{dom}\,\mathit{fields}.\ \mathit{fields}(f) = (\text{InitFields}(\mathit{classID})(f), \mathit{tags}(f)) \\ \underline{\mathcal{H}}' = \underline{\mathcal{H}}[r \to (\mathit{classID}, \mathit{fields})] \\ (\{t\} \cup \text{range}\ \mathit{tags}) \cap \mathit{used} = \emptyset \\ t \notin \text{range}\ \mathit{tags} \\ \mathit{tags} \text{ is a bijection} \end{array}}{\begin{array}{c} [\text{mode: R}\textsc{unning}\text{, pc: } pc\text{, wstack: } \underline{\mathcal{S}}\text{, heap: } \underline{\mathcal{H}}, \mathit{used}, \rho] \\ \mathrel{\underline{\Rightarrow}} [\text{mode: T}\textsc{hrowing}\text{, pc: } pc\text{, wstack: } (r, t) :: \varepsilon\text{, heap: } \underline{\mathcal{H}}', \mathit{used} \cup \{t\}, \rho] \end{array}} \quad (49)$$

$$\frac{\mathit{handler} = \text{CatchBlockOffset}((\mathit{method}, \mathit{offset}), \text{HeapObjClass}(\underline{\mathcal{H}}(\text{Val}(\mathit{ref}))))}{\begin{array}{c} [\text{mode: T}\textsc{hrowing}\text{, pc: } (\mathit{method}, \mathit{offset})\text{, wstack: } \mathit{ref} :: \varepsilon\text{, heap: } \underline{\mathcal{H}}, \rho] \\ \mathrel{\underline{\Rightarrow}} [\text{mode: R}\textsc{unning}\text{, pc: } (\mathit{method}, \mathit{handler})\text{, wstack: } \mathit{ref} :: \varepsilon\text{, heap: } \underline{\mathcal{H}}, \rho] \end{array}} \quad (50)$$

$$\frac{((\mathit{method}, \mathit{offset}), \text{HeapObjClass}(\underline{\mathcal{H}}(\text{Val}(\mathit{ref})))) \notin \text{dom CatchBlockOffset}}{\begin{array}{c} [\text{mode: T}\textsc{hrowing}\text{, pc: } pc\text{, wstack: } \mathit{ref} :: \varepsilon\text{, locals: } \underline{\mathcal{L}}\text{, mstack: } (pc', \underline{\mathcal{S}}', \underline{\mathcal{L}}') :: \underline{\mathcal{F}}\text{, heap: } \underline{\mathcal{H}}, \rho] \\ \mathrel{\underline{\Rightarrow}} [\text{mode: T}\textsc{hrowing}\text{, pc: } pc'\text{, wstack: } \mathit{ref} :: \varepsilon\text{, locals: } \underline{\mathcal{L}}'\text{, mstack: } \underline{\mathcal{F}}\text{, heap: } \underline{\mathcal{H}}, \rho] \end{array}} \quad (51)$$

Figure 3-5. Rules defining the tagged transition relation

The following two lemmas express the fact that executions in the tagged semantics mirror executations in the untagged semantics.

Lemma 3-1. $\forall \underline{\Xi}_1, \underline{\Xi}_2.\ \underline{\Xi}_1 \mathrel{\underline{\Rightarrow}} \underline{\Xi}_2 \Rightarrow \text{Untag}(\underline{\Xi}_1) \Rightarrow \text{Untag}(\underline{\Xi}_2)$

Lemma 3-2. $\forall \underline{\Xi}_1, \Xi_2.\ \text{Untag}(\underline{\Xi}_1) \Rightarrow \Xi_2 \Rightarrow (\exists \underline{\Xi}_2.\ \text{Untag}(\underline{\Xi}_2) = \Xi_2 \wedge \underline{\Xi}_1 \mathrel{\underline{\Rightarrow}} \underline{\Xi}_2)$

The proofs are by case analysis of the hypothesized transition relation. I present one case for the proof of each lemma to illustrate the form of the proofs.

Proof of Lemma 3-1: Suppose $\underline{\Xi}_1 \mathrel{\underline{\Rightarrow}} \underline{\Xi}_2$ and consider the case in which the transition is justified by the `iadd` rule. From the `iadd` tagged transition rule,

$$\begin{aligned} \underline{\Xi}_1 &= [\text{pc: } pc\text{, wstack: } v_1 :: v_2 :: \underline{\mathcal{S}}\text{, used: } \mathit{used}, \rho] \\ \underline{\Xi}_2 &= [\text{pc: } pc + 1\text{, wstack: } (\text{Val}(v_1) + \text{Val}(v_2), t) :: \underline{\mathcal{S}}\text{, used: } \mathit{used} \cup \{t\}, \rho] \\ \text{Instruction}(pc) &= \texttt{iadd} \end{aligned}$$

Then

$$\begin{aligned} \text{Untag}(\underline{\Xi}_1) &= [\text{pc: } pc\text{, wstack: } \text{Val}(v_1) :: \text{Val}(v_2) :: \text{Untag}_{\mathcal{S}}(\underline{\mathcal{S}}), \text{Untag}_\rho(\rho)] \\ \text{Untag}(\underline{\Xi}_2) &= [\text{pc: } pc + 1\text{, wstack: } \text{Val}(v_1) + \text{Val}(v_2) :: \text{Untag}_{\mathcal{S}}(\underline{\mathcal{S}}), \text{Untag}_\rho(\rho)] \end{aligned}$$

Hence $\text{Untag}(\underline{\Xi}_1) \Rightarrow \text{Untag}(\underline{\Xi}_2)$ as required.

Proof of Lemma 3-2: Suppose $\text{Untag}(\underline{\Xi}_1) \Rightarrow \Xi_2$ and consider the `iadd` case.

$$\begin{aligned}
&\text{Untag}(\underline{\Xi}_1) = [\text{pc: } pc, \text{ wstack: } v_1 :: v_2 :: \mathcal{S}, \rho] \\
&\Xi_2 = [\text{pc: } pc + 1, \text{ wstack: } (v_1 + v_2) :: \mathcal{S}, \rho] \\
&\text{Instruction}(pc) = \texttt{iadd}
\end{aligned}$$

By the definition of Untag, $\underline{\Xi}_1$ must be of the form

$$\underline{\Xi}_1 = [\text{pc: } pc, \text{ wstack: } u_1 :: u_2 :: \underline{\mathcal{S}}, \text{ used: } used, \rho']$$

where

$$\begin{aligned}
&\text{Val}(u_1) = v_1 \\
&\text{Val}(u_2) = v_2 \\
&\text{Untag}_S(\underline{\mathcal{S}}) = \mathcal{S} \\
&\text{Untag}_\rho(\rho') = \rho
\end{aligned}$$

Now let t be any tag such that $t \notin used$. Such a tag always exists because the set of tags is infinite and the *used* set is always finite. Set

$$\underline{\Xi}_2 = [\text{pc: } pc + 1, \text{ wstack: } (\text{Val}(u_1) + \text{Val}(u_2), t) :: \underline{\mathcal{S}}, \text{ used: } used \cup \{t\}, \rho']$$

Then $\text{Untag}(\underline{\Xi}_2) = \Xi_2$ and $\underline{\Xi}_1 \underline{\Rightarrow} \underline{\Xi}_2$, as required.

3.4.5 Correspondence of Traces

Define *Untag$_T$* over traces as follows:

$$\text{Untag}_T(<\underline{\Xi}_0, \ldots, \underline{\Xi}_n>) = <\text{Untag}(\underline{\Xi}_0), \ldots, \text{Untag}(\underline{\Xi}_n)>.$$

Lemma 3-3. For any tagged trace $\underline{T}$, $\text{Untag}_T(\underline{T})$ is a trace. Furthermore, for any trace T, there is a tagged trace $\underline{T}$ such that $\text{Untag}_T(\underline{T}) = T$.

Proof: The proofs are by induction on the length of the traces.

Consider a tagged trace $\underline{T} = <\underline{\Xi}_0, \ldots, \underline{\Xi}_n>$. For $n = 1$, $\text{Untag}_T(\underline{T}) = <\text{Untag}(\underline{\Xi}_0)>$. From the definition of the initial state $\underline{\Xi}_0$, it follows that $\text{Untag}(\underline{\Xi}_0)$ is the inital state for the untagged semantics, hence $<\text{Untag}(\underline{\Xi}_0)>$ is a trace.

For $n > 1$, by the induction hypothesis $<\text{Untag}(\underline{\Xi}_0), \ldots, \text{Untag}(\underline{\Xi}_{n-1})>$ is a trace. It is required to prove that $\text{Untag}(\underline{\Xi}_{n-1}) \Rightarrow \text{Untag}(\underline{\Xi}_n)$. This follows immediately from $\underline{\Xi}_{n-1} \underline{\Rightarrow} \underline{\Xi}_n$ and Lemma 3-1.

Now consider an untagged trace $T = <\Xi_0, \ldots, \Xi_n>$. For $n = 1$, set $\underline{T} = <\underline{\Xi}_0>$ to be the initial state for the tagged semantics. As above, $\text{Untag}_T(\underline{T}) = <\Xi_0> = T$.

For $n > 1$, by the induction hypothesis there exists a tagged trace $\underline{T}' = <\underline{\Xi}_0, \ldots, \underline{\Xi}_{n-1}>$ such that $<\text{Untag}(\underline{\Xi}_0), \ldots, \text{Untag}(\underline{\Xi}_{n-1})> = <\Xi_0, \ldots, \Xi_{n-1}>$. Substituting $\text{Untag}(\underline{\Xi}_{n-1}) = \Xi_{n-1}$ and $\Xi_{n-1} \Rightarrow \Xi_n$ into Lemma 3-2, one obtains $\exists \underline{\Xi}_n.\ \text{Untag}(\underline{\Xi}_n) = \Xi_n \wedge \underline{\Xi}_{n-1} \underline{\Rightarrow} \underline{\Xi}_n$. Setting $\underline{T} = <\underline{\Xi}_0, \ldots, \underline{\Xi}_n>$ then gives the required result.

3.4.6 Defining the VPR Using Tags

Figure 3-6 defines evaluation of bytecode expressions in tagged states. The rules are analogous to the rules for untagged states. The only significant difference is that in Figure 3-6, in the rule for field dereferences, the object expression is evaluated to yield the tagged value (u, t), where u is the actual object reference and t is the tag, and the tag is ignored.

$$\frac{}{([\text{mode: RUNNING, pc: } pc, \text{ wstack: } v_0 :: \ldots :: v_n :: \underline{\mathcal{S}}, \rho], pc : \texttt{stack-}n) \dashrightarrow v_n} \tag{52}$$

$$\frac{\underline{\mathcal{L}}(n) = v}{([\text{mode: } mode, \text{ pc: } pc, \text{ locals: } \underline{\mathcal{L}}, \rho], pc : \texttt{local-}n) \dashrightarrow v} \tag{53}$$

$$\frac{}{([\text{mode: THROWING, pc: } pc, \text{ wstack: } v :: \varepsilon, \rho], pc : \texttt{exn}) \dashrightarrow v} \tag{54}$$

$$\frac{\underline{\mathcal{G}}(staticField) = v}{([\text{mode: } mode, \text{ pc: } pc, \text{ globals: } \underline{\mathcal{G}}, \rho], pc : staticField) \dashrightarrow v} \tag{55}$$

$$\frac{\begin{array}{c}([\text{mode: } mode, \text{ pc: } pc, \text{ heap: } \underline{\mathcal{H}}, \rho], pc : exp) \dashrightarrow (u, t) \\ \text{HeapObjFields}(\underline{\mathcal{H}}(u))(field) = v\end{array}}{([\text{mode: } mode, \text{ pc: } pc, \text{ heap: } \underline{\mathcal{H}}, \rho], pc : exp\,.\,field) \dashrightarrow v} \tag{56}$$

Figure 3-6. Rules defining the evaluation of bytecode expressions in tagged states

A *tagged trace* $\underline{T}$ of a program P is a sequence of tagged states $<\underline{\Xi}_0, \ldots, \underline{\Xi}_n>$ such that $\underline{\Xi}_0$ is the initial program state for program P, and $\forall 0 < i \le n.\ \underline{\Xi}_{i-1} \underline{\Rightarrow} \underline{\Xi}_i$.

Let e_1 and e_2 be bytecode expressions. Define the *value-point relation* $\leftrightarrow_P$ of a program P as follows:

$e_1 \leftrightarrow_P e_2$ iff
$\exists$ a tagged trace $\underline{T}$ of P and tagged states $\underline{\Xi}_i$ and $\underline{\Xi}_j$ in $\underline{T}$, such that $(\underline{\Xi}_i, e_1) \dashrightarrow (u, t)$ and $(\underline{\Xi}_j, e_2) \dashrightarrow (u, t)$ for some tagged value (u, t), where u is not equal to null.

This is the definition actually used in the remainder of the thesis, including the rest of this chapter.

3.5 Examples of Using the Value-Point Relation

This section presents some examples of extracting useful information from the VPR.

3.5.1 Finding Writers to a Field

Consider the following problem:

"Given a program P and the pc of a `getfield` instruction, find all code locations pc' of the `putfield` instructions that put values into the field being read."

This question can be formalized as the following set comprehension:

$\{\ pc' \mid \exists$ a trace T of $P = <\Xi_0, \ldots, \Xi_n>$.
$\exists p, q, objref, \mathcal{S}, val, \mathcal{S}', field, \rho.\ \mathrm{Val}(objref) \neq \mathrm{null} \wedge$
$\Xi_p = [\mathrm{pc}\colon pc, \mathrm{wstack}\colon objref :: \mathcal{S}, \rho] \wedge \mathrm{Instruction}_P(pc) = \texttt{getfield}\, field \wedge$
$\Xi_q = [\mathrm{pc}\colon pc', \mathrm{wstack}\colon val :: objref :: \mathcal{S}', \rho] \wedge \mathrm{Instruction}_P(pc') = \texttt{putfield}\, field\ \}$

This set is equal to

$\{\ pc' \mid \exists field.\ pc : \texttt{stack-0} \leftrightarrow_P pc' : \texttt{stack-1} \wedge$
$\mathrm{Instruction}_P(pc) = \texttt{getfield}\, field \wedge \mathrm{Instruction}_P(pc') = \texttt{putfield}\, field\ \}$

The translation erases all mention of dynamic properties, summarizing them with the static VPR.

3.5.2 Downcast Checking

Consider the following problem:

"Find all program locations *pc* corresponding to `checkcast` instructions which might fail."

This can be formulated as

$\{\ pc \mid \exists$ a trace T of $P = <\Xi_0, \ldots, \Xi_n>.\ \exists p, objref, \mathcal{S}, \mathcal{H}, class, \rho.\ \mathrm{Val}(objref) \neq \mathrm{null} \wedge$
$\Xi_p = [\mathrm{pc}\colon pc, \mathrm{wstack}\colon objref :: \mathcal{S}, \mathrm{heap}\colon \mathcal{H}, \rho] \wedge$
$\mathrm{Instruction}_P(pc) = \texttt{checkcast}\ class \wedge$
$\mathrm{HeapObjClass}(\mathcal{H}(\mathrm{Val}(objref))) \notin \mathrm{SubclassesOf}(class)\ \}$

This can be rewritten to use the value-point relation:

$\{\ pc \mid \exists pc', class, class'.$
$pc : \texttt{stack-0} \leftrightarrow_P pc' : \texttt{stack-0} \wedge$
$\mathrm{Instruction}_P(pc) = \texttt{checkcast}\ class \wedge$
$\mathrm{Instruction}_P(pc'-1) = \texttt{new}\ class' \wedge$
$class' \notin \mathrm{SubclassesOf}(class)\ \}$

In this example, the translation is exact; a downcast is safe if and only if some instruction creates an object which reaches the downcast instruction and which is incompatible with the required bound. Thus, if the true value-point relation is known, the unsafe downcasts can be determined precisely. Of course, in general an analysis can only compute an approximation to the true relation.

3.6 Properties of the Value-Point Relation

The VPR is symmetric. It is not reflexive, because expressions in dead code cannot be related to anything. It is not transitive either, in general. To see this, suppose $B_1 \leftrightarrow_P B_2$ and $B_2 \leftrightarrow_P B_3$. The definition of the VPR implies that for some choice of variables, $(\underline{\Xi}_i, B_1) \twoheadrightarrow \underline{v}$, $(\underline{\Xi}_j, B_2) \twoheadrightarrow \underline{v}$, $(\underline{\Xi}_k, B_2) \twoheadrightarrow \underline{u}$, and $(\underline{\Xi}_l, B_3) \twoheadrightarrow \underline{u}$. The important fact is that it is possible for $\underline{v}$ to not equal $\underline{u}$ (when $\underline{\Xi}_j \neq \underline{\Xi}_k$), so there is no way in general to justify a relationship between B_1 and B_3. For example, consider this fragment of code:

```
if (b) { x = y; } else { x = z; }
```

Let B_1 be `y`, B_2 be `x` and B_3 be `z`, all evaluated after this statement. Then this code may execute once with `b` true, inducing $B_1 \leftrightarrow_P B_2$, and then execute again with `b` false, inducing $B_2 \leftrightarrow_P B_3$, but `y` need never equal `z`.

The VPR does not explicitly encode any information about data dependence or the direction of data flow. $B_1 \leftrightarrow_P B_2$ means that B_1 and B_2 can get the same value, but nothing is revealed about whether the value appears at B_1 or B_2 first. In fact, it may be that no def-use chain leads from B_1 to B_2 or vice versa — they may both be at the end of def-use chains leading back to a common source. However, it is possible to make inferences about data dependence in an important common case: when one of the Bs corresponds to the result of a value creation operation, such as the result of a `new` instruction. In this case it is clear that the value originated at the creation operation. This seems to be sufficient for many applications. Defining a relation representing true directional data dependence would require a much more complicated definition than for the VPR.

The VPR has limited context information. For example, if $B_1 \leftrightarrow_P B_2$ and the bytecode expressions are both located in the same method, there is no way to determine whether the two states justifying the relationship actually occur during the same call to the method or during different calls to the method. For some applications, such as alias analysis for code motion, the tool is only interested in finding aliases that appear during the same call to a method, or even during the same iteration of a loop. Thus, these applications suffer a loss of accuracy using the VPR.

The VPR is simple and does not encode information about context, or scalar values, or control dependence, or many other aspects of program behavior that can be captured by static analysis. However, all these aspects can be used to improve the accuracy of an implementation of a VPR analysis. For example, although the VPR itself encodes only limited context information, SEMI uses context sensitive analysis to produce a better VPR approximation.

The VPR is undecidable. In general, an analyzer can only compute a conservative approximation to the VPR. As stated above, a conservative approximation is simply any relation whose pairs are a superset of the pairs of the true relation. In this thesis, I write an approximation relation for program P as $\overline{\leftrightarrow}_P$.

3.7 Extensions

Many tools would benefit from the ability to specify tighter context constraints, such as the *MayEqual* formulation of Boyland and Greenhouse [12]. This is an obvious candidate for future work.

Other tools require slightly different semantics for the value-point relation. For example, for some applications it is useful to consider values to be related if they are ever compared. This could be added to the dynamic semantics by having comparisons unify the tags of the operands. Static analyses would then have to be adjusted to compute the correct relationships. Ajax has been adapted to this task, but that work is beyond the scope of this thesis. Other applications require the computed VPR approximation to satisfy certain structural invariants, so that the tool can perform its own processing efficiently. An example of this is the object modelling tool in Chapter 11.

The trace *T* in the definition of the VPR is required to range over all possible executions of the program, which implies that any truly conservative approximation to the VPR will be a static analysis. However, if that requirement is relaxed so that *T* only ranges over some given finite set of executions (e.g. some actual runs of the program that were recorded), then the VPR can be computed by dynamic analysis. The “dynamic VPR” can be used by the same set of tools as the static version, except that the results of the tools must be interpreted more carefully; they are true only for the executions recorded.

4 Efficient Queries over the Value-Point Relation

4.1 Introduction

In the previous chapter, I defined the value-point relation as an abstraction of a program, generated by some analysis and consumed by some tool. That discussion focused on the mathematical properties of the relation. In practice, the analysis cannot simply compute an explicit relation and pass it to the tool, because the relation is infinite. Instead, the tool must pass certain parameters to the analysis indicating which parts of the relation must be computed. In fact, for efficiency, some of the tool's computations over the relation often need to performed by the analysis on the tool's behalf, in order to exploit analyis-specific structure. These computations are also expressed as parameters to the analysis.

The nature of this parameterization determines which analysis and tool combinations will be efficient in practice. In this chapter, I describe the parameters supported by Ajax and their motivation. I also describe some general strategies used by analyses and tools to exploit the parameters.

4.2 Analysis Parameters

The following sections explain the issues that need to be addressed by the parameterization scheme, and how each issue is addressed in Ajax. Section 4.2.5 summarizes the parameters.

4.2.1 Restricting the Domain of the Value-Point Relation

Any realistic program admits an infinite number of different bytecode expressions. For example, for any *n* one can form a meaningful expression involving a sequence of *n* field dereferences. The value-point relation is defined over all pairs of bytecode expressions — not just those that appear in the program — and therefore the relation is infinite. In practice, however, tools generally only consider a finite number of bytecode expressions.

Therefore, the simplest and most important parameter is a restriction on the domain of the relation. A tool restricts the domain by explicitly specifying two sets of bytecode expressions, *sources S* and *targets T*. The analysis computes the value-point relation projected onto S × T. Because the sets are given explicitly, they must be finite.

Section 3.5.1 showed how a tool could use the VPR to find all writers to a field. That tool would set

$S = \{ pc : \texttt{stack-0} \}$
$T = \{ pc' : \texttt{stack-1} \mid \text{Instruction}_P(pc') = \texttt{putfield}\, \textit{field} \}$

The example in Section 3.5.2 determines whether a field is always empty. It uses

$$\begin{aligned}
S &= \{\, pc : \texttt{stack-0}.\mathit{field} \,\}\\
T &= \{\, pc' : \texttt{stack-0} \mid \mathrm{Instruction}_P(pc'-1) = \texttt{new}\ \mathit{class} \vee\\
&\qquad \mathrm{Instruction}_P(pc'-1) = \texttt{instanceof}\ \mathit{class} \vee\\
&\qquad \mathrm{Instruction}_P(pc'-1) = \texttt{iadd} \vee\\
&\qquad (\mathrm{Instruction}_P(pc'-1) = \texttt{bipush}\ n \wedge n \neq 0) \,\}
\end{aligned}$$

The downcast checking example in Section 3.5.2 would set

$$\begin{aligned}
S &= \{\, pc : \texttt{stack-0} \mid \mathrm{Instruction}_P(pc-1) = \texttt{new}\ \mathit{class} \,\}\\
T &= \{\, pc' : \texttt{stack-0} \mid \mathrm{Instruction}_P(pc') = \texttt{checkcast}\ \mathit{class} \,\}
\end{aligned}$$

Since the value-point relation is symmetric, the source and target sets are interchangeable at this point in the exposition. The extensions described below break this symmetry.

4.2.2 Avoiding Explicit Products

The downcast checking example shows that, for some applications, both the S and T sets are likely to be proportional in size to the size of the program. If the analysis generates an explicit projection of the relation into $S \times T$, the size of the result could grow quadratically in the size of the program — especially if the analysis is not very precise.

However, many tools postprocess the projected relation to compute some final result that is much smaller than the relation itself. For example, the downcast checker computes just one bit of information per element of T — whether or not the downcast is safe. Furthermore any scalable analysis must be able to represent its internal data in space subquadratic in the size of the program. For efficiency, Ajax maps the tool's computation directly onto the internal data structures of the analysis, without requiring an explicit representation of the VPR approximation. Of course this must be done with only minimal assumptions about the form of that structure.

To this end, I adapted and generalized an idea from Heintze and McAllester's work on subtransitive control flow analysis [41]. The idea is to suppose that the implementation of the analysis builds a directed graph *G* with the following properties:

- There is a map G_S from S to the nodes of G.
- There is a map G_T from T to the nodes of G.
- The analysis indicates $s \overline{\leftrightarrow}_P t$ if and only if there is path from $G_S(s)$ to $G_T(t)$ in G.

In Chapter 5 and Section 6.6 I explain how such a graph is constructed by RTA and SEMI respectively.

Many tools can exploit this graph structure. Suppose a tool needs to compute:

$$\{(t, F[\{s \in S \mid s \overline{\leftrightarrow}_P t\}]) \mid t \in T\}$$

where F is some function specific to the tool. Then if F satisfies a certain *lattice-like* condition described below, the set of results can be computed by exploiting the graph. Conceptually, each node corresponding to a source *s* is first associated with an initial value F[{ *s* }]. These values are then propagated along the graph edges and merged when they meet at nodes. The result for each target *t* is read from the final value associated with the node corresponding to *t*. This process is similar in flavor to dataflow analysis.

For example, consider the downcast checking tool. Let the function F be defined as:

F[{ pc_1:`stack-0`, pc_2:`stack-0`, ..., pc_n:`stack-0` }]
= the most specific common superclass of the classes instantiated at
$pc_1-1, pc_2-1, ..., pc_n-1$

Consider the code in Figure 4-1. A simple dataflow analysis would produce the graph in Figure 4-2.

```
static void main() {
    Object a = new Integer();
    Object b = new String("Hello");
    Object c = new String("Kitty");
    Object d;
    if (...) { d = a; } else { d = b; }
    Object e;
    if (...) { e = b; } else { e = c; }
    Object f;
    if (...) { f = d; } else { f = e; }
    Object h = a;
    Object i = e;
    (Integer)h;
    (Integer)f;
    (String)i;
}
```

Figure 4-1. Example of Java code exhibiting aliasing

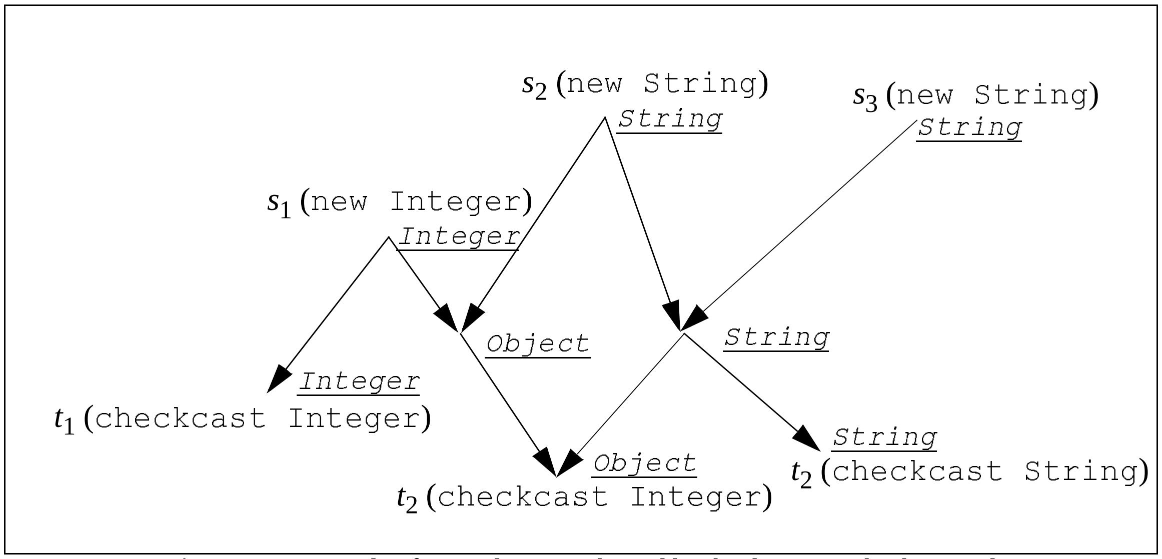


Figure 4-2. Example of an analysis graph used by the downcast checking tool

The downcast checking system finds three `new` instructions in the program, corresponding to s_1, s_2, and s_3, and three `checkcast` instructions, corresponding to t_1, t_2, and t_3, as shown. For each node *N* in the graph, it computes F applied to the set of the s_i that reach *N*.

This can be done efficiently because the value of F at each node (other than a source node) can be computed from the F of its predecessors in the graph — it is the most specific common superclass of the classes at the predecessors. The computed F values are underlined.

Once the downcast checker has determined the most specific common superclass of the classes of the objects that may reach a given downcast instruction, it compares that superclass with the bound specified in the `checkcast` instruction. If the actual superclass is a subclass of the bound (or equal to it) then the cast cannot fail. If the actual superclass is not a subclass of the bound, then the analysis has identified at least one class whose objects appear to reach the downcast instruction but which is not compatible with the bound. For more details, see Chapter 10.

This approach improves efficiency because the space required is only linear in the size of the analysis' graph, instead of proportional to the product of the size of S and the size of T.

It is tempting to assign semantics to the graphs. For example, it seems natural to interpret Figure 4-2 as a dataflow graph, in which objects of various classes flow from their creation sites to the sites of the downcast instructions, and the nodes represent intermediate sites in def-use chains. This interpretation may be correct for some analyses, but it would be mistaken in general. Without referring to a specific analysis, all one can say about the graphs is that they are encodings of the computed VPR approximation, as defined above — "$s \overline{\leftrightarrow}_P t$ if and only if there is path from $G_S(s)$ to $G_T(t)$ in G".

4.2.3 General Framework

The *lattice-like* property required of F is quite simple. There must exist a binary function D_M such that, for any two sets of source bytecode expressions *P* and *Q*,

$$F[P \cup Q] = D_M(F[P], F[Q])$$

The existence of this *merge operator* ensures that the result of F can be constructed incrementally.

Rather than passing graph structures from analyses to tools across the Ajax interface, Ajax tools pass their F functions to the analyses. This reduces the burden on tool implementors.

A tool reveals its F function to analyses by passing in the following parameters:

- The type *D* of *intermediate data* — F's result type
- The merge operator $D_M : D \times D \rightarrow D$
- The *identity* $D_E = F[\{\}]$
- The *initial assignment* $D_I : S \rightarrow D$, such that $D_I(s) = F[\{ s \}]$

These parameters fully determine F, for F can be computed as follows:

$$F[\{\ \}] = D_E$$
$$F[\{s\} \cup Q] = D_M(D_I(s), F[Q])$$

The correctness of this computation follows from the lattice-like property of F, by induction over the size of F's argument set.

The lattice-like property imposes several conditions on these parameters. In the proofs below I assume that F is surjective, i.e., that for every element *d* of D there is a set *P* such that F[*P*] = *d*. This is ensured by an appropriate choice of D.

- D_M must be commutative:

  $$D_M(F[P], F[Q]) = F[P \cup Q] = F[Q \cup P] = D_M(F[Q], F[P])$$

- D_M must be associative:

  $$D_M(F[P], D_M(F[Q], F[R])) = F[P \cup (Q \cup R)] = F[(P \cup Q) \cup R]$$
  $$= D_M(D_M(F[P], F[Q]), F[R])$$

- D_M must be idempotent:

  $$D_M(F[P], F[P]) = F[P \cup P] = F[P]$$

- D_E must be an identity for D_M:

  $$F[Q] = F[\{\} \cup Q] = D_M(F[\{\}], F[Q]) = D_M(D_E, F[Q])$$

In practice, it has not been difficult to identify the appropriate F function and D parameters for each tool. In fact, a small set of F functions has proved to be sufficient for a variety of tools. Many tools use the same F function and distinguish themselves by varying the S and T sets. Some examples are shown below in Section 4.3.

4.2.4 Tool Target Data

Sections 4.2.2 and 4.2.3 describe how analyses compute F-values for each expression in the target set T. However, the expressions T themselves are generally of no interest to a tool. For example, the downcast checker is only interested in the location of the downcast instruction. Therefore each tool specifies a map T_R associating *tool target data* with each target expression. The analysis computes

$$\{(d, F[\{s \in S \mid \exists t \in T . s \overline{\leftrightarrow}_P t \wedge T_R(t) = d\}]) \mid d \in \text{range } T_R\}$$

To compute a result for a given tool target datum, the analysis merges the results for all target expressions associated with the datum.

In the absence of tool target data, most tools would need to maintain their own maps from target expressions to data they find meaningful. The tool target data mechanism factors out this code into a shared module. Target data are also useful when a tool associates the same datum with more than one expression, because merging is automatically performed. The Ajax live code detector exploits this feature, as explained in Section 4.3.5 below.

4.2.5 Summary of Analysis Parameters

This is the final list of parameters:

- A finite set S of source expressions
- A finite set T of target expressions
- A function F described by four parameters:

  - A type D of intermediate data
  - A merge operator $D_M : D \times D \to D$ satisfying the conditions of Section 4.2.3
  - An identity D_E satisfying the conditions of Section 4.2.3
  - An initial assignment $D_I : S \to D$
- A type R of target data
- A tool target data map $T_R : T \to R$

The analysis defines

$$F[\{\ \}] = D_E$$
$$F[\{s\} \cup Q] = D_M(D_I(s), F[Q])$$

The analysis then computes the result of the query:

$$\{(d, F[\{s \in S \mid \exists t \in T . s \overleftrightarrow{\to}_P t \wedge T_R(t) = d\}]) \mid d \in \text{range } T_R\}$$

4.3 Examples

4.3.1 Finding Writers to a Field

Section 3.5.1 presents an example VPR query to find which instructions write values into a field. This query only needs to determine which target expressions are related to a given single source expression. The output of the tool is a list of the locations of those expressions.

The query parameters are simple. The function F returns true if the input set is non-empty (i.e., contains the source expression) and false otherwise.

$S = \{ pc : \texttt{stack-0} \}$
$T = \{ pc' : \texttt{stack-1} \mid \text{Instruction}_P(pc') = \texttt{putfield}\, \mathit{field} \}$
$D = \{ \text{true}, \text{false} \}$
$D_M(a, b) = a \vee b$
$D_E = \text{false}$
$D_I(pc : \texttt{stack-0}) = \text{true}$
$R = \text{CodeLoc}$
$T_R(pc' : \texttt{stack-1}) = pc'$

The analysis returns “true” for the program locations whose target expressions are related to the source expression. The tool prints out these locations.

4.3.2 Finding Unused Fields

The tool discussed in Section 3.5.2 determines whether a given `getfield` instruction always returns zero or null. Consider an extension of that tool to check all `getfield` instructions simultaneously. This tool needs to compute one bit of information for each `getfield` instruction, so we make the `getfield` instructions the targets.

$S = \{ pc' : \texttt{stack-0} \mid \text{Instruction}_P(pc'-1) = \texttt{new}\ \textit{class} \vee$
$\text{Instruction}_P(pc'-1) = \texttt{instanceof}\ \textit{class} \vee$
$\text{Instruction}_P(pc'-1) = \texttt{iadd} \vee$
$(\text{Instruction}_P(pc'-1) = \texttt{bipush}\ n \wedge n \neq 0) \}$
$T = \{ pc : \texttt{stack-0}\,.\textit{field} \mid \text{Instruction}_P(pc) = \texttt{getfield}\,\textit{field} \}$
$D = \{ \text{true}, \text{false} \}$
$D_M(a, b) = a \vee b$
$D_E = \text{false}$
$D_I(pc' : \texttt{stack-0}) = \text{true}$
$R = \text{CodeLoc}$
$T_R(pc : \texttt{stack-0}\,.\textit{field}) = pc$

Similarly to the previous example, the analysis returns “true” for the locations whose target expressions are related to any of the source expressions. These are the locations of the `getfield` instructions that might not return zero or null. The tool outputs the locations for which the analysis returns “false”.

4.3.3 Downcast Checking

These are the analysis parameters for the downcast checker:

$S = \{ pc : \texttt{stack-0} \mid \text{Instruction}_P(pc-1) = \texttt{new}\ \textit{class} \}$
$T = \{ pc' : \texttt{stack-0} \mid \text{Instruction}_P(pc') = \texttt{checkcast}\ \textit{class} \}$
D is the class lattice for *P* (see below)
D_M is the join operation in D
D_E is the bottom element in D
$D_I(pc : \texttt{stack-0}) = \textit{class}$, where $\text{Instruction}_P(pc-1) = \texttt{new}\ \textit{class}$
$R = \text{CodeLoc}$
$T_R(pc' : \texttt{stack-0}) = pc'$

The *class lattice* for program *P* is *P*’s Java class hierarchy, including interfaces, extended to form a lattice. The standard class hierarchy does not form a lattice for two reasons. It does not have a “bottom” element to serve as the identity for a join operation, and therefore we add a synthetic bottom element. Also, two classes may not have a unique most specific common superclass, such as classes `ClassP` and `ClassQ` in the hierarchy of Figure 4-3.

To complete the lattice, we add elements representing the intersections of sets of classes and interfaces. In this example, the most specific common superclass of `ClassP` and `ClassQ` is the synthetic *intersection class* “`ClassA` $\cap$ `InterfaceB`”.

For each `checkcast` instruction, the result of the analysis is the most specific common superclass of all the classes of objects subjected to the `checkcast` instruction. If this superclass is a subclass (or equal to) the bound specified in the `checkcast` instruction, then the downcast is safe, otherwise it may fail.

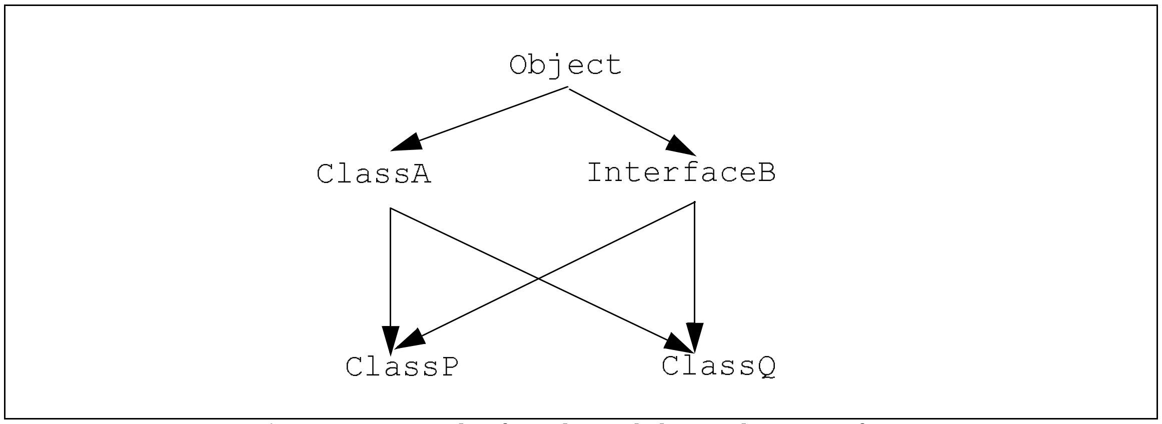


Figure 4-3. Example of non-lattice behavior due to interfaces

4.3.4 Method Call Resolution

Consider a tool designed to resolve dynamic method calls through a given method signature *M*. For each dynamic method call site, the tool determines whether there is exactly one possible callee, and if so, which method it is. Dynamic method call sites with only one possible callee can be converted into direct calls by a compiler, resulting in faster method call code and possible inlining of the callee.

Because the tool computes information for each call site, the call sites are the targets. (In general, whenever the tool's query can be phrased in the form "for every X, compute Y", the choices for X determine the set of targets T.) At each site, the target expression is the object reference upon which the call is dispatched. The source expressions are the results of the `new` instructions that create objects implementing *M*. By determining which of those sources are related to the receiving object at a call site, the call can be resolved, or found to be unresolvable.

Instead of collecting the complete list of source expressions related to each target, it is more efficient to extract just the salient information. We associate with each source expression the method implementing *M* in the new object. The tool collects the set of methods reaching each call site.

Observe that if a set of callee methods at a call site has more than one element, then the call cannot be statically resolved and the exact contents of the set are not used. Therefore each set can be abstracted to one of the following values:

- The empty set, indicating that there is no receiving object. This implies that the call site is in dead code or the receiving object reference is always null.
- A singleton method, indicating that there is at most one receiving method implementation. The call site can be resolved to the given method.
- The value "many", indicating that the set of possible method implementations may have more than one element. The call site cannot be resolved to a single method.

This abstraction is essentially the optimization proposed by Heintze and McAllester [41].

Let $Implementors_P(M)$ denote the set of all methods implementing *M*. The tool uses the following parameters:

S = { pc : `stack-0` | $\text{Instruction}_P(pc-1)$ = `new` *class* }

T = { pc' : `stack-1` | $\text{Instruction}_P(pc')$ = `invokevirtual` *M* } (`stack-1` refers to the receiving object in the call to *M*)

D = { $\emptyset$, many } $\cup$ $\text{Implementors}_P(M)$

$D_M(\emptyset, x) = D_M(x, \emptyset) = x$

$D_M(\text{many}, x) = D_M(x, \text{many}) = \text{many}$

$D_M(x, x) = x$

$D_M(x, y) = \text{many}$, when $x \neq y$

$D_E = \emptyset$

$D_I(pc$: `stack-0`$)$ = *impl*, where $\text{Instruction}_P(pc-1)$ = "`new` *class*", and *class*'s implementation of *M* has identifier *impl*

R = CodeLoc

$T_R(pc'$: `stack-`$n) = pc'$

The tool outputs a D value for each `invokevirtual` instruction specifying method signature *M*. If the value is $\emptyset$, then the instruction is never reached. If the value is "many", then the instruction cannot be statically resolved. Otherwise the value is the name of the only possible callee method.

Section 4.4.1 describes how this tool is extended to examine all `invokevirtual` instructions simultaneously.

4.3.5 Live Code Detection

Consider a tool to find the live implementations of a given method signature *M*. Such a "live code detector" is rather similar to the method call resolver in the previous section, because proper identification of which methods are live requires some resolution of dynamic method calls. However, the live code detector collects information about methods rather than call sites. Therefore the tool target data are the method implementations; the result returned for each method is "true" if it may be live, or "false" if it must be dead. The parameters are:

S = { pc' : `stack-`n | $\text{Instruction}_P(pc')$ = `invokevirtual` *M* }, where *n* is the index of the receiving object in the list of parameters of a call to *M*

T = { pc : `stack-0` | $\text{Instruction}_P(pc-1)$ = `new` *class* }

D = { true, false }

$D_M(a, b) = a \vee b$

D_E = false

$D_I(pc'$: `stack-`$n)$ = true

R = CodeLoc

$T_R(pc$: `stack-0`$)$ = *impl*, where $\text{Instruction}_P(pc-1)$ = "`new` *class*", and *class*'s implementation of *M* has identifier *impl*

In a sense, this query propagates "liveness" from call sites to method implementations, whereas the method call resolver propagates method implementations to call sites.

This is an example of a tool which associates the same target datum with more than one target expression. A method implementation is live if *M* is invoked on any object which inherits that method implementation.

The analysis specified here does not detect all live methods. Calls to static methods must be detected separately. In Java, there is also an `invokespecial` instruction which calls non-static methods using static dispatch.

4.4 Additional Features of the Ajax Implementation

4.4.1 Query Families and Query Fields

The examples in Sections 4.3.4 and 4.3.5 show how to perform method call resolution or live code detection for a specific method signature *M*. To perform these tasks for all method signatures, it suffices to perform a separate query for each signature encountered in the program. Other tools also need to make many queries varying only their S, T, D_I, and T_R parameters.

For greater efficiency and convenience, Ajax allows the remaining parameters — R, D, D_M, and D_E — to be treated as a unit, a *query family*. Each query family defines an index type, I, so that queries belonging to each query family are indexed by elements of I. In the examples above, the elements of I are the method signatures *M*. Ajax is designed to allow a query family to easily manipulate its collection of queries through the index elements. Each instance of an analysis can efficiently support many different query families and many queries within each family.

4.4.2 Incrementality

Ajax is highly incremental. New code can be added to the analyzed program at any time, in response to program modifications or environmental changes. The results of the analyses and tools are updated to reflect the dynamic changes. This requires two elaborations of the VPR interface presented in this chapter.

The query parameters S, T, D_I, and T_R cannot be explicitly stated *a priori*, because the sum of “all the code that might ever be live” is ill-defined or impractically large (for example, it includes the entire Java class library, which is very large). Therefore whenever a new method is added to the “live program,” the Ajax system calls back into the tool, notifying it of the existence of the new method. The tool responds by extending its S, T, D_I, and T_R parameters with the expressions whose locations are in the new method. The analyses must be capable of handling such dynamic updates to the parameters. For the Ajax analyses, this was tricky to implement but not conceptually difficult.

Expressions in dead methods are not related to any other expressions, even themselves. Therefore, if a tool is never notified of the existence of a method, the results for target expressions in that method are trivially equal to D_E. In practice, tools have special handling for unreachable source or target expressions. In the “find writers to a field” example, if the source expression specifying the field is in unreachable code, it is preferable to report that fact to the user rather than to report that there are no writers to the field.

Since the results of an analysis can change when the analyzed program changes, results are reported to a tool using a callback. When the analysis computes a new result for a tool target datum, it reports the datum and result pair to the tool through the callback. In fact, the analyses report results even before the analysis is complete; this results can be superceded by subsequent callbacks. Ajax makes no guarantees of any relationship between these “progressive results” and the final result for a target datum. However, the progressive results can be used for advisory purposes, such as displaying progress to a user. When an analysis completes, it signals the tool that the last reported results for each tool target datum are sound.

4.4.3 Code Mutation

Ajax supports changes being made to the program during analysis, and even after analysis has completed. If analysis has already completed, then the results are updated progressively until completion is signalled again. Many tools are not persistently attached to the program being analyzed, and terminate after the first complete results have been delivered.

The implementation of code mutation is quite simple: for each changed, live method, another “live method” notification is sent to the analysis. It is up to each analysis to decide how to handle multiple live method notifications for a single method. The analyses implemented in Ajax generate new constraints for the new code and add them to the existing set of constraints (i.e., old constraints are not revoked). This is simple and does not penalize the common case in which code is not mutated.

4.4.4 Analysis Scoping

No analysis for Java can attempt to analyze all available code, because the standard libraries are so large that performance would be unacceptable. The code to be analyzed must be identified as part of the analysis. A natural approach is to compute a fixed point from below: start by assuming that just one “main” method is live, analyze it, discover other methods that may be called, add those to the set of live methods, analyze those new methods, and so on.

Ajax’s incremental analysis makes this simple. A live code detection tool is instantiated, just as described in Section 4.3.5. It maintains a set of methods currently thought to be live; This set is initialized to a “main” method by the tool environment. The analysis then runs and reports results to the live code detection tool, which adds new live methods to the live method set. The analysis is notified of these new live methods, computes new results, reports them to the tools, and the cycle continues. This means that typically an Ajax system is configured with two tools: a live method detection tool to control the scope of the analysis, and the tool that the user is actually interested in.

This approximation of the set of live methods from below is frequently seen in prior work, for example RTA [9]. Ajax extends this work by factoring out the approximation and applying it to any analysis.

4.4.5 Intersection

A natural extension of the framework presented above is to extend the operations on the intermediate data D to make it a true lattice; i.e., to provide a *meet* operator D_N corresponding to set intersection. This requires an additional lattice-like property of the tool's F function:

$$\mathrm{F}[P \cap Q] = \mathrm{D_N}(\mathrm{F}[P], \mathrm{F}[Q])$$

This is useful for analyses that compute two or more different, but individually sound, approximations to the value-point relation. The intersection of two sound approximations to the true relation is also a sound approximation to the true relation. In other words, given relations $\overline{\leftrightarrow}_{1P}$ and $\overline{\leftrightarrow}_{2P}$, the relation $\overline{\leftrightarrow}_P$ defined as $s \overline{\leftrightarrow}_P t \equiv s \overline{\leftrightarrow}_{1P} t \wedge s \overline{\leftrightarrow}_{2P} t$ is a sound approximation to the truth, and potentially more accurate than either of the input relations.

Now consider implementing the Ajax interface with such an analysis, and computing the F values for a tool:

$$\begin{aligned}
&\{(t, \mathrm{F}[\{s \in S \mid s \overline{\leftrightarrow}_P t\}]) \mid t \in T\} \\
&= \{(t, \mathrm{F}[\{s \in S \mid s \overline{\leftrightarrow}_{1P} t \wedge s \overline{\leftrightarrow}_{2P} t\}]) \mid t \in T\} \\
&= \{(t, \mathrm{F}[\{s \in S \mid s \overline{\leftrightarrow}_{1P} t\} \cap \{s \in S \mid s \overline{\leftrightarrow}_{2P} t\}]) \mid t \in T\} \\
&= \{(t, \mathrm{D_N}(\mathrm{F}[\{s \in S \mid s \overline{\leftrightarrow}_{1P} t\}], \mathrm{F}[\{s \in S \mid s \overline{\leftrightarrow}_{2P} t\}])) \mid t \in T\}
\end{aligned}$$

Therefore, it suffices to compute the F values for the two relations separately and then apply the meet operator.

It is straightforward to implement a functor that takes a set of Ajax analyses and combines them in this way. Of course, tools must provide a suitable meet operator. The examples above which use boolean values as their intermediate data can use the boolean "and" operator as the meet.

The example using the Java class lattice explicitly represents the meet of two classes as an "intersection class" of the two classes. The representation of intersection classes can often be simplified by exploiting facts about the Java class hierarchy. For example, an intersection class containing two non-interface classes is empty unless one of the classes is a (possibly indirect) superclass of the other, because multiple inheritance is only allowed for interfaces.

Of the examples in this chapter, the method call resolution tool presents the most difficulties in defining a suitable meet operator. The problem is that when both of the operands of the meet are "many", the precise result cannot be determined. The operator must return "many". This is a safe approximation, but the analysis parameters that we introduced for efficiency are now causing us to lose information. For example, the sets { M_1, M_2 } and { M_2, M_3 } both map to the abstract value "many"; their intersection could be represented with the abstract singleton { M_2 }, but this cannot be computed from the abstract values alone. In this situation, the results returned to the tool may vary from run to run depending on the order of analysis computations, even if the underlying analyses compute the same VPR approximations in each run.

5 Implementing the Value-Point Relation With RTA

5.1 Introduction

5.1.1 Introduction to Rapid Type Analysis

Bacon and Sweeney proposed *Rapid Type Analysis* [9] as a fast algorithm for resolving dynamic method calls in statically typed object oriented programs; it was originally applied to C++ programs. RTA uses static type information to resolve dynamic method calls as follows: given a virtual call to method *m* of object reference *v*, find C_v, the static class of *v*, and compute the set *S* of all subclasses of C_v, including C_v itself. Soundness of the static type implies that these classes are a superset of the possible classes that *v* can have at run-time. Therefore if every class in *S* implementing *m* uses the same implementation of *m*, the call can be statically resolved to that implementation.

As described, this is also known as *Class Hierarchy Analysis* [32]. However, RTA adds an important extension to improve accuracy without harming efficiency. Consider the Java program in Figure 5-1.

```
abstract class Super {
    abstract void m();
    static int n;
}
class Sub1 extends Super {
    void m() { n = 1; }
}
class Sub2 extends Super {
    void m() { n = 2; }
}
class Main {
    void f() { new Sub2(); }
    void main(String[] args) { Super v = new Sub1(); v.m(); }
}
```

Figure 5-1. A simple Java program

CHA determines that `v` has two possible implementations of `m`, one from `Sub1` and one from `Sub2`, and therefore the call `v.m()` cannot be resolved. However, RTA observes that the method `f()` is never called and no object of class `Sub2` is ever created, and therefore `v`'s only possible implemention of `m` is from `Sub1`; the call is resolved.

In this example, RTA starts by assuming that `Main.main` is the only live method and that no classes are instantiated. It examines the body of `Main.main` and discovers that `Sub1` is instantiated and there is a dynamic method call to `Super.m`. At this point `Sub1` is the only class in the set of instantiated classes, so the only possible implementor of `Super.m` is `Sub1.m`, which is added to the live method set. Then `Sub1.m` is examined, which does not add any new methods or instantiated classes. Now that all the live methods have been examined, the algorithm terminates.

The efficacy of CHA is based on the observation that in most object oriented programs, many overridable methods in fact have only one implementation. These include methods in an abstract interface that has only one implementation, and methods in a class that has no subclasses. RTA extends CHA to exploit the fact that even when there is more than one implementation available, many programs will only use one implementation.

Both the RTA and CHA algorithms were originally tailored to the problem of resolving dynamic method calls. In Ajax, the technique underlying RTA is generalized away from any particular problem and used to generate VPR information in response to arbitrary queries. For example, the Ajax implementation of RTA can be used to produce information similar to that produced by the "type based alias analysis" of Diwan et al. [23].

By decoupling the analysis from its applications, Ajax makes differences between analyses more apparent. For example, it becomes clear that Diwan et al.'s basic "type based alias analysis" is actually slightly less precise than RTA, because it lacks an analogue of "exact class types" (see Section 5.2.4). The differences were previously obscured because both the analyses and their applications varied in tandem.

5.1.2 Decomposing RTA in Ajax

In Ajax, RTA is restructured into four distinct activities:

1. Computation of the set of live methods
2. Computation of the set of instantiated classes
3. Construction of an approximation to the value-point relation using static type information and the set of instantiated classes
4. Application of the value-point relation to determine the callees of dynamic method calls

Section 4.4.4 explains how for all analyses, Ajax computes a live method set using a bottom-up fixpoint procedure, just as RTA does. This subsumes the first and fourth activities above.

Computing the set of instantiated classes from the set of live methods is trivial. We simply scan the method bodies for occurrences of the `new` instruction and note the class parameter of each such instruction.

The subject of this chapter is the third activity: using static type information and knowledge of the set of instantiated classes to implement the Ajax analysis interface.

Section 5.2 describes how this information is used to approximate the value-point relation. Section 5.3 shows how to structure the computation to support the efficient analysis parameters described in Section 4.2. The chapter concludes with discussion of some extensions.

5.2 Approximating the Value-Point Relation

5.2.1 Overview

Abstractly, the task of any Ajax analysis is to determine whether a given pair of bytecode expressions (B_1, B_2) is in the value-point relation. The decision must be conservative; if there is any uncertainty, the analysis must assume that the pair is in the relation. The RTA analysis receives as input a set L of the methods in the program that it must assume to be live. It also has access to the program, so it can compute the class hierarchy.

The basic idea is to find static types for B_1 and B_2, and then compare the types to decide whether it is possible for a value to conform to both of them simultaneously. These two steps are elucidated in the next two subsections.

In this section I discuss the analysis in the context of full Java bytecode rather than the MJBC subset language, because MJBC does not define a static type discipline analogous to the Java Virtual Machine's "verification" procedure and the Java type system. RTA depends on the existence and soundness of such a type system.

5.2.2 Types for Bytecode Expressions

Each bytecode expression B_i is a pair (l_i, e_i) consisting of a program location l_i and an expression e_i to be evaluated at that location. In principle, it is not difficult for Ajax RTA to compute static types for the expressions, because the Java Virtual Machine computes them while type checking Java bytecode [48].

A full explanation of Java bytecode type reconstruction and verification is beyond the scope of this thesis. Such an explanation can be found in references such as the Java Virtual Machine Specification [48]. Simply put, the type reconstruction algorithm performs intra-procedural dataflow analysis, propagating facts about the types of values along data flow paths. The sources of type information are type annotations on the bytecode instructions.

Ajax RTA has some requirements that are not met by the standard bytecode verification algorithm.

- Ajax RTA differs from the standard JVM verifier in the way it merges object types at control flow merge points. In order to obtain slightly better accuracy for RTA, instead of moving up the class hierarchy to the most specific common superclass of the classes being merged, Ajax creates a union type of the two types. For example, suppose `Sub1` and `Sub2` are both subclasses of class `Super`. If a stack element has object type `Sub1` along one path and type `Sub2` along another path, the standard Java verifier will give the element type `Super` at the point where the paths merge. Ajax will give the element the set of types { `Sub1`, `Sub2` }, interpreted as the union of those two types. If `Super` has additional subclasses, then this union type is more precise than the type `Super`.
- The use of polymorphic bytecode subroutines can require an assignment of more than one possible type to a value-point. In particular, if the location is within a subroutine and the expression refers to a local variable that the subroutine does not touch, the subroutine may be called from multiple contexts that give different types to that variable. Ajax RTA uses dataflow analysis to compute union types for this case.

- Expressions may denote local variables or stack elements in contexts where they have not yet been initialized. In this case the “union set” of types is set to be empty, which eventually causes the analysis to report that such expressions are not related to any expression.
- For an expression denoting the field of an object, Ajax RTA simply uses the declared type of the field. (Field names in a bytecode expression are always fully qualified with the name of the class declaring the field, and are therefore unambiguous.) Therefore Ajax computes a valid type even if the expression refers to a field of an uninitialized variable. This behavior is sound, although it may lead to unnecessary pairs in the VPR approximation. In practice accuracy does not suffer, because tools do not use such expressions. (Java bytecode verification usually ensures that code cannot use unitialized variables, and tools usually refer to variables at instructions where they are used or defined.)
- Where the constant *null* occurs in the bytecode, we assign it the empty type set, because null values do not induce relationships in the VPR.

5.2.3 Computing the Relation

Suppose two expressions B_1 and B_2 have union sets of Java bytecode types S_1 and S_2 respectively. If they are related in the VPR, then at run-time there is a non-null value v appearing at both expressions. Thus, v must conform to at least one static type from S_1 and at least one static type from S_2. Ajax checks all pairs of types (s_1, s_2) in $S_1 \otimes S_2$ to see if there could be such a v conforming to both types s_1 and s_2. If such a pair does not exist, then there can be no relationship between the expressions; otherwise RTA assumes they are related and includes the pair in its VPR approximation. This strategy is efficient in practice because each set usually contains only one element; the special cases of polymorphic subroutines and merging different object types are rare. If one of the sets is empty, the algorithm yields the correct result: the expressions are not related.

Now the problem has reduced to the following: given two Java bytecode types s_1 and s_2, can there be a non-null run-time value conforming to both s_1 and s_2?

To determine the answer, Ajax constructs a directed acyclic graph representing the hierarchy of Java bytecode types. Figure 5-2 is an example. There is a root, TOP, the supertype of all other types. The primitive types `int`, `long`, `float`, and `double` are all distinct. There is a special type for bytecode return addresses, which arise when the Java `try`/`finally` construct is compiled into bytecode `jsr` and `ret` instructions. The Java class hierarchy is inserted into the type graph, rooted at class `Object`. Interfaces such as `Serializable` are also treated as types, which means that classes can have multiple direct supertypes, as shown by `String` and `Component` in the example. Each type representing a class (but not an interface) is labelled to indicate whether or not any objects with that dynamic class can actually be created by the program. In the example, the instantiated types are shown in bold. Primitive types and return addresses are always considered to be instantiated.

If a run-time value conforms to static types s_1 and s_2, then its “run-time type” must be an instantiated type. Therefore the intersection of the subgraphs rooted at s_1 and s_2 must contain at least one instantiated type. In other words, if there is no instantiated type

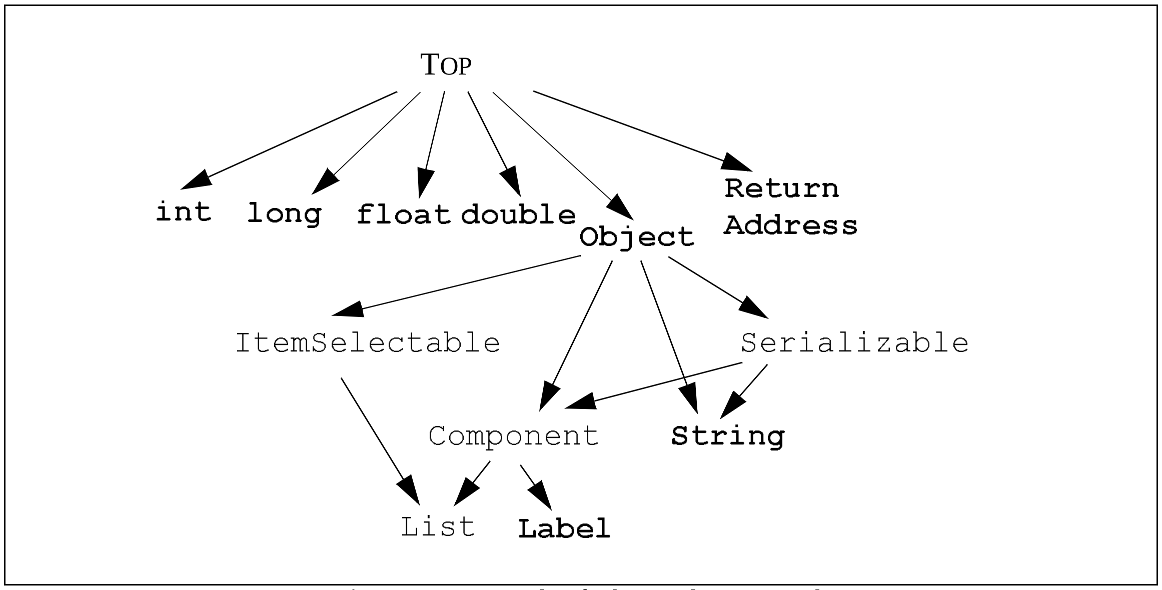


Figure 5-2. Example of a bytecode type graph

reachable from both s_1 and s_2, then no non-null run-time value can conform to both s_1 and s_2.

Figure 5-2 shows that no non-null value conforms to both `ItemSelectable` and `Serializable`, nor `Object` and `Return Address`. On the other hand, there may be a non-null value conforming to both `Serializable` and `Component`; it must be a `Label`.

The smaller primitive types `boolean`, `byte`, `short` and `char`, do not occur in the graph because the Java Virtual Machine treats them as `int`s internally; the precise type is significant only when the value is loaded or stored in an object field or array. Therefore Ajax RTA treats these types as identical to `int`.

Array types require special treatment. Every array type (e.g. `String[]`) has an associated class in the Java bytecode, but the array classes do not capture the full subtyping properties of arrays. Every array class is a subclass of `Object`, `Cloneable`, and `Serializable`, so every array type is a subtype of these types. However, every array of type `T[]` is also a subtype of `S[]` when `T` is a subtype of `S`. (This subtyping relationship is not semantically reasonable — in fact it is unsound without dedicated run-time checks — but the Java Virtual Machine does allow a variable with static type `S[]` to refer to an object of type `T[]`.) These covariant subtyping relationships are not reflected in the JBC class hierarchy. Ajax RTA adds these relationships to the graph separately.

The Top type is included because some situations arise where the type of an expression is not known. This can happen when expressions refer to native code specifications — see Section 8.3.5.

5.2.4 Exact Class Types

In general, when a variable with a class type C occurs in a Java bytecode program, we conclude that its value is an object of class C or any subclass of C. However, when the variable is the direct result of a `new` operation, we know that it is precisely the class specified in the `new` instruction. In this case, we give the variable an *exact class type* "C-Only". The only values conforming to this static type are objects of class C and no other.

This extension is necessary in order for Ajax RTA to be as accurate as traditional RTA. To see this, suppose Ajax RTA is used with the type graph of Figure 5-2 to resolve the dynamic method call `s.hashCode()` in the program fragment in Figure 5-3.

```
void f(String s, Object o) {
    s.hashCode();
    o.hashCode();
}
... x = new Object(); y = new String(); z = new Label();
...
```

Figure 5-3. A fragment illustrating the need for exact class types

The query tries to resolve the method call by collecting all classes C such that the result of a "`new` C" instruction is related to the variable `s`. Those classes are the possible receivers of the method call.

Without exact class types, the static type of `s` is `String`, and the static types of `x` and `y` are `Object` and `String` respectively. Because `Object` and `String` can have a non-null value in common (namely, any `String`), Ajax RTA would conclude that `s` is related to both sites, and therefore both `Object` and `String` can receive the method call. Because they have different implementations of `hashCode`, the call to `s.hashCode()` would not be resolved.

With exact class types, the static type of `s` is still String, but the static types of `new Object` and `new String` are the exact class types "`Object`-Only" and "`String`-Only". `Object`-Only does not have any non-null values in common with `String`. Therefore, the only `new` site matching `s` is `new String`, and the call is resolved as expected.

The changes to the type graph are simple: Every inexact class type C that is instantiated gains a new subtype, "C-Only". C-Only has no subtypes and its sole supertype is C. The instantiation annotations are changed to indicate that exact class types are instantiated directly but inexact class types are not. The graph in Figure 5-2 is transformed into the graph of Figure 5-4.

5.3 Implementing the Ajax Analysis Interface

The previous section specifies the approximation to the value-point relation computed by Ajax RTA. This section describes an efficient implementation of the Ajax analysis interface using this approximation.

Recall that the interface specifies the following parameters to the analysis:

- A type D of intermediate data to be propagated
- A type R of tool target data

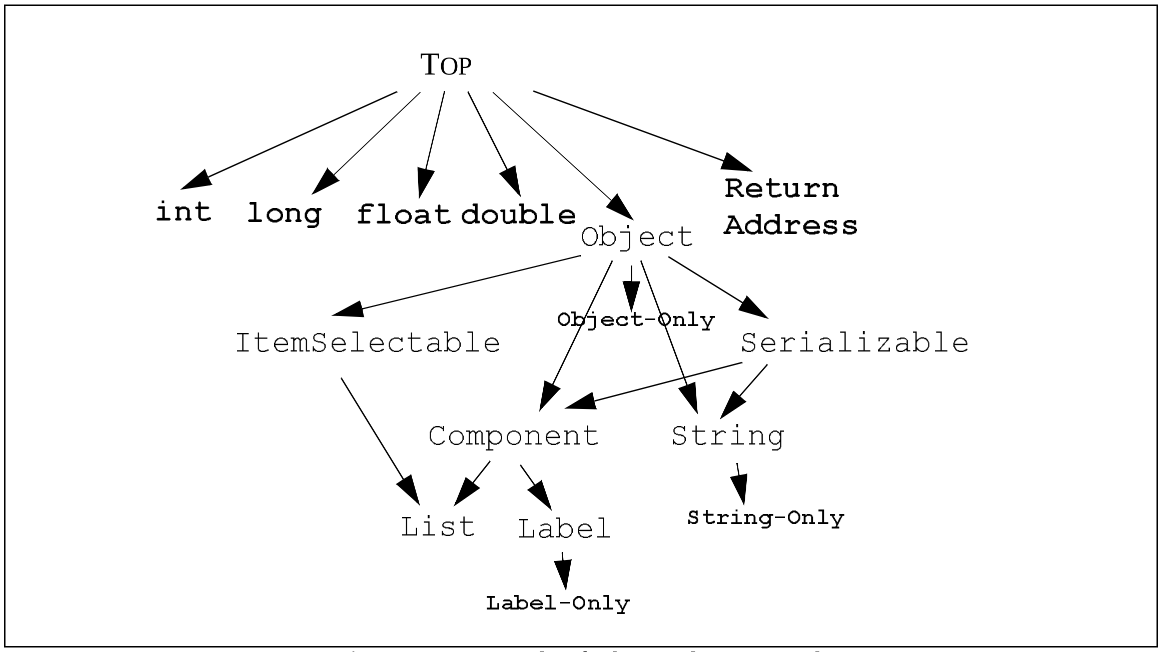


Figure 5-4. Example of a bytecode type graph

- An associative, commutative, idempotent binary "merge" operator $D_M : D \times D \rightarrow D$ with identity element D_E
- A set S of source expressions from which data will be propagated
- A set T of target value-points to which data will be propagated
- An initial assignment of intermediate data to source expressions $D_I : S \rightarrow D$
- A map from target expressions to tool target data $T_R : T \rightarrow R$

The analysis computes:

$$\{(d, F[\{s \in S \mid \exists t \in T . s \overline{\leftrightarrow}_P t \wedge T_R(t) = d\}]) \mid d \in \text{range } T_R\}$$

where

$$F[\{\ \}] = D_E$$
$$F[P \cup Q] = D_M(F[P], F[Q])$$
$$F[\{s\}] = D_I(s)$$

This is computed efficiently using an extension of the subtype graph.

5.3.1 The Data Propagation Graph

Suppose that the original type graph given above consists of types Y with a subtype relation Y_{sub}. (If y_1 has a subtype y_2 then $(y_1, y_2) \in Y_{sub}$.) Let Y_I be the subset of the Y which are actually instantiated. Ajax RTA constructs a new *propagation graph* with nodes

$$P_N = \{\text{In-}t \mid t \in Y\} \cup \{\text{Out-}t \mid t \in Y\}$$

and edges

$$\begin{aligned} P_E &= \{(\text{In-}y_1, \text{In-}y_2) \mid (y_1, y_2) \in Y_{sub}\} \\ &\cup \{(\text{Out-}y_2, \text{Out-}y_1) \mid (y_1, y_2) \in Y_{sub}\} \cup \{(\text{In-}y, \text{Out-}y) \mid y \in Y_I\} \end{aligned}$$

Informally, we make a copy of the subtype graph, flip the copy upside down, and then paste it below the original graph with edges connecting original nodes to their copies, but only for the nodes corresponding to types that are actually instantiated. The graph in Figure 5-4 is transformed into the graph shown in Figure 5-5.

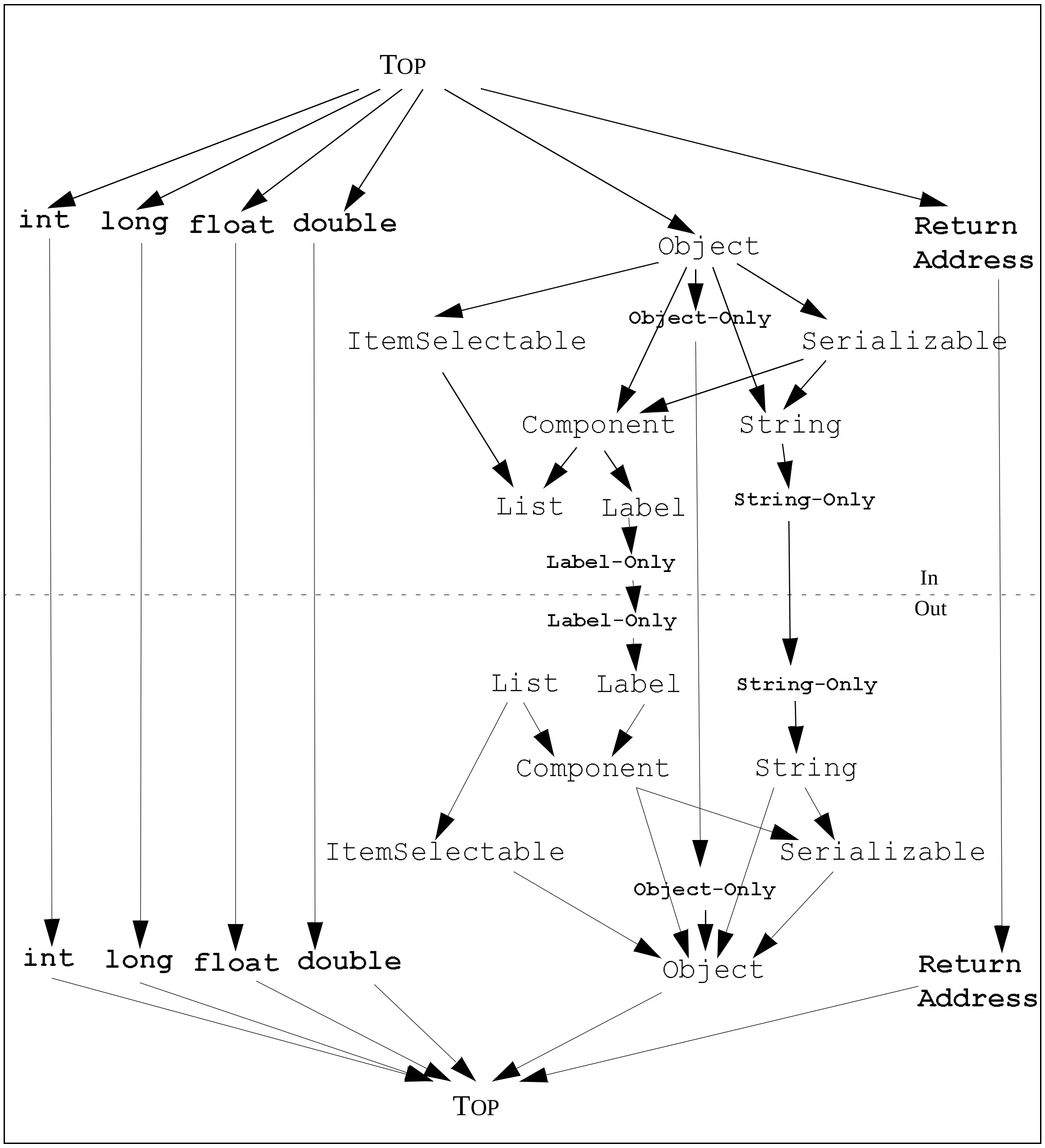


Figure 5-5. Example of a propagation graph

Lemma: Let ":" be the relation between expressions and their RTA types, as explained in Section 5.2.2. RTA relates $s \overline{\leftrightarrow} t$ if and only if there is a path from In-j_s to Out-j_t where $s : j_s$ and $t : j_t$.

Proof: The RTA approximation to the value-point relation defines $s \overline{\leftrightarrow} t$ to mean that there is an instantiated type *w* and types j_s, j_t such that *w* is a subtype of j_s and j_t, $s : j_s$ and $t : j_t$. This implies that in the original type graph there is a path from j_s to *w* and from j_t to *w*. Thus in the propagation graph there is a path from In-j_s to In-*w* and from Out-*w* to Out-j_t. There is an edge from In-*w* to Out-*w* because *w* is instantiated. Thus there is a path from In-j_s to Out-j_t.

Now suppose there is a path from In-j_s to Out-j_t where $s : j_s$ and $t : j_t$. There must exist an edge in the path connecting In-*w* to Out-*w′* for some *w* and *w′*. All such edges are of the form $(\text{In-}y, \mathit{Out}\text{-}y)$ where *y* is an instantiated type, therefore $w = w'$ and *w* is an instantiated type. Furthermore there is a path from In-j_s to In-*w*; this path passes only through In nodes (because there are no edges from any Out node back to an In node). This implies that there is a path from j_s to *w* in the original graph, which means *w* is a subtype of j_s. Likewise, the path from Out-*w* to Out-j_t implies there is a path from j_t to *w* in the original graph, meaning *w* is also a subtype of j_t. Combining all these facts about *w* shows that RTA will conclude $s \overline{\leftrightarrow} t$.

5.3.2 Computing Analysis Results

Now Ajax computes an assignment A of intermediate data D to the nodes of the propagation graph, satisfying the following for all nodes *y*:

$$A(y) = F\{D_I(s) \mid s \in S \wedge \mathit{PathFrom}(\text{In-}j_s, y) \wedge s{:}j_s\}$$

The idea is to start by assigning the initial data to each associated node, and then propagate the data along the graph edges, merging the incoming data at each node. An example is given below.

Ajax computes A iteratively as follows:

$$A_0(y) = F\{D_I(s) \mid s \in S \wedge \mathit{In}\text{-}j_s = y \wedge s{:}j_s\}$$
$$A_{n+1}(y) = F(\{A_n(p) \mid (p, y) \in P_E\} \cup \{A_n(y)\})$$

Initially A is set to the initial data associated with the In nodes. At each iteration, the value at each node is updated from the values at all the node's predecessors. The loop terminates when $_{n+1}(y) = A_n(y)$.

The result of the analysis is then:

$$\{(d, F[\{A(j_t) \mid \exists t \in T.T_R(t) = d \wedge t{:}j_t\}]) \mid d \in \text{range } T_R\}$$

For each tool target datum *d*, this last pass collects and merges the values from each graph node associated with a target expression associated with *d*.

The correctness of this result follows immediately from the lemma in Section 5.3.1.

5.3.3 Example

Consider the problem of determining the callees of the dynamic method calls in the program fragment in Figure 5-3, using the graph in Figure 5-5. The query is set up as follows:

An intermediate datum is a set of implementations of `hashcode`. The class `Label` inherits its `hashcode` method from `Object`, and therefore there are only two distinct implementations of `hashcode`: `Object.hashCode` and `String.hashcode`.

$D = \mathbb{P}(\{$ `Object.hashCode`, `String.hashCode` $\})$
$D_M = \cup$
$D_E = \varnothing$
S = { `x` at statement `x = …`, `y` at statement `y = …`, `z` at statement `z = …` }
T = { `s` at statement `s.hashCode()`, `o` at statement `o.hashCode()` }
R = { statement `s.hashCode()`, statement `o.hashCode()` }
T_R maps each expression to the statement it occurs in

The initial datum assignment maps the result of each `new` instruction to the implementation of `hashcode` used by the created object:

D_I = [`x` → { `Object.hashCode` }, `y` → { `String.hashCode` },
`z` → { `Object.hashCode` }]

The initial A is

A_0 = [In-Object-Only → { `Object.hashCode` },
In-String-Only → { `String.hashCode` },
In-Label-Only → { `Object.hashCode` }]

All types not explicitly mapped are mapped to the empty set.

These values are propagated down the graph, using set union to merge them at nodes with multiple incoming edges. The final value of A is:

A = [In-Object-Only → { `Object.hashCode` },
In-String-Only → { `String.hashCode` },
In-Label-Only → { `Object.hashCode` },
Out-Object-Only → { `Object.hashCode` },
Out-String-Only → { `String.hashCode` },
Out-Label-Only → { `Object.hashCode` },
Out-Label → { `Object.hashCode` },
Out-Component → { `Object.hashCode` },
Out-String → { `String.hashCode` },
Out-Serializable → { `String.hashCode` },
Out-Object → { `Object.hashCode`, `String.hashCode` },
Out-TOP → { `Object.hashCode`, `String.hashCode` }]

Thus Ajax RTA determines that the call to `s.hashCode` has possible receivers A(Out-String) = { `String.hashCode` }, and the call to `o.hashCode` has possible receivers A(Out-Object) = { `Object.hashCode`, `String.hashCode` }. That is, the statement `s.hashCode()` will always call the implementation in the `String` class (and could be replaced by a static method call), but the statement `o.hashCode()` may call the implementation in the `String` class or the implementation in the `Object` class.

5.3.4 Performance

Ajax RTA implements the above algorithm using a worklist. The number of steps required is simply the number of times an element of A is changed. Typically a tool chooses its D_M operator so that the data at a node can only change a small number of times before reaching a fixed point. If D_M is thought of as a lattice join operator, then the tool should choose a lattice with a small height. If the height is indeed bounded by a small constant, then the time to compute A's fixed point is proportional to the size of the propagation graph, which is roughly proportional to the size of the program. If the sizes of the S and T sets are also proportional to the size of the program, the whole algorithm runs in linear time.

Quantitative performance measurements of this implementation of RTA are presented in Section 9.4.

5.3.5 Incrementality

The algorithm described here is quite simple. However, the implementation is nontrivial because many of the inputs are updated dynamically, and the analysis must update its results dynamically in response. In particular:

- The live method set can increase at any time, which means that new classes may be found to have instances.
- The set of classes in the program can increase at any time, as they are loaded on demand. This means that classes can acquire new subclasses.
- At any time, a tool can add to its S set and T set and corresponding D_I and T_R entries.

None of these issues have a major impact on performance, but they significantly complicate the implementation, because new nodes and edges are added to the propagation graph during processing.

5.4 RTA++: Tracking Typecases

5.4.1 Motivation

Java lacks a "typecase" statement or expression. Instead, the programmer must use a combination of `instanceof` and downcasts to first test whether an object belongs to a certain class, and then downcast the object reference if it belongs to the class. Figure 5-6 shows an example; similar patterns occur frequently in many programs. The `instanceof` guard ensures that the downcast is completely safe.

I have extended Ajax RTA to prove that these downcasts are safe. The resulting analysis is called "RTA++".

```
class C {
    Object fieldA;
    Object fieldB;
    public boolean equals(Object x) {
        if (x instanceof C) {
            C c = (C)x;
            return c.fieldA.equals(fieldA)
                && c.fieldB.equals(fieldB);
        } else {
            return false;
        }
    }
}
```

Figure 5-6. A Java program using `instanceof` and `checkcast`

5.4.2 Refining the Bytecode Type Assignment

The idea is to improve the accuracy of the procedure of Section 5.2.2, which assigns static Java types to expressions. In Figure 5-6, the occurrence of `x` inside the `if` body will be assigned the Java type `C`. The analysis then concludes that `x` can only be aliased to instances of C or its subclasses; with this information, the Ajax downcast checking tool proves that the downcast is safe.

The improved static type assignment requires some simple intraprocedural data flow analysis. First, Ajax RTA computes “must alias”information for all local variables and stack elements, using value numbering. For each boolean variable or stack element, Ajax also determines whether the value corresponds to the result of an `instanceof` operation, and if so, which variable and class were tested.

The basic algorithm for computing static Java types for value-points uses standard forward data flow analysis. For each instruction, there is a “transfer function” describing how the types of variables and stack elements at the successor instruction(s) depend on the types of the variables and stack elements at the current instruction. In the RTA++ algorithm, the transfer function corresponding to a conditional branch checks to see whether the branch condition is the result of an `instanceof`. If so, then in the “branch taken” case all known aliases to the tested variable are known to be instances of the tested class. This fact is used to narrow the types assigned to the aliased variables at the successor instruction.

Similar techniques have been used by JIT compilers [18] to reduce the overhead of `instanceof`/`checkcast` pairs.

This technique could also improve the accuracy of other tools using Ajax RTA, but in practice the effect is only noticeable for the downcast checking tool.

6 The SEMI Analysis

6.1 Introduction

6.1.1 Chapter Overview

Previous work [54] investigated using Hindley-Milner style polymorphic type inference to extract a VPR-like relation from C programs. This thesis extends that work by introducing an analysis with new features, including support for Java bytecode programs. This analysis is called SEMI (short for “semiunification”). SEMI combines the following features:

- A flexible and robust framework based on *type inference with polymorphic recursion*.
- A number of modes and optimizations allowing varying tradeoffs between time, space and accuracy.
- A formal model in terms of the Micro Java Bytecode language and the value-point relation.
- A proof of soundness in terms of the model.
- An implementation within the Ajax framework which allows SEMI to be used with a variety of tools, and in combination with other analyses such as RTA. (However, SEMI is completely independent of the other analyses.)

Standard analyses based on type inference are based on constraints. They define a language of terms, including variables standing for terms, and a language of constraints holding between terms. Syntax driven rules specify the construction of an initial constraint set for any given program. The constraints are solved to find canonical or minimal solutions, i.e., assignments of terms to variables. The inference system is constructed so that the solutions represent certain invariants of the program.

SEMI follows a similar pattern. However, to simplify the presentation, SEMI does not use terms; term structures are encoded using “component constraints”, and information about term constructors is omitted. In SEMI, constraints hold only between atomic variables. A SEMI variable can be thought of as the inferred type of a program variable. More discussion of this presentation is given below in Section 6.2.1.2.

Although SEMI is inspired by type inference, and it is useful to apply intuitions about type inference to help understand SEMI, SEMI is not in fact a type inference algorithm. Formally, it is nothing more than a system for computing an approximation to the value-point relation. Nevertheless, in this chapter I use the word “type” to refer to information computed by SEMI. Java types are largely irrelevant to SEMI, and my use of the word “type” never refers to Java types unless explicitly noted.

This chapter gives a formal specification for SEMI, as applied to the Micro Java Bytecode language, and a proof that any algorithm satisfying the specification computes a conservative approximation to the VPR. The details of the implementation are deferred to the next chapter.

6.1.2 Approach

I have chosen to present a direct proof of soundness in terms of MJBC, rather than translating to and from a more traditional lambda language and doing the proof in a conventional setting. Consequently, the proof is rather long and the style may be unfamiliar. However, a proof in a conventional setting would also be rather difficult, because even after translation the system would contain the following features:

- Higher-order functions
- Polymorphic functions
- Unrestricted recursion (declarations not block-structured)
- Records
- Row-polymorphism (record types polymorphic over a set of “unknown” additional fields)
- Polymorphic recursion
- Mutable references
- Exceptions
- Soft typing

Specifying and proving the correctness of the analysis directly in terms of MJBC also keeps the formal presentation closer to the actual implementation.

6.1.3 Implications

This chapter does not merely confirm facts already believed. *It also reveals that the analysis places no static constraints on the program whatsoever*. Even though the implementation assumes that the Java program passes bytecode verification and is therefore statically well-typed according to the Java language rules, the system presented here does not. In other words, SEMI could be implemented without making any assumptions about the target program.

This is useful in practice, because it means that variations in the static verification policies of different virtual machines have no impact on SEMI. It is also useful because it means that Ajax could be applied to ill-formed programs, such as programs undergoing modifications — provided those programs can be translated into bytecode.

Note that according to the semantics of MJBC, the execution of a program which would not be statically well-typed according to Java may reach a state in which no normal transition is possible. For example, a program may attempt to fetch a field when the top of the working stack does not contain an object reference. However, according to the semantics, a spontaneous exception throw is always possible. This implies that a program will never “get stuck”; when no normal transition is possible, it will simply throw a spontaneous exception. Of course, if the exception is not caught, the method call stack will unwind and the program will eventually halt due to the uncaught exception.

This is realistic, as many VMs can report type errors during execution, when code is dynamically and lazily linked. SEMI can account for such behavior.

6.1.4 Relationship to the Implementation

The constraints and rules described here are almost the same as those implemented in SEMI, for the subset of Java bytecode corresponding to MJBC.

One small but significant departure of this formalism from the implementation is the treatment of one constraint for the `new` instruction. (See footnote “a” below, on page 112.) I believe that the implemented constraint is correct, but it would require significant additional work to extend the proof system to accommodate it.

SEMI’s implementation incorporates a number of optimizations that mean some of the constraints here never arise. For example, exceptions and the globals object are “globalized” (see Section 7.6), and no instance constraints are ever applied to them. When only one instance of a particular variable is possible, SEMI replaces the instance constraint with an equality, which gives the same results and saves time and space. (Intuitively, if there is only one instance of a polymorphic value, it may as well not be polymorphic.) These optimizations are applied in the constraint generation phase, so the constraint generation code does not correspond closely to the description here. For details, see Chapter 7.

6.1.5 Chapter Organization

Section 6.2 describes the sets of constraints used by SEMI, and defines a “closed form” for these sets that represents a solution to the constraints. All discussion of how to produce such a closed form is deferred to Chapter 7. Section 6.3 presents an informal overview of how SEMI treats Java programs, by translating Java bytecode examples into a functional language whose standard typing rules would induce similar constraints to SEMI’s. Section 6.4 defines the initial constraint set for an MJBC program and presents a complete example of a program and its analysis using constraints. In Section 6.5 the relationship between the VPR and constraint sets is formally defined. The definition requires some auxiliary judgements, which are defined and some properties of which are proved. The implementation of the Ajax tool interface using SEMI is discussed in Section 6.6.

The remainder of the chapter is Section 6.7, which proves that any closed constraint set gives rise to a sound VPR approximation. This is similar to a proof of soundness of a type system, but rather different in flavor due to the non-traditional setting. This section, and part of Section 6.5, contain a great deal of rather dense mathematics. The casual reader should focus on the statements of lemmas and theorems, which describe the invariants of SEMI that make it sound.

6.2 Constraint System

6.2.1 Constraints

6.2.1.1 Constraint Structures

The SEMI solver uses the following structures:

- V — the set of variables
  These can be thought of as type variables. Each program variable (or in general, each bytecode expression) has a SEMI variable associated with it.
- L — the set of component labels (e.g., `param`, `result`, `fieldA`)
  SEMI treats these as abstract entities and assigns no meaning to them. They are used in component constraints.
- I — the set of instance labels
  Each instance label represents a program site at which a polymorphic value is being used. SEMI treats them as abstract entities and assigns no meaning to them. They are used in instance constraints.
- C — a set of constraints of the following kinds:
  - "$u \cong v$" — an equality constraint expressing the fact that the two variables u and v are to be considered identical. In the presence of such a constraint, two bytecode expressions which are mapped to constraint variables u and v respectively will be considered related in the value-point relation.
  - "$u \triangleright_c v$" — a component constraint expressing the fact that variable u's component with label c is variable v. These constraints can be thought of as encoding the structure of terms. They are used to relate types of object references to the types of their fields, and also the types of methods to the types of their parameters and results.
  - "$u \preccurlyeq_i v$" — an instance constraint expressing the fact that variable u's instance i is variable v. Intuitively, v can be thought of as the i'th copy of u. In the presence of such a constraint, two bytecode expressions mapping to variables u and v respectively will be considered related in the value-point relation.

If the constraint $u \preccurlyeq_i v$ is present in a set, then I write "v is an instance of u" and "u is a source of v". The set should be clear from context. If "$u \triangleright_c v$" is in a set, then I write "v is a component of u" and "u is a parent of v".

The rules that assign an initial constraint set to a program are given in Section 6.4.

6.2.1.2 Relationship to Terms

To illustrate the relationship between standard polymorphic recursion [42] and this setting, consider the following code, expressed in a typed lambda calculus. This is a function to swap the two elements of a pair.

λx. (snd(x), fst(x))

where “fst” and “snd” are the standard projection operations on pairs. While performing type inference with polymorphic recursion, the following constraint arises for the type of “snd” itself, when we consider the invocation of the operator “snd”:

$$(t_0, t_1) \to t_1 \ \lessdot_{Z_1} \ u_1 \to u_2$$

This represents the fact that the type of “snd”, which is known to be $(t_0, t_1) \to t_1$ (where t_0 and t_1 are type variables standing for arbitrary types), is instantiated at program point Z_1 to some currently unknown function type $u_1 \to u_2$ (where u_1 and u_2 are also type variables standing for arbitrary types). (Z_1 would be the program point of the call to the “snd” function.) In other words, the type $u_1 \to u_2$ is constrainted to be a polymorphic instance of $(t_0, t_1) \to t_1$.

This constraint on terms could be translated into the following set of SEMI constraints:

$$\{T_{snd} \rhd_{param} T_{snd\text{-}p}, T_{snd\text{-}p} \rhd_{tuple\text{-}0} t_0, T_{snd\text{-}p} \rhd_{tuple\text{-}1} t_1, T_{snd} \rhd_{result} t_1, T_{snd} \lessdot_{Z_1} v, v \rhd_{param} u_1, v \rhd_{result} u_2\}$$

Note that the terms have been decomposed into variables related by component constraints. This has required the introduction of new variables T_{snd}, $T_{snd\text{-}p}$, and v to represent the compound terms and subterms $(t_0, t_1) \to t_1$, (t_0, t_1) and $u_1 \to u_2$ respectively. The term constructors have disappeared entirely. This is why SEMI is not suitable as a type inference system; it can never detect conflicts between type constructors. In a situation where term unification would fail due to constructor mismatch, SEMI assigns different kinds of components to the same variable. For example, it might infer that a variable has both “tuple-*n*” and “param” components, as if the variable were both a tuple and a function. This is in fact an advantage for SEMI; it will never reject a program as unsuitable for analysis. (In other words, SEMI is a “soft typing” system [85].)

The advantage of the SEMI representation is that it is very simple, yet carries all the information required to perform the analysis. Its particular advantage is in representing recursive structures, which are very common in this kind of analysis; standard term representations need to be extended with recursive constructs such as “μt.T”, where “t” occurs free in T, meaning the solution to the fixpoint equation “t = T(t)”.

6.2.2 Solutions

A solution to a constraint set C is another constraint set C' such that $C \subseteq C'$ and C' is *closed*. A closed constraint set can be thought of as a set in which all implicit relationships implied by the constraints are stated explicitly. A VPR approximation can be efficiently computed from such a set. C is closed if it satisfies the conjunction of the following conditions: (*t*, *u*, *v* and *w* range over constraint variables)

- Equality closure: equality constraints in a closed set possess the usual properties of symmetry, transitivity and substitutional equivalence.
  $\forall t, u.\ \{t \cong u\} \subseteq C \Rightarrow \{u \cong t\} \subseteq C$
  $\forall t, u, v.\ \{t \cong u, u \cong v\} \subseteq C \Rightarrow \{t \cong v\} \subseteq C$
  $\forall t, u, v, c.\ \{t \cong u, t \rhd_c v\} \subseteq C \Rightarrow \{u \rhd_c v\} \subseteq C$
  $\forall t, u, v, c.\ \{t \cong u, v \rhd_c t\} \subseteq C \Rightarrow \{v \rhd_c u\} \subseteq C$
  $\forall t, u, v, i.\ \{t \cong u, t \lessdot_i v\} \subseteq C \Rightarrow \{u \lessdot_i v\} \subseteq C$

$\forall t, u, v, i.\ \{t \cong u, v \preccurlyeq_i t\} \subseteq C \Rightarrow \{v \preccurlyeq_i u\} \subseteq C$
Equality is meant to be reflexive, but it is troublesome to require reflexivity constraints as explicit elements of the constraint set. The obvious rule $\forall u.\ \{u \cong u\} \subseteq C$ is undesirable because it requires C to contain an infinite number of constraints. A more complex definition is possible, but in fact there is no need for explicit reflexivity constraints, so they are not required to be in the set.

- Component uniqueness: a variable has at most one distinct component with a given label.
  $\forall t, u, v, c.\ \{t \rhd_c u, t \rhd_c v\} \subseteq C \Rightarrow \{u \cong v\} \subseteq C$
- Instance uniqueness: a variable has at most one distinct instance with a given label.
  $\forall t, u, v, i.\ \{t \preccurlyeq_i u, t \preccurlyeq_i v\} \subseteq C \Rightarrow \{u \cong v\} \subseteq C$
- Component propagation: if a variable has a component v, then its instances also have the component.
  $\forall t, u, v, c, i.\ \{t \rhd_c u, t \preccurlyeq_i v\} \subseteq C \Rightarrow \exists w.\ \{v \rhd_c w\} \subseteq C$
- Instance propagation: instance relationships propagate to matching components.
  $\forall t, u, v, w, c, i.\ \{t \rhd_c u, t \preccurlyeq_i v, v \rhd_c w\} \subseteq C \Rightarrow \{u \preccurlyeq_i w\} \subseteq C$

Given any finite set of constraints C, there is always a finite solution set C' such that $C \subseteq C'$ and C' is closed. For example, the set C' could be C with equality constraints added between all variables mentioned in C, and all instance and component relationships holding between all the variables. This would be a correct solution, but not a very useful one because the induced value-point relation would relate every pair of bytecode expressions.

A more realistic strategy is to interpret the closure rules as production rules. At each step, if the set of constraints is not closed, the algorithm selects a rule whose hypothesis is satisfied but whose consequent is not and adds the constraint required to satisfy the consequent. Unfortunately, this algorithm does not terminate for practical examples.

Discussion of the actual SEMI algorithm is deferred to Chapter 7. In this chapter, I treat it as a black box and show that given an appropriate set of initial constraints, any closed solution gives rise to a conservative approximation of the value-point relation.

6.2.3 Remarks

Simplifications of the closure rules give rise to a number of previously studied analyses. For example, if one takes only the equality closure rules plus two rules below forcing components and instances to be degenerate, one obtains a simple monomorphic, structureless type inference analysis similar to Steensgard's [72]:

$$\forall t, u, v, c.\ \{t \rhd_c u\} \subseteq C \Rightarrow \{u \cong t\} \subseteq C$$
$$\forall t, u, v, i.\ \{t \preccurlyeq_i u\} \subseteq C \Rightarrow \{u \cong t\} \subseteq C$$

If one takes only the equality rules and the component uniqueness rule, and forces instances to be degenerate, then one obtains a monomorphic type inference analysis with structures. This system essentially performs simple term unification. Cycles in the graph of component constraints are allowed, and correspond to recursive type terms.

With the full treatment of polymorphic instance constraints as described, the system corresponds to type inference with polymorphic recursion using semiunification, again with recursive terms allowed. (The term “polymorphic recursion” means that cycles in the graph of instance constraints are allowed, such as when a polymorphic function recursively calls itself and passes in one of its original parameters.)

In general it is not possible to compute a “most general” or “principal” closed constraint set. This is discussed further in Section 7.1.2.

6.3 The Encoding

6.3.1 Introduction

SEMI generates a set of initial constraints directly from a bytecode program and then solves them to find a closed form. However, the procedure can be viewed conceptually as a translation from the bytecode language into an extended lambda calculus, followed by generation of type constraints for the translated code, followed by solution of the type constraints to yield inferred types. Here I provide an informal description of SEMI from the latter point of view.

6.3.2 Methods

Each Java bytecode method declaration is translated to a function declaration. Each function can take multiple parameters directly — no currying is used. The implicit “this” parameter of non-static methods becomes an explicit parameter in the translation. Functions return two values: the value returned normally by the method, and the thrown exception, if any. Methods that return nothing (“void”) have a return value in the translation, but the value is always ignored. (In the formal MJBC semantics, every function returns a value, so this issue does not arise.)

Therefore this method that adds 3 to `x`

```
int add3(int x) { load x; bipush 3; iadd; ireturn; }
```

translates to the equivalent of

```
fun add3(this, x) = (x + 3, …)
```

The “…” indicates that there is no value for the exception; its type is unconstrained. This means that, after type inference, the type of the exception will be a unique type variable. SEMI will conclude that the exception is not related in the VPR to any other value, as one would hope, since there is in fact no exception. (Obviously “…” precludes the translated code from being executable, but that is not a problem.) (A sum type could be used instead of a pair, to indicate that only one of the alternatives is possible, but this leads to essentially the same type constraints.)

Methods are assigned function types. The above method would be assigned the following “type”:

`add3`: $\forall a, b, e.\ \ (a) \rightarrow (b, e)$

The intuition behind the interpretation of these types is that if two variables can be inferred to have different types, then they cannot be aliased in the VPR sense. If they are always inferred to have the same type, then they may be aliased.

Even though the `x` parameter's real type is `int`, we assign it a type variable so that we can compare its type meaningfully with the types of other variables which also hold integers. For example, here we can see that the value returned by `add3` is a new integer, different from the parameter. (We can also see that the parameter and result are both different from whatever exception may be thrown by `add3`.)

In SEMI, these inferred types become atomic constraint variables connected by component constraints as discussed in Section 6.2.1.2. For example, the above type would be represented as

`add3`: T, where the constraint set contains
$\{ \mathrm{T} \triangleright_{\text{param-0}} a, \mathrm{T} \triangleright_{\text{result}} b, \mathrm{T} \triangleright_{\text{exn}} e \}$

6.3.3 Global Variables

Global variables (Java "static fields") are passed into all functions in an extra record parameter. Each slot of the record corresponds to one global variable. For example, the method

```
int getGlobal() {
  getstatic globalVar; ireturn;
}
```

translates to the equivalent of

```
fun getGlobal(globals) =
  (globals.globalVar, …)
```

The function simply performs the assignment and then returns no result and no exception. The following type signature would be inferred for this function:

`getGlobal`: $\forall a, e, \rho.\ \ (\{\ \text{globalVar}: a; \rho\ \}) \rightarrow (a, e)$

This signature requires `globals` to have a field `globalVar` of type a, which must be the same type as the result. The polymorphic type variable ρ, sometimes referred to as a "row variable", represents the types of an unknown set of other fields of `globals` (i.e., other global variables). This signature allows the other global variables to have any type.

This treatment of globals means that all function bodies are *closed*, i.e., refer only to variables defined locally or available as parameters, or to other functions. Therefore, in the type inferred for each function, every type variable can be polymorphically generalized. (In the language of Hindley-Milner type inference, every type variable is free in the enclosing type environment.)

If global variables were instead declared as variables in the enclosing environment, e.g.,

```
let globalVar = ref 0 in
  fun getGlobal() = (globalVar, …)
```

then the type signatures would be

```
  globalVar:           a
  getGlobal:   ∀e.     () → (a, e)
```

The expression `ref 0` indicates that `globalVar` is mutable and therefore its type cannot be polymorphically generalized; usage of `globalVar` in different contexts may refer to the same runtime value, and therefore `globalVar` must have the same type a in all contexts. Similarly, in the type inferred for `getGlobal`, a cannot be polymorphically generalized because it is constrained to the type of `globalVar`.

The two strategies actually produce the same analysis results, because even when each function takes the global variable record as a polymorphic parameter, there is really only one global variable record in the program and one "canonical" type for this record (its type in the program's `main` function). This "top level" type is a polymorphic instance of every other type for the global variable record. Lemma 6-21 below and Section 7.6 explain this in more detail.

For simplicity, SEMI uses explicit global variable passing, so that every type variable in a function signature is polymorphically generalized. The implementation performs optimizations for types (such as the types of global variables) that have only one meaningful instance; this is discussed in Section 7.6. In the rest of this section the global variable passing is ignored for the sake of brevity.

The "row variables" do not occur in SEMI's constraints. They are implicit. For example, the above method would be given the following constraints:

`getGlobal:` T
where the constraint set contains
$\{ \mathrm{T} \triangleright_{\mathrm{globals}} \mathrm{T}_{\mathrm{globals}}, \mathrm{T}_{\mathrm{globals}} \triangleright_{\mathrm{globalVar}} a, \mathrm{T} \triangleright_{\mathrm{result}} a, \mathrm{T} \triangleright_{\mathrm{exn}} e \}$

6.3.4 Object Encoding

Java objects are treated as extensible records, each similar to the "global variables" record. Each slot of the record contains either a field or a method. For example, the code

```
int getX() {
  load this; getfield fieldX; return;
}
```

would translate to (ignoring the globals object for now)

```
fun getX(this) =
  (this.fieldX, …)
```

This would get type signature

`getX:` $\forall a, d, \rho.\ (\{ \text{fieldX}: a; \rho \}) \to (a, d)$

Here `this` is deconstructed into a record containing field `fieldX` of type a and some set of other fields of types ρ. Effectively, this function and its type say nothing about what other fields of `this` there may be. Any object containing a `fieldX` can be passed in. In fact, any object at all can be passed in, and the type inference algorithm will infer that it contains `fieldX`. This "row polymorphism" avoids any need for subtype polymorphism in this type system. (This complete reliance on row polymorphism distinguishes this type system from the type system of O'Caml [65], where row polymorphism is available but explicit classes and subtyping are usually used instead.) It also helps reduce the sizes of types inferred for functions, because only fields actually used by the function are given types in the function's signature.

Field names are always fully qualified with the name of the class in which they are declared, so two fields of different classes which happen to have the same name are never confused in the translation.

The Java class of an object is never represented in the translation or in the type inference system. The implications of this are discussed in the following sections. Tools based on SEMI can recover class information using the VPR; this is discussed in Chapter 10 and elsewhere.

6.3.5 Method Encoding

6.3.5.1 Static Methods

Static methods are treated as normal functions. A call to a static method is translated into a direct call to the appropriate function. For example, the code in Figure 6-1 would be translated to the equivalent of the code in Figure 6-2.

```
static int addOne(int x) {
  load x; bipush 1; iadd; ireturn;
}
static int addOneWrapper(int y) {
  load y; invokestatic addOne; ireturn;
}
```

Figure 6-1. Static Method Example

```
fun addOne(x) =
  (x + 1, …)
fun addOneWrapper(y) =
  (addOne(y), …)
```

Figure 6-2. Static Method Translation

Because the function `addOne` is a polymorphic value, its use in `addOneWrapper` is assigned a fresh polymorphic instance of the type of `addOne`. All calls to static methods are treated polymorphically. (In other words, static method calls are analyzed with calling-context sensitivity.) Intuitively, this is safe because (being closed) distinct calls to `addOne` are completely independent and cannot communicate except through the caller's environment.

6.3.5.2 Nonstatic Methods

Nonstatic methods — that is, methods involved in dynamic dispatch — are encoded by treating them as functions assigned to the slots of objects when those objects are created. For example, the code in Figure 6-3 would be translated to the equivalent of the code in Figure 6-4.

```
class MyObj {
  int fieldX;
  int MyObj_getX(MyObj this) {
    load this; getfield MyObj_fieldX; ireturn;
  }
}
static int getter(Object o) {
  load o; invokevirtual getX; ireturn;
}
static int main() {
  new MyObj; invokestatic getter; ireturn;
}
```

Figure 6-3. Nonstatic Method Example

```
fun MyObj_getX(this) =
  (this.MyObj_fieldX, …)
fun getter(o) = (o.getX)(o)
fun main() =
  let obj = { getX: MyObj_getX; MyObj_fieldX: 0; }
  in getter(obj)
```

Figure 6-4. Nonstatic Method Translation

The following types are inferred:

`MyObj_getX`:	$\forall a, e, \rho.$ ({ MyObj_fieldX: a; ρ }) $\rightarrow$ (a, e)
`getter`:	$\forall b, e, \rho.$ (t) $\rightarrow$ (b, e) where t = { getX: (t) $\rightarrow$ (b, e); ρ }
`obj` (in `main`):	u where u = { getX: (u) $\rightarrow$ (c, e); MyObj_fieldX: c; ρ } (for some c, e, ρ)

Note that objects containing methods usually have recursive types, because the type of the `this` parameter in each method type is usually the same as the object type.

Another example of the treatment of virtual method calls, expressed directly in the constraint language of SEMI, is given below in Section 6.4.7.

6.3.5.3 Type Checking/Inference For Nonstatic Methods

Given the above types and assuming standard type checking rules, it is straightforward to show that the types are consistent with the code and each other.

For example, to typecheck `getter`, we observe that the type of `o` is t, and therefore the type of `o.getX` is (t) $\rightarrow$ (b, e). In the call to `o.getX`, we indeed pass in a parameter of type t (`o`). Furthermore, the result returned from `getX` has type (b, e), which correctly matches the return type of `getter`.

Note that `getter` is typechecked (and can have its type inferred) independently of any information about the callee in the call to `getX` (`MyObj_getX`). All that is required is that the type of the `getX` method recorded in the type of `getter`'s `o` parameter is consistent with the actual usage of that method within `getter`. The type information recorded for `getX` in the type signature of `getter` effectively describes how the method is used by `getter`.

To check the type of `obj` in `main`, observe that it constrained both by the initialization of `obj` as a new `MyObj` object and by `obj` being passed as a parameter to `getter`. The initialization of `obj` requires `obj`'s type u to be the type of an object containing a `getX` method and a `MyObj_fieldX` field. Furthermore, the type of the `getX` method within u must be a polymorphic instance of the type of `MyObj_getX` (which is "$\forall a, e, \rho.$ ({ MyObj_fieldX: a; ρ }) $\rightarrow$ (a, e)"). If no method call was made on the object, we could therefore just set u = { getX: ({ MyObj_fieldX: c, ρ }) $\rightarrow$ (c, e); MyObj_fieldX: d; ρ' } (for some c, d, e, ρ, ρ').

However, the type of `obj` is also constrained by the call to `getter(obj)`. This call requires u to be some polymorphic instance of `getter`'s parameter type t, where t = { getX: (t) $\rightarrow$ (b, e); ρ }. Because the parameter type of t's `getX` method is t itself, the parameter type of u's `getX` method is also required to be u itself. Unifying this constraint with the constraints mentioned above requires u to be of the form { getX: (u) $\rightarrow$ (c, e); MyObj_fieldX: c; ρ }.

Note also that the type signature of `getter` promises that its result has the same type (b) as the result of its object parameter's `getX` method. Therefore in `main` we learn that the result of the call to `getter` will have type c.

6.3.5.4 Treatment Of Polymorphism

The call to `getter` in `main` is treated polymorphically; the caller's parameter and result types are required to be some polymorphic instance of the callee's types. On the other hand the call to `getX` from `getter` is not treated polymorphically; the caller and callee types must be identical.

The technical reason for this distinction is that we can only polymorphically generalize type variables that are not bound in the current type environment. All the type variables in the type assigned to `getter` are polymorphically generalized, because they do not occur anywhere outside the definition of `getter`. (Intuitively, this means that the assignment of types to these variables is independent of anything outside `getter`, and therefore different types can be chosen for each use of `getter`.) On the other hand, in `getter`, the type variables in the type of the callee `o.getX` are bound in the type environment; in particular they occur inside `getter`'s parameter type. (Intuitively, this means that the assignment of types to these type variables is constrained by the caller of `getter`. For example, the caller of `getter` might pass in an object whose `getX` method always returns an integer. Obviously it would be unsafe to allow `getter` to choose different return types for each call to `getX`.)

6.3.5.5 Polymorphism In Object Creation

When an object is created, such as when `obj` is created in `main`, its field and method slots are always iniitalized with constant values — either zero scalar values, or the functions that implement the methods supported by the object. The usage of these constant values is always treated polymorphically. Therefore if a method implementation is inherited into multiple classes, which are instantiated at multiple sites, the references to the method implementation at each site can be given distinct types. Similarly, fields of objects of the same class created at different sites can be given distinct types.

6.3.6 Extensible Records and Object Classes

Consider the code in Figure 6-5. This example demonstrates the use of subclass polymorphism with subclasses having distinct fields.

```
class SuperObj {
  abstract int getX(SuperObj this);
}
class MyObj {
  int fieldX;
  int getX(MyObj this) {
    load this; getfield MyObj_fieldX; ireturn;
  }
}
class YourObj {
  int otherX;
  int getX(YourObj this) {
    load this; getfield YourObj_otherX; ireturn;
  }
}
static int getter(SuperObj obj) {
  load obj; invokevirtual getX; ireturn;
}
static int main() {
  if … then new MyObj else new YourObj;
  invokevirtual getX; ireturn;
}
```

Figure 6-5. Extensible Record Example

The following types are inferred:

`MyObj_getX`:	$\forall a, e, \rho.\ (\{\ \text{MyObj\_fieldX}: a; \rho\ \}) \to (a, e)$
`YourObj_getX`:	$\forall b, e, \rho.\ (\{\ \text{YourObj\_otherX}: b; \rho\ \}) \to (b, e)$
`getter`:	$\forall c, e, \rho.\ (t) \to (c, e)$ where $t = \{\ \text{getX}: (t) \to (c, e); \rho\ \}$
object in `main`:	u where $u = \{\ \text{getX}: (u) \to (c, e); \text{MyObj\_fieldX}: c;$ $\text{YourObj\_otherX}: c; \rho\ \}$ (for some c, e, ρ)

In general, if Java declares a variable to be of class C (here, `SuperObj`), then any fields and methods belonging to C or any subclass of C (here, `MyObj` and `YourObj`) can appear in the type inferred for the variable. This can lead to the slightly counterintuitive situation where variables having the least constraining Java types (e.g., variables of type `Object`) have the most complex inferred types.

6.3.7 Mutability

Global variables and fields of objects are mutable. However, in the type system I have not distinguished mutable and immutable slots of records. The distinction is irrelevant because whenever a slot of a record is accessed, the record has a monomorphic type and therefore the type of the slot is monomorphic. Thus two accesses to the same slot of a record, whether reads or writes, always get the same type for the slot. (The fatal error would be to treat a mutable slot of a record as polymorphic; we might store a value in the slot with one type, retrieve the value with another type, and thus destroy soundness.)

6.3.8 Control Flow

Internally, a Java bytecode method is simply an array of bytecode instructions with arbitrary control flow between them. SEMI treats each bytecode instruction as a local function which takes the values of the current working stack and local variables as parameters, and calls the successor instruction(s) as tail calls. Each local function returns the final result of the method and its thrown exception.

The stack is passed as a list, so that "push" operations become "cons" and "pop" operations become "head/tail". Local variables are passed in a record.

A method executes by calling the local function for the first instruction, with method parameters placed into local variables (as required by the Java bytecode semantics).

`For example, the method`

```
int add3(int x) { load x; bipush 3; iadd; return; }
```

translates to

```
fun add3(this, x) =
  let fun f_0(st, (v0, v1)) =
    f_1(v0::st, (v0, v1))
  and fun f_1(st, (v0, v1)) = f_2(3::st, (v0, v1))
  and fun f_2(a::b::st, (v0, v1)) = f_3((a+b)::st, (v0, v1))
  and fun f_3(v::st, (v0, v1)) = (v, …)
  in f_0([], {#0: this; #1: x})
```

The encoding is simple and regular.

All kinds of control flow are easily handled. The method

```
static int isequal(int x, int y) {
  0: load 0; 1: load 1; 2: if_cmpeq 6;
  3: bipush 0; 4: store 2; 5: goto 8;
  6: bipush 1; 7: store 2;
  8: load 2; 9: return; }
```

translates to

```
fun isequal(x, y) =
  let fun f_0(st, (v0, v1, v2)) = f_1(v0::st, (v0, v1, v2))
  and fun f_1(st, (v0, v1, v2)) = f_2(v1::st, (v0, v1, v2))
  and fun f_2(v1::v0::st, ls) =
    if v1 = v0 then f_6(st, ls) else f_3(st, ls)
  and fun f_3(st, ls) = f_4(0::st, ls)
  and fun f_4(a::st, (v0, v1, v2)) = f_5(st, (v0, v1, a))
  and fun f_5(st, ls) = f_8(st, ls)
  and fun f_6(st, ls) = f_7(1::st, ls)
  and fun f_7(b::st, (v0, v1, v2)) = f_8(st, (v0, v1, b))
  and fun f_8(st, (v0, v1, v2)) = f_9(v2::st, (v0, v1, v2))
  and fun f_9(v::st) = (v, …)
```

These calls between instructions could be treated polymorphically. In theory some accuracy might be gained because at control flow merge points, the state along each incoming control flow edge could be given a different type, each an instance of the type of the state at the destination instruction. In practice this increased accuracy has not proved useful, and even with some obvious optimizations (e.g., only allow polymorphism for calls to instructions representing control flow merge points), it has proved prohibitively expensive. Therefore in practice SEMI treats these transfers monomorphically (making the types of the actual parameters and results equal to the types of the formal parameters and results, rather than instances of those types). However, in the description below, I use polymorphic constraints for instruction transfers to show that they are sound.

However, even under monomorphism it is still the case that a stack location or local variable can be given different types at different program points. For example, local variable #2 is has a different type after it is assigned to the type it had before assignment. This has the same effect as translating the program into Single Static Assignment form before performing the analysis, but it arises naturally from the encoding.

6.3.9 Exception Handling

Exception handling is performed in a way similar to other control transfers. In each method, every instruction which might throw an exception, or receive a propagated exception (which is actually all instructions, because the virtual machine can throw an “internal error” exception at any instruction), can transfer control to any applicable exception handlers defined in the method. The translation does not specify when an exception is thrown; for a given instruction, the choice of whether to throw an exception or continue normal execution is always considered to be nondeterministic (unless the instruction is an unconditional `athrow` instruction). Control transfer to an exception handler puts the current exception object onto the top of the working stack, as specified by the Java bytecode semantics.

Most methods do not have any explicit exception handlers. However, all methods must be able to propagate thrown exceptions to the caller. Each instruction which can throw an exception (or receive a propagated exception) can nondeterministically choose to return the exception value immediately as the method result, thus propagating the exception. The following code shows an example of such behavior:

```
fun callAll() =
  let (result1, exn1) = call1()
  in if ? then (…, exn1) else
    let (result2, exn2) = call2(result1)
    in if ? then (…, exn2) else
      (result2, …)
```

6.4 Initial Constraint Set

Consider a program P in the Micro Java Bytecode language, as defined in Section 3.2.2.

6.4.1 Constraint Variables

The set of initial constraints for P makes use of the following variables:

- S_{pc}: the variable for the working stack on entry to instruction *pc*
  The stack is a list, so its variable can have two components: "head", representing the top of the stack, and "tail", representing the rest of the stack.
- L_{pc}: the variable for the local variable file on entry to instruction *pc*
  The local variables are indexed by number, so L_{pc} has numbered components, one for each local variable used.
- X_{pc}: the variable for the exception thrown by the code starting at *pc*
- G_{pc}: the variable for the global variables on entry to instruction *pc*
  This variable has one component for each static field in the program.
- R_{pc}: the variable for the value that the code at *pc* eventually returns from the method
- S'_{pc}, L'_{pc}: the variables for the state on leaving instruction *pc*
- $N_{classID}$: the variable representing the prototypical object of class *classID*
- $M_{methodImpl}$: the variable representing the type of the method *methodImpl*
- $T_{pc,label}$: variables used by the instruction at *pc* for internal purposes
- $N_{classID,methodID}$: the variable representing the type of inherited method *methodID* in class *classID*
- $N_{classID,fieldID}$: the variable representing the type of field *fieldID* in class *classID*
- $N_{fieldID}$: the variable representing the type of static field *fieldID*
- Err: the variable representing the exceptions which may be thrown spontaneously by the virtual machine
- $S'_{\text{exn-}pc\text{-}classID}$: these variables represent the new stack on transfer to an exception handler when exception *classID* is thrown at *pc*

6.4.2 Instance Labels

SEMI uses the following instance labels:

- *pc*-*pc′*: an instance representing the use of (transfer of control to) one instruction from another.
  SEMI treats each instruction as a function; transferring control from one instruction to another corresponds a call to the destination instruction's function, passing in the current local variables, working stack elements and global variables as parameters. These "functions" do not return until the entire method returns; the returned value is the result of the method. The functions are treated as polymorphic, so different information can be inferred for an instruction for each incoming control path.
- *pc*: an instance representing the use of a static method (when *pc* corresponds to an `invokestatic` instruction) or the creation of a new object (when *pc* corresponds to a `new` instruction).
  A method can be thought of as a polymorphic function. Note that global variables are treated as the fields of a "globals object" which is passed as a parameter to every such function, so every such function is self-contained and has no references to any environment. A static call to a method is a direct invocation of the function, and so gets a new polymorphic instance. Creation of an object can be thought of as cloning a prototype object, and also gets a new polymorphic instance.
- *classID*-*methodID*: an instance representing the inheritance of a method implementation by a class.
  Each prototype object for a class can be thought of as a record, with one slot for each signature of the methods implemented by the class. The putative definition of the prototype assigns the function associated with each inherited method implementation to the slot for its signature. Since one method implementation can be inherited into multiple classes, each class which uses a method implementation gets a new polymorphic instance of the method.
- err-*pc*: an instance representing the creation of a spontaneously thrown exception at a particular program point.
  This is similar to the instance induced when an object is created by `new`.
- err-*classID*: an instance representing the creation of a new object when a spontaneous exception is thrown.
  A spontaneous exception creates an object which has one of many possible classes. The variable "Err" represents the type of an object which could be any one of these classes, and therefore "Err" is an instance of the object prototype for each spontaneous exception class. Each of these instances needs a different label, err-*classID*.

6.4.3 Component Labels

I make use of the following component labels:

- param-*i*: a parameter to a method.
- globals: the global variables passed into a method.
- result: the result returned by a method.
- exn: the exception thrown by a method (essentially, an alternative result).
- *i*: a local variable index.

- *fieldID*: a field slot of an object.
- *methodID*: a method slot of an object.
- head: the head element of a stack, treated as a list.
- tail: the tail of a stack.

6.4.4 Program Constraints

The set of initial constraints assigned to an MJBC program is given as

InitialConstraints(P) =

$(\cup \{ \text{IConstraints}(pc) \mid pc \in \text{dom Instruction} \})$

$\cup \quad (\cup \{ \text{MInvocation}(methodImpl) \mid (methodImpl, 0) \in \text{dom Instruction} \})$

$\cup \quad (\cup \{ \text{MDispatch}(classID, methodID) \mid (classID, methodID) \in \text{dom Dispatch} \})$

$\cup \quad (\cup \{ \text{IFields}(classID) \mid classID \in \text{dom InitFields} \})$

$\cup \quad (\cup \{ \text{CatchConstraints}(pc, classID) \mid (pc, classID) \in \text{dom CatchBlockOffset} \})$

$\cup \quad (\cup \{ \{ G_{(\text{Main}, 0)} \rhd_{fieldID} N_{fieldID} \} \mid fieldID \in \text{dom InitStaticFields} \})$

$\cup \quad (\cup \{ \{ \text{Err} \lessdot_{\text{err-}pc} X_{pc} \} \mid pc \in \text{dom Instruction} \})$

$\cup \quad (\cup \{ \{ N_{classID} \lessdot_{\text{err-}classID} \text{Err} \} \mid classID \in \text{ErrorClassIDs} \})$

This definition uses several functions:

- *IConstraints*(*pc*) is a partial function that assigns to each *pc* the initial constraints induced by the instruction at *pc*. IConstraints is defined by the rules in Table 6-1.
- *MInvocation* computes the constraints needed to hook up the type of a method body *m* to the types at the method definition.

  $$\text{MInvocation}(m) = \{M_m \rhd_{\text{param-0}} T_{m,\text{p0}}, M_m \rhd_{\text{param-1}} T_{m,\text{p1}}, M_m \rhd_{\text{globals}} G_{(m,0)}\} \cup \{M_m \rhd_{\text{exn}} X_{(m,0)}, M_m \rhd_{\text{result}} R_{(m,0)}, L_{(m,0)} \rhd_0 T_{m,\text{p0}}, L_{(m,0)} \rhd_1 T_{m,\text{p1}}\}$$

- *MDispatch* computes the constraints needed to implant the type of the method implementation *methodID* into the type of the prototype object for class *classID*.

  MDispatch(*classID*, *methodID*) =

  $$\{ M_{\text{Dispatch}(classID,\, methodID)} \lessdot_{classID\text{-}methodID} N_{classID,methodID}, N_{classID} \rhd_{methodID} N_{classID,methodID} \}$$

- *IFields* computes constraints ensuring that every object field has a type.

  IFields(*classID*) =

  $$\{ N_{classID} \rhd_{fieldID} N_{classID,fieldID} \mid fieldID \in \text{dom InitFields}(classID) \}$$

Instruction(pc)	IConstraints(pc)
`aconst_null`	$\{ S'_{pc} \triangleright_{\text{tail}} S_{pc},\ S_{pc+1} \triangleright_{\text{head}} T_{pc,\text{v}} \}$ $\cup\ \text{Succ}(pc, pc+1, S'_{pc}, L_{pc})$
`bipush` *byte*	$\{ S'_{pc} \triangleright_{\text{tail}} S_{pc},\ S_{pc+1} \triangleright_{\text{head}} T_{pc,\text{v}} \}$ $\cup\ \text{Succ}(pc, pc+1, S'_{pc}, L_{pc})$
`iadd`	$\{ S_{pc} \triangleright_{\text{tail}} T_{pc,\text{t1}},\ T_{pc,\text{t1}} \triangleright_{\text{tail}} T_{pc,\text{t2}},\ S'_{pc} \triangleright_{\text{tail}} T_{pc,\text{t2}},$ $S_{pc+1} \triangleright_{\text{head}} T_{pc,\text{v}} \}$ $\cup\ \text{Succ}(pc, pc+1, S'_{pc}, L_{pc})$
`load` *index*	$\{ L_{pc} \triangleright_{index} T_{pc,\text{v}},\ S'_{pc} \triangleright_{\text{tail}} S_{pc},\ S'_{pc} \triangleright_{\text{head}} T_{pc,\text{v}} \}$ $\cup\ \text{Succ}(pc, pc+1, S'_{pc}, L_{pc})$
`store` *index*	$\{ S_{pc} \triangleright_{\text{tail}} S'_{pc},\ S_{pc} \triangleright_{\text{head}} T_{pc,\text{v}},\ L'_{pc} \triangleright_{index} T_{pc,\text{v}} \} \cup$ $\{ L'_{pc} \triangleright_{i} T_{pc,\text{i}} \mid i \in \text{LocalNames}(pc) \wedge i \neq index \} \cup$ $\{ L_{pc} \triangleright_{i} T_{pc,\text{i}} \mid i \in \text{LocalNames}(pc) \wedge i \neq index \} \cup$ $\text{Succ}(pc, pc+1, S'_{pc}, L'_{pc})$
`if_cmpeq` *offset*	$\{ S_{pc} \triangleright_{\text{tail}} S'_{pc} \}$ $\cup\ \text{Succ}(pc, pc+1, S'_{pc}, L_{pc}, G_{pc}, X_{pc}, R_{pc})$ $\cup\ \text{Succ}(pc, (\text{CodeLocMethod}(pc), \mathit{offset}), S'_{pc}, L_{pc})$
`goto` *offset*	$\text{Succ}(pc, (\text{CodeLocMethod}(pc), \mathit{offset}), S_{pc}, L_{pc})$
`return`	$\{ S_{pc} \triangleright_{\text{head}} R_{pc} \}$
`new` *classID*	$\{ S'_{pc} \triangleright_{\text{tail}} S_{pc},\ S_{pc+1} \triangleright_{\text{head}} T_{pc,\text{v}},\ N_{classID} \preccurlyeq_{pc} T_{pc,\text{v}} \}$ $\cup\ \text{Succ}(pc, pc+1, S'_{pc}, L_{pc})$[a]
`getfield` *fieldID*	$\{ S_{pc} \triangleright_{\text{tail}} T_{pc,\text{t}},\ S_{pc} \triangleright_{\text{head}} T_{pc,\text{obj}},\ T_{pc,\text{obj}} \triangleright_{fieldID} T_{pc,\text{v}},$ $S'_{pc} \triangleright_{\text{head}} T_{pc,\text{v}},\ S'_{pc} \triangleright_{\text{tail}} T_{pc,\text{t}} \}$ $\cup\ \text{Succ}(pc, pc+1, S'_{pc}, L_{pc})$
`putfield` *fieldID*	$\{ S_{pc} \triangleright_{\text{tail}} T_{pc,\text{t}},\ S_{pc} \triangleright_{\text{head}} T_{pc,\text{v}},\ T_{pc,\text{t}} \triangleright_{\text{tail}} S'_{pc},$ $T_{pc,\text{t}} \triangleright_{\text{head}} T_{pc,\text{obj}},\ T_{pc,\text{obj}} \triangleright_{fieldID} T_{pc,\text{v}} \}$ $\cup\ \text{Succ}(pc, pc+1, S'_{pc}, L_{pc})$
`getstatic` *fieldID*	$\{ G_{pc} \triangleright_{fieldID} T_{pc,\text{v}},\ S'_{pc} \triangleright_{\text{tail}} S_{pc},\ S'_{pc} \triangleright_{\text{head}} T_{pc,\text{v}} \}$ $\cup\ \text{Succ}(pc, pc+1, S'_{pc}, L_{pc})$
`putstatic` *fieldID*	$\{ S_{pc} \triangleright_{\text{tail}} S'_{pc},\ S_{pc} \triangleright_{\text{head}} T_{pc,\text{v}},\ G_{pc} \triangleright_{fieldID} T_{pc,\text{v}} \}$ $\cup\ \text{Succ}(pc, pc+1, S'_{pc}, L_{pc})$
`invokevirtual` *methodID*	$\{ S_{pc} \triangleright_{\text{tail}} T_{pc,\text{t1}},\ S_{pc} \triangleright_{\text{head}} T_{pc,\text{v1}},\ T_{pc,\text{t1}} \triangleright_{\text{tail}} T_{pc,\text{t2}},$ $T_{pc,\text{t1}} \triangleright_{\text{head}} T_{pc,\text{v0}},\ T_{pc,\text{v0}} \triangleright_{methodID} T_{pc,\text{m}},$ $S'_{pc} \triangleright_{\text{tail}} T_{pc,\text{t2}},\ S'_{pc} \triangleright_{\text{head}} T_{pc,\text{r}} \}$ $\cup\ \text{MethodCall}(T_{pc,\text{m}}, T_{pc,\text{v0}}, T_{pc,\text{v1}}, G_{pc}, X_{pc}, T_{pc,\text{r}})$ $\cup\ \text{Succ}(pc, pc+1, S'_{pc}, L_{pc})$

Table 6-1. Instruction Constraints

Instruction(pc)	IConstraints(pc)
`invokestatic` *methodImpl*	$\{ S_{pc} \triangleright_{\text{tail}} T_{pc,\text{t1}}, S_{pc} \triangleright_{\text{head}} T_{pc,\text{v1}}, T_{pc,\text{t1}} \triangleright_{\text{tail}} T_{pc,\text{t2}}, T_{pc,\text{t1}} \triangleright_{\text{head}} T_{pc,\text{v0}}, M_{methodImpl} \lessdot_{pc} T_{pc,\text{m}}, S'_{pc} \triangleright_{\text{tail}} T_{pc,\text{t2}}, S'_{pc} \triangleright_{\text{head}} T_{pc,\text{r}} \}$ $\cup$ MethodCall($T_{pc,\text{m}}, T_{pc,\text{v0}}, T_{pc,\text{v1}}, G_{pc}, X_{pc}, T_{pc,\text{r}}$) $\cup$ Succ(pc, pc+1, S'_{pc}, L_{pc})
`checkcast` *classID*	Succ(pc, pc+1, S_{pc}, L_{pc})
`instanceof` *classID*	$\{ S_{pc} \triangleright_{\text{tail}} T_{pc,\text{t}}, S'_{pc} \triangleright_{\text{tail}} T_{pc,\text{t}}, S_{pc+1} \triangleright_{\text{head}} T_{pc,\text{v}} \}$ $\cup$ Succ(pc, pc+1, S'_{pc}, L_{pc})
`athrow`	$\{ S_{pc} \triangleright_{\text{head}} X_{pc} \}$

Table 6-1. Instruction Constraints

a. The object's type variable is plugged into S_{pc+1} instead of S'_{pc}, because for the proofs, we need the field and method components of the variable to appear at S_{pc+1}. The implementation instead has $S'_{pc} \triangleright_{\text{head}} T_{pc,\text{v}}$. The discrepancy can probably be corrected by adding "post-state" expressions to the expression syntax and extending the soundness proof to cover them.

- *CatchConstraints* gives constraints capturing the control flow for exceptions of class *classID* thrown at *pc* and caught in the method.

  CatchConstraints(pc, *classID*) =
  Succ(pc, (CodeLocMethod(pc), CatchBlockOffset(pc, *classID*)), $S'_{\text{exn-}pc\text{-}classID}$, L_{pc})
  $\cup \{ S'_{\text{exn-}pc\text{-}classID} \triangleright_{\text{head}} X_{pc} \}$

The last three sets of constraints are:

- Constraints ensuring that every static field has a type.
- Constraints expressing the possibility that an exception may be spontaneously thrown from at any instruction.
- Constraints specifying that the spontaneously thrown exceptions are objects of the classes found in the ErrorClassIDs.

The rules in Table 6-1 use the following functions:

- *LocalNames* computes the indices of the local variables used in *method*. LocalNames is used to make sure the values of all local variables are carried forward correctly when one of them is overwritten by a `store` instruction.

  LocalNames(*method*) =
  $\{ index \mid \exists i.$ Instruction($method$, i) $\in \{$`load` *index*, `store` *index*$\}\}$

- *Succ* computes the constraints that arise along control flow paths within a method, when one instruction is a successor of another in the control flow graph. Succ treats the transfer of control from one instruction to the next as if it were a function call, so that the instruction at *from* performs a “tail call” to the instruction at *to* to do the rest of the computation for the current method. S and L are the types for the working stack and the local variables respectively that are passed into *to*.

$$\mathrm{Succ}(\mathit{from}, \mathit{to}, S, L) = \\ \{\mathrm{S}_{to} \preccurlyeq_{\mathit{from\text{-}to}} S, \mathrm{L}_{to} \preccurlyeq_{\mathit{from\text{-}to}} L, \mathrm{G}_{to} \preccurlyeq_{\mathit{from\text{-}to}} \mathrm{G}_{\mathit{from}}, \mathrm{X}_{to} \preccurlyeq_{\mathit{from\text{-}to}} \mathrm{X}_{\mathit{from}}\} \cup \\ \{\mathrm{R}_{to} \preccurlyeq_{\mathit{from\text{-}to}} \mathrm{R}_{\mathit{from}}\}$$

- *MethodCall* computes the constraints needed to hook up a method call at a call site. M is the type for the method being called. $P0$ and $P1$ are the types of the parameters being passed in. G is the type of the globals object being passed in. X and R are the types of the exception and normal result returned, respectively.

$$\mathrm{MethodCall}(M, P0, P1, G, X, R) = \\ \{M \rhd_{\text{param-0}} P0, M \rhd_{\text{param-1}} P1, M \rhd_{\text{globals}} G, M \rhd_{\text{exn}} X, M \rhd_{\text{result}} R\}$$

6.4.5 Query Constraints

Additional constraints must be added to the set C to support queries over arbitrary bytecode expressions. These constraints depend on the queried expressions, and are detailed below in Section 6.5.3.2.

6.4.6 Canonical Constraint Set

C is a *canonical constraint set* if

$$\forall u, v.\ \{u \cong v\} \subseteq C \Rightarrow u = v\,.$$

Given a closed constraint set N,

Lemma 6-1. Let a closed constraint set N be given. Let M be a map from variables to variables such that

$$\forall u, v.\ \{u \cong v\} \subseteq N \vee u = v \Leftrightarrow M(u) = M(v)$$

M selects one representative element from each equivalence class. Such a map exists for any choice of N, because the closure of N implies the relation $\cong$ is an equivalence relation in N (lacking only reflexivity, which I restore with the disjunction).

Let C be defined as:

$$C = \{M(t) \rhd_c M(u) \mid \{t \rhd_c u\} \subseteq N\} \\ \cup \{M(t) \preccurlyeq_i M(u) \mid \{t \preccurlyeq_i u\} \subseteq N\} \\ \cup \{M(t) \cong M(t) \mid t \in \mathrm{dom}\, M\}$$

(C replaces each variable in N with the representative of its equivalence class.) Then C is trivially canonical. Furthermore, C is closed.

Proof: I prove the closure condition that $\{t \rhd_c u, t \rhd_c v\} \subseteq C$ implies $\{u \cong v\} \subseteq C$.

Suppose $\{t \rhd_c u, t \rhd_c v\} \subseteq C$. Then

$$\exists t', t'', u', v' \,.\, t = M(t') \wedge u = M(u') \wedge t = M(t'') \wedge v = M(v') \text{ where}$$
$$\{t' \rhd_c u', t'' \rhd_c v'\} \subseteq N$$

By definition of M,

$$M(t') = M(t'') \Rightarrow \{t' \cong t''\} \subseteq N \vee t' = t''$$

In either case of the disjunction,

$$\{t' \rhd_c u', t' \rhd_c v'\} \subseteq N$$

By closure of N,

$$\{u' \cong v'\} \subseteq N$$

This gives

$$M(u') = M(v') \text{, i.e., } u = v \text{ and therefore } \{u \cong v\} \subseteq C$$

The other closure conditions follow similarly. ■

The remainder of this chapter deals with canonical closed constraint sets. This eliminates the need to explicitly deal with equivalence constraints.

6.4.7 Example

The Java code in Figure 6-6 would generate bytecode as shown in Table 6-2.

```
class X {
           X f(X a)       { return this; }
    static X g(X c, X d) { return c.f(d); }
    static X main(X b)    { return g(new X(), b); }
}
```

Figure 6-6. A Simple Java Program

For this program, one might ask “can `main`’s result equal the new `X` object it creates?” We shall see how this question is answered by computing initial constraints (shown in Table 6-2) and then finding a closed form.

6.4.7.1 Initial Constraints

The constraints shown in Table 6-2 have been simplified from the real constraints in order to make the example simultaneously tractable and interesting. In particular, all the “successor instance” constraints have been replaced with equalities, which have then been eliminated by substitution. All of the constraints within methods relating to the stack (S) and local variable (L) variables have been solved and eliminated. All constraints relating to global variables and exceptions are irrelevant and have been elided.

6.4.7.2 Finding a Closed Form

SEMI would close the constraint set by generating additional constraints, as follows:

The equality constraints within `f` give

$$\{ M_f \rhd_{result} T_{f,p0} \}.$$

Bytecode	Induced Initial Constraints	
`class X {`	$M_f \preccurlyeq_{X\text{-}f} N_{X,f}$	$C_X \triangleright_f N_{X,f}$
`  f(this, p1) {`	$M_f \triangleright_{\text{param-0}} T_{f,p0}$	$M_f \triangleright_{\text{param-1}} T_{f,p1}$
	$M_f \triangleright_{\text{result}} R_{(f,0)}$	
`0:    load this;`		
`1:    return;`	$R_{(f,0)} = T_{f,p0}$	
`  }`		
`  static g(p0, p1) {`	$M_g \triangleright_{\text{param-0}} T_{g,p0}$	$M_g \triangleright_{\text{param-1}} T_{g,p1}$
	$M_g \triangleright_{\text{result}} R_{(g,0)}$	
`0:    load p0`		
`1:    load p1`		
`2: invokevirtual f;`	$T_{g,p0} \triangleright_f T_{(g,2),m}$	$T_{(g,2),m} \triangleright_{\text{param-0}} T_{g,p0}$
	$T_{(g,2),m} \triangleright_{\text{param-1}} T_{g,p1}$	
`3:    return;`	$T_{(g,2),m} \triangleright_{\text{result}} R_{(g,0)}$	
`  }`		
`  static main(p0) {`	$M_{main} \triangleright_{\text{param-0}} T_{main,p0}$	$M_{main} \triangleright_{\text{result}} R_{(main,0)}$
`0:    new X;`	$C_X \preccurlyeq_{(main,0)} T_{(main,0),v}$	
`1:    load p0;`		
`2:    invokestatic g;`	$M_g \preccurlyeq_{(main,2)} T_{(main,2),m}$	$T_{(main,2),m} \triangleright_{\text{param-0}} T_{(main,0),v}$
	$T_{(main,2),m} \triangleright_{\text{param-1}} T_{main,p0}$	
`3:    return;`	$T_{(main,2),m} \triangleright_{\text{result}} R_{(main,0)}$	
`  }`		
`}`		

Table 6-2. A Simple Bytecode Program and its Constraints

We propagate components of M_f to $N_{X,f}$ (using $M_f \preccurlyeq_{X\text{-}f} N_{X,f}$), getting

$\{ N_{X,f} \triangleright_{\text{param-0}} v, N_{X,f} \triangleright_{\text{result}} v \}$ (for some v where $T_{f,p0} \preccurlyeq_{X,f} v$).

Now we propagate $N_{X,f}$ and its components to the instance of C_X in `main` (using $C_X \preccurlyeq_{(main,0)} T_{(main,0),v}$), yielding

$\{ T_{(main,0),v} \triangleright_f s\ , s \triangleright_{\text{param-0}} v', s \triangleright_{\text{result}} v' \}$ (for some s and v where $N_{X,f} \preccurlyeq_{(main,0)} s$ and $v \preccurlyeq_{(main,0)} v'$).

In other words, we know in `main` that the object's `f` method aliases its first parameter and result. Now we need to work on `g`. The constraints for `g` contain $\{ T_{g,p0} \triangleright_f T_{(g,2),m}, T_{(g,2),m} \triangleright_{\text{param-0}} T_{g,p0}, T_{(g,2),m} \triangleright_{\text{result}} R_{(g,0)} \}$. So inside `g`, we know that we pass `p0` into `p0`'s `f` method, and the result of that method is returned from `g`. We do not assume anything else about `f` here.

We propagate `g`'s constraints to `main`, obtaining

$\{ T_{g,p0} \preccurlyeq_{(main,2)} T_{(main,0),v}\ , R_{(g,0)} \preccurlyeq_{(main,2)} R_{(main,0)} \}$

From here we get

{ $T_{(main,0),v} \rhd_f u$, $u \rhd_{param\text{-}0} T_{(main,0),v}$, $u \rhd_{result} R_{(main,0)}$ } (for some u where $T_{(g,2),m} \lesssim_{(main,2)} u$).

Now $T_{(main,0),v} \rhd_f u$ and $T_{(main,0),v} \rhd_f s$ require us to set

{ $u \cong s$ }

In other words, we have "discovered" the implementation of `f` that `g` uses.

From the param-0 components of u and s, we get

{ $v' \cong T_{(main,0),v}$, $v' \cong R_{(main,0)}$ }

Thus

{ $R_{(main,0)} \cong T_{(main,0),v}$ }.

Because the result of `new X` in main is assigned type $T_{(main,0),v}$, the conclusion is that the result of `main` may be the new `X`.

6.5 Extracting the VPR Approximation

In this section, I consider a canonical closed constraint set C, with associated map M mapping from the original variables to the variables of C, and a pair of bytecode expressions e_1 and e_2, and show how SEMI decides whether e_1 and e_2 are related in the VPR approximation.

6.5.1 Overview

Below, I define a judgement $(e, \bar{x}) \to (u, \bar{x}')$ that relates a bytecode expression e in some context $\bar{x}$ to a SEMI variable u with some "leftover context" $\bar{x}'$, which is a suffix of $\bar{x}$. A *context* is a sequence of instance labels. For first-order code, it corresponds to a call stack, each label naming a method call site or an instruction transition (recall that instruction transitions are treated as tail calls).

The SEMI variable u is referred to as the *ground type* of the expression in the context. A ground type is obtained by first ignoring the context and computing the *base type t* assigned to the expression by SEMI, for example, the type variable assigned to a local variable. Then we follow the chain of instances starting at t and labelled by the instance labels in the context $\bar{x}$ as far as possible, to obtain u, the "most specific" instance of t in context $\bar{x}$.

The "leftover context" $\bar{x}'$ is the suffix of $\bar{x}$ that was not dereferenced; it represents the outermost context at which some instance of t appears. For example, when u occurs as part of the type of a global variable, the leftover context is empty because an instance of u will occur at the top level.

The analysis concludes $e_1 \overline{\leftrightarrow} e_2$ if and only if

$$\exists u, \bar{x}_1, \bar{x}_2, \bar{x}_1', \bar{x}_2'. \ (e_1, \bar{x}_1) \to (u, \bar{x}_1') \wedge (e_2, \bar{x}_2) \to (u, \bar{x}_2')$$

The idea is that u is the type of a witness value that causes e_1 and e_2 to be related. The expressions are related if there is some plausible type u that is an instance (in any contexts) of both of the base types of e_1 and e_2.

6.5.2 Relating Bytecode Expressions to Variables

The inference rules in Figures 6-7, 6-8 and 6-9 define judgements of the form "$e \to u\langle \bar{c} \rangle$" (the "expression decomposition" relation), "$u\langle c \rangle \to u'$" (the "component evaluation" relation), and "$(u', \bar{x}) \to (v, \bar{x}')$" (the "instance evaluation" relation). These judgements are combined in Figure 6-10 to form the judgement "$(e, \bar{x}) \to (v, \bar{x}')$". In this section I prove a number of simple structural properties of these relations.

$$\frac{}{pc : \texttt{stack-0} \to \mathrm{S}_{pc}\langle \mathrm{head} :: \varepsilon \rangle}$$

$$\frac{}{pc : \texttt{exn} \to \mathrm{X}_{pc}\langle \varepsilon \rangle}$$

$$\frac{pc : \texttt{stack-}(n-1) \to \mathrm{S}_{pc}\langle \bar{c}' \rangle \qquad n > 0}{pc : \texttt{stack-}n \to \mathrm{S}_{pc}\langle \mathrm{tail} :: \bar{c}' \rangle}$$

$$\frac{}{pc : \texttt{local-}n \to \mathrm{L}_{pc}\langle n :: \varepsilon \rangle}$$

$$\frac{}{pc : \mathit{staticField} \to \mathrm{G}_{pc}\langle \mathit{staticField} :: \varepsilon \rangle}$$

$$\frac{pc : \mathit{exp} \to u\langle c_1 :: \ldots :: c_k :: \varepsilon \rangle}{pc : \mathit{exp}\,.\,\mathit{field} \to u\langle c_1 :: \ldots :: c_k :: \mathit{field} :: \varepsilon \rangle}$$

Figure 6-7. Rules defining the mapping from bytecode expressions to constraint variables and components

$$\frac{}{u\langle \varepsilon \rangle \to u}$$

$$\frac{\{u \rhd_c u''\} \subseteq C \qquad u''\langle \bar{c} \rangle \to u'}{u\langle c :: \bar{c} \rangle \to u'}$$

Figure 6-8. Rules defining evaluation through components

The expression decomposition relation maps a bytecode expression e to a representation of its base type, given as a basic type variable u (one of S_{pc}, G_{pc}, X_{pc}, or L_{pc}, for some pc), and a sequence of component labels $\bar{c}$ that must be followed from u to reach the base type for e. The component evaluation relation then takes u and dereferences the chain of component labels to reach a variable u' corresponding to the actual base type of e. Finally the instance evaluation relation finds the most specific instance of u' in context $\bar{x}'$.

The rest of this subsection proves several formal properties of these evaluation relations. Many of them are generalizations of the closure properties of constraint sets.

$$\frac{\{u \preccurlyeq_i u''\} \subseteq C \qquad (u'', \bar{x}) \to (u', \bar{x}')}{(u, i :: \bar{x}) \to (u', \bar{x}')}$$

$$\frac{\forall u'.\ u \preccurlyeq_i u' \notin C}{(u, i :: \bar{x}) \to (u, i :: \bar{x})}$$

$$\frac{}{(u, \varepsilon) \to (u, \varepsilon)}$$

Figure 6-9. Rules defining evaluation through instances

$$\frac{e \to u\langle\bar{c}\rangle \qquad M(u)\langle\bar{c}\rangle \to u' \qquad (u', \bar{x}) \to (v, \bar{x}')}{(e, \bar{x}) \to (v, \bar{x}')}$$

Figure 6-10. Rule assigning a ground variable to an expression in a given context

Lemma 6-2. Existence property. Instance evaluation is total:

$$\forall u, \bar{x}.\ \exists v, \bar{x}'.\ (u, \bar{x}) \to (v, \bar{x}')$$

Proof: The proof is by induction on the length of $\bar{x}$. The base case is trivial with $\bar{x} = \varepsilon$ and $v = u$. For the induction step, suppose $\bar{x} = i :: \bar{x}''$; either $\forall u'.\ u \preccurlyeq_i u' \notin C$ or $\exists u'.\ \{u \preccurlyeq_i u'\} \subseteq C$. In the former case, the result is trivial with $v = u$, $\bar{x}' = \bar{x}$. In the latter case, the induction hypothesis gives $(u', \bar{x}'') \to (v, \bar{x}')$ for some v, $\bar{x}'$ and the result follows. ■

Lemma 6-3. Uniqueness properties. Each of the relations is a (partial) function.

Proof: It is clear that exactly one rule from Figure 6-7 applies for each bytecode expression e. Therefore:

$$\forall e, u, u', \bar{c}, \bar{c}'.\ e \to u\langle\bar{c}\rangle \wedge e \to u'\langle\bar{c}'\rangle \Rightarrow u = u' \wedge \bar{c} = \bar{c}'$$

Exactly one rule from Figure 6-8 applies for each $u\langle\bar{c}\rangle$. (Note that if $\{u \rhd_c u'\} \subseteq C$ and $\{u \rhd_c u''\} \subseteq C$ then by closure of C, $\{u' \cong u''\} \subseteq C$ and hence $u' = u''$.) Therefore:

$$\forall u, \bar{c}, v, v'.\ u\langle\bar{c}\rangle \to v \wedge u\langle\bar{c}\rangle \to v' \Rightarrow v = v'$$

Exactly one rule from Figure 6-9 applies for each $(u, \bar{x})$. (Note that if $\{u \preccurlyeq_i u'\} \subseteq C$ and $\{u \preccurlyeq_i u''\} \subseteq C$ then by closure of C, $\{u' \cong u''\} \subseteq C$ and hence $u' = u''$.) Therefore:

$$\forall u, \bar{x}, v, v'.\ (u, \bar{x}) \to (v, \bar{x}') \wedge (u, \bar{x}) \to (v', \bar{x}'') \Rightarrow v = v' \wedge \bar{x}' = \bar{x}''$$

Putting these together gives:

$$\forall e, \bar{x}, v, v'.\ (e, \bar{x}) \to (v, \bar{x}') \wedge (e, \bar{x}) \to (v', \bar{x}'') \Rightarrow v = v' \wedge \bar{x}' = \bar{x}''$$ ■

Lemma 6-4. Component transitivity property. Component evaluation respects concatenation of component lists.

$$\forall u, \bar{c}, \bar{c}', v.\ u\langle \bar{c} \oplus \bar{c}'\rangle \to v \Leftrightarrow \exists t.\ u\langle \bar{c}\rangle \to t \wedge t\langle \bar{c}'\rangle \to v$$

Proof: The proof is by induction on the length of $\bar{c}$. The base case $\bar{c} = \varepsilon$ is trivial, with $t = u$. For the induction step, suppose $\bar{c} = c :: \bar{c}''$. In the forward direction, we have $u\langle c :: \bar{c}'' \oplus \bar{c}'\rangle \to v$. This requires $\{u \rhd_c u'\} \subseteq C \wedge u'\langle \bar{c}'' \oplus \bar{c}'\rangle \to v$ By the induction hypothesis, $\exists t.\ u'\langle \bar{c}''\rangle \to t \wedge t\langle \bar{c}'\rangle \to v$. But then $u\langle c :: \bar{c}''\rangle \to t$, as required.

In the reverse direction, we have $\exists t.\ u\langle c :: \bar{c}''\rangle \to t \wedge t\langle \bar{c}'\rangle \to v$. This requires $\{u \rhd_c u'\} \subseteq C \wedge u'\langle \bar{c}''\rangle \to t$. By the induction hypothesis, $u'\langle \bar{c}'' \oplus \bar{c}'\rangle \to v$. Then $u\langle c :: \bar{c}'' \oplus \bar{c}'\rangle \to v$, as required. ■

Lemma 6-5. Instance suffix property. In instance evaluation, the leftover context is a suffix of the initial context. When the difference between those contexts is itself used as the context for evaluation, the resulting leftover context is empty.

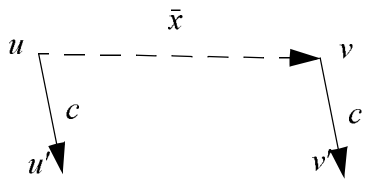
$$\forall u, \bar{x}, u', \bar{x}'.\ (u, \bar{x}) \to (u', \bar{x}') \Rightarrow (\exists v, \bar{y}.\ \bar{x} = \bar{y} \oplus \bar{x}' \wedge (u, \bar{y}) \to (v, \varepsilon))$$

Proof: The proof is by induction on the length of $\bar{x}$. For the base case $\bar{x} = \varepsilon$, the result is trivial, with $v = u$ and $\bar{y} = \varepsilon$. For the induction step, suppose $\bar{x} = i :: \bar{x}''$. Then either $\forall u''.\ u \preccurlyeq_i u'' \notin C$ or $\exists u''.\ \{u \preccurlyeq_i u''\} \subseteq C$. In the former case, the result is trivial with $v = u$, $\bar{x}' = \bar{x}$ and $\bar{y} = \varepsilon$. In the latter case, we have $(u'', \bar{x}'') \to (u', \bar{x}')$. The induction hypothesis gives $\exists v, \bar{y}.\ \bar{x}'' = \bar{y} \oplus \bar{x}' \wedge (u'', \bar{y}) \to (v, \varepsilon)$. Then $\bar{x} = (i :: \bar{y}) \oplus \bar{x}' \wedge (u, i :: \bar{y}) \to (v, \varepsilon)$, as required (substituting $i :: \bar{y}$ for $\bar{y}$). ■

Lemma 6-6. Component propagation property. Components propagate along instance chains.

$$\forall u, \bar{x}, u', v, c.\ (u, \bar{x}) \to (v, \varepsilon) \wedge \{u \rhd_c u'\} \subseteq C \Rightarrow$$
$$(\exists v'.\ (u', \bar{x}) \to (v', \varepsilon) \wedge \{v \rhd_c v'\} \subseteq C)$$

This property can be illustrated using the following diagram. In all the illustrations representing constraint sets, nodes represent variables. A dashed edge represents an instance constraint, or (as in this case) a sequence of instance constraints. A solid edge represents a component constraint, or a sequence of component constraints. The edges are labelled with their instance or component labels; the nodes are labelled with the names of the variables.


$$\begin{tikzcd} u \arrow[r, dashed, "\bar{x}"] \arrow[d, "c"] & v \arrow[d, "c"] \\ u' & v' \end{tikzcd}$$

Any closed set containing the left-hand component must also contain the right-hand component.

Proof: The proof is by induction on the length of $\bar{x}$. For the base case $\bar{x} = \varepsilon$, the result is trivial, with $v = u$ and $v' = u'$. For the induction step, suppose $\bar{x} = i :: \bar{x}''$. Then for some t, $\{u \preccurlyeq_i t\} \subseteq C$ and $(t, \bar{x}'') \to (v, \varepsilon)$. By closure of C, there exists t' such that

$\{t \rhd_c t'\} \subseteq C$ and $\{u' \preccurlyeq_i t'\} \subseteq C$. By the induction hypothesis, $\exists v'.\ (t', \bar{x}'') \to (v', \varepsilon) \wedge \{v \rhd_c v'\} \subseteq C$. It follows immediately that $(u', i :: \bar{x}'') \to (v', \varepsilon)$, as required. ■

Lemma 6-7. Instance transitivity property.

$$\forall u, \bar{x}, u'.\ (u, \bar{x}) \to (u', \varepsilon) \Rightarrow (\forall \bar{x}', v, \bar{w}.\ (u, \bar{x} \oplus \bar{x}') \to (v, \bar{w}) \Leftrightarrow (u', \bar{x}') \to (v, \bar{w}))$$

This property can be illustrated using the following diagram:

$$\begin{tikzcd} & u' \arrow[dr, dashed, "\bar{x}'"] & \\ u \arrow[ur, dashed, "\bar{x}"] \arrow[rr, dashed, "\bar{x} \oplus \bar{x}'"'] & & v \; {\scriptstyle \bar{w}} \end{tikzcd}$$

The small $\bar{w}$ indicates that the instance chains converge at v, in both cases yielding the same leftover instances $\bar{w}$.

Proof: The proof is by induction on the length of $\bar{x}$. For the base case $\bar{x} = \varepsilon$, the result is trivial, with $u = u'$. For the induction step, suppose $\bar{x} = i :: \bar{x}''$. Then for some t, $\{u \preccurlyeq_i t\} \subseteq C$ and $(t, \bar{x}'') \to (u', \varepsilon)$. By the induction hypothesis, $\forall \bar{x}', v, \bar{w}.\ (t, \bar{x}'' \oplus \bar{x}') \to (v, \bar{w}) \Leftrightarrow (u', \bar{x}') \to (v, \bar{w})$. Suppose $(u', \bar{x}') \to (v, \bar{w})$; then $(t, \bar{x}'' \oplus \bar{x}') \to (v, \bar{w})$ and hence $(u, i :: \bar{x}'' \oplus \bar{x}') \to (v, \bar{w})$, as required. On the other hand, suppose $(t, \bar{x}'' \oplus \bar{x}') \to (v, \bar{w})$; then $(u', \bar{x}') \to (v, \bar{w})$ as required. ■

Lemma 6-8. Instance convergence property. Suppose that u, u', s, s', c are given such that $\{u \rhd_c s, u' \rhd_c s'\} \subseteq C$. Suppose also that $(u, \bar{x}) \to (v, \bar{w})$, $(u', \bar{x}') \to (v, \bar{w})$, and $(s, \bar{x}) \to (t, \bar{w}')$, for some given $v, \bar{w}, t, \bar{w}'$. Then $(s', \bar{x}') \to (t, \bar{w}')$.

This can be illustrated as follows:

$$\begin{tikzcd} u \arrow[rr, dashed, "\bar{x}"] \arrow[d, "c"'] & & v \; {\scriptstyle \bar{w}} \arrow[d, "c"] & \\ s \arrow[rrr, dashed, "\bar{x}"] & & v' & t \; {\scriptstyle \bar{w}'} \\ u' \arrow[uurr, dashed, "\bar{x}'"] \arrow[d, "c"] & & & \\ s' \arrow[uurrr, dashed, "\bar{x}'"'] & & & \end{tikzcd}$$

Note how the instance evaluations of u and u' in contexts $\bar{x}$ and $\bar{x}'$ terminate at v with leftover instances $\bar{w}$, but evaluation of s and s' in the same contexts may "go past" v's corresponding component. (This can happen because v' may have some instances that v does not have. Conceptually, v could be the type of something that is local to a function, but which has a component v' that escapes to a wider context.) The important result here is that even though the evaluations of s and s' do not necessarily yield v', they do yield the same result.

Proof: The proof is as follows: By Lemma 6-5 (instance suffix), there exist $\bar{y}, \bar{y}'$ such that $\bar{x} = \bar{y} \oplus \bar{w} \wedge (u, \bar{y}) \to (v, \varepsilon)$ and $\bar{x}' = \bar{y}' \oplus \bar{w} \wedge (u', \bar{y}') \to (v, \varepsilon)$. By Lemma 6-6 (component propagation), $\exists v'.\ (s, \bar{y}) \to (v', \varepsilon) \wedge \{v \rhd_c v'\} \subseteq C$. Then by Lemma 6-7 (instance transitivity), $\forall r, \bar{z}.\ (s, \bar{y} \oplus \bar{w}) \to (r, \bar{z}) \Leftrightarrow (v', \bar{w}) \to (r, \bar{z})$. This implies $(v', \bar{w}) \to (t, \bar{w}')$.

By another application of component propagation, $\exists v''.\ (s', \bar{y}') \to (v'', \varepsilon) \wedge \{v \rhd_c v''\} \subseteq C$. Because C is closed and canonical, $v'' = v'$ (being matching components of v). Thus $(s', \bar{y}') \to (v', \varepsilon)$. Invoking instance transitivity, $\forall r, \bar{z}.\ (s', \bar{y}' \oplus \bar{w}) \to (r, \bar{z}) \Leftrightarrow (v', \bar{w}) \to (r, \bar{z})$. But $(v', \bar{w}) \to (t, \bar{w}')$ and therefore $(s', \bar{y}' \oplus \bar{w}) \to (t, \bar{w}')$, i.e. $(s', \bar{x}') \to (t, \bar{w}')$ as required. ■

Lemma 6-9. Generalized instance convergence property.

Suppose that $u, u', s, s', \bar{c}$ are given such that $u\langle\bar{c}\rangle \to s \wedge u'\langle\bar{c}\rangle \to s'$. Suppose also that $(u, \bar{x}) \to (v, \bar{w})$, $(u', \bar{x}') \to (v, \bar{w})$, and $(s, \bar{x}) \to (t, \bar{w}')$ for some given $v, \bar{w}, t, \bar{w}'$. Then $(s', \bar{x}') \to (t, \bar{w}')$.

$$\begin{tikzcd}
u \arrow[rr, dashed, "\bar{x}"] \arrow[d, "\bar{c}"] & & v \ {\scriptstyle \bar{w}} \arrow[d, "\bar{c}"] & \\
s \arrow[rr, dashed, "\bar{x}"] & & v' \arrow[r, dashed] & t \ {\scriptstyle \bar{w}'} \\
u' \arrow[urr, dashed, "\bar{x}'"] \arrow[d, "\bar{c}"] & & & \\
s' \arrow[uurr, dash, dashed, "\bar{x}'"] & & &
\end{tikzcd}$$

Proof: The proof is by induction on the length of $\bar{c}$. The base case is vacuous with $u = s$ and $u' = s'$. For the induction step, suppose $\bar{c} = c :: \bar{c}'$. Then $\exists r, r'.\ \{u \rhd_c r, u' \rhd_c r'\} \subseteq C \wedge r\langle\bar{c}'\rangle \to s \wedge r'\langle\bar{c}'\rangle \to s'$. By the existence property (Lemma 6-2), $\exists v', \bar{w}''.\ (r, \bar{x}) \to (v', \bar{w}'')$. Applying Lemma 6-8 (instance convergence), $(r', \bar{x}') \to (v', \bar{w}'')$. Then applying the induction hypothesis, $(s', \bar{x}') \to (t, \bar{w}')$. ■

Lemma 6-10. Instance propagation property.

$$\forall u, \bar{c}, v, u'.\ u\langle\bar{c}\rangle \to v \wedge \{u \preccurlyeq_i u'\} \subseteq C \Rightarrow (\exists v'.\ u'\langle\bar{c}\rangle \to v' \wedge \{v \preccurlyeq_i v'\} \subseteq C)$$

$$\begin{tikzcd}
u \arrow[r, dashed, "i"] \arrow[d, "\bar{c}"] & u' \arrow[d, "\bar{c}"] \\
v \arrow[r, dashed, "i"] & v'
\end{tikzcd}$$

Proof: The proof is by induction on the length of $\bar{c}$. It is trivially true for $\bar{c} = \varepsilon$, with $v = u$ and $v' = u'$. Suppose $\bar{c} = c :: \bar{c}'$. Then for some u'' we have $\{u \rhd_c u''\} \subseteq C$ and $u''\langle\bar{c}'\rangle \to v$. By closure of C, $\exists t.\ \{u'' \preccurlyeq_i t, u' \rhd_c t\} \subseteq C$. The induction hypothesis yields $\exists v'.\ t\langle\bar{c}'\rangle \to v' \wedge \{v \preccurlyeq_i v'\} \subseteq C$. Then $u'\langle\bar{c}\rangle \to v'$ as required. ■

6.5.3 Constraints to Support Query Expressions

6.5.3.1 Inadequacy of Program Constraints

The analysis requires variables to be associated with arbitrary bytecode expressions. This may not be possible using only the constraints that are derived from the program.

For example, consider the following method `m`:

```
static void m(Foo f) { System.out.println("Hello Kitty"); }
```

Suppose some tool requires SEMI to decide whether `m#0:f.fieldA` $\overline{\leftrightarrow}$ `m#0:f.fieldB` is in the VPR. (The syntax "`m#0`" denotes bytecode offset 0 in method `m`.) The method `m` does not mention `f`, and therefore there are no constraints naming the components of `f` in the context of `m`. Therefore, although one can show `m#0:f.fieldA` $\rightarrow \mathrm{L}_{\texttt{m\#0}}\langle 0 :: \texttt{fieldA}\rangle$, $\mathrm{L}_{\texttt{m\#0}}\langle 0 :: \texttt{fieldA}\rangle$ does not evaluate to any ground variable. If this situation were to stand, then the analysis would incorrectly deduce that the two expressions are not related, when in fact they may be.

6.5.3.2 Query Constraints

To solve this problem, SEMI takes as input a set Q of bytecode expressions required for the query, and decides $e_1 \overline{\leftrightarrow} e_2$ only for those e_i in Q. For each expression e in Q, constraints are added to the constraint set C, ensuring that for any context $\bar{x}$, $(e, \bar{x}) \rightarrow (u', \bar{x}')$ holds for some u', $\bar{x}'$.

Formally, for each e in Q, compute u and $c_1, ..., c_k$ such that $e \rightarrow u\langle c_1 :: ... :: c_k :: \varepsilon\rangle$. Choose fresh variables $v_1, ..., v_k$, and add the constraints $\{u \rhd_{c_1} v_1, v_1 \rhd_{c_2} v_2, ..., v_{k-1} \rhd_{c_k} v_k\}$ to N. Then $M(u)\langle c_1 :: ... :: c_k :: \varepsilon\rangle \rightarrow M(v_k)$. (If $k = 0$, then set $v_k = u$ and the result holds.) Thus we have, for any context $\bar{x}$, and for all e in Q, $e \rightarrow u\langle\bar{c}\rangle$ and $u\langle\bar{c}\rangle \rightarrow v$ for some $u, \bar{c}, v$. From above, $\forall\bar{x}.\ \exists t, \bar{x}'.\ (v, \bar{x}) \rightarrow (t, \bar{x}')$. Therefore, in summary:

$$\forall e \in Q.\ \forall\bar{x}.\ \exists t, \bar{x}'.\ (e, \bar{x}) \rightarrow (t, \bar{x}')$$

6.6 Implementing the Ajax Interface

The previous section specifies the approximation to the value-point relation computed by SEMI. This section describes an efficient implementation of the Ajax interface using this approximation. I describe how the Ajax interface is implemented in terms of a given closed constraint set; SEMI's algorithm for computing a closed constraint set is described in the next chapter.

Recall that the Ajax API specifies the following parameters to the analysis:

- A type D of intermediate data to be propagated
- A type R of tool target data
- An associative, commutative, idempotent binary "merge" operator $\mathrm{D_M} : \mathrm{D} \times \mathrm{D} \rightarrow \mathrm{D}$ with identity element $\mathrm{D_E}$

- A set S of source value-points from which data will be propagated
- A set T of target value-points to which data will be propagated
- An initial assignment of intermediate data to source value-points $D_I : S \to D$
- A map from target expressions to tool target data $T_R : T \to R$

The analysis computes:

$$\lambda t \in T.\ D_M\{D_S(s) \mid s \in S \wedge s \overline{\leftrightarrow} t\}$$

This is computed efficiently using a graph, similar to the method used by RTA (Section 5.3).

Note that the set of bytecode expressions Q used above in Section 6.5.3.2 can be taken simply as the union of S and T.

Multiple queries are treated separately. The intermediate data computations described below are local to each query.

6.6.1 The Graph

SEMI constructs a *propagation graph* with nodes

$$P_N = \{\text{In-}t \mid t \in \text{Variables}(C)\} \cup \{\text{Out-}t \mid t \in \text{Variables}(C)\}$$

and edges

$$\begin{aligned} P_E = {} & \{(\text{In-}u, \text{In-}v) \mid \exists i.\ \{u \lessdot_i v\} \subseteq C\} \\ & \cup \{(\text{Out-}v, \text{Out-}u) \mid \exists i.\ \{u \lessdot_i v\} \subseteq C\} \\ & \cup \{(\text{In-}t, \text{Out-}t) \mid t \in \text{Variables}(C)\} \end{aligned}$$

Lemma 6-11. Path invariant. SEMI relates $e_1 \overline{\leftrightarrow} e_2$ if and only if there is a path from In-u to Out-v where $e_1 \to u'\langle\bar{c}\rangle$, $u'\langle\bar{c}\rangle \to u$, $e_2 \to v'\langle\bar{d}\rangle$, and $v'\langle\bar{d}\rangle \to v$ for some u, v, u', v', $\bar{c}$, $\bar{d}$.

Intuitively, the two base types for the expressions have a common instance type if and only if there is a path from one base type to the other in the propagation graph (which is essentially two copies of the instance graph pasted together).

Proof: Suppose that SEMI relates $e_1 \overline{\leftrightarrow} e_2$. Then

$$\exists t, \bar{x}_1, \bar{x}_2, \bar{x}_1', \bar{x}_2'.\ (e_1, \bar{x}_1) \to (t, \bar{x}_1') \wedge (e_2, \bar{x}_2) \to (t, \bar{x}_2')$$

From the uniqueness properties of the relations, we have $(u, \bar{x}_1) \to (t, \bar{x}_1')$ and $(v, \bar{x}_2) \to (t, \bar{x}_2')$. (The existence of u, v, u', v', $\bar{c}$, $\bar{d}$ follows from the added query constraints, as discussed above in Section 6.4.5.) It follows that there is a path in the graph from In-u to In-t and from Out-t to Out-v. There is an edge from In-t to Out-t. Therefore, there is a path from In-u to Out-v.

Conversely, suppose there is a path from In-u to Out-v. There must exist an edge in the path connecting In-t to Out-t' for some t and t'. All such edges are of the form $(\text{In-}t, \text{Out-}t)$, therefore $t' = t$. Furthermore there is a path from In-u to In-t; this path passes only through In nodes (because there are no edges from any Out node back to an In node). This implies that for some sequence of instances $\bar{x}_1$, $(u, \bar{x}_1) \to (t, \varepsilon)$. Similarly there is path from Out-t to Out-v and for some $\bar{x}_2$, $(v, \bar{x}_2) \to (t, \varepsilon)$. Therefore $(e_1, \bar{x}_1) \to (t, \varepsilon) \wedge (e_2, \bar{x}_2) \to (t, \varepsilon)$ and SEMI will conclude $s \overline{\leftrightarrow} t$. ■

6.6.2 Computing Analysis Results

The results are computed efficiently over the graph using almost exactly the same algorithm as for RTA (Section 5.3.2). The only difference is the way in which expressions are mapped to nodes in the graph.

The assignment A over graph nodes is computed iteratively as follows:

$$A_0(y) = D_M\{D_S(s) \mid s \in S \wedge (\exists u, \bar{c}, u'.\ s \to u'\langle\bar{c}\rangle \wedge u'\langle\bar{c}\rangle \to u \wedge y = \text{In-}u)\}$$
$$A_{n+1}(y) = D_M(\{A_n(p) \mid (p, y) \in P_E\} \cup \{A_n(y)\})$$

The algorithm terminates when $A_{n+1}(y) = A_n(y)$. The result of the analysis is then:

$$\{(d, \mathrm{F}[\{\mathrm{A}(j_t) \mid \exists t \in \mathrm{T}.\mathrm{T_R}(t) = d \wedge t{:}j_t\}]) \mid d \in \text{range } \mathrm{T_R}\}$$

6.6.3 Incrementality

The algorithm for computing the closed constraint set is incremental, in the sense that adding new constraints to the initial set (e.g., in response to changes in the input program) will cause new constraints to be added to the closed result set. This process is discussed further in Chapter 7.

This means that new edges and nodes are added to an existing propagation graph. The results are updated incrementally in response to changes in the graph and in the analysis parameters, in much the same way as the RTA implementation operates (Section 5.3.5).

Because incremental extensions to the initial constraints are supported, there is actually no need to know the set Q of query expressions in advance. Whenever a new query expression is encountered, it is added to Q and everything is updated appropriately.

6.7 Proving Soundness

6.7.1 Overview

6.7.1.1 Strategy

Suppose a tagged trace $\underline{T} = <\underline{\Xi}_0, \ldots, \underline{\Xi}_n>$ is given.

In Section 6.7.2 below, we define a function Creation(v) mapping each tagged value v occurring in the trace to a pair (i, e'). The idea is that the first occurrence of v is in state $\underline{\Xi}_i$, and can be obtained by evaluating e' in that state.

In Section 6.7.4 we define a function Context(i), mapping each state index i to a context associated with state $\underline{\Xi}_i$. This context can be thought of as identifying, for each method in the call stack, which of the polymorphic instances of the method is active. The definition of the Context function requires an auxiliary CallerState function, defined in Section 6.7.3. CallerState(k) finds the state at which the "current method" executing in state $\underline{\Xi}_k$ was invoked.

Section 6.7.5 proves the following *conformance lemma*:

$$\forall i, e, v, u, \bar{x}' . (\underline{\Xi}_i, e) \twoheadrightarrow v \wedge (e, \text{Context}(i)) \rightarrow (u, \bar{x}') \Rightarrow$$
$$\exists i', e' . \text{Creation}(v) = (i', e') \wedge i' \leq i \wedge (e', \text{Context}(i')) \rightarrow (u, \bar{x}')$$

The idea is that given an expression evaluating to a value in a particular state, we can look back to where the value was created and determine the expression's ground type in terms of that creation state.

Soundness is a corollary of this lemma. By definition, two expressions related by the VPR must give the same value when evaluated in some pair of states. Applying the conformance lemma twice, once for each expression in its associated state, we show that the ground types of the expressions are both equal to the ground type of the value, and therefore equal to each other. Thus we can be sure that SEMI relates the two expressions.

Formally, suppose $e_1 \leftrightarrow e_2$ where $e_1, e_2 \in Q$. Then by definition there is a tagged trace $\underline{T}$ and states $\underline{\Xi}_i$ and $\underline{\Xi}_j$ in $\underline{T}$ such that $(\underline{\Xi}_i, e_1) \twoheadrightarrow v$ and $(\underline{\Xi}_j, e_2) \twoheadrightarrow v$ for some tagged v.

Choose u_1, $\bar{x}_1'$ such that $(e_1, \text{Context}(i)) \rightarrow (u_1, \bar{x}_1')$ and u_2, $\bar{x}_2'$ such that $(e_2, \text{Context}(j)) \rightarrow (u_2, \bar{x}_2')$ (they must exist according to Section 6.5.3.2). Then by the conformance lemma,

$$\exists i', e' . \text{Creation}(v) = (i', e') \wedge i' \leq i \wedge (e', \text{Context}(i')) \rightarrow (u_1, \bar{x}_1')$$
$$\exists i'', e'' . \text{Creation}(v) = (i'', e'') \wedge i'' \leq j \wedge (e'', \text{Context}(i'')) \rightarrow (u_2, \bar{x}_2')$$

In Section 6.7.2.1 below, I show that Creation is a function — i.e., $i' = i''$ and $e' = e''$. Therefore $u_1 = u_2$ and $\bar{x}_1' = \bar{x}_2'$ (Lemma 6-3). Thus the analysis concludes $e_1 \overline{\leftrightarrow} e_2$.

6.7.1.2 Note: Unique Justification for Transitions

Many of the proofs perform a case analysis of a transition $\underline{\Xi}_i \Rightarrow \underline{\Xi}_{i+1}$. This depends on the fact that, given two states related in this way, there is always exactly one inference rule justifying the transition.

To see that this is so consider the mode fields of the states $\underline{\Xi}_i$ and $\underline{\Xi}_{i+1}$. There are four possibilities:

Mode($\underline{\Xi}_i$)	Mode($\underline{\Xi}_{i+1}$)	
THROWING	THROWING	"Exception return" is the only applicable rule.
THROWING	RUNNING	"Exception catch" is the only applicable rule.
RUNNING	THROWING	"Spontaneous exception throw" is the only applicable rule.
RUNNING	RUNNING	The applicable rule is uniquely determined by the value of Instruction(PC($\underline{\Xi}_i$)).

6.7.2 The Creation Function

The creation function is defined by the rules given in Figure 6-11. I demonstrate two important properties: that it is a function, and that it is defined for all tagged values that appear in the trace.

6.7.2.1 "Creation" Is a Function

Lemma 6-12. For some arbitrary v, suppose that $\mathrm{Creation}(v) = (i, e)$ and $\mathrm{Creation}(v) = (i', e')$. We show that $i = i'$ and $e = e'$.

Proof: From the definition of the Creation function, $(\Xi_i, e) \twoheadrightarrow v$ and $(\Xi_{i'}, e') \twoheadrightarrow v$.

If $i = i' = 0$, then e must be of the form $(\mathrm{Main}, 0) : \mathit{staticField}$ and e' of the form $(\mathrm{Main}, 0) : \mathit{staticField'}$. Then $\mathrm{Tag}(v) = \mathrm{InitalTag}(\mathit{staticField}) = \mathrm{InitalTag}(\mathit{staticField'})$, hence $\mathit{staticField} = \mathit{staticField'}$ by the fact that InitialTag is defined to be a bijection.

If $i = 0$ but $i' > 0$, then $\mathrm{Tag}(v) \in \mathrm{Used}(\Xi_0)$, and then $\mathrm{Tag}(v) \in \mathrm{Used}(\Xi_{i'-1})$, since $\forall i, i'.\ i \le i' \Rightarrow \mathrm{Used}(\Xi_i) \subseteq \mathrm{Used}(\Xi_{i'})$. this fact is easily observed from the transition rules. But given that $\mathrm{Creation}(v) = (i', e')$, for each rule that can justify $\Xi_{i'-1} \Rightarrow \Xi_{i'}$, there is a constraint that $\mathrm{Tag}(v) \notin \mathrm{Used}(\Xi_{i'-1})$. Therefore this situation is impossible. Similar reasoning excludes $i' = 0$ with $i > 0$.

Consider $i > 0$ and $i' > 0$. Then $\mathrm{Tag}(v) \notin \mathrm{Used}(\Xi_{i-1}) \wedge \mathrm{Tag}(v) \notin \mathrm{Used}(\Xi_{i'-1})$, but $\mathrm{Tag}(v) \in \mathrm{Used}(\Xi_i) \wedge \mathrm{Tag}(v) \in \mathrm{Used}(\Xi_{i'})$, therefore $\mathrm{Tag}(v) \in \mathrm{Used}(\Xi_j) \wedge \mathrm{Tag}(v) \in \mathrm{Used}(\Xi_{i'})$ for all $j \ge i$ and $j \ge i'$. Therefore $i' - 1 < i$ and $i - 1 < i'$, i.e., $i = i'$.

Now consider the transition $\Xi_{i-1} \Rightarrow \Xi_i$. If it is justified by one of the rules for `aconst_null`, `bipush`, `iadd`, or `instanceof`, then $e = e' = \mathrm{PC}(\Xi_i) : \texttt{stack-0}$.

If the transition is justified by the rule for `new`, and $e \ne e'$, then one of e or e' must be of the form $\mathrm{PC}(\Xi_i) : \texttt{stack-0}.\mathit{field}$. Without loss of generality, suppose $e = \mathrm{PC}(\Xi_i) : \texttt{stack-0}.\mathit{field}$. Then there are two cases, $e' = \mathrm{PC}(\Xi_i) : \texttt{stack-0}$ or $e' = \mathrm{PC}(\Xi_i) : \texttt{stack-0}.\mathit{field'}$ where $\mathit{field} \ne \mathit{field'}$. Consulting the transition rule, the former case is impossible because $\mathrm{Tag}(v) = t = \mathit{tags}(\mathit{field})$ violates the condition $t \notin \mathrm{range}\ \mathit{tags}$. The latter case is impossible because $\mathrm{Tag}(v) = \mathit{tags}(\mathit{field}) = \mathit{tags}(\mathit{field'})$ violates the condition that *tags* is a bijection.

The same reasoning applies to the case in which the transition is justified by the rule for spontaneous exception throws, except that $e = \mathrm{PC}(\Xi_i) : \texttt{exn}.\mathit{field}$ and $e' = \mathrm{PC}(\Xi_i) : \texttt{exn}$ or $e' = \mathrm{PC}(\Xi_i) : \texttt{exn}.\mathit{field'}$. ■

6.7.3 The CallerState Function

6.7.3.1 Definition

The CallerState function determines at which state in a trace a method invocation began:

$$\mathrm{CallerState}(k) = \max\ \{ i \mid i < k \wedge \exists \mathit{frame}.\ \mathrm{MStack}(\Xi_k) = \mathit{frame} :: \mathrm{MStack}(\Xi_i) \}$$

$$\frac{\begin{array}{c}\underline{\Xi}_{i-1} \underline{\Rightarrow} \underline{\Xi}_i \quad \text{justified by rule for } \texttt{aconst\_null} \\ (\underline{\Xi}_i, \mathrm{PC}(\underline{\Xi}_i) : \texttt{stack-0}) \dashrightarrow v\end{array}}{\mathrm{Creation}(v) = (i, \mathrm{PC}(\underline{\Xi}_i) : \texttt{stack-0})}$$

$$\frac{\begin{array}{c}\underline{\Xi}_{i-1} \underline{\Rightarrow} \underline{\Xi}_i \quad \text{justified by rule for } \texttt{bipush } \textit{byte} \\ (\underline{\Xi}_i, \mathrm{PC}(\underline{\Xi}_i) : \texttt{stack-0}) \dashrightarrow v\end{array}}{\mathrm{Creation}(v) = (i, \mathrm{PC}(\underline{\Xi}_i) : \texttt{stack-0})}$$

$$\frac{\begin{array}{c}\underline{\Xi}_{i-1} \underline{\Rightarrow} \underline{\Xi}_i \quad \text{justified by rule for } \texttt{iadd} \\ (\underline{\Xi}_i, \mathrm{PC}(\underline{\Xi}_i) : \texttt{stack-0}) \dashrightarrow v\end{array}}{\mathrm{Creation}(v) = (i, \mathrm{PC}(\underline{\Xi}_i) : \texttt{stack-0})}$$

$$\frac{\begin{array}{c}\underline{\Xi}_{i-1} \underline{\Rightarrow} \underline{\Xi}_i \quad \text{justified by rule for } \texttt{new } \textit{classID} \\ (\underline{\Xi}_i, \mathrm{PC}(\underline{\Xi}_i) : \texttt{stack-0}) \dashrightarrow v\end{array}}{\mathrm{Creation}(v) = (i, \mathrm{PC}(\underline{\Xi}_i) : \texttt{stack-0})}$$

$$\frac{\begin{array}{c}\underline{\Xi}_{i-1} \underline{\Rightarrow} \underline{\Xi}_i \quad \text{justified by rule for } \texttt{new } \textit{classID} \\ (\underline{\Xi}_i, \mathrm{PC}(\underline{\Xi}_i) : \texttt{stack-0}.\textit{field}) \dashrightarrow v\end{array}}{\mathrm{Creation}(v) = (i, \mathrm{PC}(\underline{\Xi}_i) : \texttt{stack-0}.\textit{field})}$$

$$\frac{\begin{array}{c}\underline{\Xi}_{i-1} \underline{\Rightarrow} \underline{\Xi}_i \quad \text{justified by rule for } \texttt{instanceof } \textit{classID} \\ (\underline{\Xi}_i, \mathrm{PC}(\underline{\Xi}_i) : \texttt{stack-0}) \dashrightarrow v\end{array}}{\mathrm{Creation}(v) = (i, \mathrm{PC}(\underline{\Xi}_i) : \texttt{stack-0})}$$

$$\frac{\begin{array}{c}\underline{\Xi}_{i-1} \underline{\Rightarrow} \underline{\Xi}_i \quad \text{justified by rule for spontaneous exception throw} \\ (\underline{\Xi}_i, \mathrm{PC}(\underline{\Xi}_i) : \texttt{exn}) \dashrightarrow v\end{array}}{\mathrm{Creation}(v) = (i, \mathrm{PC}(\underline{\Xi}_i) : \texttt{exn})}$$

Figure 6-11. Rules defining the Creation function

$$\frac{\underline{\Xi}_{i-1} \Rightarrow \underline{\Xi}_i \quad \text{justified by rule for spontaneous exception throw} \qquad (\underline{\Xi}_i, \mathrm{PC}(\underline{\Xi}_i) : \texttt{exn}\,.\mathit{field}) \twoheadrightarrow v}{\mathrm{Creation}(v) = (i, \mathrm{PC}(\underline{\Xi}_i) : \texttt{exn}\,.\mathit{field})}$$

$$\frac{(\underline{\Xi}_0, (\mathrm{Main}, 0) : \mathit{staticField}) \twoheadrightarrow v}{\mathrm{Creation}(v) = (0, (\mathrm{Main}, 0) : \mathit{staticField})}$$

Figure 6-11. Rules defining the Creation function

It computes the state number i which called into the method active at state k, by finding the most recent state at which the call stack was one element shorter than the current call stack.

This function is used below to define the Context function. Here we prove some "obvious" but useful properties of the CallerState function that are required below. These properties are really invariants of the MJBC semantics ensuring that the call stack and the program counter behave in a disciplined way.

6.7.3.2 Scope of Definition

CallerState is defined whenever the run time stack is nonempty (i.e., the current method was called by some other method).

Lemma 6-13. The function CallerState is defined for all k such that $\mathrm{MStack}(\underline{\Xi}_k) \neq \varepsilon$.

Proof: To prove this, it suffices to prove that the set

$$\{i \mid i < k \wedge \exists \mathit{frame}.\ \mathrm{MStack}(\underline{\Xi}_k) = \mathit{frame} :: \mathrm{MStack}(\underline{\Xi}_i)\}$$

is nonempty if $\mathrm{MStack}(\underline{\Xi}_k) \neq \varepsilon$. This is shown by induction on k.

For $k = 0$, $\mathrm{MStack}(\underline{\Xi}_k) = \varepsilon$.

For $k > 0$, consider the transition $\underline{\Xi}_{k-1} \Rightarrow \underline{\Xi}_k$. If the transition was not justified by a rule for method invocation, method return, or exception return, then $\mathrm{MStack}(\underline{\Xi}_{k-1}) = \mathrm{MStack}(\underline{\Xi}_k)$ and the result follows from the induction hypothesis.

If the transition was a method return or exception return, then $\mathrm{MStack}(\underline{\Xi}_{k-1}) = f :: \mathrm{MStack}(\underline{\Xi}_k)$ for some f, and therefore $\mathrm{MStack}(\underline{\Xi}_{k-1}) \neq \varepsilon$. Applying the induction hypothesis, $\mathrm{CallerState}(k-1)$ is defined. Therefore there exists an j such that

$$j < k - 1 \wedge \exists \mathit{frame}.\ \mathrm{MStack}(\underline{\Xi}_{k-1}) = \mathit{frame} :: \mathrm{MStack}(\underline{\Xi}_j)$$

Hence $\mathrm{MStack}(\underline{\Xi}_j) = \mathrm{MStack}(\underline{\Xi}_k)$. Then, using the induction hypothesis again, if $\mathrm{MStack}(\underline{\Xi}_k) = \mathrm{MStack}(\underline{\Xi}_j) \neq \varepsilon$, then

$$\{i \mid i < j \wedge \exists \mathit{frame}.\ \mathrm{MStack}(\underline{\Xi}_j) = \mathit{frame} :: \mathrm{MStack}(\underline{\Xi}_i)\} \neq \varnothing$$
$$\{i \mid i < k \wedge \exists \mathit{frame}.\ \mathrm{MStack}(\underline{\Xi}_k) = \mathit{frame} :: \mathrm{MStack}(\underline{\Xi}_i)\} \neq \varnothing$$

If the transition was a method invocation, then for some f, $\mathrm{MStack}(\underline{\Xi}_k) = f :: \mathrm{MStack}(\underline{\Xi}_{k-1})$. Then the set

$\{ i \mid i < k \wedge \exists \mathit{frame}.\ \mathrm{MStack}(\underline{\Xi}_k) = \mathit{frame} :: \mathrm{MStack}(\underline{\Xi}_i) \}$ contains $k - 1$ and is nonempty. ■

6.7.3.3 Nested Call Stack

The call stack for the current state is a suffix of the call stack in every state during the lifetime of the current method invocation. In other words, the call stack may grow downward due to this method calling into another method, but the current activation record and the records above it on the stack are not popped or modified. We only need to prove this for states between the current state and the invocation of the current method.

Lemma 6-14. If $c = \mathrm{CallerState}(k)$ then
$\forall i.\ c < i \le k \Rightarrow (\mathrm{MStack}(\underline{\Xi}_k)$ is a suffix of $\mathrm{MStack}(\underline{\Xi}_i))$.

Proof: The proof is by induction on $k - i$.

For $k - i = 0$, the result is trivial.

Now consider $k - i = p$ where the induction hypothesis holds for $k - i = p - 1$. That is, assume $c < i < k$ and $\mathrm{MStack}(\underline{\Xi}_k)$ is a suffix of $\mathrm{MStack}(\underline{\Xi}_{i+1})$. Consider the transition $\underline{\Xi}_i \underline{\Rightarrow} \underline{\Xi}_{i+1}$.

If the transition is not justified by a rule for method invocation, method return, or exception return, then $\mathrm{MStack}(\underline{\Xi}_i) = \mathrm{MStack}(\underline{\Xi}_{i+1})$ and it follows immediately that $\mathrm{MStack}(\underline{\Xi}_k)$ is a suffix of $\mathrm{MStack}(\underline{\Xi}_i)$.

If the transition is a method return or exception return, then $\mathrm{MStack}(\underline{\Xi}_i) = f :: \mathrm{MStack}(\underline{\Xi}_{i+1})$ for some f, and again the result follows immediately.

If the transition is a method invocation, then for some f, $\mathrm{MStack}(\underline{\Xi}_{i+1}) = f :: \mathrm{MStack}(\underline{\Xi}_i)$. By the induction hypothesis, either $\mathrm{MStack}(\underline{\Xi}_{i+1}) = \mathrm{MStack}(\underline{\Xi}_k)$ or $\mathrm{MStack}(\underline{\Xi}_k)$ is a proper suffix of $\mathrm{MStack}(\underline{\Xi}_{i+1})$. In the latter case, $\mathrm{MStack}(\underline{\Xi}_k)$ is a suffix of $\mathrm{MStack}(\underline{\Xi}_i)$. In the former case, one obtains $\mathrm{MStack}(\underline{\Xi}_k) = f :: \mathrm{MStack}(\underline{\Xi}_i)$. But then i is an element of the set $\{ i' \mid i' < k \wedge \exists \mathit{frame}.\ \mathrm{MStack}(\underline{\Xi}_k) = \mathit{frame} :: \mathrm{MStack}(\underline{\Xi}_{i'}) \}$ and $i > c$, contradicting the definition of c. ■

6.7.3.4 Preservation of Caller State

The activation record on top of the call stack reflects the state just before we began the current method invocation.

Lemma 6-15. If $c = \mathrm{CallerState}(k)$ and $\mathrm{MStack}(\underline{\Xi}_k) = (pc, \underline{\mathcal{S}}, \underline{\mathcal{L}}) :: \underline{\mathcal{F}}$ then $\underline{\Xi}_c = [\text{pc: } pc, \text{wstack: } \underline{\mathcal{S}}, \text{locals: } \underline{\mathcal{L}}, \text{mstack: } \underline{\mathcal{F}}, \rho]$ for some value of ρ.

Proof: By the nested call stack lemma, $\mathrm{MStack}(\underline{\Xi}_k)$ is a suffix of $\mathrm{MStack}(\underline{\Xi}_{c+1})$. By the definition of c, $\exists \mathit{frame}.\ \mathrm{MStack}(\underline{\Xi}_k) = \mathit{frame} :: \mathrm{MStack}(\underline{\Xi}_c)$. Therefore $\mathrm{MStack}(\underline{\Xi}_c)$ is a proper suffix of $\mathrm{MStack}(\underline{\Xi}_{c+1})$, implying that the transition $\underline{\Xi}_c \underline{\Rightarrow} \underline{\Xi}_{c+1}$ must be a method call. The method call rules guarantee that $\mathrm{MStack}(\underline{\Xi}_{c+1}) = (pc, \underline{\mathcal{S}}, \underline{\mathcal{L}}) :: \underline{\mathcal{F}}$ where $\underline{\Xi}_c = [\text{pc: } pc, \text{wstack: } \underline{\mathcal{S}}, \text{locals: } \underline{\mathcal{L}}, \text{mstack: } \underline{\mathcal{F}}, \rho]$. Since $\mathrm{MStack}(\underline{\Xi}_k) = \mathit{frame} :: \underline{\mathcal{F}}$ and $\mathrm{MStack}(\underline{\Xi}_k)$ is a suffix of $(pc, \underline{\mathcal{S}}, \underline{\mathcal{L}}) :: \underline{\mathcal{F}}$, it follows that $\mathrm{MStack}(\underline{\Xi}_k) = (pc, \underline{\mathcal{S}}, \underline{\mathcal{L}}) :: \underline{\mathcal{F}}$. ■

6.7.3.5 Method Entry Correspondence

On beginning the current method invocation, the program counter was set to bytecode offset zero of the current method. The important thing to prove is that the method invocation actually invoked the same method as the current method.

Lemma 6-16. If $c = \mathrm{CallerState}(k)$ then $\mathrm{PC}(\underline{\Xi}_{c+1}) = (\mathrm{CodeLocMethod}(\mathrm{PC}(\underline{\Xi}_k)), 0)$.

Proof: The proof is by induction on $k - c$. Since $k > c$, the base case is $k = c + 1$. Let $(m, \mathit{offset}) = \mathrm{PC}(\underline{\Xi}_{c+1})$. Then $\mathrm{CodeLocMethod}(\mathrm{PC}(\underline{\Xi}_k)) = m$. Furthermore the transition $\underline{\Xi}_c \underline{\Rightarrow} \underline{\Xi}_{c+1}$ is a method call, and therefore $\mathit{offset} = 0$, as required.

Now suppose $c = \mathrm{CallerState}(k)$ and consider the transition $\underline{\Xi}_{k-1} \underline{\Rightarrow} \underline{\Xi}_k$. Whenever $\mathrm{MStack}(\underline{\Xi}_{k-1}) = \mathrm{MStack}(\underline{\Xi}_k)$ then the transition rule also requires $\mathrm{CodeLocMethod}(\mathrm{PC}(\underline{\Xi}_{k-1})) = \mathrm{CodeLocMethod}(\mathrm{PC}(\underline{\Xi}_k))$, and then the result follows from the induction hypothesis.

If the transition was a method invocation, then for some f $\mathrm{MStack}(\underline{\Xi}_k) = f :: \mathrm{MStack}(\underline{\Xi}_{k-1})$. But that implies $c = \mathrm{CallerState}(k) = k - 1$, which only occurs in the base case.

If the transition was a method return or exception return, then $\mathrm{MStack}(\underline{\Xi}_{k-1}) = (\mathrm{PC}(\underline{\Xi}_k) - x, \underline{\mathcal{S}}, \underline{\mathcal{L}}) :: \mathrm{MStack}(\underline{\Xi}_k)$ for some $\underline{\mathcal{S}}, \underline{\mathcal{L}}$, where $x = 0$ for exceptional returns and $x = 1$ for normal returns. Let $d = \mathrm{CallerState}(k - 1)$. By preservation of caller state (Lemma 6-15), $\mathrm{PC}(\underline{\Xi}_d) = \mathrm{PC}(\underline{\Xi}_k) - x$ and $\mathrm{MStack}(\underline{\Xi}_d) = \mathrm{MStack}(\underline{\Xi}_k)$. This also gives $\mathrm{CodeLocMethod}(\mathrm{PC}(\underline{\Xi}_d)) = \mathrm{CodeLocMethod}(\mathrm{PC}(\underline{\Xi}_k))$. Furthermore, by the nested call stack lemma (Lemma 6-14),

$$\forall i.\ d < i \le k - 1 \Rightarrow \mathrm{MStack}(\underline{\Xi}_k) \text{ is a suffix of } \mathrm{MStack}(\underline{\Xi}_i)$$

Therefore

$$\begin{aligned}\mathrm{CallerState}(k) &= \max\ \{i \mid i < k \wedge \exists \mathit{frame}.\ \mathrm{MStack}(\underline{\Xi}_k) = \mathit{frame} :: \mathrm{MStack}(\underline{\Xi}_i)\}\\ &= \max\ \{i \mid i \le d \wedge \exists \mathit{frame}.\ \mathrm{MStack}(\underline{\Xi}_k) = \mathit{frame} :: \mathrm{MStack}(\underline{\Xi}_i)\}\end{aligned}$$

But $\mathrm{MStack}(\underline{\Xi}_d) = \mathrm{MStack}(\underline{\Xi}_k)$ and therefore

$$\mathrm{CallerState}(k) = \max\ \{i \mid i < d \wedge \exists \mathit{frame}.\ \mathrm{MStack}(\underline{\Xi}_k) = \mathit{frame} :: \mathrm{MStack}(\underline{\Xi}_i)\}$$

That is, $\mathrm{CallerState}(k) = \mathrm{CallerState}(d) = c$. Now we appeal to the induction hypothesis applied to d. ■

6.7.4 The Context Function

The Context function maps a state index to a list of instance labels, identifying exactly which polymorphic instance of each currently active method was invoked.

6.7.4.1 Definition of the Context Function

The Context function is defined inductively as follows:

$$\text{Context}(0) = \varepsilon$$

For $i > 0$, Context(i) depends on the form of the transition $\underline{\Xi}_{i-1} \Rightarrow \underline{\Xi}_i$.

Case: The transition is justified by the rule for `invokestatic`.

$$\text{Context}(i) = \text{PC}(\underline{\Xi}_{i-1}) :: \text{Context}(i-1)$$

Case: The transition is justified by the rule for `invokevirtual`.

Then $\underline{\Xi}_{i-1}$ is of the form $[\text{pc: } pc, \text{ wstack: } v_1 :: v_0 :: \mathcal{S}, \text{ locals: } \mathcal{L}, \text{ mstack: } \mathcal{J}, \text{ heap: } \mathcal{H}, \rho]$, and $\text{Instruction}(pc) = \texttt{invokevirtual}\ \mathit{methodID}$. Let $(i', e) = \text{Creation}(v_0)$ and $\mathit{classID} = \text{HeapObjClass}(\mathcal{H}(\text{Val}(v_0)))$. Now consider the transition $\underline{\Xi}_{i'-1} \Rightarrow \underline{\Xi}_{i'}$. If it is justified by the rule for `new`, set

$$\text{Context}(i) = \mathit{classID}\text{-}\mathit{methodID} :: \text{PC}(\underline{\Xi}_{i'-1}) :: \text{Context}(i')$$

Otherwise it is justified by the rule for spontaneous exception throws, since that is the only other creating rule which adds a mapping for $\text{Val}(v_0)$ to $\mathcal{H}$. Set

$$\text{Context}(i) = \mathit{classID}\text{-}\mathit{methodID} :: \text{err-}\mathit{classID} :: \text{err-PC}(\underline{\Xi}_{i'-1}) :: \text{Context}(i')$$

Case: The transition is justified by the rule for `return`.

$$\text{Context}(i) = (\text{PC}(\underline{\Xi}_i) - 1)\text{-PC}(\underline{\Xi}_i) :: \text{Context}(\text{CallerState}(i-1))$$

$\text{CallerState}(i-1)$ is well-defined because $\text{MStack}(\underline{\Xi}_{i-1})$ must be nonempty for the `return` to execute successfully.

Case: The transition is justified by the rule for exceptional returns.

$$\text{Context}(i) = \text{Context}(\text{CallerState}(i-1))$$

The reason for the asymmetry between normal and exceptional returns is that a normal return transfers control to the instruction following the method invocation, but an exceptional return does not.

Case: The transition is justified by a rule for exception throws (either an execution of `athrow` or a spontaneous exception throw)..

$$\text{Context}(i) = \text{Context}(i-1)$$

Exception throw transitions simply change the state from RUNNING to THROWING and do not themselves transfer control.

Case: All other transitions induce the following rule:

$$\text{Context}(i) = \text{PC}(\underline{\Xi}_{i-1})\text{-PC}(\underline{\Xi}_i) :: \text{Context}(i-1)$$

6.7.4.2 Preservation of Return Types

This lemma proves that the return type R_{pc} and the type X_{pc} of any thrown exception at some instruction *pc* map correctly to the actual return type and exception type of the method.

Lemma 6-17. The return type and thrown exception type inferred for a method correspond to the return type and exception type actually used in all contexts.

$$\forall i, m, c.\ m = \text{CodeLocMethod}(\text{PC}(\Xi_i)) \wedge c = \text{CallerState}(i) \Rightarrow$$
$$\exists \bar{w}.\ \text{Context}(i) = \bar{w} \oplus \text{Context}(c+1)$$
$$\wedge\ (M(\text{R}_{\text{PC}(\Xi_i)}), \bar{w}) \to (M(\text{R}_{(m,0)}), \varepsilon) \wedge (M(\text{X}_{\text{PC}(\Xi_i)}), \bar{w}) \to (M(\text{X}_{(m,0)}), \varepsilon)$$

Proof: The proof is by induction on $i - c$.

The fact $c = \text{CallerState}(i)$ implies $c < i$. Therefore the base case is $i = c + 1$. Set $\bar{w} = \varepsilon$, and the result is trivial, noting $\text{PC}(\Xi_{c+1}) = (m, 0)$ by the method entry correspondence lemma (Lemma 6-16).

Now consider the transition $\Xi_{i-1} \Rightarrow \Xi_i$.

Case: The transition is an exception throw. Then $\text{PC}(\Xi_{i-1}) = \text{PC}(\Xi_i)$ and $\text{Context}(i) = \text{Context}(i-1)$. Also $\text{MStack}(\Xi_{i-1}) = \text{MStack}(\Xi_i)$ implying $c = \text{CallerState}(i) = \text{CallerState}(i-1)$. We apply the induction hypothesis to get

$$\exists \bar{w}.\ \text{Context}(i-1) = \bar{w} \oplus \text{Context}(c+1)$$
$$\wedge\ (M(\text{R}_{\text{PC}(\Xi_{i-1})}), \bar{w}) \to (M(\text{R}_{(m,0)}), \varepsilon) \wedge (M(\text{X}_{\text{PC}(\Xi_{i-1})}), \bar{w}) \to (M(\text{X}_{(m,0)}), \varepsilon)$$

This is equivalent to the desired result.

Case: The transition is the normal execution of an instruction other than `invokestatic`, `invokevirtual` or `return`. Then let $pc = \text{PC}(\Xi_{i-1})$ and $pc' = \text{PC}(\Xi_i)$; then $\text{CodeLocMethod}(pc) = \text{CodeLocMethod}(pc')$, and $\text{Context}(i) = pc\text{-}pc' :: \text{Context}(i-1)$. Also $\text{MStack}(pc) = \text{MStack}(pc')$ implying $c = \text{CallerState}(i) = \text{CallerState}(i-1)$. We apply the induction hypothesis to get

$$\exists \bar{w}'.\ \text{Context}(i-1) = \bar{w}' \oplus \text{Context}(c+1)$$
$$\wedge\ (M(\text{R}_{pc}), \bar{w}') \to (M(\text{R}_{(m,0)}), \varepsilon) \wedge (M(\text{X}_{pc}), \bar{w}') \to (M(\text{X}_{(m,0)}), \varepsilon)$$

The executed instruction induces the constraints $\text{Succ}(pc, pc', s, l)$ for some s and l. Therefore $\{M(\text{R}_{pc'}) \lessdot_{pc\text{-}pc'} M(\text{R}_{pc}), M(\text{X}_{pc'}) \lessdot_{pc\text{-}pc'} M(\text{X}_{pc})\} \subseteq C$. Set $\bar{w} = pc\text{-}pc' :: \bar{w}'$. Then $\text{Context}(i) = \bar{w} \oplus \text{Context}(c+1)$ and

$$(M(\text{R}_{pc'}), \bar{w}) \to (M(\text{R}_{(m,0)}), \varepsilon) \wedge (M(\text{X}_{pc'}), \bar{w}) \to (M(\text{X}_{(m,0)}), \varepsilon)$$

as required.

Case: The transition was a method invocation. Then for some f $\text{MStack}(\Xi_i) = f :: \text{MStack}(\Xi_{i-1})$. But that implies $c = \text{CallerState}(i) = i - 1$, which only occurs in the base case, so this case cannot occur.

Case: The transition was a method return or exceptional return. Then $\mathrm{MStack}(\underline{\Xi}_{i-1}) = (\mathrm{PC}(\underline{\Xi}_i) - x, \underline{\mathcal{S}}, \underline{\mathcal{L}}) :: \mathrm{MStack}(\underline{\Xi}_i)$ for some $\underline{\mathcal{S}}, \underline{\mathcal{L}}$, where $x = 0$ for exceptional returns and $x = 1$ for normal returns. Let $d = \mathrm{CallerState}(i-1)$. By preservation of caller state, $\mathrm{PC}(\underline{\Xi}_d) = \mathrm{PC}(\underline{\Xi}_i) - x$ and $\mathrm{MStack}(\underline{\Xi}_d) = \mathrm{MStack}(\underline{\Xi}_i)$. This also gives $m = \mathrm{CodeLocMethod}(\mathrm{PC}(\underline{\Xi}_i)) = \mathrm{CodeLocMethod}(\mathrm{PC}(\underline{\Xi}_d))$. Furthermore, by the nested call stack lemma, $\forall i'.\ d < i' \le k - 1 \Rightarrow \mathrm{MStack}(\underline{\Xi}_i)$ is a suffix of $\mathrm{MStack}(\underline{\Xi}_{i'})$. Therefore $c = \mathrm{CallerState}(i) = \mathrm{CallerState}(d)$. Now we appeal to the induction hypothesis applied to d, yielding

$$\begin{aligned}&\exists \overline{w}'.\ \mathrm{Context}(d) = \overline{w}' \oplus \mathrm{Context}(c+1)\\ &\wedge (M(\mathrm{R}_{\mathrm{PC}(\underline{\Xi}_{di})}), \overline{w}') \to (M(\mathrm{R}_{(m,0)}), \varepsilon) \wedge (M(\mathrm{X}_{\mathrm{PC}(\underline{\Xi}_d)}), \overline{w}') \to (M(\mathrm{X}_{(m,0)}), \varepsilon)\end{aligned}$$

If the transition was an exceptional return, then $\mathrm{PC}(\underline{\Xi}_d) = \mathrm{PC}(\underline{\Xi}_i)$ and $\mathrm{Context}(i) = \mathrm{Context}(d)$; the required result is obtained by setting $\overline{w} = \overline{w}'$.

Otherwise the transition was a normal return. Then $\mathrm{PC}(\underline{\Xi}_d) = \mathrm{PC}(\underline{\Xi}_i) - 1$ and $\mathrm{Context}(i) = (\mathrm{PC}(\underline{\Xi}_i) - 1)\text{-}\mathrm{PC}(\underline{\Xi}_i) :: \mathrm{Context}(\mathrm{CallerState}(i-1))$. The method invocation instruction at d induces the constraints $\mathrm{Succ}(\mathrm{PC}(\underline{\Xi}_d), \mathrm{PC}(\underline{\Xi}_i), s, l)$ for some s and l. Therefore

$$\begin{aligned}&\{M(\mathrm{R}_{\mathrm{PC}(\underline{\Xi}_i)}) \preccurlyeq_{(\mathrm{PC}(\underline{\Xi}_i)-1)\text{-}\mathrm{PC}(\underline{\Xi}_i)} M(\mathrm{R}_{\mathrm{PC}(\underline{\Xi}_i)-1}),\\ &M(\mathrm{X}_{\mathrm{PC}(\underline{\Xi}_i)}) \preccurlyeq_{(\mathrm{PC}(\underline{\Xi}_i)-1)\text{-}\mathrm{PC}(\underline{\Xi}_i)} M(\mathrm{X}_{\mathrm{PC}(\underline{\Xi}_i)-1})\} \subseteq C\end{aligned}$$

Set $\overline{w} = (\mathrm{PC}(\underline{\Xi}_i) - 1)\text{-}\mathrm{PC}(\underline{\Xi}_i) :: \overline{w}'$. Then

$$\begin{aligned}&\mathrm{Context}(i) = \overline{w} \oplus \mathrm{Context}(c+1)\\ &(M(\mathrm{R}_{\mathrm{PC}(\underline{\Xi}_i)}), \overline{w}) \to (M(\mathrm{R}_{(m,0)}), \varepsilon) \wedge (M(\mathrm{X}_{\mathrm{PC}(\underline{\Xi}_i)}), \overline{w}) \to (M(\mathrm{X}_{(m,0)}), \varepsilon) \qquad \blacksquare\end{aligned}$$

6.7.5 Proving the Conformance Lemma

Lemma 6-18. To reprise Section 6.7.1.1, the conformance lemma states:

$$\begin{aligned}&\forall i, e, v, u, \bar{x}'.\ (\underline{\Xi}_i, e) \dashrightarrow v \wedge (e, \mathrm{Context}(i)) \to (u, \bar{x}) \Rightarrow\\ &\exists i', e'.\ \mathrm{Creation}(v) = (i', e') \wedge i' \le i \wedge (e', \mathrm{Context}(i')) \to (u, \bar{x})\end{aligned}$$

The proof is by induction on i. The induction hypothesis is strengthened to note that, in every state, the ground type for the global variable record is the type given to it at the beginning of Main:

$$\begin{aligned}&\forall i.\ ((M(\mathrm{G}_{\mathrm{PC}(\underline{\Xi}_i)}), \mathrm{Context}(i)) \to (M(\mathrm{G}_{(\mathrm{Main},0)}), \varepsilon)\\ &\wedge \forall e, v, u, \bar{x}'.\ (\underline{\Xi}_i, e) \dashrightarrow v \wedge (e, \mathrm{Context}(i)) \to (u, \bar{x}) \Rightarrow\\ &\exists i', e'.\ \mathrm{Creation}(v) = (i', e') \wedge i' \le i \wedge (e', \mathrm{Context}(i')) \to (u, \bar{x}))\end{aligned}$$

The base case is proved in Section 6.7.5.1. It is trivial.

For the induction step, I assume the hypothesis is true for $i \le k$ and prove it true for $i = k + 1$.

The basic strategy to prove the induction result is to show that most transitions $\Xi_k \Rightarrow \Xi_{k+1}$ "preserve types" by extending the context with an instance label (corresponding to method call or intra-method control flow) and by making the types of local variables (and stack locations) at the old code location appropriate instances of the types of local variables (and stack locations) at the new code location. This ensures that the ground type obtained for e evaluated in $\mathrm{Context}(k+1)$ is the same as when it is evaluated in $\mathrm{Context}(k)$, and we can appeal to the induction hypothesis to show that it is the correct $(u, \bar{x})$.

This is not possible for all transitions, because most transitions change the program state, and therefore for some expressions e the value obtained by evaluating e in the new state differs from the result of evaluating e in the old state. Typically these cases are proved by showing that the initial constraints require the type of e to be related to the type of some other expression e', where e' in the old state evaluates to the same value as e in the new state. This allows us to again appeal to the induction hypothesis.

Some other cases require different techniques. For example, transitions that create new values prove the result by appealing directly to the definition of Creation, without resorting to the induction hypothesis. As another example, the return instruction truncates the Context for the current state back to the Context of the caller; this case requires the "preservation of return types" Lemma 6-17 from above, as well as other machinery.

In Section 6.7.5.3 we prove the first part of the induction result itself: $(M(\mathrm{G}_{\mathrm{PC}(\Xi_{k+1})}), \mathrm{Context}(i)) \to (M(\mathrm{G}_{(\mathrm{Main},\,0)}), \varepsilon)$. The proof is relatively simple because it does not depend on e and only requires a case analysis of the transition $\Xi_k \Rightarrow \Xi_{k+1}$. Furthermore, only a few transitions modify global variables.

Section 6.7.5.4 proves the rest of the induction result for expressions e of the form $\mathrm{PC}(\Xi_{k+1}) : exp\,.f$, assuming it holds for $\mathrm{PC}(\Xi_{k+1}) : exp$. This step also requires case analysis of $\Xi_k \Rightarrow \Xi_{k+1}$. Again, most of the cases are easy because most transitions do not modify object fields.

Section 6.7.5.5 proves the result for expressions of the form $\mathrm{PC}(\Xi_{k+1}) : staticField$. Again, only a few transitions modify static fields.

The simple expressions referring to stack and local variables require the most work, and are handled in Section 6.7.5.6 and following sections. For these expressions, we perform a case analysis of the form of the transition and then break down the expression type within each transition, according to the manner in which stack and local variables are modified by the transition. (Almost every transition modifies the working stack or local variables in some way.)

The proof is simplified by codifying the strategy described above (which relates the expression e to some expression e', where e' in the old state evaluates to the same value as e in the new state) using a "reduction function" (Section 6.7.5.7) mapping e to e'. The proof also uses a "succession lemma" (Section 6.7.5.8), which captures the invariants induced by the use of the Succ function in the initial constraints. Nevertheless for each transition, some case analysis of the form of e is required.

One key supporting lemma is proved in the context of the induction hypothesis: Lemma 6-19 in Section 6.7.5.2. This lemma shows that at the invocation of a virtual method, the type of the method body actually invoked matches the type assigned to the method at the invocation site, in the sense that they have the same set of ground types. (It is not necessarily the case that one is an instance of the other.) This is used to show that virtual method calls and returns preserve types. This lemma follows by showing that the type assigned to the object at the invocation site matches the object's type at its creation, which is a consequence of the induction hypothesis.

6.7.5.1 Base Case

The base case is $i = 0$. Suppose $(\underline{\Xi}_0, e) \twoheadrightarrow v$. By the definition of a trace,

$\underline{\Xi}_0$ = [mode: RUNNING, pc: (Main, 0), wstack: ε, locals: [], mstack: ε, heap: [], globals: <u>InitStaticFields</u>, used: range InitialTags]

In this state, expressions of the form pc : `stack-`n and pc : `local-`n do not evaluate to anything. Also, since the heap is empty, expressions of the form $pc : exp\,.field$ do not evaluate to anything. Therefore e must be of the form (Main, 0):*staticField*. Therefore $\mathrm{Creation}(v) = (0, e)$, i.e. $i' = 0 = i$ and $e' = e$; noting that $\mathrm{PC}(\underline{\Xi}_0) = (\mathrm{Main}, 0)$ and $\mathrm{Context}(0) = \varepsilon$ gives the induction result.

6.7.5.2 Preservation of Virtual Call Types

Lemma 6-19. The types inferred for a virtual method implementation match up with the types inferred at each call site.

$$
\begin{aligned}
&\forall i, \mathit{methodID}, \mathit{methodImpl}, c, v, v', u, \bar{x}.\\
&\underline{\Xi}_i \Rrightarrow \underline{\Xi}_{i+1} \wedge i \le k \wedge \mathrm{Instruction}(\mathrm{PC}(\underline{\Xi}_i)) = \texttt{invokevirtual}\ \ \mathit{methodID}\\
&\wedge \mathrm{PC}(\underline{\Xi}_{i+1}) = (\mathit{methodImpl}, 0) \wedge \mathrm{Mode}(\underline{\Xi}_i) = \mathrm{Mode}(\underline{\Xi}_{i+1}) = \mathrm{RUNNING}\\
&\wedge \mathcal{M}(\mathrm{T}_{\mathrm{PC}(\underline{\Xi}_i),\,\mathrm{v0}})\langle \mathit{methodID} :: c :: \varepsilon\rangle \rightarrow v \wedge \{\mathcal{M}(\mathrm{M}_{\mathit{methodImpl}}) \rhd_c v'\} \subseteq C\\
&\Rightarrow\\
&((v, \mathrm{Context}(i)) \rightarrow (u, \bar{x}) \Leftrightarrow (v', \mathrm{Context}(i+1)) \rightarrow (u, \bar{x}))
\end{aligned}
$$

Proof: Then $\underline{\Xi}_i$ is of the form [pc: pc, wstack: $v_1 :: v_0 :: \underline{\mathcal{S}}$, locals: $\underline{\mathcal{L}}$, mstack: $\underline{\mathcal{Q}}$, heap: $\underline{\mathcal{H}}$, ρ]. Let $(i', e') = \mathrm{Creation}(v_0)$ and $\mathit{classID} = \mathrm{HeapObjClass}(\underline{\mathcal{H}}(\mathrm{Val}(v_0)))$. We then let $pc' = \mathrm{PC}(\underline{\Xi}_{i+1}) = (\mathit{methodImpl}, 0)$, where $\mathit{methodImpl} = \mathrm{Dispatch}(\mathit{classID}, \mathit{methodID})$.

Let $pc'' = \mathrm{PC}(\underline{\Xi}_{i'-1})$. Consider the transition $\underline{\Xi}_{i'-1} \Rrightarrow \underline{\Xi}_{i'}$. The transition adds a mapping for v_0 in the heap, therefore the transition is either an execution of `new` or a spontaneous exception throw. In Lemma 6-20 below, we show that in either case, for some $\bar{w}, s, s', s'', \bar{c}$, $e' \rightarrow s'\langle\bar{c}\rangle$, $s'\langle\bar{c}\rangle \rightarrow s''$, $s''\langle \mathit{methodID} :: c :: \varepsilon\rangle \rightarrow s$ and $(v', \bar{w}) \rightarrow (s, \varepsilon)$

where $\text{Context}(i+1) = \bar{w} \oplus \text{Context}(i')$. This means that the created object, in the context in which it is created, has a type *s* for the given component *c* of the object's method *methodID*, and *s* is an instance of the type v' we observe for the method's component in state Ξ_{i+1}.

The constraints for the `invokevirtual` instruction include

$$\{ S_{pc} \rhd_{\text{tail}} T_{pc,\text{t1}}, S_{pc} \rhd_{\text{head}} T_{pc,\text{v1}}, T_{pc,\text{t1}} \rhd_{\text{tail}} T_{pc,\text{t2}}, T_{pc,\text{t1}} \rhd_{\text{head}} T_{pc,\text{v0}}, \\ T_{pc,\text{v0}} \rhd_{methodID} T_{pc,\text{m}}, T_{pc,\text{m}} \rhd_{\text{globals}} G_{pc} \}$$

We have $(\Xi_k, pc:\texttt{stack-1}) \twoheadrightarrow v_0$, $pc:\texttt{stack-1} \to (S_{pc}, \text{tail} :: \text{head} :: \varepsilon)$, and $(M(S_{pc}), \text{tail} :: \text{head} :: \varepsilon) \to M(T_{pc,\text{v0}})$. Now, for some $u', \bar{x}'$, $(M(T_{pc,\text{v0}}), \text{Context}(i)) \to (u', \bar{x}')$ and then $(pc:\texttt{stack-1}, \text{Context}(i)) \to (u', \bar{x}')$. By the induction hypothesis, $(e', \text{Context}(i')) \to (u', \bar{x}')$, i.e. $(s'', \text{Context}(i')) \to (u', \bar{x}')$. It is valid to apply the induction hypotheses because $i \le k$.

Now assume $(v, \text{Context}(i)) \to (u, \bar{x})$. Applying the generalized instance convergence property with $\bar{c} = methodID :: c :: \varepsilon$ gives $(s, \text{Context}(i')) \to (u, \bar{x})$. Then, recalling $(v', \bar{w}) \to (s, \varepsilon)$, we have $(v', \bar{w} \oplus \text{Context}(i')) \to (u, \bar{x})$, i.e. $(v', \text{Context}(i+1)) \to (u, \bar{x})$.

Conversely, assuming $(v', \text{Context}(i+1)) \to (u, \bar{x})$, i.e. $(v', \bar{w} \oplus \text{Context}(i')) \to (u, \bar{x})$, and knowing $(v', \bar{w}) \to (s, \varepsilon)$, the instance transitivity property shows $(s, \text{Context}(i')) \to (u, \bar{x})$. Applying the generalized instance convergence property with $\bar{c} = methodID :: c :: \varepsilon$ gives $(v, \text{Context}(i)) \to (u, \bar{x})$. ■

Lemma 6-20. Sub-lemma of Lemma 6-19: For some $\bar{w}, s, s', s'', \bar{c} : e' \to s' \langle \bar{c} \rangle$, $s' \langle \bar{c} \rangle \to s''$, $s'' \langle methodID :: c :: \varepsilon \rangle \to s$ and $(v', \bar{w}) \to (s, \varepsilon)$ where $\text{Context}(i+1) = \bar{w} \oplus \text{Context}(i')$.

Proof: The proof is by a case analysis of the transition $\Xi_{i'-1} \Rightarrow \Xi_{i'}$, introduced above.

Case: The transition $\Xi_{i'-1} \Rightarrow \Xi_{i'}$ is justified by the rule for `new`. Then $\text{PC}(\Xi_{i'}) = pc'' + 1$ and $\text{Context}(i+1) = classID\text{-}methodID :: pc'' :: \text{Context}(i')$

The constraints for `new` give

$$\{ S'_{pc''} \rhd_{\text{tail}} S_{pc''}, S_{pc''+1} \rhd_{\text{head}} T_{pc'',\text{v}}, N_{classID} \lessdot_{pc''} T_{pc'',\text{v}} \} \cup \\ \text{Succ}(pc'', pc''+1, S'_{pc''}, L_{pc''})$$

We also have the initial constraints

$$\{ M_{methodImpl} \lessdot_{classID\text{-}methodID} N_{classID,methodID}, N_{classID} \rhd_{methodID} N_{classID,methodID} \}$$

Because v_0 has a mapping in the heap, $e' = \text{PC}(\Xi_{i'}):\texttt{stack-0}$ (the other expressions created by `new` do not have heap mappings.). Now $\text{PC}(\Xi_{i'}):\texttt{stack-0} \to S_{pc''+1} \langle \text{head} :: \varepsilon \rangle$ and $M(S_{pc''+1}) \langle \text{head} :: \varepsilon \rangle \to T_{pc'',\text{v}}$.

From the program constraints and the assumption $\{M(\mathrm{M}_{methodImpl}) \triangleright_c v'\} \subseteq C$, we get $(v', classID\text{-}methodID :: pc'' :: \varepsilon) \to (s, \varepsilon)$ for some s, where $M(\mathrm{T}_{pc'', \mathrm{v}})\langle methodID :: c :: \varepsilon\rangle \to s$.

So we set $\bar{w} = classID\text{-}methodID :: pc'' :: \varepsilon$, $\bar{c} = \text{head} :: \varepsilon$, $s' = \mathrm{S}_{pc'' + 1}$, and $s'' = \mathrm{T}_{pc'', \mathrm{v}}$.

Case: The transition $\Xi_{i' - 1} \Rightarrow \Xi_{i'}$ is justified by the rule for spontaneous exception throws. Then $\mathrm{PC}(\Xi_{i'}) = pc''$ and

$$\mathrm{Context}(i + 1) = classID\text{-}methodID :: \text{err-}classID :: \text{err-}pc'' :: \mathrm{Context}(i').$$

The relevant initial constraints are

$$\{ \mathrm{Err} \leqslant_{\text{err-}pc''} \mathrm{X}_{pc''},\ \mathrm{N}_{classID} \leqslant_{\text{err-}classID} \mathrm{Err},\ \mathrm{M}_{\mathrm{Dispatch}(classID,\, methodID)} \leqslant_{classID\text{-}methodID} \mathrm{N}_{classID,methodID},\ \mathrm{N}_{classID} \triangleright_{methodID} \mathrm{N}_{classID,methodID} \}$$

Thus $(M(\mathrm{G}_{\mathrm{PC}(\Xi_{k+1})}), classID\text{-}methodID :: \text{err-}classID :: \text{err-PC}(\Xi_{i'-1}) :: \varepsilon) \to (s, \varepsilon)$ for some s, where $M(\mathrm{X}_{pc''})\langle methodID :: \text{globals} :: \varepsilon\rangle \to s$.

Because v_0 has a mapping in the heap, $e' = \mathrm{PC}(\Xi_{i'}) : \texttt{exn}$ (the other expressions created do not have heap mappings.). Now $\mathrm{PC}(\Xi_{i'}) : \texttt{exn} \to \mathrm{X}_{pc''}\langle\varepsilon\rangle$ and $\mathrm{X}_{pc''}\langle\varepsilon\rangle \to \mathrm{X}_{pc''}$.

From the program constraints and the assumption $\{M(\mathrm{M}_{methodImpl}) \triangleright_c v'\} \subseteq C$, we get $(v', classID\text{-}methodID :: \text{err-}classID :: \text{err-PC}(\Xi_{i'-1}) :: \varepsilon) \to (s, \varepsilon)$ for some s, where $\mathrm{X}_{pc''}\langle methodID :: c :: \varepsilon\rangle \to s$.

So we set $\bar{w} = classID\text{-}methodID :: \text{err-}classID :: \text{err-PC}(\Xi_{i'-1}) :: \varepsilon$, $\bar{c} = \varepsilon$, and $s' = s'' = \mathrm{X}_{pc''}$. ■

6.7.5.3 Globals Hypothesis

Here we prove the global variables "ground type" invariant that we used to strengthen the induction hypothesis

Lemma 6-21. Consider the cases governing the form of $\mathrm{Context}(k + 1)$. For each case we show

$$(M(\mathrm{G}_{\mathrm{PC}(\Xi_{k+1})}), \mathrm{Context}(k + 1)) \to (M(\mathrm{G}_{(\mathrm{Main},\, 0)}), \varepsilon).$$

Proof: The proof is by a case analysis of the form of the transition $\Xi_k \Rightarrow \Xi_{k+1}$.

Case: The transition $\Xi_k \Rightarrow \Xi_{k+1}$ is justified by the rule for `invokestatic`.

Then

$$\mathrm{Context}(k + 1) = \mathrm{PC}(\Xi_k) :: \mathrm{Context}(k)$$

Let $methodImpl = \mathrm{CodeLocMethod}(\mathrm{PC}(\Xi_{k+1}))$. By the induction hypothesis, $(M(\mathrm{G}_{\mathrm{PC}(\Xi_k)}), \mathrm{Context}(k)) \to (\mathrm{G}_{(\mathrm{Main},\, 0)}, \varepsilon)$. The `invokestatic` instruction induces the constraints in N:

$\{ \mathrm{M}_{methodImpl} \preccurlyeq_{\mathrm{PC}(\Xi_j)} \mathrm{T}_{\mathrm{PC}(\Xi_k),\, \mathrm{m}}, \mathrm{M}_{methodImpl} \rhd_{\mathrm{globals}} \mathrm{G}_{\mathrm{PC}(\Xi_{k+1})},$
$\mathrm{T}_{\mathrm{PC}(\Xi_k),\, \mathrm{m}} \rhd_{\mathrm{globals}} \mathrm{G}_{\mathrm{PC}(\Xi_k)} \}$

By closure of C, $\{M(\mathrm{G}_{\mathrm{PC}(\Xi_{k+1})}) \preccurlyeq_{\mathrm{PC}(\Xi_j)} M(\mathrm{G}_{\mathrm{PC}(\Xi_k)})\} \subseteq C$.

Therefore

$$(M(\mathrm{G}_{\mathrm{PC}(\Xi_{k+1})}), \mathrm{Context}(k+1)) \to (M(\mathrm{G}_{(\mathrm{Main},\,0)}), \varepsilon).$$

Case: The transition is justified by the rule for `invokevirtual`.

Choose *methodID* such that $\mathrm{Instruction}(\mathrm{PC}(\Xi_k)) = \texttt{invokevirtual}\ \ methodID$, and *methodImpl* such that $\mathrm{PC}(\Xi_{k+1}) = (methodImpl, 0)$. Set $c = \mathrm{globals}$, $i = k$, $v' = M(\mathrm{G}_{\mathrm{PC}(\Xi_{k+1})})$ and $v = M(\mathrm{G}_{\mathrm{PC}(\Xi_k)})$. The intial constraints contain

$$\{\mathrm{M}_{methodImpl} \rhd_{\mathrm{globals}} \mathrm{G}_{\mathrm{PC}(\Xi_{k+1})}, \mathrm{T}_{\mathrm{PC}(\Xi_k),\, \mathrm{m}} \rhd_{\mathrm{globals}} \mathrm{G}_{\mathrm{PC}(\Xi_k)}, \mathrm{T}_{\mathrm{PC}(\Xi_k),\, \mathrm{v0}} \rhd_{methodID} \mathrm{G}_{\mathrm{PC}(\Xi_k)}\}$$

Also, by the induction hypothesis, $(M(\mathrm{G}_{\mathrm{PC}(\Xi_k)}), \mathrm{Context}(k)) \to (M(\mathrm{G}_{(\mathrm{Main},\,0)}), \varepsilon)$.

Now we appeal to the preservation of virtual call types (Lemma 6-19) to obtain

$$(M(\mathrm{G}_{\mathrm{PC}(\Xi_{k+1})}), \mathrm{Context}(k+1)) \to (M(\mathrm{G}_{(\mathrm{Main},\,0)}), \varepsilon)$$

Case: The transition is justified by the rule for `return`.

Then

$$\mathrm{Context}(k+1) = (\mathrm{PC}(\Xi_{k+1}) - 1)\text{-}\mathrm{PC}(\Xi_{k+1}) :: \mathrm{Context}(\mathrm{CallerState}(k))$$

Let $pc = \mathrm{PC}(\Xi_{\mathrm{CallerState}(k)})$. The rule for `return` implies $\mathrm{PC}(\Xi_{k+1}) = pc + 1$, using an application of Lemma 6-15 regarding preservation of caller state.

By the induction hypothesis, $(M(\mathrm{G}_{pc}), \mathrm{Context}(\mathrm{CallerState}(k))) \to (M(\mathrm{G}_{(\mathrm{Main},\,0)}), \varepsilon)$. The method invocation instructions both induce the constraints $\mathrm{Succ}(pc, pc{+}1, \mathrm{S}'_{pc}, \mathrm{L}_{pc})$, which include

$$\{\mathrm{G}_{pc+1} \preccurlyeq_{(\mathrm{PC}(\Xi_{k+1}) - 1)\text{-}\mathrm{PC}(\Xi_{k+1})} \mathrm{G}_{pc}\}$$

Therefore

$$(M(\mathrm{G}_{\mathrm{PC}(\Xi_{k+1})}), \mathrm{Context}(k+1)) \to (M(\mathrm{G}_{(\mathrm{Main},\,0)}), \varepsilon)$$

Case: The transition is justified by the rule for exceptional returns.

Then

$$\mathrm{Context}(k+1) = \mathrm{Context}(\mathrm{CallerState}(k))$$

Let $pc = \mathrm{PC}(\Xi_{\mathrm{CallerState}(k)})$. The rule for exceptional returns implies $\mathrm{PC}(\Xi_{k+1}) = pc$. But then $M(\mathrm{G}_{\mathrm{PC}(\Xi_{k+1})}) = M(\mathrm{G}_{\mathrm{PC}(\Xi_{\mathrm{CallerState}(k)})})$; applying the induction hypothesis gives

$$(M(\mathrm{G}_{\mathrm{PC}(\Xi_{\mathrm{CallerState}(k)})}), \mathrm{Context}(\mathrm{CallerState}(k))) \to (M(\mathrm{G}_{(\mathrm{Main},\,0)}), \varepsilon)$$

This is identical to the required result, taking the equalities into account.

Case: The transition is justified by a rule for exception throws.

Then

$$\text{Context}(k+1) = \text{Context}(k)$$

The two exception throw transition rules guarantee $\text{PC}(\Xi_k) = \text{PC}(\Xi_{k+1})$. Therefore applying the induction hypothesis gives

$$(M(\text{G}_{\text{PC}(\Xi_k)}), \text{Context}(k)) \to (M(\text{G}_{(\text{Main}, 0)}), \varepsilon)$$

Case: All other transitions induce the following rule:

$$\text{Context}(k+1) = \text{PC}(\Xi_k)\text{-PC}(\Xi_{k+1}) :: \text{Context}(k)$$

Let $pc = \text{PC}(\Xi_k)$ and $pc' = \text{PC}(\Xi_{k+1})$.

By the induction hypothesis, $(M(\text{G}_{pc}), \text{Context}(k)) \to (M(\text{G}_{(\text{Main}, 0)}), \varepsilon)$. The rules for these transitions all require the execution of an instruction which induces the constraints $\text{Succ}(pc, pc', \text{S}'_{pc}, \text{L}_{pc})$ — except for the rule for exception catch. The exception catch rule requires $handler = \text{CatchBlockOffset}((method, offset), \text{HeapObjClass}(\mathcal{H}(ref)))$ where $pc' = (method, handler)$ and $pc = (method, offset)$ for some $offset$. But then the constraints $\text{Succ}(pc, pc', \text{S}'_{\text{exn-}pc\text{-}classID}, \text{L}_{pc})$ are in the initial constraints. In either case,

$$\{M(\text{G}_{pc'}) \lessdot_{pc\text{-}pc'} M(\text{G}_{pc})\} \subseteq C$$

and therefore

$$(M(\text{G}_{pc'}), \text{Context}(k+1)) \to (M(\text{G}_{(\text{Main}, 0)}), \varepsilon) \qquad \blacksquare$$

6.7.5.4 Field Dereferences

Now we prove Lemma 6-18 for expressions e of the form $\text{PC}(\Xi_{k+1}) : exp\,.f$.

The rules for expression evaluation require that for some value of ref, $(\Xi_{k+1}, \text{PC}(\Xi_{k+1}) : exp) \twoheadrightarrow ref$ and $v = \text{HeapObjFields}(\text{Heap}(\Xi_{k+1})(\text{Val}(ref)))(f)$. Let p be defined as

$$p = \min\,\{i \mid v = \text{HeapObjFields}(\text{Heap}(\Xi_i)(\text{Val}(ref)))(f)\}$$

Note that $p > 0$ because $\text{Heap}(\Xi_0)$ is empty, and $p \le k+1$. Therefore $v \ne \text{HeapObjFields}(\text{Heap}(\Xi_{p-1})(\text{Val}(ref)))(f)$. Inspection of the tagged transition rules shows that there are three rules that could change the mapping for $\text{Val}(ref)$ from state Ξ_{p-1} to state Ξ_p: the rule for `new`, the rule for spontaneous exception throws, and the rule for `putfield`. In each case, the changed field(s) require

$$f \in \text{dom InitFields}(\text{HeapObjClass}(\mathcal{H}(\text{Val}(ref)))).$$

Let $pc = \text{PC}(\Xi_{p-1})$.

We can use the induction hypothesis to obtain

$$\forall v', u'.\ (\Xi_{k+1}, \text{PC}(\Xi_{k+1}) : exp) \twoheadrightarrow v' \wedge (\text{PC}(\Xi_{k+1}) : exp, \text{Context}(k+1)) \to (u', \bar{x}') \Rightarrow$$
$$\exists i', e'.\ \text{Creation}(v') = (i', e') \wedge i' \le k+1 \wedge (e', \text{Context}(i')) \to (u', \bar{x}')$$

We have $(\underline{\underline{\Xi}}_{k+1}, \mathrm{PC}(\underline{\underline{\Xi}}_{k+1}) : exp) \twoheadrightarrow ref$. Also, $(\mathrm{PC}(\underline{\underline{\Xi}}_{k+1}) : exp\,.f, \mathrm{Context}(k+1)) \to (u, \bar{x})$ requires, for some $t, \bar{c}, t'$, $\mathrm{PC}(\underline{\underline{\Xi}}_{k+1}) : exp\,.f \to t\langle \bar{c} \oplus (f :: \varepsilon)\rangle$ where $\mathrm{PC}(\underline{\underline{\Xi}}_{k+1}) : exp \to t\langle \bar{c}\rangle$, $M(t)\langle \bar{c} \oplus (f :: \varepsilon)\rangle \to t'$, and $(t', \mathrm{Context}(k+1)) \to (u, \bar{x})$.

By Lemma 6-6, there exists t'' such that $M(t)\langle \bar{c}\rangle \to t''$ and $t''\langle f :: \varepsilon\rangle \to t'$, i.e. $\{t'' \rhd_f t'\} \subseteq C$. By Lemma 6-2, there exist u'', $\bar{x}''$ such that $(t'', \mathrm{Context}(k+1)) \to (u'', \bar{x}'')$. Thus $\mathrm{PC}(\underline{\underline{\Xi}}_{k+1}) : exp \to (u'', \bar{x}'')$ and $\mathrm{Creation}(ref) = (i', e') \wedge i' \le k+1 \wedge (e', \mathrm{Context}(i')) \to (u'', \bar{x}'')$ for some i', e'.

The rest of the induction hypothesis is proven using a case split on the form of the transition $\underline{\underline{\Xi}}_{p-1} \underline{\Rightarrow} \underline{\underline{\Xi}}_p$.

Case: $\underline{\underline{\Xi}}_{p-1} \underline{\Rightarrow} \underline{\underline{\Xi}}_p$ is justified by the rule for `new` *classID*, where $classID = \mathrm{HeapObjClass}(\underline{\mathscr{H}}(\mathrm{Val}(ref)))$.

Then $(\underline{\underline{\Xi}}_p, \mathrm{PC}(\underline{\underline{\Xi}}_p) : \texttt{stack-0}) \twoheadrightarrow ref$ and $(\underline{\underline{\Xi}}_p, \mathrm{PC}(\underline{\underline{\Xi}}_p) : \texttt{stack-0}\,.f) \twoheadrightarrow v$, giving $\mathrm{Creation}(v) = (p, \mathrm{PC}(\underline{\underline{\Xi}}_p) : \texttt{stack-0}\,.f)$ and $\mathrm{Creation}(ref) = (p, \mathrm{PC}(\underline{\underline{\Xi}}_p) : \texttt{stack-0})$ by definition.

It remains to be shown that $(\mathrm{PC}(\underline{\underline{\Xi}}_p) : \texttt{stack-0}\,.f, \mathrm{Context}(p)) \to (u, \bar{x})$. From above, we have $\mathrm{Creation}(ref) = (i', e') \wedge i' \le k+1 \wedge (e', \mathrm{Context}(i')) \to (u'', \bar{x}'')$ for some i', e'. But because Creation is a function (Lemma 6-12), we have $i' = p$ and $e' = \mathrm{PC}(\underline{\underline{\Xi}}_p) : \texttt{stack-0}$, giving $(\mathrm{PC}(\underline{\underline{\Xi}}_p) : \texttt{stack-0}, \mathrm{Context}(p)) \to (u'', \bar{x}'')$. Therefore for some s, $\{M(\mathrm{S}_{pc+1}) \rhd_{\mathrm{head}} s\} \subseteq C$ and $(s, \mathrm{Context}(p)) \to (u'', \bar{x}'')$.

The `new` instruction induces these constraints in N:

$$\{\ \mathrm{S}'_{pc} \rhd_{\mathrm{tail}} \mathrm{S}_{pc},\ \mathrm{S}_{pc+1} \rhd_{\mathrm{head}} \mathrm{T}_{pc,\mathrm{v}},\ \mathrm{N}_{classID} \preccurlyeq_{pc} \mathrm{T}_{pc,\mathrm{v}}\ \}$$
$$\cup\ \{\ \mathrm{T}_{pc,\mathrm{v}} \rhd_f \mathrm{T}_{pc,f}\ \} \cup \mathrm{Succ}(pc, pc+1, \mathrm{S}'_{pc}, \mathrm{L}_{pc})$$

These imply $\{\mathrm{S}_{pc+1} \rhd_{\mathrm{head}} \mathrm{T}_{pc,\mathrm{v}}, \mathrm{T}_{pc,\mathrm{v}} \rhd_f \mathrm{T}_{pc,f}\} \subseteq N$, which in turn imply $\{M(\mathrm{S}_{pc+1}) \rhd_{\mathrm{head}} M(\mathrm{T}_{pc,\mathrm{v}}), M(\mathrm{T}_{pc,\mathrm{v}}) \rhd_f M(\mathrm{T}_{pc,f})\} \subseteq C$. Clearly

$$\mathrm{PC}(\underline{\underline{\Xi}}_p) : \texttt{stack-0}\,.f \to \mathrm{S}_{pc+1}\langle \mathrm{head} :: f :: \varepsilon\rangle$$
$$M(\mathrm{S}_{pc+1})\langle \mathrm{head} :: f :: \varepsilon\rangle \to M(\mathrm{T}_{pc,f})$$

Thus all that remains to be proved is $(M(\mathrm{T}_{pc,f}), \mathrm{Context}(p)) \to (u, \bar{x})$.

The facts $\{M(\mathrm{S}_{pc+1}) \rhd_{\mathrm{head}} s\} \subseteq C$ and $\{M(\mathrm{S}_{pc+1}) \rhd_{\mathrm{head}} M(\mathrm{T}_{pc,\mathrm{v}}), M(\mathrm{T}_{pc,\mathrm{v}}) \rhd_f M(\mathrm{T}_{pc,f})\} \subseteq C$ give $s = M(\mathrm{T}_{pc,\mathrm{v}})$ and $\{s \rhd_f M(\mathrm{T}_{pc,f})\} \subseteq C$. Above we showed $(s, \mathrm{Context}(p)) \to (u'', \bar{x}'')$, $(t', \mathrm{Context}(k+1)) \to (u, \bar{x})$, $(t'', \mathrm{Context}(k+1)) \to (u'', \bar{x}'')$, and $\{t'' \rhd_f t'\} \subseteq C$. Now we can invoke the instance convergence property (Lemma 6-8) to obtain the required

$$(M(\mathrm{T}_{pc,f}), \mathrm{Context}(p)) \to (u, \bar{x})$$

Case: $\underline{\underline{\Xi}}_{p-1} \underline{\Rightarrow} \underline{\underline{\Xi}}_p$ is justified by the rule for `putfield`*f*.

Then $(\underline{\Xi}_{p-1}, \mathrm{PC}(\underline{\Xi}_{p-1}):\texttt{stack-1}) \twoheadrightarrow \mathit{ref}$ and $(\underline{\Xi}_{p-1}, \mathrm{PC}(\underline{\Xi}_{p-1}):\texttt{stack-0}) \twoheadrightarrow v$. We show that $(\mathrm{PC}(\underline{\Xi}_{p-1}):\texttt{stack-0}, \mathrm{Context}(p-1)) \to (u, \bar{x})$; the main result then follows immediately by appealing to the induction hypothesis.

The `putfield` instruction induces these constraints in N:

$$\{ \mathrm{S}_{pc} \rhd_{\mathrm{tail}} \mathrm{T}_{pc,\mathrm{t}}, \mathrm{S}_{pc} \rhd_{\mathrm{head}} \mathrm{T}_{pc,\mathrm{v}}, \mathrm{T}_{pc,\mathrm{t}} \rhd_{\mathrm{tail}} \mathrm{S}'_{pc}, \mathrm{T}_{pc,\mathrm{t}} \rhd_{\mathrm{head}} \mathrm{T}_{pc,\mathrm{obj}}, \mathrm{T}_{pc,\mathrm{obj}} \rhd_f \mathrm{T}_{pc,\mathrm{v}} \} \cup \mathrm{Succ}(pc, pc{+}1, \mathrm{S}'_{pc}, \mathrm{L}_{pc})$$

Clearly then, $\mathrm{PC}(\underline{\Xi}_{p-1}):\texttt{stack-1} \to \mathrm{S}_{pc}\langle \mathrm{tail} :: \mathrm{head} :: \varepsilon\rangle$ and $M(\mathrm{S}_{pc})\langle \mathrm{tail} :: \mathrm{head} :: \varepsilon\rangle \to M(\mathrm{T}_{pc,\,\mathrm{obj}})$. Therefore for some r, $\bar{z}$, we have $(M(\mathrm{T}_{pc,\,\mathrm{obj}}), \mathrm{Context}(p-1)) \to (r, \bar{z})$, and $(\mathrm{PC}(\underline{\Xi}_{p-1}):\texttt{stack-1}, \mathrm{Context}(p-1)) \to (r, \bar{z})$. By the induction hypothesis, $\exists i'', e''.\ \mathrm{Creation}(\mathit{ref}) = (i'', e'') \wedge i'' \le i \wedge (e'', \mathrm{Context}(i'')) \to (r, \bar{z})$. But then $i'' = i'$ and $e'' = e'$, and indeed $r = u''$, $\bar{z} = \bar{x}''$.

Let $s = M(\mathrm{T}_{pc,\,\mathrm{obj}})$. Then $\{s \rhd_f M(\mathrm{T}_{pc,\,\mathrm{v}})\} \subseteq C$ and $(s, \mathrm{Context}(p-1)) \to (u'', \bar{x}'')$. From the preamble to this section (6.7.5.4), $(t', \mathrm{Context}(k+1)) \to (u, \bar{x})$, $(t'', \mathrm{Context}(k+1)) \to (u'', \bar{x}'')$, and $\{t'' \rhd_f t'\} \subseteq C$. The instance convergence property gives $(M(\mathrm{T}_{pc,\,\mathrm{v}}), \mathrm{Context}(p-1)) \to (u, \bar{x})$. The `putfield` constraints show $\mathrm{PC}(\underline{\Xi}_{p-1}):\texttt{stack-0} \to \mathrm{S}_{pc}\langle \mathrm{head} :: \varepsilon\rangle$ and $M(\mathrm{S}_{pc})\langle \mathrm{head} :: \varepsilon\rangle \to M(\mathrm{T}_{pc,\,\mathrm{v}})$. Putting these together gives

$$\mathrm{PC}(\underline{\Xi}_{p-1}):\texttt{stack-0} \to (u, \bar{x})$$

Case: $\underline{\Xi}_{p-1} \Rightarrow \underline{\Xi}_p$ is justified by the rule for spontaneous exception throw.

Then $(\underline{\Xi}_p, \mathrm{PC}(\underline{\Xi}_p):\texttt{exn}) \twoheadrightarrow \mathit{ref}$ and $(\underline{\Xi}_p, \mathrm{PC}(\underline{\Xi}_p):\texttt{exn}.f) \twoheadrightarrow v$, giving $\mathrm{Creation}(v) = (p, \mathrm{PC}(\underline{\Xi}_p):\texttt{exn}.f)$ and $\mathrm{Creation}(\mathit{ref}) = (p, \mathrm{PC}(\underline{\Xi}_p):\texttt{exn})$ by definition.

It remains to be shown that $(\mathrm{PC}(\underline{\Xi}_p):\texttt{exn}.f, \mathrm{Context}(p)) \to (u, \bar{x})$. From above, we have $\mathrm{Creation}(\mathit{ref}) = (i', e') \wedge i' \le k+1 \wedge (e', \mathrm{Context}(i')) \to (u'', \bar{x}'')$ for some i', e'. But because Creation is a function (Lemma 6-12), we have $i' = p$ and $e' = \mathrm{PC}(\underline{\Xi}_p):\texttt{exn}$, giving $(\mathrm{PC}(\underline{\Xi}_p):\texttt{exn}, \mathrm{Context}(p)) \to (u'', \bar{x}'')$. Therefore $(M(\mathrm{W}_{pc}), \mathrm{Context}(p)) \to (u'', \bar{x}'')$.

The initial constraints require of N:

$$\{ \mathrm{Err} \lessdot_{\mathrm{err}\text{-}pc} \mathrm{W}_{pc}, \mathrm{W}_{pc} \lessdot_{\mathrm{exn}\text{-}pc} \mathrm{X}_{pc} \} \cup \{ \mathrm{N}_{classID} \lessdot_{\mathrm{err}\text{-}classID} \mathrm{Err} \} \cup \{ \mathrm{N}_{classID} \rhd_f \mathrm{N}_{classID,f} \}$$

Therefore for some some s', $\{M(\mathrm{W}_{pc}) \rhd_f s'\} \subseteq C$. Clearly $\mathrm{PC}(\underline{\Xi}_p):\texttt{exn}.f \to \mathrm{W}_{pc}\langle f :: \varepsilon\rangle$ and $M(\mathrm{W}_{pc})\langle f :: \varepsilon\rangle \to s'$. Thus all that remains to be proved is $(s', \mathrm{Context}(p)) \to (u, \bar{x})$.

To recap, I have $\{M(\mathrm{W}_{pc}) \rhd_f s'\} \subseteq C$, $(M(\mathrm{W}_{pc}), \mathrm{Context}(p)) \to (u'', \bar{x}'')$, $(t', \mathrm{Context}(k+1)) \to (u, \bar{x})$, $(t'', \mathrm{Context}(k+1)) \to (u'', \bar{x}'')$, and $\{t'' \rhd_f t'\} \subseteq C$. Now

I can invoke the instance convergence property (Lemma 6-8) to obtain $(s', \text{Context}(p)) \to (u, \bar{x})$, as required.

6.7.5.5 Static Field Expressions

Suppose e is of the form $\text{PC}(\Xi_{k+1}) : \mathit{staticField}$. Then the rules for expression evaluation require $v = \text{Globals}(\Xi_{k+1})(\mathit{staticField})$. We also have the assumption $(\text{PC}(\Xi_{k+1}) : \mathit{staticField}, \text{Context}(k+1)) \to (u, \bar{x})$, implying for some t, $\text{PC}(\Xi_{k+1}) : \mathit{staticField} \to \text{G}_{\text{PC}(\Xi_{k+1})}\langle \mathit{staticField} :: \varepsilon \rangle$, $M(\text{G}_{\text{PC}(\Xi_{k+1})})\langle \mathit{staticField} :: \varepsilon \rangle \to t$, and $(t, \text{Context}(k+1)) \to (u, \bar{x})$.

We have already proven that $(M(\text{G}_{\text{PC}(\Xi_{k+1})}), \text{Context}(k+1)) \to (M(\text{G}_{(\text{Main}, 0)}), \varepsilon)$. Then by the component propagation property,

$$\exists v'.\ (t, \text{Context}(k+1)) \to (v', \varepsilon) \wedge \{M(\text{G}_{(\text{Main}, 0)}) \rhd_{\mathit{staticField}} v'\} \subseteq C$$

This implies $u = v'$ and $\bar{x} = \varepsilon$.

Let p be defined as

$$p = \min \{i \mid v = \text{Globals}(\Xi_i)(\mathit{staticField})\}$$

Clearly $0 \le p \le k+1$.

If $p = 0$ then, by the definition of Creation and the initial state Ξ_0, $\text{Creation}(v) = ((\text{Main}, 0) : \mathit{staticField})$. Now $((\text{Main}, 0) : \mathit{staticField}) \to M(\text{G}_{(\text{Main}, 0)})\langle \mathit{staticField} :: \varepsilon \rangle$, $M(\text{G}_{(\text{Main}, 0)})\langle \mathit{staticField} :: \varepsilon \rangle \to v'$ and $(v', \varepsilon) \to (v', \varepsilon)$; therefore, as required,

$$((\text{Main}, 0) : \mathit{staticField}, \text{Context}(0)) \to (v', \varepsilon)$$

Suppose $p > 0$. Then $\text{Globals}(\Xi_{p-1})(\mathit{staticField}) \neq v$. The only transition which can change the mapping of $\mathcal{G}$ is the execution of a `putstatic` *staticField* instruction. The rule for that instruction requires $\Xi_{p-1} = [\text{pc: } pc, \text{wstack: } v :: \mathcal{S}, \rho]$ for some $pc, \rho, \mathcal{S}$. Therefore $(\Xi_{p-1}, pc : \texttt{stack-0}) \twoheadrightarrow v$.

This instruction induces the constraints

$$\{ \text{S}_{pc} \rhd_{\text{tail}} \text{S}'_{pc}, \text{S}_{pc} \rhd_{\text{head}} \text{T}_{pc,\text{v}}, \text{G}_{pc} \rhd_{\mathit{fieldID}} \text{T}_{pc,\text{v}} \}$$

Therefore $pc : \texttt{stack-0} \to \text{S}_{pc}\langle \text{head} :: \varepsilon \rangle$ and $M(\text{S}_{pc})\langle \text{head} :: \varepsilon \rangle \to M(\text{T}_{pc,\text{v}})$.

Applying the induction hypothesis gives $(M(\text{G}_{pc}), \text{Context}(p-1)) \to (M(\text{G}_{(\text{Main}, 0)}), \varepsilon)$. Then applying the component propagation property with $\{M(\text{G}_{pc}) \rhd_{\mathit{staticField}} M(\text{T}_{pc,\text{v}})\} \subseteq C$ gives

$$\exists v''.\ (M(\text{T}_{pc,\text{v}}), \text{Context}(p-1)) \to (v'', \varepsilon) \wedge \{M(\text{G}_{(\text{Main}, 0)}) \rhd_{\mathit{staticField}} v''\} \subseteq C$$

Therefore $v'' = v'$. Combining the above gives $(pc : \texttt{stack-0}, \text{Context}(p-1)) \to (v', \varepsilon)$. Now we appeal to the induction hypothesis at $p-1$ to directly obtain the required result.

6.7.5.6 Cases For Simple Expressions

The remaining cases prove the induction result for the simple expressions of the form `stack-`m, `local-`m and `exn`, for each form of transition. The rest of this chapter proves those cases, ordered by the form of the transition. For most instructions, the strategy is to map the expression evaluated after transition to an expression evaulated before transition, and show that their values are the same and their types are suitably related.

6.7.5.7 Reduction Function

For each case, I define a partial function R : BExpRoot ↦ BExpRoot satisfying the following conditions:

$$\forall exp, v.\ (\Xi_{k+1}, \mathrm{PC}(\Xi_{k+1}) : exp) \twoheadrightarrow v \Rightarrow (\Xi_k, \mathrm{PC}(\Xi_k) : \mathrm{R}(exp)) \twoheadrightarrow v$$

$$\forall exp, u, \bar{x}.$$
$$(\mathrm{PC}(\Xi_{k+1}) : exp, \mathrm{Context}(k+1)) \rightarrow (u, \bar{x}) \Rightarrow (\mathrm{PC}(\Xi_k) : exp, \mathrm{Context}(k)) \rightarrow (u, \bar{x})$$

For those exp on which R is defined, we immediately obtain $(\Xi_k, \mathrm{PC}(\Xi_k) : \mathrm{R}(exp)) \twoheadrightarrow v$ and $(\mathrm{PC}(\Xi_k) : exp, \mathrm{Context}(k)) \rightarrow (u, \bar{x})$; the required result follows immediately from the induction hypothesis.

In all the cases, we set $pc = \mathrm{PC}(\Xi_k)$.

6.7.5.8 Succession Lemma

Lemma 6-22. This lemma is very helpful for showing the preservation of types during normal control flow. It states that if an instruction does not modify the value of a stack variable or local variable (implying that it only transfers control within the current method), then the type is preserved.

$$\forall exp, j, S', L'.\ (\mathrm{PC}(\Xi_{k+1}) : exp, \mathrm{PC}(\Xi_j)\text{-}\mathrm{PC}(\Xi_{k+1}) :: \mathrm{Context}(j)) \rightarrow (u, \bar{x})$$
$$\wedge\ exp \neq \texttt{exn} \wedge \mathrm{Succ}(\mathrm{PC}(\Xi_j), \mathrm{PC}(\Xi_{k+1}), S', L') \subseteq N$$
$$\Rightarrow \exists t, \bar{c}, s, t'.\ \mathrm{PC}(\Xi_{k+1}) : exp \rightarrow t\langle\bar{c}\rangle \wedge s\langle\bar{c}\rangle \rightarrow t' \wedge (t', \mathrm{Context}(j)) \rightarrow (u, \bar{x})$$
$$\wedge\ s = M(\mathrm{F}(exp, S', L'))$$

Here F is defined as follows:

$$\mathrm{F}(\texttt{stack-}m, S', L') = S'_{pc}$$
$$\mathrm{F}(\texttt{local-}m, S', L') = L'_{pc}$$

Note that F is not defined for the expression `exn`; the expression exp can only be `exn` when the abstract machine is in exception-handling mode.

Proof: By definition, $(\mathrm{PC}(\Xi_{k+1}) : exp, \mathrm{PC}(\Xi_j)\text{-}\mathrm{PC}(\Xi_{k+1}) :: \mathrm{Context}(k)) \rightarrow (u, \bar{x})$ requires $\mathrm{PC}(\Xi_{k+1}) : exp \rightarrow t\langle\bar{c}\rangle$, $M(t)\langle\bar{c}\rangle \rightarrow t''$ and $(t'', \mathrm{PC}(\Xi_j)\text{-}\mathrm{PC}(\Xi_{k+1}) :: \mathrm{Context}(k)) \rightarrow (u, \bar{x})$ for some $t, \bar{c}, t''$.

Consider the two cases for *exp*; we show that in both cases, $\{t \lessdot_{\mathrm{PC}(\Xi_j)\text{-}\mathrm{PC}(\Xi_{k+1})} s\} \subseteq N$ where $s = M(\mathrm{F}(exp, S'_{pc}, L'_{pc}))$.

Case: $exp = \texttt{stack-}m$. Then $t = \mathrm{S}_{\mathrm{PC}(\Xi_{k+1})}$ and $s = M(S'_{pc})$. We have $\{\mathrm{S}_{\mathrm{PC}(\Xi_{k+1})} \lessdot_{\mathrm{PC}(\Xi_j)\text{-}\mathrm{PC}(\Xi_{k+1})} S'\} \subseteq \mathrm{Succ}(\mathrm{PC}(\Xi_j), \mathrm{PC}(\Xi_{k+1}), S', L')$

Case: $exp = \texttt{local-}m$. Then $t = \mathrm{L}_{pc}$ and $s = M(L'_{pc})$. We have $\{\mathrm{L}_{\mathrm{PC}(\Xi_{k+1})} \lessdot_{\mathrm{PC}(\Xi_j)\text{-}\mathrm{PC}(\Xi_{k+1})} L'\} \subseteq \mathrm{Succ}(\mathrm{PC}(\Xi_j), \mathrm{PC}(\Xi_{k+1}), S', L')$

Now by the instance propagation property (Section 6-10), there exists t' such that $s\langle \bar{c}\rangle \rightarrow t'$ and $\{t'' \lessdot_{\mathrm{PC}(\Xi_j)\text{-}\mathrm{PC}(\Xi_{k+1})} t'\} \subseteq C$. This implies $(t', \mathrm{Context}(k)) \rightarrow (u, \bar{x})$, as required. ■

6.7.5.9 Step: `load` rule

The rule for `load` gives

$$\mathrm{Instruction}(pc) = \texttt{load}\ index$$
$$\Xi_k = [\text{pc: } pc, \text{wstack: } \underline{S}, \text{locals: } \underline{L}, \rho]$$
$$\Xi_{k+1} = [\text{pc: } pc+1, \text{wstack: } \underline{L}(index) :: \underline{S}, \text{locals: } \underline{L}, \rho].$$

The function R is:

$\mathrm{R}(\texttt{stack-}m)$	$= \texttt{stack-}(m-1)$	$m > 0$
$\mathrm{R}(\texttt{stack-0})$	$= \texttt{local-}index$	
$\mathrm{R}(\texttt{local-}n)$	$= \texttt{local-}n$	

Now consider the different cases for *exp*. Because R is defined for all $\texttt{stack-}m$ and $\texttt{local-}n$, this proof suffices to guarantee the induction hypothesis. Note that *exp* cannot be `exn` since the machine is in state RUNNING.

N contains the constraints

$$\{ \mathrm{L}_{pc} \rhd_{index} \mathrm{T}_{pc,\mathrm{v}}, \mathrm{S}'_{pc} \rhd_{\mathrm{tail}} \mathrm{S}_{pc}, \mathrm{S}'_{pc} \rhd_{\mathrm{head}} \mathrm{T}_{pc,\mathrm{v}} \} \cup \mathrm{Succ}(pc, pc{+}1, \mathrm{S}'_{pc}, \mathrm{L}_{pc})$$

We also have $\mathrm{Context}(k+1) = pc\text{-}(pc+1) :: \mathrm{Context}(k)$ and therefore $(\mathrm{PC}(\Xi_{k+1}) : exp, pc\text{-}(pc+1) :: \mathrm{Context}(k)) \rightarrow (u, \bar{x})$. This implies that

$$\exists t, \bar{c}, s, t'.\ \mathrm{PC}(\Xi_{k+1}) : exp \rightarrow t\langle \bar{c}\rangle \wedge s\langle \bar{c}\rangle \rightarrow t' \wedge (t', \mathrm{Context}(k)) \rightarrow (u, \bar{x})$$
$$\wedge\ s = M(\mathrm{F}(exp, \mathrm{S}'_{pc}, \mathrm{L}_{pc}, \mathrm{G}_{pc}))$$

Case: $exp = \texttt{stack-}m$, $m > 0$. Then $\mathrm{R}(exp) = \texttt{stack-}(m-1)$.

The evaluation rules show $\underline{L}(index) :: \underline{S}$ is of the form $v_0 :: \ldots :: v_m :: \underline{S}'$ where $v_m = v$. Therefore $\underline{S} = v_1 :: \ldots :: v_m :: \underline{S}'$ and $(\Xi_k, pc : \texttt{stack-}(m-1)) \twoheadrightarrow v_m = v$, as required.

In this case we apply the succession lemma (6-22) with $t = \mathrm{S}_{pc+1}$ and $\bar{c} = \mathrm{tail} :: \ldots :: \mathrm{tail} :: \mathrm{head} :: \varepsilon$, with m occurrences of "tail". Also, $s = M(\mathrm{S}'_{pc})$. Therefore $M(\mathrm{S}'_{pc})\langle \bar{c}\rangle \rightarrow t'$ where $(t', \mathrm{Context}(k)) \rightarrow (u, \bar{x})$; this implies $M(\mathrm{S}_{pc})\langle \bar{c}'\rangle \rightarrow t'$, where $\bar{c} = \mathrm{tail} :: \bar{c}'$. The sequence $\bar{c}'$ has $m-1$ tails, therefore

$\mathrm{PC}(\underline{\Xi}_k) : \texttt{stack-}(m-1) \rightarrow M(\mathrm{S}_{pc+1})\langle \bar{c}' \rangle$. All together then, $(\mathrm{PC}(\underline{\Xi}_k) : \texttt{stack-}(m-1), \mathrm{Context}(k)) \rightarrow (u, \bar{x})$ as required.

Case: $exp = \texttt{stack-0}$. Then $\mathrm{R}(exp) = \texttt{local-}index$.

The evaluation rules show $\underline{\mathcal{L}}(index) :: \underline{\mathcal{S}}$ is of the form $v_0 :: \ldots :: v_m :: \underline{\mathcal{S}}'$ where $v_0 = v = \underline{\mathcal{L}}(index)$. Therefore $(\underline{\Xi}_k, pc : \texttt{local-}index) \twoheadrightarrow \underline{\mathcal{L}}(index) = v$, as required.

In this case $t = \mathrm{S}_{pc+1}$ and $\bar{c} = \mathrm{head} :: \varepsilon$. Also, $s = M(\mathrm{S}'_{pc})$. Therefore $M(\mathrm{S}'_{pc})\langle \bar{c} \rangle \rightarrow M(\mathrm{T}_{pc,\mathrm{v}})$, i.e. $t' = M(\mathrm{T}_{pc,\mathrm{v}})$. This, plus the constraints in N, implies $M(\mathrm{L}_{pc})\langle index \rangle \rightarrow t'$. Also, $\mathrm{PC}(\underline{\Xi}_k) : \texttt{local-}index \rightarrow M(\mathrm{L}_{pc})\langle index :: \varepsilon \rangle$; all together then, $(\mathrm{PC}(\underline{\Xi}_k) : \texttt{local-}index, \mathrm{Context}(k)) \rightarrow (u, \bar{x})$ as required.

Case: $exp = \texttt{local-}n$. Then $\mathrm{R}(exp) = \texttt{local-}n$.

The evaluation rules show $\underline{\mathcal{L}}(n) = v$. Therefore $(\underline{\Xi}_k, pc : \texttt{local-}n) \twoheadrightarrow \underline{\mathcal{L}}(n) = v$, as required.

In this case $t = \mathrm{L}_{pc+1}$ and $\bar{c} = n :: \varepsilon$. Also, $s = M(\mathrm{L}_{pc})$. Therefore $M(\mathrm{L}_{pc})\langle \bar{c} \rangle \rightarrow t'$. Also, $\mathrm{PC}(\underline{\Xi}_k) : \texttt{local-}n \rightarrow M(\mathrm{L}_{pc})\langle \bar{c} \rangle$; all together then, $(\mathrm{PC}(\underline{\Xi}_k) : \texttt{local-}n, \mathrm{Context}(k)) \rightarrow (u, \bar{x})$ as required.

6.7.5.10 Induction Step: `store` rule

The rule for `store` gives

$$\mathrm{Instruction}(pc) = \texttt{store}\ index$$
$$\underline{\Xi}_k = [\text{pc: } pc, \text{ wstack: } v' :: \underline{\mathcal{S}}, \text{ locals: } \underline{\mathcal{L}}, \rho]$$
$$\underline{\Xi}_{k+1} = [\text{pc: } pc+1, \text{ wstack: } \underline{\mathcal{S}}, \text{ locals: } \underline{\mathcal{L}}[index : v'], \rho].$$

The function R is:

$R(\texttt{stack-}m)$	$= \texttt{stack-}(m+1)$	
$R(\texttt{local-}index)$	$= \texttt{stack-0}$	
$R(\texttt{local-}n)$	$= \texttt{local-}n$	$n \neq index$

Now consider the different cases for *exp*. Because R is defined for all BExpRoots other than `exn`, this proof suffices to guarantee the induction hypothesis.

N contains the constraints

$$\{ \mathrm{S}_{pc} \triangleright_{\mathrm{tail}} \mathrm{S}'_{pc}, \mathrm{S}_{pc} \triangleright_{\mathrm{head}} \mathrm{T}_{pc,\mathrm{v}}, \mathrm{L}'_{pc} \triangleright_{index} \mathrm{T}_{pc,\mathrm{v}} \} \cup$$
$$\{ \mathrm{L}'_{pc} \triangleright_i \mathrm{T}_{pc,\mathrm{i}} \mid i \in \mathrm{LocalNames}(pc) \wedge i \neq index \} \cup$$
$$\{ \mathrm{L}_{pc} \triangleright_i \mathrm{T}_{pc,\mathrm{i}} \mid i \in \mathrm{LocalNames}(pc) \wedge i \neq index \} \cup \mathrm{Succ}(pc, pc+1, \mathrm{S}'_{pc}, \mathrm{L}'_{pc})$$

We also have $\mathrm{Context}(k+1) = pc\text{-}(pc+1) :: \mathrm{Context}(k)$ and therefore $(\mathrm{PC}(\underline{\Xi}_{k+1}) : exp, pc\text{-}(pc+1) :: \mathrm{Context}(k)) \rightarrow (u, \bar{x})$. This implies that

$$\exists t, \bar{c}, s, t'.\ \mathrm{PC}(\underline{\Xi}_{k+1}) : exp \rightarrow t\langle \bar{c} \rangle \wedge s\langle \bar{c} \rangle \rightarrow t' \wedge (t', \mathrm{Context}(k)) \rightarrow (u, \bar{x})$$
$$\wedge\ s = M(\mathrm{F}(exp, \mathrm{S}'_{pc}, \mathrm{L}_{pc}, \mathrm{G}_{pc}))$$

Case: $exp = \texttt{stack-}m$. Then $\mathrm{R}(exp) = \texttt{stack-}(m+1)$.

The evaluation rules show $\underline{\mathcal{S}}$ is of the form $v_0 :: ... :: v_m :: \underline{\mathcal{S}}'$ where $v_m = v$. Therefore $\mathrm{Stack}(\Xi_k) = v' :: v_0 :: ... :: v_m :: \underline{\mathcal{S}}'$ and $(\Xi_k, pc : \texttt{stack-}(m+1)) \twoheadrightarrow v_m = v$, as required.

In this case I apply the succession lemma (6-22) with $t = \mathrm{S}_{pc+1}$ and $\bar{c} = \text{tail} :: ... :: \text{tail} :: \text{head} :: \varepsilon$, with m occurrences of "tail". Also, $s = M(\mathrm{S'}_{pc})$. Therefore $M(\mathrm{S'}_{pc})\langle\bar{c}\rangle \to t'$; this implies $M(\mathrm{S}_{pc})\langle\bar{c}'\rangle \to t'$, where $\bar{c}' = \text{tail} :: \bar{c}$. The sequence $\bar{c}$ has $m+1$ tails, therefore $\mathrm{PC}(\Xi_k) : \texttt{stack-}(m+1) \to M(\mathrm{S}_{pc+1})\langle\bar{c}'\rangle$. All together then, $(\mathrm{PC}(\Xi_k) : \texttt{stack-}(m+1), \mathrm{Context}(k)) \to (u, \bar{x})$ as required.

Case: $exp = \texttt{local-}index$. Then $\mathrm{R}(exp) = \texttt{stack-0}$.

The evaluation rules show $v' = v$. Therefore $(\Xi_k, pc : \texttt{stack-0}) \twoheadrightarrow v' = v$, as required.

I apply the succession lemma (6-22) with $t = \mathrm{L}_{pc+1}$ and $\bar{c} = index :: \varepsilon$. Also, $s = M(\mathrm{L'}_{pc})$. Therefore $M(\mathrm{L'}_{pc})\langle\bar{c}\rangle \to t'$, i.e. $t' = M(\mathrm{T}_{pc,\,\mathrm{v}})$. This, plus the constraints in N, implies $M(\mathrm{S}_{pc})\langle head :: \varepsilon\rangle \to t'$. Also, $\mathrm{PC}(\Xi_k) : pc : \texttt{stack-0} \to M(\mathrm{S}_{pc})\langle head :: \varepsilon\rangle$; all together then, $(\mathrm{PC}(\Xi_k) : pc : \texttt{stack-0}, \mathrm{Context}(k)) \to (u, \bar{x})$ as required.

Case: $exp = \texttt{local-}n$, where $n \neq index$. Then $\mathrm{R}(exp) = \texttt{local-}n$.

The evaluation rules show $\underline{\mathcal{L}}(n) = v$. Therefore $(\Xi_k, pc : \texttt{local-}n) \twoheadrightarrow \underline{\mathcal{L}}(n) = v$, as required.

In this case $t = \mathrm{L}_{pc+1}$ and $\bar{c} = n :: \varepsilon$. Also, $s = M(\mathrm{L'}_{pc})$. Therefore $M(\mathrm{L'}_{pc})\langle\bar{c}\rangle \to t'$ and $t' = M(\mathrm{T}_{pc,\,n})$. This, plus the constraints in N, implies $M(\mathrm{L}_{pc})\langle n :: \varepsilon\rangle \to t'$ Also, $\mathrm{PC}(\Xi_k) : \texttt{local-}n \to M(\mathrm{L}_{pc})\langle n :: \varepsilon\rangle$; all together then, $(\mathrm{PC}(\Xi_k) : \texttt{local-}n, \mathrm{Context}(k)) \to (u, \bar{x})$ as required.

6.7.5.11 Induction Step: `new` rule

The rule for `new` gives

$$\mathrm{Instruction}(pc) = \texttt{new}\ classID$$
$$\Xi_k = [\text{pc: } pc,\ \text{wstack: } \underline{\mathcal{S}},\ \text{locals: } \underline{\mathcal{L}},\ \rho]$$
$$\Xi_{k+1} = [\text{pc: } pc+1,\ \text{wstack: } ref :: \underline{\mathcal{S}},\ \text{locals: } \underline{\mathcal{L}},\ \rho]$$

The function R is:

$R(\texttt{stack-}m)$	$= \texttt{stack-}(m-1)$	$m > 0$
$R(\texttt{stack-0})$	is undefined	
$R(\texttt{local-}n)$	$= \texttt{local-}n$	

For the expressions on which R is defined, the proof of R's correctness is identical to the cases for `load`, and is not repeated here.

For $exp = \texttt{stack-0}$, $\mathrm{Creation}(v) = (k+1, (pc+1) : \texttt{stack-0})$ by the definition of Creation; thus the induction result is trivially satisfied.

6.7.5.12 Induction Step: `aconst_null` **rule**
The proof for this case is the same as for the `new` rule.

6.7.5.13 Induction Step: `bipush` **rule**
The proof for this case is the same as for the `new` rule.

6.7.5.14 Induction Step: rule for spontaneous exception throw
The rule for spontaneous exception throw gives

$$\begin{aligned}&\mathit{classID} \in \text{ErrorClassIDs}\\ &\underline{\Xi}_k = [\text{mode: RUNNING, pc: } pc\text{, wstack: } \underline{\mathcal{S}}\text{, locals: } \underline{\mathcal{L}}, \rho]\\ &\underline{\Xi}_{k+1} = [\text{mode: THROWING, pc: } pc\text{, wstack: } \mathit{ref} :: \varepsilon\text{, locals: } \underline{\mathcal{L}}, \rho]\,.\end{aligned}$$

Furthermore $\text{Context}(k) = \text{Context}(k+1)$.

The function R is:

$R(\texttt{stack-}m)$	is undefined
$R(\texttt{exn})$	is undefined
$R(\texttt{local-}n)$	$= \texttt{local-}n$

Case: $\mathit{exp} = \texttt{stack-}m$.

This case cannot occur because stack expressions do not evaluate to anything in the THROWING state.

Case: $\mathit{exp} = \texttt{local-}n$. Then $\text{R}(\mathit{exp}) = \texttt{local-}n$.

The evaluation rules show $\underline{\mathcal{L}}(n) = v$. Therefore $(\underline{\Xi}_k, pc : \texttt{local-}n) \dashrightarrow \underline{\mathcal{L}}(n) = v$. Furthermore, since $\text{PC}(\underline{\Xi}_k) = pc = \text{PC}(\underline{\Xi}_{k+1})$ and $\text{Context}(k) = \text{Context}(k+1)$, $(\text{PC}(\underline{\Xi}_k) : \texttt{local-}n, \text{Context}(k)) \to (u, \bar{x})$. The result then follows from the induction hypothesis.

Case: $\mathit{exp} = \texttt{exn}$.

R is undefined for $pc : \texttt{exn}$. However $(\underline{\Xi}_{k+1}, pc : \texttt{exn}) \dashrightarrow v$ implies $\text{Creation}(v) = (k+1, pc : \texttt{exn})$; thus the induction result is trivially satisfied.

6.7.5.15 Induction Step: `invokestatic` **rule**
The rule for `invokestatic` gives

$$\begin{aligned}&\text{Instruction}(pc) = \texttt{invokestatic}\ \mathit{methodImpl}\\ &\underline{\Xi}_k = [\text{pc: } pc\text{, wstack: } v_1 :: v_0 :: \underline{\mathcal{S}}\text{, locals: } \underline{\mathcal{L}}\text{, mstack: } \mathcal{F}, \rho]\\ &\underline{\Xi}_{k+1} = [\text{pc: } pc'\text{, wstack: } \varepsilon\text{, locals: } [0\text{: } v_0,\ 1\text{: } v_1]\text{, mstack: } (pc, \underline{\mathcal{S}}, \underline{\mathcal{L}}) :: \mathcal{F}, \rho]\\ &pc' = (\mathit{methodImpl}, 0)\end{aligned}$$

Furthermore, $\text{Context}(k+1) = pc :: \text{Context}(k)$. The induced constraints include

$\{ \mathrm{S}_{pc} \rhd_{\mathrm{tail}} \mathrm{T}_{pc,\mathrm{t1}}, \mathrm{S}_{pc} \rhd_{\mathrm{head}} \mathrm{T}_{pc,\mathrm{v1}}, \mathrm{T}_{pc,\mathrm{t1}} \rhd_{\mathrm{tail}} \mathrm{T}_{pc,\mathrm{t2}}, \mathrm{T}_{pc,\mathrm{t1}} \rhd_{\mathrm{head}} \mathrm{T}_{pc,\mathrm{v0}}, \mathrm{M}_{methodImpl} \lessdot_{pc} \mathrm{T}_{pc,\mathrm{m}}, \mathrm{T}_{pc,\mathrm{m}} \rhd_{\text{param-0}} \mathrm{T}_{pc,\mathrm{v0}}, \mathrm{T}_{pc,\mathrm{m}} \rhd_{\text{param-1}} \mathrm{T}_{pc,\mathrm{v1}} \}$

The initial constraints also contain

$\{\mathrm{M}_{methodImpl} \rhd_{\text{param-0}} \mathrm{T}_{methodImpl,\mathrm{p0}}, \mathrm{M}_{methodImpl} \rhd_{\text{param-1}} \mathrm{T}_{methodImpl,\mathrm{p1}}, \mathrm{L}_{pc'} \rhd_0 \mathrm{T}_{methodImpl,\mathrm{p0}}, \mathrm{L}_{pc'} \rhd_1 \mathrm{T}_{methodImpl,\mathrm{p1}}\}$

The function R is:

$R(\texttt{local-}n)$	$= \texttt{stack-}(1-n)$	$0 \le n \le 1$
$R(exp)$	is undefined	otherwise

Case: $exp = \texttt{stack-}m$. This case cannot occur because $\mathrm{WStack}(\Xi_{k+1}) = \varepsilon$.

Case: $exp = \texttt{local-}n$. Then $\mathrm{R}(exp) = \texttt{stack-}(1-n)$.

In this case n must be 0 or 1 and $v = v_n$. Then the evaluation rules show that $(\Xi_k, pc : \texttt{stack-}(1-n)) \twoheadrightarrow v_n = v$.

Now, $(pc' : \texttt{local-}n, \mathrm{Context}(k+1)) \to (u, \bar{x})$ implies that $(M(\mathrm{T}_{methodImpl,\mathrm{p}n}), pc :: \mathrm{Context}(k)) \to (u, \bar{x})$. Combining this with

$\{M(\mathrm{M}_{methodImpl}) \rhd_{\text{param-}n} M(\mathrm{T}_{methodImpl,\mathrm{p}n}), M(\mathrm{M}_{methodImpl}) \lessdot_{pc} M(\mathrm{T}_{pc,\mathrm{m}}), M(\mathrm{T}_{pc,\mathrm{m}}) \rhd_{\text{param-0}} M(\mathrm{T}_{pc,\mathrm{v}n})\} \subseteq C$

gives $\{M(\mathrm{T}_{methodImpl,\mathrm{p}n}) \lessdot_{pc} M(\mathrm{T}_{pc,\mathrm{v}n})\} \subseteq C$. Therefore $(M(\mathrm{T}_{pc,\mathrm{v}n}), \mathrm{Context}(k)) \to (u, \bar{x})$.

If $n = 0$ then $pc : \texttt{stack-}(1-n) \to \mathrm{S}_{pc}\langle \mathrm{tail} :: \mathrm{head} :: \varepsilon\rangle$ and $M(\mathrm{S}_{pc})\langle \mathrm{tail} :: \mathrm{head} :: \varepsilon\rangle \to M(\mathrm{T}_{pc,\mathrm{v0}})$. Otherwise $n = 1$, $pc : \texttt{stack-}(1-n) \to \mathrm{S}_{pc}\langle \mathrm{head} :: \varepsilon\rangle$ and $M(\mathrm{S}_{pc})\langle \mathrm{head} :: \varepsilon\rangle \to M(\mathrm{T}_{pc,\mathrm{v1}})$. Either way, $(pc : \texttt{stack-}(1-n), \mathrm{Context}(k)) \to (u, \bar{x})$. The result then follows directly from the induction hypothesis.

6.7.5.16 Induction Step: `invokevirtual` rule

The rule for `invokevirtual` gives

$\mathrm{Instruction}(pc) = \texttt{invokevirtual}\ methodID$
$\Xi_k = [\text{pc: } pc, \text{wstack: } v_1 :: v_0 :: \mathcal{S}, \text{locals: } \mathcal{L}, \text{mstack: } \mathcal{Q}, \rho]$
$\Xi_{k+1} = [\text{pc: } pc', \text{wstack: } \varepsilon, \text{locals: } [0{:}\ v_0, 1{:}\ v_1], \text{mstack: } (pc, \mathcal{S}, \mathcal{L}) :: \mathcal{Q}, \rho]$

where $pc' = (methodImpl, 0)$.

The induced constraints include

$\{ \mathrm{S}_{pc} \rhd_{\mathrm{tail}} \mathrm{T}_{pc,\mathrm{t1}}, \mathrm{S}_{pc} \rhd_{\mathrm{head}} \mathrm{T}_{pc,\mathrm{v1}}, \mathrm{T}_{pc,\mathrm{t1}} \rhd_{\mathrm{tail}} \mathrm{T}_{pc,\mathrm{t2}}, \mathrm{T}_{pc,\mathrm{t1}} \rhd_{\mathrm{head}} \mathrm{T}_{pc,\mathrm{v0}}, \mathrm{T}_{pc,\mathrm{v0}} \rhd_{methodID} \mathrm{T}_{pc,\mathrm{m}}, \mathrm{S}'_{pc} \rhd_{\mathrm{tail}} \mathrm{T}_{pc,\mathrm{t2}}, \mathrm{S}'_{pc} \rhd_{\mathrm{head}} \mathrm{T}_{pc,\mathrm{r}}, \mathrm{T}_{pc,\mathrm{m}} \rhd_{\text{param-0}} \mathrm{T}_{pc,\mathrm{v0}}, \mathrm{T}_{pc,\mathrm{m}} \rhd_{\text{param-1}} \mathrm{T}_{pc,\mathrm{v1}} \}$

The initial constraints also contain

$$\{\mathrm{M}_{methodImpl} \triangleright_{\text{param-0}} \mathrm{T}_{methodImpl,\,\mathrm{p0}}, \mathrm{M}_{methodImpl} \triangleright_{\text{param-1}} \mathrm{T}_{methodImpl,\,\mathrm{p1}},$$
$$\mathrm{L}_{pc'} \triangleright_0 \mathrm{T}_{methodImpl,\,\mathrm{p0}}, \mathrm{L}_{pc'} \triangleright_1 \mathrm{T}_{methodImpl,\,\mathrm{p1}}\}$$

The function R is:

$R(\texttt{local-}n)$	$= \texttt{stack-}(1-n)$	$0 \le n \le 1$
$R(exp)$	is undefined	otherwise

Case: $exp = \texttt{stack-}m$. This case cannot occur because $\mathrm{WStack}(\underline{\Xi}_{k+1}) = \varepsilon$.

Case: $exp = \texttt{local-}n$. Then $\mathrm{R}(exp) = \texttt{stack-}(1-n)$.

In this case n must be 0 or 1 and $v = v_n$. Then the evaluation rules show that $(\underline{\Xi}_k, pc : \texttt{stack-}(1-n)) \rightsquigarrow v_n = v$.

Now, $(pc' : \texttt{local-}n, \mathrm{Context}(k+1)) \to (u, \bar{x})$ implies that $(M(\mathrm{T}_{methodImpl,\,\mathrm{p}n}), \mathrm{Context}(k+1)) \to (u, \bar{x})$. Apply the preservation of virtual call types lemma, setting $c = \text{param-}n$, $v = \mathrm{T}_{pc,\,\mathrm{v}n}$ and $v' = \mathrm{T}_{methodImpl,\,\mathrm{p}n}$, giving $(M(\mathrm{T}_{pc,\,\mathrm{v}n}), \mathrm{Context}(k)) \to (u, \bar{x})$.

If $n = 0$ then $pc : \texttt{stack-}(1-n) \to \mathrm{S}_{pc}\langle \text{tail} :: \text{head} :: \varepsilon\rangle$ and $M(\mathrm{S}_{pc})\langle \text{tail} :: \text{head} :: \varepsilon\rangle \to M(\mathrm{T}_{pc,\,\mathrm{v0}})$. Otherwise $n = 1$, $pc : \texttt{stack-}(1-n) \to \mathrm{S}_{pc}\langle \text{head} :: \varepsilon\rangle$ and $M(\mathrm{S}_{pc})\langle \text{head} :: \varepsilon\rangle \to M(\mathrm{T}_{pc,\,\mathrm{v1}})$. Either way, $(pc : \texttt{stack-}(1-n), \mathrm{Context}(k)) \to (u, \bar{x})$. The result then follows directly from the induction hypothesis.

6.7.5.17 Induction Step: `return` rule

The rule for `return` gives

$$\mathrm{Instruction}(pc) = \texttt{return}$$
$$\underline{\Xi}_k = [\text{pc: } pc, \text{ wstack: } v' :: \mathcal{S}, \text{ locals: } \mathcal{L}, \text{ mstack: } (pc'', \mathcal{S}', \mathcal{L}') :: \mathcal{Q}, \rho]$$
$$\underline{\Xi}_{k+1} = [\text{pc: } pc'' + 1, \text{ wstack: } v' :: \mathcal{S}', \text{ locals: } \mathcal{L}', \text{ mstack: } \mathcal{Q}, \rho]$$

Let $c = \mathrm{CallerState}(k)$ and $pc' = \mathrm{PC}(\underline{\Xi}_c)$. The transition $\underline{\Xi}_c \underline{\Rightarrow} \underline{\Xi}_{c+1}$ must be an application of `invokestatic` or `invokevirtual`, because only those rules extend $\mathcal{Q}$. Therefore $\mathrm{Instruction}(pc') = \texttt{invokevirtual}\ \ methodID$ or $\mathrm{Instruction}(pc') = \texttt{invokestatic}\ \ methodImpl$. In the latter case, $methodImpl = \mathrm{CodeLocMethod}(\mathrm{PC}(\underline{\Xi}_{c+1}))$; in the former case, define $methodImpl = \mathrm{CodeLocMethod}(\mathrm{PC}(\underline{\Xi}_{c+1}))$.

In either case, N contains the constraints

$$\{ \mathrm{S}_{pc'} \triangleright_{\text{tail}} \mathrm{T}_{pc',\mathrm{t1}}, \mathrm{S}_{pc'} \triangleright_{\text{head}} \mathrm{T}_{pc',\mathrm{v1}}, \mathrm{T}_{pc',\mathrm{t1}} \triangleright_{\text{tail}} \mathrm{T}_{pc',\mathrm{t2}}, \mathrm{T}_{pc',\mathrm{t1}} \triangleright_{\text{head}} \mathrm{T}_{pc',\mathrm{v0}},$$
$$\mathrm{S}'_{pc'} \triangleright_{\text{tail}} \mathrm{T}_{pc',\mathrm{t2}}, \mathrm{S}'_{pc'} \triangleright_{\text{head}} \mathrm{T}_{pc',\mathrm{r}} \} \cup \mathrm{MethodCall}(\mathrm{T}_{pc,\mathrm{m}}, \mathrm{T}_{pc,\mathrm{v0}}, \mathrm{T}_{pc,\mathrm{v1}}, \mathrm{G}_{pc}, \mathrm{W}_{pc},$$
$$\mathrm{T}_{pc,\mathrm{r}}) \cup \mathrm{Succ}(pc', pc' + 1, \mathrm{S}'_{pc'}, \mathrm{L}_{pc'})$$

Note also that $\text{Context}(k+1) = pc'\text{-}(pc'+1) :: \text{Context}(c)$.

By the lemma governing preservation of caller state (Lemma 6-15),
$\underline{\Xi}_c = [\text{pc: } pc'', \text{wstack: } v'_1 :: v'_0 :: \mathcal{S}', \text{locals: } \underline{\mathcal{L}}', \text{mstack: } \mathcal{J}, \rho]$. This implies $pc' = pc''$.

Case: $exp = \texttt{local-}n$ for some n.

Then $v = \underline{\mathcal{L}}'(n)$, and therefore $(\underline{\Xi}_c, pc' : exp) \Rrightarrow v$.

In this case I apply the succession lemma (6-22) at $j = c$ with $t = \text{L}_{pc'+1}$ and $\bar{c} = n :: \varepsilon$. Also, $s = M(\text{L}_{pc'})$. Therefore $M(\text{L}_{pc'})\langle\bar{c}\rangle \to t'$. Also, $\text{PC}(\underline{\Xi}_c) : \texttt{local-}n \to M(\text{L}_{pc'})\langle\bar{c}\rangle$; all together then, $(\text{PC}(\underline{\Xi}_c) : \texttt{local-}n, \text{Context}(c)) \to (u, \bar{x})$. Applying the induction hypothesis setting $i = c$ gives the required result.

Case: $exp = \texttt{stack-}m$ for some $m > 0$.

The evaluation rules show $v' :: \underline{\mathcal{S}}'$ is of the form $v_0 :: \ldots :: v_m :: \underline{\mathcal{S}}''$ where $v_m = v$. Therefore $\underline{\mathcal{S}}' = v_1 :: \ldots :: v_m :: \underline{\mathcal{S}}''$. Now
$\text{MStack}(\underline{\Xi}_c) = v'_1 :: v'_0 :: \mathcal{S}' = v'_1 :: v'_0 :: v_1 :: \ldots :: v_m :: \underline{\mathcal{S}}''$; therefore
$(\underline{\Xi}_c, pc' : \texttt{stack-}(m+1)) \Rrightarrow v_m = v$.

We apply the succession lemma (6-22) at $j = c$ with $t = \text{S}_{pc'+1}$ and and $\bar{c} = \text{tail} :: \ldots :: \text{tail} :: \text{head} :: \varepsilon$, with m occurrences of “tail”. Also, $s = M(\text{S}'_{pc'})$. Therefore $M(\text{S}'_{pc'})\langle\bar{c}\rangle \to t'$. This implies $M(\text{S}_{pc'})\langle\bar{c}'\rangle \to t'$, where $\bar{c}' = \text{tail} :: \bar{c}$. Therefore $\text{PC}(\underline{\Xi}_c) : \texttt{stack-}(m+1) \to M(\text{S}_{pc'})\langle\bar{c}'\rangle$. All together then, $(\text{PC}(\underline{\Xi}_c) : \texttt{stack-}(m+1), \text{Context}(c)) \to (u, \bar{x})$.

Applying the induction hypothesis setting $i = c$ gives the required result.

Case: $exp = \texttt{stack-0}$.

Then $v = v'$, and therefore $(\underline{\Xi}_k, pc : \texttt{stack-0}) \Rrightarrow v$. I will prove that $(pc : \texttt{stack-0}, \text{Context}(k)) \to (u, \bar{x})$; the correctness of this case then follows immediately using the induction hypothesis.

From $(\text{PC}(\underline{\Xi}_{k+1}) : \texttt{stack-0}, \text{Context}(k+1)) \to (u, \bar{x})$ and the induced constraints, it follows that $\text{PC}(\underline{\Xi}_{k+1}) : \texttt{stack-0} \to \text{S}_{pc'+1}\langle\text{head} :: \varepsilon\rangle$, $M(\text{S}_{pc'+1})\langle\text{head} :: \varepsilon\rangle \to t$ and $(t, \text{Context}(k+1)) \to (u, \bar{x})$, for some t.

We also have $\{M(\text{S}_{pc'+1}) \lessdot_{pc'\text{-}(pc'+1)} M(\text{S}_{pc'}), M(\text{S}_{pc'}) \rhd_{\text{head}} M(\text{T}_{pc',\text{r}})\} \subseteq C$ by the induced constraints. Therefore $\{t \lessdot_{pc'\text{-}(pc'+1)} M(\text{T}_{pc',\text{r}})\} \subseteq C$ and then $(M(\text{T}_{pc',\text{r}}), \text{Context}(c)) \to (u, \bar{x})$.

We apply the preservation of return types lemma (Section 6.7.4.2) at $i = k$, obtaining $\exists\bar{w}.\ \text{Context}(k) = \bar{w} \oplus \text{Context}(c+1) \wedge (M(\text{R}_{pc}), \bar{w}) \to (M(\text{R}_{(methodImpl,\,0)}), \varepsilon)$.

Now $pc : \texttt{stack-0} \to \text{S}_{pc}\langle\text{head} :: \varepsilon\rangle$. The constraint induced by the return instruction is $\{M(\text{S}_{pc}) \rhd_{\text{head}} M(\text{R}_{pc})\} \subseteq C$, i.e. $M(\text{S}_{pc})\langle\text{head} :: \varepsilon\rangle \to M(\text{R}_{pc})$. We just obtained $(M(\text{R}_{pc}), \bar{w}) \to (M(\text{R}_{(methodImpl,\,0)}), \varepsilon)$. All that remains to be shown is $(M(\text{R}_{(methodImpl,\,0)}), \text{Context}(c+1)) \to (u, \bar{x})$.

Consider the case in which the method was invoked by `invokestatic`. Then $\text{Context}(c+1) = pc' :: \text{Context}(c)$. The constraints $\{ \text{M}_{methodImpl} \preccurlyeq_{\text{pc}'} \text{T}_{pc',\text{m}}, \text{T}_{pc',\text{m}} \rhd_{\text{result}} \text{T}_{pc',\text{r}} \}$ are induced by the rule for `invokestatic`. Therefore $\{M(\text{R}_{(methodImpl,\,0)}) \preccurlyeq_{pc'} M(\text{T}_{pc',\text{r}})\} \subseteq C$. Combining this with $(M(\text{T}_{pc',\text{r}}), \text{Context}(c)) \to (u, \bar{x})$ gives $(M(\text{R}_{(methodImpl,\,0)}), \text{Context}(c+1)) \to (u, \bar{x})$ as required.

Consider the case in which the method was invoked by `invokevirtual`. Choose *methodID* such that $\text{Instruction}(pc') = \texttt{invokevirtual}\ \ methodID$. Set $c = \text{result}$, $i = c$, $v' = M(\text{R}_{(methodImpl,\,0)})$ and $v = M(\text{T}_{pc',\text{r}})$. The intial constraints contain $\{\text{M}_{methodImpl} \rhd_{\text{result}} \text{R}_{(methodImpl,\,0)}, \text{T}_{pc',\text{m}} \rhd_{\text{result}} \text{T}_{pc',\text{r}}, \text{T}_{pc',\text{v0}} \rhd_{methodID} \text{T}_{pc',\text{m}}\}$. Now we appeal to the preservation of virtual call types (Lemma 6-19), applied to $(M(\text{T}_{pc',\text{r}}), \text{Context}(c)) \to (u, \bar{x})$, to obtain $(M(\text{R}_{(methodImpl,\,0)}), \text{Context}(c+1)) \to (u, \bar{x})$, as required.

6.7.5.18 Induction Step: exceptional returns

The rule for exceptional returns gives

$$\Xi_k = [\text{mode: THROWING, pc: } pc, \text{ wstack: } ref :: \varepsilon, \text{ locals: } \mathcal{L}, \text{ mstack: } (pc'', \mathcal{S}', \mathcal{L}') :: \mathcal{Q}, \rho]$$
$$\Xi_{k+1} = [\text{mode: THROWING, pc: } pc'', \text{ wstack: } ref :: \varepsilon, \text{ locals: } \mathcal{L}', \text{ mstack: } \mathcal{Q}, \rho]$$

Let $c = \text{CallerState}(k)$ and $pc' = \text{PC}(\Xi_c)$. The transition $\Xi_c \Rightarrow \Xi_{c+1}$ must be an application of `invokestatic` or `invokevirtual`, because only those rules extend $\mathcal{Q}$. Therefore $\text{Instruction}(pc') = \texttt{invokevirtual}\ \ methodID$ or $\text{Instruction}(pc') = \texttt{invokestatic}\ \ methodImpl$. In the latter case, $methodImpl = \text{CodeLocMethod}(\text{PC}(\Xi_{c+1}))$; in the former case, define $methodImpl = \text{CodeLocMethod}(\text{PC}(\Xi_{c+1}))$. In either case, N contains

$$\{ \text{S}_{pc'} \rhd_{\text{tail}} \text{T}_{pc',\text{t1}}, \text{S}_{pc'} \rhd_{\text{head}} \text{T}_{pc',\text{v1}}, \text{T}_{pc',\text{t1}} \rhd_{\text{tail}} \text{T}_{pc',\text{t2}}, \text{T}_{pc',\text{t1}} \rhd_{\text{head}} \text{T}_{pc',\text{v0}},$$
$$\text{S}'_{pc'} \rhd_{\text{tail}} \text{T}_{pc',\text{t2}}, \text{S}'_{pc'} \rhd_{\text{head}} \text{T}_{pc',\text{r}} \} \cup$$
$$\text{MethodCall}(\text{T}_{pc,\text{m}}, \text{T}_{pc,\text{v0}}, \text{T}_{pc,\text{v1}}, \text{G}_{pc}, \text{W}_{pc}, \text{T}_{pc,\text{r}})$$

Note also that $\text{Context}(k+1) = \text{err-}pc :: \text{Context}(c)$.

By the lemma governing preservation of caller state (Lemma 6-15), $\Xi_c = [\text{pc: } pc'', \text{ wstack: } v'_1 :: v'_0 :: \mathcal{S}', \text{ locals: } \mathcal{L}', \text{ mstack: } \mathcal{Q}, \rho]$. This implies $pc'' = pc'$.

Case: $exp = \texttt{stack-}m$.

This case cannot occur because stack expressions do not evaluate to anything in the THROWING state.

Case: $exp = \texttt{local-}n$ for some n.

Then $v = \mathcal{L}'(n)$, and therefore $(\Xi_c, pc' : exp) \twoheadrightarrow v$. From $(\text{PC}(\Xi_{k+1}) : \texttt{local-}n, \text{Context}(k+1)) \to (u, \bar{x})$, and observing that $\text{Context}(k+1) = \text{Context}(c)$ and $\text{PC}(\Xi_{k+1}) = \text{PC}(\Xi_c)$, clearly

$(\mathrm{PC}(\Xi_c) : \texttt{local-}n, \mathrm{Context}(c)) \rightarrow (u, \bar{x})$. Applying the induction hypothesis setting $i = c$ gives the required result.

Case: $exp = \texttt{exn}$.

Then $v = ref$, and therefore $(\Xi_k, pc : \texttt{exn}) \twoheadrightarrow v$. I will prove that $(pc : \texttt{exn}, \mathrm{Context}(k)) \rightarrow (u, \bar{x})$; the correctness of this case then follows immediately using the induction hypothesis.

From $(\mathrm{PC}(\Xi_{k+1}) : \texttt{exn}, \mathrm{Context}(k+1)) \rightarrow (u, \bar{x})$ and the induced constraints, it follows that $\mathrm{PC}(\Xi_{k+1}) : \texttt{exn} \rightarrow \mathrm{X}_{pc'}\langle\varepsilon\rangle$ where $(M(\mathrm{X}_{pc'}), \mathrm{Context}(k+1)) \rightarrow (u, \bar{x})$.

I apply the preservation of return types lemma (Section 6.7.4.2) at $i = k$, obtaining $\exists \bar{w}.\ \mathrm{Context}(k) = \bar{w} \oplus \mathrm{Context}(c+1) \wedge (M(\mathrm{X}_{pc}), \bar{w}) \rightarrow (M(\mathrm{X}_{(methodImpl,\,0)}), \varepsilon)$.

Now $pc : \texttt{exn} \rightarrow \mathrm{X}_{pc}\langle\varepsilon\rangle$. All that remains to be shown is $(M(\mathrm{X}_{(methodImpl,\,0)}), \mathrm{Context}(c+1)) \rightarrow (u, \bar{x})$.

Consider the case in which the method was invoked by `invokestatic`. Then $\mathrm{Context}(c+1) = pc' :: \mathrm{Context}(c)$. The constraints $\{\mathrm{M}_{methodImpl} \leqslant_{\mathrm{pc'}} \mathrm{T}_{pc',\mathrm{m}}, \mathrm{T}_{pc',\mathrm{m}} \rhd_{\mathrm{exn}} \mathrm{X}_{pc'}\}$ are induced by the rule for `invokestatic`. Therefore $\{M(\mathrm{X}_{(methodImpl,\,0)}) \leqslant_{pc'} M(\mathrm{X}_{pc'})\} \subseteq C$. Combining this with $(M(\mathrm{X}_{pc'}), \mathrm{Context}(c)) \rightarrow (u, \bar{x})$ gives $(M(\mathrm{X}_{(methodImpl,\,0)}), \mathrm{Context}(c+1)) \rightarrow (u, \bar{x})$ as required.

Consider the case in which the method was invoked by `invokevirtual`. Choose $methodID$ such that $\mathrm{Instruction}(pc') = \texttt{invokevirtual}\ \ methodID$. Set $c = \mathrm{exn}$, $i = c$, $v' = M(\mathrm{X}_{(methodImpl,\,0)})$ and $v = M(\mathrm{X}_{pc'})$. The intial constraints contain $\{\mathrm{M}_{methodImpl} \rhd_{\mathrm{exn}} \mathrm{X}_{(methodImpl,\,0)}, \mathrm{T}_{pc',\mathrm{m}} \rhd_{\mathrm{exn}} \mathrm{X}_{pc'}, \mathrm{T}_{pc',\mathrm{v0}} \rhd_{methodID} \mathrm{T}_{pc',\mathrm{m}}\}$. Now I appeal to the preservation of virtual call types, applied to $(M(\mathrm{X}_{pc'}), \mathrm{Context}(c)) \rightarrow (u, \bar{x})$, to obtain $(M(\mathrm{X}_{(methodImpl,\,0)}), \mathrm{Context}(c+1)) \rightarrow (u, \bar{x})$, as required.

6.7.5.19 Induction Step: `athrow` rule

The rule for `athrow` gives

$$\mathrm{Instruction}(pc) = \texttt{athrow}$$
$$\Xi_k = [\text{mode: RUNNING, pc: } pc, \text{ wstack: } v' :: \underline{\mathcal{S}}, \text{ locals: } \underline{\mathcal{L}}, \rho]$$
$$\Xi_{k+1} = [\text{mode: THROWING, pc: } pc, \text{ wstack: } v' :: \varepsilon, \text{ locals: } \underline{\mathcal{L}}, \rho]$$

Furthermore $\mathrm{Context}(k) = \mathrm{Context}(k+1)$, and the induced constraint is $\{\mathrm{S}_{pc} \rhd_{\mathrm{head}} \mathrm{X}_{pc}\}$.

The function R is:

$R(\texttt{stack-}m)$	is undefined
$R(\texttt{exn})$	$= \texttt{stack-0}$
$R(\texttt{local-}n)$	$= \texttt{local-}n$

Case: $exp = \texttt{stack-}m$. Then $\mathrm{R}(exp) = \texttt{stack-}m$.

This case cannot occur because stack expressions do not evaluate to anything in the THROWING state.

Case: $exp = \texttt{exn}$. Then $\mathrm{R}(exp) = \texttt{stack-0}$.

The evaluation rules show $v = v'$ and therefore $(\underline{\Xi}_k, pc : \texttt{stack-0}) \twoheadrightarrow v' = v$.

Now, $(pc : \texttt{exn}, \mathrm{Context}(k+1)) \to (u, \bar{x})$ implies that $(\mathrm{X}_{pc}, \mathrm{Context}(k)) \to (u, \bar{x})$. But since $\{M(\mathrm{S}_{pc}) \rhd_{\mathrm{head}} M(\mathrm{X}_{pc})\} \subseteq C$, it follows that $(pc : \texttt{stack-0}, \mathrm{Context}(k)) \to (u, \bar{x})$. The result then follows directly from the induction hypothesis.

Case: $exp = \texttt{local-}n$.

The proof for this case is identical to the proof for the corresponding case for spontaneous exception throws.

6.7.5.20 Induction Step: rule for exception catching

The rule for exception catching gives

$$\underline{\Xi}_k = [\text{mode: THROWING, pc: } (method, offset), \text{ wstack: } ref :: \varepsilon, \text{ locals: } \underline{\mathcal{L}}, \rho]$$
$$\underline{\Xi}_{k+1} = [\text{mode: RUNNING, pc: } (method, handler), \text{ wstack: } ref :: \varepsilon, \text{ locals: } \underline{\mathcal{L}}, \rho]$$

where for some $classID$, $handler = \mathrm{CatchBlockOffset}((method, offset), classID)$.

The the initial constraints contain

$$\mathrm{Succ}((method, offset), (method, handler), \mathrm{S}'_{\text{exn-}(method,\ offset)\text{-}classID}, \mathrm{L}_{(method,\ offset)}) \cup$$
$$\{ \mathrm{S}'_{\text{exn-}(method,\ offset)\text{-}classID} \rhd_{\mathrm{head}} \mathrm{X}_{(method,\ offset)} \}.$$

The function R is:

$R(\texttt{stack-}m)$	is undefined	$m > 0$
$R(\texttt{stack-0})$	$= \texttt{exn}$	
$R(\texttt{local-}n)$	$= \texttt{local-}n$	

We also have $\mathrm{Context}(k+1) = (method, offset)\text{-}(method, handler) :: \mathrm{Context}(k)$.

Case: $exp = \texttt{stack-}m$.

Since $(\underline{\Xi}_{k+1}, (method, handler) : \texttt{stack-}m) \twoheadrightarrow v$, $v = ref$ and $m = 0$. Then $\mathrm{R}(exp) = \texttt{exn}$. The rules for evaluation give $(\underline{\Xi}_k, (method, offset) : \texttt{exn}) \twoheadrightarrow v$.

Now, $((method, handler) : \texttt{stack-0}, \mathrm{Context}(k+1)) \to (u, \bar{x})$ implies that for some t, $\{M(\mathrm{S}_{(method,\ handler)}) \rhd_{\mathrm{head}} t\} \subseteq C$ and $(t, (method, offset)\text{-}(method, handler) :: \mathrm{Context}(k)) \to (u, \bar{x})$. We also have $\{M(\mathrm{S}_{(method,\ handler)}) \preccurlyeq_{(method,\ offset)\text{-}(method,\ handler)} M(\mathrm{S}'_{\text{exn-}(method,\ offset)\text{-}classID})\} \subseteq C$. Therefore, for some t', $\{t \preccurlyeq_{(method,\ offset)\text{-}(method,\ handler)} t', M(\mathrm{S}'_{\text{exn-}(method,\ offset)\text{-}classID}) \rhd_{\mathrm{head}} t'\} \subseteq C$. Indeed,

$t' = M(X_{(method,\ offset)})$. Therefore $(M(X_{(method,\ offset)}), \text{Context}(k)) \to (u, \bar{x})$. This implies $((method, offset):\texttt{exn}, \text{Context}(k)) \to (u, \bar{x})$. The result then follows from the induction hypothesis.

Case: $exp = \texttt{local-}n$.

The proof of this case is identical to that for the corresponding case for `load`.

6.7.5.21 Induction Step: `getfield` rule

The rule for `getfield` gives

$\text{Instruction}(pc) = \texttt{getfield}\ fieldID$
$\underline{\Xi}_k = [\text{pc: } pc, \text{ wstack: } ref :: \underline{\mathcal{S}}, \text{ heap: } \underline{\mathcal{H}}, \text{ locals: } \underline{\mathcal{L}}, \rho]$
$\underline{\Xi}_{k+1} = [\text{pc: } pc + 1, \text{ wstack: HeapObjFields}(\underline{\mathcal{H}}(\text{Val}(ref)))(fieldID) :: \underline{\mathcal{S}}, \text{ heap: } \underline{\mathcal{H}}, \text{ locals: } \underline{\mathcal{L}}, \rho]$

Also, the induced constraints are

$\{ S_{pc} \rhd_{\text{tail}} T_{pc,\text{t}}, S_{pc} \rhd_{\text{head}} T_{pc,\text{obj}}, T_{pc,\text{obj}} \rhd_{fieldID} T_{pc,\text{v}}, S'_{pc} \rhd_{\text{head}} T_{pc,\text{v}},$
$S'_{pc} \rhd_{\text{tail}} T_{pc,\text{t}} \} \cup \text{Succ}(pc, pc{+}1, S'_{pc}, L_{pc}, G_{pc}, X_{pc}, R_{pc})$

The function R is:

$R(\texttt{stack-}m)$	$= \texttt{stack-}m$	$m > 0$
$R(\texttt{stack-0})$	$= \texttt{stack-0}.fieldID$	
$R(\texttt{local-}n)$	$= \texttt{local-}n$	

Case: $exp = \texttt{stack-}m$, $m > 0$. Then $R(exp) = \texttt{stack-}m$.

The evaluation rules show $\text{HeapObjFields}(\underline{\mathcal{H}}(\text{Val}(ref)))(fieldID) :: \underline{\mathcal{S}}$ is of the form $v_0' :: v_1' :: \ldots :: v_m' :: \underline{\mathcal{S}}'$ where $v_m' = v$. Therefore $\text{MStack}(\underline{\Xi}_k) = ref :: v_1' :: \ldots :: v_m' :: \underline{\mathcal{S}}'$ and $(\underline{\Xi}_k, pc : \texttt{stack-}m) \twoheadrightarrow v_m' = v$, as required.

In this case I apply the succession lemma (6-22) with $t = S_{pc+1}$ and $\bar{c} = \text{tail} :: \ldots :: \text{tail} :: \text{head} :: \varepsilon$, with $m > 0$ occurrences of "tail". Also, $s = M(S'_{pc})$. Therefore $M(S'_{pc})\langle\bar{c}\rangle \to t'$ where $(t', \text{Context}(k)) \to (u, \bar{x})$; this implies $M(T_{pc,\text{t}})\langle\bar{c}'\rangle \to t'$, where $\bar{c} = \text{tail} :: \bar{c}'$. Then $M(S_{pc})\langle\bar{c}\rangle \to t'$. Also $\text{PC}(\underline{\Xi}_k) : \texttt{stack-}m \to M(S_{pc})\langle\bar{c}\rangle$. All together then, $(\text{PC}(\underline{\Xi}_k) : \texttt{stack-}m, \text{Context}(k)) \to (u, \bar{x})$; the result follows immediately from the induction hypothesis.

Case: $exp = \texttt{stack-0}$. Then $R(exp) = \texttt{stack-0}.fieldID$.

The evaluation rules give $v = \text{HeapObjFields}(\underline{\mathcal{H}}(\text{Val}(ref)))(fieldID)$ and $(\underline{\Xi}_k, pc : \texttt{stack-0}.fieldID) \twoheadrightarrow v$.

In this case I apply the succession lemma (6-22) with $t = S_{pc+1}$ and $\bar{c} = \text{head} :: \varepsilon$. Also, $s = M(S'_{pc})$. Therefore $M(S'_{pc})\langle\text{head} :: \varepsilon\rangle \to t'$ where $(t', \text{Context}(k)) \to (u, \bar{x})$; this implies $t' = M(T_{pc,\text{v}})$. Furthermore,

$\mathrm{PC}(\Xi_k) : \texttt{stack-0}.\mathit{fieldID} \rightarrow \mathrm{S}_{pc}\langle \mathrm{head} :: \mathit{fieldID} :: \varepsilon \rangle$ and $\mathcal{M}(\mathrm{S}_{pc})\langle \mathrm{head} :: \mathit{fieldID} :: \varepsilon \rangle \rightarrow \mathcal{M}(\mathrm{T}_{pc,\mathrm{v}})$. All together then, $(\mathrm{PC}(\Xi_k) : \texttt{stack-0}.\mathit{fieldID}, \mathrm{Context}(k)) \rightarrow (u, \bar{x})$; the result follows immediately from the induction hypothesis.

Case: $exp = \texttt{local-}n$.

The proof of this case is identical to that for the corresponding case for `load`.

6.7.5.22 Induction Step: `putfield` rule

The rule for `putfield` gives

$$\mathrm{Instruction}(pc) = \texttt{putfield}\ \mathit{fieldID}$$
$$\Xi_k = [\mathrm{pc}: pc,\ \mathrm{wstack}: v' :: \mathit{ref} :: \underline{\mathcal{S}},\ \mathrm{locals}: \underline{\mathcal{L}},\ \rho]$$
$$\Xi_{k+1} = [\mathrm{pc}: pc + 1,\ \mathrm{wstack}: \underline{\mathcal{S}},\ \mathrm{locals}: \underline{\mathcal{L}},\ \rho]$$

The induced constraints are

$$\{ \mathrm{S}_{pc} \rhd_{\mathrm{tail}} \mathrm{T}_{pc,\mathrm{t}}, \mathrm{S}_{pc} \rhd_{\mathrm{head}} \mathrm{T}_{pc,\mathrm{v}}, \mathrm{T}_{pc,\mathrm{t}} \rhd_{\mathrm{tail}} \mathrm{S}'_{pc}, \mathrm{T}_{pc,\mathrm{t}} \rhd_{\mathrm{head}} \mathrm{T}_{pc,\mathrm{obj}}, \mathrm{T}_{pc,\mathrm{obj}} \rhd_{\mathit{fieldID}} \mathrm{T}_{pc,\mathrm{v}} \} \cup \mathrm{Succ}(pc, pc{+}1, \mathrm{S}'_{pc}, \mathrm{L}_{pc}, \mathrm{G}_{pc}, \mathrm{X}_{pc}, \mathrm{R}_{pc})$$

The function R is:

$R(\texttt{stack-}m)$	$= \texttt{stack-}(m+2)$
$R(\texttt{local-}n)$	$= \texttt{local-}n$

Case: $exp = \texttt{stack-}m$. Then $\mathrm{R}(exp) = \texttt{stack-}(m+2)$.

The evaluation rules show $\underline{\mathcal{S}}$ is of the form $v_0' :: v_1' :: \ldots :: v_m' :: \underline{\mathcal{S}}'$ where $v_m' = v$. Therefore $\mathrm{MStack}(\Xi_k) = v' :: \mathit{ref} :: v_0' :: \ldots :: v_m' :: \underline{\mathcal{S}}'$ and $(\Xi_k, pc : \texttt{stack-}(m+2)) \twoheadrightarrow v_m' = v$, as required.

In this case I apply the succession lemma (6-22) with $t = \mathrm{S}_{pc+1}$ and $\bar{c} = \mathrm{tail} :: \ldots :: \mathrm{tail} :: \mathrm{head} :: \varepsilon$, with m occurrences of "tail". Also, $s = \mathcal{M}(\mathrm{S}'_{pc})$. Therefore $\mathcal{M}(\mathrm{S}'_{pc})\langle \bar{c} \rangle \rightarrow t'$ where $(t', \mathrm{Context}(k)) \rightarrow (u, \bar{x})$; this implies $\mathcal{M}(\mathrm{S}_{pc})\langle \mathrm{tail} :: \mathrm{tail} :: \bar{c} \rangle \rightarrow t'$. Also $\mathrm{PC}(\Xi_k) : \texttt{stack-}(m+2) \rightarrow \mathcal{M}(\mathrm{S}_{pc})\langle \mathrm{tail} :: \mathrm{tail} :: \bar{c} \rangle$. All together then, $(\mathrm{PC}(\Xi_k) : \texttt{stack-}(m+2), \mathrm{Context}(k)) \rightarrow (u, \bar{x})$; the result follows immediately from the induction hypothesis.

Case: $exp = \texttt{local-}n$.

The proof of this case is identical to that for the corresponding case for `load`.

6.7.5.23 Induction Step: `getstatic` rule

The rule for `getstatic` gives

$$\mathrm{Instruction}(pc) = \texttt{getstatic}\ \ \mathit{staticField}$$
$$\Xi_k = [\mathrm{pc}: pc,\ \mathrm{wstack}: \underline{\mathcal{S}},\ \mathrm{globals}: \underline{\mathcal{G}},\ \mathrm{locals}: \underline{\mathcal{L}},\ \rho]$$
$$\Xi_{k+1} = [\mathrm{pc}: pc + 1,\ \mathrm{wstack}: \underline{\mathcal{G}}(\mathit{staticField}) :: \underline{\mathcal{S}},\ \mathrm{globals}: \underline{\mathcal{G}},\ \mathrm{locals}: \underline{\mathcal{L}},\ \rho]$$

The induced constraints are

$\{ \mathrm{G}_{pc} \rhd_{staticField} \mathrm{T}_{pc,\mathrm{v}}, \mathrm{S}'_{pc} \rhd_{\mathrm{tail}} \mathrm{S}_{pc}, \mathrm{S}'_{pc} \rhd_{\mathrm{head}} \mathrm{T}_{pc,\mathrm{v}} \} \cup$
$\mathrm{Succ}(pc, pc{+}1, \mathrm{S}'_{pc}, \mathrm{L}_{pc}, \mathrm{G}_{pc}, \mathrm{X}_{pc}, \mathrm{R}_{pc})$.

The function R is:

$R(\texttt{stack-}m)$	$= \texttt{stack-}(m-1)$	$m > 0$
$R(\texttt{stack-0})$	$= staticField$	
$R(\texttt{local-}n)$	$= \texttt{local-}n$	

Case: $exp = \texttt{stack-}m$, $m > 0$. Then $\mathrm{R}(exp) = \texttt{stack-}m$.

The proof for this case is identical to that for the corresponding case for `load`.

Case: $exp = \texttt{stack-0}$. Then $\mathrm{R}(exp) = staticField$.

The evaluation rules give $v = \mathcal{G}(staticField)$ and therefore $(\Xi_k, pc : staticField) \rightsquigarrow v$.

In this case I apply the succession lemma (6-22) with $t = \mathrm{S}_{pc+1}$ and $\bar{c} = \mathrm{head} :: \varepsilon$. Also, $s = M(\mathrm{S}'_{pc})$. Therefore $M(\mathrm{S}'_{pc})\langle \mathrm{head} :: \varepsilon \rangle \to t'$ where $(t', \mathrm{Context}(k)) \to (u, \bar{x})$; this implies $t' = M(\mathrm{T}_{pc,\mathrm{v}})$. Furthermore, $\mathrm{PC}(\Xi_k) : staticField \to \mathrm{G}_{pc} \langle staticField :: \varepsilon \rangle$ and $M(\mathrm{G}_{pc}) \langle staticField :: \varepsilon \rangle \to M(\mathrm{T}_{pc,\mathrm{v}})$. All together then, $(\mathrm{PC}(\Xi_k) : staticField, \mathrm{Context}(k)) \to (u, \bar{x})$; the result follows immediately from the induction hypothesis.

Case: $exp = \texttt{local-}n$.

The proof of this case is identical to that for the corresponding case for `load`.

6.7.5.24 Induction Step: `putstatic` rule

The rule for `putstatic` gives

$\mathrm{Instruction}(pc) = \texttt{putstatic}\ fieldID$
$\Xi_k = [\text{pc: } pc, \text{wstack: } v' :: \underline{\mathcal{S}}, \text{locals: } \underline{\mathcal{L}}, \rho]$
$\Xi_{k+1} = [\text{pc: } pc + 1, \text{wstack: } \underline{\mathcal{S}}, \text{locals: } \underline{\mathcal{L}}, \rho]$

The induced constraints are

$\{ \mathrm{S}_{pc} \rhd_{\mathrm{tail}} \mathrm{S}'_{pc}, \mathrm{S}_{pc} \rhd_{\mathrm{head}} \mathrm{T}_{pc,\mathrm{v}}, \mathrm{G}_{pc} \rhd_{fieldID} \mathrm{T}_{pc,\mathrm{v}} \} \cup$
$\mathrm{Succ}(pc, pc{+}1, \mathrm{S}'_{pc}, \mathrm{L}_{pc}, \mathrm{G}_{pc}, \mathrm{X}_{pc}, \mathrm{R}_{pc})$.

The function R is:

$R(\texttt{stack-}m)$	$= \texttt{stack-}(m+1)$
$R(\texttt{local-}n)$	$= \texttt{local-}n$

Case: $exp = \texttt{stack-}m$. Then $\mathrm{R}(exp) = \texttt{stack-}(m+1)$.

The proof for this case is identical to that for the corresponding case for `store`.

Case: $exp = \texttt{local-}n$.

The proof of this case is identical to that for the corresponding case for `load`.

6.7.5.25 Induction Step: `iadd` **rule**

The rule for `iadd` gives

$$\text{Instruction}(pc) = \texttt{iadd}\ classID$$
$$\underline{\Xi}_k = [\text{pc: } pc, \text{wstack: } v_1 :: v_2 :: \underline{\mathcal{S}}, \text{locals: } \underline{\mathcal{L}}, \rho]$$
$$\underline{\Xi}_{k+1} = [\text{pc: } pc + 1, \text{wstack: } (\text{Val}(v_1) + \text{Val}(v_2), t) :: \underline{\mathcal{S}}, \text{locals: } \underline{\mathcal{L}}, \rho]$$

The induced constraints are

$$\{ \text{S}_{pc} \rhd_{\text{tail}} \text{T}_{pc,\text{t1}}, \text{T}_{pc,\text{t1}} \rhd_{\text{tail}} \text{T}_{pc,\text{t2}}, \text{S}'_{pc} \rhd_{\text{tail}} \text{T}_{pc,\text{t2}}, \text{S}'_{pc} \rhd_{\text{head}} \text{T}_{pc,\text{v}} \} \cup$$
$$\text{Succ}(pc, pc{+}1, \text{S}'_{pc}, \text{L}_{pc}, \text{G}_{pc}, \text{X}_{pc}, \text{R}_{pc})$$

The function R is:

$R(\texttt{stack-}m)$	$= \texttt{stack-}(m+1)$	$m > 0$
$R(\texttt{stack-0})$	is undefined	
$R(\texttt{local-}n)$	$= \texttt{local-}n$	

Case: $exp = \texttt{stack-}m$. Then $\text{R}(exp) = \texttt{stack-}(m+1)$.

The evaluation rules show $(\text{Val}(v_1) + \text{Val}(v_2), t) :: \underline{\mathcal{S}}$ is of the form $v_0' :: v_1' :: \ldots :: v_m' :: \underline{\mathcal{S}}'$ where $v_m' = v$. Therefore $\text{MStack}(\underline{\Xi}_k) = v_1 :: v_2 :: v_1' :: \ldots :: v_m' :: \underline{\mathcal{S}}'$ and $(\underline{\Xi}_k, pc : \texttt{stack-}(m+1)) \twoheadrightarrow v_m' = v$, as required.

In this case I apply the succession lemma (6-22) with $t = \text{S}_{pc+1}$ and $\bar{c} = \text{tail} :: \ldots :: \text{tail} :: \text{head} :: \varepsilon$, with $m > 0$ occurrences of "tail". Also, $s = M(\text{S}'_{pc})$. Therefore $M(\text{S}'_{pc})\langle\bar{c}\rangle \to t'$ where $(t', \text{Context}(k)) \to (u, \bar{x})$. This implies that $\bar{c}$ is of the form $\text{tail} :: \bar{c}'$ where $M(\text{T}_{pc,\text{t2}})\langle\bar{c}'\rangle \to t'$. This in turn implies $M(\text{S}_{pc})\langle\text{tail} :: \text{tail} :: \bar{c}'\rangle \to t'$. Also $\text{PC}(\underline{\Xi}_k) : \texttt{stack-}(m+1) \to M(\text{S}_{pc})\langle\text{tail} :: \bar{c}\rangle$. All together then, $(\text{PC}(\underline{\Xi}_k) : \texttt{stack-}(m+1), \text{Context}(k)) \to (u, \bar{x})$; the result follows immediately from the induction hypothesis.

Case: $exp = \texttt{stack-0}$.

Then $\text{Creation}(v) = (k+1, (pc+1) : \texttt{stack-0})$ by the definition of Creation, so the induction result is satisfied.

Case: $exp = \texttt{local-}n$.

The proof of this case is identical to that for the corresponding case for `load`.

6.7.5.26 Induction Step: `ifcmpeq` **rules**

The rules for `ifcmpeq` give

$\text{Instruction}(pc) = \texttt{if\_cmpeq}\ \textit{offset}$
$\underline{\Xi}_k = [\text{pc: } pc, \text{ wstack: } v' :: \underline{\mathcal{S}}, \text{ locals: } \underline{\mathcal{L}}, \rho]$
$\underline{\Xi}_{k+1} = [\text{pc: } pc', \text{ wstack: } \underline{\mathcal{S}}, \text{ locals: } \underline{\mathcal{L}}, \rho]$

where either $pc' = pc + 1$ or $pc' = (\text{CodeLocMethod}(pc), \textit{offset})$.

The induced constraints are

$\{ \text{S}_{pc} \triangleright_{\text{tail}} \text{S}'_{pc} \} \cup \text{Succ}(pc, pc{+}1, \text{S}'_{pc}, \text{L}_{pc}, \text{G}_{pc}, \text{X}_{pc}, \text{R}_{pc}) \cup$
$\text{Succ}(pc, (\text{CodeLocMethod}(pc), \textit{offset}), \text{S}'_{pc}, \text{L}_{pc}, \text{G}_{pc}, \text{X}_{pc}, \text{R}_{pc}).$

The function R is:

$$R(\texttt{stack-}m) = \texttt{stack-}(m+1)$$
$$R(\texttt{local-}n) = \texttt{local-}n$$

Case: $exp = \texttt{stack-}m$.

The proof for this case is identical to that for the corresponding case for `store`. The successor lemma is applicable regardless of which branch is taken.

Case: $exp = \texttt{local-}n$.

The proof of this case is identical to that for the corresponding case for `load`. The successor lemma is applicable regardless of which branch is taken.

6.7.5.27 Induction Step: `goto` rule

The rules for `goto` give

$\text{Instruction}(pc) = \texttt{goto}\ \textit{offset}$
$\underline{\Xi}_k = [\text{pc: } pc, \text{ wstack: } \underline{\mathcal{S}}, \text{ locals: } \underline{\mathcal{L}}, \rho]$
$\underline{\Xi}_{k+1} = [\text{pc: } (\text{CodeLocMethod}(pc), \textit{offset}), \text{ wstack: } \underline{\mathcal{S}}, \text{ locals: } \underline{\mathcal{L}}, \rho]$

The induced constraints are

$\text{Succ}(pc, (\text{CodeLocMethod}(pc), \textit{offset}), \text{S}_{pc}, \text{L}_{pc}, \text{G}_{pc}, \text{X}_{pc}, \text{R}_{pc})$

The function R is:

$$R(\texttt{stack-}m) = (\texttt{stack-}m)$$
$$R(\texttt{local-}n) = \texttt{local-}n$$

Case: $exp = \texttt{stack-}m$.

The evaluation rules show $\underline{\mathcal{S}}$ is of the form $v_0' :: v_1' :: \ldots :: v_m' :: \underline{\mathcal{S}}'$ where $v_m' = v$. Therefore $(\underline{\Xi}_k, pc : \texttt{stack-}m) \dashrightarrow v_m' = v$, as required.

In this case I apply the succession lemma (6-22) with $t = \text{S}_{pc+1}$ and $\bar{c} = \text{tail} :: \ldots :: \text{tail} :: \text{head} :: \varepsilon$, with m occurrences of “tail”. Also, $s = M(\text{S}'_{pc})$. Therefore $M(\text{S}'_{pc})\langle\bar{c}\rangle \to t'$ where $(t', \text{Context}(k)) \to (u, \bar{x})$. Also

$\mathrm{PC}(\underline{\Xi}_k) : \texttt{stack-}m \rightarrow M(\mathrm{S}_{pc})\langle\bar{c}\rangle$. All together then,
$(\mathrm{PC}(\underline{\Xi}_k) : \texttt{stack-}m, \mathrm{Context}(k)) \rightarrow (u, \bar{x})$; the result follows immediately from the induction hypothesis.

Case: $exp = \texttt{local-}n$.

The proof of this case is identical to that for the corresponding case for `load`.

6.7.5.28 Induction Step: `instanceof` rules

The rules for `instanceof` give

$$\begin{aligned}
&\mathrm{Instruction}(pc) = \texttt{instanceof}\ \mathit{fieldID} \\
&\underline{\Xi}_k = [\text{pc: } pc, \text{ wstack: } ref :: \underline{\mathcal{S}}, \text{ locals: } \underline{\mathcal{L}}, \rho] \\
&\underline{\Xi}_{k+1} = [\text{pc: } pc + 1, \text{ wstack: } (v', t) :: \underline{\mathcal{S}}, \text{ locals: } \underline{\mathcal{L}}, \rho]
\end{aligned}$$

for some value of v'.

The induced constraints are

$$\{ \mathrm{S}_{pc} \triangleright_{\text{tail}} \mathrm{T}_{pc,\text{t}}, \mathrm{S}'_{pc} \triangleright_{\text{tail}} \mathrm{T}_{pc,\text{t}}, \mathrm{S}'_{pc} \triangleright_{\text{head}} \mathrm{T}_{pc,\text{v}} \} \cup$$
$$\mathrm{Succ}(pc, pc{+}1, \mathrm{S}'_{pc}, \mathrm{L}_{pc}, \mathrm{G}_{pc}, \mathrm{X}_{pc}, \mathrm{R}_{pc})$$

The function R is:

$R(\texttt{stack-}m)$	$= \texttt{stack-}m$	$m > 0$
$R(\texttt{stack-0})$	is undefined	
$R(\texttt{local-}n)$	$= \texttt{local-}n$	

Case: $exp = \texttt{stack-}m$, $m > 0$. Then $\mathrm{R}(exp) = \texttt{stack-}m$.

The proof for this case is the same as the proof for the corresponding case for the `getfield` rule.

Case: $exp = \texttt{stack-0}$.

Then $\mathrm{Creation}(v) = (k + 1, (pc + 1) : \texttt{stack-0})$ by the definition of Creation, so the induction result is trivially satisfied.

Case: $exp = \texttt{local-}n$.

The proof of this case is identical to that for the corresponding case for `load`.

6.7.5.29 Induction Step: `checkcast` rule

The proof for this case is the same as for the `goto` rule. A successful `checkcast` does not change the state in any way.

7 SEMI Implementation

7.1 Introduction

Chapter 6 describes the SEMI constraint system and how it is used to derive safe approximations to the value-point relation. That chapter assumes the existence of an algorithm for deriving a *closed set* of constraints from a given initial set. In this chapter, I describe such an algorithm, as implemented in Ajax's SEMI analysis engine.

First I describe the basic algorithm, and then I present a series of improvements to the algorithm that improve its performance. I also discuss some changes to the algorithm that I tried and rejected because they decreased performance.

Finally, I discuss some changes to the constraint generation phase that simplify the initial constraint set while leading to the same results.

7.1.1 Solver Specification

Given an initial constraint set C_I, the job of the solver is simply to find a closed set C containing C_I.

C_I represents constraints induced by the program under analysis. C represents an extension of those constraints into a complete and consistent description of the "types" in the program.

Note that such a C always exists. For example, given C_I, we can add constraints making all variables equal and making all component and instance relationships hold between all variables. (The resulting set is finite because only the variables, component labels and instance labels that occur in C_I need be considered.) Effectively this gives all expressions the same type. In practice this result would not be useful — it is preferable to retain distinctions between types whenever possible. However, this example illustrates that implementations of the specification can trade off accuracy for performance.

7.1.2 Decidability and Performance

Henglein [42] shows that the problem of finding a principal (i.e., most general) type is undecidable in the general setting of polymorphic recursion. However, in practice all examples seem tractable. In fact, Henglein's algorithm is reported to be quite efficient at inferring types for functional programs.

SEMI is similar to Henglein's algorithm and likewise has no guarantee of termination. (In fact, because SEMI can infer recursive types, the situation is theoretically even more dire than for Henglein's algorithm: typable programs exist that have no principal types. See Appendix A for details.) However, nonterminating cases have always been traced back to errors in the solver implementation. Because the worst cases may not even terminate, efficiency depends on the characteristics of "average case" programs. Therefore we must measure performance and precision empirically.

In fact, the problem of finding a closed constraint set is not the same problem as finding principal types. As noted above, there is no unique solution to the problem of finding a closed set, and a trivial closed set can always be found.[1] However, for the sake of precision we want the analysis to distinguish types whenever possible, just as we do when inferring principal types.

7.1.3 Refined Specification

The SEMI analysis engine extracts an approximate value-point relation from the closed set C. This relation is the only function of C that is used. Therefore we can relax the specification of the engine to allow it to produce any set C' that (for a given set Q of query expressions) gives the same relation as that derived from a closed set C. I will call such a set C' *quasi-closed* with respect to Q. This relaxation enables many optimizations.

The analysis engine actually computes a propagation graph from the constraint set and not a direct approximation to the value-point relation (see Section 6.6.1). However, as shown in Section 6.6, the results computed over the graph are completely determined by the approximate value-point relation defined for the constraint set. Therefore if C' induces the same approximate relation, the results obtained from the propagation graph on C' will be be the same as the results for C's graph.

From the definition of the approximate value-point relation in Section 6.5.1, the analysis concludes $e_1 \overline{\leftrightarrow} e_2$ if and only if

$$\exists u, \bar{x}_1, \bar{x}_2, \bar{x}_1', \bar{x}_2'.\ (e_1, \bar{x}_1) \to (u, \bar{x}_1') \wedge (e_2, \bar{x}_2) \to (u, \bar{x}_2')$$

By the instance transitivity property (Lemma 6-7), this is equivalent to

$$\exists u, \bar{x}_1, \bar{x}_2.\ (e_1, \bar{x}_1) \to (u, \varepsilon) \wedge (e_2, \bar{x}_2) \to (u, \varepsilon)$$

Let M be a map from bytecode expressions to constraint variables, defined as $\mathrm{M}(e) = u$ where $\exists \bar{c}, u'.\ e \to u'\langle \bar{c} \rangle \wedge u'\langle \bar{c} \rangle \to u$. $\mathrm{M}(e)$ is defined for all expressions in the query set Q; this is guaranteed by the precautions in Section 6.4.5. Then the analysis concludes $e_1 \overline{\leftrightarrow} e_2$ if and only if

$$\exists u, \bar{x}_1, \bar{x}_2.\ (\mathrm{M}(e_1), \bar{x}_1) \to (u, \varepsilon) \wedge (\mathrm{M}(e_2), \bar{x}_2) \to (u, \varepsilon)$$

From these definitions, it follows that C' is quasi-closed if there exists a C such that

- C is closed
- C contains $\mathrm{C_I}$
- $\forall t, v \in \mathrm{Variables}(\mathrm{C_I}).\ \exists u, \bar{x}_1, \bar{x}_2.\ (t, \bar{x}_1) \to (u, \varepsilon) \wedge (v, \bar{x}_2) \to (u, \varepsilon)$ in C if and only if $\exists u, \bar{x}_1, \bar{x}_2.\ (t, \bar{x}_1) \to (u, \varepsilon) \wedge (v, \bar{x}_2) \to (u, \varepsilon)$ in C'.

1. For this reason, we could guarantee termination by timing out and falling back to an algorithm that is guaranteed to terminate. SEMI does not do this, however; choosing a suitable timeout interval and selecting an algorithm to fall back on appear to be rather complex problems.

7.1.4 Basic Structure

This chapter describes a series of algorithms leading up to the full SEMI algorithm, each more sophisticated than the last. All the algorithms commence with the initial constraint set C_I and add constraints to the set until it is closed (or quasi-closed).

Because the addition of new constraints to the set is a fundamental operation in the algorithms, it is not difficult to extend these algorithms to be incremental. One can add to the initial constraint set C_I at any time and then continue to add derived constraints until reaching (quasi-) closure.

7.2 Basic Algorithm

The basic algorithm presented in this section corresponds to Henglein's type inference procedure [42].

The general procedure is to start with a set of initial constraints (the input) and repeatedly add constraints to the set until it reaches closed form (the output). This is complicated by the fact that the initial constraint set can increase during processing, and the new constraints can be observed by tools as soon as they are added (i.e., the results are reported incrementally).

Therefore, in reality, the SEMI solver takes a set of constraints as input. If the set is already in closed form, it reports termination, otherwise it adds some constraints to the set and reports the changes in the output of the analysis. The added constraints are chosen to move the set "closer" to closure; that is, if the constraint set output by one step is always used as the input to the next step, the algorithm should terminate (although as discussed above, we cannot guarantee that it will terminate).

7.2.1 Representation of Equality

Like every algorithm of this kind, the SEMI solver uses a representation of the constraint set that avoids explicit equality constraints. Whenever a constraint of the form "$a \cong b$" is encountered or produced, it is discarded, and the solver substitutes b for a (or a for b) in all other constraints. This can be implemented efficiently by treating each variable as an equivalence class and employing the union-find algorithm to merge equivalence classes.

7.2.2 Functional Representation of Components and Instances

The component consistency rule guarantees that for a given variable t and component label c, there is at most one v such that $t \rhd_c v$ (after taking into account equivalencies). Thus the component constraints are represented as a curried partial function $F_\rhd : V \to L \mapsto V$.

Likewise the instance constraints are represented as $F_\preccurlyeq : V \to I \mapsto V$.

In the implementation, each variable v has two hash tables associated with it, one representing $F_\rhd(v)$ and the other $F_\preccurlyeq(v)$.

When a variable v is substituted for u because u and v have been made equal, u's $L \mapsto V$ component map is merged into v's $L \mapsto V$ component map. The tricky part of this process is that for each l in the intersection of their domains, the variable $F_\rhd(u)(l)$ is made equal to the variable $F_{\preccurlyeq}(v)(l)$; thus, the merge procedure can invoke itself recursively. The procedure corresponds to term unification.

The algorithm also merges u's $I \mapsto V$ instance map into v's $I \mapsto V$ instance map. This is similar to the case of the component maps, and can also result in recursive merge calls.

7.2.3 Component Propagation

The above normalization procedures ensure that the constraint set is always closed under all rules except for the component and instance propagation rules.

We treat the remaining rules as production rules:

- Component propagation

Upon detecting $\{ t \preccurlyeq_i u, t \rhd_c v \} \subseteq C$ for some t, u, v, i and c, add a new variable w and constraint $u \rhd_c w$ (unless there is already a w such that $u \rhd_c w$).

- Instance propagation

Upon detecting $\{ t \preccurlyeq_i u, t \rhd_c v, u \rhd_c w \} \subseteq C$ for some t, u, v, w, i and c, add a constraint $v \preccurlyeq_i w$ (if not already present).

These are implemented using a worklist. The algorithm maintains a list of "dirty" component constraints (e.g. "$t \rhd_c v$") that must be checked by the component propagation rule. All component constraints in C_I start off in the dirty list. Whenever a new component constraint is added to C_I, it is added to the dirty list. Whenever a variable t is substituted for another variable w, all the components of t that do not already appear in w are made dirty, and likewise all the components of w that do not already appear in t are made dirty. Formally:

$$\{ t \rhd_c v \mid \{ t \rhd_c v \} \subseteq C \wedge (\neg\exists u.\ \{ w \rhd_c u \} \subseteq C) \} \cup$$
$$\{ w \rhd_c v \mid \{ w \rhd_c v \} \subseteq C \wedge (\neg\exists u.\ \{ t \rhd_c u \} \subseteq C) \}$$

Also, whenever an instance constraint $t \preccurlyeq_i u$ is added, all the components of t and u are made dirty.

During each iteration of the solver, it pulls one dirty component constraint $t \rhd_c v$ from the dirty list. Then for each u and i such that $\{ t \preccurlyeq_i u \} \subseteq C$, the two production rules are checked. Also, for each u and i such that $\{ u \preccurlyeq_i t \} \subseteq C$, the second production rule is checked, swapping u with t and v with w so that the actual rule checked is

- Upon detecting $\{ u \preccurlyeq_i t, u \rhd_c w, t \rhd_c v \} \subseteq C$ for some t, u, v, w, i and c, add a constraint $w \preccurlyeq_i v$ (if not already present).

Note that when checking this rule, since u and c are known, there can be at most one applicable w.

Iteration continues until the worklist of dirty component constraints is empty. Upon termination, the constraint set is closed.

When an equality constraint is processed by applying a substitution to the entire constraint set, the same substitution is applied to the elements of the worklist. Of course, this is done efficiently using a union-find data structure.

7.2.4 Saving Time By Recording Additional Dirtiness Information

For some variables t there may be many u such that $t \preccurlyeq_i u$ or $u \preccurlyeq_i t$. When a dirty component $t \rhd_c v$ is being processed, it can be slow to scan all the instances u such that $t \preccurlyeq_i u$ and all the sources u such that $u \preccurlyeq_i t$. Therefore for each dirty component $t \rhd_c v$, we maintain a list of all the $t \preccurlyeq_i u$ and $u \preccurlyeq_i t$ that need to be inspected in conjunction with the $t \rhd_c v$ constraint. For every situation in which a component constraint may become dirty, there is an associated set of instance and source constraints that will need to be inspected.

When a new component constraint $t \rhd_c v$ is added, all constraints of the form $t \preccurlyeq_i u$ and $u \preccurlyeq_i t$ need to be inspected in conjunction with $t \rhd_c v$.

When a variable t is substituted for variable w, then for each $t \rhd_c v$ such that $\{ t \rhd_c v \} \subseteq C \wedge (\neg\exists u.\ \{ w \rhd_c u \} \subseteq C)$, all constraints of the form $w \preccurlyeq_i u$ and $u \preccurlyeq_i w$ need to be inspected in conjunction with $t \rhd_c v$. Likewise, for each $w \rhd_c v$ such that $\{ w \rhd_c v \} \subseteq C \wedge (\neg\exists u.\ \{ t \rhd_c u \} \subseteq C)$, all constraints of the form $t \preccurlyeq_i u$ and $u \preccurlyeq_i t$ need to be inspected in conjunction with $w \rhd_c v$.

Whenever an instance constraint $t \preccurlyeq_i u$ is added, then for each $t \rhd_c v$ in C, the instance constraint $t \preccurlyeq_i u$ must be inspected in conjunction with $t \rhd_c v$. Also, for each $u \rhd_c v$ in C, the source constraint $t \preccurlyeq_i u$ must be inspected in conjunction with $u \rhd_c v$.

This additional bookkeeping greatly improves runtime, while adding some space overhead.

7.2.5 Overview of an Algorithm Step

An iteration of the solver proceeds as follows:

1. Remove a dirty component constraint $t \rhd_c v$ from the worklist, with its associated sets of dirty source constraints S and dirty instance constraints I.
2. For each dirty source constraint $u \preccurlyeq_i t$ in S, we have $\{ u \preccurlyeq_i t, t \rhd_c v \} \subseteq C$. Each production rule has premises of the form $P \subseteq C$. For each rule, and for each instantiation of the free variables of P such that $\{ u \preccurlyeq_i t, t \rhd_c v \} \subseteq P$ and $P \subseteq C$, SEMI applies the rule to obtain a set of constraints that must be included in the new constraint set. Each new constraint not already in the set is added and the dirty worklist is updated appropriately.
3. For each dirty instance constraint $t \preccurlyeq_i u$ in I, we have $\{ t \preccurlyeq_i u, t \rhd_c v \} \subseteq C$. For each production rule, and for each instantiation of the free variables of the rule's premises P such that $\{ t \preccurlyeq_i u, t \rhd_c v \} \subseteq P$ and $P \subseteq C$, SEMI applies the rule to obtain a set of constraints to add, as above.

For each rule, it is easy to determine the possible values of P given that $\{ u \preccurlyeq_i t, t \rhd_c v \} \subseteq P$ or $\{ t \preccurlyeq_i u, t \rhd_c v \} \subseteq P$.

Consider the component propagation rule. P is of the form $\{ q \preccurlyeq_i r, q \rhd_c s \}$. When checking dirty instances, we have $\{ t \preccurlyeq_i u, t \rhd_c v \} \subseteq P$. The only possibility is $P = \{ t \preccurlyeq_i u, t \rhd_c v \}$, so the consequence of the rule is $\exists w.\ \{u \rhd_c w\} \subseteq C$. When checking dirty sources, we have $\{ u \preccurlyeq_i t, t \rhd_c v \} \subseteq P$. The only possibility is $P = \{ u \preccurlyeq_i t, t \rhd_c v \}$, but then since P is of the form $\{ q \preccurlyeq_i r, q \rhd_c s \}$, we must have $u = t$ and $P = \{ t \preccurlyeq_i t, t \rhd_c v \}$. In this case the consequence of the rule ($\exists w.\ \{t \rhd_c w\} \subseteq C$) is already satisfied with $w = v$, and so this case need not be checked.

Consider the instance propagation rule. P is of the form $\{ q \preccurlyeq_i r, q \rhd_c s, r \rhd_c z \}$. When checking dirty instances, we have $\{ t \preccurlyeq_i u, t \rhd_c v \} \subseteq P$. The only possibility is that $P = \{ t \preccurlyeq_i u, t \rhd_c v, u \rhd_c z \}$ for some z. Since u and c are known, there can only be one possible value for w and it can be found by inspecting C, i.e., P is completely determined. When checking dirty sources, we have $\{ u \preccurlyeq_i t, t \rhd_c v \} \subseteq P$ and the only possibility is that $P = \{ u \preccurlyeq_i t, u \rhd_c s, t \rhd_c v \}$ for some s. Again u and c are known, so the value of s is determined.

Subsequent sections describe enhancements to the basic algorithm which introduce new rules, but in each case it is just as easy to determine how the variables of the rules are to be instantiated.

7.2.6 The Extended Occurs Check

It is easy to construct constraint sets for which this algorithm does not terminate. Furthermore, these sets do arise in practice.

For example, consider the set $\{ T_f \rhd_{result} T_r, T_f \preccurlyeq_i T_r \}$. This could arise from an analysis of the following program:

```
f() { return f; }
```

`f`'s result is an instance of `f`. (This is a contrived example. Real examples in Java are more complicated, e.g., a method M that returns a reference to a new object which contains M.)

Suppose we apply the above algorithm to this constraint set:

- Apply component propagation to $\{ T_f \rhd_{result} T_r, T_f \preccurlyeq_i T_r \}$:
  add T_1 and constraint $\{ T_r \rhd_{result} T_1 \}$
- Apply instance propagation to $\{ T_f \rhd_{result} T_r, T_f \preccurlyeq_i T_r, T_r \rhd_{result} T_1 \}$:
  add constraint $\{ T_r \preccurlyeq_i T_1 \}$
- Apply component propagation to $\{ T_r \rhd_{result} T_1, T_r \preccurlyeq_i T_1 \}$:
  add T_2 and constraint $\{ T_1 \rhd_{result} T_2 \}$
- Apply instance propagation to $\{ T_r \rhd_{result} T_1, T_r \preccurlyeq_i T_1, T_1 \rhd_{result} T_2 \}$:
  add constraint $\{ T_1 \preccurlyeq_i T_2 \}$
- …

In type inference, the type of `f` would be an infinite term:

$$\text{void} \rightarrow (\text{void} \rightarrow (\text{void} \rightarrow \ldots))$$

This recursive type is not valid in Henglein's scheme; therefore his algorithm detects this situation and reports failure. He calls this detection the "extended occurs check". (It is analogous to the occurs check performed during term unification.) In terms of the SEMI formalism, the extended occurs check fires whenever, for some sets of variables t_i and u_i:

$$\{ t_1 \preccurlyeq_{i1} u_1, \ldots, u_{n\text{-}1} \preccurlyeq_{in} u_n, t_1 \rhd_{comp1} t_2, \ldots, t_m \rhd_{compn} u_n \} \subseteq C$$

This means that the extended occurs check is applicable whenever we have a variable t_1 with a transitive instance u_n which is also transitively a component of t_1.

When the extended occurs check fires in SEMI, the solver simply forms a recursive type by adding the constraint $t_1 \cong u_n$, and continues. In the example, the extended occurs check detects the constraints { $T_f \rhd_{result} T_r$, $T_f \preccurlyeq_i T_r$ } and adds the constraint $T_r \cong T_f$, halting the expansion.

Note that adding this equality forces variables to be equal that do not necessarily need to be equal according to the initial constraints. This is why SEMI does not compute a most general (i.e., principal) solution. The demonstration of non-existence of principal types in Appendix A is based on a similar example.

The implementation of the SEMI solver performs an extended occurs check whenever the instance propagation rule adds a new instance constraint $t \preccurlyeq_i u$ to C. It sets $u_{n-1} = t$, $u_n = u$, and $i_n = i$, and then searches the component and instance graphs for a variable t_1 satisfying the check. Any such variables found are bound to u. The search proceeds by first scanning the instance graph backwards, finding all candidate t_1s that are transitive sources of t (including t itself), and for each candidate, scanning its components transitively looking for u.

This check could easily be changed from worst case O(N^2) time, where N is the number of variables, to O(N) time, simply by finding all transitive sources of *t* first, storing them in a hashtable-based set, then scanning all of *t*'s transitive parents (variables that have *t* as a transitive component) and testing for membership in the set. In practice, however, the average numbers of transitive instances, sources, components or parents that a variable has are all very large, and a check that is linear time in any of these quantities is prohibitively expensive (since the extended occurs check is performed frequently). Therefore SEMI uses a more complex approach, described below, which builds on the basic algorithm above. It turns out that with the help of those optimizations, the worst case O(N^2) version performs significantly better.

7.2.7 Nondeterminism

The algorithm presented here is nondeterministic, as are all the following elaborations and the implementation itself. There is always flexibility in choosing the order in which to remove constraints from the worklist. Different orderings can lead to different results of the algorithm, because the extended occurs check may fire at different times and induce different equality constraints.

The implementation also produces non-deterministic results because it is written in Java, and Java's semantics does not fully define the behavior of the implementation. In particular, the "identity hash code" of an object is not defined by the Java language specification. The identity hash code is returned by the default implementation of `Object.hashCode()`; the only requirement is that it always return the same value for any given object. When the same program is run multiple times on the same Java virtual machine implementation, the identity hash codes assigned to its objects are often observed to vary between runs. This leads to observable variations in behavior, because the enumeration order of the elements of hash tables and related data structures depends on the values of the identity hash codes.

In practice, Ajax almost always returns the same results for multiple runs of a given query.

7.3 Optimizing the Occurs Check: Clusters

The naïve approach to performing the extended occurs check can be sped up by exploiting the structure of constraints induced by a Java program (or any program that has layers in its architecture, i.e., almost all programs).

7.3.1 Constraint Structure

SEMI generates instance constraints from a Java program in the following situations:

- A method body M_1 makes a "static" call to another method M_2 (M_1 depends on M_2).
- A method body M_1 creates a new object of a class C (M_1 depends on C).
- A method body M_1 is installed in the dynamic dispatch table of a class C (C depends on M_1).

Due to the layered structure of most programs, the graph of dependencies is "mostly" acyclic. (However, the JDK class library itself contains a number of surprisingly complex cycles, so it is important to be able to handle cycles well.)

7.3.2 Clusters

Normally (i.e., in the absence of a cycle of mutually recursive dependencies), the variables associated with parameters, local variables, results, and intermediate values within a given method, and variables which are components of those variables, are related only by component constraints. Instance constraints (and **only** instance constraints) relate these variables to variables associated with other methods. Similarly, in a class there are variables associated with the method slots, and a variable for the prototype object of the class, which are related to each other by component constraints only. Instance constraints relate these variables to variables in the methods that create objects of the class, and to variables in the method bodies used by the class.

The SEMI solver explicitly captures this structure. The variables are partitioned into abstract *clusters*; the partition is written $R : V \rightarrow X$ (where X is the set of cluster labels). The only required property of R is that if $t \triangleright_c u$ is a constraint, then $R(t) = R(u)$. In other words, all variables related by only component constraints are in the same cluster. Typically, Java programs give rise to a large number of small clusters (one cluster per method).

It is not strictly necessary to have R be the most refined partition possible, but that is easy to implement and gives the best results. That is, if t and u are not related by any chain of component constraints, ignoring direction, then $R(t) \neq R(u)$.

The implementation maintains the cluster map dynamically, taking account of variable merging and the introduction of new constraints.

7.3.3 Optimizing the Extended Occurs Check Using Clusters

The cluster map is used to short-circuit the subroutine that computes "Is u a transitive component of t_1?" If $R(u) \neq R(t_1)$, then the result must be false. Since clusters are generally small and numerous, and following an instance constraint usually leads to another (different) cluster, $R(u) \neq R(t_1)$ almost always holds during the extended occurs check search.

7.3.4 Cluster Levels

Unfortunately, even scanning all transitive sources of a variable and performing a constant-time check for each is too expensive, given the frequency with which extended occurs checks are performed.

SEMI resolves this problem by explicitly capturing the "mostly acyclic" structure of the inter-cluster instance graph. The instance constraints are projected onto the clusters; i.e., the clusters are assembled into a directed graph G such that for each $t \preccurlyeq_i u$, $(R(t), R(u))$ is an edge in G. Then the graph is partitioned into strongly connected components, called *cluster levels*. This partition is written $S : X \rightarrow Z$, where Z is the set of cluster level labels. By definition, G projected onto cluster levels is acyclic (excluding self-loops). The fact that G itself is "mostly acyclic" means that most cluster levels contain just one cluster.

The implementation maintains the cluster levels dynamically, as the underlying constraint system changes. SEMI does this efficiently, but the implementation is tricky because detecting cycles can be expensive. It is helpful to delay cycle detection until the cluster levels are required to be in a consistent (acyclic) state (i.e., until the next extended occurs check). SEMI maintains a "dirty" bit for each cluster level, indicating that it may be part of a cycle of cluster levels because of the addition of new instance constraints incident to the cluster level. When acyclicity is required, the algorithm performs a worst-case linear time traversal of the cluster level graph — a depth-first search backwards along the instance edges, starting from the dirty cluster levels. Any cycles found are recorded. Finally, the cluster levels in each cycle are merged. It requires care to make sure that **all** cycles are detected, since the straightforward depth-first search algorithm for cycle detection is only guaranteed to find one cycle (assuming a cycle exists).

In SEMI, the cost of maintaining the cluster levels is usually negligible and never the performance bottleneck.

7.3.5 Optimizing the Extended Occurs Check Using Cluster Levels

The cluster level map is used to optimize the subroutine that scans the source graph for all candidate t_1s that are transitive sources of t.

The extended occurs check subroutine receives t and u where u is an instance of t. Therefore every candidate t_1 has u as a transitive instance. Now suppose for some candidate t_1, $S(R(u)) \neq S(R(t_1))$. There must be a path from $S(R(t_1))$ to $S(R(u))$ in the instance graph projected onto the cluster levels, because there is a path from t_1 to u in the instance graph. Because the cluster level instance graph is acyclic, there cannot be a path from $S(R(u))$ to $S(R(t_1))$. Therefore, for all transitive sources s of t_1, $S(R(s)) \neq S(R(u))$ and therefore $R(s) \neq R(u)$, because otherwise we would have an instance path from $S(R(s)) = S(R(u))$ to $S(R(t_1))$.

Therefore, whenever the extended occurs check subroutine detects $S(R(u)) \neq S(R(t_1))$, t_1's sources need not be searched. In practice this prunes the search tremendously. In particular, if $S(R(u)) \neq S(R(t))$ then neither t nor its sources need be checked; the entire check takes constant time.

In the special case in which there are no recursive dependencies in the original program, the instance graph projected onto clusters is acyclic, i.e., S is one-to-one. Then the extended occurs check always completes in constant time. In other words, this optimization ensures that the extended occurs check only incurs a cost (apart from the cost of maintaining the clusters and cluster levels) when polymorphic recursion is actually being used.

7.3.6 Replacing the Extended Occurs Check with a Conservative Approximation

In the case $S(R(u)) = S(R(t))$, instead of performing the rest of the extended occurs check, one could simply add the equality constraint $t \cong u$. The new instance constraint $t \preccurlyeq_i u$ is reduced to a self-loop in the instance graph, which forestalls the nonterminating behavior that the extended occurs check is designed to prevent. This approach is similar to the Hindley-Milner algorithm, which (interpreted in this context) prohibits any polymorphism constraints within a cluster level. This behavior can lead to smaller constraint sets because of the “unnecessary” equalities that are introduced, which improves performance but does yield a noticeable decrease in accuracy for some applications of the analysis.

7.4 Scheduling the Worklist Using Cluster Levels

It turns out that the acyclic cluster level graph is useful for tasks other than optimizing the extended occurs check.

7.4.1 The Scheduling Problem

Components propagate from sources to instances, but not the other way around. Therefore as changes are made to constraints at the “bottom” of the instance graph, they tend to “bubble up” to instances. It improves performance to do as much work as possible at the bottom of the instance graph before making changes further up the graph, by reducing the number of times each component is visited or examined.

7.4.2 Using Cluster Levels

A cluster level l is “dirty” if there is a component constraint in the worklist of the form $t \rhd_c u$, where $S(R(t)) = l$.

Whenever SEMI chooses a component constraint from the worklist, it chooses a constraint $t \rhd_c u$ where the cluster level $S(R(t))$ has no dirty cluster levels below it in the instance graph projected onto the cluster levels. Such a constraint is guaranteed to exist because the cluster level instance graph is acyclic.

Making this choice efficiently is tricky, but requires negligible time and space in the SEMI implementation. The dirty component constraints are stored on the worklist indexed by cluster levels; the problem reduces to finding an appropriate cluster level to work on. SEMI explicitly records the dirtiness of each cluster level. It also caches two facts in each cluster level: whether it is known that there is *at least one* dirty cluster level below it in the cluster level instance graph, and whether it is known that there are *no* dirty cluster levels below it in the graph. In practice, this cache can be updated and invalidated efficiently in response to changes in dirty state and changes in the underlying constraint set.

The system keeps a list of dirty cluster levels, separated into two parts: the set of dirty cluster levels that are known to have no dirty cluster levels below them on the projected instance graph (the “ready list”), and the rest (the “blocked list”). When a constraint is selected from the worklist, if the ready list is non-empty then a cluster level is chosen from it and one of the cluster level’s dirty constraints is selected.

If the ready list is empty, then a cluster level *l* is chosen from the blocked list. The algorithm performs a depth-first search of the cluster level instance graph, backwards from *l*, from instances to sources. During this search, each visited cluster level is marked as either having dirty cluster levels below it, or not. If not, then the visited cluster level is moved from the blocked list to the ready list. The acyclicity of the cluster level instance graph guarantees that after this procedure, at least one dirty cluster level will be found with no dirty cluster levels below it (unless there are no dirty cluster levels left, in which case the algorithm terminates).

7.5 Suppressing Components: Advertisements

7.5.1 Useless Component Propagation

Suppose F is a function in the program for which we infer a large “type”, T_F. This means that T_F is the root of a large graph of component constraints. At every use of F (a direct call or the use of F to fill a slot in a method table), a new instance *i* of T_F is created, and a constraint $T_F \preccurlyeq_i t$ is added. The component propagation rule will effectively copy the transitive components of T_F (i.e., the component graph under T_F) to the instance. Often, however, much of this structure will not be used. For example, consider this Java code:

```
Foo x = bar();
println(x.kitty);
```

Given the code for `bar`, the analysis may work out some complex type structure for its return value, including information about the various methods and fields of `x`. All this information will be propagated to the caller, but only one field is used, and therefore the rest of the information is irrelevant.

Furthermore, suppose `bar` is implemented as a wrapper:

```
Foo bar() { return baz(5); }
```

Such constructs are common, and defeat purely local schemes for suppressing useless structure.

7.5.2 Illustration

Consider the constraint set Q shown in Figure 7-1. This diagram and the diagrams that follow represent constraint sets as graphs. Nodes correspond to variables. A constraint of the form $t \rhd_c u$ is displayed as a solid edge from *t*’s node to *u*’s node labelled with $\rhd_c$. A constraint of the form $t \preccurlyeq_i u$ is displayed as a dotted edge from *t*’s node to *u*’s node labelled with $\preccurlyeq_i$.

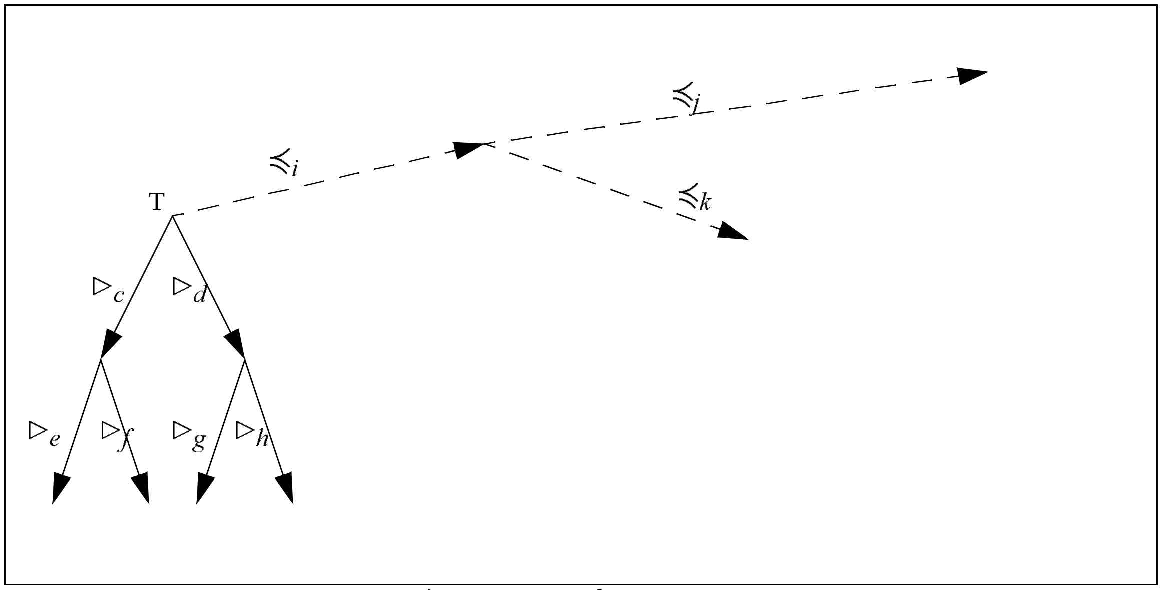


Figure 7-1. Initial constraint set

T represents the type of some compound object with an instance *i* and further instances *j* and *k*. Assume Q contains the initial constraint set, C_I. The basic algorithm extends Q to the closed set C shown in Figure 7-2.

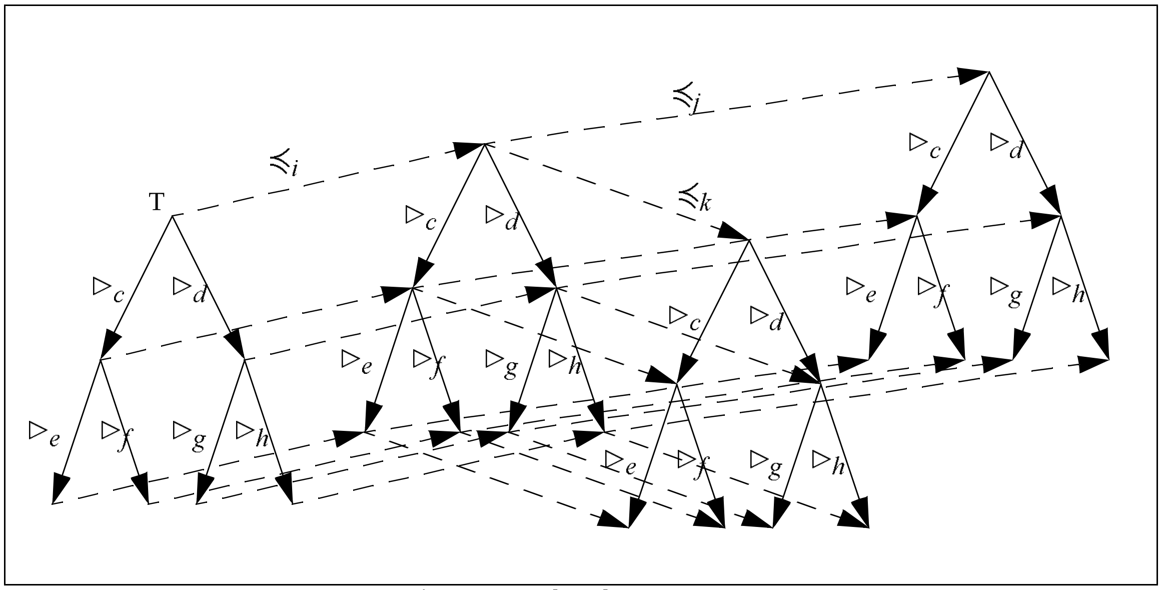


Figure 7-2. Closed constraint set

The basic algorithm reaches C by copying T's component tree to all the instances, and connecting the components with instance relationships.

7.5.3 Quasi-closure Conditions

These new components are all unnecessary — Q is, in fact, quasi-closed. To see this, consider two variables in C_I, *u* and *v*. We must show that *u* and *v* are related in Q if and only if they are related in C.

The notation “$u \lesssim v$” means that there is a chain of instance constraints from u to v.

There are two cases:

- Suppose u and v are not related in C. Then $\neg\exists x.\ u \lesssim_C x \wedge v \lesssim_C x$. It follows that $\neg\exists x.\ u \lesssim_Q x \wedge v \lesssim_Q x$, since C is a superset of Q. Therefore u and v are not related in Q.
- Suppose u and v are related according to C. Then $\exists x.\ u \lesssim_C x \wedge v \lesssim_C x$. We show that $\exists p.\ u \lesssim_Q p \wedge v \lesssim_Q p$, by induction on the length of the shortest chain of instances justfiying $u \lesssim_C x$.
  Regardless of the length of the chain, if x occurs in Q, then $u \lesssim_Q x \wedge v \lesssim_Q x$, since the chains of instances justifying $u \lesssim_C x$ and $v \lesssim_C x$ are also in Q. (In other words, every instance constraint in C that holds between variables in Q is is already in Q.) Thus the induction hypothesis holds, setting $p = x$.
  If the length of the chain is zero, then $x = u$, hence x is in Q and the hypothesis holds.
  If x is not in Q, then it must be a child variable of one of the new component constraints. Each such variable has a unique predecessor P_x in C such that $P_x \lesssim x$. The chains $u \lesssim_C x$ and $v \lesssim_C x$ must have length at least one, since x is not in Q and therefore does not equal u or v. Therefore the last link of each chain must be $P_x \lesssim x$. Therefore, $u \lesssim_C P_x \wedge v \lesssim_C P_x$ also holds. By the induction hypothesis, $\exists p.\ u \lesssim_Q p \wedge v \lesssim_Q p$.

This argument can be generalized. A general set Q is quasi-closed over C_I if:

1. Equalities have been eliminated from Q, and it is closed under the instance and component consistency rules (guaranteed by my representation).
2. Q contains C_I.
3. Q is closed under the instance propagation rule.
4. For all t, u, v, c, x, y, if $t \lesssim_Q u \wedge u \lesssim_Q v \wedge \{\ t \rhd_c x,\ v \rhd_c y\ \} \subseteq$ Q, then there is a w such that $\{\ u \rhd_c w\ \} \subseteq$ Q.
5. For all t, u, c, v, if $t \lesssim_Q u \wedge \{\ t \rhd_c v\ \} \subseteq$ Q but $\{\ u \rhd_c w\ \}$ is not in Q for any w, then the set $\{\ x \mid \exists j, w, y.\ y \lesssim_Q u \wedge \{\ x \lesssim_j y, x \rhd_c w\ \} \subseteq Q \wedge \neg(\exists z.\ \{\ y \rhd_c z\ \} \subseteq Q)\ \} = \{\ t\ \}$.

Conditions 1 and 2 are fundamental. Conditions 3 and 4 are required to justify the “x in Q” part of the proof; they require Q to be closed except possibly for some unexpanded instances of compound structures. Condition 5 is required to justify the “x not in Q” part of the proof; it ensures that if a component c is not propagated to u, then there is a unique instance-chain predecessor that has a real component that we can fall back to.

7.5.4 Advertisements

The system reaches this state by propagating components lazily. When the component propagation rule fires, it actually propagates an *advertisement*, representing the possibility of a component being present in the instance. An advertisement is a pair: the parent variable, v, and a component label, c, written $v \trianglerighteq_c$. These advertisements are propagated along the instance graph using two rules:

- **Advertisement propagation from component**
  Upon detecting $\{\ t \lesssim_i u,\ t \rhd_c v\ \} \subseteq$ C for some t, u, v, i and c, add $u \trianglerighteq_c$.

- **Advertisement propagation from advertisement**
  Upon detecting $\{ t \preccurlyeq_i u, t \trianglerighteq_c \} \subseteq C$ for some t, u, i and c, add $u \trianglerighteq_c$.

If a variable t already has a component c, then it does not need an advertisement for the same component.

- **Redundant advertisement suppression**
  Upon detecting $\{ t \trianglerighteq_c, t \triangleright_c v \} \subseteq C$ for some t, v and c, delete $t \trianglerighteq_c$.

These rules replace the component propagation rule. They guarantee that quasi-closure conditions 1, 2 and 3 hold upon termination.

7.5.5 Example

Consider Figure 7-3. Instead of copying T’s entire component tree, we have added advertisements for T’s immediate components.

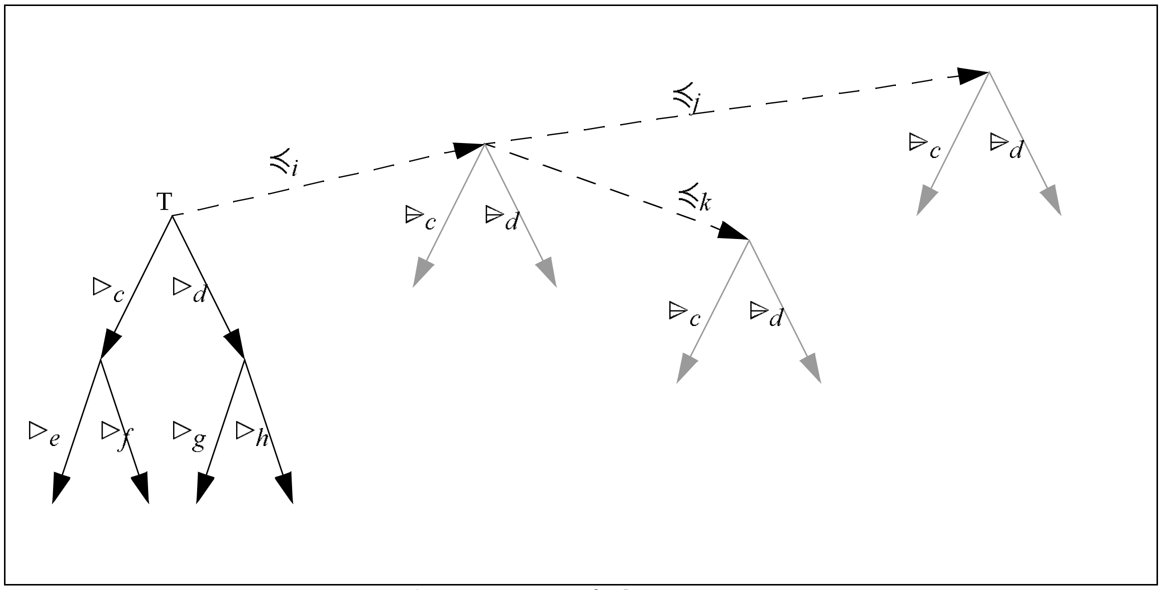


Figure 7-3. Use of advertisements

7.5.6 Ensuring Quasi-closure: Fill-in

To satisfy quasi-closure condition 4, the algorithm “fills in” an advertisement that has a real component above it in the instance graph:

- **Advertisement fill-in**
  Upon detecting $\{ t \preccurlyeq_i u, t \trianglerighteq_c, u \triangleright_c w \} \subseteq C$ for some t, u and w, add $t \triangleright_c v$, where v is a fresh variable.

For example, consider the initial set shown in Figure 7-4.

SEMI adds an advertisement between T and U, as shown in Figure 7-5. The fill-in rule will ensure that the advertisement is replaced with a real component, as shown in Figure 7-6. The instance propagation rule will then ensure that the instance chain from T_c to U_c is completed, as shown in Figure 7-7.

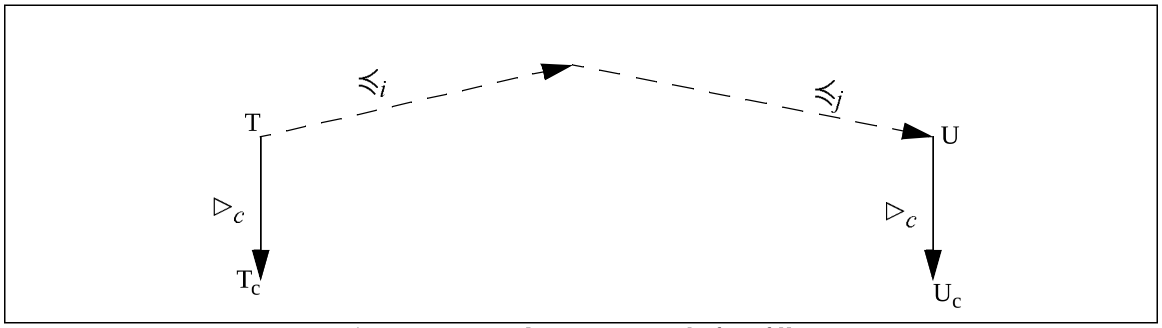


Figure 7-4. Initial constraint set before fill-in

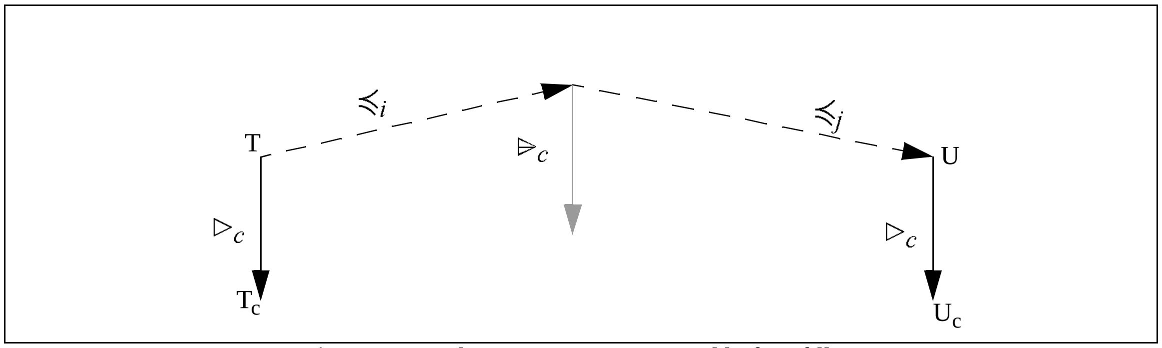


Figure 7-5. Advertisement constructed before fill-in

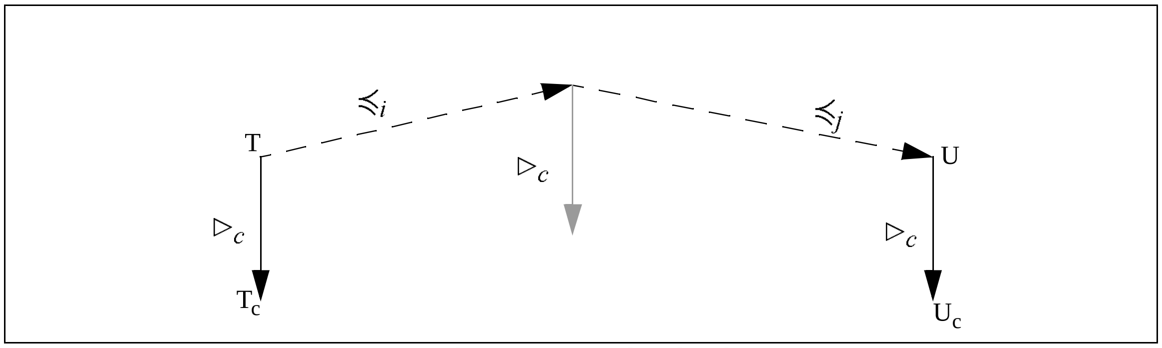


Figure 7-6. Advertisement replaced with component

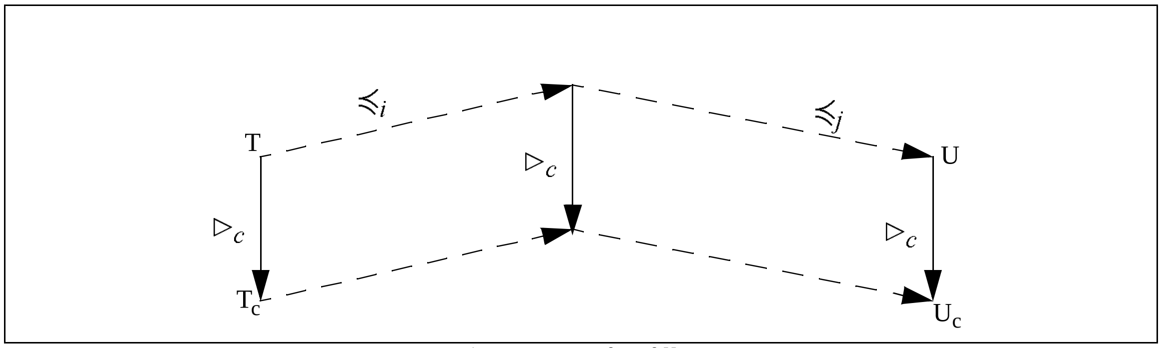


Figure 7-7. After fill-in

7.5.7 Ensuring Quasi-closure: Detecting Conflicting Sources

To satisfy quasi-closure condition 5, each advertisement is associated with an *advertisement source*, *s*, that records the variable the advertisement is derived from. The advertisement is written $t \trianglerighteq_c [s]$. Quasi-closure condition 5 becomes the "unique source condition":

If the advertisement $u \trianglerighteq_c [s]$ exists, then
$\{ x \mid \exists j, w, y.\ \{ x \preccurlyeq_j y, y \preccurlyeq_C u, x \triangleright_c w \} \subseteq C \wedge (\forall z.\ \text{"}y \triangleright_c z\text{"} \notin C) \} = \{ s \}$.

The advertisement rules are extended:

- **Advertisement propagation from component**
  Upon detecting $\{ t \preccurlyeq_i u, t \triangleright_c v \} \subseteq C$ for some t, u, v, i and c, add $u \trianglerighteq_c [t]$.
- **Advertisement propagation from advertisement**
  Upon detecting $\{ t \preccurlyeq_i u, t \trianglerighteq_c [s] \} \subseteq C$ for some t, u, s, i and c, add $u \trianglerighteq_c [s]$.
- **Redundant advertisement suppression**
  Upon detecting $\{ t \trianglerighteq_c [s], t \triangleright_c v \} \subseteq C$ for some t, v, s and c, delete $t \trianglerighteq_c [s]$.
- **Advertisement fill-in**
  Upon detecting $\{ t \preccurlyeq_i u, t \trianglerighteq_c [s], u \triangleright_c w \} \subseteq C$ for some t, u, s and w, add $t \triangleright_c v$, where v is a fresh variable.

When a conflict arises — two advertisements for the same component show different sources — we collapse the advertisements and make a real component.

- **Conflicting advertisement detection**
  Upon detecting $\{ t \trianglerighteq_c [s], t \trianglerighteq_c [r] \} \subseteq C$ for some t, s, c and r, where $r \neq s$, create a new w and add $t \triangleright_c w$.

This rule tests for the inequality of two variables. This can be tricky because variables can become equal during the run of the algorithm, but in fact it only means that conflicts may be detected that in the end may not be "true" conflicts. Since replacing an advertisement with a real component is always a conservative operation (possibly hurting performance, but never correctness), this is not a problem.

The conflicting advertisement rule guarantees that upon termination, the unique source condition is satisfied.

7.5.8 Simple Example

For example, consider the C_I in Figure 7-8.

The algorithm propagates advertisements from U and T to V, but since U ≠ T, the conflict detection rule fires and a real component is created for V. This is necessary to make the result quasi-closed.

7.5.9 Advertisement Source Updates

The conflicting advertisement detection rule alone is not satisfactory, however. Consider the example in Figure 7-9.

Suppose the algorithm propagates an advertisement from T to V and then W, and then propagates an advertisement from U to V. (This schedule might be chosen because of additional constraints not shown.) Now at V there are conflicting advertisements, with sources U and T. The algorithm creates a real component at V. The resulting state is shown

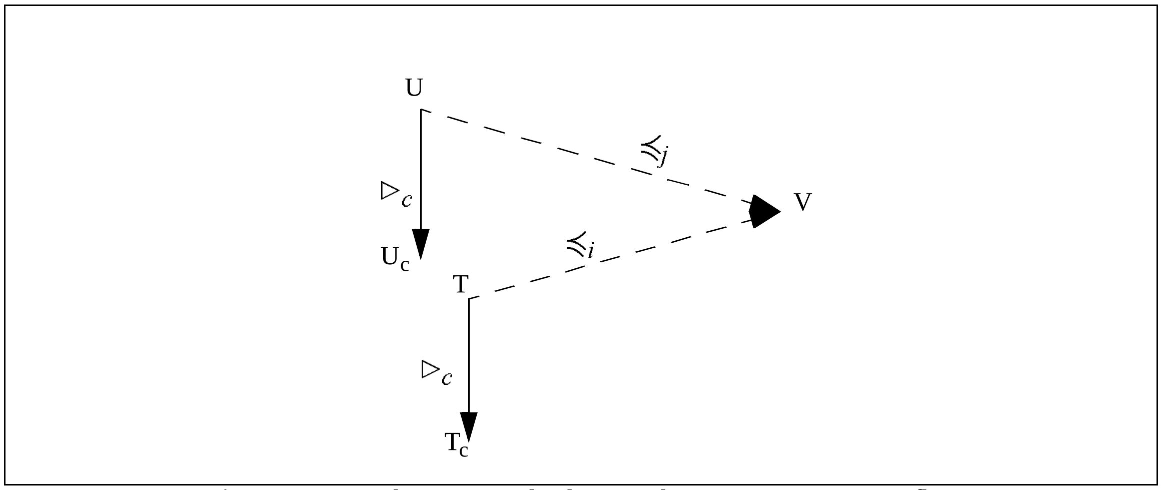


Figure 7-8. Initial constraints leading to advertisement source conflict

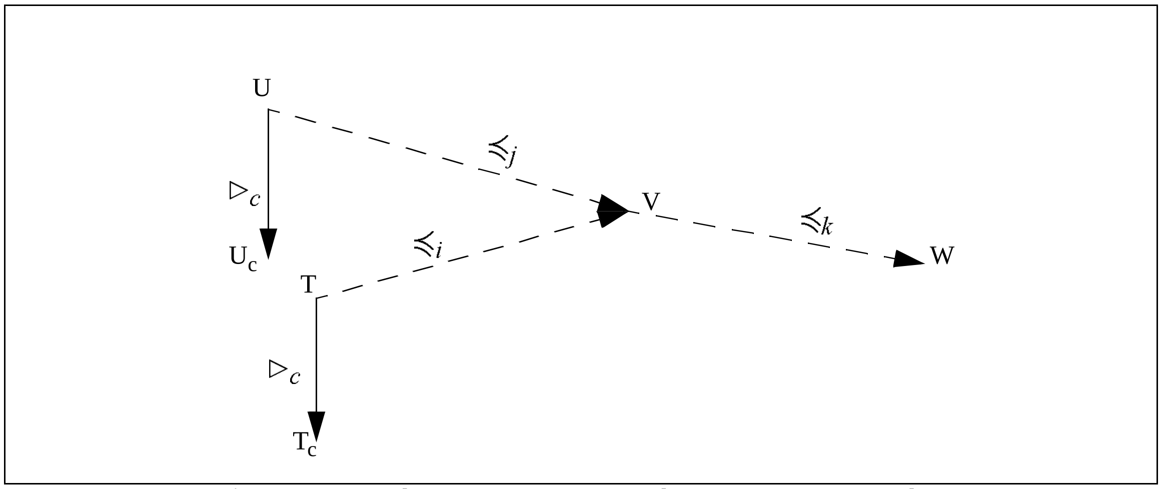


Figure 7-9. Initial constraints requiring advertisement source update

in Figure 7-10. Next the algorithm propagates an advertisement for that component to W. Now there are conflicting advertisements at W, with sources T and V, so a new component must also be created at W. This is suboptimal because W could simply have an advertisement with source V.

To avert such situations, it suffices to destroy the advertisements that could be affected by a new component; they will be regenerated with correct source information, if possible.

- **Advertisement source update**
  Upon detecting $\{ t \unrhd_c [s], y \rhd_c z \} \subseteq C$ for some t, s, y, z and c, where $s \preccurlyeq_C y, y \preccurlyeq_C t$ and $s \neq y$, delete "$t \unrhd_c [s]$".

7.5.10 Implementation

Advertisement constraints are easily added by treating them as a degenerate kind of component. Propagation and fill-in detection are implemented by allowing advertisements as well as components to be on the dirty worklist. Conflicting advertisement detection is straightforward to implement and is done eagerly.

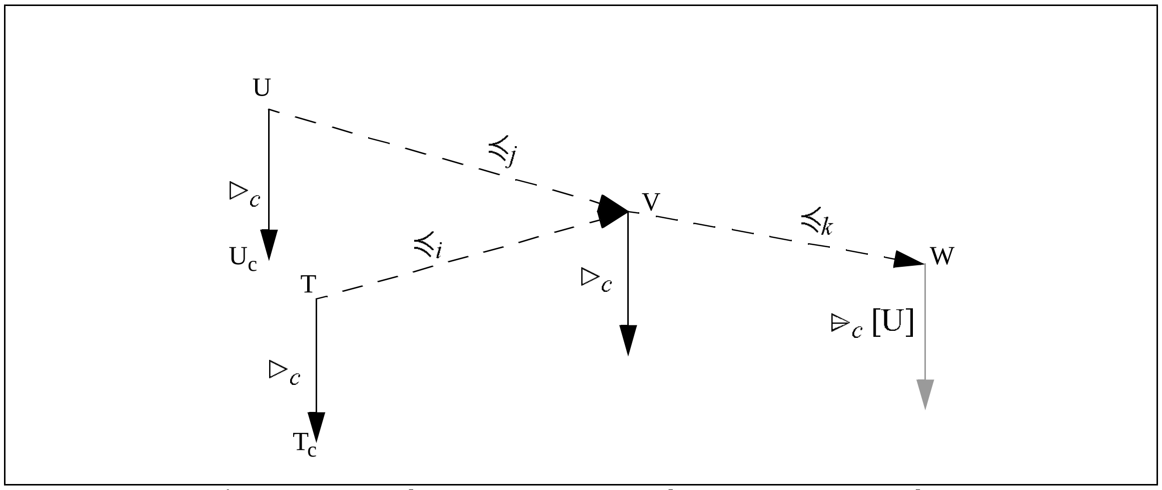


Figure 7-10. Initial constraints requiring advertisement source update

The advertisement source update is difficult to implement efficiently. The straightforward implementation can destroy and recreate many advertisements each time a component is added. SEMI uses an alternative representation for the source field of an advertisement. An advertisement for *c* at *t* records a "bottleneck variable" *v* such that every instance chain from the true source *s* to *t* passes through *v*. *v* may be *s*, or it may be some instance of *s*, in which case *v* also has an advertisement for *c* (and its own bottleneck variable, etc). The true source *s* for *t* can be found quickly; it is either *v*, or it is *v*'s true source. When *v* is not *s*, components may be added along the path from *s* to *v* without having to update the information cached in the advertisement at *t*.

7.6 Globals

7.6.1 Handling Program Global Variables

It is straightforward to encode a program's global variables ("static fields" in Java) in the constraint system presented. They can be treated as a single "globals" object with one field for each variable, which is passed into each function as a parameter. However, this is not very efficient because globals information must be copied into each method type. It is much more efficient, and no less accurate, to have just one variable representing the globals object and one copy of the information for the global variables. Lemma 6-21 shows that this is no less accurate. The lemma states that the information inferred for the globals object in any context is always the same.

7.6.2 Characterization of Constraints for Globals

In terms of the constraints, a constraint variable v in an initial set C_I can be said to be *global* if, for all closed sets C containing C_I, $\exists g. \forall y.\ v \preccurlyeq_C y \Rightarrow y \preccurlyeq_C g$. This means that there is a "top level" constraint variable g representing all instances of the global data. Lemma 6-21 shows that the constraint variables corresponding to static fields in the bytecode have this property.

It is easy to see that an instance of a global constraint variable is also global. Furthermore, a component of a global constraint variable is also global, because all instance chains propagate down the component constraint.

Suppose that global constraint variables t and u are related according to the VPR approximation derived from a quasi-closed constraint set (Section 7.1.3). Then $\exists x.\ t \preccurlyeq_C x \wedge u \preccurlyeq_C x$. Choose g such that $\forall y.\ t \preccurlyeq_C y \Rightarrow y \preccurlyeq_C g$. Then $x \preccurlyeq_C g$, and therefore $u \preccurlyeq_C g$. This implies that g and u are related according to the VPR. Thus, t's global representative g behaves identically to t in the VPR. We can unify all global constraint variables with their global representatives without changing the derived VPR.

7.6.3 Implementation

SEMI marks constraint variables corresponding to static Java variables as global and gives these constraint variables special treatment:

- If $t \rhd_c v$ and t is global then v is marked global.
- When a global constraint variable is unified with another constraint variable, the resulting variable is marked global.
- If $t \preccurlyeq_i u$ and t is global, then the algorithm sets $t \cong u$ and deletes the instance constraint. This leads to u being marked global.
- Global variables do not belong to any cluster or cluster level. The cluster invariant is modified to "if $t \rhd_c v$ and v is not global then t and v belong to the same cluster". The scheduler keeps a separate list of dirty constraints on global variables and always processes them last, when no dirty clusters are available.

7.6.4 Exceptions

SEMI encodes exceptions thrown by methods as auxiliary result components of method types. In real Java programs, as far as SEMI can tell any exception thrown by a method may propagate to the top level. (This is because catch clauses that catch all exceptions always rethrow the caught exception, and in the case of selective catch clauses SEMI cannot distinguish between the exceptions that are caught and the exceptions that are not caught.) This means that variables corresponding to thrown exceptions (or their components) satisfy the same constraint property given above for variables corresponding to global data. Therefore SEMI uses the "globalization" optimization for variables corresponding to thrown exceptions. This technique causes no loss of precision, and in practice the savings in space and time are significant.

7.7 A Failed Optimization: Cut-throughs

7.7.1 Example

Consider the following program:

```
Foo f1() { return new Foo(); }
Foo f2() { return f1(); }
Foo f3() { return f2(); }
… f3() …
```

Any necessary components of the new `Foo` will be propagated to the call site for `f3`. The variables corresponding to the results of `f2` and `f1` will also get copies of the components. This is unsatisfying because handling these semantically meaningless layers of abstraction could exact a significant cost in time and space for the solver.

7.7.2 Cut-throughs

I attempted to resolve this problem by introducing a notion of a "cut-through instance": a single instance constraint that summarizes a chain of instance constraints. In the example, a single cut-through instance could connect the result of "`new Foo`" with the result of `f3`. This meant that the components of the object need not be expanded in the results of `f2` and `f1`.

It was very difficult to implement. A large amount of bookkeeping was required to ensure consistency, and it was tricky to implement efficiently. To make the implementation tractable, I had to carefully restrict the circumstances in which cut-through edges could be used. Unfortunately, experiments showed that on real examples cut-through instances were hardly ever being used. I do not recommend introducing this style of optimization, and SEMI does not perform it.

7.8 Reducing the Number of Initial Constraints

7.8.1 Dynamic Method Call Resolution

In SEMI, "virtual" method calls are usually more costly to treat than static method calls because the inferred type of the method will often be copied into the types of many objects. Therefore it is advantageous to apply a preprocessing step to reduce as many dynamic method calls as possible to static ones. This is implemented in SEMI by allowing an Ajax analysis to be specified as an optional parameter; SEMI will issue a query using this analysis, and use the results to resolve as many dynamic method calls as possible.

For this strategy to be useful, the subordinate analysis should be significantly cheaper than SEMI. My experiments use RTA++ for this purpose.

Ajax provides incremental updates to the results of an analysis. For a dynamic method call resolution query, this means that a call site with multiple possible callees will initially be reported as "dead" (callee set is empty), then reported as "statically resolvable" (callee set is a singleton), and then reported as "unresolvable" (callee set has two or more elements). Because SEMI does not support revocation of constraints, if it were to observe the "statically resolvable" state and immediately add appropriate constraints for static method invocation, it would then not be able to revoke them if the state changed in the future to indicate "unresolvable". This would not harm correctness, but it would reduce accuracy. To avoid this problem, the subordinate analysis is run to completion before SEMI uses its results.

This technique also improves both performance and accuracy. Accuracy improves because the statically resolved method call is treated polymorphically rather than monomorphically.

7.8.2 Lazy Method Slot Stuffing

The initial constraints install an instance of each method implementation's signature into the signature for each class C which uses that method implementation. The SEMI implementation delays installing such an instance until it has been determined that that class's method slot may actually be used, i.e., an `invokevirtual` instruction calls the appropriate method on a class that is a superclass of (or equal to) C. Thus, nonstatic methods of a class which are not actually called will usually not contribute to C's inferred type information; this vastly reduces the amount of work for SEMI.

The determination of which nonstatic methods may actually be called takes advantage of the information recovered for dynamic method call resolution.

7.8.3 Instance Suppression

If a polymorphic value in the program has only one instance, one loses no accuracy by treating it as if it were not polymorphic. Suppose the label for the instance is *i*. Then all instance constraints labelled *i* can be replaced with equality constraints. This can greatly reduce the number of variables and constraints in the system. This optimization is used in the following situations:

- Instructions with only one predecessor in the control-flow graph for their method need not be treated polymorphically. This provides a vast saving.
- Methods called from only one call site, where the callee is statically known, need not be treated polymorphically. The information required to implement this is gathered in much the same way as for dynamic method call resolution, discussed above.
- Classes created at only one site need not be treated polymorphically.

7.8.4 Disabling Intra-method Polymorphism

As mentioned in Section 6.3.8, control transfers within a method are modelled as function calls, and instructions at control flow merge points can be treated as polymorphic functions with multiple callers (one caller for each incoming control flow path). In practice, however, allowing such instructions to be treated polymorphically provides little or no accuracy benefit, and imposes a significant burden on performance. Therefore I have turned this option off for all my experiments; all control transfers are treated non-polymorphically.

7.8.5 Structural Shortcuts

In the formal presentation, I have sets of variables for the stack (S), local variable file (L), and global variable table (G). The former two sets of variables can be (and are) eliminated, along with the component constraints binding them to particular stack and local variable elements, by "pre-solving" those constraints. In the implementation this amounts to a form of def-use analysis, and greatly reduces the number of constraints generated. (However, since these constraints are always local to a method, the overall performance impact may be limited.) This optimization is performed even when intra-method polymorphism is enabled; in that case, the constraint generator “manually” adds the correct instance constraints that would have been propagated from the constraints on the Ss and Ls.

The globalization optimization described above in Section 7.6 facilitates the removal of explicit variables and constraints for the global variable table. Variables for individual globals are resolved directly to their top-level variables, and no constraints involving the Gs need be recorded.

7.9 Reducing the Number of Inferred Constraints

7.9.1 Component Partitioning

Consider a Java class C with a number of (possibly inherited) fields or methods, and a constraint variable *v*, which in some traces corresponds to objects of class C. The variable *v* may have a number of component constraints, as illustrated in Figure 7-11. Each component constraint generates an advertisement at each instance.

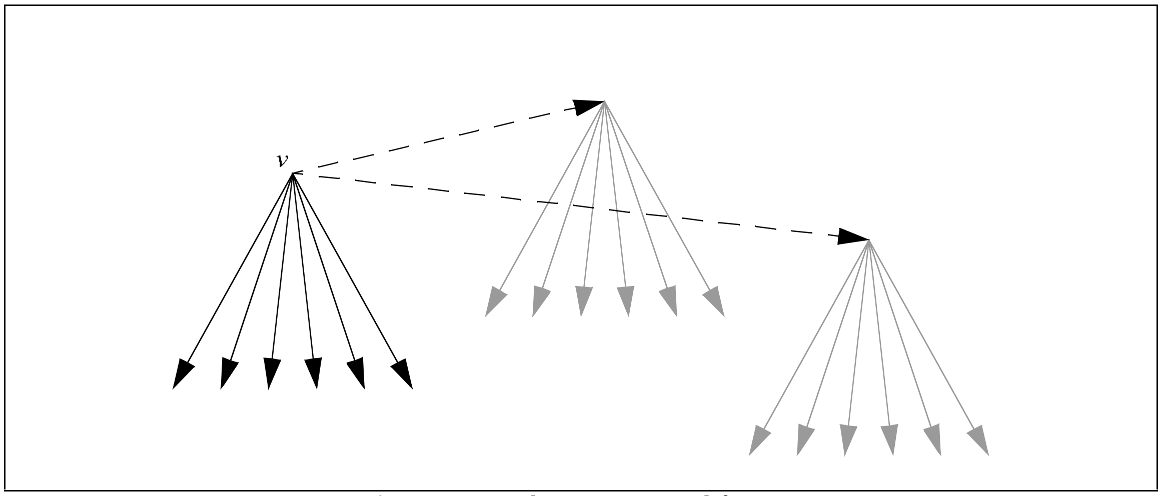


Figure 7-11. Advertisement proliferation

Suppose we partition the fields of C. We then replace a direct component constraint for a field with a pair of constraints, one identifying the partition, and one identifying the actual field within the partition. Continuing the above example, suppose that there are two equal-sized partitions. The result is shown in Figure 7-12.

If a single partitioning scheme is used consistently everywhere, the results obtained will be identical to those obtained by the simple constraint system. As this example shows, the partitioned component constraints may require fewer advertisements to be generated, although more component constraints are required.

A simple and natural partitioning scheme is to have one partition for each Java class and assign the component constraint for a field or method to the class in which that field or method is declared. A more elaborate scheme would be to form a hierarchy of partitions corresponding to the class hierarchy of the program.

Section 9.5.4 compares performance results for the different schemes. The simple partitioning scheme is superior to the elaborate scheme, and is also superior to no partitioning.

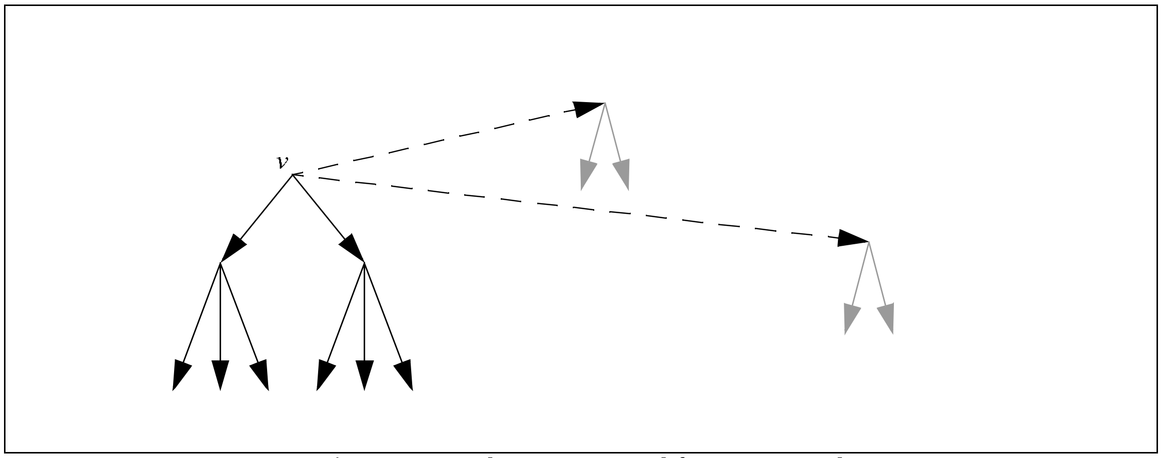


Figure 7-12. Advertisement proliferation averted

7.10 Suppressing Components: Modality

7.10.1 Example

Consider the following Java code:

```
Foo x = b ? new Bar() : new Baz();
println(x.kitty);
```

The advertisement algorithm does not perform well on this code. Consider Figure 7-13. Suppose T_x is the constraint variable associated with `x`. For each dynamically dispatched method *m* defined in both classes `Bar` and `Baz`, T_x will get two advertisements for component *m*, one from `Bar` and one from `Baz`. If the method implementations are different, then the advertisements will have conflicting sources, so the structure of the method's inferred type will be expanded (forming the unification of the types of `Bar`'s *m* and `Baz`'s *m*). This can result in a large number of unnecessary constraints.

7.10.2 Approach

SEMI annotates component constraints with *mode* information indicating how that component is used. A component constraint is written $t \triangleright_c^- u$, $t \triangleright_c^c u$, $t \triangleright_c^d u$, or $t \triangleright_c^{cd} u$. The superscript “c” means that the component is used in “constructor” mode. The superscript “d” means that the component is used in “destructor” mode. The superscript “-“ means that the component is not used in any mode. “cd” means that the component is used in both modes.

The idea comes from the realm of functional languages. In that domain, component constraints are associated with the use of type constructors, such as the arrow type for functions. The type rules for these languages have two forms: one form that introduces a new occurrence of the constructor (“constructor mode”), e.g., the “lambda” rule for creating a new function, and another form that eliminates an occurrence of the constructor and uses the components (“destructor mode”), e.g., the “app” rule for applying a function. The intuition I rely on is that if a component is not used in both constructor and destructor modes, then no useful information is transmitted through it. For example, if a function type

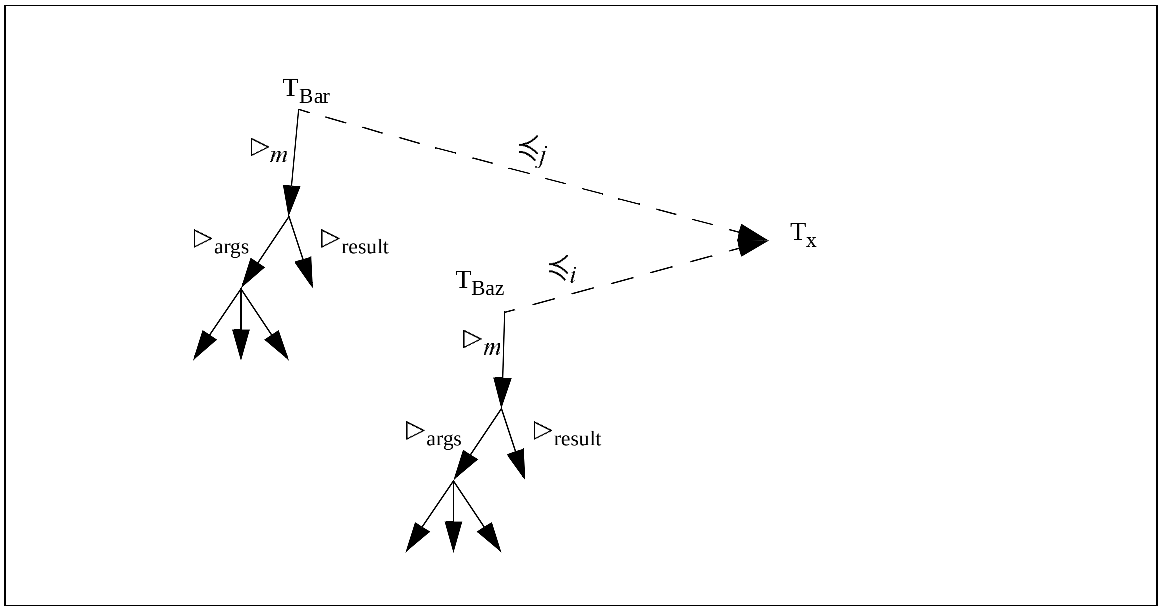


Figure 7-13. Constraint Structures Leading to Excessive Merging

is introduced through the "lambda" rule but is never subject to the "app" rule, then it does not matter what its components are. Similarly, if there is an "app" with no corresponding "lambda" then the components do not matter. (In this case, the code performing the application must be dead.)

When SEMI gathers constraints from the original Java bytecode program, it adds mode annotations to the component constraints as follows:

- Installing a method implementation into a new object type adds a component constraint in constructor mode.
- Calling a virtual method in an object type adds a component constraint in destructor mode.
- Writing a field of an object type adds a component constraint in constructor mode.
- Reading a field of an object type adds a component constraint in destructor mode.
- Calling a method adds parameter and result component constraints to the method type in destructor mode.
- Declaring a method adds parameter and result component constraints to the method type in constructor mode.

This mode information changes the interface to the solver and its specification. The relevant change is in the definition of closure. The following parts of the definition of closure are altered:

- **Component propagation rule**
  Components propagate through instances, with nondecreasing modes:
  $\{ t \preccurlyeq_i u, t \rhd_c^m v \} \subseteq C \Rightarrow \exists w, m'. \{ u \rhd_c^{m'} w \wedge m \subseteq m' \} \subseteq C$

The benefit of modes is that we can safely inhibit some instance propagation.

- **Instance propagation rule**
  $\{\ t \preccurlyeq_i u,\ t \rhd_c^{m} v,\ u \rhd_c^{m'} w\ \} \subseteq \mathrm{C} \wedge (\exists y, z.\ u \preccurlyeq_{\mathrm{C}} y \wedge \{\ y \rhd_c^{\mathrm{cd}} z\ \} \subseteq \mathrm{C}) \Rightarrow \{\ v \preccurlyeq_i w\ \} \subseteq \mathrm{C}$
  The instance constraint is only propagated to the component if there is some transitive instance of the component constraint that is used in both constructor and destructor mode. Otherwise the instance constraint need not be propagated.

7.10.3 Solver Rules

The solver rules given in previous sections remain in force. Rules that match a component constraint match any mode annotation. Rules that add component constraints add constraints with the "no mode" annotation. We introduce a separate rule to propagate annotation information:

- **Mode propagation**
  Upon detecting $\{\ t \preccurlyeq_i u,\ t \rhd_c^{m} v,\ u \rhd_c^{m'} w\ \} \subseteq \mathrm{C}$ for some t, u, v, i, c, w, m and m', replace "$u \rhd_c^{m'} w$" with "$u \rhd_c^{m \cup m'} w$".
- **Instance propagation**
  Upon detecting $\{\ t \preccurlyeq_i u,\ t \rhd_c^{m} v,\ u \rhd_c w\ \} \subseteq \mathrm{C}$ for some t, u, v, w, i, c, and m, if $\exists y, z.\ u \preccurlyeq_{\mathrm{C}} y \wedge y \rhd_c^{\mathrm{cd}} z$, then add constraint $v \preccurlyeq_i w$ (if not already present).

7.10.4 Example

The example above is transformed to the following:

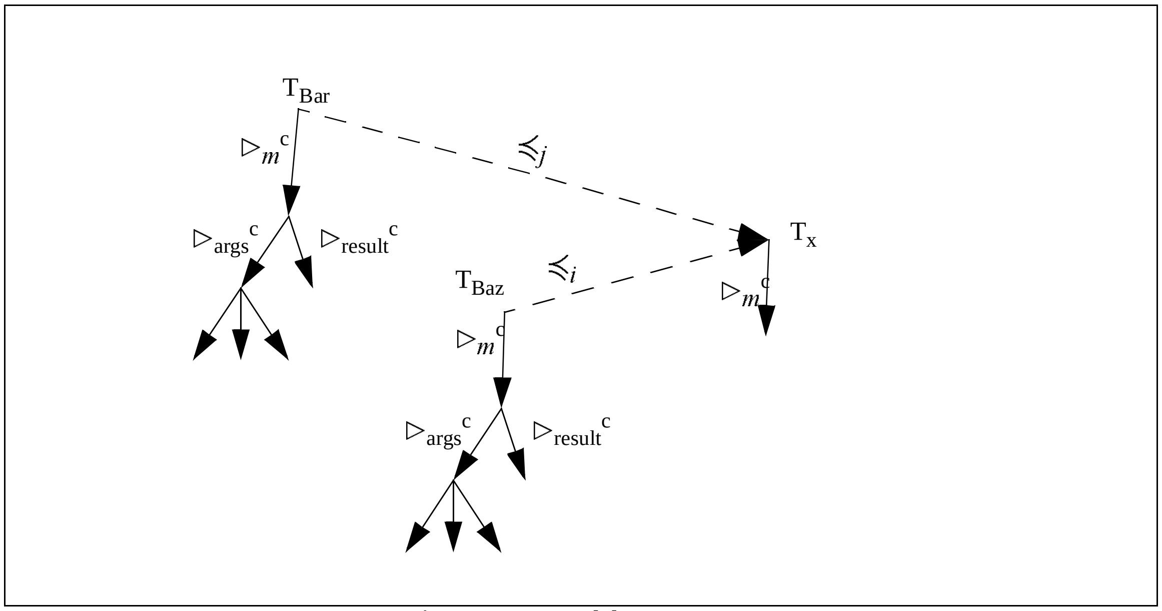


Figure 7-14. Modal Annotations

7.10.5 Implementation

These rules are not difficult to implement, and cost very little in time and space. Mode propagation takes place along with the other work on each dirty constraint from the worklist. The instance propagation check is performed very efficiently by tracking, for each

$t \rhd_c v$, whether there is an instance of the component with the "cd" annotation; this "instance mode" information is propagated from instances to sources.

7.10.6 Detecting Unused Fields

Suppose that F is a field of some class, and e is a bytecode expression, where in some traces e evaluates to real objects, but none of those objects ever have the field F. Because SEMI is sound, it will determine that the relation "$e \overline{\leftrightarrow} e$" holds. This means that SEMI has a translation for e into some constraint variable u. Now consider checking the relation "e.F $\overline{\leftrightarrow}$ e.F". SEMI will translate both occurrences of "e.F" into some constraint variable v such that $u \rhd_F v$. SEMI will therefore conclude that "e.F $\overline{\leftrightarrow}$ e.F" holds, even though it does not hold in the true relation (because the assumptions indicate that "e.F" never evaluates to any value). For some analyses, such as object modelling (see Chapter 11), it is important to be able to detect that such fields are actually unused.

The SEMI solution is illustrated in Figure 7-15.

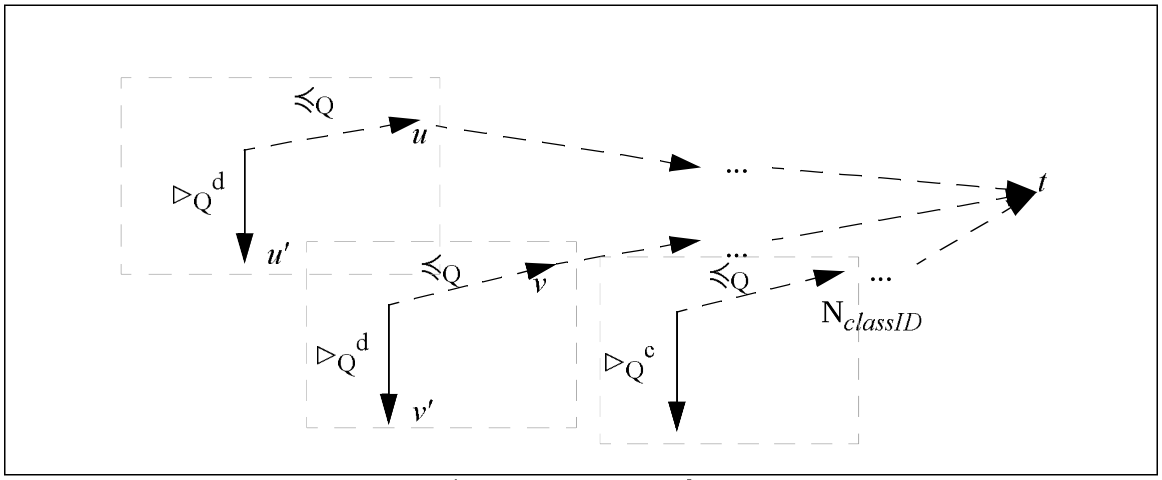


Figure 7-15. Query widget

Suppose that we have two expressions e_1 and e_2, where e_1 maps to constraint variable u and e_2 maps to constraint variable v. The two expressions are related because u and v have a common instance t. However, instead of taking u and v to be the constraint variables for the expressions, I insert the "Q-d-constraints" indicated in boxes, and assign u' and v' as the constraint variables for the expressions. Also, for each constraint variable $N_{classID}$ representing the prototypical object of each class, I insert the "Q-c-constraints" indicated in the box. Q is a single predefined component and instance label.

Now if, in fact, e_1 and e_2 can both evaluate to a single real object, then the soundness of SEMI guarantees that for some *classID* there will be a chain of instances leading from $N_{classID}$ to the common instance t. Therefore t will have a component "$t \rhd_Q^{cd} w$" for some w, and instance chains will be created leading from u' to w and from v' to w. Therefore SEMI's analysis of the instance graph will deduce that e_1 and e_2 are related.

On the other hand, if e_1 and e_2 do not evaluate to any actual objects, then there may be no such *classID* such that t is transitively an instance of $N_{classID}$. In that case t will have the component "$t \rhd_Q^d w$", i.e., the constructor mode will not be present. Therefore instance chains will not be created leading from u' to w or from v' to w, and SEMI will not deduce that e_1 and e_2 are related.

7.11 Nondeterministic Virtual Method Calls

A large contributor to the size of the constraint sets is the presence of structures corresponding to “method types” in the signatures of objects. This is a direct consequence of the way SEMI encodes virtually-invoked methods: as first-class functions carried in the slots of objects. The burden of having method types in object signatures can be eliminated by encoding each virtual method call as a nondeterministic call to one of the possible callees for that call site. The set of callees at each call site can be determined by some simpler algorithm (e.g., RTA++).

This transformation effectively reduces the program to first-order code, and allows Ajax to handle significantly larger examples. Of course, the penalty is that the analysis results may be of lower quality because higher-order control flow is not tracked as effectively. On the other hand, accuracy can improve for some examples, because at each virtual call site we can use a fresh polymorphic instance of the type of the callee. In the standard mode, because the callee is extracted from a slot of an object passed in as a parameter, its type cannot be used polymorphically. In practice we find that accuracy does decrease somewhat. The effects are quantified in Chapter 9.

Ajax does not actually generate transformed representations of programs. SEMI is configured with an arbitrary “preparatory” analysis, and then issues queries against the perparatory analysis to compute the sets of possible callees at each call site.

7.12 Future Work and Related Work

Each of these optimizations (except for cut-throughs) made significant improvements to the performance of Ajax. However, there are additional possibilities for optimizing the system. For example, there seem to be further opportunities to reduce space by implicitly representing some instance/component constraints and reconstructing them on demand. However, SEMI already seems too complex, and the generality of the constraint system seems to slow it down, especially compared to non-constraint-based polymorphic type inference systems [69] [54]. It remains unclear which strategies offer the best opportunities for future performance improvements.

Other researchers [31] have described how to improve the accuracy of this kind of analysis by labelling polymorphic instance constraints as “positive” and/or “negative”, encoding a simple kind of directionality information. For example, function results are instantiated with “positive” instance constraints, and function arguments are instantiated with “negative” instance constraints. This feature could easily be added to SEMI.

The SEMI algorithm is superficially similar to other analysis engines based on polymorphic recursion [31], since they are all based on Henglein’s algorithm. However, SEMI is the only engine that attempts to combine polymorphic recursion with handling of structures with multiple fields. The presence of types with a high degree of “fan-out” in their representation graphs motivates many of the improvements to SEMI.

8 Analyzing The Inscrutable

8.1 Introduction

This chapter discusses several features of Java that pose fundamental problems to practical, sound, whole-program static analysis, and presents Ajax's strategies for dealing with them:

- Foreign and unknown code
- Reflection and serialization
- The Java `String` "constant pool"

8.2 Foreign and Unknown Code

8.2.1 Foreign Code

One goal of Ajax is to produce sound results: The results of an analysis must account for all possible runtime behaviors of the program. I have described methods for such analysis of programs which are completely described by Java bytecode. However, all real Java programs depend on the behavior of components that are not described by Java code. For example, the standard Java class library depends on "native code" libraries for some of its functionality.

In many languages and environments foreign code is essentially subservient, providing support to the main system but influencing it only in limited ways. For example, all realistic languages provide input and output routines. However, the effects of simple routines like "print a string" and "read a string" are easily accounted for: "print a string" can be ignored, and "read a string" can be treated as code that creates a String object and fills it with an unknown number of unknown characters.

In Java, interaction between foreign code and Java code is much richer. Foreign code in standard libraries such as the Abstract Window Toolkit modifies Java-visible data (including variables holding object references, affecting aliasing), calls Java methods, and creates new Java objects. If these behaviors are ignored, then some of the program's live methods will appear to be dead, and some of the program's instantiated classes will appear not to be instantiated.

Foreign code also initializes the Java environment and transfers control to the Java program in an appropriate state. This code can be complex for programs packaged as "applets" or "servlets".

8.2.2 Unknown Code

The question of how to handle "foreign code" generalizes immediately to the question of how to handle "unknown code," which may be foreign or may simply be Java code that is inaccessible to the analysis. For example, some tasks require that an application be analyzed independently of the implementation of the Java libraries. One such task is stripping dead code from an application being packaged for execution on multiple different Java virtual machines, each with its own implementation of the standard libraries [79].

Ajax requires access to all Java bytecode for a program. The solutions that I discuss in this chapter are only applied to foreign code. However, the techniques and most of the discussion are certainly applicable to unknown code and modular analysis in general.

8.2.3 Possible Approaches

One approach is simply to make "worst case" assumptions about foreign code. Unfortunately, foreign code is almost all-powerful in Java. Most foreign code interacts with the Java virtual machine through the prescribed "Java Native Interface", but that interface allows the code to do almost anything. Some foreign code bypasses JNI and accesses Java program state directly. Therefore, if one makes worst case assumptions about the behavior of foreign code, little can be known about the behavior of Java programs.

Another approach is to make pessimistic assumptions about foreign code, tempered with "realistic" assumptions limiting the code's behavior. For example, we may assume that the foreign code used by the standard Java libraries has no knowledge of user application code, and will therefore not create application objects, modify the state of such objects or directly call methods on those objects. However, this assumption does not help us analyze the standard Java libraries. It is also possible for applications to pass knowledge — such as the names of application classes and methods — down into the standard libraries, that can then be used to violate assumptions about reasonableness.

The latter approach is feasible, but very conservative, making it difficult to evaluate the effectiveness of the actual analysis engines and Ajax tools. Therefore I have taken a third approach: manual specification of the behavior of all foreign code.

8.3 Salamis: A Specification Language for Foreign Code

8.3.1 The Need For A Separate Specification Language

One way to specify foreign code is to write a Java bytecode "dummy implementation" of each foreign subroutine. My previous system, Lackwit, took this approach of writing dummy implementations in C. This has the advantage of requiring little or no work on the part of the analysis implementor, and providing a familiar language to the specification writer.

Experience with Lackwit revealed a serious problem with this approach: it is difficult to write dummy implementations, because it is unclear which implementation details are relevant to the analysis and which are not. This is true even when the specification writer is the same person who implemented the analysis. Use of multiple complex analyses exacerbates the problem.

Therefore I created a dedicated specification language for foreign code, called Salamis[1]. Salamis has limited expressivity; for example, there is no arithmetic, and conditional branches are completely nondeterministic. The specification writer is forced to abstract away from details which are irrelevant to most large scale analyses.

To reduce the effort required for parsing and analysis, I made the language as simple as possible.

8.3.2 Example and Overview

Consider the Java code fragment in Figure 8-1.

```
...
FileDescriptor myFD = new FileDescriptor();
...
FileInputStream stream = new FileInputStream(myFD);
stream.open();
...
```

Figure 8-1. Application code using using native methods

Suppose the programmer wishes to find code that modifies her `FileDescriptor` object. The `FileDescriptor` is modified by the native method `FileInputStream.open`, but this knowledge is only available in native code specifications.

Figure 8-2 shows some code from the standard library code specification that defines the behavior of the native method `open` in the class `java.io.FileInputStream`.

```
_stringconst() {
    return = java.lang.String#internstr;
}
makeIOException() {
    STR = _stringconst();
    EXN = new java.io.IOException;
    java.io.IOException.<init>(EXN);
    java.io.IOException.<init>(EXN, STR);
    return = choose EXN;
}
java.io.FileInputStream.open(THIS, NAME) {
    FD = THIS java.io.FileInputStream.fd;
    NEW_OS_FD = choose;
    FD java.io.FileDescriptor.fd := NEW_OS_FD;
    throw = makeIOException();
}
```

Figure 8-2. Specification for `java.io.FileInputStream.open`

Each block delimited by braces defines a Salamis function. Each Salamis function either defines a native method with a fully qualified method name, such as

1. “Salamis” is the name of the island on which Ajax is said to have been buried.

"`java.io.FileInputStream.open`", or defines an internal function, such as "`makeIOException`", to be used by other specifications.

Statements within blocks are delimited by semicolons. Each statement evaluates a simple expression, with the result optionally assigned to some local variable (using the syntax "A = B").

The expression "`FD = THIS java.io.FileInputStream.fd`" reads the contents of the `fd` field declared in `java.io.FileInputStream` from the object referred to by `THIS`, and stores the resulting reference in local variable `FD`. Note that in Salamis all "this" parameters are explicit. There is no syntactic distinction between static and non-static methods. Note also that all method and field names are fully qualified with the name of their class; this avoids the need to have any static type information associated with Salamis local variables.

The statement "`NEW_OS_FD = choose;`" creates an undetermined scalar value and stores it in the local variable `NEW_OS_FD`. This statement models the retrieval of some unknown file descriptor value from the operating system.

The statement "`FD java.io.FileDescriptor.fd := NEW_OS_FD;`" stores the value of `NEW_OS_FD` into the `fd` field of the object referenced by `FD`. Syntactically, this is actually an "store expression" that is not assigned into any local variable. Note that the `fd` field here is different to the field read above. Also note that writing "`FD java.io.FileDescriptor.fd := choose;`" directly would be syntactically invalid, because every statement has exactly one expression.

The constructor of `FileInputStream` called in Figure 8-1 internally sets the stream's `fd` field to `myFD`. Static analysis then reveals that `myFD`'s own `fd` field can be modified by the call to `FileInputStream.open`. This information is reported to the programmer.

8.3.3 Salamis Syntax

The grammar of Salamis is presented in Figure 8-3. Apart from the literal strings shown in the grammer, the only tokens are *Identifiers* and quoted *Strings*.

The core of the language is the expressions:

- Object creation, e.g.,
  `new java.io.IOException`
  The object constructor must be called explicitly in a separate statement.
- Nondeterministic choice, e.g.,
  `choose EXN`
  The result of the expression is chosen nondeterministically from the comma-separated list of operands. In this example there is only one operand, so the expression simply evaluates to the value of `EXN`. If the list is empty, then the result is a fresh, unknown scalar value.

```
CompilationUnit::=Function*

Function    ::=  Name ( Identifiers ) { Statement }

Name        ::=  Identifier
             |   Identifier . Name
             |   Identifier # Name

Identifiers ::=  Identifier
             |   Identifier , Identifiers

Statement   ::=  Label? goto Identifiers ;
             |   Label? Definition? Expression ;

Label       ::=  Identifier :

Definition  ::=  Identifier =

Expression  ::=  new Name
             |   choose Identifiers?
             |   Identifier? Name
             |   Identifier? Name := Identifier
             |   Name ( Identifiers? ) String?
             |   catch ( Name? ) Identifiers
```

Figure 8-3. Salamis grammar

- Object field access, e.g.,
  `THIS java.io.FileInputStream.fd`
  This expression extracts the value of the named field from the object referred to by the operand. The first operand is omitted if and only if the field is static.
- Object field assignment, e.g.,
  `FD java.io.FileDescriptor.fd := NEW_OS_FD`
  The value of the field is set to the second operand. The first operand is omitted if and only if the field is static.
- Method call, e.g.,
  `java.io.IOException.<init>(EXN)`
  The named method is called with the provided parameters. If the method is `static`, `private`, a constructor (method named `<init>`), or `final`, then a static method call is used, otherwise a dynamic method call is used. The result of the expression is the value returned by the method, if any.
  An optional quoted string is allowed. This string contains the Java type signature of the method to call, in Java bytecode format (e.g., `"([C)V"` for a method taking an array of characters and returning void). Using this signature, Salamis can unambiguously call overloaded methods. Note that the JVM requires native methods to be uniquely named, so there is no need to define overloaded methods in Salamis.

- Salamis function call, e.g.,
  `_stringconst()`
  This is syntactically the same as a method call, but no class name is present in the method name. All Salamis function calls are static (i.e., Salamis functions are not first-class).
- Exception catching, e.g.,
  ```
  BYTE = java.io.ObjectInputStream.readByte(THIS);
  catch (java.lang.Throwable) BYTE
  ```
  This expression catches exceptions which are subclasses of `Throwable` and thrown by the statement assigning `BYTE`. The result of the expression is any caught exception. If not caught, exceptions are not propagated through Salamis code; they are simply ignored. Therefore exceptions must be explicitly propagated from callee to caller. If no class bound is given, all exceptions are caught.
- There is one kind of statement that is not an expression: “goto”, e.g.,
  `goto B, S, C, I, J, Z, F, D, L`
  Control is transferred to one of the labelled statements. Statements are labelled by prepending them with the label name and a colon.

8.3.4 Other Salamis Features

The value of the special local variable “return” is returned by each function or method. The value of the local variable “throw” is the thrown exception, if any. Salamis specifications do not specify whether an exception is thrown or the method (or function) returns normally.

Every statement that does not assign to a local variable is conditional; it may or may not actually execute. Therefore in `makeIOException`, it is unspecified whether one, both, or none of the `IOException` constructors (methods named `<init>`) are executed.

Sometimes it is necessary to associate values with objects that do not belong in the fields declared for the object in Java. One example is the lengths of arrays. For such cases, Salamis supports synthetic “specification only” fields (called “spec fields”). Static spec fields are also supported, e.g., `java.lang.String#internstr` above refers to the global spec variable “internstr”. This fields are not declared anywhere; conceptually, they are simply created as needed when accessed.

All updates to object fields in Salamis are treated as conditional; The previous value of the field may persist. Thus many of the Salamis specifications use a single object reference in a spec field to refer to a whole collection of objects. For example, `java.lang.String#internstr` refers to one of the entire collection of interned string objects; whether there is one or many is irrelevant to any analysis, because the semantics of Salamis are the same in either case.

Array accesses are treated by identifying the elements of an array object with special spec fields of the object, depending on the type of the array: `#intarrayelement`, `#longarrayelement`, `#floatarrayelement`, `#doublearrayelement`, and `#arrayelement` (for arrays of object references). Arrays of bytes, shorts, and characters have their contents mapped to `#intarrayelement`.

Sometimes it is necessary to refer to the names of array classes. These are given the internal Java Virtual Machine names (e.g., `[I` for an array of integers, `[Ljava.lang.Object;` for an array of objects).

8.3.5 Implementation

Salamis code is compiled into Java data structures by a simple front end. The data structures are then serialized into “specification resources” that are located and loaded by Ajax at analysis time.

When an analysis encounters live foreign code, it looks up the specification and then analyzes the specification directly. In other words, all analyses have to be able to analyze Java code and also Salamis specifications. In practice this is not too difficult, although it is rather cumbersome and leads to some duplication of code.

This approach also requires the language of bytecode expressions to be extended to include Salamis variables. Tools also have to be extended to scan Salamis specifications as well as Java bytecode.

8.4 Salamis Specifications

Appendix B presents the Salamis specifications for the portion of the JVM class library used by my examples.

8.4.1 Omissions

The specifications cover only the foreign code exercised by my test applications, which includes the example applications for my thesis plus some other applications. Also, they specify the code used by only the Windows implementation of the Sun JDK 1.1. Other JDK versions and implementations on other platforms use different Java libraries, which rely on different foreign code, and may therefore need different Salamis specifications. Even given these limitations, there are over 2,500 lines of specifications covering such complex areas as the Java Abstract Window Toolkit, which manages the interaction between Java and the underlying Windows graphical user interface toolkit.

There are a few places where it is impossible or undesirable to specify the foreign code adequately. The most important such area is the reflection services, which are discussed below.

8.4.2 Risks

The behavior of foreign code used by the Java libraries is difficult to deduce. Much of it is internal to the library implementation, and much of the rest is under-documented. I have proceeded by reverse-engineering the Java library bytecode, and by observing the behavior of the Java Virtual Machine. This approach is difficult and error-prone. Even with access to the JVM source code, this task would still be difficult; the JVM and its libraries are large and complicated pieces of code.

It is impossible in principle to rigorously prove that the specifications actually match the behavior of the foreign code. In practice it is also difficult to test for conformance. My testing consisted of running live code analyses using the specifications and comparing the results to profile data gathered by running the example programs in the JVM; profiled methods that are declared dead by the analysis clearly indicate bugs, either in the specifications or the analysis itself. I found many incomplete specifications this way. However, it is difficult to achieve high confidence in the completeness of the specifications.

8.4.3 Handling Strings

One quirk in the semantics of the JVM shows up in the specification of certain `String` methods. The JVM maintains a set of `String` objects called "interned `String`s": at runtime, each possible string of characters has at most one corresponding "interned `String`" object. When a JVM instruction accesses a string constant, it returns a reference to the interned `String` for that string of characters. Also, it is possible to obtain the interned `String` for an arbitary string, by calling the method `String.intern()`. This facility is provided to save space, and to allow interned `String`s to be compared for string equality merely be comparing the object references.

The unfortunate result in Ajax is that every object reference that could refer to a `String` constant must be related in the VPR to every other object reference that could refer to a `String` constant. I model this behavior faithfully in order to satisfy the definition of the VPR. Furthermore, some programs can depend on it in practice, for example when object references are compared. This is why the Salamis example above gets `String` constants from the global `#internstr` spec field. The bytecode instructions that fetch references to `String` constants also get the reference from this field. In many cases it would make sense to relax this behavior and support unsound handling of Strings.

8.4.4 Other Areas Of Interest

The Salamis code for `sun.awt.windows.WToolkit.eventLoop` is particularly interesting. This method runs indefinitely on a special AWT thread, pulling events from the Windows event queue and processing them. It responds to the native Windows events by calling methods on Java "peer" objects associated with each underlying Windows interface object. If the callbacks are not modelled correctly, then the peer object methods appear never to be invoked, and large chunks of a program's code may never be triggered.

Much of the Salamis code is devoted to ensuring that appropriate exceptions are potentially thrown by each method. Also, there is a special function `_magicexn`, which returns one of the exceptions which may be raised at any time by the Java Virtual Machine (e.g., `VirtualMachineError`). This function is used by the analyses to ensure that code which can catch such exceptions is handled soundly; the result of this function is added to the set of objects which may be caught by the code. The `_magicexn` function also includes exceptions for run-time errors that can occur so commonly that they might as well be thrown anywhere, such as `ArrayIndexOutOfBoundsException`, `NullPointerException` and `ClassCastException`. (These are the exceptions belonging to the set ErrorClassIDs in the MJBC language; see Section 3.2.5.) This results in no loss of accuracy with the existing Ajax analyses, because they do not accurately capture which exceptions can be thrown by which methods.

8.5 Reflection And Serialization

8.5.1 Introduction

An especially interesting application of foreign code is the standard Java reflection library. It allows programs to query and manipulate the elements of a Java program at run time. For example, a program can obtain, as a string, the name of the class of any object. Conversely, given the name of a class as a string, it can create an object of the class. It can obtain a list of the names of the fields and methods of an object, and other information about those members. It can even call the methods and modify the fields by name.

Reflection is extremely powerful and useful, and it is widely used by real programs. Many important Java programming paradigms depend on it (for example, Java Beans). Unfortunately, it is almost completely impervious to static analysis.

A specialized form of reflection is Java *serialization* — a facility for storing and retrieving object structures from a byte stream. Serialization uses reflection to traverse the contents of objects without requiring the user to write traversal code for each class.

8.5.2 The Reflection Services

Reflection is not an esoteric feature used by just a few applications. In fact, the Java libraries themselves depend on it. For example, the Sun JDK library reads the name of the current locale from a text file, prepends it with the string `sun.io.CharToByteConverter`, and then loads the class with that name and creates an object of the class.

Many applications, including some of the applications I chose for my benchmark suite, also depend on reflection internally. (The benchmark applications are described in the next chapter, in Section 9.2.2.) For example, the Ladybug specification checker tool [44] has a user interface shell wrapped around an abstract formula solution engine. The UI shell accesses the engine through a Java interface, and has no compile-time dependence on any particular implementation of the interface. At run time, Ladybug uses reflection to load the engine class by name and create an object of that class. The object is downcast into a reference to the engine interface, and can then be used by the user interface shell. This pattern of using reflection to break compile-time dependencies is quite common.

Another interesting use of reflection is in the Jess expert system shell [35]. Jess interprets rule sets, which are essentially programs. These programs can contain directives to create and manipulate Java objects; these directives are interpreted by Jess by simply passing them down to the Java reflection API (along with some wrapping and unwrapping between Java object references and Jess data). By this simple mechanism, the full power of the Java platform is available to Jess programs. Clearly, static analysis of Jess alone in the presence of these directives is no longer possible; one would have to analyze Jess in combination with the Jess rules being interpreted. When I use Jess as one of my example programs for this thesis, I assume that these particular directives are not used.

Of course, Java's original source of popularity was that it can dynamically load and run code from arbitrary sources. This ability depends on the use of reflection. It also requires the use of ClassLoaders, but ClassLoaders do not present any real problems for Ajax above and beyond the difficulties of reflection.

Another, rather obscure, use of reflection is built into the Java compiler. The Java language construct `ClassName.class` obtains the metaclass `Class` object for the class named "`ClassName`". The Sun Java compiler implements this feature by compiling in a call to `Class.forName("ClassName")`, along with some caching of the return value to speed up cases where the expression is evaluated frequently.

8.5.3 Reflection Specifications

Ajax allows the programmer to manually provide specifications describing how a program uses reflection, e.g., which classes it can create instances of and which methods it can call using the reflection API. Appendix C gives the actual specifications used in the experiments.

Reflection specifications describe a set of *reflective methods*, the methods that perform reflection operations. For each reflective method, the specifications list the caller methods, and for each caller, the specifications enumerate the classes, methods or fields it may access through the callee reflection method. For example, consider Figure 8-4.

```
java.lang.reflect.Constructor.newInstance [
    javafig.gui.ModularEditor.handleCommandCallback {
        class=javafig.commands.*
    }
    ajax.tools.benchmarks.GeneralBenchmark.makePrintSinkStream {
        class=java.io.PrintStream
    }
]
```

Figure 8-4. Sample reflection specification

Figure 8-4 specifies that `Constructor.newInstance` is reflective. (This method creates a new object using a constructor chosen at run time.) The specification states that there are only two callers of this reflective method. The first caller, `handleCommandCallback`, only uses the method to create objects of classes whose fully qualified names start with "`javafig.commands.`" The second caller uses it only to create objects of class `java.io.PrintStream`. Note that once again every class, method and field name is fully qualified with the declaring class name and package.

This specification format has two advantages. Ajax can check during analysis that every caller to a reflective method is actually listed in the specifications, and issue warnings when unknown callers are found. This is an essential aid to locating all uses of reflection in a program. Also, the usage of reflection can be computed based on the methods that Ajax finds to be live; dead code that uses reflection does not impact the analysis. This means that one specification file can describe the reflection behavior of the Java libraries and a set of user applications. The only other analysis system with documented support for reflection specifications, Jax [79], only allows the programmer to specify one list of methods and classes accessed via reflection, and does not allow the programmer to specify which program methods perform reflective actions; thus it does not have these advantages.

Another advantage of this format is that wrappers around reflective methods can be added to the specifications as a new reflective method. This allows its callers to be easily located and reported by Ajax.

Ajax has a separate mechanism to handle the compiler generated use of `Class.forName` discussed above. During analysis, it detects when `Class.forName` is called with a constant string parameter, and adds the named class to the list of classes which are reflected. Therefore uses of the `ClassName.class` expression do not need to be listed in the reflection specifications.

8.5.4 Reflection Specification Syntax

The syntax is very simple. The example above demonstrates almost all the syntactic features of the language. A reflective method can have an arbitrary number of callees, and each callee can specify an arbitrary number of “reflection targets”. A reflective method and its callees are specified as fully qualified method names; if disambiguation of overloaded methods is required, the method name can be extended with a list of parameter types and quoted as a string. The grammar is given in Figure 8-5. As for Salamis, the tokens are the literal strings occuring in the grammer, plus *Identifiers* and quoted *Strings*.

```
ReflectionSpec::=ReflectiveMethod*

ReflectiveMethod::=MethodName { Caller* }

MethodName::=  Name
            |  String

Name        ::=  Identifier
            |  Identifier . Name

Caller      ::=  Name { ReflectionTarget* }

ReflectionTarget::=TargetType = TargetSpec

TargetType  ::=  class
            |  field
            |  method
            |  serialized

TargetSpec  ::=  WildcardName
            |  WildcardName < Name

WildcardName::=Name
            |  Name .? *
            |  * .? Name
```

Figure 8-5. Reflection specification grammar

Reflection targets identify the classes, methods or fields that may be referenced by the reflective operation. There are four kinds of reflection targets:

- Classes
- Methods
- Fields
- Serialized Classes

None of the examples I have analyzed use field reflection.

The “serialized class” targets are used to specify which classes of objects may be read from storage using the `ObjectInputStream` deserialization machinery. If a class is a “serialized class” target, then instances of that class may be returned from calls to `ObjectInputStream.readObject`. The `ObjectInputStream` constructor is treated as a reflective method; callers of the constructor specify which classes they will deserialize using the stream. Strictly speaking the constructor is not a reflective method, because objects are not deserialized and created until `readObject` is called on the stream. However it is more helpful to identify creators of object input streams than readers of objects from those streams.

The language supports two shorthand ways to specify reflection targets, corresponding to ways that reflection is frequently used in practice:

- Wildcard names, e.g.,
  `javafig.commands.*`
  This means any class (or method) whose fully qualified name starts with “`javafig.commands.`” Wildcards need not be in trailing positions, e.g., “`*.Handler`” is allowed. Ajax searches through all the available classes, methods or fields to find the ones whose names match the pattern. These patterns are very useful because programs often prepend or append some constant string to a variable before passing a name to the reflection API.
- Interface constraints, e.g.,
  `jess.*<jess.Test`
  This means any class matching the pattern “`jess.*`” which implements the named interface `jess.Test`. This is also very useful because programs creating objects via reflection usually require those objects to satisfy some known interface.

Serialized class targets undergo additional processing. Every serialized class target must implement the `java.io.Serializable` interface, or it will be ignored. Also, for every field of a serialized class which is not marked `transient`, the field’s declared class is added as a serialized class target. (This is because Java serialization automatically serializes such fields.) Similarly, if an array class is serialized, then the array content class is also serialized.

8.5.5 Creating The Specifications

Writing reflection specifications requires some reverse engineering of the reflection-using code. I used a combination of dynamic and static methods. I ran the example programs and noted which classes were loaded and which methods were called. I also examined the bytecode (and source code, when available) and determined which classes and methods could be accessed.

The specifications I produced use two simplifications to reduce the number of possible classes that may be loaded. First, the character set locale name is assumed to be “Cp1252”, the Windows Latin character set. Secondly, the locale is assumed to be US English. If all available character sets and locales are allowed, the very large amount of code loaded to support them totally dominates the size of my example programs, and most configurations of SEMI are quite impractical.

8.5.6 Using Reflection Specifications

Reflective methods ultimately depend on foreign code. (The reflective methods that appear in the Java library are actually wrappers around foreign methods that do the real work.) I have written Salamis specifications for those foreign methods that take care of mundane aspects such as throwing exceptions, and delegate the essential reflective operations to a special set of foreign functions. These functions are:

- `ReflectionHandler_makeObjectAndCallZeroArgConstructor`
  Creates an instance of some reflected class with a constructor that takes no arguments, and invokes that constructor on the object.
- `ReflectionHandler_makeObjectAndCallArbitraryConstructor`
  Creates an instance of some reflected class and invokes one of the constructors on the object; the parameters to the constructor are passed to this function as an array.
- `ReflectionHandler_callArbitraryMethod`
  Calls a reflected method on some object. The parameters are passed into this function as an array.
- `ReflectionHandler_makeSerializedObject`
  Creates an instance of a serialized non-array class. No constructor is invoked.
- `ReflectionHandler_makeSerializedArray`
  Creates an instance of a serialized array class.
- `ReflectionHandler_assignSerializedField`
  This is actually a family of functions, one per primitive type and one for `Object`. Given an object and a value of the appropriate type, it sets one of the serialized fields of the object to the given value.
- `ReflectionHandler_getSerializedField`
  This is actually a family of functions, one per primitive type and one for `Object`. Given an object, it returns the value of one of the serialized fields of the object with the appropriate type.
- `ReflectionHandler_invoke_readObject`
  Given an object which has a `private readObject` method implementing custom serialization behavior, this function calls that method on the object.

- `ReflectionHandler_invoke_writeObject`
  Given an object which has a `private writeObject` method implementing custom serialization behavior, this function calls that method on the object.

Since none of my examples use reflection to modify object fields (other than for serialization), I did not build support for that functionality.

These functions cannot be specified statically in Salamis code because they depend on knowing the set of reflected classes, methods, and serialized classes. Instead, their specifications are generated dynamically. As analysis progresses and live methods are discovered, they are looked up in the reflection specification. Any induced reflected classes, methods or serialized classes are added to a global list of reflected entities. Whenever this list is updated, Ajax generates new specifications for the primitive reflection functions. (Ajax analyses support code mutation, so they can handle changes in the specifications even if the reflection functions have already been analyzed.)

8.6 Conclusions

Java programs have rich interactions with their environment. These interactions must be modelled accurately to achieve sound and accurate analysis. Unfortunately, this is very difficult to do; the details of the environment are inaccessible, incomprehensible, and subject to change. Even worse, the environment provides reflection facilities allowing Java programs to modify their own behavior in ways that are opaque to static analysis.

Ajax addresses these concerns by providing ways to specify the environment and a program's reflective behavior. These mechanisms work, but they can be laborious for both the tool implementor and user. More seriously, any attempt to specify the environment and reflective behavior seems doomed to be fragile, for the reasons explained above.

Although these concerns can be tightly constrained or eliminated in some domains (e.g., embedded systems), general purpose systems design is moving in the direction of *more* of these kinds of problems. Distributed systems, dynamism and introspection are increasingly likely to be the norm. Even embedded systems are increasingly likely to be attached to networks and to exhibit these features — for example, the Jini "smart devices" framework depends on them. Static analysis cannot ignore this challenge.

9 Performance

9.1 Introduction

This chapter describes the resource consumption and accuracy of the basic analyses RTA++ and SEMI for some simple applications: resolving virtual method calls and identifying each program's live code. The focus is on measured performance rather than theoretical estimates or bounds, because performance depends crucially on the characteristics of the programs being analyzed.

The results report accuracy in terms of application metrics (e.g., the number of virtual call sites successfully resolved to a single callee). Metrics internal to an analysis algorithm (e.g., the average size of points-to sets) can be useful for diagnosing the behavior of a particular algorithm, but are not as useful for comparing different analysis algorithms.

Before I describe the performance of the algorithms, I describe the suite of example programs and the test setup. It is difficult to measure the sizes of the programs, partly because it is difficult to describe precisely what code constitutes each program. This is interesting because it also makes whole-program static analysis hard.

One goal of this thesis was to test the scalability of SEMI-style analysis applied to Java programs. My results show that treating methods as functions passed around in records imposes a significant penalty, and prevents the largest examples from being treated within the resource limits I have set. However, this treatment can handle some large and interesting programs, including the Ajax system itself with all the libraries on which it depends.

Ajax has many tunable parameters that can alter the accuracy and resource consumption of the sytem. In my results here, and in subsequent chapters, I focus on proving or disproving specific hypotheses rather than attempting to characterize completely the performance of the system in all possible configurations.

9.2 Benchmark Environment

9.2.1 System

Table 9-1 gives the specifications of the machine running the test.

9.2.2 Benchmark Examples

I use a suite of ten benchmark programs, described in Table 9-2. Each program is analyzed in conjunction with the libraries provided in Sun's JDK 1.1.7. These programs cover a range of sizes and programming styles.

CPU	500MHz Pentium II
RAM	256MB
Swap Space	600MB
Java VM	Sun JDK 1.3.0, Hotspot Client VM
Java Heap Size	192MB
Operating System	Windows NT 4.0, Service Pack 5

Table 9-1. Environment specifications

Program Name	**Description**
Ajax	The downcast checking tool of my analysis system
CTAS	The Connection Manager for a prototype air traffic control system, in a test harness, from Daniel Jackson’s group at MIT [43]
Jar	The JAR compressed archive manager from Sun’s JDK 1.1.7
Java2HTML	Converts Java source code to pretty HTML, from Rustan Leino at DEC/Compaq SRC
JavaC	The Java source-to-bytecode compiler from Sun’s JDK 1.1.7
JavaCC	The Java Compiler Compiler from Sun Labs, version 0.8pre1 (similar to Yacc)
JavaFIG	The JavaFIG 1.3.4 drawing editor from Universitaet Hamburg
JavaP	The Java bytecode disassembler supplied with Sun’s JDK 1.1.7
Jess	Java Expert System Shell version 4.4, from Sandia National Labs [35]
Ladybug	The Ladybug specification checker, by Craig Damon at CMU [44]

Table 9-2. The example programs

Table 9-3 records the program sizes. Measuring the size of a program in this context is perplexing. The first difficulty is that only four of the programs — Ajax, CTAS, Jess and Java2HTML — come with complete source code, so measures such as “lines of code” are inapplicable.

More seriously, for each example, the code actually analyzed is neither a superset nor a subset of the code comprising the “application.” (By “application,” I mean a body of code that one downloads and installs as a unit.) In most cases the analyzed code is much larger than the application code, because Ajax analyzes all libraries on which the application depends, as well as the application itself. On the other hand, Ajax only analyzes the code that it detects to be live. Some applications, such as Ajax and JavaFIG, consist of several independently runnable programs; therefore, whichever program is analyzed, a significant amount of the application code falls outside the program. For Jar, JavaP and JavaC there is no clear boundary between the application and the JDK libraries, and the separation into application and library code is somewhat arbitrary.

Name	**App. Source Lines**	**App. Classes**	**App. Methods**	**App. Bytecode Bytes**	**Total Live Classes**	**Total Live Methods**	**Total Live Bytecode Bytes**
Ajax	45,086	505	3,145	171,237	537	3,463	197,398
CTAS	6,909	60	365	17,350	283	1,527	86,523
Jar	N/A	8	85	6,142	304	1,752	104,979
Java2HTML	543	5	32	2,498	101	388	12,316
JavaC	N/A	122	948	68,859	417	2,817	192,528
JavaCC	N/A	134	1,975	250,653	161	1,322	170,741
JavaFIG	N/A	175	2,139	170,655	496	3,902	250,725
JavaP	N/A	58	577	52,215	143	705	32,026
Jess	36,366	173	821	51,468	383	1,854	110,526
Ladybug	~57,000	389	3,109	238,755	731	5,277	346,491

Table 9-3. Size statistics for the example programs

Some features of the example programs skew these statistics. Ajax and JavaCC contain JavaCC-generated code, although Ajax's generated code is not actually analyzed. Ladybug contains code generated by a different parser generator, JavaCUP. Thus, the characteristics of these programs are partly determined by the design of the parser generator. These characteristics may be different to the characteristics of "handwritten" code, but it is important and interesting to examine both handwritten and machine generated code.

Another problem is that static "class initializer" methods are often unlike other methods in the program. The Java bytecode format has no way to represent an initialized array; therefore all constant arrays are constructed at run time within the class' static initializer. Usually at least five bytes of bytecode instructions are required per array element. Thus, many class initializer methods are huge compared to other methods, and in some programs they dominate the overall bytecode instruction count. All results in this thesis exclude static class initializer methods from statistics about methods. In particular, the method counts and bytecode byte counts in Table 9-3 exclude static class initializer methods. This does mean that some legitimate code is excluded from the reports, but it improves the meaningfulness of the results overall. These omissions are only in the reporting of results — the analyses take the behavior of the static class initializers fully into account.

In Table 9-3, the "Total Live Classes" number is simply the number of classes containing at least one method body which Ajax determines to be live. The "Total Live Methods" records the number of method bodies determined to be live (excluding static class initializers), and the "Total Live Bytecode Bytes" is the sum of the sizes of those methods. Here the set of live methods was computed using the "RTA++" analysis. (Other analyses compute smaller sets of live methods.)

JavaFIG and Ladybug are the only two applications that use the AWT user interface library, and that library accounts for much of the code that is pulled in from outside the application.

Figure 9-1 shows the size of each example program, as the number of live methods. Figures 9-2 and 9-3 show that the number of live methods is a reasonably good measure of program size, being well correlated with the number of classes and number of bytes of bytecode instructions for each program. This correlation is improved by the fact that the programs share a great deal of code (the JDK libraries).

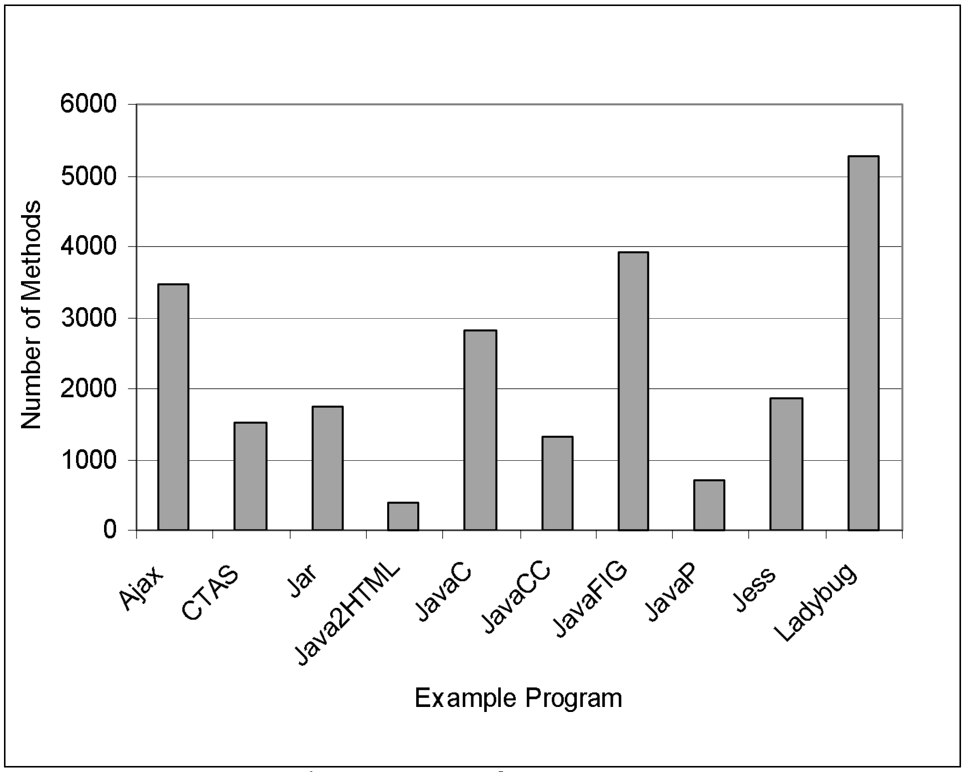


Figure 9-1. Example program sizes

Figure 9-4 shows that, considering only code outside the JDK library, the correlation between bytecode bytes and number of methods is still nearly linear, except that Ajax has unusually small methods and JavaCC has unusually large ones.

Figure 9-5 shows that for application code, the number of methods per class varies greatly.

9.3 Tools

In this chapter, I consider two tools: virtual method call resolution and live code identification. Other tools and their performance are discussed in later chapters. Here I focus on comparing the performance of different algorithms and configurations.

9.3.1 Virtual Call Resolution

Virtual call resolution is the problem of determining, for each virtual method invocation site, a superset of the actual method bodies that may be invoked by the call. This chapter examines the performance of the virtual call resolution technique described in Section 4.3.4.

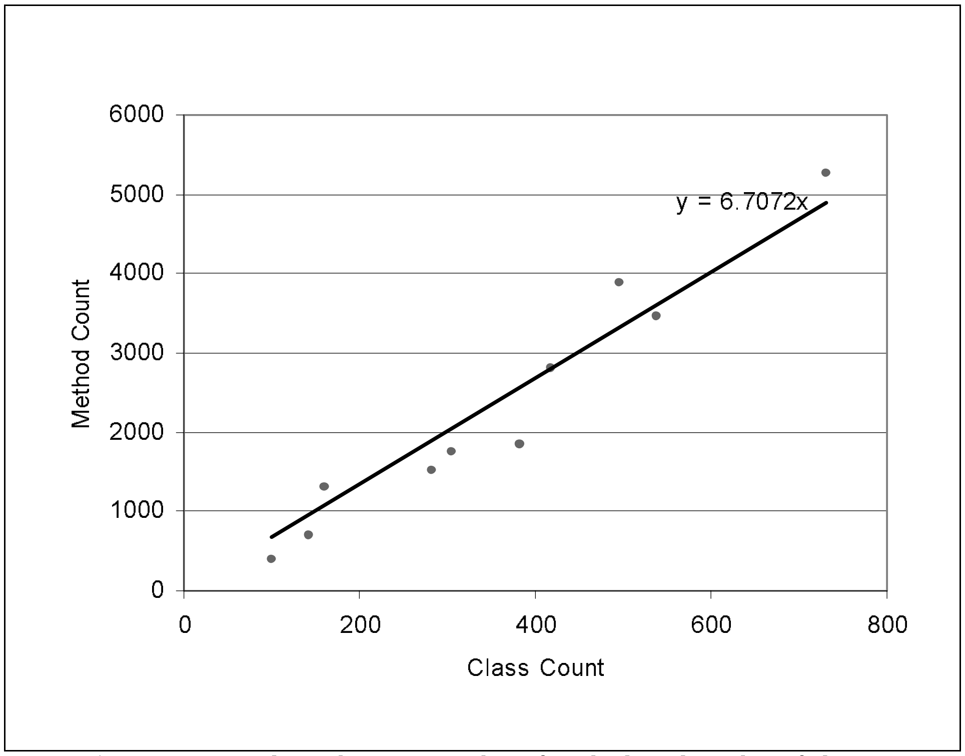


Figure 9-2. Correlation between number of methods and number of classes

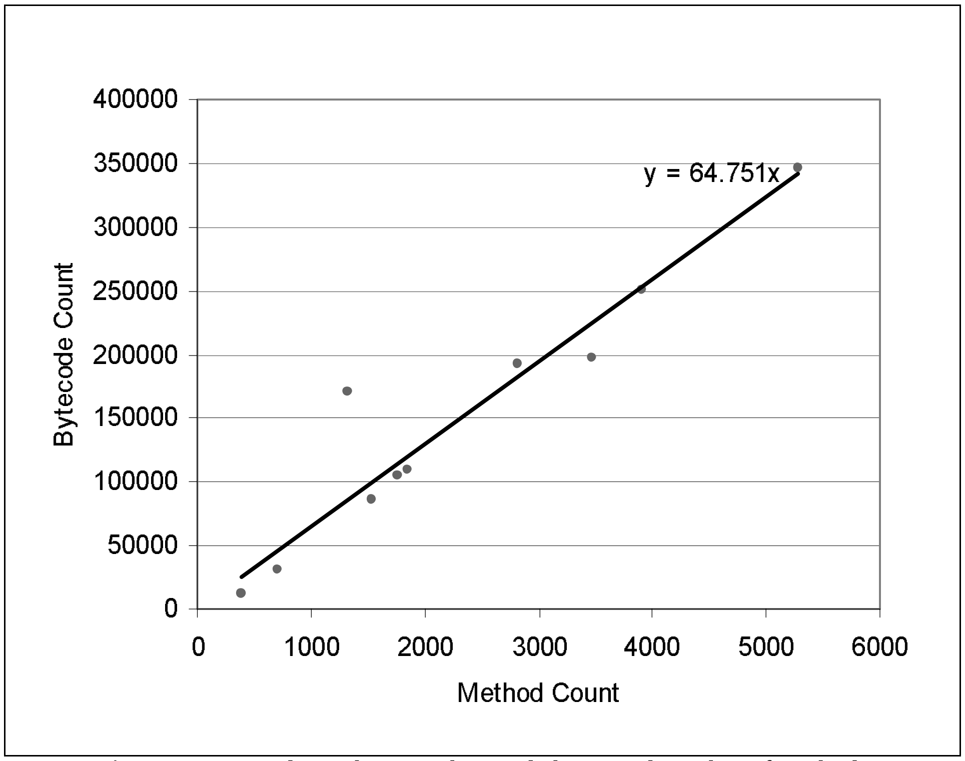


Figure 9-3. Correlation between bytecode bytes and number of methods

The virtual call resolution tool scans each live method found by the analysis and identifies the occurrences of `invokevirtual` and `invokeinterface` instructions. Each such

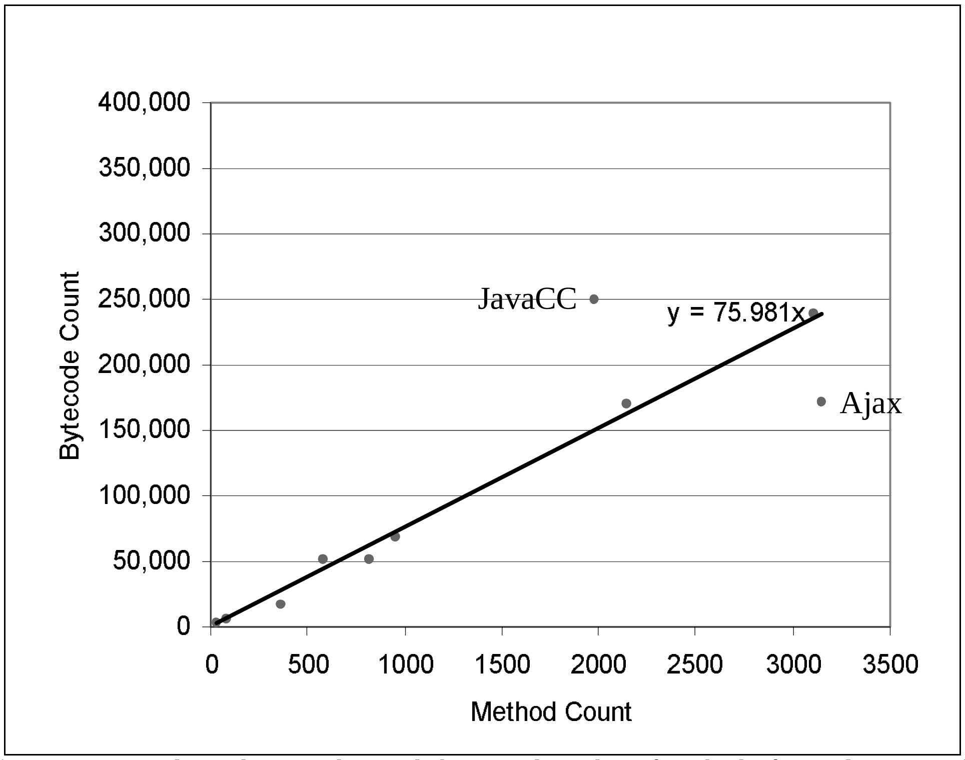


Figure 9-4. Correlation between bytecode bytes and number of methods, for application code

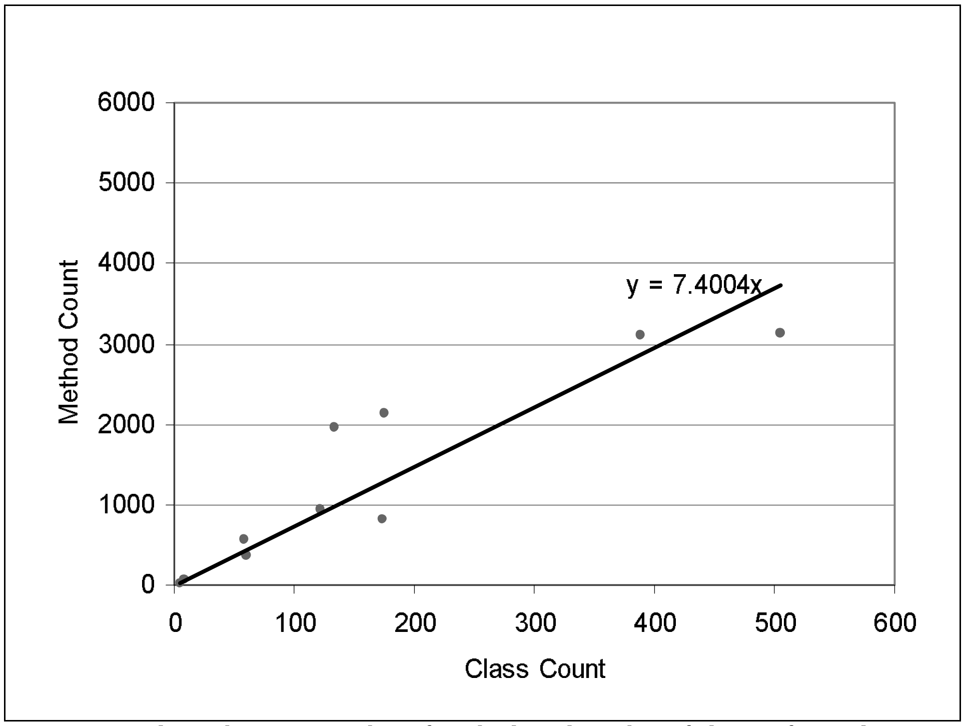


Figure 9-5. Correlation between number of methods and number of classes, for application code

instruction is considered a “virtual method invocation site”, unless the callee method is declared `final` or its declaring class is `final`, in which case it is ignored (being trivial to resolve statically). For each site, the tool collects and outputs the set of possible callee method implementations. Section 4.3.4 describes how sets with more than one element are abstracted to a single “many” value. In the implementation, the threshold is configurable; the entire set of possible callees can be retrieved by setting it to a large integer.

Note that calls to `private` methods, constructors, `static` methods, and superclass methods (via `super`) all use the `invokestatic` or `invokespecial` instructions and so are ignored by the virtual call resolver.

The tool summarizes its results by reporting three numbers:

- The number of virtual method invocation sites found.
- The number of sites resolved, i.e., the number of sites with zero or one possible callees.
- The number of sites dead, i.e., the number of sites with zero callees. A dead site is either never executed or else, whenever it is executed, the object reference used for dispatch is always null (and therefore an exception is thrown).

The key accuracy metric is the ratio of the first two numbers: the percentage of sites resolved.

As discussed above, because of the frequently anomalous nature of class initializer methods, sites within class initializer methods are not included in the statistics.[1]

9.3.2 Live Code Identification

Live code identification is the task of determining a set of method bodies that is a superset of the actual method bodies that may be executed by the program. (Alternatively, it can be thought of as the task of determining a set of method bodies that are guaranteed never to be executed by the program.) This chapter benchmarks the VPR-based technique described in Section 4.3.5.

The tool summarizes its results by reporting two numbers:

- The number of dead method bodies found in the application code
- The total number of method bodies found in the application code

The ratio of these two numbers is the key accuracy metric here: the percentage of methods in the application found to be dead.

Class initializer methods are counted in these statistics because they cannot significantly skew the results.

The results for this task do not vary much across analyses. A simple analysis such as RTA seems to get close to the “true” set of live methods, so there is little room for improvement.

1. One example is the class initializer for the class `sun.io.CharacterEncoding`, which contains 411 virtual calls to `Hashtable.put`. This would account for more than half of the virtual call sites in some examples.

9.4 Performance of RTA++

Figure 9-6 shows the memory required for Ajax to analyze the example programs with RTA++ for the two tasks of virtual method call resolution and live code identification. Figure 9-7 shows the time taken. **RTA++ is fast in each case. The two tasks have similar resource requirements.**

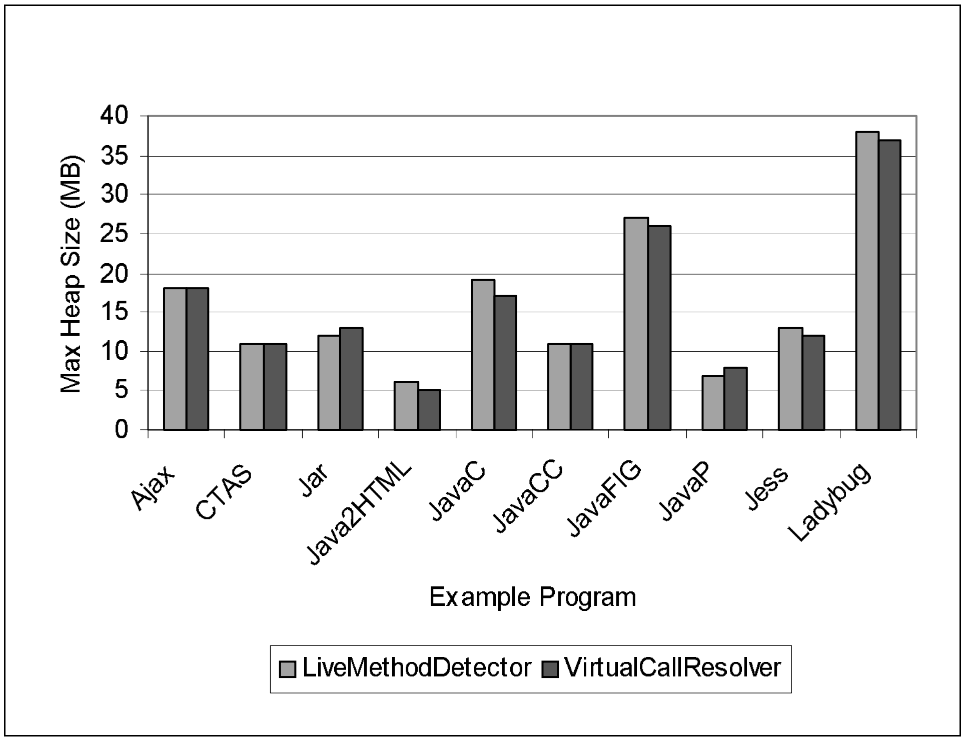


Figure 9-6. Memory consumption of RTA++

The quality of the RTA++ results is presented later, in comparison with the results for SEMI.

9.5 Performance of SEMI

9.5.1 Overview

Figure 9-8 shows the amount of memory used by SEMI in a "high accuracy" configuration, for both the virtual call resolution and live code identification tasks. Figure 9-9 shows the time taken. The missing bars indicate that the analysis did not terminate within three hours.

All configurations of SEMI presented in this chapter use RTA++ to resolve virtual method invocations where possible before applying SEMI (see Section 7.8.1). In this "high accuracy" configuration, SEMI performs precise analysis for the remaining virtual method calls but turns off full polymorphic recursion; this decision is explained below.

These results also show that **using SEMI, differences in the resource requirements of the two tools are more pronounced**. The reason is that the tool-specific data are propagated over much larger graphs for SEMI than for RTA++.

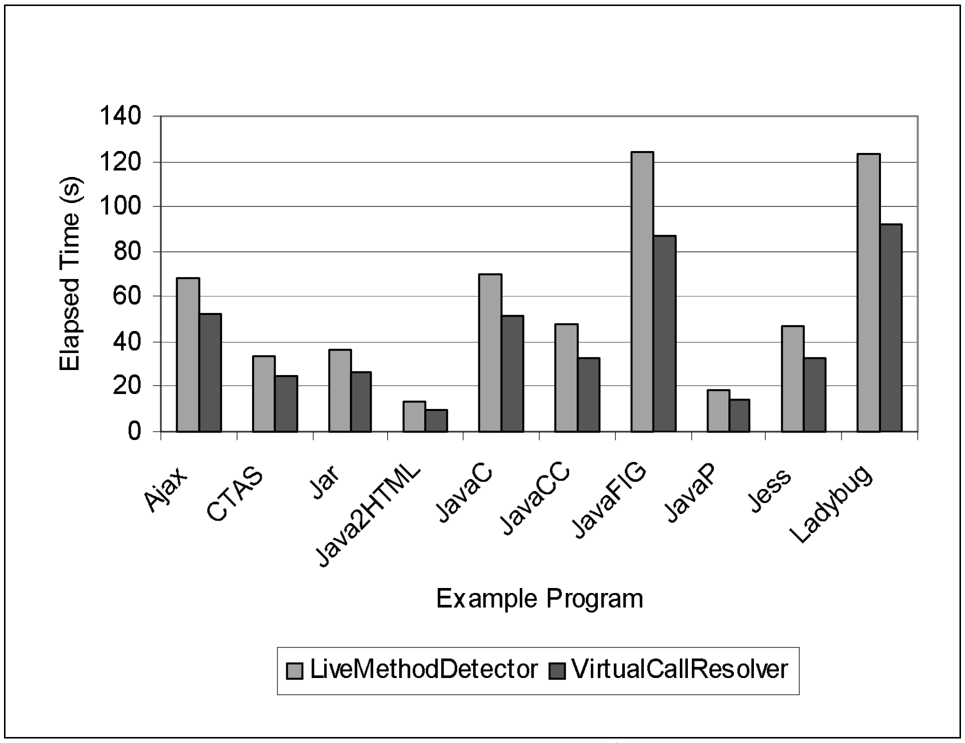


Figure 9-7. Time consumption of RTA++

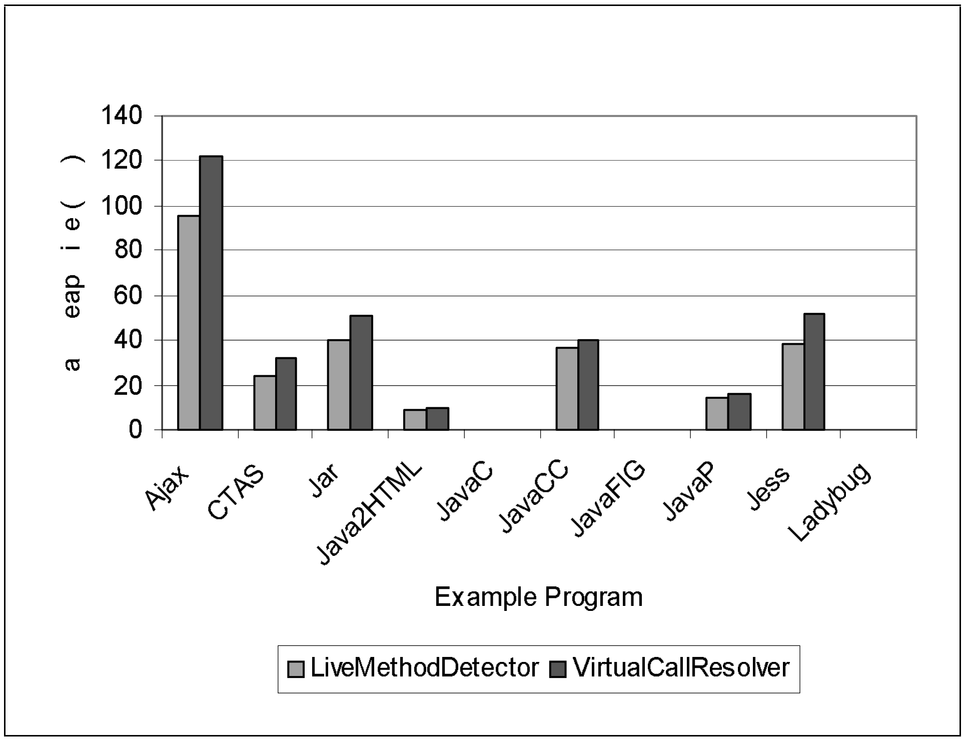


Figure 9-8. Space consumption of SEMI configured for high accuracy

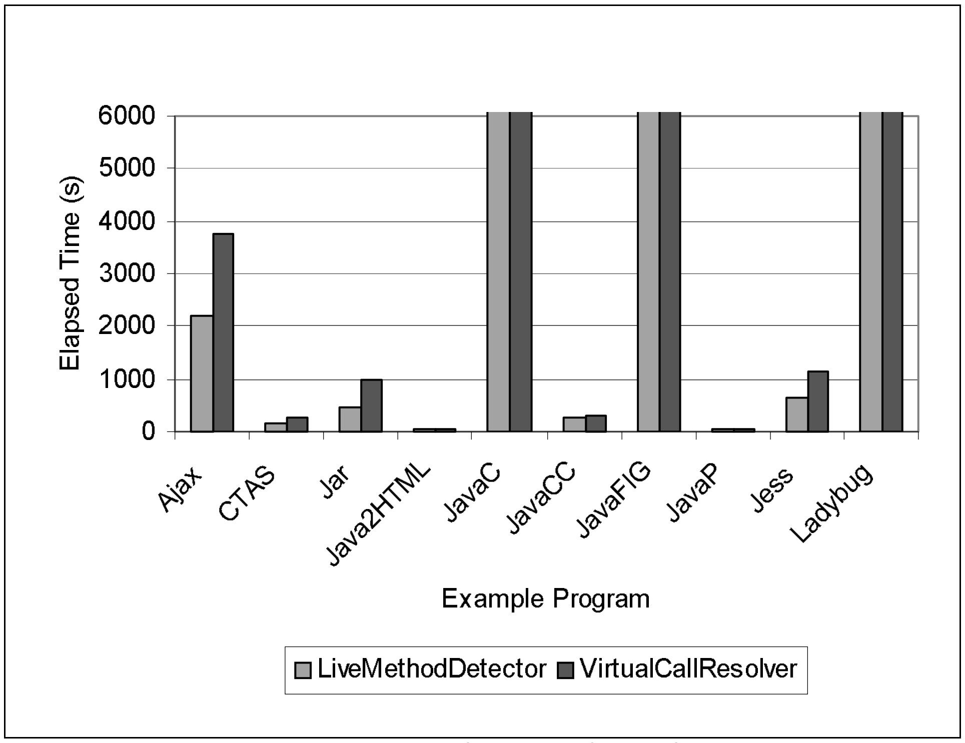


Figure 9-9. Time consumption of SEMI configured for high accuracy

9.5.2 Performance of SEMI in Different Configurations

Now I consider configuring SEMI for reduced accuracy but greater efficiency. Figure 9-10 shows the memory consumption for live method detection using all combinations of the PolyRec and HighOrder options. Figure 9-11 shows the time used.

- When *PolyRec* is enabled, full polymorphic recursion is used. Otherwise polymorphic recursion is mostly suppressed (see Section 7.3.6).
- When *HighOrder* is enabled, virtual method calls are analyzed by the precise techniques described in Chapter 6, otherwise the program is treated as first-order by SEMI, using RTA++ to compute all the possible callees of each virtual call site (see Section 7.11).

The technique described in Section 7.11 for transforming the programs to first-order code significantly reduces the resource usage, making some large examples tractable that were previously intractable. Abandoning full polymorphic recursion reduces resource requirements with HighOrder enabled, but gives mixed results with HighOrder disabled.

9.5.3 Accuracy of SEMI in Different Configurations

The settings of the PolyRec and HighOrder options affect the accuracy of the analysis. Figure 9-12 shows results for live method detection. Figure 9-13 shows results for virtual call resolution.

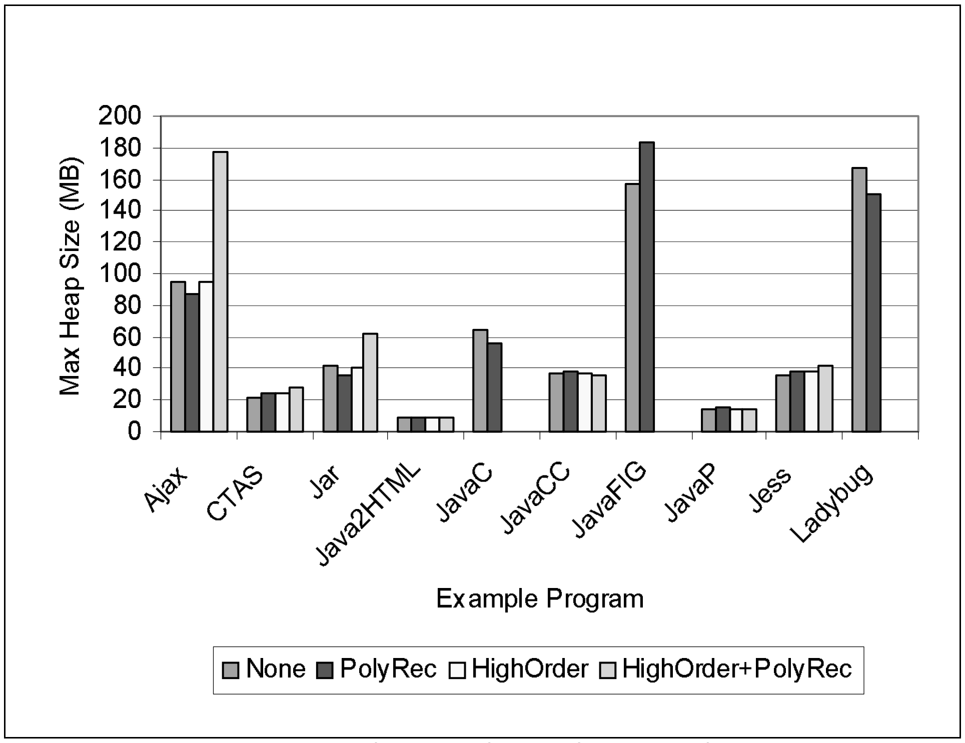


Figure 9-10. Space consumption of SEMI in four configurations, for live method detection

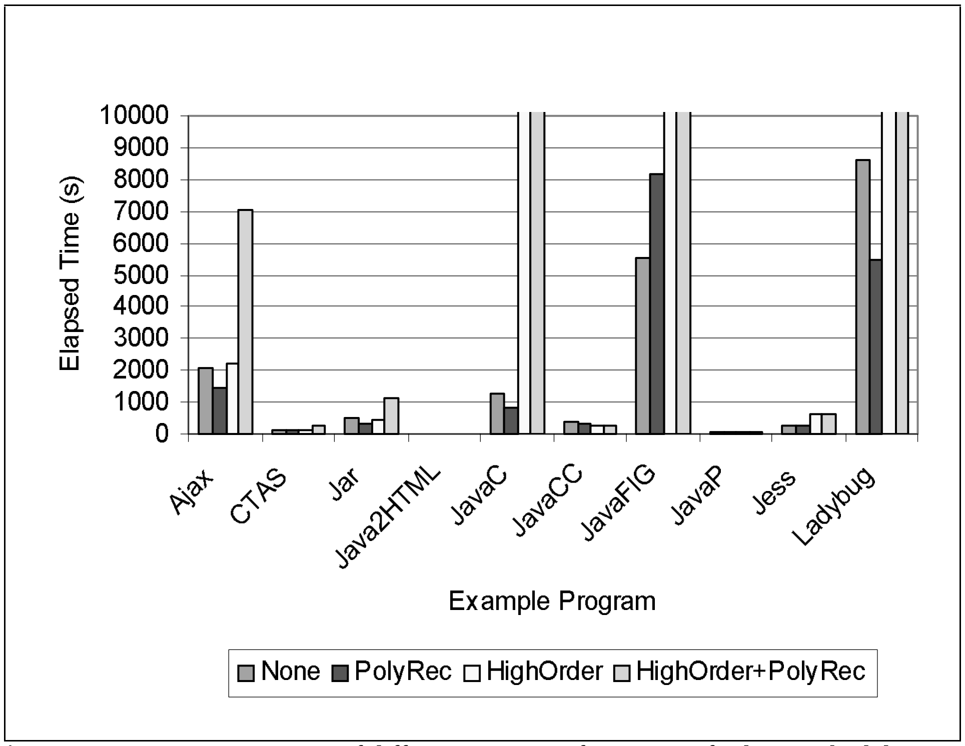


Figure 9-11. Time consumption of different SEMI configurations, for live method detection

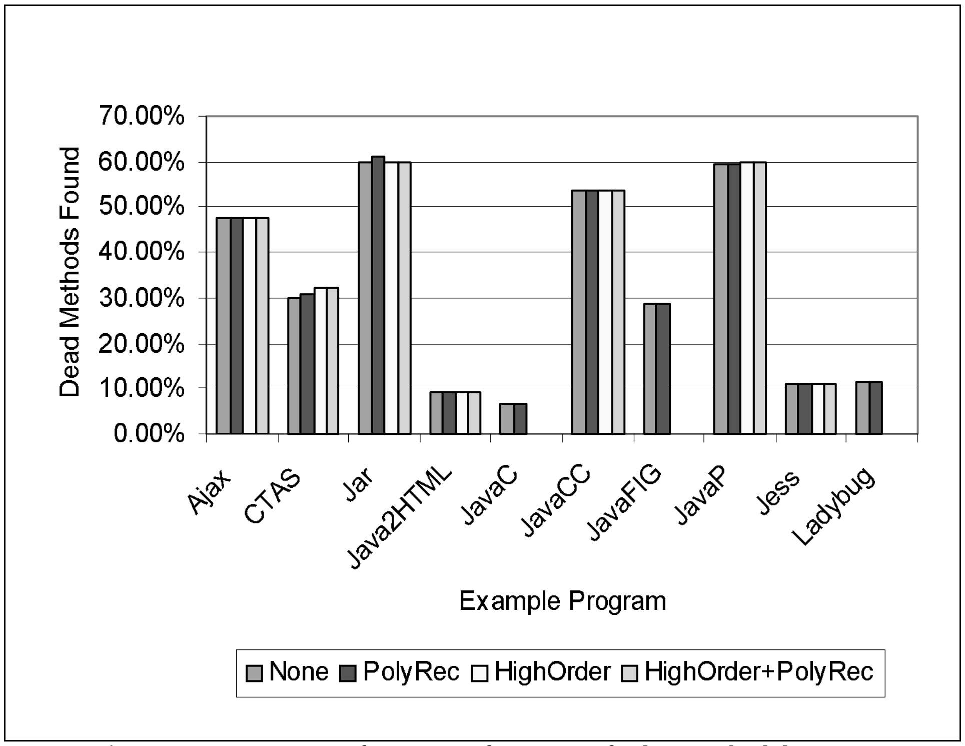


Figure 9-12. Accuracy of SEMI configurations for live method detection

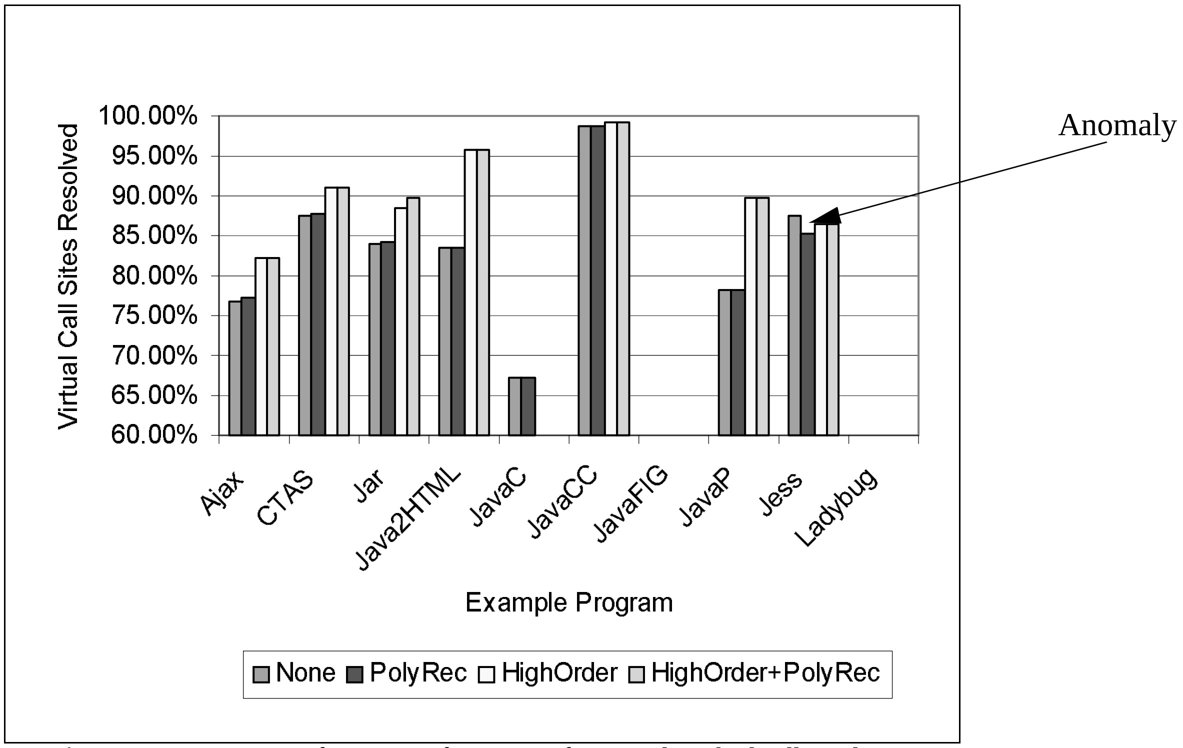


Figure 9-13. Accuracy of SEMI configurations for virtual method call resolution

A large number of dead methods are found in the application code of Ajax, CTAS, Jar, JavaCC, JavaFIG and JavaP. In these examples, the “application code” actually comprises several different programs, only one of which is analyzed by Ajax.

The results for virtual call resolution show a slight anomaly: turning off full polymorphic recursion actually improves accuracy for Jess. Normally, restricting polymorphic recursion can only decrease accuracy. In this case, slight variations in the order of constraint processing determine whether calls to `System.err.println` are resolved or not.

Restricting polymorphic recursion does not significantly affect accuracy for either live method detection or virtual call resolution.

Different SEMI configurations produce little variation in the results for live method detection.

For virtual call resolution, enabling HighOrder significantly improves accuracy. Many virtual method call sites do have more than one possible callee, so even an oracle would resolve fewer than 100% of virtual call sites. Therefore, an improvement from (for example) 88% to 89% of call sites resolved is significant, as it should be considered a reduction of at least 10% in the number of resolvable but unresolved call sites.

Using HighOrder never decreases accuracy in practice. Section 7.11 explains why this might not necessarily be so.

9.5.4 Component Partitioning in SEMI

In Section 7.9.1, I claimed that component partitioning improved the performance of SEMI, in particular when object field components were partitioned according to the declaring class of each field. Figure 9-14 shows the memory consumption of three different configurations of SEMI applied to the live method detection problem. Figure 9-15 shows the time consumption. The configurations all use PolyRec but not HighOrder, and each configuration uses a different partitioning scheme.

Clearly, “by class” uses about the same amount of memory as having no partitioning. “By hierarchy” (see Section 7.9) uses substantially more in most cases. Furthermore, “by hierarchy” is often much slower and “by class” is usually fastest, sometimes significantly faster than “none”.

These results verify the claim that **partitioning object field components according to the declaring class of each field is a good idea.**

9.6 RTA++ and SEMI Intersection

9.6.1 Basic Results

Ajax can be configured to compute the intersection of the results of two analyses, and the result is guaranteed to be at least as accurate as each analysis applied separately. Because RTA++ is cheap, intersecting it with SEMI is not much more expensive than running SEMI alone. The resulting analysis is denoted “SEMI & RTA++”.

Figure 9-17 compares the accuracy of SEMI & RTA++, SEMI, and RTA++, using neither HighOrder nor PolyRec, for virtual call resolution. **The results show that SEMI &**

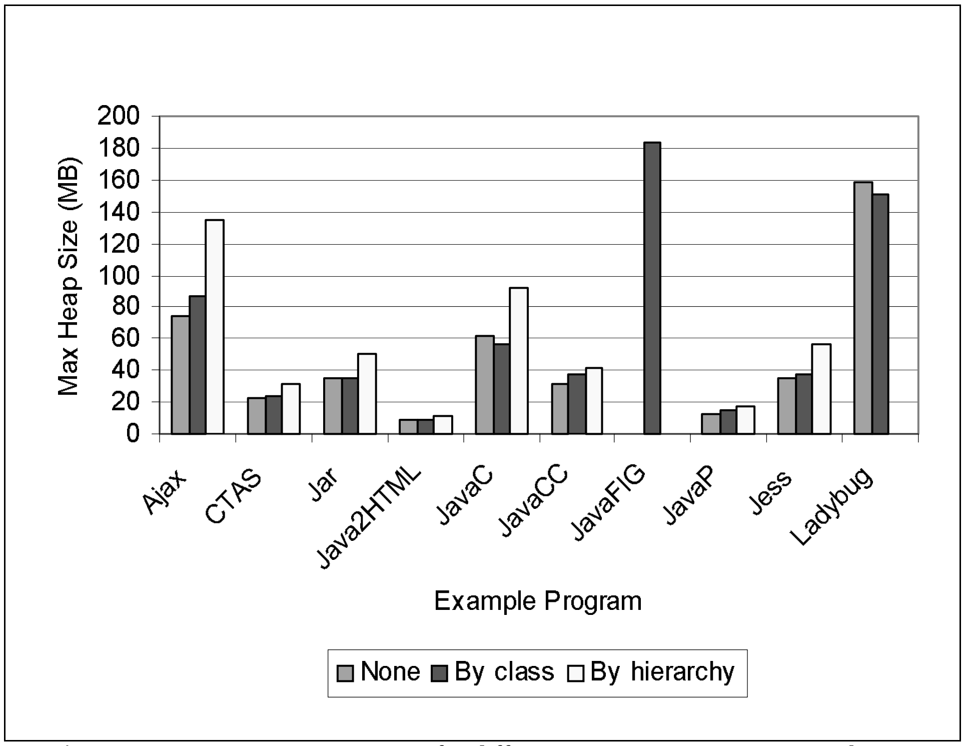


Figure 9-14. Memory consumption for different component partitioning schemes

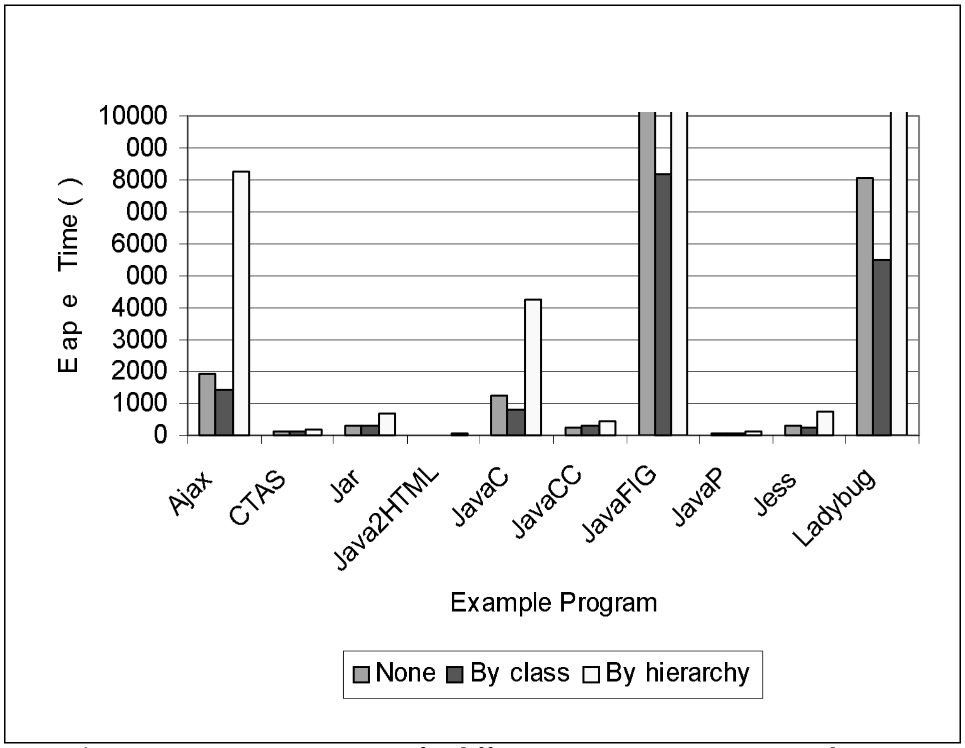


Figure 9-15. Time consumption for different component partitioning schemes

RTA++ is significantly more accurate than SEMI for this task, and SEMI is usually more accurate than RTA++.

RTA++ improves on SEMI because RTA++ can use information about downcasts that SEMI ignores. For example, consider the code in Figure 9-16. SEMI cannot accurately encode the downcast in the type system; downcasts are treated as identity functions. Therefore SEMI infers the same type for `s`, `i`, the contents of `v`, and `s2`, and SEMI concludes that `s2` and `i` may be aliased. However, using the Java type information with RTA++, it is clear that `s2` and `i` are not aliased.

```
void myMethod(Vector v, String s, Integer i) {
    v.addElement(… ? s : i);
    …
    if (…) {
        String s2 = (String)v.elementAt(0);
        …
    }
}
```

Figure 9-16. Example Of RTA++ Improving SEMI

Figure 9-18 gives the same results for live method detection. **This task has the same pattern as virtual call resolution but, as before, the differences are much smaller.** v

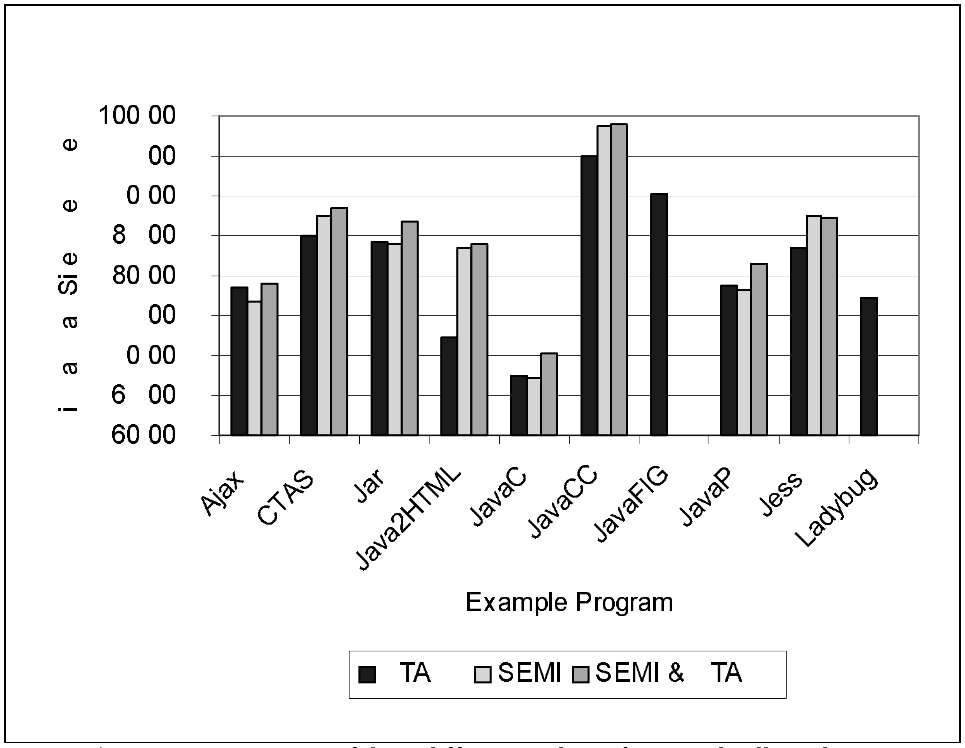


Figure 9-17. Accuracy of three different analyses for virtual call resolution

Figure 9-19 gives the time used for virtual call resolution, for the three analysis. Figure 9-20 gives the space consumed. **SEMI & RTA++ is not much more expensive than running SEMI alone.**

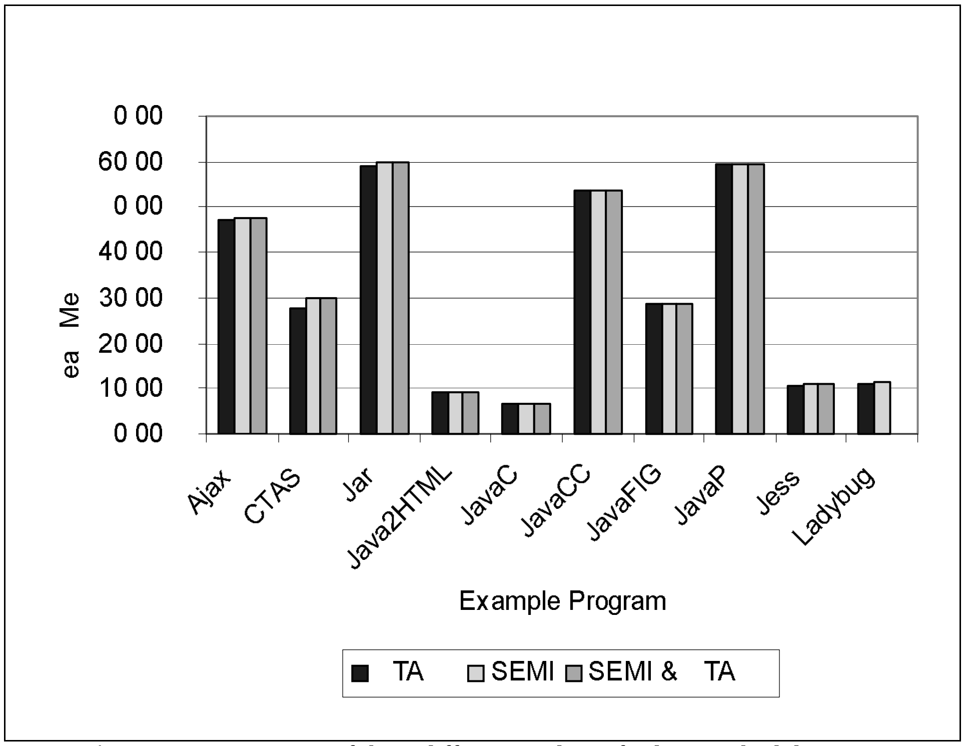


Figure 9-18. Accuracy of three different analyses for live method detection

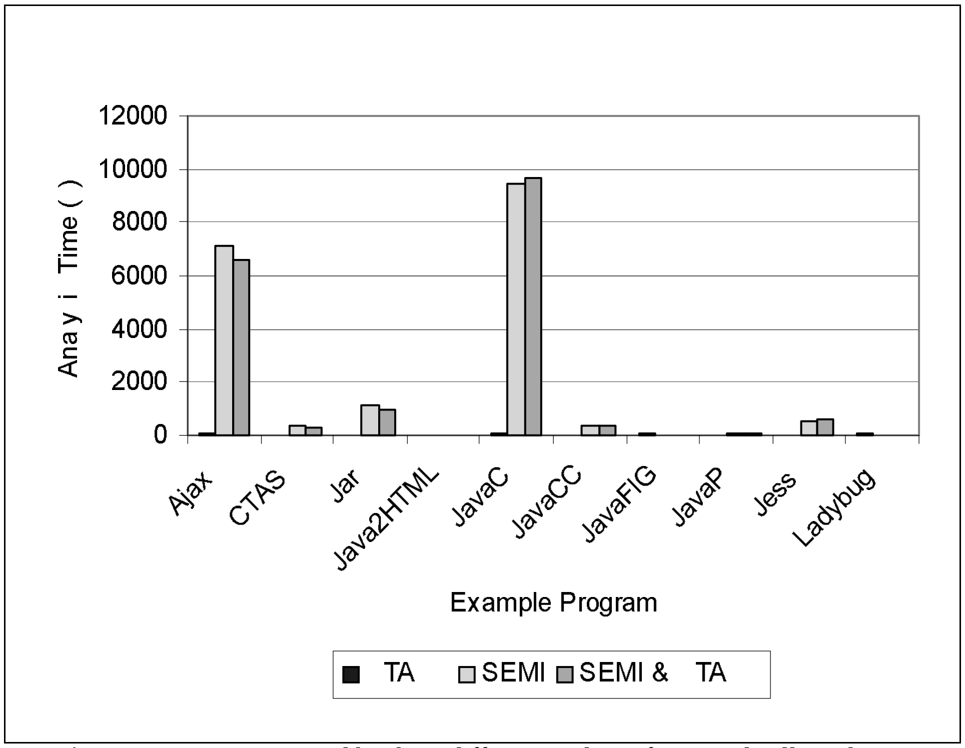


Figure 9-19. Time required by three different analyses for virtual call resolution

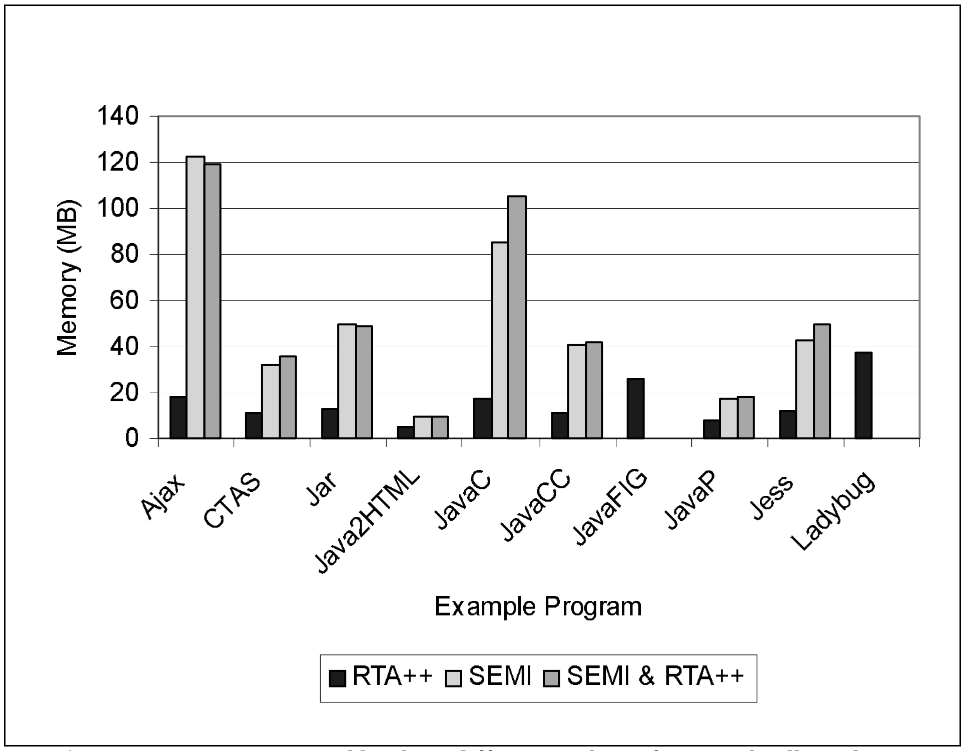


Figure 9-20. Space required by three different analyses for virtual call resolution

9.6.2 Set Sizes

As discussed in Section 4.3.4 and Section 4.4.5, the accuracy of an intersection-based analysis can depend on the maximum size of the data sets allowed by the set abstraction function. Figure 9-21 shows the results of SEMI & RTA++ using different set sizes. **Changing the set size has no practical effect on the accuracy of SEMI & RTA++**.

9.7 Summary of Ajax Performance

9.7.1 Algorithm Selection

Based on the results above, it is clear that the intersection analysis SEMI & RTA++ is preferred over SEMI. It is also clear that, for these tools, polymorphic recursion can be turned off (Section 7.3.6) with little accuracy penalty. SEMI's handling of higher-order code should be enabled if the program being analyzed is not too large.

9.7.2 Summary Results

Now I compare the three algorithms RTA++, SEMI & RTA++ with HighOrder, and SEMI & RTA++ without HighOrder. Figure 9-22 shows the accuracy results for virtual call resolution. Figure 9-23 shows the space requirements and Figure 9-24 shows the time used. **SEMI is far more expensive than RTA++ for large programs, but produces much better results.**

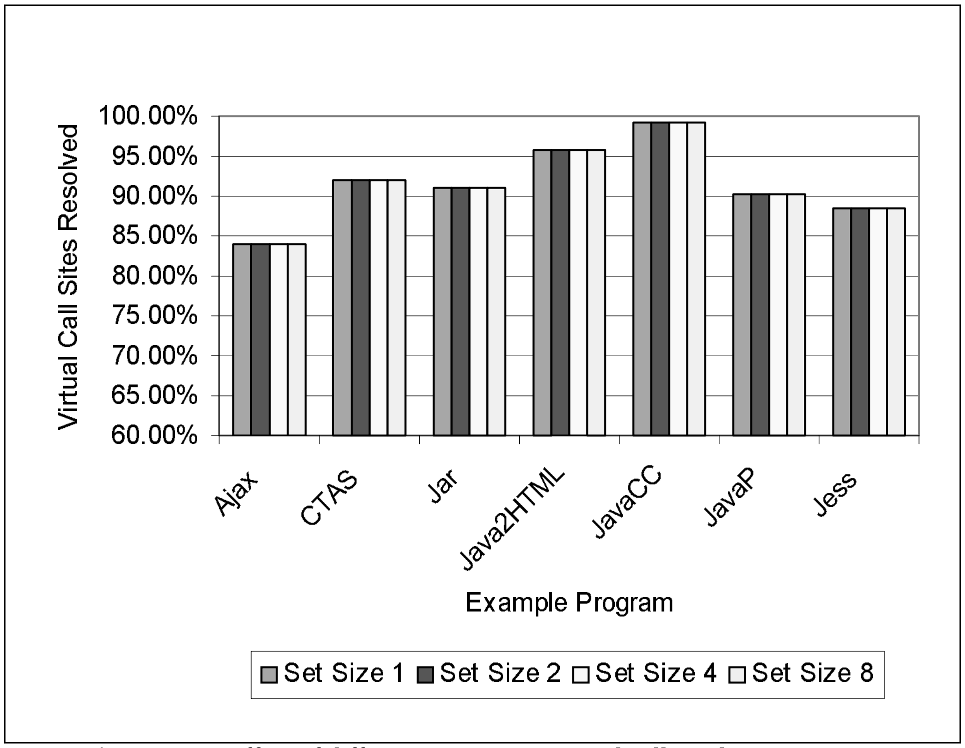


Figure 9-21. Effect of different set sizes on virtual call resolution accuracy

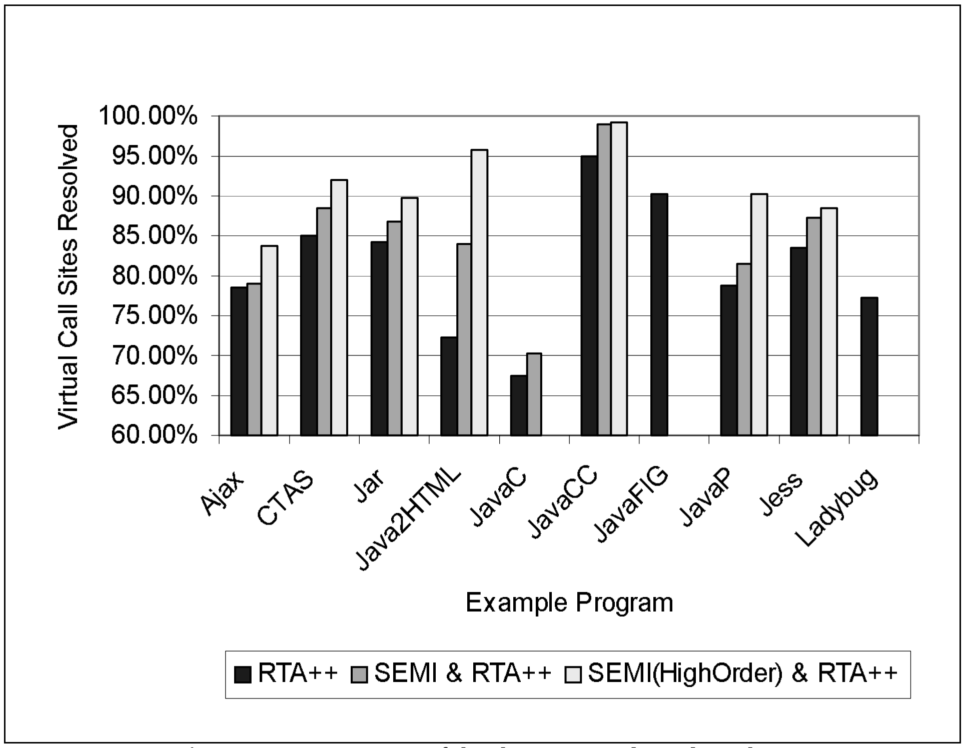


Figure 9-22. Accuracy of the three contending algorithms

9.7.3 Conclusions

Clearly, SEMI is not scalable enough to handle very large programs. The limiting factor is time. However, it does handle realistically-sized programs, and it provides a major

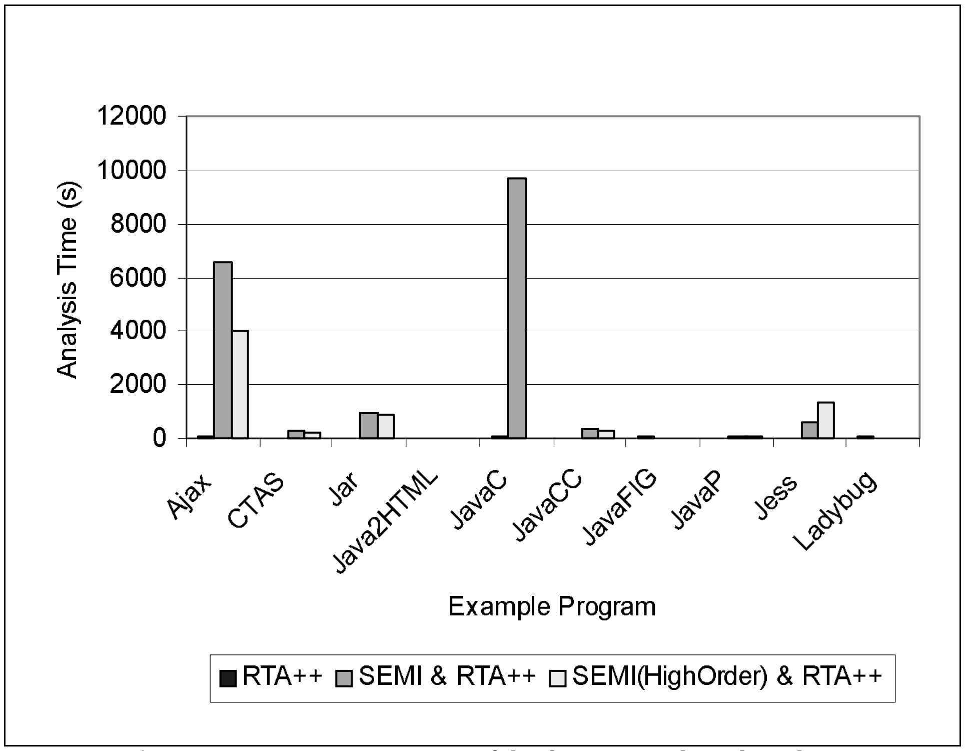


Figure 9-23. Time consumption of the three contending algorithms

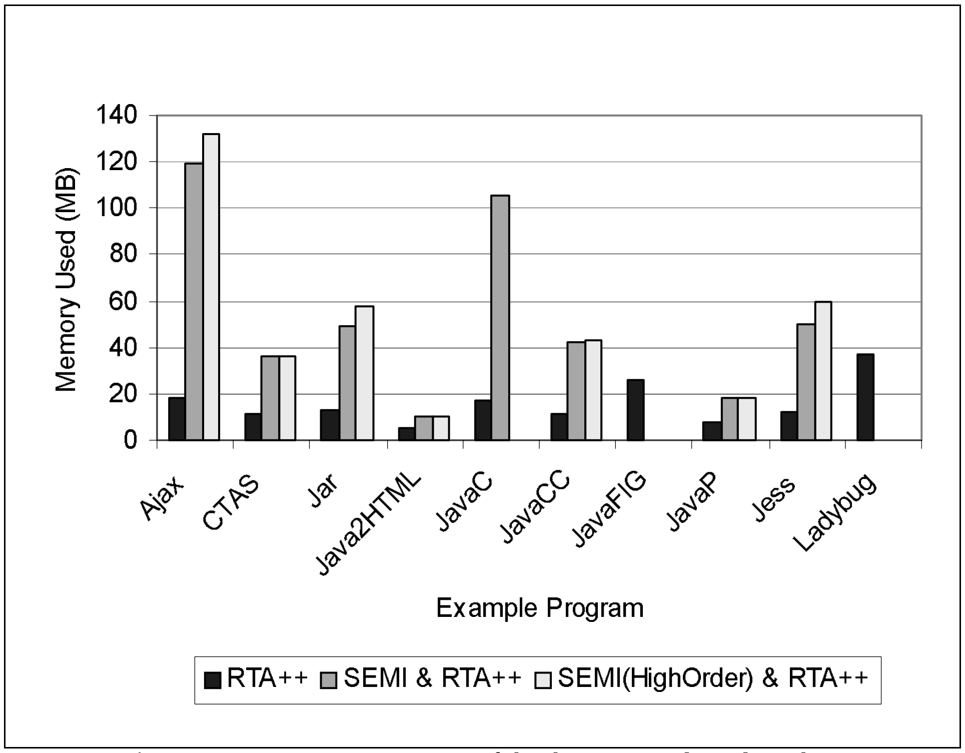


Figure 9-24. Space consumption of the three contending algorithms

improvement over RTA for resolving virtual method calls. The task of identifying dead application code is well solved by RTA and little improvement seems to be possible there.

10 Proving Downcast Safety

10.1 Introduction

10.1.1 Parametric Polymorphism and Downcasts

Java lacks parametric polymorphism. Data structures such as containers, which would be parametrically polymorphic if the language permitted, are usually implemented by replacing the parameter type with some "generic" type which is a supertype of the possible instantiations of the parameter type. For example, a Java container class usually holds references to objects of class `Object`. Methods to insert objects into the collection take a parameter of class `Object`, and methods to extract objects return a value of class `Object`.

For example, consider Figure 10-1. The class `java.util.Vector` declares the methods `addElement` and `elementAt`, among others. To store and retrieve objects of a particular known class, such as String in this case, one must use downcasts.

```
class Vector {
    public Vector() { ... }
    public final synchronized void addElement(Object obj) { ... }
    public final synchronized Object elementAt(int index) { ... }
    ...
}
...
static void main(String[] args) {
    Vector v = new Vector();
    v.addElement(args[0]);
    String s = (String)v.elementAt(0);
}
```

Figure 10-1. Example of a Java generic container requiring downcasts

Without the downcast to `String`, the code will not compile because the result of `elementAt` is not known to be assignable to a `String` object reference. The information needed to prove the assignment safe without the downcast would normally be expressed using parametric polymorphism, but cannot be expressed in Java's type system.

10.1.2 Using SEMI To Prove Downcasts Correct

SEMI is effectively a type inference system with parametric polymorphism. SEMI can reconstruct type parametricity information that Java's type system cannot express. The most straightforward application is to prove that certain downcasts will always succeed. In the example above, Ajax will prove that the downcast to `String` always succeeds. A compiler or run-time system could use this information to eliminate run-time checks associated with the downcast. The programmer is assured that the types of elements in the container are consistent with expectations.

The rest of this chapter presents the design of the Ajax downcast checking tool, which is simple given the Ajax infrastructure. I present some quantitative results on the efficacy of the downcast checker on my example programs. These results also include some interesting comparisons between different analysis configurations. I also discuss some of the especially interesting or problematic pieces of code in the examples. I conclude with a comparison of Ajax downcast checking to support for parametric polymorphism in the language, and a discussion of some other similar ways to use Ajax.

10.2 The Downcast Checking Tool

10.2.1 Interface to the VPR

Section 4.3.3 presents the design of a VPR-based tool for proving downcasts safe. The tool selects a set of occurrences of downcast instructions for analysis; by default, it chooses all the downcasts in the program code found to be live. Then, using the VPR, for each downcast instruction it computes an upper bound in the Java class hierarchy for the classes of all objects that occur as operands to the downcast instruction. This bound is compared to the class specified by the downcast; if the bound is equal to or is a subclass of the specified class, the downcast is reported to be safe.

10.2.2 User Interface

The downcast checking tool is exceptionally simple to use. The user specifies the program to be analyzed by giving a “class path” and the name of the “main” class. The tool then prints out a list of all the downcasts that were found in live code. For each downcast, the tool prints out the location (method name and instruction offset), the class specified by the instruction, the bound actually detected by the analysis, and whether or not the downcast is proven safe.

10.3 Quantitative Results

10.3.1 Proving Downcasts Safe Using RTA++

Section 5.4 describes how RTA is extended with intraprocedural flow analysis to track the use of `instanceof` in conditional expressions, in order to refine the type information known about variables at certain program points. This information can be used to prove the downcast safe in the common “typecase” idiom in Java. For example, given the code

```
if (x instanceof C) {
    C c = (C)x;
    ...
}
```

it is easy for the Ajax downcast checking tool, using RTA++, to prove that the downcast is safe. While this technique has been used by others [18], its effectiveness has not previously been published.

Figure 10-2 shows the percentage of live downcasts proven safe using basic RTA and the RTA++ extension. The results indicate that RTA++ is effective for many programs. Note that even basic RTA can sometimes prove a downcast safe, for example when an abstract class has only one concrete subclass and we downcast from the abstract class to the subclass.

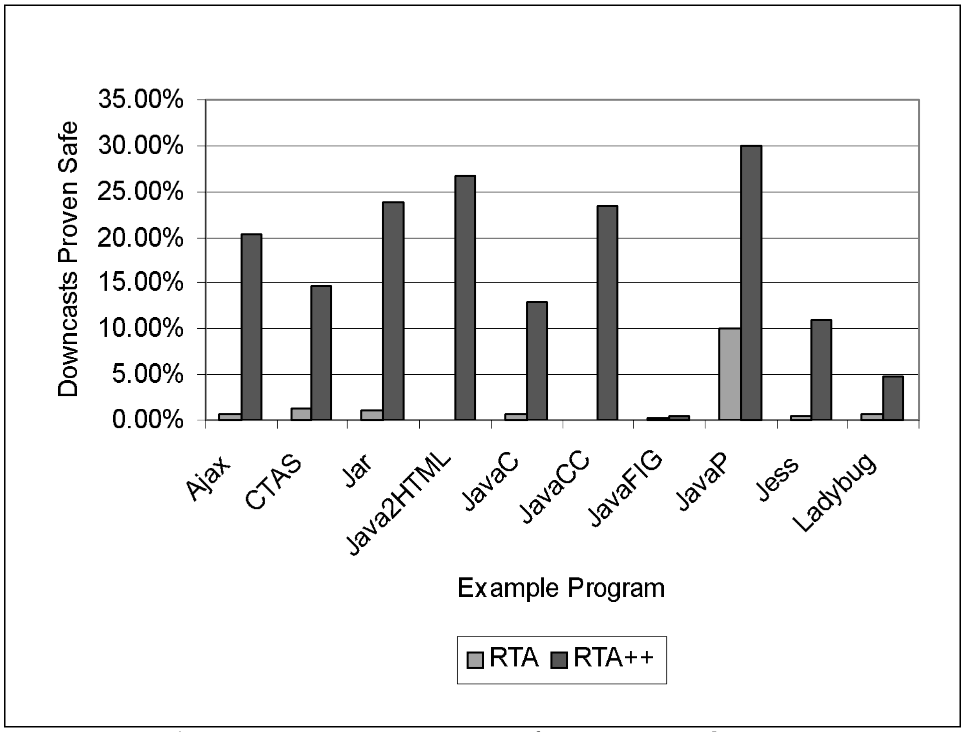


Figure 10-2. Downcasts proven safe using RTA and RTA++

10.3.2 Proving Downcasts Safe Using SEMI

Figure 10-3 shows the results of using SEMI in its four configurations (with or without HighOrder and PolyRec).

In most cases, **SEMI alone is able to prove more downcasts safe than RTA++**, although we will see below that the downcasts it proves safe are different from the ones RTA++ can prove safe. As shown for the tools in the previous chapter, unrestricted polymorphic recursion is not helpful if HighOrder is enabled. However, **when HighOrder is disabled, the situation is different: unrestricted polymorphic recursion significantly improves downcast checking.**

10.3.3 Proving Downcasts Safe Using SEMI with RTA++

Taking the intersection of the information obtained by SEMI with that obtained by RTA++, as described in Section 4.4.5, gives the best of both worlds. Figure 10-4 shows the results of using SEMI & RTA++ (with full polymorphic recursion) compared to SEMI or RTA++ alone.

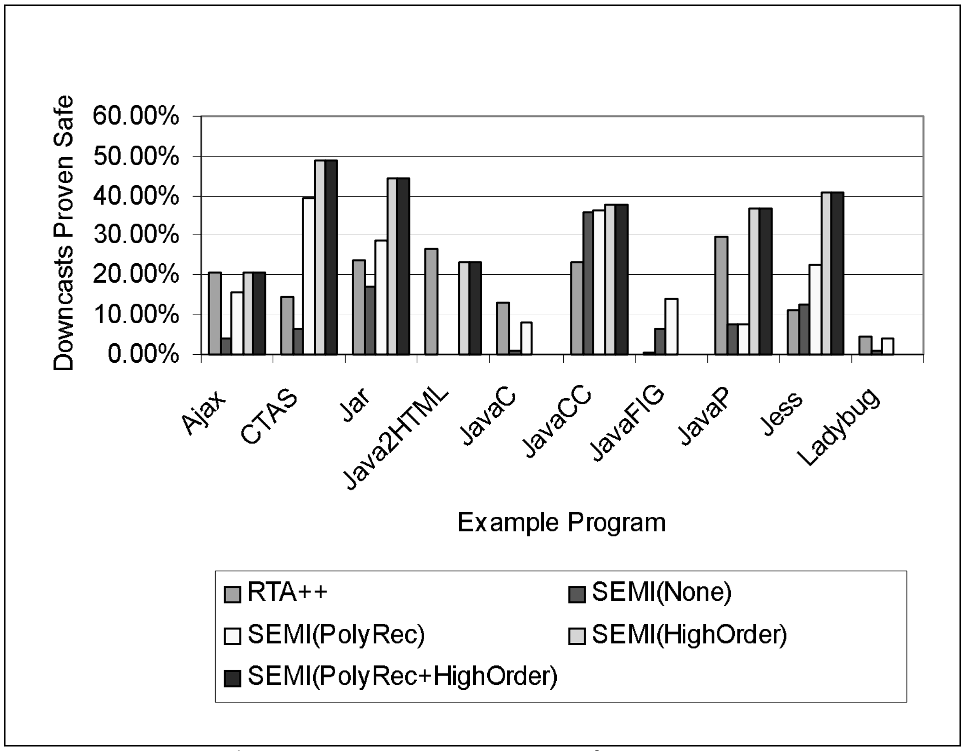


Figure 10-3. Downcasts proven safe using SEMI

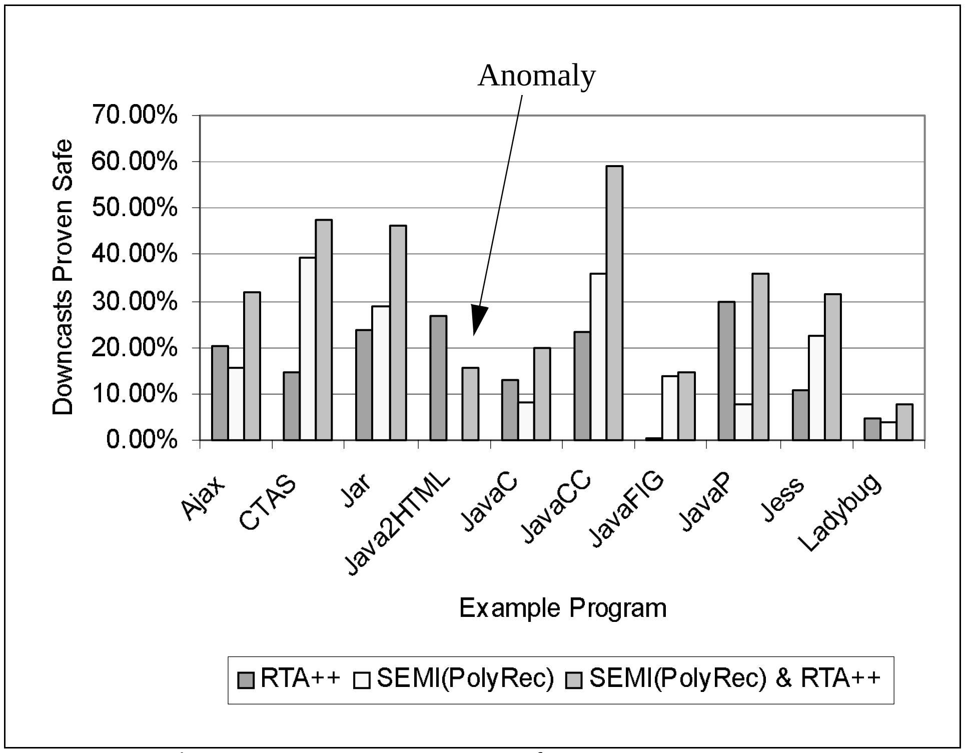


Figure 10-4. Downcasts proven safe using SEMI & RTA++

One can see that the number of downcasts proven safe by SEMI & RTA++ is close to the sum of the downcasts proven safe by SEMI and RTA++. This is unsurprising. To a rough approximation, RTA++ resolves downcasts introduced because Java lacks sum types (see Section 5.4.1), and SEMI resolves downcasts introduced because Java lacks type parametricity.

There is an oddity in the results for the Java2HTML example: SEMI & RTA++ obtains a worse percentage of downcasts proven safe than RTA++ alone. This is because Java2HTML is a very small program; RTA++ finds only fifteen live downcasts and proves four of them safe, but SEMI & RTA++ finds only thirteen live downcasts, proving two of them safe. That is, SEMI & RTA++ proved that two of RTA++'s safe downcasts are actually dead code, and excluded them from its results.

10.3.4 Summary

Figure 10-5 shows the overall results using the best analyses available. The results for SEMI(HighOrder+PolyRec) & RTA++ are almost identical to those for SEMI(HighOrder) & RTA++.

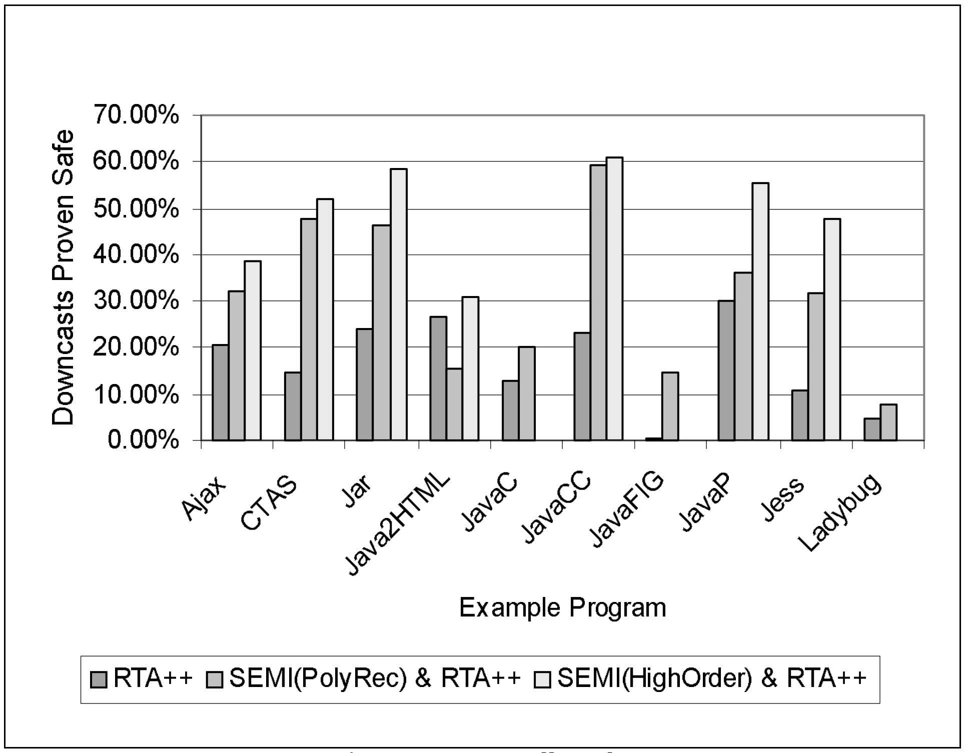


Figure 10-5. Overall results

For some large, realistic programs — Jar, JavaCC, and JavaP — Ajax is able to prove the safety of more than 50% of the downcasts.

Unfortunately, the accuracy seems to deteriorate as programs get larger. Many fewer downcasts are resolved in JavaC, JavaFIG and Ladybug than in the other programs. From these results, it is hard to tell whether this is because of the kind of code people write in larger programs, or whether there is some more subtle reason. Anecdotal evidence suggests that larger programs are more likely to contain sections of “difficult” code that destroy the quality of the analysis results in a non-local way. This is discussed further below.

10.4 Unresolvable Downcasts

I have already mentioned the kind of code for which SEMI & RTA++ can prove downcast safety. In this section I focus on some negative examples — usage patterns for downcasts that SEMI & RTA++ is unable to handle.

10.4.1 Confusion Involving Sum Types

A useful example is Sun’s Java disassembler JavaP. Analyzed by SEMI & RTA++ with polymorphic recursion and higher-order treatment, it is found to have 38 live downcasts of which 21 are proven safe.

One of the downcasts not proven safe is at offset 8 in `sun.tools.util.LoadEnvironment.getClassDeclaration`. This downcast is applied after extracting an object from a `Hashtable` containing `ClassDeclaration`s. The problem is that the same `ClassDeclaration` objects are also placed into a container of general “constant pool items”, which include `Strings`, `Integers` and other constants. The unification behavior of SEMI leads it to conclude that those other constants may also be present in the `Hashtable`. This is one example of a common class of problems: the use of sum types in one context causes inaccuracy in another context. Most of the failures to resolve downcasts in JavaP can be traced back to this problem with the “constant pool”.

Flow sensitive analysis techniques could help to reduce the damage caused by the use of such sums.

10.4.2 “Out Of Band” Dynamic Type Knowledge

Another generally common problem that occurs in JavaP is the use of special knowledge to discriminate sum types. For example, JavaP code often assumes that certain constant pool items have certain types, based on arithmetic invariants governing indices into the constant pool array (e.g., two halves of a 64-bit value are always stored at consecutive locations in the array). It then downcasts to the known type without any guarding `instanceof` check.

Another example is the method

```
sun.tools.java.MethodType.equalArguments(sun.tools.java.Type)
```

The parameter is downcast to a `MethodType` without checking, because other code establishes a precondition that the parameter is indeed a `MethodType`. Propagating such invariants interprocedurally would require more sophisticated analysis than that provided by Ajax.

10.5 Conclusions

10.5.1 Summary

The Ajax downcast checking tool is able to prove more than half of the downcasts correct for some real programs. However, as programs get larger the accuracy decreases. This appears to be because as the program gets larger, there is an increasing chance of encountering some code idiom that pollutes the results for a large fraction of the program. The use of sums is often the culprit.

10.5.2 Other Applications

Proving the safety of downcasts could be useful for Java run-time systems as well as programmers. Many Java programs could be sped up by eliminating the run-time checks.

Another use of this technology would be to reverse engineer type parametricity in existing Java programs, in order to translate them into a language that supports parametericity such as Generic Java [13]. It would not be difficult to implement such a tool based on the tools I have already built.

10.5.3 Limitations of Downcast Checking

Checking downcasts is not the only use of type parametricity information, and checking downcasts does not produce all the benefits that a language with parametric polymorphism provides. For example, in Java it is common to implement a set using a `Hashtable` where objects are put into the `Hashtable`, and the presence of keys is tested using a method returning a boolean value, but no object extraction (and downcasting) ever occurs. Downcast checking will say that everything is safe even if all sorts of different objects are added to the set. In a language with parametric polymorphism, the user could declare the desired element type and the language would detect any usage inconsistent with the declaration.

A completely automatic tool cannot detect such errors. Without user annotations, or at least some heuristics, it is impossible to determine the intended type parametricity of a data structure. If such annotations were available, then it would be easy to design an Ajax tool to check them.

11 Ajax Object Models

11.1 Introduction

In this chapter, I describe what object models mean in Ajax, and how Ajax can construct them. Then I present examples taken from real programs, and discuss the advantages and disadvantages of using Ajax to construct these object models.

11.1.1 Overview of Object Models

An object model is a graph-based abstraction of a set of program states. In this thesis, each node represents a collection of runtime objects that occur in the states. Edges represent relationships between the collections, such as class inheritance and field reference.

For example, Figure 11-1 shows an object model for the program in Figure 11-2. A dotted edge indicates an inheritance relationship. A solid line represents a field edge, labelled with the name of the referring field. Each node is labelled with the class name of the objects it represents. For example, from this diagram we can see at a glance that `X` has two fields referring to `Y` objects, some of which may actually be of class `Z`.

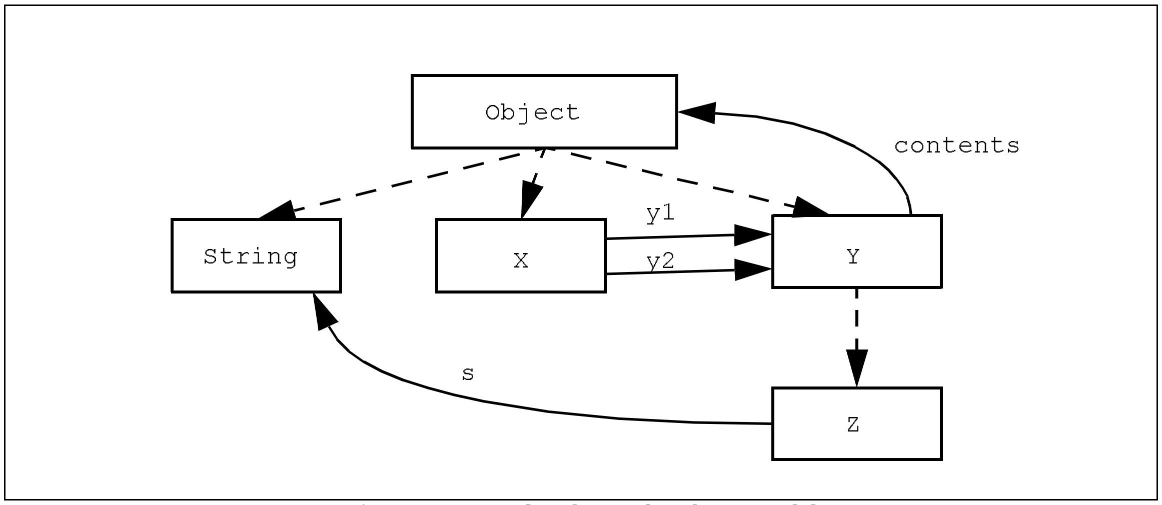


Figure 11-1. A class hierarchy object model

This object model was obtained directly from the program's class declarations. However, more elaborate object models are possible and useful. For example, Figure 11-3 shows another object model for the same program. This object model reveals more information, such as the fact that `X`'s `y1` and `y2` fields both refer specifically to objects of class `Y` and not `Z`. This information cannot be obtained from the class declarations alone; different objects of class `Y` must be represented by different nodes.

```
class X {
    Y y1;
    Y y2;
    X() {
        y1 = new Y(this);
        y2 = new Y(... ? new Z() : this);
    }
    static void main(String[] args) {
        X x = new X();
    }
}
class Y {
    Object contents;
    Y(Object p) {
        contents = p;
    }
}
class Z extends Y {
    String s = "Football";
    Z() {
        super(s);
    }
}
```

Figure 11-2. An example Java program

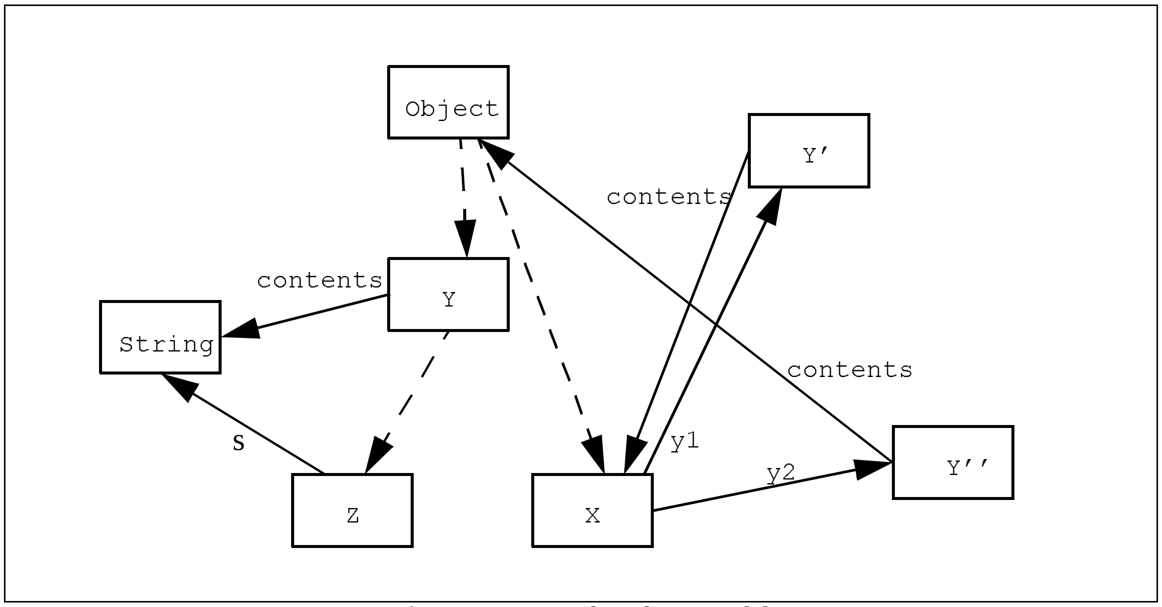


Figure 11-3. A richer object model

An object model is a directed graph. Each node in the graph is *associated* with a set of runtime objects. There are two kinds of edges: *field edges*, labelled with field names, and *inheritance edges*, which are unlabelled. A field edge from A to B labelled F indicates that at least one of A's objects has a field F containing a reference to an object in B. An inheritance edge from A to B indicates that B's objects are a subset of A's objects.

The class hierarchy of a Java program can be interpreted as an object model. Each node corresponds to a class C, and is associated with the set of objects of class C or some subclass of C. Field edges are drawn from C's node to the nodes corresponding to the declared class types of the object reference fields declared in C. Inheritance edges are drawn from each class to its subclasses.

Object models visualize the structure of a program's data. In object-oriented programs, the structure of the data reflects the overall organization of the program. Programmers can use object models to capture this organization graphically.

An object model can be thought of as a static projection of all possible runtime heap states of a program.

11.1.2 A Definition of Object Models

The following definition is as flexible as possible to accommodate various ideas about what an object model is, how it can be constructed, and how it can be used.

The class hierarchy object model has the following properties:

1. The field edges are sound; field relationships in all program states are reflected in the model. Formally, if in some program state an object O_1 has a field F containing a reference to object O_2, and O_1 and O_2 are represented in the model (i.e., they are associated with at least one node), then there are nodes A and B and a field edge from A to B labelled F such that O_1 is associated with A and O_2 is associated with B.

   For example, in Figure 11-3, in the final program state, `x.y2` refers to an object associated with the `Y''` node. Since the object `x` is associated with node X, an edge labelled `y2` must be drawn from node `X` (or node `Object`) to node `Y''`.
2. Inheritance edges obey the subset relationship: if O_1 is associated with node A, and there is an inheritance edge from A to B, then O_1 is associated with node B.

   In Figure 11-3, all objects associated with node `Z` must also be associated with the `Y` node and the `Object` node.
3. Every object has a "most specific" node: if O is associated with nodes A and B, then there is a node C such that O is associated with C and there is a path in the inheritance edges from A to C and from B to C

   The most specific node for `x` in the example is the node labelled `X`. There is a path from the other node associated with `x` (`Object`) to the most specific node.
4. If there is a field edge E from A to B labelled F, and a node C such that there is a path in the inheritance edges from C to A, and C has an outgoing field edge labelled F, then A equals C and that edge is E itself.

   For example, it would not be permissible to have an edge emanating from node `Y` labelled `s`, unless the `s`-edge emanating from node `Z` was deleted.

We take these properties as definitional, and call any graph satisfying them an object model. Property 1 is useful because it assigns meaning to the field edges of the graph — more precisely, it assigns meaning to the *absence* of field edges in the graph. Properties 2 and 3 impose structure on the associations between nodes and objects; in particular property 3 means that given a map from each object to its "most specific" node (e.g., its class), we can find all the nodes associated with any given object. Property 4 guarantees that each field of an object maps to at most one edge in the model.

The class hierarchy model has the following additional "completeness" properties:

5. Objects are complete: given an object O_1 containing a field F, a node A such that O_1 is associated with node A, and an object O_2 such that O_1.F = O_2, then for some node B there is an edge in the model from A to B labelled F.
6. All objects are included: given an object O_1, there is a node A such that O_1 is associated with node A.

A useful object model need not satisfy these properties. The object models created by Ajax satisfy property 5 but not property 6.

11.2 Computing Object Models with Ajax

Ajax includes an object modelling tool based on the VPR. Building object models requires extensive post-processing of the raw value-point relation. This section describes this processing, first giving the series of steps required, and then elaborating on the difficult steps.

11.2.1 Overview

Previous work on object model construction [46] starts with a class hierarchy and applies transformations to obtain more refined models. In contrast, Ajax builds a refined object model and then applies transformations to simplify the model.

- Ajax first constructs a simple model that uses no inheritance edges and does not obey property 4 (unique field edges). The model associates each object with at most one node. This model is simply a conservative static approximation to the heap graph reachable from a given set of "root objects", specified by bytecode expressions provided by the user. Property 5 ("object completeness") is obeyed, but not property 6 (because not all objects are included). The construction of this heap graph is described in more detail in Section 11.2.2.

  Figure 11-4 gives this basic model for the program in Figure 11-2. The root objects are the objects evaluated to by the expression `x` in the `main` method. Note that the node "some other `Y`" has two outgoing edges labelled `contents`, violating property 4.
- Next, a simple object model is obtained from the heap graph by merging nodes in order to satisfy property 4. That is, whenever we have a node A with two outgoing field edges labelled F to nodes B and C, we merge nodes B and C and delete one of the field edges.

  In the example, Ajax merges the `X` and `Z` nodes; see Figure 11-5.

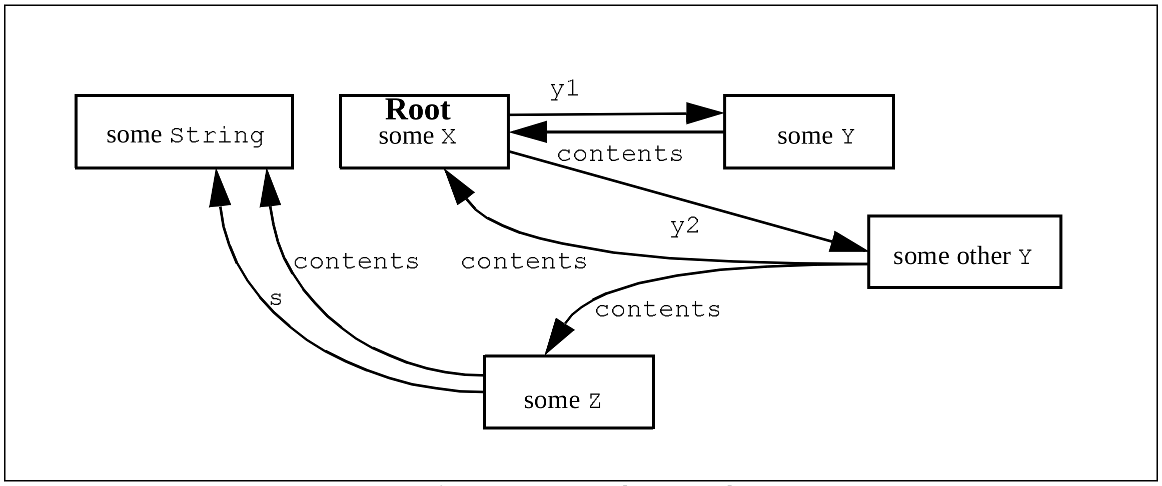


Figure 11-4. Ajax heap graph

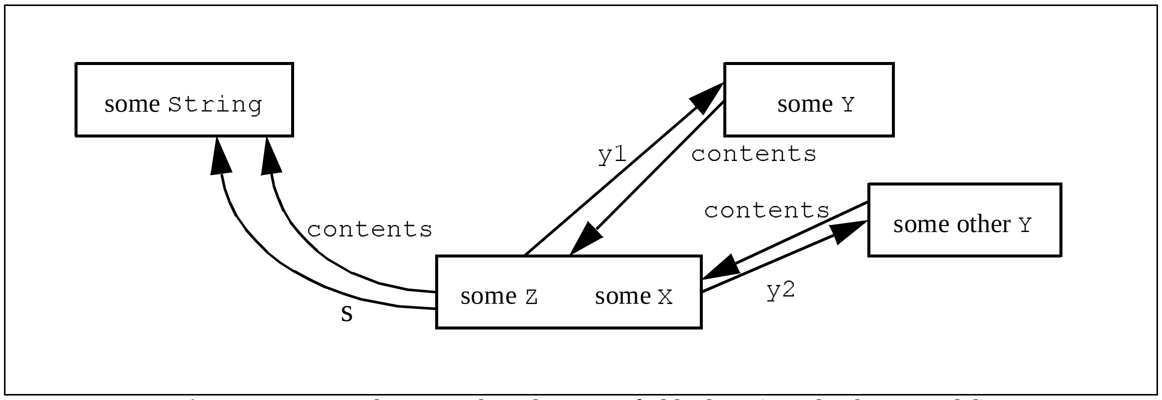


Figure 11-5. Ajax heap graph with unique field edges (simple object model)

- In the next pass, each node explodes into a set of subnodes, one for each class of objects associated with the node and one for each of their superclasses. An inheritance edge is introduced between each class and its superclass. The origin of each field edge is set to the subnode for the class in which the field is declared. The target is the subnode of the original target node for the class the field is declared as.

  See Figure 11-6. The rounded boxes group the subnodes extracted from each original node. For example, the node “some `Z`, some `X`” is exploded into four nodes: one for class `Z`, one for class `X`, and one each for their superclasses `Y` and `Object`. The edge for field `y1` has its origin at the subnode for `X`, because field `y1` is declared in class `X`. The edge points to the subnode for class `Y` because `y1` is declared as class `Y`.

- Sometimes the target of a field edge is known to be of a more specific class than the declared class. (This information is obtained by a separate Ajax query to compute the most specific common superclass of the target objects.) The field edge is retargeted to the more specific class.

  For example, in Figure 11-6, `Z`’s field `contents` is known to contain only `Strings`. The edge is updated to point to the `String` node.

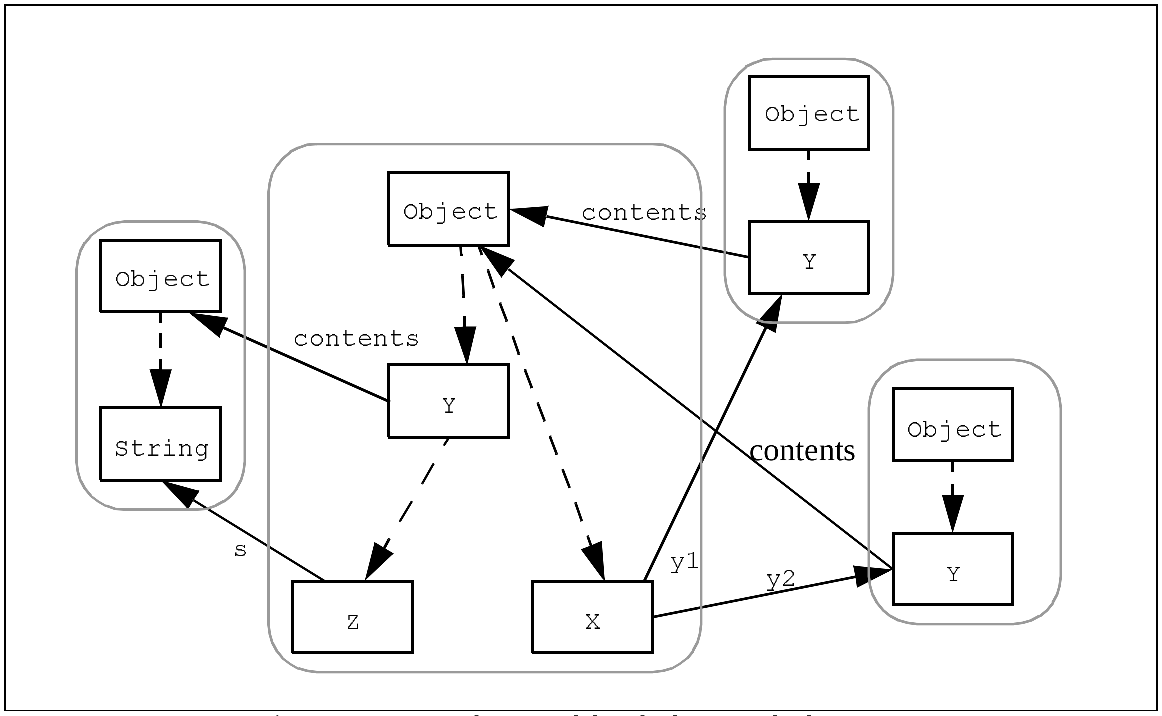


Figure 11-6. Ajax object model with classes and inheritance

- In Figure 11-6, three of the `Object` nodes are not useful because the only edges incident to them are outgoing inheritance edges. All such nodes are deleted, giving Figure 11-7. Since this can create more nodes incident only to outgoing inheritance edges, the operation is repeated until no applicable nodes remain. Other pruning can also be performed at this stage; this is discussed in more detail in Section 11.2.3.

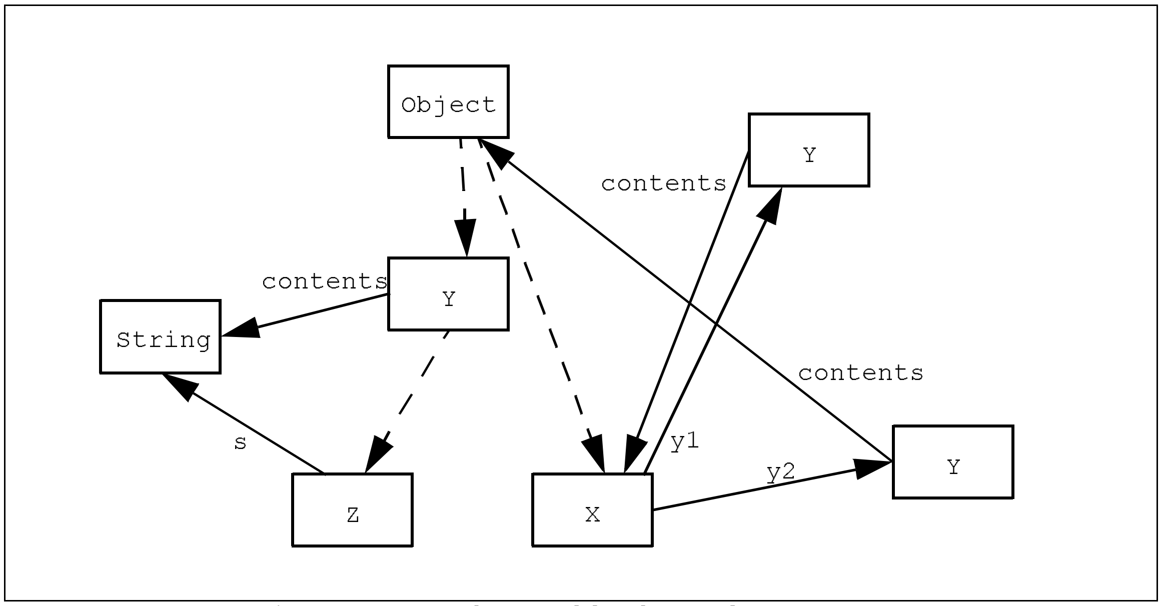


Figure 11-7. Ajax object model with superclass suppression

In a final (optional) pass, Ajax identifies isomorphic subgraphs within the model and merges them to save space. Figure 11-7 does not contain any isomorphic subgraphs; therefore it is the graph produced by Ajax for the example program. This is the same model shown in Figure 11-3.

11.2.2 Computing Heap Graphs With The VPR

The first step is to construct a heap graph. Clearly the VPR is not a natural encoding of a heap graph; we must extract a heap graph using Ajax queries.

11.2.2.1 Approach

Suppose a "root expression" *exp* is given. This expression can be chosen by the user as described in Section 11.2.4.

Ajax constructs a heap graph with a root node representing the objects to which *exp* evaluates. Then, for each field name F in the program, it checks whether *exp*.F $\leftrightarrow$ *exp*.F. If not, then the objects for the root node never have a field F, or their F fields always contain null. Otherwise Ajax adds a field edge labelled F, emanating from the root node and pointing to a new node — the node representing objects evaluated to by "*exp*.F". We repeat this procedure, taking each new node and adding outgoing edges for its fields, building a tree representing the objects reachable from the root objects.

Many nodes in the tree may correspond to overlapping (or identical) sets of objects. Therefore we test, for each pair of nodes, whether the expressions associated with the nodes are related by the value-point relation. If the expressions are related then we merge the nodes. This means that the tree may become a general graph.

11.2.2.2 Method

The procedure is shown in Figure 11-8.

It is impractical to build such a tree and then subsequently merge the nodes. The initial tree is simply too large, and in the case of cyclic data structures, it may even be infinite. Instead, before creating a new node (label `3`), Ajax checks to see whether the node's expression is related to any of the expressions associated with already existing nodes (label `1`). If so then the new node need not be created; the matching existing node is used instead (label `2`).

11.2.2.3 Correctness

Using the standard value-point relation, the above procedure is not sound. It assumes that when two nodes are related in the VPR, they have exactly the same behavior. More precisely, the algorithm above is only correct if the VPR has the *substitutability property*:

$$\forall e_1, e_2.\ e_1 \leftrightarrow e_2 \Rightarrow (\forall e.\ e_1 \leftrightarrow e \Leftrightarrow e_2 \leftrightarrow e) \wedge (\forall e, F.\ e_1.F \leftrightarrow e \Leftrightarrow e_2.F \leftrightarrow e)$$

This means that if e_1 and e_2 are related, substitution of one for the other does not change whether an expression pair is in the VPR.

This property is not implied by the definition of the VPR. Consider the example in Figure 11-9. According to the VPR, `f:x` $\leftrightarrow$ `f:y` and `f:y.length` $\leftrightarrow$ `f:len`. However, substituting `x` for `y`, `f:x.length` $\leftrightarrow$ `f:len` does not hold. Informally, the reason is that

```
Initialize the graph G to contain a single node, the root
Let M be a map from G's nodes to expressions
Initialize the map M to map the root node to exp
Repeat {
  For each field F in the program {
    For each node N1 in G {
      If M(N1).F <-> M(N1).F is in the VPR {
1:      For each node N2 in G {
          If M(N1).F <-> M(N2) is in the VPR {
            If there is no edge from N1 to N2 labelled F {
2:            Add to G an edge from N1 to N2 labelled F
            }
          }
        }
        If N1 has no outgoing edge labelled F {
3:        Create a new node N
          Extend M with a mapping from N to M(N1).F
          Add to G an edge from N1 to N labelled F
        }
      }
    }
  }
} Until G does not change
```
Figure 11-8. Basic heap graph construction algorithm

```
static void f(Object x, Object y, int len) {
}
static void main(String[] args) {
  String[] zoo = { "lion", "tiger" };
  f(zoo, zoo, args.length);
  f(zoo, args, args.length);
}
```
Figure 11-9. Example of substitutability violation

the two antecedent relation pairs hold in different contexts, so no conclusion can be drawn from their conjunction.

11.2.2.4 Solution

Therefore, the object modelling tool notifies the analysis that it must produce a VPR approximation satisfying the substitutability property. For increased flexibility, the tool specifies a program point *l* at which expressions must be substitutable; all other expressions need not be substitutable. The exact property demanded is:

$$\forall e_1, e_2.\ l{:}e_1 \leftrightarrow l{:}e_2 \Rightarrow$$
$$(\forall e, F.\ l{:}e_1.F \leftrightarrow e \Leftrightarrow l{:}e_2.F \leftrightarrow e) \wedge (\forall e.\ l{:}e_1 \leftrightarrow e \Leftrightarrow l{:}e_2 \leftrightarrow e)$$

This suffices because all queries required to build the heap graph are based on one or more root expressions, which are all at the same program point. Limiting the property to one program point means that other queries using the same VPR approximation (e.g., the liveness query used to limit the scope of the analysis) are not seriously impacted.

11.2.2.5 Implementing Substitutability In RTA++

It is easy to enforce substitutability in RTA++. We simply assign the static bytecode type TOP to any expression of the form $l{:}e$, where l is the program point where substitutability is required. This ensures that every such expression is related to all other expressions in the computed VPR.

This approximation is not particularly useful, because it implies $l{:}e_1 \leftrightarrow l{:}e_2$ regardless of the values of e_1 and e_2, so using RTA++ alone, the heap graph will collapse to a point. Unfortunately it is necessary. For suppose that for some e, $l{:}e$ has Java type `Object`. (The existence of such an e is almost certain in practice.) Then for any e_1 and e_2 such that $l{:}e_1$ and $l{:}e_2$ have Java class types, RTA++ will give $l{:}e \leftrightarrow l{:}e_1$ and $l{:}e \leftrightarrow l{:}e_2$. The substitutability property then requires that $l{:}e_1 \leftrightarrow l{:}e_2$.

Therefore RTA++ alone is not suitable as the analysis engine for the Ajax object modeling tool.

11.2.2.6 Implementing Substitutability In SEMI

Suppose that $l{:}e_1$ and $l{:}e_2$ both map to SEMI constraint variables that have no instance constraints emanating from them. Then in SEMI, $l{:}e_1 \leftrightarrow l{:}e_2$ if and only if $l{:}e_1$ and $l{:}e_2$ map to the same constraint variable. If indeed they map to the same constraint variable, the substitutability property is satisfied for $l{:}e_1$ and $l{:}e_2$, because SEMI's VPR is a function of the constraint variables mapped to by the expressions.

Therefore, to enforce the substitutability property in SEMI, I force all expressions of the form $l{:}e$ to have no instance constraints emanating from them, by forcing their constraint variables to be global (see Section 7.6.3).

11.2.2.7 Improving The Heap Graph Algorithm

The algorithm described above is rather inefficient. The implementation of the object modelling tool speeds it up by exploiting the power of the Ajax interface. The algorithm is presented in Figure 11-10.

The improved algorithm uses a series of iterations. It maintains a set of "fringe" nodes, the nodes added in the last iteration (set `T`). At each step, the fields of the fringe nodes are examined and potential new target nodes for those fields are created (label `1`). A new node that is related to an existing node is merged into the existing node (label `2`). New nodes that are related to each other are merged (label `4`). New nodes that are not even related to themselves are deleted (label `3`). (The field never refers to any objects.) Surviving new nodes are added to the graph (label `5`) and become the new fringe set.

11.2.2.8 Reducing Space Consumption

The above algorithm exploits the Ajax interface, but peak memory usage can still be very large: accumulating the complete set of source nodes matching each target node can require space quadratic in the number of candidate new nodes.

Another improvement to the algorithm reduces peak space consumption. The basic idea is to compute just one or two elements of the set of source nodes reaching each target node. This is enough information to merge nodes. The query repeats several times, merging nodes

```
Initialize the graph G to contain a single node, the root
Let S (the fringe set) contain the root node
Let M be a map from G’s nodes to expressions
Initialize the map M to map the root node to exp
While S is nonempty {
  Let T be the new fringe set, initially empty
  Let T_M be an empty map from T’s nodes to expressions
  Let P be an empty map from nodes to sets of (node, field) pairs
  // P(n) records edges to be created pointing into node n

  For each nonstatic field F in the program {
    For each element S_e of S {
1:      Create a new node N
        Add N to T
        Extend T_M with a mapping from N to M(S_e).F
        Extend P with a mapping from N to {(S_e, F)}
    }
  }


  // Begin query processing
  Run a query with the following parameters:
    sources = T_M
    targets = M U T_M
    R = results = for each target node, the set of source nodes
whose expressions are related to the target node’s expression

  // Any new nodes that are related to existing nodes are
  // replaced by the existing nodes
  For each node G_e in G {
    Extend P with a mapping from G_e to {}
    For each element T_e of R(G_e) {
      If T_e is still in T then {
2:        Extend P with a mapping from G_e to P(T_e) U P(G_e)
          Delete the mapping for T_e from P
          Delete T_e from T and T_M
      }
    }
  }
```
Figure 11-10. More efficient heap graph construction algorithm

```
   For each node T_e in T {
     // New nodes that aren't even related to themselves are dead
     If R(T_e) is empty then {
3:     Delete T_e from T and T_M
       Delete the mapping for T_e from P
     } else {
       For each element T_r of R(T_e) {
         If T_r is still in T and T_r is not equal to T_e {
4:         // Merge T_r into T_e because they're related
           Extend P with a mapping from T_e to P(T_e) U P(T_r)
           Delete the mapping for T_r from P
           Delete T_r from T and T_M
         }
       }
     }
   }
   // End query processing

   Let S = T
   For each node N in the domain of P {
     Extend M with a mapping from N to T_M(N)
     For each element (S_e, F) of P(N) {
5:     Add an edge to G from S_e to N labelled F
     }
   }
}
```

Figure 11-10. More efficient heap graph construction algorithm

after each iteration, until the algorithm converges to the same state it would have reached in one step of the previous algorithm.

There are two kinds of queries. Each query is parameterized by a set of source expressions and a set of target expressions. For each target expression e_1, the first kind of query computes and returns a source expression e_2 such that $e_1 \leftrightarrow e_2$, or returns "unknown" if no such e_2 exists. The second kind of query computes and returns two distinct source expressions e_2 and e_3 such that $e_1 \leftrightarrow e_2$ and $e_1 \leftrightarrow e_3$ (it may also return just one expression or "unknown" if two such expressions do not exist). These queries are implemented in the Ajax framework similarly to the abstract set query in Section 4.3.4, except that when a set overflows its bound, its current contents are remembered and propagated. For example, for the second kind of query, the result of { e_2 } merged with { e_3, e_4 } could be abstracted to "at least { e_2, e_3 }".

Note that if intersection operations are applied to this "bounded set" query data, we may have a result consisting of an "overflowing" set but with no elements known to be in the set. (For example, consider the intersection of the abstract set "at least { e_1 }" with the abstract set "at least { e_2 }"; the result can only be "at least {}".) This information is not useful to the heap graph algorithm. Therefore this implementation of the object modeling tool does not work with multiple intersecting analyses.

The query processing of the above algorithm is modified as shown in Figure 11-11. In practice few iterations of the inner loop are required.

```
  // Begin query processing
  Run a query of the first kind with the following parameters:
    sources = T_M
    targets = M U T_M
    R = results = for each target node, 0-1 source nodes
whose expressions are related to the target node’s expression

  // Any new nodes that are related to existing nodes are
  // replaced by the existing nodes
  For each node G_e in G {
    Extend P with a mapping from G_e to {}
    For each element T_e of R(G_e) {
      If T_e is still in T then {
        Extend P with a mapping from G_e to P(T_e) U P(G_e)
        Delete the mapping for T_e from P
        Delete T_e from T and T_M
      }
    }
  }

  For each node T_e in T {
    // New nodes that aren’t even related to themselves are dead
    If R(T_e) is empty then {
      Delete T_e from T and T_M
      Delete the mapping for T_e from P
    } else {
      For each element T_r of R(T_e) {
        If T_r is still in T and T_r is not equal to T_e {
          // Merge T_r into T_e because they’re related
          Extend P with a mapping from T_e to P(T_e) U P(T_r)
          Delete the mapping for T_r from P
          Delete T_r from T and T_M
        }
      }
    }
  }
```
Figure 11-11. Heap graph construction algorithm with reduced peak space consumption

```
  Repeat {
    Run a query of the second kind:
      sources = T_M
      targets = T_M
      R = results = for each target node, 0-2 source nodes
whose expressions are related to the target node’s expression

    For each node T_e in T {
      For each element T_r of R(T_e) {
        If T_r is still in T and T_r is not equal to T_e {
          // Merge T_r into T_e because they’re related
          Extend P with a mapping from T_e to P(T_e) U P(T_r)
          Delete the mapping for T_r from P
          Delete T_r from T and T_M
        }
      }
    }
  } until R(T_e) = { T_e } for every T_e in T
  // End query processing
```

Figure 11-11. Heap graph construction algorithm with reduced peak space consumption

11.2.3 Lossless Improvement to the Model

After constructing the heap graph and elaborating it with class and field information, the object model may contain superfluous nodes that can be eliminated.

11.2.3.1 Superflous Leaf Classes

Field edges can be retargeted from their declared classes to some actual class that is more specific than the declared class. In the example of Figure 11-12, the analysis engine may suggest that the `name` field refers to an abstract object which could be an `Integer` or a `String`, but since the `name` field is declared to be a `String` and no other fields reference the abstract object, the `name` field is retargeted to `String`. This can leave nodes such as `Integer` which are not reachable, i.e., no field edge points to the class or any of its super-classes or subclasses.

Such nodes can never correspond to real objects in the program, so they can be deleted. In the example, the `Integer` subclass can be removed. (The `Object` superclass can then also be hidden.) These nodes can occur because of inaccuracy in the underlying analysis engine.

11.2.3.2 Merging Identical Subgraphs

Consider the example on the left hand side of Figure 11-13. Suppose a programmer is interested in discovering the Java types of the objects that may be (indirectly) referenced by `Orb`, and which field dereference paths are involved.

Clearly it is unnecessary to distinguish the two `Vectors` for this task — the fact that the two `Vectors` are not aliased is not important. In this case, one can save space in the model by merging identical subgraphs. The Ajax object modeling tool provides this as an option. The above example would be reduced as shown in Figure 11-13.

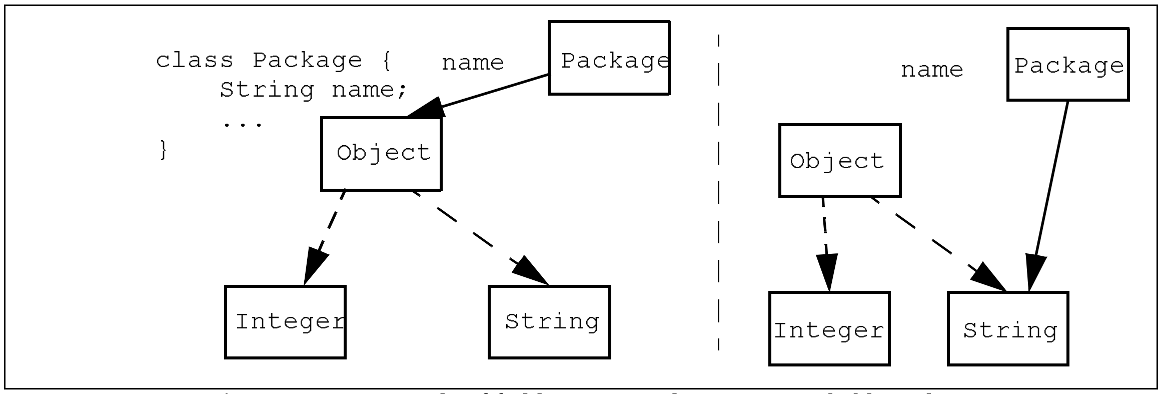


Figure 11-12. Example of field retargeting leaving unreachable nodes

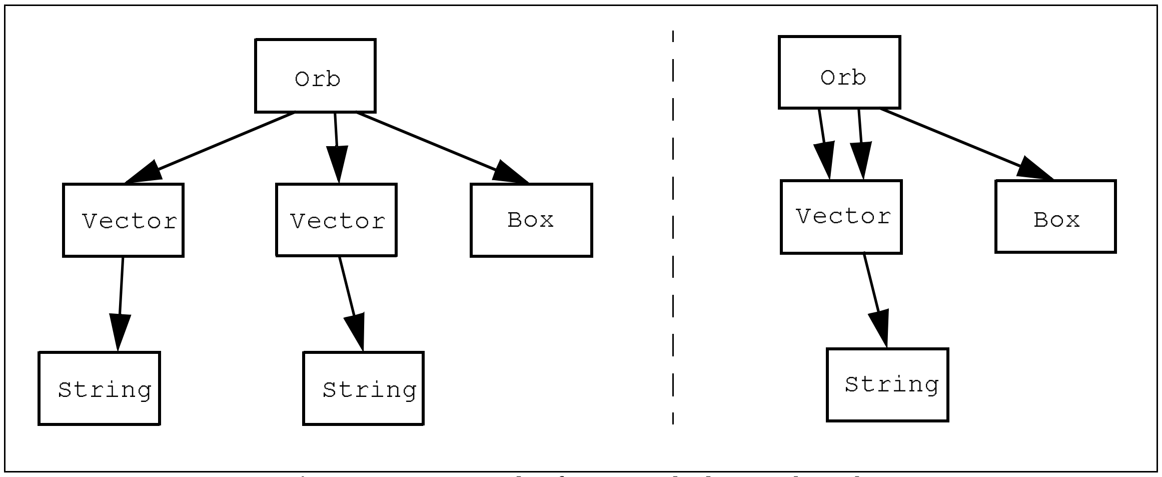


Figure 11-13. Example of merging duplicate subgraphs

11.2.4 User Interface

The Ajax object modeling tool has a simple user interface. The user specifies the program to be analyzed by giving the “class path” and the name of the “main” class. By default, the tool uses as root expressions all the local variables at the last instruction in the main class reachable by non-exceptional control flow. The user can specify an explicit root expression instead, if desired. The tool computes the model and outputs the results in a format suitable for processing by AT&T’s `dot` tool for graph layout [36].

11.3 Examples

11.3.1 JavaP Example

Figure 11-14 shows the object model produced by Ajax applied to Sun’s JavaP disassembler tool. Isomorphic subgraphs have not been merged. This example clearly shows the strengths and limitations of the Ajax object modeling tool.

This model uses the default set of root expressions — all the local variables at the last instruction in `JavaP.main` reachable by non-exceptional control flow. The tool uses the SEMI analysis.

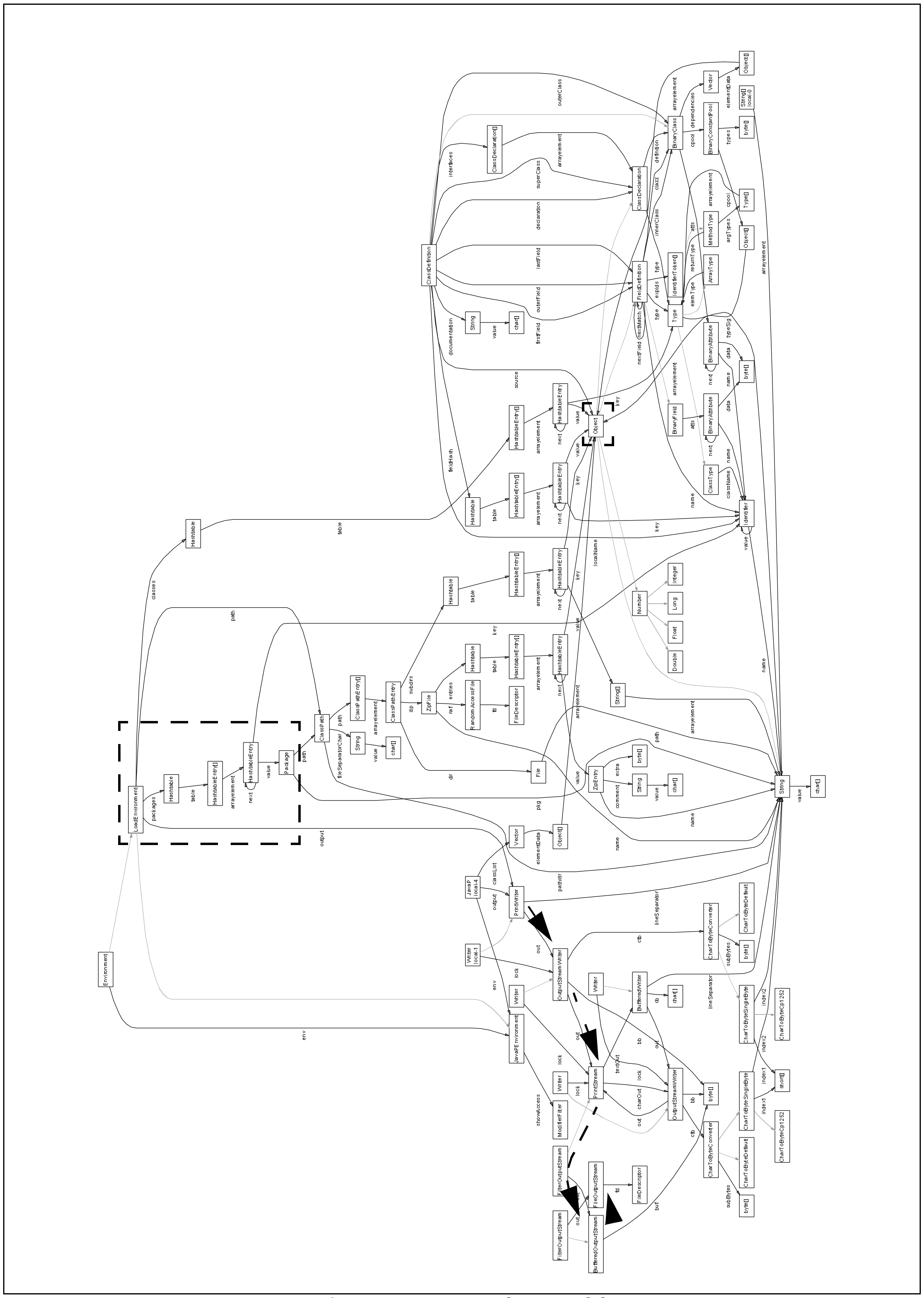


Figure 11-14. JavaP object model

The figure shows multiple occurrences of the `Hashtable` class. Each `Hashtable` has an array of `HashtableEntries`, and each `HashtableEntry` has a key and value. In Java, the keys and values are declared as `Objects`, but in most cases Ajax has been able to resolve them to specific classes, revealing the actual keys and values of each Hashtable. For example, we can see that `LocalEnvironment.packages` is a Hashtable mapping `Identifiers` to `Packages` (in the dashed outline).

On the left hand side of the model are a number of occurrences of stream-related classes. This part of the model reveals, for example, that the `JavaP` object's `output` field is a `PrintWriter` wrapping an `OutputStreamWriter` wrapping a `PrintStream` wrapping a `BufferedOutputStream` wrapping a `FileOutputStream` (as indicated by the fat dashed arrows). Each of these Writer or Stream objects contains an `out` field referencing the Writer or Stream it wraps. None of these relationships are apparent from the Java class declarations alone, because the `out` fields are simply declared as `Writer` or `OutputStream`.

On the right hand side of the model is an `Object` node with many edges leading into it, e.g., from the `key` and `value` fields of several Hashtables. Here the analysis was not powerful enough to distinguish the objects referenced by the incoming fields or to precisely determine their classes. The model reveals only that the referenced objects are either `Strings`, `Numbers`, `FieldDefinitions`, `ClassDeclarations`, or subclasses of one of those classes. This is a problem that becomes increasingly severe as the analyzed programs grow: imprecision in the analysis leads to a few nodes covering a very large number of different kinds of run-time objects. Field edges that lead to such nodes do not convey much useful information.

A fundamental problem revealed by this example is that this graph is about as large as one can usefully lay out and read. It has 96 nodes and 157 edges, and JavaP is a relatively small Java program. As graphs get larger, it becomes rapidly more difficult to visualize them in a reasonable way.

11.3.2 CTAS Example

Figure 11-15 shows the object model produced by Ajax applied to the CTAS example. The setup is the same as for the previous example. This graph has 122 nodes and 166 edges.

This model reveals some interesting facts, e.g., that the `postRecvHandlers`, `sendHandlers` and `mainRecvHandlers` of `HandlerManager` are all empty. (They are used by other applications based on this code, but not by the test program under analysis.) The model reveals that `ConnectionManager.socketQueue` is a `Vector` of `Sockets`, and is able to distinguish many different uses of CTAS's `HandlerTable` class.

On the negative side, again there is an `Object` node covering a large number of different kinds of objects, that seem to be unrelated but which are not being distinguished by the analysis.

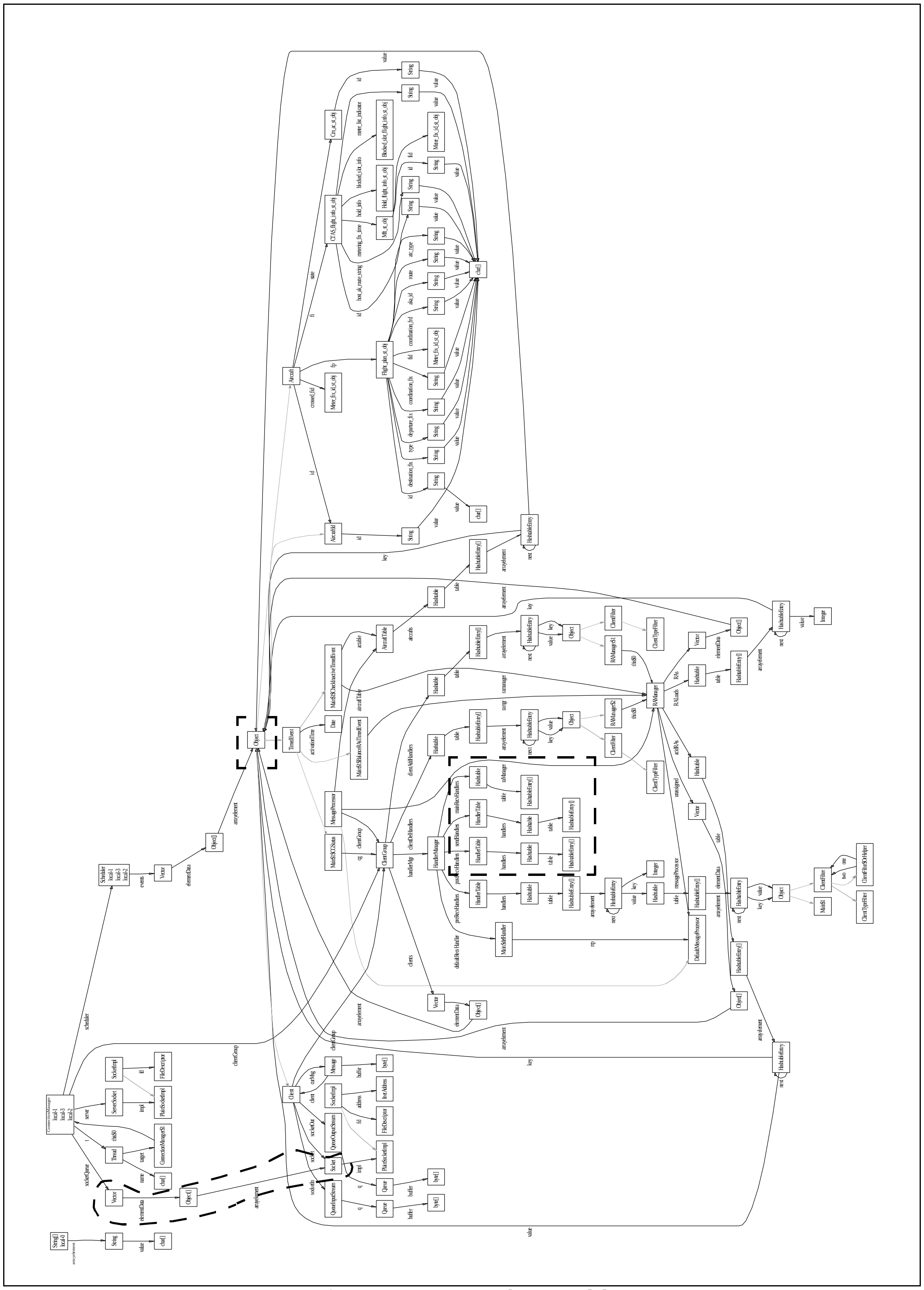


Figure 11-15. CTAS object model

11.3.3 Improving The Model By Discarding Information

11.3.3.1 Removing "Lumps"

Ajax object models for large programs are often crippled by the "large lump" problem, where the analysis creates one or more `Object` nodes covering a large number of different kinds of objects that are not truly related. These "lumps" cause the model's graph to be overconstrained, making it difficult to lay out and obscuring useful information.

One way to extract some useful information from these models is to detect and remove inaccurate "lumps" from the model graph. A useful heuristic is to remove nodes corresponding to abstract objects whose most specific known superclass is `Object` and which have many incoming edges. The field edges leading to such nodes are annotated to indicate that the referent of the field is not known. Nodes with many incoming edges especially impede comprehensible graph layout using hierarchy-based layout tools such as `dot`, so it is especially advantageous to remove them.

This approach sacrifices some information in the hope that some of the remaining information may still be useful to the user. A model that presents some information in a usable form is more useful than an incomprehensibly large model.

11.3.3.2 Hiding Strings And Other Classes

As described in Section 8.4.3, most references to `String` objects are aliased because they may refer to `String` objects extracted from the "constant pool". Thus, in an object model, most fields of type `String` lead to a common node. This clutters the graph layout with a large number of long edges. Furthermore, few programmers are interested in disambiguating `String` references even when this is possible. Therefore the Ajax object modeling tool can optionally remove the common `String` node and annotate relevant field edges to indicate that the referent is some unknown `String`.

The same technique can also be useful for other classes. The Ajax object modeling tool allows the user to explicitly specify an arbitrary set of classes to be elided; optionally, all subclasses of a specified class can be elided.

11.3.4 Jess Example

Figure 11-16 illustrates these techniques applied to an object model for the Java Expert System Shell example. To produce a model of manageable size, the details of the stream-related classes are elided by the tool using the techniques described in Section 11.3.3.2. The rules for elision are specified manually. In this case the rules are:

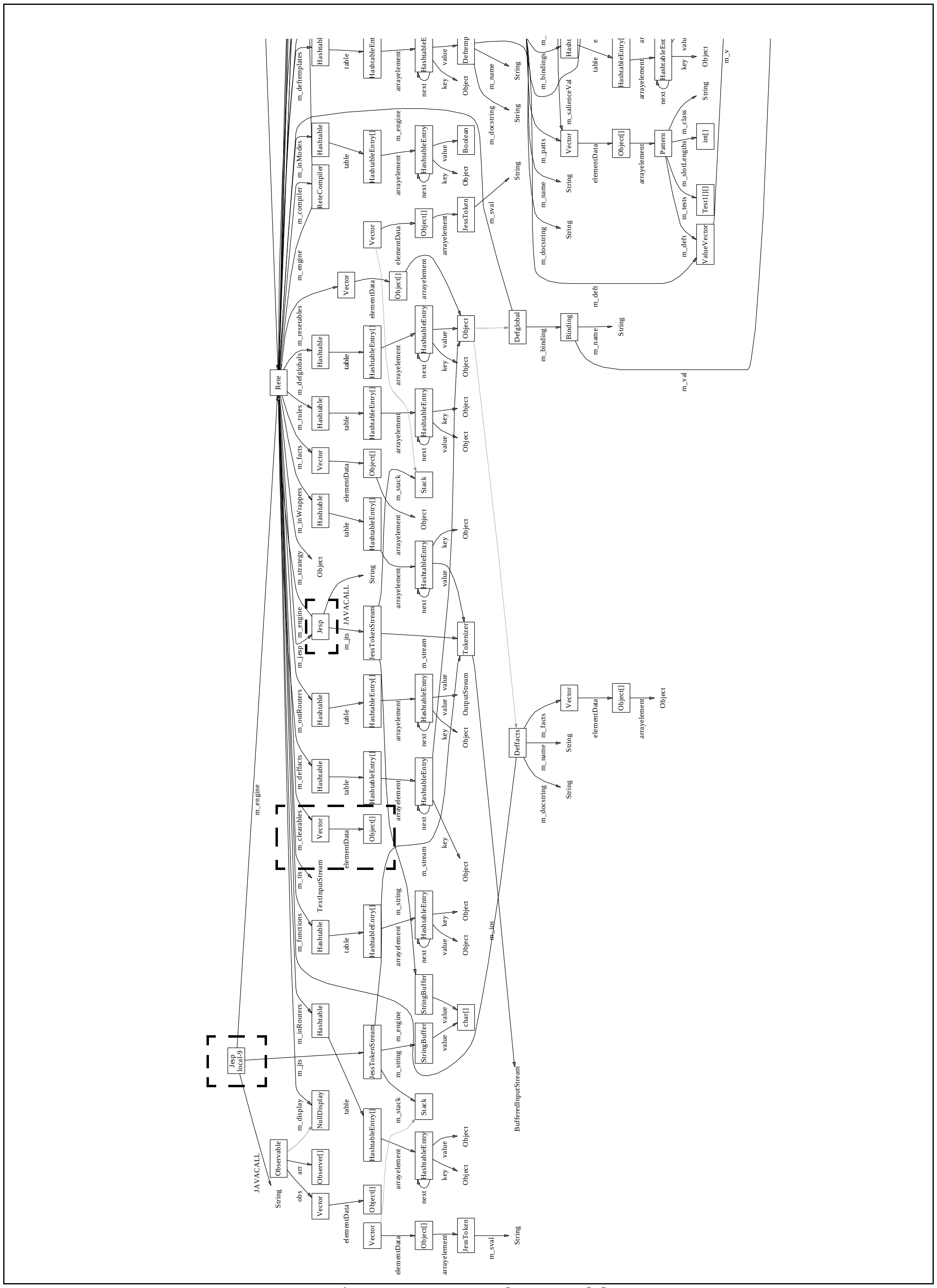


Figure 11-16. Jess object model

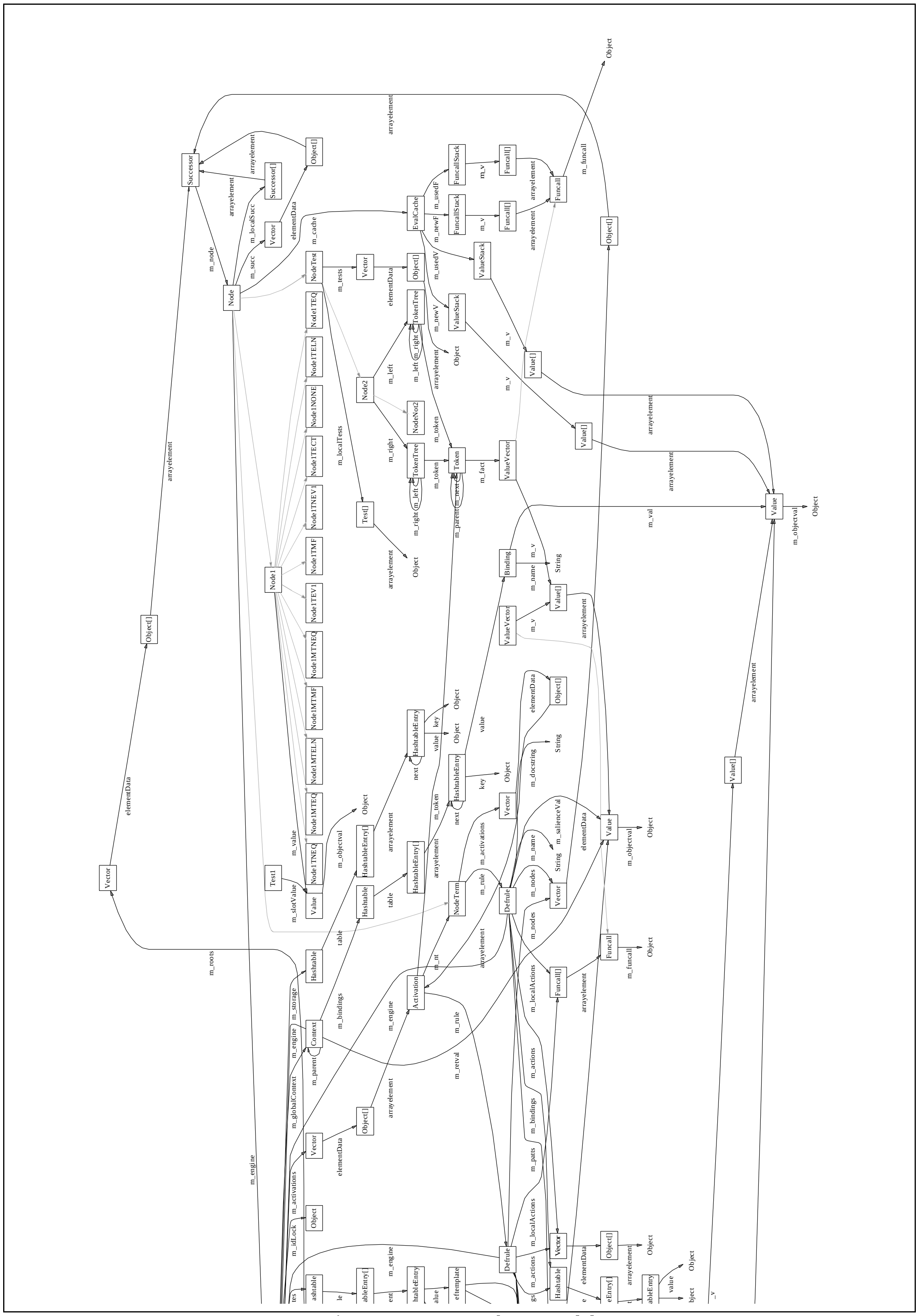


Figure 11-16. Jess object model

- Elide all lumps with more than seven incoming edges.
- Elide all `Strings`.
- Elide all subclasses of `InputStream`.
- Elide all subclasses of `OutputStream`.

As in the previous examples, this example reveals the contents of many of the container objects. It also reveals some information that may be surprising; for example, the `Rete`'s `m_clearables` `Vector` is always empty. Also, there are (at least) two distinct instances of the `Jesp` engine object.

This graph contains 189 nodes and 243 edges. The corresponding complete graph (without any node elision) contains 885 nodes and 1173 edges. The complete graph is much too complex to be automatically laid out in a comprehensible way. Therefore, although this reduced graph contains less information, in practice it is much more useful because its information is much more accessible.

This example shows one remaining problem with Ajax object models: it reveals unimportant implementation details of library classes. For example, the details of the implementation of `Hashtable` are revealed, when it would be better to simply show that `Hashtables` contain keys and values.

11.4 Conclusions

11.4.1 Contributions

Using the Ajax VPR, it is possible to construct heap graphs and object models. However, inaccuracies in the analysis and the sheer size of the graphs produced can cripple the usefulness of these graphs. Simple pruning countermeasures result in graphs that contain accessible, useful and surprising information, even for large programs. This information cannot be easily automatically obtained using other techniques, especially those that rely on declared class information.

The Ajax VPR is not the ideal abstraction to use for computing heap graphs. Extensive postprocessing is required. A tool with direct access to SEMI's constraint structures would be more efficient. Given the Ajax infrastructure, however, it seemed to be less work to compute the heap graphs from the VPR than to bypass the VPR and hook into the SEMI implementation.

11.4.2 Future Work

One major remaining problem with these models is that they have no notion of scope. In particular, they expose the implementation of library data structures. Instead it would be preferable to only show classes and fields visible to the user. On the other hand, sometimes information about private fields is useful to the user — for example, the `key` and `value` fields of `HashtableEntry` convey very useful information. Heuristics or other techniques to resolve this problem are an interesting area for future inquiry.

12 A Scanning Tool

12.1 Introduction

Programmers are adept at using simple tools such as “grep” to scan programs. More advanced cross-referencing and scanning tools such as class browsers, indexed full-text search engines, and hyperlinked source browsers such as LXR [91], are also very popular. However, none of these tools are semantics-based; they use syntactic or lexical information.

Using the Ajax analysis toolkit, it is not difficult to build similar tools that utilize semantic information about the program. To demonstrate this, I built a simple example called “JGrep”, and used it to reverse engineer some of the example programs.

12.2 The JGrep Tool

12.2.1 User Interface

JGrep has a simple “command line” interface, although it would be trivial to incorporate it into a graphical or Web-based interface such as LXR. The user specifies the program to analyze, and a program expression (including a code location). The expression need not actually occur in the program text. JGrep reports information about all the objects which might be returned as the result of the expression at the given location.

Four kinds of information are returned:

- *New* sites: all program locations where the objects are created.
- *Call* sites: all program locations where one of the objects is passed as the “this” parameter to a method call.
- *Read* sites: all program locations where a field of one of the objects is read.
- *Write* sites: all program locations where a field of one of the objects is written.

Since Ajax performs conservative analysis, some spurious sites may be returned along with the true sites.

The user can control which kinds of sites are returned, using command line options.

12.2.2 Implementation

JGrep is easy to implement using the Ajax toolkit. It comprises 462 lines of code. Collecting the sets of sites is a simple application of the value-point relation. The source set S is a singleton set containing the user-specified expression, and the target set T contains expressions for all the sites the user is interested in:

- New: The results of all “new” instructions, i.e., the top of the operand stack at the instruction after each `new`, `newarray`, `anewarray` and `multinewarray` instruction.
- Call: The stack element representing the “this” operand at every `invokevirtual`, `invokespecial` and `invokeinterface` instruction.
- Read: The top of the operand stack at each `getfield` instruction.
- Write: The top of the operand stack at each `putfield` instruction.

The “intermediate data” propagated by the analysis are boolean values, initially set to false and then set to true for the solitary source expression and all expressions reachable from it in the analyzer’s graph. For each target expression receiving the value “true”, the tool prints out the code location associated with that expression — i.e., the location of the “new” instruction, the “call” instruction, the `getfield` instruction or the `putfield` instruction.

JGrep currently accepts and prints code locations as the fully qualified name of a method and a bytecode offset within that method, e.g., “`jess.Main.main#373:local-9`” — local variable 9 in class `jess.Main`, method `main`, bytecode offset 373. It would be easy — and highly desirable — to input and output source line numbers and source-level expressions instead.

JGrep currently reanalyzes the program for every query, which means that there is a large delay between posing a query and receiving an answer. However, it would be easy to have JGrep run the analysis engine once and then answer a succession of queries.

12.3 Examples

12.3.1 Checking an Anomaly

The object model for Jess presented in Section 11.3.4 shows that the `Rete`’s `m_clearables` Vector is always empty. To investigate further, one simply submits to JGrep an expression corresponding to a path to the desired node in the object model:

```
jess.Main.main#373:local-9,jess.Jesp.m_engine,
jess.Rete.m_clearables
```

This expression specifes local variable 9 at offset 373 in the method `main` in class `jess.Main`, a reference to the Jesp application object, followed by two field dereferences: first, the dereference of field `m_engine` declared in class `jess.Jesp`, to get the Rete engine, and then the dereference of field `m_clearables` in class `jess.Rete`.

The “New” and “Call” sites output are shown in Figure 12-1.

The single “NEW” site reveals immediately that the `Vector` is created in `Rete`’s constructor (`jess.Rete.<init>`). The call to `java.util.Vector.elements` shows that the `Vector`’s elements are scanned in the method `Rete.clear()`. The call to `java.util.Vector.removeAllElements` indicates that it is emptied in `Rete.clear()`. There are no calls to methods that add elements to the `Vector`.

```
CALL to method void java.lang.Object.<init>()
    Offset 1 in method void java.util.Vector.<init>(int, int)
CALL to method void java.util.Vector.<init>(int, int)
    Offset 3 in method void java.util.Vector.<init>(int)
CALL to method void java.util.Vector.<init>(int)
    Offset 3 in method void java.util.Vector.<init>()
NEW of class java.util.Vector:
    Offset 182 in method void jess.Rete.<init>(jess.ReteDisplay)
CALL to method void java.util.Vector.<init>()
    Offset 186 in method void jess.Rete.<init>(jess.ReteDisplay)
CALL to method java.util.Enumeration java.util.Vector.elements()
    Offset 67 in method void jess.Rete.clear()
CALL to method void java.util.Vector.removeAllElements()
    Offset 249 in method void jess.Rete.clear()
```

Figure 12-1. Output of the creation sites and method calls on the `m_clearables` object

This information is helpful because it indicates to the programmer that if there were any elements in the `Vector`, they could only be used in the method `Rete.clear`. Therefore further investigation of this anomaly should focus on that method. If such investigation proves that an empty `m_clearables` is benign, then the entire field can be removed and we can be sure that no other code will be affected.

This example illustrates the power of the SEMI analysis; a simpler analysis such as RTA would not have been able to distinguish the different `Vectors` used in the program. Running "grep" over the Jess sources finds 43 occurrences of the name `Vector`, 5 occurrences of the name `removeAllElements`, 27 occurrences of the name `elements`, 34 occurrences of the name `elementAt`, and 22 occurrences of the name `addElement`. It would require significant effort to sort through these occurrences to find the three sites specifically operating on the `m_clearables` `Vector`.

12.3.2 Checking Field Accesses

In JavaC, there is a class `BatchEnvironment` with a public `flags` field. It is natural to wonder whether and how this field is accessed — is there an abstraction violation occurring, and in what form? JGrep provides the answer, using a query for the read and write accesses to the objects denoted by the expression:

```
sun.tools.javac.BatchEnvironment.<init>
(java.io.OutputStream, sun.tools.java.ClassPath,
sun.tools.javac.ErrorConsumer)#0
:local-0
```

This expression denotes the "this" objects of the most general constructor for `BatchEnvironment`. The results for the `flags` field are shown in Figure 12-2.

All the accesses are from one of three methods:

`sun.tools.javac.Main.compile` (read and written)

`sun.tools.javac.BatchEnvironment.getFlags` (read only)

`sun.tools.javac.BatchEnvironment.reportError` (read and written)

```
READ from field "flags" of class sun.tools.javac.BatchEnvironment:
    Offset 742 in method boolean
sun.tools.javac.Main.compile(java.lang.String[])
WRITE to field "flags" of class sun.tools.javac.BatchEnvironment:
    Offset 714 in method boolean
sun.tools.javac.Main.compile(java.lang.String[])
WRITE to field "flags" of class sun.tools.javac.BatchEnvironment:
    Offset 749 in method boolean
sun.tools.javac.Main.compile(java.lang.String[])
READ from field "flags" of class sun.tools.javac.BatchEnvironment:
    Offset 708 in method boolean
sun.tools.javac.Main.compile(java.lang.String[])
READ from field "flags" of class sun.tools.javac.BatchEnvironment:
    Offset 1 in method int sun.tools.javac.BatchEnvironment.getFlags()
READ from field "flags" of class sun.tools.javac.BatchEnvironment:
    Offset 216 in method void
sun.tools.javac.BatchEnvironment.reportError(java.lang.Object, int,
java.lang.String, java.lang.String)
WRITE to field "flags" of class sun.tools.javac.BatchEnvironment:
    Offset 222 in method void
sun.tools.javac.BatchEnvironment.reportError(java.lang.Object, int,
java.lang.String, java.lang.String)
WRITE to field "flags" of class sun.tools.javac.BatchEnvironment:
    Offset 92 in method void
sun.tools.javac.BatchEnvironment.reportError(java.lang.Object, int,
java.lang.String, java.lang.String)
READ from field "flags" of class sun.tools.javac.BatchEnvironment:
    Offset 86 in method void
sun.tools.javac.BatchEnvironment.reportError(java.lang.Object, int,
java.lang.String, java.lang.String)
```

Figure 12-2. Accesses to the `flags` field of `BatchEnvironment`

Note that this example does not particularly benefit from SEMI. The same results are obtained using Ajax's RTA engine, because there is really only one instance of `BatchEnvironment` used in the program.

12.4 Conclusions

Using the alias information obtained by Ajax, it is easy to write simple and useful search tools. These tools improve on the functionality available from lexical and syntactic tools in a natural way. Additional postprocessing could improve the utility of the results, but even the simplest approaches are useful. There is significant scope for new searching and visualization tools based on these techniques.

13 Conclusions

13.1 Summary

Ajax demonstrates that sound, static, global alias analysis can be used as the basis for a variety of software engineering tools. These tools produce interesting and nontrivial results that cannot be obtained by other existing methods.

The Ajax design shows that it is practical to separate analysis implementations from tools that consume alias information. The specification for an analysis engine is semantically simple, as defined by the value-point relation, but powerful enough to enable cheap construction of a wide range of tools. The interface is also efficient; for most configurations, the scalability of the system is constrained by the scalability of the underlying analysis and not by the overhead of the VPR interface. The exception is the object modelling tool. It takes a significant amount of code and execution resources to reconstruct a “heap graph” from the VPR, and also requires a strengthened definition of the VPR.

Ajax also shows that it is possible to implement the VPR interface using very different analyses — RTA, based on declared language types, SEMI, based on polymorphic type inference, and a hybrid analysis based on the “intersection” of these two analysis engines. The strong separation between analyses and tools ensures that all tools work correctly regardless of the analysis configuration. The analysis technique can be selected at run time according to the desired accuracy for the task at hand and the execution resources available. For example, for finding the set of possibly live methods, RTA is usually good enough, but SEMI is much better for resolving virtual method calls, albeit more expensive.

The VPR interface also enables easy composition of analyses. It is trivial to build an analysis that computes the intersection of the results of two or more other analyses. Ajax can also provide “sequential composition”; for example, SEMI can use some other arbitrary analysis to compute the call graph it uses to reduce programs to first order.

SEMI shows that type inference with polymorphic recursion can usefully be applied to large Java programs, especially if the program is conservatively reduced to first-order code before the application of SEMI. I have proven SEMI sound with respect to a simplified — but still very rich — model of the Java bytecode, and shown that SEMI can even analyze programs which do not conform to the static safety checks usually performed by Java. SEMI provides a significant improvement in accuracy over a wide range of tools and example programs, and well captures implicit type parametricity in Java programs, proving a large percentage of downcasts safe in most programs. However, SEMI is less accurate in larger programs, because imprecision in analyzing one part of the code spills over into other parts of the code. Although SEMI can indeed analyze some large programs (Ladybug having over 5,000 methods), its scalability in terms of resource consumption and accuracy still leaves much to be desired.

Polymorphic recursion plays an interesting role in SEMI. I have described several techniques required to make the SEMI implementation of polymorphic recursion practical. The benefits of polymorphic recursion vary by tool: in the virtual call resolution tool, polymorphic recursion improves accuracy only a little, but for checking downcasts, unrestricted polymorphic recursion improves accuracy a great deal — but only when the program is initially reduced to first order. The generality of the SEMI constraint solving engine seems to limit its performance compared to other systems based on Hindley-Milner type inference [54] [69].

My work shows that composing RTA and SEMI by intersection is very useful. RTA is so cheap that performance is not noticeably affected, and for many tools the combined analyses are significantly more accurate than either analysis alone.

Most of the Ajax tools were easy and cheap to build. Of all the tools, I personally feel that the most immediately useful is "JGrep", having used it myself to reverse-engineer some of the example programs for which source code is not available. It is very useful to be able to track down all accesses to one instance of a commonly reused class. The object modelling tool demonstrates that starting with alias information and transforming it into an object model can produce more precise models than existing techniques, which start with a class hierarchy model and improve its precision using heuristics or other analysis [46].

Accounting for the behavior of non-Java code — i.e., native code and reflection — required a great deal of work. This is an important problem because real programs (especially the standard Java libraries) use these features often, and in a variety of ways. Ajax provides thorough handling of non-Java code by accepting specifications describing how non-Java code is used by the application. However, unavailability of the whole program remains a fundamental problem.

13.2 Outlook

There are many possible future directions for this work:

- SEMI is too slow at analyzing very large programs. It may be possible to reimplement a similar analysis to achieve much higher performance, perhaps using a design similar to Ruf's escape analysis for Java [69]. Alternatively, it may be possible to design a simpler analysis with some of the desirable features of SEMI.
- SEMI's accuracy degrades as program size increases. Addressing this may required improved analysis techniques. Some limited flow-sensitive analysis might improve accuracy, as might tighter integration of language type information into SEMI's computations. One improvement that would be almost certain to provide increased accuracy would be the introduction and use of "parity annotations" on instance constraints, as described by Fähndrich, Rehof and Das [31].
- It would be very interesting to implement more analyses in the Ajax framework. Ajax provides a great deal of infrastructure to make it easier to implement analyses. Ajax also provides a tool suite; once an analysis has been implemented, it can be immediately applied to a wide range of problems. Analysis composition is also very easy in Ajax, and can compensate for weaknesses in one particular analysis technique. Also,

because Ajax provides a single description of the behavior of non-Java code and a fixed specification of sound analysis results, it is both easy and fair to compare the accuracy and performance of different analyses implemented in Ajax.

- The VPR is not the ultimate abstraction of program behavior. It has very limited expression of context: for example, it is impossible to ask whether two expressions in a method get the same value during the same invocation of the method. It is also impossible to specify that an expression should apply not just at a particular program point, but also when its method has a particular caller. SEMI can capture some of this information. The VPR could be extended to allow this information to be communicated to tools.
- The VPR could also be extended to accomodate different behaviors of tags in the tagged bytecode semantics. For example, one might wish to have addition take two operands with the same tag and return a result with the same tag as the operands. Thus an expression referring to the result of an addition would match an expression referring to one of the operands. This would allow Ajax to address additional tasks.
- More tools could easily be built in the Ajax framework. Accessible alias analysis opens up many possibilities for new tools for various programmer tasks.
- Sound, global, static analysis of Java programs is inherently difficult because Java programs use Java features that are not amenable to static analysis, such as reflection. Furthermore, modern software environments consist of dynamically configured components, often interacting over channels not amenable to static analysis, e.g., by exchanging XML data. Thus many applications are not amenable to sound global static analysis.
    - It may be necessary to perform local static analysis. In particular, it would be interesting to make “worst case” assumptions about missing code and then measure the accuracy of the resulting analyses. It would also be interesting to introduce “reasonable” heuristics to approximate the behavior of missing code and then measure analysis accuracy.
    - It is easy to change the definition of the VPR to quantify over some fixed finite set of program traces (e.g., some program traces that have actually been obtained by running the program) instead of all traces. An Ajax analysis could compute a precise VPR for a program by running it on test data and recording the execution. The existing Ajax tools would be immediately usable with this dynamic analysis.

I predict that in the forseeable future, tasks such as program understanding, which do not absolutely require sound static analysis of code, will best be addressed by other means, such as dynamic analysis or unsound static analysis. Tasks which do require sound static analysis, such as compilers or verification tools, will need to perform local analysis of individual components, relying on whatever explicit (run time checkable) annotations exist at component boundaries to specify the behavior of “external” code.

Bibliography

[1] O. Agesen. The Cartesian Product Algorithm: Simple And Precise Type Inference Of Parametric Polymorphism. Proceedings of the 9th European Conference on Object-Oriented Programming, Åarhus, Denmark, August 1995, Springer-Verlag LNCS 952, pp. 2-26.

[2] A. Aiken, M. Fähndrich, J. Foster and Z. Su. A Toolkit For Constructing Type- And Constraint-Based Program Analyses. Proceedings of the Second International Workshop on Types in Compilation, Kyoto, Japan, March 1998, Springer-Verlag LNCS 1473, pp. 78-96.

[3] A. Aiken and E. Wimmers. Type Inclusion Constraints And Type Inference. Proceedings of the International Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, June 1993, pp. 31-41.

[4] J. Aldrich, C. Chambers, E. Gun Sirer, and S. Eggers. Static Analyses For Eliminating Unnecessary Synchronization From Java Programs. Proceedings of the 6th International Static Analysis Symposium, September 1999, Springer-Verlag LNCS 1694, pp. 19-38.

[5] L. Andersen. Program Analysis and Specialization For The C Programming Language. Technical Report 94-19, University of Copenhagen, Copenhagen, Denmark, 1994.

[6] J. Ashley and R. Dybvig. A Practical And Flexible Flow Analysis For Higher-Order Languages. ACM Transactions on Programming Languages and Systems, Volume 20, No. 4, July 1998, pp. 845-868.

[7] R. Bowdidge and W. Griswold. Automated Support For Encapsulating Abstract Data Types. Proceedings of the ACM Conference On Foundations of Software Engineering, New Orleans, USA, December 1994, pp. 97-110.

[8] A. Bondorf and J. Jørgensen. Efficient Analyses For Realistic Off-line Partial Evaluation. Journal of Functional Programming, Volume 3, No. 3, July 1993, pp. 315-346.

[9] D. Bacon and P. Sweeney. Fast Static Analysis Of C++ Virtual Function Calls. Proceedings of the ACM SIGPLAN '96 Conference on Object-Oriented Programming Systems, Languages and Applications, San Jose, USA, October 1996, pp. 324-341.

[10] B. Blanchet. Escape Analysis For Object-Oriented Languages: Application To Java. Proceedings of the ACM SIGPLAN '99 Conference on Object-Oriented Programming Systems, Languages and Applications, Denver, USA, November 1999, pp. 20-34.

[11] J. Bogda and U. Hölzle. Removing Unnecessary Synchronization In Java. Proceedings of the ACM SIGPLAN '99 Conference on Object-Oriented Programming Systems, Languages and Applications, Denver, USA, November 1999, pp. 35-46.

[12] J. Boyland and A. Greenhouse. May Equal: A New Alias Question. Presented at the Intercontinental Workshop on Aliasing in Object Oriented Systems, Lisbon, Portugal, June 1999.

[13] G. Bracha, M. Odersky, D. Stoutamire and P. Wadler. Making The Future Safe For The Past: Adding Genericity To The Java Programming Language. Proceedings of the ACM SIGPLAN '98 Conference on Object-Oriented Programming Systems, Languages and Applications, Vancouver, Canada, October 1998, pp. 183-200.

[14] R. Chatterjee, B. Ryder and W. Landi. Relevant Context Inference. Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, USA, January 1999, pp. 133-146.

[15] Y.-F. Chen, M. Nishimoto, and C. Ramamoorthy. The C Information Abstraction System. IEEE Transactions on Software Engineering, Volume 16, No. 3, March 1990, pp. 325-334.

[16] B. Cheng and W. Hwu. Modular Interprocedural Pointer Analysis Using Access Paths: Design, Implementation, And Evaluation. Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation, Vancouver, Canada, June 2000, p. 57-69.

[17] J. Choi, M. Gupta, M. Serrano, V. Sreedhar and S. Midkiff. Escape Analysis For Java. Proceedings of the ACM SIGPLAN '99 Conference on Object-Oriented Programming Systems, Languages and Applications, Denver, USA, November 1999, pp. 1-19.

[18] M. Cierniak, G. Lueh and J. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. Proceedings of the ACM SIGPLAN ’00 Conference on Programming Language Design and Implementation, Vancouver, Canada, June 2000, pp. 13-26.

[19] M. Das. Unification-Based Pointer Analysis With Directional Assignments. Proceedings of the ACM SIGPLAN ’00 Conference on Programming Language Design and Implementation, Vancouver, Canada, June 2000, pp. 35-46.

[20] J. Dean, D. Grove, and C. Chambers. Optimization Of Object-Oriented Programs Using Static Class Hierarchy Analysis. Proceedings of the 9th European Conference on Object-Oriented Programming, Åarhus, Denmark, August 1995, Springer-Verlag LNCS 952, pp. 77-101.

[21] G. DeFouw, D. Grove and C. Chambers. Fast Interprocedural Class Analysis. Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, USA, January 1998, pp. 222-236.

[22] A. Diwan, J. Moss, and K. McKinley. Simple And Effective Analysis Of Statically-Typed Object-Oriented Programs. Proceedings of the ACM SIGPLAN '96 Conference on Object-Oriented Programming Systems, Languages and Applications, San Jose, USA, October 1996, pp. 292-305.

[23] A. Diwan, J. Moss, and K. McKinley. Type-Based Alias Analysis. Proceedings of the ACM SIGPLAN ’98 Conference on Programming Language Design and Implementation, Montreal, Canada, June 1998, pp. 106-117.

[24] J. Dolby and A. Chien. An Automatic Object Inlining Optimization And Its Evaluation. Proceedings of the ACM SIGPLAN ’00 Conference on Programming Language Design and Implementation, Vancouver, Canada, June 2000, pp. 345-357.

[25] D. Duggan. Modular Type-Based Reverse Engineering Of Parameterized Types In Java Code. Proceedings of the ACM SIGPLAN '99 Conference on Object-Oriented Programming Systems, Languages and Applications, Denver, USA, November 1999, pp. 97-113.

[26] P. Eidorff, F. Henglein, C. Mossin, H. Niss, M. Sørensen and M. Tofte. AnnoDomini: From Type Theory To Year 2000 Conversion Tool. Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, USA, January 1999, pp. 1-14.

[27] J. Eifrig, S. Smith, and V. Trifonov. Sound Polymorphic Type Inference For Objects. Proceedings of the ACM SIGPLAN '95 Conference on Object-Oriented Programming Systems, Languages and Applications, Austin, USA, October 1995, pp. 169-184.

[28] M. Fähndrich. BANE: A Library for Scalable Constraint-Based Program Analysis. PhD Thesis, Computer Science Division, University of California, Berkeley, USA, March 1999.

[29] M. Fähndrich and A. Aiken. Program Analysis Using Mixed Term And Set Constraints. Proceedings of the 4th International Static Analysis Symposium, September 1997, Springer-Verlag LNCS 1302, pp. 114-126.

[30] M. Fähndrich, J. Foster, Z. Su and A. Aiken. Partial Online Cycle Elimination In Inclusion Constraint Graphs. Proceedings of the ACM SIGPLAN ’98 Conference on Programming Language Design and Implementation, Montreal, Canada, June 1998, pp. 85-96.

[31] M. Fähndrich, J. Rehof and M. Das. Scalable Context-Sensitive Flow Analysis Using Instantiation Constraints. Proceedings of the ACM SIGPLAN ’00 Conference on Programming Language Design and Implementation, Vancouver, Canada, June 2000, pp. 253-263.

[32] M. Fernandez, Simple And Effective Link-Time Optimization Of Modula-3 Programs. Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, La Jolla, USA, June 1995, pp. 103-115.

[33] C. Flanagan and M. Felleisen. Componential Set-Based Analysis. ACM Transactions on Programming Languages and Systems, Volume 21, No. 2, March 1999, pp. 370-416.

[34] J. Foster, M. Fähndrich and A. Aiken. Polymorphic Versus Monomorphic Flow-Insensitive Points-To Analysis For C. Proceedings of the 7th International Static Analysis Symposium, September 2000, Springer-Verlag LNCS 1824, pp. 175-198.

[35] E. Friedman-Hill. Jess, The Java Expert System Shell. Technical Report SAND98-8206 (revised), Distributed Computing Systems, Sandia National Laboratories, Livermore, California, January 2000.

[36] E. Gansner and S. North. An Open Graph Visualization System And Its Applications To Software Engineering. Software Practice and Experience, Volume 30, No. 11, September 2000, pp. 1203-1233.

[37] D. Grove, G. DeFouw, J. Dean and C. Chambers. Call Graph Construction In Object-Oriented Languages. Proceedings of the ACM SIGPLAN '97 Conference on Object-Oriented Programming Systems, Languages and Applications, Atlanta, USA, October 1997, pp. 108-124.

[38] D. Gifford, P. Jouvelot, J. Lucassen, and M. Sheldon. FX-87 Reference Manual. Technical Report MIT/LCS/TR-407, MIT Laboratory for Computer Science, Boston, USA, September 1987.

[39] N. Heintze. Set-Based Analysis Of ML Programs. Proceedings of the ACM Conference on Lisp and Functional Programming, Orlando, USA, June 1994, pp. 306-317.

[40] N. Heintze. Control-Flow Analysis And Type Systems. Proceedings of the 2nd Static Analysis Symposium, September 1995, Springer-Verlag LNCS 983, pp. 189-206.

[41] N. Heintze and D. McAllester. Linear-Time Subtransitive Control Flow Analysis. Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, Las Vegas, USA, June 1997, pp. 261-272.

[42] F. Henglein. Type Inference With Polymorphic Recursion. ACM Transactions on Programming Languages and Systems, Volume 15, No. 2, April 1993, pp. 253-289.

[43] D. Jackson and J. Chapin. Redesigning Air-Traffic Control: A Case Study In Software Design. IEEE Software, Volume 17, No. 3, May/June 2000, pp. 63-70.

[44] D. Jackson, S. Jha and C. Damon. Isomorph-Free Model Enumeration. ACM Transactions on Programming Languages and Systems, Volume 20, No. 2, March 1998, pp. 302-343.

[45] D. Jackson and E. Rollins. Abstractions Of Program Dependencies For Reverse Engineering. Proceedings of the ACM Conference On Foundations of Software Engineering, New Orleans, USA, December 1994, pp. 2-10.

[46] D. Jackson and A. Waingold. Lightweight Extraction Of Object Models From Bytecode. Proceedings of the 1999 International Conference on Software Engineering, Los Angeles, USA, May 1999, pp. 194-202.

[47] S. Jagannathan and S. Weeks. A Unified Treatment Of Flow Analysis In Higher-Order Languages. Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, USA, January 1995, pp. 393-407.

[48] T. Lindholm and F. Yellin. The Java Virtual Machine Specification, Second Edition. Addison Wesley, 1997.

[49] R. Milner. A Theory Of Type Polymorphism In Programming. Journal of Computer and System Sciences, Volume 17, 1978, pp. 348-375.

[50] R. Milner, M. Tofte and R. Harper. The Definition Of Standard ML. MIT Press, 1990.

[51] G. Murphy and D. Notkin. Lightweight Source Model Extraction. Proceedings of the ACM Conference On Foundations of Software Engineering, Washington DC, USA, October 1995, pp. 116-127.

[52] G. Murphy and D. Notkin. Software Reflexion Models: Bridging The Gap Between Source And High-Level Models. Proceedings of the ACM Conference On Foundations of Software Engineering, Washington DC, USA, October 1995, pp. 18-28.

[53] R. O’Callahan. A Simple, Comprehensive Type System For Java Bytecode Subroutines. Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, USA, January 1999, pp. 70-78.

[54] R. O'Callahan and D. Jackson. Lackwit: A Program Understanding Tool Based On Type Inference. Proceedings of the 1997 International Conference on Software Engineering, Boston, USA, 1997, p. 338-348.

[55] R. O'Callahan and D. Jackson. Lackwit: Large-Scale Analysis Of C Programs Using Type Inference. Technical Report CMU-CS-96-130, Carnegie Mellon University Computer Science Department, 1996.

[56] N. Oxhøj, J. Palsberg and M. Schwartzbach. Making Type Inference Practical. Proceedings of the 6th European Conference on Object-Oriented Programming, Utrecht, The Netherlands, June 1992, Springer-Verlag LNCS 615, pp. 329-349.

[57] J. Palsberg. Efficient Inference Of Object Types. Information and Computation, Volume 123, No. 2, 1995, pp. 198-209.

[58] J. Palsberg and P. O'Keefe. A Type System Equivalent To Flow Analysis. ACM Transactions on Programming Languages and Systems, Volume 17, No. 4, July 1995, pp. 576-599.

[59] J. Palsberg and C. Pavlopoulou. From Polyvariant Flow Information To Intersection And Union Types. Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, USA, January 1998, pp. 197-208.

[60] J. Palsberg and M. Schwartzbach. Object-Oriented Type Inference. Proceedings of the ACM SIGPLAN '91 Conference on Object-Oriented Programming Systems, Languages and Applications, Phoenix, USA, October 1991, pp. 146-161.

[61] X. Leroy and F. Pessaux. Type-Based Analysis Of Uncaught Exceptions. ACM Transactions on Programming Languages and Systems, Volume 22, No. 2, March 2000, pp. 340-377.

[62] D. Liang and M. Harrold. Efficient Points-to Analysis For Whole-Program Analysis. Proceedings of the ACM Conference On Foundations of Software Engineering, Toulouse, France, September 1999, Springer-Verlag LNCS 1687, pp. 199-215.

[63] J. Plevyak. Optimization Of Object-Oriented And Concurrent Programs. PhD Thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1996.

[64] Z. Qian. A Formal Specification Of Java Virtual Machine Instructions. Technical Report, Universitat Bremen, Bremen, Germany, November 1997.

[65] D. Rémy and J. Vouillon. Objective ML: A Simple Object-Oriented Extension Of ML. Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, January 1997, pp. 40-53.

[66] A. Rountev, A. Milanova, and B. Ryder. Points-to Analysis For Java Using Annotated Inclusion Constraints. Technical Report DCS-TR-417, Department of Computer Science, Rutgers University, Piscataway, USA, July 2000.

[67] E. Ruf. Context-Insensitive Alias Analysis Reconsidered. Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, La Jolla, USA, June 1995, pp. 13-22.

[68] E. Ruf. Partitioning Data Flow Analysis Using Types. Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, January 1997, pp. 15-26.

[69] E. Ruf. Effective Synchronization Removal For Java. Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation, Vancouver, Canada, June 2000, pp. 208-218.

[70] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen. Object Oriented Modeling And Design, Prentice Hall, 1991.

[71] O. Shivers. Control Flow Analysis In Scheme. Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta,, USA, June 1988, pp. 164-174.

[72] B. Steensgaard. Points-To Analysis In Almost Linear Time. Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, USA, January 1996, pp. 32-41.

[73] B. Steensgaard. Points-To Analysis By Type Inference Of Programs With Structures And Unions. Proceedings of the 1996 International Conference on Compiler Construction, Springer-Verlag LNCS 1060, April 1996, pp. 136-150.

[74] P. Stocks, B. Ryder, and W. Landi. Comparing Flow- And Context-Sensitivity On The Modification-Side-Effects Problem. Technical Report DCS-TR-335, Department of Computer Science, Rutgers University, August 1997.

[75] Z. Su, M. Fähndrich and A. Aiken. Projection Merging: Reducing Redundancies In Inclusion Constraint Graphs. Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, USA, January 2000, pp. 81-95.

[76] V. Sundaresan, L. Hendren, C. Razafimahefa, R Vallee-Rai, P. Lam, E. Gagnon, C. Godin. Practical Virtual Method Call Resolution For Java. Proceedings of the ACM SIGPLAN '00 Conference on Object-Oriented Programming Systems, Languages and Applications, Minneapolis, USA, October 2000, pp. 264-280.

[77] J.-P. Talpin and P. Jouvelot. The Type And Effect Discipline. Proceedings of the 7th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, Santa Cruz, USA, 1992, pp. 162-173.

[78] F. Tip. A Survey Of Program Slicing Techniques. Journal of Programming Languages, Vol. 3, No. 3, September 1995, pp. 121-189.

[79] F. Tip, C. Laffra, P. Sweeney and D. Streeter. Practical Experience With An Application Extractor For Java. Proceedings of the ACM SIGPLAN '99 Conference on Object-Oriented Programming Systems, Languages and Applications, Denver, USA, November 1999, p. 292-305.

[80] F. Tip and J. Palsberg. Scalable Propagation-Based Call Graph Construction Algorithms. Proceedings of the ACM SIGPLAN '00 Conference on Object-Oriented Programming Systems, Languages and Applications, Minneapolis, USA, October 2000, pp. 281-293.

[81] M. Tofte and J.-P. Taplin. Implementation Of The Typed Call-By-Value λ-Calculus Using A Stack of Regions. Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, USA, January 1994, pp. 188-201.

[82] M. Weiser. Program Slicing. IEEE Transactions on Software Engineering, Volume 10, No. 7, July 1984, pp. 352-357.

[83] J. Whaley and M. Rinard. Compositional Pointer And Escape Analysis For Java Programs. Proceedings of the ACM SIGPLAN '99 Conference on Object-Oriented Programming Systems, Languages and Applications, Denver, USA, November 1999, pp. 187-206.

[84] R. Wilson and M. Lam. Efficient Context-Sensitive Pointer Analysis For C Programs. Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, La Jolla, USA, June 1995, pp. 1-12.

[85] A. Wright and R. Cartwright. A Practical Soft Type System For Scheme. Proceedings of the 1994 ACM Conference on Lisp and Functional Programming, Orlando, Florida, June 1994, pp. 250-262.

[86] S. Zhang, B. Ryder, and W. Landi. Program Decomposition For Pointer Aliasing: A Step Towards Practical Analyses. Proceedings of the 4th Annual ACM Symposium on the Foundations of Software Engineering, San Francisco, USA, October 1996, pp. 81-92.

[87] S. Zhang, B. Ryder and W. Landi. Experiments With Combined Analysis For Pointer Aliasing. Proceedings of the ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering, Montreal, Canada, June 1998, pp. 11-18.

[88] Bugzilla Project Home Page.
`http://www.mozilla.org/projects/bugzilla`

[89] CodeSurfer Home Page.
`http://www.codesurfer.com`

[90] Imagix Corporation Home Page
`http://www.imagix.com`

[91] Linux Cross Reference
`http://lxr.linux.no`

Appendix A: Polymorphic Recursion, Unrestricted Recursive Types and Principal Types

Consider a standard lambda language with a type system having polymorphic recursion and unrestricted (μ) recursive types. I prove that there exist typable program terms that have no principal type.

A.1 Intuition

In the setting of μ-recursive types, a type T for a term `f` is principal iff T is a type of `f` and every type of `f` is equivalent to an instance of T, where type equivalence means that the (possibly infinite) regular labelled trees corresponding to the types are identical.

Consider the following function, written in ML-like syntax:

```
fun f (a, b) = f b
```

This function is typable using polymorphic recursion and unrestricted recursive types, but there is no principal type. A list of valid types is below. All free variables are assumed to be universally quantified.

$$(\mu t.\ v \times t) \rightarrow u$$
$$w \times (\mu t.\ v \times t) \rightarrow u$$
$$x \times (w \times (\mu t.\ v \times t)) \rightarrow u$$

Informally we could write these types as "$(v, (v, (v, \ldots))) \rightarrow u$", "$(w, (v, (v, (v, \ldots)))) \rightarrow u$", and "$(x, (w, (v, (v, (v, \ldots))))) \rightarrow u$". This leads to the intuition that the principal type would need to have an unbounded number of quantified variables — but such types do not exist.

A.2 Proof

More formally, suppose T is the principal type of the function `f` given above. We show that this leads to a contradiction.

Let m be the number of free variables in T. Define

$$J_0 = \mu t.\ v \times t$$
$$J_n = w_n \times J_{n\text{-}1} \qquad (n > 0)$$

For all n, $J_n \rightarrow u$ is a type of `f`. This is easily shown by induction on n.

Therefore there is a substitution S such that S(T) is equivalent to $J_m \rightarrow u$. $J_m \rightarrow u$ has more free variables than T; therefore, there is a free variable of T (referred to as e) such that S maps e to a term equivalent to a subterm of $J_m \rightarrow u$ containing at least two free variables. I will refer to the latter subterm as the "expansion term". These are the subterms of $J_m \rightarrow u$, modulo equivalence:

1. $J_m \rightarrow u$
2. u
3. J_i $(1 \leq \mathrm{i} \leq m)$

4. w_i $(1 \le \mathrm{i} \le m)$

5. v

6. $\mu t.\ v \times t$

Cases 2, 4, 5 and 6 do not contain at least two free variables, hence cannot be the expansion term. Case 1 cannot be the expansion term, for then $\mathrm{T} = e$, a single free variable, which is not a type of `f`. Therefore the expansion term is J_i (for some i, $1 \le \mathrm{i} \le m$).

Let S' be the same substitution as S except that e is mapped to "int". S'(T) is equivalent to the tree for $\mathrm{J}_m \to u$ with one or more subtrees equivalent to J_i replaced by "int". But since w_i occurs just once in the tree for "$\mathrm{J}_m \to u$", there is only one such subtree — the actual occurrrence of J_i introduced by the production rules. Therefore $\mathrm{S'(T)} = \mathrm{K}_m \to u$ where

$$\mathrm{K}_i = \mathrm{int}$$
$$\mathrm{K}_n = w_n \times \mathrm{K}_{n-1} \quad (n > 0)$$

It is easy to see that this is not a type of `f`, violating the assumption that T is a principal type.

A.3 Comments

The principal type T of a term in Henglein's type system is also a valid type of the term when the type system has recursive types. The reason that principal typing fails is because the addition of recursive types may allow new types for the term which are not instances of T.

Appendix B: Ajax Foreign Code Specifications

I provide the complete text of the foreign code specifications used by Ajax. They cover a large part of the JDK 1.1 class library for Windows, but not all of the library. I provide the specifications to indicate how extensive they are and how much modelling is required. Also, the curious reader can see how I modelled the behavior of specific functions.

```
/* Special definitions used by the SEMI analyzer.
   These definitions are used by the SEMI analyzer and by
other native code specifications.
   These may not have constraints generated for them
using the normal path (guided by the liveness query);
SEMI may just decide to generate its own constraints for
them as needed. We do this so that the details of how
they are used are kept internal to SEMI.
*/

makeCharArray() {
    VALUE = new [C;
    java.lang.Object.<init>(VALUE);
    LEN = choose;
    VALUE java.lang.Object#arraylength := LEN;

L:  CH = choose;
    VALUE java.lang.Object#intarrayelement := CH;
    goto L, N;

N:  return = choose VALUE;
}

accessStringChars(STR) {
    STR java.lang.String.value;
    STR java.lang.String.offset;
    STR java.lang.String.count;
}

makeIntArray() {
    VALUE = new [I;
    java.lang.Object.<init>(VALUE);
    LEN = choose;
    VALUE java.lang.Object#arraylength := LEN;

L:  I = choose;
    VALUE java.lang.Object#intarrayelement := I;
    goto L, N;

N:  return = choose VALUE;
}

makeByteArray() {
    VALUE = new [B;
    java.lang.Object.<init>(VALUE);
    LEN = choose;
    VALUE java.lang.Object#arraylength := LEN;

L:  B = choose;
    VALUE java.lang.Object#intarrayelement := B;
    goto L, N;

N:  return = choose VALUE;
}

makeString() {
    VALUE = makeCharArray();
    STR = new java.lang.String;
    java.lang.String.<init>(STR, VALUE) "([C)V";
    return = choose STR;
}

mungeStrings(STR1, STR2) {
    VALUE = makeCharArray();
    goto L1, L2, N;

L1: CHARS = STR1 java.lang.String.value;
    goto R;

L2: CHARS = STR2 java.lang.String.value;

R:  CH = CHARS java.lang.Object#intarrayelement;
    VALUE java.lang.Object#intarrayelement := CH;
    goto L1, L2, N;

N:  STR = new java.lang.String;
    java.lang.String.<init>(STR, VALUE) "([C)V";
    return = choose STR, STR1, STR2;
}

initStringconst() {
    STR = makeString();
    java.lang.String#internstr := STR;
}

/* Exception functions */

/* _stringconst is invoked to generate a String constant
used by one of the ldc* instructions.
   It's also used in native code specifications. */
_stringconst() {
    return = java.lang.String#internstr;
}

/* _magicexn is invoked at the start of a catch block to
generate all the exceptions that could be caught there.
*/
_magicexn() {
    goto L0, L1, L2, L3, L4, L5, L6, L7, L8, L9, L10,
L11, L12, L13, L14, L15, L16, L18, L19, L20, L21, L22,
L23, L24, L25;
L0:
    STR = _stringconst();
    EXN = new java.lang.VirtualMachineError;
    java.lang.VirtualMachineError.<init>(EXN);
    java.lang.VirtualMachineError.<init>(EXN, STR);
    goto L;
L1:
    STR = _stringconst();
    EXN = new java.lang.LinkageError;
    java.lang.LinkageError.<init>(EXN);
    java.lang.LinkageError.<init>(EXN, STR);
    goto L;
L2:
    STR = _stringconst();
    EXN = new java.lang.NullPointerException;
    java.lang.NullPointerException.<init>(EXN);
    java.lang.NullPointerException.<init>(EXN, STR);
    goto L;
L3:
    STR = _stringconst();
    EXN = new java.lang.ArrayIndexOutOfBoundsException;
    INT = choose; // not linked to the actual array
                  // index used
    java.lang.ArrayIndexOutOfBoundsException.<init>(EXN);
    java.lang.ArrayIndexOutOfBoundsException.<init>(EXN,
INT) "(I)V";
    java.lang.ArrayIndexOutOfBoundsException.<init>(EXN,
STR) "(Ljava.lang.String;)V";
    goto L;
L4:
    STR = _stringconst();
    EXN = new java.lang.ArrayStoreException;
    java.lang.ArrayStoreException.<init>(EXN);
    java.lang.ArrayStoreException.<init>(EXN, STR);
    goto L;
L5:
    STR = _stringconst();
    EXN = new java.lang.ArithmeticException;
    java.lang.ArithmeticException.<init>(EXN);
    java.lang.ArithmeticException.<init>(EXN, STR);
    goto L;
L6:
    STR = _stringconst();
    EXN = new java.lang.NegativeArraySizeException;
    java.lang.NegativeArraySizeException.<init>(EXN);
    java.lang.NegativeArraySizeException.<init>(EXN,
STR);
    goto L;
L7:
    STR = _stringconst();
    EXN = new java.lang.ClassCastException;
    java.lang.ClassCastException.<init>(EXN);
    java.lang.ClassCastException.<init>(EXN, STR);
    goto L;
L8:
    STR = _stringconst();
    EXN = new java.lang.IllegalMonitorStateException;
    java.lang.IllegalMonitorStateException.<init>(EXN);
    java.lang.IllegalMonitorStateException.<init>(EXN,
STR);
    goto L;
L9:
    EXN = new java.lang.ThreadDeath;
    java.lang.ThreadDeath.<init>(EXN);
    goto L;
L10:
    STR = _stringconst();
    EXN = new java.lang.InternalError;
    java.lang.InternalError.<init>(EXN);
    java.lang.InternalError.<init>(EXN, STR);
    goto L;
L11:
    STR = _stringconst();
    EXN = new java.lang.OutOfMemoryError;
    java.lang.OutOfMemoryError.<init>(EXN);
    java.lang.OutOfMemoryError.<init>(EXN, STR);
    goto L;
L12:
    STR = _stringconst();
    EXN = new java.lang.StackOverflowError;
    java.lang.StackOverflowError.<init>(EXN);
    java.lang.StackOverflowError.<init>(EXN, STR);
    goto L;
L13:
    STR = _stringconst();
    EXN = new java.lang.UnknownError;
    java.lang.UnknownError.<init>(EXN);
    java.lang.UnknownError.<init>(EXN, STR);
    goto L;
L14:
    STR = _stringconst();
    EXN = new java.lang.AbstractMethodError;
    java.lang.AbstractMethodError.<init>(EXN);
    java.lang.AbstractMethodError.<init>(EXN, STR);
    goto L;
L15:
    STR = _stringconst();
    EXN = new java.lang.ClassCircularityError;
    java.lang.ClassCircularityError.<init>(EXN);
    java.lang.ClassCircularityError.<init>(EXN, STR);
    goto L;
L16:
    STR = _stringconst();
    EXN = new java.lang.ClassFormatError;
    java.lang.ClassFormatError.<init>(EXN);
    java.lang.ClassFormatError.<init>(EXN, STR);
    goto L;
L18:
    STR = _stringconst();
    EXN = new java.lang.IllegalAccessError;
    java.lang.IllegalAccessError.<init>(EXN);
    java.lang.IllegalAccessError.<init>(EXN, STR);
    goto L;
L19:
    STR = _stringconst();
    EXN = new java.lang.IncompatibleClassChangeError;
    java.lang.IncompatibleClassChangeError.<init>(EXN);
    java.lang.IncompatibleClassChangeError.<init>(EXN,
STR);
    goto L;
L20:
    STR = _stringconst();
    EXN = new java.lang.InstantiationError;
    java.lang.InstantiationError.<init>(EXN);
    java.lang.InstantiationError.<init>(EXN, STR);
    goto L;
L21:
    STR = _stringconst();
    EXN = new java.lang.NoClassDefFoundError;
    java.lang.NoClassDefFoundError.<init>(EXN);
    java.lang.NoClassDefFoundError.<init>(EXN, STR);
    goto L;
L22:
    STR = _stringconst();
    EXN = new java.lang.NoSuchFieldError;
    java.lang.NoSuchFieldError.<init>(EXN);
    java.lang.NoSuchFieldError.<init>(EXN, STR);
    goto L;
L23:
    STR = _stringconst();
    EXN = new java.lang.NoSuchMethodError;
    java.lang.NoSuchMethodError.<init>(EXN);
    java.lang.NoSuchMethodError.<init>(EXN, STR);
    goto L;
L24:
    STR = _stringconst();
    EXN = new java.lang.UnsatisfiedLinkError;
    java.lang.UnsatisfiedLinkError.<init>(EXN);
    java.lang.UnsatisfiedLinkError.<init>(EXN, STR);
    goto L;
L25:
    STR = _stringconst();
    EXN = new java.lang.VerifyError;
    java.lang.VerifyError.<init>(EXN);
    java.lang.VerifyError.<init>(EXN, STR);
    goto L;
L:  return = choose EXN;
}

/* _wrapclassinitializerexn is invoked when a class
initializer method <clinit> is
   called. Any exception thrown by <clinit> is passed
through here to simulate the
   fact that the VM translates it to an
ExceptionInInitializerError. */
_wrapclassinitializerexn(REALEXN) {
    STR = _stringconst();
    EXN = new java.lang.ExceptionInInitializerError;
    java.lang.ExceptionInInitializerError.<init>(EXN);
    java.lang.ExceptionInInitializerError.<init>(EXN,
REALEXN) "(Ljava.lang.Throwable;)V";
    java.lang.ExceptionInInitializerError.<init>(EXN,
STR) "(Ljava.lang.String;)V";
    return = choose EXN;
}

makeIOException() {
    STR = _stringconst();
    EXN = new java.io.IOException;
    java.io.IOException.<init>(EXN);
    java.io.IOException.<init>(EXN, STR);
    return = choose EXN;
}

/* java.io.ObjectInputStream */

java.io.ObjectInputStream.loadClass0(C, NAME) {
    return = java.lang.Class.forName(NAME);
}

makeInvalidClassException(CLASS) {
    STR = _stringconst();
    CNAME = _stringconst();
    EXN = new java.io.InvalidClassException;
    java.io.InvalidClassException.<init>(EXN, CNAME);
    java.io.InvalidClassException.<init>(EXN, CNAME,
STR);
    return = choose EXN;
}

makeStreamCorruptedException() {
    STR = _stringconst();
    EXN = new java.io.StreamCorruptedException;
    java.io.StreamCorruptedException.<init>(EXN);
    java.io.StreamCorruptedException.<init>(EXN, STR);
    return = choose EXN;
}

java.io.ObjectInputStream.inputClassFields(THIS, OBJ,
CLASS, FIELDS) {
    FIELD = FIELDS java.lang.Object#arrayelement;

    goto B, S, C, I, J, Z, F, D, L;

B:  BYTE = java.io.ObjectInputStream.readByte(THIS);
    EXN1 = catch (java.lang.Throwable) BYTE;
    ReflectionHandler_assignSerializedFieldBYTE(OBJ,
CLASS, BYTE);
    goto DONE;

S:  SHORT = java.io.ObjectInputStream.readShort(THIS);
    EXN1 = catch (java.lang.Throwable) SHORT;
    ReflectionHandler_assignSerializedFieldSHORT(OBJ,
CLASS, SHORT);
    goto DONE;

C:  CHAR = java.io.ObjectInputStream.readChar(THIS);
```

```
    EXN1 = catch (java.lang.Throwable) CHAR;
    ReflectionHandler_assignSerializedFieldCHAR(OBJ,
CLASS, CHAR);
    goto DONE;

I:  INT = java.io.ObjectInputStream.readInt(THIS);
    EXN1 = catch (java.lang.Throwable) INT;
    ReflectionHandler_assignSerializedFieldINT(OBJ,
CLASS, INT);
    goto DONE;

J:  LONG = java.io.ObjectInputStream.readLong(THIS);
    EXN1 = catch (java.lang.Throwable) LONG;
    ReflectionHandler_assignSerializedFieldLONG(OBJ,
CLASS, LONG);
    goto DONE;

Z:  BOOL = java.io.ObjectInputStream.readBoolean(THIS);
    EXN1 = catch (java.lang.Throwable) BOOL;
    ReflectionHandler_assignSerializedFieldBOOL(OBJ,
CLASS, BOOL);
    goto DONE;

F:  FLOAT = java.io.ObjectInputStream.readFloat(THIS);
    EXN1 = catch (java.lang.Throwable) FLOAT;
    ReflectionHandler_assignSerializedFieldFLOAT(OBJ,
CLASS, FLOAT);
    goto DONE;

D:  DOUBLE = java.io.ObjectInputStream.readDouble(THIS);
    EXN1 = catch (java.lang.Throwable) DOUBLE;
    ReflectionHandler_assignSerializedFieldDOUBLE(OBJ,
CLASS, DOUBLE);
    goto DONE;

L:  OBJECT = java.io.ObjectInputStream.readObject(THIS);
    EXN1 = catch (java.lang.Throwable) OBJECT;
    ReflectionHandler_assignSerializedFieldOBJECT(OBJ,
CLASS, OBJECT);

DONE:
    EXN2 = makeClassNotFoundException();
    EXN3 = makeInvalidClassException(CLASS);
    EXN4 = makeStreamCorruptedException();
    throw = choose EXN1, EXN2, EXN3, EXN4;
}

java.io.ObjectInputStream.allocateNewObject(ACLASS,
INITCLASS) {
    OBJ = ReflectionHandler_makeSerializedObject(ACLASS);
    EXN1 = makeInstantiationException();
    EXN2 = makeIllegalAccessException();
    throw = choose EXN1, EXN2;
    return = choose OBJ;
}

java.io.ObjectInputStream.allocateNewArray(ARRAYCLASS,
LENGTH) {
    OBJ =
ReflectionHandler_makeSerializedArray(ARRAYCLASS);
    return = choose OBJ;
}

java.io.ObjectInputStream.invokeObjectReader(THIS, OBJ,
CLASS) {
    IO = ReflectionHandler_invoke_readObject(OBJ, CLASS,
THIS);

    EXN1 = catch (java.lang.Throwable) IO;
    EXN2 = makeClassNotFoundException();
    EXN3 = makeInvalidClassException(CLASS);
    EXN4 = makeStreamCorruptedException();
    throw = choose EXN1, EXN2, EXN3, EXN4;
}

/* java.io.ObjectOutputStream */

java.io.ObjectOutputStream.outputClassFields(THIS, OBJ,
CLASS, FIELDS) {
    FIELD = FIELDS java.lang.Object#arrayelement;

    goto B, S, C, I, J, Z, F, D, L;

B:  BYTE = ReflectionHandler_getSerializedFieldBYTE(OBJ,
CLASS);
    IO = java.io.ObjectOutputStream.writeByte(THIS,
BYTE);
    EXN1 = catch (java.lang.Throwable) IO;
    goto DONE;

S:  SHORT =
ReflectionHandler_getSerializedFieldSHORT(OBJ, CLASS);
    IO = java.io.ObjectOutputStream.writeShort(THIS,
SHORT);
    EXN1 = catch (java.lang.Throwable) IO;
    goto DONE;

C:  CHAR = ReflectionHandler_getSerializedFieldCHAR(OBJ,
CLASS);
    IO = java.io.ObjectOutputStream.writeChar(THIS,
CHAR);
    EXN1 = catch (java.lang.Throwable) IO;
    goto DONE;

I:  INT = ReflectionHandler_getSerializedFieldINT(OBJ,
CLASS);
    IO = java.io.ObjectOutputStream.writeInt(THIS, INT);
    EXN1 = catch (java.lang.Throwable) IO;
    goto DONE;

J:  LONG = ReflectionHandler_getSerializedFieldLONG(OBJ,
CLASS);
    IO = java.io.ObjectOutputStream.writeLong(THIS,
LONG);
    EXN1 = catch (java.lang.Throwable) IO;
    goto DONE;

Z:  BOOL = ReflectionHandler_getSerializedFieldBOOL(OBJ,
CLASS);
    IO = java.io.ObjectOutputStream.writeBoolean(THIS,
BOOL);
    EXN1 = catch (java.lang.Throwable) IO;
    goto DONE;

F:  FLOAT =
ReflectionHandler_getSerializedFieldFLOAT(OBJ, CLASS);
    IO = java.io.ObjectOutputStream.writeFloat(THIS,
FLOAT);
    EXN1 = catch (java.lang.Throwable) IO;
    goto DONE;

D:  DOUBLE =
ReflectionHandler_getSerializedFieldDOUBLE(OBJ, CLASS);
    IO = java.io.ObjectOutputStream.writeDouble(THIS,
DOUBLE);
    EXN1 = catch (java.lang.Throwable) IO;
    goto DONE;

L:  OBJECT =
ReflectionHandler_getSerializedFieldOBJECT(OBJ, CLASS);
    IO = java.io.ObjectOutputStream.writeObject(THIS,
OBJECT);
    EXN1 = catch (java.lang.Throwable) IO;

DONE:
    EXN2 = makeInvalidClassException(CLASS);
    throw = choose EXN1, EXN2;
}

java.io.ObjectOutputStream.invokeObjectWriter(THIS, OBJ,
CLASS) {
    IO = ReflectionHandler_invoke_writeObject(OBJ, CLASS,
THIS);

    throw = catch (java.lang.Throwable) IO;
}

/* java.io.ObjectStreamClass */

java.io.ObjectStreamClass.getClassAccess(C) {
    return = java.lang.Class.getModifiers(C);
}

java.io.ObjectStreamClass.getMethodSignatures(C) {
    return = makeConstStringArray();
}

java.io.ObjectStreamClass.getMethodAccess(C, SIG) {
    return = choose;
}

java.io.ObjectStreamClass.getFieldSignatures(C) {
    return = makeConstStringArray();
}

java.io.ObjectStreamClass.getFieldAccess(C, SIG) {
    return = choose;
}

java.io.ObjectStreamClass.getFields0(C) {
    LIST = new [Ljava.io.ObjectStreamField;
    java.lang.Object.<init>(LIST);
    LEN = choose;
    LIST java.lang.Object#arraylength := LEN;
L:  VALUE = new java.io.ObjectStreamField;
    NAME = _stringconst();
    T = choose;
    O = choose;
    TS = _stringconst();
    java.io.ObjectStreamField.<init>(VALUE, NAME, T, O,
TS);
    LIST java.lang.Object#arrayelement := VALUE;
    goto L, N;

N:  return = choose LIST;
}

java.io.ObjectStreamClass.getSerialVersionUID(C) {
    return = choose;
}

java.io.ObjectStreamClass.hasWriteObject(C) {
    return = choose;
}

/* java.io.FileDescriptor */

java.io.FileDescriptor.initSystemFD(FD, DESC) {
    FD java.io.FileDescriptor.fd := DESC;
    return = choose FD;
}

java.io.FileDescriptor.valid() {
    return = choose;
}

java.io.FileDescriptor.sync() {
    EXN = new java.io.SyncFailedException;
    STR = _stringconst();
    java.io.SyncFailedException.<init>(EXN, STR);
    throw = choose EXN;
}

/* java.io.FileInputStream */

java.io.FileInputStream.open(THIS, NAME) {
    FD = THIS java.io.FileInputStream.fd;
    NEWFD = choose;
    FD java.io.FileDescriptor.fd := NEWFD;
    throw = makeIOException();
}

makeInterruptedIOException() {
    STR = _stringconst();
    EXN = new java.io.InterruptedIOException;
    java.io.InterruptedIOException.<init>(EXN);
    java.io.InterruptedIOException.<init>(EXN, STR);
    NUM = choose;
    EXN java.io.InterruptedIOException.bytesTransferred
:= NUM;
    return = choose EXN;
}

java.io.FileInputStream.read(THIS) {
    return = choose;
    EXN1 = makeIOException();
    EXN2 = makeInterruptedIOException();
    throw = choose EXN1, EXN2;
    FD = THIS java.io.FileOutputStream.fd;
    OSFD = FD java.io.FileDescriptor.fd;
}

java.io.FileInputStream.readBytes(THIS, B, OFF, LEN) {
    return = choose LEN;
    EXN1 = makeIOException();
    EXN2 = makeInterruptedIOException();
    throw = choose EXN1, EXN2;
    FD = THIS java.io.FileOutputStream.fd;
    OSFD = FD java.io.FileDescriptor.fd;
}

java.io.FileInputStream.skip(THIS, N) {
    return = choose N;
    throw = makeIOException();
    FD = THIS java.io.FileOutputStream.fd;
    OSFD = FD java.io.FileDescriptor.fd;
}

java.io.FileInputStream.available(THIS) {
    return = choose;
    throw = makeIOException();
    FD = THIS java.io.FileOutputStream.fd;
    OSFD = FD java.io.FileDescriptor.fd;
}

java.io.FileInputStream.close(THIS) {
    throw = makeIOException();
    FD = THIS java.io.FileOutputStream.fd;
    OSFD = FD java.io.FileDescriptor.fd;
}

/* java.io.FileOutputStream */

java.io.FileOutputStream.open(THIS, NAME) {
    FD = THIS java.io.FileOutputStream.fd;
    NEWFD = choose;
    FD java.io.FileDescriptor.fd := NEWFD;
    throw = makeIOException();
}

java.io.FileOutputStream.openAppend(THIS, NAME) {
    FD = THIS java.io.FileOutputStream.fd;
    NEWFD = choose;
    FD java.io.FileDescriptor.fd := NEWFD;
    throw = makeIOException();
}

java.io.FileOutputStream.write(THIS, B) {
    EXN1 = makeIOException();
    EXN2 = makeInterruptedIOException();
    throw = choose EXN1, EXN2;
    FD = THIS java.io.FileOutputStream.fd;
    OSFD = FD java.io.FileDescriptor.fd;
}

java.io.FileOutputStream.writeBytes(THIS, B, OFF, LEN) {
    EXN1 = makeIOException();
    EXN2 = makeInterruptedIOException();
    throw = choose EXN1, EXN2;
    FD = THIS java.io.FileOutputStream.fd;
    OSFD = FD java.io.FileDescriptor.fd;
}

java.io.FileOutputStream.close(THIS) {
    throw = makeIOException();
    FD = THIS java.io.FileOutputStream.fd;
    OSFD = FD java.io.FileDescriptor.fd;
}

/* java.io.File */

java.io.File.lastModified0(THIS) {
    return = choose;
}

java.io.File.length0(THIS) {
    return = choose;
}

java.io.File.exists0(THIS) {
    return = choose;
}

java.io.File.canWrite0(THIS) {
    return = choose;
}

java.io.File.canRead0(THIS) {
    return = choose;
}

java.io.File.isFile0(THIS) {
    return = choose;
}

java.io.File.isDirectory0(THIS) {
    return = choose;
}

java.io.File.mkdir0(THIS) {
    return = choose;
}

java.io.File.delete0(THIS) {
    return = choose;
}

java.io.File.rmdir0(THIS) {
    return = choose;
}

java.io.File.renameTo0(THIS, DEST) {
    PATH = DEST java.io.File.path;
    THIS java.io.File.path := PATH;
    return = choose;
}
```

```
makeDynamicStringArray() {
    LIST = new [Ljava.lang.String;
    java.lang.Object.<init>(LIST);
    LEN = choose;
    LIST java.lang.Object#arraylength := LEN;

L:  STR = makeString();
    LIST java.lang.Object#arrayelement := STR;
    goto L, N;

N:  return = choose LIST;
}

makeConstStringArray() {
    LIST = new [Ljava.lang.String;
    java.lang.Object.<init>(LIST);
    LEN = choose;
    LIST java.lang.Object#arraylength := LEN;

L:  STR = _stringconst();
    LIST java.lang.Object#arrayelement := STR;
    goto L, N;

N:  return = choose LIST;
}

java.io.File.list0(THIS) {
    return = makeDynamicStringArray();
}

java.io.File.canonPath(THIS) {
    CURPATH = THIS java.io.File.path;
    STR = makeString();
    return = mungeStrings(CURPATH, STR);
}

java.io.File.isAbsolute(THIS) {
    return = choose;
}

/* java.io.RandomAccessFile */

java.io.RandomAccessFile.open(THIS, NAME, WRITEABLE) {
    FD = THIS java.io.RandomAccessFile.fd;
    NEWFD = choose;
    FD java.io.FileDescriptor.fd := NEWFD;
    throw = makeIOException();
}

java.io.RandomAccessFile.read(THIS) {
    return = choose;
    EXN1 = makeIOException();
    EXN2 = makeInterruptedIOException();
    throw = choose EXN1, EXN2;
}

java.io.RandomAccessFile.readBytes(THIS, B, OFF, LEN) {
    return = choose LEN;
    EXN1 = makeIOException();
    EXN2 = makeInterruptedIOException();
    throw = choose EXN1, EXN2;
}

java.io.RandomAccessFile.write(THIS, B) {
    EXN1 = makeIOException();
    EXN2 = makeInterruptedIOException();
    throw = choose EXN1, EXN2;
}

java.io.RandomAccessFile.writeBytes(THIS, B, OFF, LEN) {
    EXN1 = makeIOException();
    EXN2 = makeInterruptedIOException();
    throw = choose EXN1, EXN2;
}

java.io.RandomAccessFile.getFilePointer(THIS) {
    return = choose;
    throw = makeIOException();
}

java.io.RandomAccessFile.seek(THIS, POS) {
    throw = makeIOException();
}

java.io.RandomAccessFile.length(THIS) {
    return = choose;
    throw = makeIOException();
}

java.io.RandomAccessFile.close(THIS) {
    throw = makeIOException();
}
```

```
/* java.lang.Object */

java.lang.Object.hashCode(THIS) {
    HASH = THIS java.lang.Object#identity;
    return = choose HASH;
}

java.lang.Object.getClass(THIS) {
    return = makeClass();
}

java.lang.Object.clone(THIS) {
    STR = _stringconst();
    EXN1 = new java.lang.CloneNotSupportedException;
    java.lang.CloneNotSupportedException.<init>(EXN1);
    java.lang.CloneNotSupportedException.<init>(EXN1,
STR);
    throw = choose EXN1;
    return = choose THIS;
}

makeIllegalMonitorStateException() {
    STR = _stringconst();
    EXN = new java.lang.IllegalMonitorStateException;
    java.lang.IllegalMonitorStateException.<init>(EXN);
    java.lang.IllegalMonitorStateException.<init>(EXN,
STR);
    return = choose EXN;
}

java.lang.Object.notify(THIS) {
    throw = makeIllegalMonitorStateException();
}

java.lang.Object.notifyAll(THIS) {
    throw = makeIllegalMonitorStateException();
}

java.lang.Object.wait(THIS, TIMEOUT) {
    throw = makeIllegalMonitorStateException();
}

java.lang.Object.wait(THIS, TIMEOUT) {
    EXN1 = makeIllegalMonitorStateException();
    STR = _stringconst();
    EXN2 = new java.lang.IllegalArgumentException;
    java.lang.IllegalArgumentException.<init>(EXN1);
    java.lang.IllegalArgumentException.<init>(EXN1, STR);
    STR = _stringconst();
    EXN3 = new java.lang.InterruptedException;
    java.lang.InterruptedException.<init>(EXN3);
    java.lang.InterruptedException.<init>(EXN3, STR);
    throw = choose EXN1, EXN2, EXN3;
}

/* java.lang.Math */

java.lang.Math.sin(A) {
    return = choose;
}

java.lang.Math.cos(A) {
    return = choose;
}

java.lang.Math.tan(A) {
    return = choose;
}

java.lang.Math.asin(A) {
    return = choose;
}

java.lang.Math.acos(A) {
    return = choose;
}

java.lang.Math.atan(A) {
    return = choose;
}

java.lang.Math.exp(A) {
    return = choose;
}

java.lang.Math.log(A) {
    return = choose;
}

java.lang.Math.sqrt(A) {
    return = choose;
```

```
}

java.lang.Math.IEEERemainder(F1, F2) {
    return = choose;
}

java.lang.Math.ceil(A) {
    return = choose;
}

java.lang.Math.floor(A) {
    return = choose;
}

java.lang.Math.rint(A) {
    return = choose;
}

java.lang.Math.atan2(A, B) {
    return = choose;
}

java.lang.Math.pow(A, B) {
    return = choose;
}

/* java.lang.Float */

java.lang.Float.floatToIntBits(FLOAT) {
    return = choose;
}

java.lang.Float.intBitsToFloat(BITS) {
    return = choose;
}

/* java.lang.Double */

java.lang.Double.doubleToLongBits(DOUBLE) {
    return = choose;
}

java.lang.Double.longBitsToDouble(BITS) {
    return = choose;
}

java.lang.Double.valueOf0(S) {
    EXN = new java.lang.NumberFormatException;
    STR = _stringconst();
    java.lang.NumberFormatException.<init>(EXN);
    java.lang.NumberFormatException.<init>(EXN, STR);
    throw = choose EXN;
    return = choose;
}

/* java.lang.Throwable */

java.lang.Throwable.fillInStackTrace(THIS) {
    TRACE = choose;
    THIS java.lang.Throwable.backtrace := TRACE;
    return = choose THIS;
}

/* This doesn't really work. The printStackTrace0
documentation says that the STREAM should have a
println(char[]) method, but we don't know what class it's
in, so how can we call it? We probably need lots of extra
ugly support to get this really right. For now we just
ignore the STREAM. */
java.lang.Throwable.printStackTrace0(THIS, STREAM) {
}

/* java.lang.Thread */

java.lang.Thread.currentThread() {
    T = java.lang.Thread#currentthread;
    return = choose T;
}

java.lang.Thread.yield() {
}

java.lang.Thread.sleep(MILLIS) {
    EXN = new java.lang.InterruptedException;
    STR = _stringconst();
    java.lang.InterruptedException.<init>(EXN);
    java.lang.InterruptedException.<init>(EXN, STR);
    throw = choose EXN;
}

java.lang.Thread.start(THIS) {
    EXN = new java.lang.IllegalThreadStateException;
    STR = _stringconst();
    java.lang.IllegalThreadStateException.<init>(EXN);
    java.lang.IllegalThreadStateException.<init>(EXN,
STR);
    throw = choose EXN;
    java.lang.Thread.run(THIS);
}

// not sure what this does
java.lang.Thread.isInterrupted(THIS, CLEAR) {
    return = choose;
}

java.lang.Thread.isAlive(THIS) {
    return = choose;
}

java.lang.Thread.countStackFrames(THIS) {
    return = choose;
}

java.lang.Thread.setPriority0(THIS, PRIORITY) {
}

java.lang.Thread.stop0(THIS) {
}

java.lang.Thread.suspend0(THIS) {
}

java.lang.Thread.resume0(THIS) {
}

java.lang.Thread.interrupt0(THIS) {
}

/* java.lang.Compiler */

java.lang.Compiler.initialize() {
}

java.lang.Compiler.compileClass(C) {
    return = choose;
}

java.lang.Compiler.compileClasses(CS) {
    return = choose;
}

java.lang.Compiler.commmand(C) {
    return = choose;
}

java.lang.Compiler.enable() {
}

java.lang.Compiler.disable() {
}

/* java.lang.Win32Process */

java.lang.Win32Process.exitValue() {
    result = choose;
}

java.lang.Win32Process.waitFor() {
    result = choose;
}

java.lang.Win32Process.destroy() {
}

java.lang.Win32Process.create(CMD, ENV) {
    accessStringChars(CMD);
    accessStringChars(ENV);
}

java.lang.Win32Process.close() {
}

/* java.lang.Runtime */

java.lang.Runtime.exitInternal(THIS, STATUS) {
}

java.lang.Runtime.runFinalizersOnExit0(THIS, VALUE) {
}

java.lang.Runtime.execInternal(THIS, CMDARRAY, ENVP) {
    PROCESS = new java.lang.Win32Process;
    java.lang.Win32Process.<init>(PROCESS, CMDARRAY,
ENVP);
```

```
    return = choose PROCESS;
}

java.lang.Runtime.freeMemory(THIS) {
    return = choose;
}

java.lang.Runtime.totalMemory(THIS) {
    return = choose;
}

java.lang.Runtime.gc(THIS) {
}

java.lang.Runtime.runFinalization(THIS) {
}

java.lang.Runtime.traceInstructions(THIS, ON) {
}

java.lang.Runtime.traceMethodCalls(THIS, ON) {
}

java.lang.Runtime.initializeLinkerInternal(THIS) {
    return = java.lang.String#internstr;
}

java.lang.Runtime.buildLibName(THIS, PATHNAME, FILENAME)
{
    BUF = new java.lang.StringBuffer;
    java.lang.StringBuffer.<init>(BUF, PATHNAME)
"(Ljava.lang.String;)V";
    STR = java.lang.String#internstr;
    java.lang.StringBuffer.append(BUF, STR)
"(Ljava.lang.String;)Ljava.lang.StringBuffer;";
    java.lang.StringBuffer.append(BUF, FILENAME)
"(Ljava.lang.String;)Ljava.lang.StringBuffer;";
    STR = java.lang.String#internstr;
    java.lang.StringBuffer.append(BUF, STR)
"(Ljava.lang.String;)Ljava.lang.StringBuffer;";
    return = java.lang.StringBuffer.toString(BUF);
}

java.lang.Runtime.loadFileInternal(THIS, FILENAME) {
    return = choose;
}

/* java.lang.String */
java.lang.String.intern(THIS) {
    goto Y, N;

Y:  java.lang.String#internstr := THIS;

N:  return = java.lang.String#internstr;
}

/* java.lang.System */

java.lang.System.currentTimeMillis() {
    // this just returns an arbitrary fresh value
    return = choose;
}

java.lang.System.identityHashCode(OBJ) {
    HASH = OBJ java.lang.Object#identity;
    return = choose HASH;
}

// This one might need to be changed. In particular, it
might call
// Properties.read
java.lang.System.initProperties(PROPS) {
    PROP = makeString();
    STR = makeString();
    java.util.Hashtable.put(PROPS, PROP, STR);
    return = choose PROPS;
}

java.lang.System.setIn0(IN) {
    java.lang.System.in := IN;
}

java.lang.System.setOut0(OUT) {
    java.lang.System.out := OUT;
}

java.lang.System.setErr0(ERR) {
    java.lang.System.err := ERR;
}

java.lang.System.setIn0(IN) {
    java.lang.System.in := IN;
}

java.lang.System.arraycopy(FROM, FROMOFF, TO, TOOFF, LEN)
{
    VAL = FROM java.lang.Object#arrayelement;
    TO java.lang.Object#arrayelement := VAL;
    VAL = FROM java.lang.Object#intarrayelement;
    TO java.lang.Object#intarrayelement := VAL;
    VAL = FROM java.lang.Object#floatarrayelement;
    TO java.lang.Object#floatarrayelement := VAL;
    VAL = FROM java.lang.Object#longarrayelement;
    TO java.lang.Object#longarrayelement := VAL;
    VAL = FROM java.lang.Object#doublearrayelement;
    TO java.lang.Object#doublearrayelement := VAL;
}

/* java.lang.Class */

makeClass() {
    CLASS = new java.lang.Class;
    java.lang.Class.<init>(CLASS);
    java.lang.Class#internclass := CLASS;
    return = java.lang.Class#internclass;
}

makeSigner() {
    return = java.lang.Class#internsigner;
}

makeClassArray() {
    CS = new [Ljava.lang.Class;
    java.lang.Object.<init>(CS);
    LEN = choose;
    CS java.lang.Object#arraylength := LEN;

L:  C = makeClass();
    CS java.lang.Object#arrayelement := C;
    goto L, N;

N:  return = choose CS;
}

makeField(CLASS) {
    FIELD = new java.lang.reflect.Field;
    java.lang.reflect.Field.<init>(FIELD);
    FIELD java.lang.reflect.Field.clazz := CLASS;
    SLOT = choose;
    FIELD java.lang.reflect.Field.slot := SLOT;
    NAME = _stringconst();
    FIELD java.lang.reflect.Field.name := NAME;
    TYPE = makeClass();
    FIELD java.lang.reflect.Field.type := TYPE;

    java.lang.Field#internfield := FIELD;
    return = java.lang.Field#internfield;
}

makeMethod(CLASS) {
    METHOD = new java.lang.reflect.Method;
    java.lang.reflect.Method.<init>(METHOD);
    METHOD java.lang.reflect.Method.clazz := CLASS;
    SLOT = choose;
    METHOD java.lang.reflect.Method.slot := SLOT;
    NAME = _stringconst();
    METHOD java.lang.reflect.Method.name := NAME;
    RETURNTYPE = makeClass();
    METHOD java.lang.reflect.Method.returnType :=
RETURNTYPE;
    PARAMETERTYPES = makeClassArray();
    METHOD java.lang.reflect.Method.parameterTypes :=
PARAMETERTYPES;
    EXCEPTIONTYPES = makeClassArray();
    METHOD java.lang.reflect.Method.exceptionTypes :=
EXCEPTIONTYPES;
    MODS = choose;
    METHOD java.lang.reflect.Constructor#mods := MODS;

    java.lang.reflect.Method#internmethod := METHOD;
    return = java.lang.reflect.Method#internmethod;
}

makeConstructor(CLASS) {
    CONSTRUCTOR = new java.lang.reflect.Constructor;
    java.lang.reflect.Constructor.<init>(CONSTRUCTOR);
    CONSTRUCTOR java.lang.reflect.Constructor.clazz :=
CLASS;
    SLOT = choose;
    CONSTRUCTOR java.lang.reflect.Constructor.slot :=
SLOT;
    PARAMETERTYPES = makeClassArray();
    CONSTRUCTOR
java.lang.reflect.Constructor.parameterTypes :=
PARAMETERTYPES;
    EXCEPTIONTYPES = makeClassArray();
    CONSTRUCTOR
java.lang.reflect.Constructor.exceptionTypes :=
EXCEPTIONTYPES;
    MODS = choose;
    CONSTRUCTOR java.lang.reflect.Constructor#mods :=
MODS;

    java.lang.reflect.Constructor#internconstructor :=
CONSTRUCTOR;
    return =
java.lang.reflect.Constructor#internconstructor;
}

makeInstantiationException() {
    STR = _stringconst();
    EXN = new java.lang.InstantiationException;
    java.lang.InstantiationException.<init>(EXN);
    java.lang.InstantiationException.<init>(EXN, STR);
    return = choose EXN;
}

makeIllegalAccessException() {
    STR = _stringconst();
    EXN = new java.lang.IllegalAccessException;
    java.lang.IllegalAccessException.<init>(EXN);
    java.lang.IllegalAccessException.<init>(EXN, STR);
    result = choose EXN;
}

makeIllegalArgumentException() {
    STR = _stringconst();
    EXN = new java.lang.IllegalArgumentException;
    java.lang.IllegalArgumentException.<init>(EXN);
    java.lang.IllegalArgumentException.<init>(EXN, STR);
    result = choose EXN;
}

makeInvocationTargetException(CATCH) {
    STR = _stringconst();
    EXN = new
java.lang.reflect.InvocationTargetException;

java.lang.reflect.InvocationTargetException.<init>(EXN);

java.lang.reflect.InvocationTargetException.<init>(EXN,
CATCH);

java.lang.reflect.InvocationTargetException.<init>(EXN,
CATCH, STR);
    result = choose EXN;
}

makeClassNotFoundException() {
    STR = _stringconst();
    EXN = new java.lang.ClassNotFoundException;
    java.lang.ClassNotFoundException.<init>(EXN);
    java.lang.ClassNotFoundException.<init>(EXN, STR);
    return = choose EXN;
}

java.lang.Class.forName(NAME) {
    throw = makeClassNotFoundException();
    return = makeClass();
}

java.lang.Class.newInstance(CLASS) {
    OBJ =
ReflectionHandler_makeObjectAndCallZeroArgConstructor(CLA
SS);
    EXN1 = makeInstantiationException();
    EXN2 = makeIllegalAccessException();
    throw = choose EXN1, EXN2;
    return = choose OBJ;
}

java.lang.Class.isInstance(C) {
    return = choose;
}

java.lang.Class.isAssignableFrom(C) {
    return = choose;
}

java.lang.Class.isInterface(C) {
    return = choose;
}

java.lang.Class.isArray(C) {
    return = choose;
}

java.lang.Class.isPrimitive(C) {
    return = choose;
}

java.lang.Class.getName(C) {
    STR = _stringconst();
    return = choose STR;
}

java.lang.Class.getClassLoader(C) {
    return = makeClassLoader();
}

java.lang.Class.getSuperclass(C) {
    return = makeClass();
}

java.lang.Class.getInterfaces(C) {
    return = makeClassArray();
}

java.lang.Class.getComponentType(C) {
    return = makeClass();
}

java.lang.Class.getModifiers(C) {
    return = choose;
}

java.lang.Class.getSigners(C) {
    OS = new [Ljava.lang.Object;
    java.lang.Object.<init>(OS);
    LEN = choose;
    OS java.lang.Object#arraylength := LEN;

L:  O = makeSigner();
    OS java.lang.Object#arrayelement := O;
    goto L, N;

N:  return = choose OS;
}

java.lang.Class.setSigners(OS) {
L:  O = OS java.lang.Object#arrayelement;
    java.lang.Class#internsigner := O;
    goto L, N;

N:  return = choose;
}

java.lang.Class.getPrimitiveClass(NAME) {
    return = makeClass();
}

java.lang.Class.getDeclaringClass(C) {
    return = makeClass();
}

java.lang.Class.getClasses(C) {
    return = makeClassArray();
}

java.lang.Class.getFields0(THIS, WHICH) {
    FS = new [Ljava.lang.reflect.Field;
    java.lang.Object.<init>(FS);
    LEN = choose;
    FS java.lang.Object#arraylength := LEN;

L:  F = makeField(THIS);
    FS java.lang.Object#arrayelement := F;
    goto L, N;

N:  return = choose FS;
}

java.lang.Class.getField0(THIS, NAME, WHICH) {
    STR = _stringconst();
    EXN = new java.lang.NoSuchFieldException;
    java.lang.NoSuchFieldException.<init>(EXN);
    java.lang.NoSuchFieldException.<init>(EXN, STR);
    throw = choose EXN;

    return = makeField(THIS);
}

java.lang.Class.getMethods0(THIS, WHICH) {
    MS = new [Ljava.lang.reflect.Method;
    java.lang.Object.<init>(MS);
    LEN = choose;
    MS java.lang.Object#arraylength := LEN;

L:  M = makeMethod(THIS);
    MS java.lang.Object#arrayelement := M;
    goto L, N;

N:  return = choose MS;
}

makeNoSuchMethodException() {
    STR = _stringconst();
    EXN = new java.lang.NoSuchMethodException;
    java.lang.NoSuchMethodException.<init>(EXN);
    java.lang.NoSuchMethodException.<init>(EXN, STR);
    return = choose EXN;
}

java.lang.Class.getMethod0(THIS, NAME, PARAMETERTYPES,
WHICH) {
    throw = makeNoSuchMethodException();
    return = makeMethod(THIS);
}

java.lang.Class.getConstructors0(THIS, WHICH) {
    CS = new [Ljava.lang.reflect.Constructor;
    java.lang.Object.<init>(CS);
    LEN = choose;
    CS java.lang.Object#arraylength := LEN;

L:  C = makeConstructor(THIS);
    CS java.lang.Object#arrayelement := C;
    goto L, N;

N:  return = choose CS;
}

java.lang.Class.getConstructor0(THIS, PARAMETERTYPES,
WHICH) {
    throw = makeNoSuchMethodException();
    return = makeConstructor(THIS);
}

/* java.lang.ClassLoader */

makeClassLoader() {
    return = java.lang.ClassLoader#internloader;
}

java.lang.ClassLoader.init(THIS) {
    java.lang.ClassLoader#internloader := THIS;
}

java.lang.ClassLoader.defineClass0(THIS, NAME, DATA,
OFFSET, LENGTH) {
    return = makeClass();
}

java.lang.ClassLoader.resolveClass0(THIS, C) {
}

java.lang.ClassLoader.findSystemClass0(THIS, NAME) {
    throw = makeClassNotFoundException();
    return = makeClass();
}

java.lang.ClassLoader.getSystemResourceAsStream0(THIS,
NAME) {
    URL = java.lang.ClassLoader.getSystemResource(NAME);
    return = java.net.URL.openStream(URL);
}

java.lang.ClassLoader.getSystemResourceAsName0(THIS,
NAME) {
    return = _stringconst();
}

/* java.lang.reflect.Constructor */

java.lang.reflect.Constructor.getModifiers(THIS) {
    return = THIS java.lang.reflect.Constructor#mods;
}

java.lang.reflect.Constructor.newInstance(THIS, ARGS) {
    ARGS java.lang.Object#arraylength;
    OBJ =
ReflectionHandler_makeObjectAndCallArbitraryConstructor(A
RGS);
    CATCH = catch (java.lang.Throwable) OBJ;
    EXN1 = makeInstantiationException();
    EXN2 = makeIllegalAccessException();
    EXN3 = makeIllegalArgumentException();
    EXN4 = makeInvocationTargetException(CATCH);
    throw = choose EXN1, EXN2, EXN3, EXN4;
    return = choose OBJ;
}

/* java.lang.reflect.Method */

java.lang.reflect.Method.getModifiers(THIS) {
    return = THIS java.lang.reflect.Method#mods;
}

java.lang.reflect.Method.invoke(THIS, TARGET, ARGS) {
    ARGS java.lang.Object#arraylength;
    OBJ = ReflectionHandler_callArbitraryMethod(TARGET,
ARGS);
    CATCH = catch (java.lang.Throwable) OBJ;
    EXN2 = makeIllegalAccessException();
    EXN3 = makeIllegalArgumentException();
    EXN4 = makeInvocationTargetException(CATCH);
    throw = choose EXN2, EXN3, EXN4;
    return = choose OBJ;
}

/* java.util.ResourceBundle */

java.util.ResourceBundle.getClassContext() {
    return = makeClassArray();
}

/* java.util.zip.Inflater */

java.util.zip.Inflater.setDictionary(THIS, B, OFF, LEN) {
    THIS java.util.zip.Inflater.strm;

    NEWNEEDDICT = choose;
    THIS java.util.zip.Inflater.needsDictionary :=
NEWNEEDDICT;
}

java.util.zip.Inflater.inflate(THIS, B, OFF, LEN) {
    THIS java.util.zip.Inflater.strm;

    VAL = choose;
    B java.lang.Object#intarrayelement := VAL;
    NEWLEN = choose;
    THIS java.util.zip.Inflater.len := NEWLEN;
    NEWTOTALIN = choose;
    THIS java.util.zip.Inflater#totalIn := NEWTOTALIN;
    NEWTOTALOUT = choose;
    THIS java.util.zip.Inflater#totalOut := NEWTOTALOUT;
    NEWOFF = choose;
    THIS java.util.zip.Inflater.off := NEWOFF;
    NEWFINISHED = choose;
    THIS java.util.zip.Inflater.finished := NEWFINISHED;
    NEWNEEDDICT = choose;
    THIS java.util.zip.Inflater.needsDictionary :=
NEWNEEDDICT;

    EXN = new java.util.zip.DataFormatException;
    STR = _stringconst();
    java.util.zip.DataFormatException.<init>(EXN);
    java.util.zip.DataFormatException.<init>(EXN, STR);
    throw = choose EXN;
}

java.util.zip.Inflater.getAdler(THIS) {
    THIS java.util.zip.Inflater.strm;

    return = choose;
}

java.util.zip.Inflater.getTotalIn(THIS) {
    THIS java.util.zip.Inflater.strm;

    return = THIS java.util.zip.Inflater#totalIn;
}

java.util.zip.Inflater.getTotalOut(THIS) {
    THIS java.util.zip.Inflater.strm;

    return = THIS java.util.zip.Inflater#totalOut;
}

java.util.zip.Inflater.reset(THIS) {
    THIS java.util.zip.Inflater.strm;

    NEWTOTALIN = choose;
    THIS java.util.zip.Inflater#totalIn := NEWTOTALIN;
    NEWTOTALOUT = choose;
    THIS java.util.zip.Inflater#totalOut := NEWTOTALOUT;
    NEWFINISHED = choose;
    THIS java.util.zip.Inflater.finished := NEWFINISHED;
    NEWNEEDDICT = choose;
    THIS java.util.zip.Inflater.needsDictionary :=
NEWNEEDDICT;
}

java.util.zip.Inflater.end(THIS) {
    THIS java.util.zip.Inflater.strm;
}

java.util.zip.Inflater.init(THIS, NOWRAP) {
    STRM = choose;
    THIS java.util.zip.Inflater.strm := STRM;
    java.util.zip.Inflater.reset(THIS);
}

/* java.util.zip.Deflater */

accessDeflater(THIS) {
    THIS java.util.zip.Deflater.setParams;
    THIS java.util.zip.Deflater.strm;
    THIS java.util.zip.Deflater.finish;
    THIS java.util.zip.Deflater.level;
    THIS java.util.zip.Deflater.strategy;

    FALSE = choose;
    THIS java.util.zip.Deflater.setParams := FALSE;
}

java.util.zip.Deflater.setDictionary(THIS, B, OFF, LEN) {
    accessDeflater(THIS);
}

java.util.zip.Deflater.deflate(THIS, B, OFF, LEN) {
    accessDeflater(THIS);

    VAL = choose;
    B java.lang.Object#intarrayelement := VAL;
    NEWLEN = choose;
    THIS java.util.zip.Deflater.len := NEWLEN;
    NEWTOTALIN = choose;
    THIS java.util.zip.Deflater#totalIn := NEWTOTALIN;
    NEWTOTALOUT = choose;
    THIS java.util.zip.Deflater#totalOut := NEWTOTALOUT;
    NEWOFF = choose;
    THIS java.util.zip.Deflater.off := NEWOFF;
    NEWFINISHED = choose;
    THIS java.util.zip.Deflater.finished := NEWFINISHED;

    return = choose;
}

java.util.zip.Deflater.getAdler(THIS) {
    accessDeflater(THIS);

    return = choose;
}

java.util.zip.Deflater.getTotalIn(THIS) {
    accessDeflater(THIS);

    return = THIS java.util.zip.Deflater#totalIn;
}

java.util.zip.Deflater.getTotalOut(THIS) {
    accessDeflater(THIS);

    return = THIS java.util.zip.Deflater#totalOut;
}

java.util.zip.Deflater.reset(THIS) {
    accessDeflater(THIS);

    NEWTOTALIN = choose;
    THIS java.util.zip.Deflater#totalIn := NEWTOTALIN;
    NEWTOTALOUT = choose;
    THIS java.util.zip.Deflater#totalOut := NEWTOTALOUT;
    NEWFINISHED = choose;
    THIS java.util.zip.Deflater.finished := NEWFINISHED;
}

java.util.zip.Deflater.end(THIS) {
    accessDeflater(THIS);
}

java.util.zip.Deflater.init(THIS, NOWRAP) {
    STRM = choose;
    THIS java.util.zip.Deflater.strm := STRM;
    java.util.zip.Deflater.reset(THIS);
}

/* java.util.zip.CRC32 */

java.util.zip.CRC32.update(THIS, B, OFF, LEN) {
    VAL = choose;
    THIS java.util.zip.CRC32.crc := VAL;

    B java.lang.Object#intarrayelement;
}

java.util.zip.CRC32.update1(THIS, B) {
    VAL = choose;
    THIS java.util.zip.CRC32.crc := VAL;
}

/* java.awt.image.ColorModel */

java.awt.image.ColorModel.deletepData(THIS) {
}

/* sun.awt.windows.WToolkit */

sun.awt.windows.WToolkit.init(THIS, EVENTTHREAD /*
java.lang.Thread */) {
}

sun.awt.windows.WToolkit.eventLoop(THIS) {
T:  goto EA, EB, EC, ED, EE, EF, EG, EH, EI, EJ, EK, EL,
EM, EN, EY, EZ, E0, E1, E2, E3, E4, E5, E6, EXIT;

EA: TARGET = sun.awt.windows.WComponentPeer#allPeers;
    ACTION = choose;
    sun.awt.windows.WChoicePeer.handleAction(TARGET,
ACTION);
    goto T;

EB: TARGET = sun.awt.windows.WComponentPeer#allPeers;
    sun.awt.windows.WButtonPeer.handleAction(TARGET);
    goto T;

EC: TARGET = sun.awt.windows.WComponentPeer#allPeers;
    AMT = choose;
    sun.awt.windows.WScrollbarPeer.lineUp(TARGET, AMT);
    goto T;

ED: TARGET = sun.awt.windows.WComponentPeer#allPeers;
    AMT = choose;
    sun.awt.windows.WScrollbarPeer.lineDown(TARGET, AMT);
    goto T;

EE: TARGET = sun.awt.windows.WComponentPeer#allPeers;
    AMT = choose;
    sun.awt.windows.WScrollbarPeer.pageUp(TARGET, AMT);
    goto T;

EF: TARGET = sun.awt.windows.WComponentPeer#allPeers;
    AMT = choose;
    sun.awt.windows.WScrollbarPeer.pageDown(TARGET, AMT);
    goto T;

EG: TARGET = sun.awt.windows.WComponentPeer#allPeers;
    AMT = choose;
    sun.awt.windows.WScrollbarPeer.dragBegin(TARGET,
AMT);
    goto T;

EH: TARGET = sun.awt.windows.WComponentPeer#allPeers;
    AMT = choose;
    sun.awt.windows.WScrollbarPeer.dragAbsolute(TARGET,
AMT);
    goto T;

EI: TARGET = sun.awt.windows.WComponentPeer#allPeers;
    AMT = choose;
    sun.awt.windows.WScrollbarPeer.dragEnd(TARGET, AMT);
    goto T;

EJ: TARGET = sun.awt.windows.WMenuItemPeer#menuItemPeers;
    CODE = choose;
    sun.awt.windows.WMenuItemPeer.handleAction(TARGET,
CODE);
    goto T;

EK: TARGET = sun.awt.windows.WComponentPeer#allPeers;
    sun.awt.windows.WFileDialogPeer.handleCancel(TARGET);
    goto T;

EL: TARGET = sun.awt.windows.WComponentPeer#allPeers;
    STR = makeString();

sun.awt.windows.WFileDialogPeer.handleSelected(TARGET,
STR);
    goto T;

EM: TARGET = sun.awt.windows.WComponentPeer#allPeers;
```

```
sun.awt.windows.WWindowPeer.postFocusOnActivate(TARGET);
    goto T;

EN: TARGET = sun.awt.windows.WComponentPeer#allPeers;
    sun.awt.windows.WTextFieldPeer.handleAction(TARGET);
    goto T;

EY: TARGET = sun.awt.windows.WComponentPeer#allPeers;
    X = choose;
    Y = choose;
    W = choose;
    H = choose;
    sun.awt.windows.WComponentPeer.handleRepaint(TARGET,
X, Y, W, H);
    goto T;

EZ: TARGET = sun.awt.windows.WComponentPeer#allPeers;
    X = choose;
    Y = choose;
    W = choose;
    H = choose;
    sun.awt.windows.WComponentPeer.handleExpose(TARGET,
X, Y, W, H);
    goto T;

E0: TARGET = sun.awt.windows.WComponentPeer#allPeers;
    X = choose;
    Y = choose;
    W = choose;
    H = choose;
    sun.awt.windows.WComponentPeer.handlePaint(TARGET, X,
Y, W, H);
    goto T;

E1: CLIPBOARD = sun.awt.windows.WToolkit#theClipboard;

sun.awt.windows.WClipboard.lostSelectionOwnership(CLIPBOA
RD);
    goto T;

E2: EVT = new java.awt.event.KeyEvent;
    TARGET = sun.awt.windows.WComponentPeer#allPeers;
    TARGET = TARGET sun.awt.windows.WObjectPeer.target;
    ID = choose;
    WHEN = choose;
    MODS = choose;
    KEYCODE = choose;
    KEYCHAR = choose;
    java.awt.event.KeyEvent.<init>(EVT, TARGET, ID, WHEN,
MODS, KEYCODE, KEYCHAR);
    goto POST;

E3: EVT = new java.awt.event.MouseEvent;
    TARGET = sun.awt.windows.WComponentPeer#allPeers;
    TARGET = TARGET sun.awt.windows.WObjectPeer.target;
    ID = choose;
    WHEN = choose;
    MODS = choose;
    X = choose;
    Y = choose;
    CLICKS = choose;
    POPUP = choose;
    java.awt.event.MouseEvent.<init>(EVT, TARGET, ID,
WHEN, MODS, X, Y, CLICKS, POPUP);
    goto POST;

E4: EVT = new java.awt.event.WindowEvent;
    TARGET = sun.awt.windows.WComponentPeer#allPeers;
    TARGET = TARGET sun.awt.windows.WObjectPeer.target;
    ID = choose;
    java.awt.event.WindowEvent.<init>(EVT, TARGET, ID);
    goto POST;

E5: TARGET = sun.awt.windows.WComponentPeer#allPeers;

sun.awt.windows.WTextComponentPeer.valueChanged(TARGET);
    goto T;

E6: EVT = new java.awt.event.FocusEvent;
    TARGET = sun.awt.windows.WComponentPeer#allPeers;
    TARGET = TARGET sun.awt.windows.WObjectPeer.target;
    ID = choose;
    ISTMP = choose;
    java.awt.event.FocusEvent.<init>(EVT, TARGET, ID,
ISTMP);
    goto POST;

POST:
    sun.awt.windows.WToolkit.postEvent(EVT);
    goto T;

EXIT:
    choose;
}

sun.awt.windows.WToolkit.getComboHeightOffset() {
    return = choose; /* int */
}

sun.awt.windows.WToolkit.makeColorModel() {
    BITS = choose;

    RMASK = choose;
    GMASK = choose;
    BMASK = choose;
    AMASK = choose;
    M1 = new java.awt.image.DirectColorModel;
    java.awt.image.DirectColorModel.<init>(M1, BITS,
RMASK, GMASK, BMASK, AMASK);

    SIZE = choose;
    CMAP = makeByteArray();
    START = choose;
    HASALPHA = choose;
    TRANS = choose;
    M2 = new java.awt.image.IndexColorModel;
    java.awt.image.IndexColorModel.<init>(M2, BITS, SIZE,
CMAP, START, HASALPHA, TRANS) "(II[BIZI)V";

    return = choose M1, M2;
}

sun.awt.windows.WToolkit.getScreenResolution(THIS) {
    return = choose; /* int */
}

sun.awt.windows.WToolkit.getScreenWidth(THIS) {
    return = choose; /* int */
}

sun.awt.windows.WToolkit.getScreenHeight(THIS) {
    return = choose; /* int */
}

sun.awt.windows.WToolkit.sync(THIS) {
}

sun.awt.windows.WToolkit.beep(THIS) {
}

sun.awt.windows.WToolkit.loadSystemColors(THIS,
COLORARRAY /* int[] */) {
    COLORARRAY java.lang.Object#arraylength;
    VAL = choose;
    COLORARRAY java.lang.Object#intarrayelement := VAL;
}

/* sun.awt.windows.WObjectPeer */

sun.awt.windows.WObjectPeer.initIDs() {
}

/* sun.awt.windows.WComponentPeer */

makePoint(X, Y) {
    X = choose;
    Y = choose;
    P = new java.awt.Point;
    java.awt.Point.<init>(P, X, Y);
    return = choose P;
}

sun.awt.windows.WComponentPeer._beginValidate(THIS) {
}

sun.awt.windows.WComponentPeer.endValidate(THIS) {
}

sun.awt.windows.WComponentPeer.start(THIS) {
    X = choose;
    Y = choose;
    THIS sun.awt.windows.WComponentPeer#X := X;
    THIS sun.awt.windows.WComponentPeer#Y := Y;
    sun.awt.windows.WComponentPeer#allPeers := THIS;
}

sun.awt.windows.WComponentPeer._dispose(THIS) {
}

sun.awt.windows.WComponentPeer.disable(THIS) {
}

sun.awt.windows.WComponentPeer.enable(THIS) {
}

sun.awt.windows.WComponentPeer.hide(THIS) {
}

sun.awt.windows.WComponentPeer.show(THIS) {
}

sun.awt.windows.WComponentPeer.reshape(THIS, X, Y, W, H)
{
    THIS sun.awt.windows.WComponentPeer#X := X;
    THIS sun.awt.windows.WComponentPeer#Y := Y;
}

sun.awt.windows.WComponentPeer.getLocationOnScreen(THIS)
{
    X = THIS sun.awt.windows.WComponentPeer#X;
    Y = THIS sun.awt.windows.WComponentPeer#Y;
    P = new java.awt.Point;
    java.awt.Point.<init>(P, X, Y);
    return = choose P;
}

sun.awt.windows.WComponentPeer.setCursor(THIS, CURSOR) {
}

sun.awt.windows.WComponentPeer.setFont(THIS, FONT) {
}

sun.awt.windows.WComponentPeer.setZOrderPosition(THIS,
COMPONENT) {
}

sun.awt.windows.WComponentPeer._setBackground(THIS,
COLOR) {
}

sun.awt.windows.WComponentPeer._setForeground(THIS,
COLOR) {
}

sun.awt.windows.WComponentPeer.addNativeDropTarget(THIS)
{
}

sun.awt.windows.WComponentPeer.removeNativeDropTarget(THI
S) {
}

sun.awt.windows.WComponentPeer.nativeHandleEvent(THIS,
EVENT) {
}

sun.awt.windows.WComponentPeer.requestFocus(THIS) {
}

/* sun.awt.windows.WWindowPeer */

sun.awt.windows.WWindowPeer.create(THIS, PARENT) {
    PDATA = choose;
    THIS sun.awt.windows.WObjectPeer.pData := PDATA;
}

sun.awt.windows.WWindowPeer._setResizable(THIS, BOOL) {
}

sun.awt.windows.WWindowPeer._setTitle(THIS, STR) {
}

sun.awt.windows.WWindowPeer.toBack(THIS) {
}

sun.awt.windows.WWindowPeer.toFront(THIS) {
}

sun.awt.windows.WWindowPeer.updateInsets(THIS, INSETS) {
}

sun.awt.windows.WWindowPeer.getContainerElement(THIS,
CONTAINER, INDEX) {
    return = java.awt.Container.getComponent(CONTAINER,
INDEX);
}

/* sun.awt.windows.WFramePeer */

sun.awt.windows.WFramePeer.create(THIS, PARENT) {
    PDATA = choose;
    THIS sun.awt.windows.WObjectPeer.pData := PDATA;
    STATE = choose;
    THIS sun.awt.windows.WFramePeer#state := STATE;
}

sun.awt.windows.WFramePeer.getState(THIS) {
    return = THIS sun.awt.windows.WFramePeer#state;
}

sun.awt.windows.WFramePeer._setIconImage(THIS, REP) {
}

sun.awt.windows.WFramePeer.getSysIconHeight(THIS) {
    return = choose;
}

sun.awt.windows.WFramePeer.getSysIconWidth(THIS) {
    return = choose;
}

sun.awt.windows.WFramePeer.pSetIMMOption(THIS, STR) {
}

sun.awt.windows.WFramePeer.reshape(THIS, X, Y, W, H) {
   sun.awt.windows.WComponentPeer.reshape(THIS, X, Y, W,
H);
}

sun.awt.windows.WFramePeer.setIconImageFromIntRasterData(
THIS, BITS, DATAWIDTH, PIXHEIGHT, PIXWIDTH) {
}

sun.awt.windows.WFramePeer.setMenuBar0(THIS, MENUBAR) {
}

sun.awt.windows.WFramePeer.setState(THIS, STATE) {
    THIS sun.awt.windows.WFramePeer#state := STATE;
}

/* sun.awt.windows.WDialogPeer */

sun.awt.windows.WDialogPeer.create(THIS, PARENT) {
    PDATA = choose;
    THIS sun.awt.windows.WObjectPeer.pData := PDATA;
}

sun.awt.windows.WDialogPeer.showModal(THIS) {
}

sun.awt.windows.WDialogPeer.endModal(THIS) {
}

sun.awt.windows.WDialogPeer.pSetIMMOption(THIS, STR) {
}

/* sun.awt.windows.WFileDialogPeer */

sun.awt.windows.WFileDialogPeer.initIDs() {
}

sun.awt.windows.WFileDialogPeer.show(THIS) {
}

sun.awt.windows.WFileDialogPeer.targetSetDirectory_NoClie
ntCode(THIS, DIALOG, STR) {
    DIALOG java.awt.FileDialog.file := STR;
}

sun.awt.windows.WFileDialogPeer.targetSetFile_NoClientCod
e(THIS, DIALOG, STR) {
    DIALOG java.awt.FileDialog.dir := STR;
}

/* sun.awt.windows.WCanvasPeer */

sun.awt.windows.WChoicePeer.create(THIS, PARENT) {
    PDATA = choose;
    THIS sun.awt.windows.WObjectPeer.pData := PDATA;
}

sun.awt.windows.WChoicePeer.addItem(THIS, STR, INDEX) {
}

sun.awt.windows.WChoicePeer.remove(THIS, INDEX) {
}

sun.awt.windows.WChoicePeer.select(THIS, INDEX) {
}

sun.awt.windows.WChoicePeer.reshape(THIS, X, Y, W, H) {
   sun.awt.windows.WComponentPeer.reshape(THIS, X, Y, W,
H);
}

/* sun.awt.windows.WCanvasPeer */
```

```
sun.awt.windows.WCanvasPeer.create(THIS, PARENT) {
    PDATA = choose;
    THIS sun.awt.windows.WObjectPeer.pData := PDATA;
}

/* sun.awt.windows.WMenuItemPeer */

sun.awt.windows.WMenuItemPeer.create(THIS, MENU) {
    PDATA = choose;
    THIS sun.awt.windows.WObjectPeer.pData := PDATA;
    sun.awt.windows.WMenuItemPeer#menuItemPeers := THIS;
}

sun.awt.windows.WMenuItemPeer._dispose(THIS) {
}

sun.awt.windows.WMenuItemPeer._setLabel(THIS, STR) {
}

sun.awt.windows.WMenuItemPeer.enable(THIS, BOOL) {
}

sun.awt.windows.WMenuItemPeer.initIDs() {
}

/* sun.awt.windows.WMenuPeer */

sun.awt.windows.WMenuPeer.createMenu(THIS, MENUBAR) {
    PDATA = choose;
    THIS sun.awt.windows.WObjectPeer.pData := PDATA;
}

sun.awt.windows.WMenuPeer.createSubMenu(THIS, MENU) {
    PDATA = choose;
    THIS sun.awt.windows.WObjectPeer.pData := PDATA;
}

sun.awt.windows.WMenuPeer.addSeparator(THIS) {
}

sun.awt.windows.WMenuPeer.delItem(THIS, INDEX) {
}

/* sun.awt.windows.WMenuBarPeer */

sun.awt.windows.WMenuBarPeer.create(THIS, FRAME) {
    PDATA = choose;
    THIS sun.awt.windows.WObjectPeer.pData := PDATA;
}

sun.awt.windows.WMenuBarPeer.addMenu(THIS, MENU) {
}

sun.awt.windows.WMenuBarPeer.delMenu(THIS, INDEX) {
}

/* sun.awt.windows.WCheckboxMenuItemPeer */

sun.awt.windows.WCheckboxMenuItemPeer.setState(THIS,
BOOL) {
}

/* sun.awt.windows.WTextComponentPeer */

sun.awt.windows.WTextComponentPeer.enableEditing(THIS,
BOOL) {
}

sun.awt.windows.WTextComponentPeer.getSelectionStart(THIS
) {
    return = THIS
sun.awt.windows.WTextComponentPeer#selectfrom;
}

sun.awt.windows.WTextComponentPeer.getSelectionEnd(THIS)
{
    return = THIS
sun.awt.windows.WTextComponentPeer#selectto;
}

sun.awt.windows.WTextComponentPeer.select(THIS, FROM, TO)
{
    THIS sun.awt.windows.WTextComponentPeer#selectfrom :=
FROM;
    THIS sun.awt.windows.WTextComponentPeer#selectto :=
TO;
}

sun.awt.windows.WTextComponentPeer.getText(THIS) {
    return = THIS
sun.awt.windows.WTextComponentPeer#text;
}

sun.awt.windows.WTextComponentPeer.setText(THIS, STR) {
    THIS sun.awt.windows.WTextComponentPeer#text := STR;
}

sun.awt.windows.WTextComponentPeer.initIDs() {
}

/* sun.awt.windows.WTextAreaPeer */

sun.awt.windows.WTextAreaPeer.create(THIS, PARENT) {
    PDATA = choose;
    THIS sun.awt.windows.WObjectPeer.pData := PDATA;
}

sun.awt.windows.WTextAreaPeer.insertText(THIS, STR, POS)
{
    TEXT = THIS sun.awt.windows.WTextComponentPeer#text;
    NEWTEXT = mungeStrings(TEXT, STR);
    THIS sun.awt.windows.WTextComponentPeer#text :=
NEWTEXT;
}

sun.awt.windows.WTextAreaPeer.replaceText(THIS, STR,
FROM, TO) {
    TEXT = THIS sun.awt.windows.WTextComponentPeer#text;
    NEWTEXT = mungeStrings(TEXT, STR);
    THIS sun.awt.windows.WTextComponentPeer#text :=
NEWTEXT;
}

/* sun.awt.windows.WTextFieldPeer */

sun.awt.windows.WTextFieldPeer.create(THIS, PARENT) {
    PDATA = choose;
    THIS sun.awt.windows.WObjectPeer.pData := PDATA;
}

sun.awt.windows.WTextFieldPeer.setEchoCharacter(THIS, CH)
{
}

/* sun.awt.windows.WLabelPeer */

sun.awt.windows.WLabelPeer.create(THIS, PARENT) {
    PDATA = choose;
    THIS sun.awt.windows.WObjectPeer.pData := PDATA;
}

sun.awt.windows.WLabelPeer.setAlignment(THIS, ALIGN) {
}

sun.awt.windows.WLabelPeer.setText(THIS, STR) {
}

/* sun.awt.windows.WCheckboxPeer */

sun.awt.windows.WCheckboxPeer.create(THIS, PARENT) {
    PDATA = choose;
    THIS sun.awt.windows.WObjectPeer.pData := PDATA;
}

sun.awt.windows.WCheckboxPeer.setCheckboxGroup(THIS,
GROUP) {
}

sun.awt.windows.WCheckboxPeer.setLabel(THIS, STR) {
}

sun.awt.windows.WCheckboxPeer.setState(THIS, BOOL) {
}

/* sun.awt.windows.WButtonPeer */

sun.awt.windows.WButtonPeer.create(THIS, PARENT) {
    PDATA = choose;
    THIS sun.awt.windows.WObjectPeer.pData := PDATA;
}

sun.awt.windows.WButtonPeer.initIDs() {
}

sun.awt.windows.WButtonPeer.setLabel(THIS, STR) {
}

/* sun.awt.windows.WListPeer */

sun.awt.windows.WListPeer.create(THIS, PARENT) {
    PDATA = choose;
    THIS sun.awt.windows.WObjectPeer.pData := PDATA;
    MAXWIDTH = choose;
    THIS sun.awt.windows.WListPeer#maxwidth := MAXWIDTH;
}

sun.awt.windows.WListPeer._addItem(THIS, STR, INDEX,
WIDTH) {
    goto Y, N;

Y:  THIS sun.awt.windows.WListPeer#maxwidth := WIDTH;

N:  choose;
}

sun.awt.windows.WListPeer.addItem0(THIS, STR, INDEX,
WIDTH) {
    goto Y, N;

Y:  THIS sun.awt.windows.WListPeer#maxwidth := WIDTH;

N:  choose;
}

sun.awt.windows.WListPeer.delItems(THIS, FROM, TO) {
}

sun.awt.windows.WListPeer.setMultipleSelections(THIS,
BOOL) {
}

sun.awt.windows.WListPeer.select(THIS, INDEX) {
}

sun.awt.windows.WListPeer.deselect(THIS, INDEX) {
}

sun.awt.windows.WListPeer.isSelected(THIS, INDEX) {
    return = choose;
}

sun.awt.windows.WListPeer.makeVisible(THIS, INDEX) {
}

sun.awt.windows.WListPeer.updateMaxItemWidth(THIS) {
}

sun.awt.windows.WListPeer.getMaxWidth(THIS, WIDTH) {
    return = THIS sun.awt.windows.WListPeer#maxwidth;
}

/* sun.awt.windows.WClipboard */

sun.awt.windows.WClipboard.getClipboardText(THIS) {
    goto N, R;

N:  STR = makeString();
    THIS sun.awt.windows.WClipboard#text := STR;

R:  return = THIS sun.awt.windows.WClipboard#text;
}

sun.awt.windows.WClipboard.init() {
}

sun.awt.windows.WClipboard.setClipboardObject(THIS, OBJ)
{
    sun.awt.windows.WToolkit#theClipboard := THIS;
    THIS sun.awt.windows.WClipboard#text := OBJ;
}

sun.awt.windows.WClipboard.setClipboardText(THIS, STRSEL)
{
    sun.awt.windows.WToolkit#theClipboard := THIS;
    DATA = STRSEL
java.awt.datatransfer.StringSelection.data;
    THIS sun.awt.windows.WClipboard#text := DATA;
}

/* sun.awt.windows.WColor */

sun.awt.windows.WColor.getDefaultColor(INDEX) {
    return = choose;
}

/* sun.awt.windows.WFontMetrics */

sun.awt.windows.WFontMetrics.initIDs() {
}

sun.awt.windows.WFontMetrics.init(THIS) {
    INTS = makeIntArray();
    THIS sun.awt.windows.WFontMetrics.widths := INTS;
    V = choose;
    THIS sun.awt.windows.WFontMetrics.ascent := V;
    V = choose;
    THIS sun.awt.windows.WFontMetrics.descent := V;
    V = choose;
    THIS sun.awt.windows.WFontMetrics.leading := V;
    V = choose;
    THIS sun.awt.windows.WFontMetrics.height := V;
    V = choose;
    THIS sun.awt.windows.WFontMetrics.maxAscent := V;
    V = choose;
    THIS sun.awt.windows.WFontMetrics.maxDescent := V;
    V = choose;
    THIS sun.awt.windows.WFontMetrics.maxHeight := V;
    V = choose;
    THIS sun.awt.windows.WFontMetrics.maxAdvance := V;
}

sun.awt.windows.WFontMetrics.bytesWidth(THIS, BYTES,
INDEX, LEN) {
    return = choose;
}

sun.awt.windows.WFontMetrics.charsWidth(THIS, CHARS,
INDEX, LEN) {
    return = choose;
}

sun.awt.windows.WFontMetrics.stringWidth(THIS, STR) {
    return = choose;
}

sun.awt.windows.WFontMetrics.needsConversion(FONT,
FONTDESC) {
    return = choose;
}

sun.awt.windows.WFontMetrics.getMFCharSegmentWidth(THIS,
FONT, FONTDESC, BOOL, CHARS, FROM, TO, SEGS, LEN) {
    return = choose;
}

/* sun.awt.windows.WDefaultFontCharset */

sun.awt.windows.WDefaultFontCharset.initIDs() {
}

sun.awt.windows.WDefaultFontCharset.canConvert(THIS, CH)
{
    return = choose;
}

/* sun.awt.windows.WPrintJob */

sun.awt.windows.WPrintJob.initIDs() {
}

sun.awt.windows.WPrintJob.newPage(THIS) {
}

sun.awt.windows.WPrintJob.flushPageImpl(THIS) {
}

sun.awt.windows.WPrintJob.endImpl(THIS) {
}

/* sun.awt.windows.WGraphics */

sun.awt.windows.WGraphics.initIDs() {
}

sun.awt.windows.WGraphics.checkNoDDraw() {
    return = choose;
}

sun.awt.windows.WGraphics.createFromComponent(THIS, COMP)
{
    PDATA = choose;
    THIS sun.awt.windows.WGraphics.pData := PDATA;
}

sun.awt.windows.WGraphics.createFromGraphics(THIS, G) {
    PDATA = choose;
    THIS sun.awt.windows.WGraphics.pData := PDATA;
}

sun.awt.windows.WGraphics.createFromHDC(THIS, HDC) {
    PDATA = choose;
    THIS sun.awt.windows.WGraphics.pData := PDATA;
}

sun.awt.windows.WGraphics.createFromPrintJob(THIS, JOB) {
    PDATA = choose;
    THIS sun.awt.windows.WGraphics.pData := PDATA;
}
```

```
sun.awt.windows.WGraphics.disposeImpl(THIS) {
}

sun.awt.windows.WGraphics.W32LockViewResources(THIS,
DATA, VIEWX, VIEWY, VIEWW, VIEWH, LOCKMETHOD) {
    return = choose;
}

sun.awt.windows.WGraphics.W32UnLockViewResources(THIS,
DATA) {
    return = choose;
}

sun.awt.windows.WGraphics.getClipBounds(THIS) {
    X = choose;
    Y = choose;
    W = choose;
    H = choose;
    RECT = new java.awt.Rectangle;
    java.awt.Rectangle.<init>(RECT, X, Y, W, H);
    return = choose RECT;
}

sun.awt.windows.WGraphics.changeClip(THIS, X, Y, W, H,
BOOL) {
}

sun.awt.windows.WGraphics.removeClip(THIS) {
}

sun.awt.windows.WGraphics.clearRect(THIS, X, Y, W, H) {
}

sun.awt.windows.WGraphics.drawRect(THIS, X, Y, W, H) {
}

sun.awt.windows.WGraphics.fillRect(THIS, X, Y, W, H) {
}

sun.awt.windows.WGraphics.drawLine(THIS, X, Y, X2, Y2) {
}

sun.awt.windows.WGraphics.copyArea(THIS, X, Y, W, H, DX,
DY) {
}

sun.awt.windows.WGraphics.drawArc(THIS, X, Y, W, H, FROM,
TO) {
}

sun.awt.windows.WGraphics.fillArc(THIS, X, Y, W, H, FROM,
TO) {
}

sun.awt.windows.WGraphics.drawOval(THIS, X, Y, W, H) {
}

sun.awt.windows.WGraphics.fillOval(THIS, X, Y, W, H) {
}

sun.awt.windows.WGraphics.drawPolygon(THIS, XS, YS,
COUNT) {
}

sun.awt.windows.WGraphics.fillPolygon(THIS, XS, YS,
COUNT) {
}

sun.awt.windows.WGraphics.drawPolyline(THIS, XS, YS,
COUNT) {
}

sun.awt.windows.WGraphics.drawRoundRect(THIS, X, Y, W, H,
RX, RY) {
}

sun.awt.windows.WGraphics.fillRoundRect(THIS, X, Y, W, H,
RX, RY) {
}

sun.awt.windows.WGraphics.print(THIS, COMPONENT) {
}

sun.awt.windows.WGraphics.devClearRect(THIS, X, Y, W, H)
{
}

sun.awt.windows.WGraphics.devCopyArea(THIS, X, Y, W, H,
DX, DY) {
}

sun.awt.windows.WGraphics.devDrawArc(THIS, X, Y, W, H,
FROM, TO) {
}

sun.awt.windows.WGraphics.devFillArc(THIS, X, Y, W, H,
FROM, TO) {
}

sun.awt.windows.WGraphics.devDrawLine(THIS, X, Y, X2, Y2)
{
}

sun.awt.windows.WGraphics.devDrawOval(THIS, X, Y, W, H) {
}

sun.awt.windows.WGraphics.devFillOval(THIS, X, Y, W, H) {
}

sun.awt.windows.WGraphics.devDrawPolygon(THIS, XS, YS,
COUNT) {
}

sun.awt.windows.WGraphics.devFillPolygon(THIS, XS, YS,
COUNT) {
}

sun.awt.windows.WGraphics.devDrawPolyline(THIS, XS, YS,
COUNT) {
}

sun.awt.windows.WGraphics.devDrawRect(THIS, X, Y, W, H) {
}

sun.awt.windows.WGraphics.devFillRect(THIS, X, Y, W, H) {
}

sun.awt.windows.WGraphics.devDrawRoundRect(THIS, X, Y, W,
H, RX, RY) {
}

sun.awt.windows.WGraphics.devFillRoundRect(THIS, X, Y, W,
H, RX, RY) {
}

sun.awt.windows.WGraphics.devFillSpans(THIS, ITERATOR,
LONG) {
}

sun.awt.windows.WGraphics.devPrint(THIS, COMPONENT) {
}

sun.awt.windows.WGraphics.drawSFChars(THIS, CHARS, FROM,
TO, X, Y) {
}

sun.awt.windows.WGraphics.drawMFCharsSegment(THIS, FONT,
FONTDESC, CHARS, FROM, TO, X, Y) {
    return = choose;
}

sun.awt.windows.WGraphics.drawMFCharsConvertedSegment(THI
S, FONT, FONTDESC, BYTES, LEN, X, Y) {
    return = choose;
}

sun.awt.windows.WGraphics.drawBytes(THIS, BYTES, FROM,
TO, X, Y) {
    return = choose;
}

sun.awt.windows.WGraphics.drawBytesWidth(THIS, BYTES,
FROM, TO, X, Y) {
    return = choose;
}

sun.awt.windows.WGraphics.drawCharsWidth(THIS, CHARS,
FROM, TO, X, Y) {
    return = choose;
}

sun.awt.windows.WGraphics.drawStringWidth(THIS, STR, X,
Y) {
    return = choose;
}

sun.awt.windows.WGraphics.pSetFont(THIS, FONT) {
}

sun.awt.windows.WGraphics.pSetForeground(THIS, COLOR) {
}

sun.awt.windows.WGraphics.setPaintMode(THIS) {
}

sun.awt.windows.WGraphics.pSetPaintMode(THIS) {
}

sun.awt.windows.WGraphics.setXORMode(THIS, COLOR) {
}

sun.awt.windows.WGraphics.pSetXORMode(THIS, COLOR) {
}

sun.awt.windows.WGraphics.setOrigin(THIS, X, Y) {
}

sun.awt.windows.WGraphics.imageCreate(THIS, IMAGE) {
}

/* sun.awt.image.ImageRepresentation */

sun.awt.image.ImageRepresentation.offscreenInit(THIS,
COLOR) {
}

sun.awt.image.ImageRepresentation.disposeImage(THIS) {
}

convertPixel(CM, DATA) {
    PIXEL = DATA java.lang.Object#intarrayelement;
    java.awt.image.ColorModel.getAlpha(CM, PIXEL);
    java.awt.image.ColorModel.getRed(CM, PIXEL);
    java.awt.image.ColorModel.getGreen(CM, PIXEL);
    java.awt.image.ColorModel.getBlue(CM, PIXEL);
}

sun.awt.image.ImageRepresentation.setBytePixels(THIS, X,
Y, W, H, CM, BYTES, OFF, LEN) {
    convertPixel(CM, BYTES);
}

sun.awt.image.ImageRepresentation.setIntPixels(THIS, X,
Y, W, H, CM, INTS, OFF, LEN) {
    convertPixel(CM, INTS);
}

sun.awt.image.ImageRepresentation.finish(THIS, BOOL) {
}

sun.awt.image.ImageRepresentation.imageDraw(THIS, G, X,
Y, COLOR) {
}

sun.awt.image.ImageRepresentation.imageStretch(THIS, G,
X, Y, W, H, FROMX, FROMY, FROMW, FROMH, COLOR) {
}

/* sun.awt.image.OffScreenImageSource */

sun.awt.image.OffScreenImageSource.sendPixels(THIS) {
    CONSUMER = THIS
sun.awt.image.OffScreenImageSource.theConsumer;

L:  X = choose;
    Y = choose;
    W = choose;
    H = choose;
    CM = sun.awt.windows.WToolkit.makeColorModel();
    BYTES = makeByteArray();
    OFF = choose;
    LEN = choose;
    java.awt.image.ImageConsumer.setPixels(CONSUMER, X,
Y, W, H, CM, BYTES, OFF, LEN)
"(IIIILjava.awt.image.ColorModel;[BII)V";
    goto L, EX;

EX: choose;
}

/* sun.awt.image.JPEGImageDecoder */

sun.awt.image.JPEGImageDecoder.readImage(THIS, STREAM,
BYTES) {
N:  INPUT = makeByteArray();
    OFF = choose;
    LEN = choose;
    BYTE = java.io.InputStream.read(STREAM, BYTES, OFF,
LEN);
    EXN1 = catch (java.lang.Throwable) BYTE;

    DATA = choose;
    BYTES java.lang.Object#intarrayelement := DATA;

    goto N, EX;

EX: STR = _stringconst();
    ERR = sun.awt.image.JPEGImageDecoder.error(STR);
    EXN2 = catch (java.lang.Throwable) ERR;

    throw = choose EXN1, EXN2;
}

/* sun.awt.image.GifImageDecoder */

sun.awt.image.GifImageDecoder.parseImage(THIS, X, Y, W,
H, BOOL, FLAGS, HEADER, OUTPUT, CM) {
N:  INPUT = makeByteArray();
    OFF = choose;
    LEN = choose;
    RESULT =
sun.awt.image.GifImageDecoder.readBytes(THIS, INPUT, OFF,
LEN);

    DATA = choose;
    OUTPUT java.lang.Object#intarrayelement := DATA;

    goto N, EX;

EX: return = choose;
}
```

Appendix C: Ajax Reflection Specifications

Here I provide the complete text of the reflection specifications used by Ajax. They cover the examples I used for this thesis.

```
java.lang.Class.newInstance [
    ajax.analyzer.test.ReflectionTest.main {
        class=ajax.analyzer.test.ReflectionTest
    }
    sun.io.CharToByteConverter.getDefault {
        class=sun.io.CharToByteCp1252
           # sun.io.CharToByte*
    }
    sun.io.ByteToCharConverter.getDefault {
        class=sun.io.ByteToCharCp1252
           # sun.io.ByteToChar*
    }
    sun.io.ByteToCharConverter.getConverter {
        class=sun.io.ByteToCharCp1252
           # sun.io.ByteToChar*
    }
    java.net.URL.getURLStreamHandler {
        class=*.Handler
    }
    java.net.InetAddress.<clinit> {
        class=java.net.*InetAddressImpl
    }
    java.security.Security.getImpl {
        class=sun.security.provider.*
    }
    java.security.Provider.loadProvider {
        class=sun.security.provider.Sun
    }
    java.util.ResourceBundle.findBundle {
        class=java.text.resources.DateFormatZoneData
          # java.text.resources.DateFormatZoneData*
        class=java.text.resources.DateFormatZoneData_en
        class=java.text.resources.LocaleElements
          # java.text.resources.LocaleElements*
        class=java.text.resources.LocaleElements_en
    }
    sun.security.x509.AlgorithmId.buildAlgorithmId {

class=sun.security.*<sun.security.x509.AlgorithmId
    }
    sun.security.x509.X509Key.buildX509Key {
        class=sun.security.x509.X509Key
    }
    java.awt.Toolkit.getDefaultToolkit {
        class=sun.awt.windows.WToolkit
    }
    ladybug.engine.FormulaSolver.createSolver {
        class=ladybug.selenum.createSolver
    }
    sun.awt.SunToolkit.<init> {
        class=java.awt.EventQueue
    }
    sun.awt.windows.WFontPeer.getFontCharset {
        class=sun.io.CharToByteCp1252
    }
    sun.awt.windows.WFontMetrics.getMFStringWidth {
        class=sun.io.CharToByteCp1252
    }
    sun.awt.windows.WGraphics.drawMFChars {
        class=sun.io.CharToByteCp1252
    }
    ladybug.parse.Formula.createPeer {
    }
    ladybug.parse.Term.createPeer {
    }
    jess.Main.main {
        class=jess.StringFunctions
        class=jess.PredFunctions
        class=jess.MultiFunctions
        class=jess.MiscFunctions
        class=jess.MathFunctions
        class=jess.BagFunctions
        class=jess.reflect.ReflectFunctions
        class=jess.view.ViewFunctions
    }
    jess.Funcall.loadIntrinsics {
        class=jess.Assert
        class=jess.Retract
        class=jess.RetractString
        class=jess.Printout
        class=jess.ExtractGlobal
        class=jess.Open
        class=jess.Close
        class=jess.Foreach
        class=jess.Read
        class=jess.Readline
        class=jess.GensymStar
        class=jess.While
        class=jess.If
        class=jess.Bind
        class=jess.Modify
        class=jess.And
        class=jess.Not
        class=jess.Or
        class=jess.Eq
        class=jess.EqStar
        class=jess.Equals
        class=jess.NotEquals
        class=jess.Gt
        class=jess.Lt
        class=jess.GtOrEq
        class=jess.LtOrEq
        class=jess.Neq
        class=jess.Mod
        class=jess.Plus
        class=jess.Times
        class=jess.Minus
        class=jess.Divide
        class=jess.SymCat
        class=jess.LoadFacts
        class=jess.SaveFacts
        class=jess.AssertString
        class=jess.UnDefrule
        class=jess.Try
    }
    jess.LoadPkg.call {
        class=jess.*<jess.Userpackage
    }
    jess.LoadFn.call {
        class=jess.*<jess.Userfunction
    }
    jess.SetStrategy.call {
        class=jess.*<jess.Strategy
    }
    "jess.NodeTest.addTest(int,int,int,jess.Value)" {
        class=jess.*<jess.Test
    }
    jess.Rete.<init> {
        class=jess.depth
    }
]

java.lang.Class.forName [
    ajax.analyzer.test.ReflectionTest.main {
        class=ajax.analyzer.test.ReflectionTest
    }

ajax.tools.benchmarks.GeneralBenchmark.makePrintSinkStrea
m {
        class=java.io.OutputStream
        class=java.io.PrintStream
    }
    sun.io.CharToByteConverter.getConverterClass {
        class=sun.io.CharToByteCp1252     #
sun.io.CharToByte*
    }
    sun.io.ByteToCharConverter.getConverterClass {
        class=sun.io.ByteToCharCp1252     #
sun.io.ByteToChar*
    }
    java.io.ObjectStreamClass.<clinit> {
        class=java.io.Serializable
        class=java.io.Externalizable
    }
    java.net.URL.getURLStreamHandler {
        class=*.Handler
    }
    java.net.InetAddress.<clinit> {
        class=java.net.*InetAddressImpl
    }
    java.security.Security.getImpl {
        class=sun.security.provider.*
    }
    java.security.Provider.loadProvider {
        class=sun.security.provider.Sun
    }
    sun.security.x509.AlgorithmId.buildAlgorithmId {

class=sun.security.*<sun.security.x509.AlgorithmId
    }
    sun.security.x509.X509Key.buildX509Key {
        class=sun.security.x509.X509Key
    }
    java.awt.Toolkit.getDefaultToolkit {
        class=sun.awt.windows.WToolkit
    }
    ladybug.engine.SchemaSolver.solverClasses {
        class=ladybug.selenum.SelEnumSolver
    }
    sun.awt.SunToolkit.<init> {
        class=java.awt.EventQueue
    }
    sun.awt.windows.WFontPeer.getFontCharset {
        class=sun.io.CharToByteCp1252
    }
    javafig.objects.FigAttribs.<clinit> {
        class=java.awt.geom.AffineTransform
    }
    jess.Main.main {
        class=jess.StringFunctions
        class=jess.PredFunctions
        class=jess.MultiFunctions
        class=jess.MiscFunctions
        class=jess.MathFunctions
        class=jess.BagFunctions
        class=jess.reflect.ReflectFunctions
        class=jess.view.ViewFunctions
    }
    jess.Funcall.loadIntrinsics {
        class=jess.Assert
        class=jess.Retract
        class=jess.RetractString
        class=jess.Printout
        class=jess.ExtractGlobal
        class=jess.Open
        class=jess.Close
        class=jess.Foreach
        class=jess.Read
        class=jess.Readline
        class=jess.GensymStar
        class=jess.While
        class=jess.If
        class=jess.Bind
        class=jess.Modify
        class=jess.And
        class=jess.Not
        class=jess.Or
        class=jess.Eq
        class=jess.EqStar
        class=jess.Equals
        class=jess.NotEquals
        class=jess.Gt
        class=jess.Lt
        class=jess.GtOrEq
        class=jess.LtOrEq
        class=jess.Neq
        class=jess.Mod
        class=jess.Plus
        class=jess.Times
        class=jess.Minus
        class=jess.Divide
        class=jess.SymCat
        class=jess.LoadFacts
        class=jess.SaveFacts
        class=jess.AssertString
        class=jess.UnDefrule
        class=jess.Try
    }
    jess.LoadPkg.call {
        class=jess.*<jess.Userpackage
    }
    jess.LoadFn.call {
        class=jess.*<jess.Userfunction
    }
    jess.SetStrategy.call {
        class=jess.*<jess.Strategy
    }
    "jess.NodeTest.addTest(int,int,int,jess.Value)" {
        class=jess.*<jess.Test
    }
    jess.Rete.<init> {
        class=jess.depth
    }
]

java.lang.Class.getConstructor [
    javafig.gui.ModularEditor.handleCommandCallback {
    }

ajax.tools.benchmarks.GeneralBenchmark.makePrintSinkStrea
m {
    }
]

java.lang.reflect.Constructor.newInstance [
    javafig.gui.ModularEditor.handleCommandCallback {
        class=javafig.commands.*
    }

ajax.tools.benchmarks.GeneralBenchmark.makePrintSinkStrea
m {
        class=java.io.PrintStream
    }
]

java.lang.Class.getMethod [
    ajax.analyzer.test.ReflectionTest.main {
        method=ajax.analyzer.test.ReflectionTest.*
    }
    ajax.analyzer.test.ReflectionTest.hello {
        method=ajax.analyzer.test.ReflectionTest.*
    }
    javafig.gui.ModularEditor.call {
        method=javafig.gui.ModularEditor.doCancel
        method=javafig.gui.ModularEditor.doUndo
        method=javafig.gui.ModularEditor.doRedo
        method=javafig.gui.ModularEditor.doFlushUndoStack
        method=javafig.gui.ModularEditor.doDeleteAll

method=javafig.gui.ModularEditor.doCopyToClipboard
        method=javafig.gui.ModularEditor.doCutToClipboard

method=javafig.gui.ModularEditor.doPasteFromClipboard
        method=javafig.gui.ModularEditor.doCreateCircle
        method=javafig.gui.ModularEditor.doCreateEllipse

method=javafig.gui.ModularEditor.doCreateRectangle

method=javafig.gui.ModularEditor.doCreateRoundRectangle
        method=javafig.gui.ModularEditor.doCreatePolyline
        method=javafig.gui.ModularEditor.doCreatePolygon
        method=javafig.gui.ModularEditor.doCreateSpline

method=javafig.gui.ModularEditor.doCreateClosedSpline
        method=javafig.gui.ModularEditor.doCreateBezier

method=javafig.gui.ModularEditor.doCreateClosedBezier
        method=javafig.gui.ModularEditor.doCreateArc
        method=javafig.gui.ModularEditor.doCreateImage
        method=javafig.gui.ModularEditor.doCreateText
        method=javafig.gui.ModularEditor.doCreateLink
        method=javafig.gui.ModularEditor.doCreateCompound
        method=javafig.gui.ModularEditor.doBreakCompound
        method=javafig.gui.ModularEditor.doMoveObject
        method=javafig.gui.ModularEditor.doCopyObject
        method=javafig.gui.ModularEditor.doDeleteObject
        method=javafig.gui.ModularEditor.doMovePoint
        method=javafig.gui.ModularEditor.doInsertPoint
        method=javafig.gui.ModularEditor.doCutPoint
        method=javafig.gui.ModularEditor.doMirrorXObject
        method=javafig.gui.ModularEditor.doMirrorYObject
        method=javafig.gui.ModularEditor.doScaleObject
        method=javafig.gui.ModularEditor.doAlignObjects

method=javafig.gui.ModularEditor.doSnapObjectToGrid
        method=javafig.gui.ModularEditor.doConvertObject
        method=javafig.gui.ModularEditor.doResizeText
        method=javafig.gui.ModularEditor.doUpdate
        method=javafig.gui.ModularEditor.doCancelUpdate
        method=javafig.gui.ModularEditor.enableUpdateAll
        method=javafig.gui.ModularEditor.enableUpdateNone

method=javafig.gui.ModularEditor.enableUpdateInvert
        method=javafig.gui.ModularEditor.doEditObject

method=javafig.gui.ModularEditor.doEditGlobalAttributes
        method=javafig.gui.ModularEditor.doZoomRegion
        method=javafig.gui.ModularEditor.doZoomIn
        method=javafig.gui.ModularEditor.doZoomOut
        method=javafig.gui.ModularEditor.doZoom11
        method=javafig.gui.ModularEditor.doPanHome
        method=javafig.gui.ModularEditor.doPanLeft
        method=javafig.gui.ModularEditor.doPanRight
        method=javafig.gui.ModularEditor.doPanUp
        method=javafig.gui.ModularEditor.doPanDown
        method=javafig.gui.ModularEditor.doSetGridNone
        method=javafig.gui.ModularEditor.doSetGridCoarse
        method=javafig.gui.ModularEditor.doSetGridMedium
        method=javafig.gui.ModularEditor.doSetGridFine
        method=javafig.gui.ModularEditor.doSetNoSnap
        method=javafig.gui.ModularEditor.doSetSnap12
        method=javafig.gui.ModularEditor.doSetSnap14
        method=javafig.gui.ModularEditor.doSetSnap18
        method=javafig.gui.ModularEditor.doSetUnitsInches

method=javafig.gui.ModularEditor.doSetUnitsMillimeter

method=javafig.gui.ModularEditor.doSetUnitsXfigMillimeter

method=javafig.gui.ModularEditor.doSnapAllObjectsToGrid

method=javafig.gui.ModularEditor.doClearUserColors

method=javafig.gui.ModularEditor.doWriteHadesResource
        method=javafig.gui.ModularEditor.doRedraw

method=javafig.gui.ModularEditor.doStartNewDrawing
        method=javafig.gui.ModularEditor.doSelectFile
        method=javafig.gui.ModularEditor.doMergeFile
        method=javafig.gui.ModularEditor.doSelectURL
        method=javafig.gui.ModularEditor.doMergeURL

method=javafig.gui.ModularEditor.handleParserCallback

method=javafig.gui.ModularEditor.handleParserMergeCallbac
k

method=javafig.gui.ModularEditor.handleCommandCallback
        method=javafig.gui.ModularEditor.doQuit
        method=javafig.gui.ModularEditor.doSaveFile
        method=javafig.gui.ModularEditor.doSaveFileAs
        method=javafig.gui.ModularEditor.doSaveToConsole
        method=javafig.gui.ModularEditor.doPrintViaAWT
        method=javafig.gui.ModularEditor.doPrintUndoStack
        method=javafig.gui.ModularEditor.doPrintClipboard
        method=javafig.gui.ModularEditor.doPrintObjects
        method=javafig.gui.ModularEditor.doShowMessages

method=javafig.gui.ModularEditor.doShowAboutDialog

method=javafig.gui.ModularEditor.doShowLicenseDialog

method=javafig.gui.ModularEditor.doShowDeadlockDialog

method=javafig.gui.ModularEditor.doShowChangesDialog

method=javafig.gui.ModularEditor.doShowMouseButtonDialog

method=javafig.gui.ModularEditor.doShowShortcutKeysDialog
        method=javafig.gui.ModularEditor.doShowFaqDialog
        method=javafig.gui.ModularEditor.doShowHelpDialog
        method=javafig.gui.ModularEditor.doShowDemoGold
        method=javafig.gui.ModularEditor.doShowDemoHouse
        method=javafig.gui.ModularEditor.doShowDemoWatch

method=javafig.gui.ModularEditor.doShowDemoCircuit
        method=javafig.gui.ModularEditor.doShowDemoLayout

method=javafig.gui.ModularEditor.doShowDemoPictures

method=javafig.gui.ModularEditor.doShowDemoRotated

method=javafig.gui.ModularEditor.doShowDemoUnicode

method=javafig.gui.ModularEditor.doShowDemoWelcome
    }
    javafig.gui.EditTextDialog.getStatusMessage {
        method=javafig.*.getStatusMessage
    }
    javafig.gui.EditPolylineDialog.getStatusMessage {
        method=javafig.*.getStatusMessage
    }
    javafig.gui.EditEllipseDialog.getStatusMessage {
        method=javafig.*.getStatusMessage
    }
    javafig.gui.EditTriggerDialog.getStatusMessage {
        method=javafig.*.getStatusMessage
    }
    javafig.gui.EditImageDialog.getStatusMessage {
        method=javafig.*.getStatusMessage
    }
    javafig.gui.EditRectangleDialog.getStatusMessage {
        method=javafig.*.getStatusMessage
    }

javafig.gui.EditGlobalAttributesDialog.getStatusMessage {
        method=javafig.*.getStatusMessage
    }
    javafig.commands.ZoomRegionCommand.execute {
        method=javafig.*.doZoomRegion
    }
]

java.lang.reflect.Method.invoke [
    ajax.analyzer.test.ReflectionTest.main {
        method=ajax.analyzer.test.ReflectionTest.*
    }
    ajax.analyzer.test.ReflectionTest.hello {
        method=ajax.analyzer.test.ReflectionTest.*
    }
    javafig.gui.ModularEditor.call {
        method=javafig.gui.ModularEditor.doCancel
        method=javafig.gui.ModularEditor.doUndo
        method=javafig.gui.ModularEditor.doRedo
        method=javafig.gui.ModularEditor.doFlushUndoStack
        method=javafig.gui.ModularEditor.doDeleteAll

method=javafig.gui.ModularEditor.doCopyToClipboard
        method=javafig.gui.ModularEditor.doCutToClipboard

method=javafig.gui.ModularEditor.doPasteFromClipboard
        method=javafig.gui.ModularEditor.doCreateCircle
        method=javafig.gui.ModularEditor.doCreateEllipse

method=javafig.gui.ModularEditor.doCreateRectangle

method=javafig.gui.ModularEditor.doCreateRoundRectangle
        method=javafig.gui.ModularEditor.doCreatePolyline
        method=javafig.gui.ModularEditor.doCreatePolygon
        method=javafig.gui.ModularEditor.doCreateSpline

method=javafig.gui.ModularEditor.doCreateClosedSpline
        method=javafig.gui.ModularEditor.doCreateBezier

method=javafig.gui.ModularEditor.doCreateClosedBezier
        method=javafig.gui.ModularEditor.doCreateArc
        method=javafig.gui.ModularEditor.doCreateImage
        method=javafig.gui.ModularEditor.doCreateText
        method=javafig.gui.ModularEditor.doCreateLink
        method=javafig.gui.ModularEditor.doCreateCompound
        method=javafig.gui.ModularEditor.doBreakCompound
        method=javafig.gui.ModularEditor.doMoveObject
        method=javafig.gui.ModularEditor.doCopyObject
        method=javafig.gui.ModularEditor.doDeleteObject
        method=javafig.gui.ModularEditor.doMovePoint
        method=javafig.gui.ModularEditor.doInsertPoint
        method=javafig.gui.ModularEditor.doCutPoint
        method=javafig.gui.ModularEditor.doMirrorXObject
        method=javafig.gui.ModularEditor.doMirrorYObject
        method=javafig.gui.ModularEditor.doScaleObject
        method=javafig.gui.ModularEditor.doAlignObjects

method=javafig.gui.ModularEditor.doSnapObjectToGrid
        method=javafig.gui.ModularEditor.doConvertObject
        method=javafig.gui.ModularEditor.doResizeText
        method=javafig.gui.ModularEditor.doUpdate
        method=javafig.gui.ModularEditor.doCancelUpdate
        method=javafig.gui.ModularEditor.enableUpdateAll
        method=javafig.gui.ModularEditor.enableUpdateNone

method=javafig.gui.ModularEditor.enableUpdateInvert
        method=javafig.gui.ModularEditor.doEditObject

method=javafig.gui.ModularEditor.doEditGlobalAttributes
        method=javafig.gui.ModularEditor.doZoomRegion
        method=javafig.gui.ModularEditor.doZoomIn
        method=javafig.gui.ModularEditor.doZoomOut
        method=javafig.gui.ModularEditor.doZoom11
        method=javafig.gui.ModularEditor.doPanHome
        method=javafig.gui.ModularEditor.doPanLeft
        method=javafig.gui.ModularEditor.doPanRight
        method=javafig.gui.ModularEditor.doPanUp
        method=javafig.gui.ModularEditor.doPanDown
        method=javafig.gui.ModularEditor.doSetGridNone
        method=javafig.gui.ModularEditor.doSetGridCoarse
        method=javafig.gui.ModularEditor.doSetGridMedium
        method=javafig.gui.ModularEditor.doSetGridFine
        method=javafig.gui.ModularEditor.doSetNoSnap
        method=javafig.gui.ModularEditor.doSetSnap12
        method=javafig.gui.ModularEditor.doSetSnap14
        method=javafig.gui.ModularEditor.doSetSnap18
        method=javafig.gui.ModularEditor.doSetUnitsInches

method=javafig.gui.ModularEditor.doSetUnitsMillimeter
```

```
method=javafig.gui.ModularEditor.doSetUnitsXfigMillimeter

method=javafig.gui.ModularEditor.doSnapAllObjectsToGrid

method=javafig.gui.ModularEditor.doClearUserColors

method=javafig.gui.ModularEditor.doWriteHadesResource
        method=javafig.gui.ModularEditor.doRedraw

method=javafig.gui.ModularEditor.doStartNewDrawing
        method=javafig.gui.ModularEditor.doSelectFile
        method=javafig.gui.ModularEditor.doMergeFile
        method=javafig.gui.ModularEditor.doSelectURL
        method=javafig.gui.ModularEditor.doMergeURL

method=javafig.gui.ModularEditor.handleParserCallback

method=javafig.gui.ModularEditor.handleParserMergeCallbac
k

method=javafig.gui.ModularEditor.handleCommandCallback
        method=javafig.gui.ModularEditor.doQuit
        method=javafig.gui.ModularEditor.doSaveFile
        method=javafig.gui.ModularEditor.doSaveFileAs
        method=javafig.gui.ModularEditor.doSaveToConsole
        method=javafig.gui.ModularEditor.doPrintViaAWT
        method=javafig.gui.ModularEditor.doPrintUndoStack
        method=javafig.gui.ModularEditor.doPrintClipboard
        method=javafig.gui.ModularEditor.doPrintObjects
        method=javafig.gui.ModularEditor.doShowMessages

method=javafig.gui.ModularEditor.doShowAboutDialog

method=javafig.gui.ModularEditor.doShowLicenseDialog

method=javafig.gui.ModularEditor.doShowDeadlockDialog

method=javafig.gui.ModularEditor.doShowChangesDialog

method=javafig.gui.ModularEditor.doShowMouseButtonDialog

method=javafig.gui.ModularEditor.doShowShortcutKeysDialog
        method=javafig.gui.ModularEditor.doShowFaqDialog
        method=javafig.gui.ModularEditor.doShowHelpDialog
        method=javafig.gui.ModularEditor.doShowDemoGold
        method=javafig.gui.ModularEditor.doShowDemoHouse
        method=javafig.gui.ModularEditor.doShowDemoWatch

method=javafig.gui.ModularEditor.doShowDemoCircuit
        method=javafig.gui.ModularEditor.doShowDemoLayout

method=javafig.gui.ModularEditor.doShowDemoPictures

method=javafig.gui.ModularEditor.doShowDemoRotated

method=javafig.gui.ModularEditor.doShowDemoUnicode

method=javafig.gui.ModularEditor.doShowDemoWelcome
    }
    javafig.gui.EditTextDialog.getStatusMessage {
        method=javafig.*.getStatusMessage
    }
    javafig.gui.EditPolylineDialog.getStatusMessage {
        method=javafig.*.getStatusMessage
    }
    javafig.gui.EditEllipseDialog.getStatusMessage {
        method=javafig.*.getStatusMessage
    }
    javafig.gui.EditTriggerDialog.getStatusMessage {
        method=javafig.*.getStatusMessage
    }
    javafig.gui.EditImageDialog.getStatusMessage {
        method=javafig.*.getStatusMessage
    }
    javafig.gui.EditRectangleDialog.getStatusMessage {
        method=javafig.*.getStatusMessage
    }

javafig.gui.EditGlobalAttributesDialog.getStatusMessage {
        method=javafig.*.getStatusMessage
    }
    javafig.commands.ZoomRegionCommand.execute {
        method=javafig.*.doZoomRegion
    }
]

"java.lang.ClassLoader.defineClass(java.lang.String,byte[
],int,int)" [
]

java.lang.ClassLoader.findSystemClass [
    java.util.SystemClassLoader.loadClass {
    }
]

java.util.SystemClassLoader.loadClass [
]

java.io.ObjectInputStream.<init> [
    ajax.jbc.util.salamis.SalamisCodeLoader.readCode {
        serialized=ajax.jbc.util.salamis.*
        serialized=ajax.jbc.util.*
        serialized=java.util.Hashtable
    }
    sun.security.provider.IdentityDatabase.fromStream {
        serialized=sun.security.*
        serialized=java.security.*
    }
]

sun.awt.windows.WFontPeer.getFontCharset [
]
```