ABSTRACTION-BASED PROGRAM SPECIALIZATION

By

SHAWN MICHAEL LAUBACH

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma
1999

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July 1999

ABSTRACTION-BASED PROGRAM

SPECIALIZATION

Thesis Approved:

Mansur Samadzadeh
Thesis Advisor

D. E. Hedman

J Chandler

Wayne B. Powell
Dean of the Graduate College

PREFACE

Abstraction-based program specialization (ABPS) was investigated so that it could be applied to Java and make automated improvements to help with finite state verification. Research was conducted on partial evaluation and abstract interpretation. A prototype to do abstraction-based program specialization was constructed by Hatcliff, Dwyer, and Laubach. This work scaled the prototype to a subset of Java and made some general improvements. Today's software is large and complex. Because of this complexity, traditional validation and program testing techniques are hard to apply. One method in use is finite-state verification (FSV). FSV requires a program to be modeled as a finite-state transition system. Currently, the modeling is done by hand, an error-prone process. Also, the state space of a non-trivial program is extremely large (potentially infinite).

This thesis created an ABPS that uses partial evaluation and abstract interpretation to reduce a program model's state space. Partial evaluation performs symbolic execution; it specializes programs by folding constants and pruning infeasible branches from the computation tree. The abstract interpretation component replaces program data types with small sets of abstract tokens that capture information relevant to properties being verified. This can dramatically reduce a program's state space. Abstraction-based program specialization is a viable option for improving code and automating the use of finite state verifiers. Much work still needs to be done to completely scale abstraction-based program specialization to include all of Java and to make the process more automatic. Finally, several examples illustrate how ABPS can be applied to automatically create models of simple software systems.

ACKNOWLEDGMENTS

I would like to thank Dr. John M. Hatcliff for convincing me to continue my education. He has helped me gain a whole new understanding and insight. I would especially like to thank him for bringing me into his research and allowing me to make such a contribution.

I also want to thank my wife, Angela R. Laubach. She has stood behind me for every decision and has prodded me onto completion.

I would also like to thank Dr. Mansur H. Samadzadeh, Dr. G. E. Hedrick, and Dr. John P. Chandler for serving on my committee. I would especially like to thank Dr. Samadzadeh for being my advisor and for his high standards that improved the quality of this work.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Modern Software Systems

Many of today's software systems are large, concurrent systems developed by teams of programmers. Because of their complexity, traditional validation and program testing are hard to apply to these systems. Usually, one turns to systematic tools that use semantic and formal methods. Proof-based methods usually involve proving the partial or total correctness of the relevant piece of software. Model-based methods, *i.e.*, verification or model checking, check whether certain specified invariants hold.

Proof-based methods are often difficult to use because they require the user to construct manually (with some degree of automated assistance) a complete proof of the program's correctness. While model-based methods cannot establish properties as strong as proof-based methods can, model-based methods are highly automated and relatively easy to use.

1.2 Finite State Verification

Model-based, finite-state verification (FSV) [18] techniques can be used to check that a system satisfies certain properties. For example, FSV can verify that the system is dead-lock free or that when the program arrives at a certain point a variable has a particular value. To apply FSV, first one models the system to be reasoned about as a

finite-state transition system [18]. Then one describes the specification. Finally, one gives the finite-state transition system and the specification to a verifier. The verifier finds all the reachable states while ensuring that the specification is satisfied at each state. If the specification does not hold, FSV will give a trace counter-example that caused the check to fail.

FSV was originally developed for hardware verification [10], but is now being applied to software to assure high quality. FSV has been used effectively to validate many applications including network protocols [18, 25, 32], graphical user interfaces [10], railway interlocking systems [6], and industrial control systems [5, 14].

FSV is a promising technique for verifying software. However, it does have a large drawback: the size of the state space. The state space for a software system can be very large (potentially infinite), so it is difficult to check each state. One solution is to map the software components to suitable abstractions with small finite state spaces [17].

This has been done in the past by performing the mappings by hand [10, 14, 32]. This requires unfolding loops, in-lining methods (most FSV tools cannot handle method calls), changing dynamic memory allocation to compile time, and other steps. In addition, the user must come up with valid, usable abstractions that safely abstract or model the system. This works, but it is tedious, slow, and error prone.

1.3 Abstraction-Based Program Specialization

What is needed is an automatic tool for constructing abstract models. It appears that this can be done with two semantic based techniques: partial evaluation and abstract interpretation. Partial evaluation is an automatic technique for specializing programs based on information known about the environment or expected patterns of use. Abstract interpretation is a rigorous methodology for static program analysis

by manipulating abstract tokens. These together can reduce the state space by symbolically executing portions of the program and by reducing the conditional branches. The exact abstract representation used depends on the properties to be verified. It seems possible to combine these methods into a tool: an abstraction-based program specializer (ABPS). ABPS would be a set of automatic tools that can do partial evaluation (see Section 2.1) and can do abstract interpretation (see Section 2.2) on programs to create models for FSV.

1.3.1 ABPS Example

As an example, taken from [17], consider a connector used in the construction of concurrent software [11, 12]. This connector describes the topology, inter-connection, and communication constraints of replicated worker-style computations. The workers communally accesses shared pool of work items. It is implemented in Ada.

Figure 1.1 illustrates a code skeleton for the pool component; significant detail has been in-lined to streamline the example. Since FSV works by enumerating and checking all possible program states, even the `Natural` domain for the single variable `wc` that maintains the number of elements in the work pool causes state-space enumeration to be intractable. To obtain a usable state space, we abstract the values of `wc : Natural` can be abstracted using a *counter abstraction* whose abstract domain ranges over just four values: *unknown*, *zero*, and *positive*. Figure 1.2 gives the ordering of these values and the associated abstract operators.

Figure 1.1 also illustrates the results of applying ABPS using the counter abstraction. The results of specialization are given in Ada comments (*e.g.*, `-- wc : AbsNatural`). The type `Natural` and the associated operations are specialized to the type `AbsNatural` (an enumerated type containing the values `zero`, `positive`, `unknown`) and associated operations. In summary, ABPS yields a source-level abstraction of the original pro-

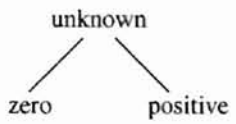
```
task body ActivePool is
 wc :  Natural;
 -- wc :  AbsNatural;
begin
 wc := 0;
 -- wc := zero;
 Outer:  loop
  loop
   select accept ShutDown;
    exit Outer;
   or accept Start(...);
    exit;
   or accept Put(...)  do
     wc := wc + 1;
     -- wc := positive;
    end Put;
   end select;
  end loop;
```

```
loop
 select when ...  or wc > 0 =>
 -- select when ...  or wc = positive =>
  accept Get(...)  do
   wc := wc - 1;
   -- wc := unknown;
  end Get;
 or accept Put(...)  do
  wc := wc + 1;
  -- wc := positive;
  end Put;
 or ...
 end select;
 if ...  and wc=0 then
 -- if ...  and wc=zero then
  exit;
 end if;
end loop;
end loop Outer;
end ActivePool;
```

Figure 1.1: Skeletal Ada for Pool Task

gram's behavior (see Section 3.2). Information about the specific number of workers wc has been abstracted; we only maintain information about whether wc is 0, positive, or unknown. After the program has been abstracted, it can be automatically translated into the input languages of SPIN, SMV, and other model checkers using a tool set constructed by Jay Corbett [26]. The resulting model can then be checked against specifications written in various model logics.

This simple use of ABPS enables, for example, verification of the specification, "whenever the computation terminates the work pool is empty," that is, whenever the outer loop is exited wc=0. Furthermore, this abstraction does not require the user to specify any bound on the size of the work pool. Other specifications may require different abstractions for effective verification. Dwyer and Pasareanu [13] outline the



assignZero(v) = zero;
incPos(v) = positive;
decPos(v) = unknown;
isGtZero(v) = positive;
isEqZero(v) = zero;
identity(v) = v;

Figure 1.2: Counter Abstract Interpretation

methodology that we expect one to follow when choosing appropriate abstractions.

1.4 Goals of the Work

The long range goal of this work is to develop a full-scale abstraction-base program specializer for Java Byte Code. The following steps summarize the approach taken in this thesis.

1. Perform an initial investigation using a very simple flowchart language called FCL. This is not the FCL by Wulf, *et al.* in [33].

2. Based on the experience gained, the work can be scaled to Java Byte Code.

3. The system can be run on various Java examples and the effectiveness and usability of the system can be assessed.

1.4.1 Initial Investigation

The first step is to do the initial investigation on FCL (see Section 3.1). FCL is small enough to allow a clean semantic presentation, but rich enough conceptually to illustrate a multitude of issues associated with program specialization [15, 20, 19].

- Formalization of ABPS for FCL

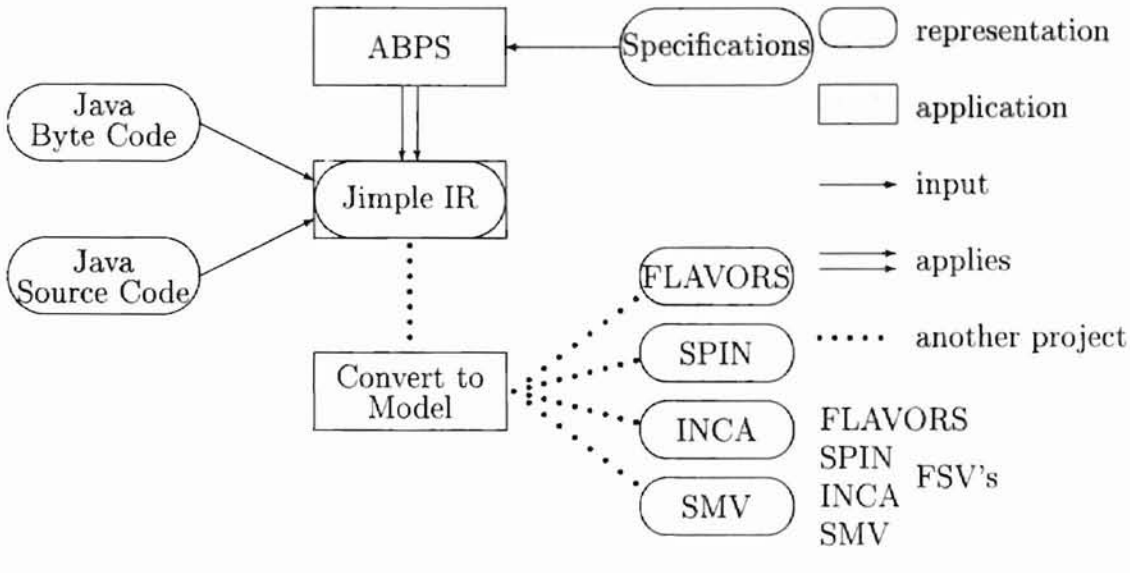


Figure 1.3: System Diagram

First, ABPS is formalized for FCL. This included describing the many rules and functions needed for the ABPS system.

- Proving correctness for FCL

  Next, ABPS is proved correct for FCL. This involved showing that the specialized program produced by ABPS is a safe abstraction of the original program.

- Prototype for FCL

  Finally, a working prototype is created to work on FCL. This prototype works with concrete and abstract examples.

1.4.2 Scaling to Java Byte Code

The second stage is to scale the ABPS for FCL to Java Byte Code. The system is shown in Figure 1.3. Jimple is a set of tools and an intermediate representation (IR) of Java source code and Java Byte Code developed by researchers at McGill

University was used. These tools are part of the Soot project, which can be found at http://www.sable.mcgill.ca/soot/. Currently, Jimple is applied to Java Byte Code, and a Jimple representation of the class is produced. The Jimple representation is an abstract syntax tree (AST). It is the purpose of this work to create a set of ABPS tools to work on this Jimple representation. FLAVERS, SPIN, INCA, and SMV are all FSV's. The dotted lines represent work being done by others. This work includes a set of tools that translates a representation to a model that one of the FSV's will be able to use. James Corbett, from the University of Hawaii, is currently working on this development called BIRC [26].

Scaling to Java includes the following tasks.

- Appropriate intermediate language

  Obtaining an appropriate IR is perhaps the most difficult aspect of scaling ABPS to Java. Stack based code, such as Java Byte Code, is difficult to analyze [31]. To solve this problem, the IR does not use a stack based representation, but it converts all stack positions to variables to use in expressions. There is still much code involved to remove the stack and convert the byte code to an easier to use representation.

- Foundations of ABPS for Java Byte Code

  Because a full ABPS system is beyond the scope of this thesis, the next step included deciding what constructs to include and what to leave for later work. It was decided to limit the Java to integer arithmetic, simple control flow (*i.e.* **goto**'s, **if**'s, and **return**'s), and in-lining of static methods. Techniques were then designed to handle the new constructs and to represent abstractions.

- Java implementation

  Finally, an implementation of ABPS was written for a Java. Further research will be required to get a full implementation working for Java.

1.4.3 Assessment

The last part of the system development is to evaluate the software. The full evaluation cannot be done until a more complete implementation is done. There are some tests that can be done but these are limited by the limits placed on the implementation. To do the evaluation, the resulting systems will be run on example programs. Various aspects including the following will be assessed.

- Usability

  The usability of the system, *i.e.*, issues such as how difficult it is to use and what types of problems the user encountered, will be assessed. This includes the things the user must apply (the program and specification).

- Effectiveness

  The question of whether the system worked well on a illustrative program. Plus, it will determine whether the output needed any additional modifications, or whether it could be passed straight to a verifier.

- Technological challenges

  The execution of the system will be tested on an example. Slow and inefficient parts will be identified and recorded to increase speed or decrease memory usage. Also, difficult parts of the implementation will be discussed.

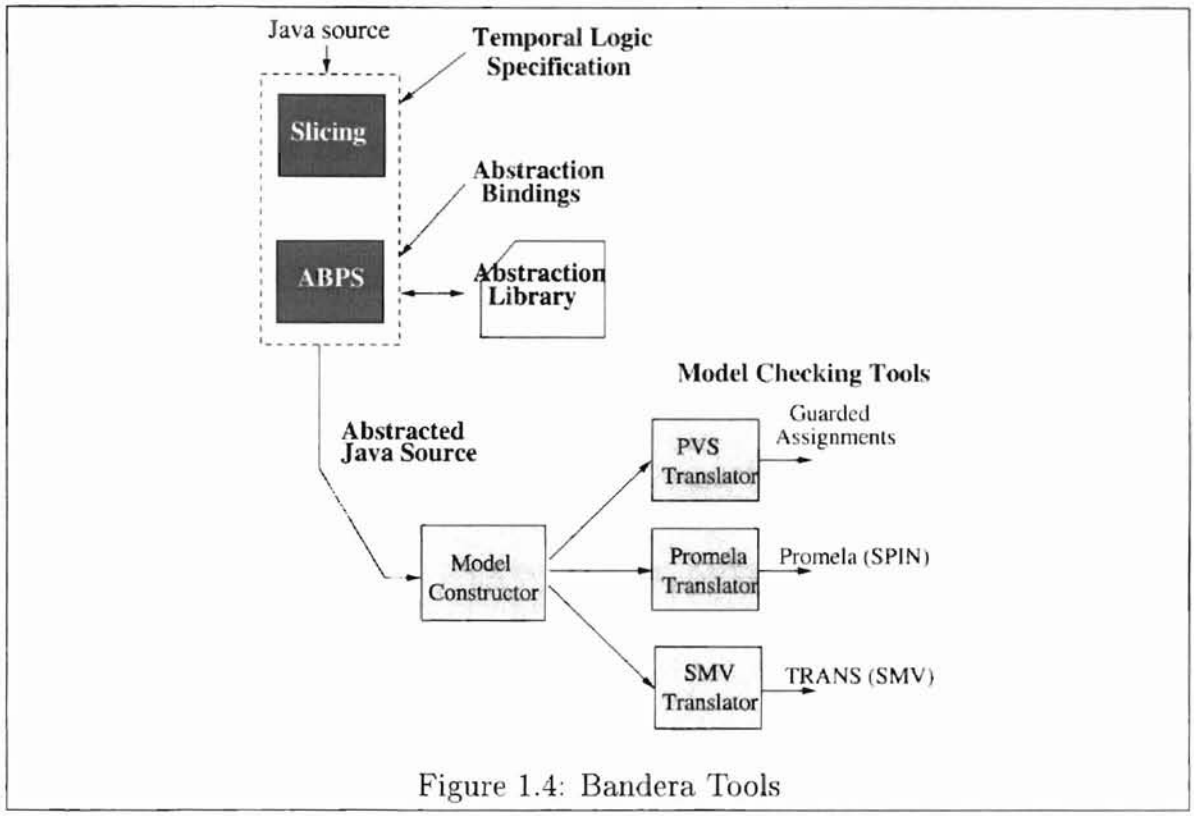


Figure 1.4: Bandera Tools

1.5 Overview

The work on ABPS is part of a larger project, called the Bandera project, funded by DARPA/NASA on automatically constructing models for finite-state verification of software. Figure 1.4 diagrams the tools. This project includes the verification of Java Byte Code, slicing, and mapping byte code to finite state machines. Other researchers from the University of Hawaii, the University of Massachusetts, and Kansas State University are collaborating on these projects [26].

1.6 What Follows

The rest of this thesis is organized as follows. Chapter II describes the basic principles of partial evaluation and abstract interpretation. It highlights the techniques needed to apply in this work. Chapter III gives a brief overview of abstraction-based pro-

gram specialization. Chapter IV provides the system overview and the intermediate representation used. Chapter V describes the use of Jimple and Jimple's conversion to structures used by the specializer. Chapter VI contains a description of many of the data structures used. Chapter VII describes the methods that make up the core of the specializer. Chapter VIII describes the specialization operators used by the specialization. Chapter IX presents the results of the work. Chapter X presents the conclusion with a discussion of future work.

CHAPTER II

BACKGROUND

Abstraction-based program specialization is a combination of partial evaluation and abstract interpretation. This chapter provides background material on these two technologies.

2.1 Conventional Partial Evaluation

Historically, the main goal of partial evaluation is to generate efficient programs from general ones by completely automatic methods [20]. Usually, general programs are simpler but less efficient than a specialized program produced by a partial evaluator.

2.1.1 What Partial Evaluation Is

Partial evaluation (PE) is a technology for automatic program specializations and customization. A partial evaluator is given a subject program p together with part of its input data, in_1. Its effect is to construct a new program p_{in_1} which, when given p's remaining input in_2, will yield the same result that p would have been produced given both inputs [20]. Therefore, a partial evaluator is a program specializer, usually called mix, as in Figure 2.1.

Figure 2.2 shows a two input program that computes m^n. This program can be specialized to the one in Figure 2.3, if n has a known value of 3. This is done by

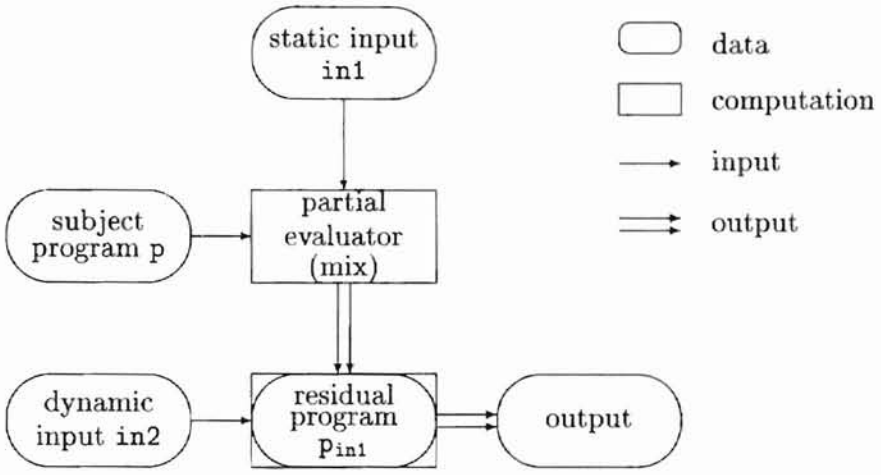


Figure 2.1: A Partial Evaluator

precomputing all expressions involving n and unfolding the loop. The unfolding can be done because the control depends upon n. If, however, we tried to partial evaluate where $m = 3$ and n is unknown, we would achieve nothing because the control flow is not known (the partial evaluator would go into an infinite loop).

2.1.2 How Partial Evaluation Works

As Jones [20] notes, three main partial evaluation techniques are well known from program transformation: symbolic computation, unfolding, and program point specialization. Figures 2.2 and 2.3 applied the first two techniques; the third was unnecessary since the specialized program had no function calls. The idea of program point specialization is that a single function or label in program p may appear in the specialized program p_{in_1} in several specialized versions, each corresponding to data determined at partial evaluation time. For example, there are three versions of the source program line $result := result * m$ corresponding to situations where the line was process with $n = 3, 2, 1$.

```
int pow(int m, int n)
{
   result = 1;
   while (n > 0)
   {
      result = result * m;
      n = n - 1;
   }

   return result;
}
```

Figure 2.2: Power Function

```
int pow3(int m)
{
   result = 1;
   result = result * m;
   result = result * m;
   result = result * m;

   return result;
}
```

Figure 2.3: Specialized Power Function

To determine what to residualize (put into the output program) and what to compute away, an analysis needs to be done. The analysis can be done while the specializer is running, called on-line PE, or before it is run as preprocessing, called off-line PE. During on-line PE, the values in the store are tagged whether they are static or dynamic. Static data is known while dynamic data is unknown. The PE uses this to determine if an expression is computable. During off-line PE, the analysis is run before the specializer and determines whether each expression, statement, and other language constructs are static or dynamic and tags the construct appropriately.

Then during specialization, this information determines if the expression should be evaluated or residualized [20].

Partial evaluation uses two data structures to schedule programs pointers for specialization: a pending list and a "seen-before" set. A pending list is a list of program states to be specialized. The seen-before set contains all the states that have been specialized.

At the start of specialization, a PE adds the start state to the pending list. The start state is the initial label and the initial store. Each time through, the PE gets the next state out of the pending list and checks to see if it is in the seen before set. This stops the specializer from specializing a state more than once. If it is a new state, the corresponding basic block is retrieved then specialized.

To specialize a block, each statement must be specialized. The specializer checks to determine whether the statement is static or dynamic. If the statement is static, the specializer evaluates the statement, otherwise, it residualizes the statement. Next, the jump is specialized. If the jump is a **goto** or **return**, it is residualized as is. If it is an **if**, it is checked to see if the expression is static or dynamic. If static, it can be determined which branch to follow, so it can be residualized as a **goto**. If the expression cannot be determined, it is residualized back as an **if**. Finally, all the states reachable from the block are added to the pending. This continues until the pending list is empty.

2.1.3 Related Work

2.1.3.1 C-mix. C-mix is a partial evaluator for ANSI C developed as part of Andersen's Ph.D. dissertation [2]. It incorporates complex features of the imperative language C, such as, structures, multidimensional arrays, and pointers, and it performs sophisticated analysis to handle those features.

2.1.3.2 Tempo. Tempo was developed at University of Rennes/IRISA. Unlike C-mix, Tempo focuses on system software written in C. This simplifies the structure of the partial evaluator and enhances some solutions, but it cannot handle the full range of ANSI C.

2.2 Abstract Interpretation

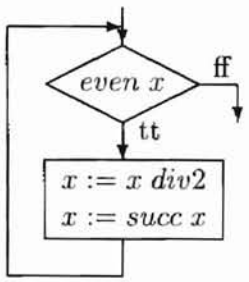
Much of the work on abstract interpretation was originally done by Cousot and Cousot [8]. The discussion below is adapted from material in survey articles by Schmidt [27] and Jones and Neilson [21].

The execution trace of a program when applied to its run-time data is a concrete interpretation (CI). When the data are tokens that denote properties of run-time data, the execution trace is an abstract interpretation (AI). In other words, AI is a "symbolic execution" where the symbols have semantic content. For example, a type inference implementation is an AI that uses tokens such as integer and boolean instead of the concrete values 5 and false.

When the run-time data sets are replaced by tokens, the operators must be revised to work on the tokens. For example, an addition operator for concrete integers must be revised to define addition on the data tokens, such as:

$$a_1 + a_2 \quad = \quad \begin{cases} even & \begin{cases} a_1 = a_2 = even \\ a_1 = a_2 = odd \end{cases} \\ odd & \begin{cases} a_1 = even, a_2 = odd \\ a_1 = odd, a_2 = even \end{cases} \end{cases}$$

A crucial issue of AI is termination. A CI of a program may terminate with its run-time data, the AI may not. This is because the tokens are less precise and nondeterminism arises. When a test cannot be decided on because the values are tokens, both execution paths must be traversed. For example, if a test was $x > 0$ and x's value is an integer, then the result of the test is unknown and both the true and



Concrete Semantics:
$Val = Nat$
$(2n \vdash even\ x) \rightarrow (2n \vdash x := x\ div2)$
$(2n+1 \vdash even\ x) \rightarrow (2n+1 \vdash exit)$
$(2n \vdash x := x\ div2) \rightarrow (n \vdash x := succ\ x)$
$(n \vdash succ\ x) \rightarrow (n+1 \vdash x := even\ x)$

Concrete Tree
$4 \vdash even\ x$
$4 \vdash x := x\ div2$
$2 \vdash x := succ\ x$
$3 \vdash even\ x$
$3 \vdash exit$

Figure 2.4: Flowchart and Concrete Interpretation

false branches must be traversed. This means loops have the potential to be traversed forever. One strategy to solve this problem is for every infinite path in the program's abstract tree to contain a repetition of a node seen earlier in the path (similar to the seen-before set in a partial evaluator). This means the trace is a regular tree, a tree where every infinite path has a repetition node [27], and the construction of the tree can be terminated at these repetition nodes.

2.2.1 Abstract Interpretation of Flowchart Programs

Figure 2.4 shows a flowchart program [27] that uses a store with a single variable x. A state is a store/program point pair, $(v \vdash pp)$, where v is the value of x and pp is the current program point. The concrete semantics rules specific to the flowchart are listed in the middle column of Figure 2.4. The program's concrete tree has one path when executed with input 4.

Let us say better target code can be generated for commands whose inputs are always even numbers. Figure 2.5 displays the abstract semantics of such a situation. The Val set is abstracted to $AbsVal = \{e, o\}$, denoting even and odd numbers, respectively. Also, each operator is revised to the abstract rules. For this example, the abstract semantics is nondeterministic for the interpretation of $div2$. This means

Abstract Transitions:
$AbsVal = \{e, o\}$
$(e \vdash even\ x) \rightarrow (e \vdash x := x\ div 2)$
$(o \vdash even\ x) \rightarrow (o \vdash exit)$
$(e \vdash x := x\ div 2) \rightarrow (e \vdash x := succ\ x)$
$(e \vdash x := x\ div 2) \rightarrow (o \vdash x := succ\ x)$
$(e \vdash succ\ x) \rightarrow (o \vdash x := even\ x)$
$(o \vdash succ\ x) \rightarrow (e \vdash x := even\ x)$

Abstract tree:

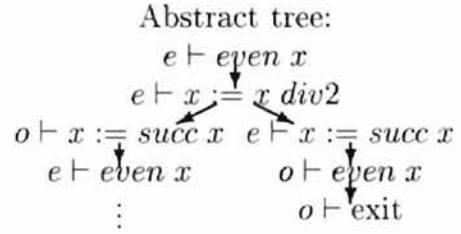


Figure 2.5: Abstract Interpretation of Flowchart

that the abstract tree is nondeterministic. By nondeterministic interpretation, it is meant that the decision cannot be made as to what path in the tree should be taken.

The abstract tree contains more paths than the concrete tree and it is infinite. There is, however, a repetition node in every infinite path. Thus the tree is regular and has a finite representation, shown in Figure 2.5, meaning termination is not a problem, because there is a finite number of nodes in the tree.

2.2.2 Relating Concrete to Abstract Trees

To establish the correctness of an AI, we need a function to map concrete data to the abstract tokens that best represent them. Let $\beta : Val \rightarrow AbsVal$ be such a function. In technical terms, the function β is a homomorphism between concrete and abstract values. For the Figures 2.4 and 2.5, β would be $\beta(2n) \rightarrow e$ and $\beta(2n+1) \rightarrow o$ for $n \geq 0$. For the transition relation, the basic correctness property for transitions is: for all program points, pp, pp', and $c, c' \in Val$,

> $(c \vdash pp) \rightarrow (c' \vdash pp')$ implies there exists $a' \in AbsVal$ and there exists a transition $(\beta(c) \vdash pp) \rightarrow (a' \vdash pp')$ such that $\beta(c') \sqsubseteq a'$

where $\sqsubseteq$ is a partial order on the concrete and abstract values. Computations involving abstract values cannot be more precise that those involving actual values, so we allow the values computed abstractly to be less precise than the result of exact computation followed by abstraction. For example, if we coded the $div2$ operator so that it is deterministic, then $(e \vdash x := x\ div2) \rightarrow (\top \vdash x := succ\ x)$, where $\top$ represents even or odd. If we use the extra element $\top$, then we need approximation ordering on $AbsVal = \{e, o, \top\}$: $a \sqsubseteq \top$ and $a \sqsubseteq a$, for all $a \in AbsVal$. We require the abstract transition relation to be monotonic with respect to the ordering:

$(a_1 \vdash pp) \rightarrow (a_1' \vdash pp')$ and $a_1 \sqsubseteq a_2$ imply there exists a transition $(a_2 \vdash pp) \rightarrow (a_2' \vdash pp')$ such that $a_1' \sqsubseteq a_2'$

Intuitively, this means that the transition relation on abstract values preserves the degree of information reflected in the tokens.

Let us define a binary relation $\text{safe}_{\text{Val}} \subseteq Val \times AbsVal$ as

$$c\ \text{safe}_{\text{Val}}\ a \text{ iff } \beta(c) \sqsubseteq a$$

which means a safely approximates c. Now let us define a safety relation upon states as

$$(c \vdash pp)\ \text{safe}_{\text{State}}\ (a \vdash pp) \text{ iff } c\ \text{safe}_{\text{Val}}\ a$$

which says an abstract state safely approximates a concrete one if the input values are related and the program points are the same pp.

Finally, for program points pp, pp' and values $c, c' \in Val$,

$c\ \text{safe}_{\text{Val}}\ a$ and $(c \vdash pp) \rightarrow (c' \vdash pp')$ imply there exists $a' \in AbsVal$ and there exits $(a \vdash pp) \rightarrow (a' \vdash pp')$ such that $c'\ \text{safe}_{\text{Val}}\ a'$

Pictorially, we have

$$\begin{array}{ccc} (c \vdash pp) & \text{safe}_{\text{State}} & (a \vdash pp) \\ \downarrow & & \downarrow \\ (c' \vdash pp') & \text{safe}_{\text{State}} & (a' \vdash pp') \end{array}$$

So, for any concrete transition, there is a corresponding safe abstract transition. In other words, for any concrete trace of a program, there is a corresponding safe abstract transition.

Abstract interpretation can be used for binding time analysis, type inference, live variable analysis, and many other analyses [16].

CHAPTER III

ABSTRACTION-BASED PROGRAM SPECIALIZATION

This chapter presents a formalization of ABPS using FCL from Hatcliff, Dwyer, and Laubach [17]. This chapter is basically a minor revision of that article. This author's main contribution to the work was the implementation of ABPS to FCL in Java. This author also contributed advice on the semantics of ABPS. Other works on ABPS include Consel and Khoo [7] and Jones [19], which developed the formal frameworks to support the idea. ABPS, however, has not been incorporated into full-fledged implementations at the completion of this thesis.

3.1 Flowchart Language FCL

FCL is a simple flow chart language that can be used to study many of the features of a full language. This allows us to create methods for ABPS that is easy to understand and use. It is also very easy to add features to FCL. This makes it easy to use and allows us to scale gracefully.

3.1.1 Syntax

Figure 3.1 presents the syntax of an FCL program. An FCL program $(l)\ b^+$ consists of a list of basic blocks b^+ and the label of the initial basic block. Each basic block has a label, a list of assignments (possibly empty), and a jump. FCL has three kinds

Syntax Domains:

p	$\in$	Programs[Σ]	x	$\in$	Variables[Σ]
b	$\in$	Blocks[Σ]	e	$\in$	Expressions[Σ]
l	$\in$	Block-Labels[Σ]	j	$\in$	Jumps[Σ]
a	$\in$	Assignments[Σ]	o	$\in$	Operations[Σ]
al	$\in$	Assignment-Lists[Σ]	t	$\in$	Tests[Σ]

Grammar:

$$
\begin{array}{lcl}
p & ::= & (l)\ b^+ \\
b & ::= & l:\ al\ j \\
al & ::= & a\ al \mid \cdot \\
a & ::= & x := e; \\
e & ::= & x \mid o(e^*) \\
j & ::= & \textbf{goto}\ l; \mid \textbf{return}; \mid \textbf{if}\ t(x^*)\ \textbf{then}\ l_1\ \textbf{else}\ l_2;
\end{array}
$$

Figure 3.1: Syntax of the Flowchart Language FCL

of jumps: an unconditional **goto**, a conditional jump **if**, and a special jump **return** that terminates a program's execution. The output is the collective value of all the program's variables for simplicity.

A signature Σ parameterizes the syntax of FCL. Σ contains operations, tests, and constants. Σ specifies the set of operator symbols Operations[Σ] and a set of test symbols Tests[Σ]. Both have an associated arity $\mathsf{arity}(o)$. Constants are 0-ary operators and are denoted by Constants[Σ].

3.1.2 Semantics

The meaning of a Σ-program, a program to which Σ is applied, is parameterized by a Σ-algebra A that provides an interpretation for the signature Σ. A Σ-algebra A consists of a carrier set Values[A] (*e.g.*, an upper semi-lattice) with partial order $\sqsubseteq_A$, sets Operations[A] and Tests[A] that contain functions implementing the operations and tests of Σ, and a map $[\![\cdot]\!]_A^\Sigma$ that maps each operation and test symbol in Σ to

the corresponding implementation in Operations[A] and Tests[A]. Different types of traces (concrete and abstract) can be obtained by substituting different Σ-algebras.

Figure 3.2 formalizes the semantics of a Σ-program with respect to a Σ-algebra A in terms of traces. A trace shows the transitions

$$(l_0, \sigma_0) \rightarrow (l_1, \sigma_1) \rightarrow (l_2, \sigma_2) \rightarrow \ldots$$

a program can make between computational states $(l_i, \sigma_i) \in$ States[A] where $l_i \in$ Labels[Σ] is the label of the current basic block and $\sigma_i \in$ Stores[A] is the current store. A store $\sigma \in$ Stores[A] is a partial function from Variables[Σ] to Values[A]. The set of defined variables in the domain of σ is written $dom(\sigma)$. A σ is p-compatible when it defines only the variables contained in program p.

A special label halt is added to Block-Labels[Σ] that maps a label $l \in$ Block-Labels[Σ] to a block $b \in$ Blocks[Σ] to represent the terminal state. All finite branches of a trace will end in a state (halt, σ) for some store σ.

A program is represented using a partial function Γ called a block-map that maps a label $l \in$ Block-Labels[Σ] to a block $b \in$ Blocks[Σ]. Γ is defined for exactly the labels that name blocks in the program being interpreted.

3.2 Abstraction-Based Specialization

Our abstraction-based specialization framework combines the trace generation system with code generation. The idea is to carry out the trace while simultaneously generating code that is specialized with respect to the information accumulated in the trace.

Figures 3.3 and 3.4 present the abstraction-based program specializer. The specializer is parameterized on a specialization structure

$$\Xi = (\Sigma, \Sigma_{res}, A, \pi, \theta, R, \mathit{lift}).$$

Semantic Domains:

$$v \in \text{Values}[A] \qquad o_A \in \text{Operations}[A] \qquad t_A \in \text{Tests}[A]$$

$$\begin{array}{lcll} \sigma & \in & \text{Stores}[A] & = & \text{Variables}[\Sigma] \rightharpoonup \text{Values}[A] \\ l & \in & \text{Labels}[\Sigma] & = & \text{Block-Labels}[\Sigma] \cup \{\mathsf{halt}\} \\ \Gamma & \in & \text{Block-Maps[FCL]} & = & \text{Block-Labels}[\Sigma] \rightharpoonup \text{Blocks}[\Sigma] \end{array}$$

Expressions:

$$\frac{}{\sigma \Vdash_{expr} x \Rightarrow \sigma(x)} \qquad \frac{\sigma \Vdash_{expr} e_i \Rightarrow v_i \qquad o_A(v_1 \ldots v_n) = v}{\sigma \Vdash_{expr} o(e_1 \ldots e_n) \Rightarrow v}$$

Assignments:

$$\frac{\sigma \Vdash_{expr} e \Rightarrow v}{\sigma \Vdash_{assign} x := e; \Rightarrow \sigma[x \mapsto v]} \qquad \frac{}{\sigma \Vdash_{assigns} \cdot \Rightarrow \sigma}$$

$$\frac{\sigma \Vdash_{assign} a \Rightarrow \sigma' \qquad \sigma' \Vdash_{assigns} al \Rightarrow \sigma''}{\sigma \Vdash_{assigns} a\, al \Rightarrow \sigma''}$$

Jumps:

$$\frac{}{\sigma \Vdash_{jump} \textbf{goto } l; \Rightarrow \{(l, \sigma)\}} \qquad \frac{}{\sigma \Vdash_{jump} \textbf{return}; \Rightarrow \{(\mathsf{halt}, \sigma)\}}$$

$$\frac{t_A(x^*, l_1, l_2, \sigma) \;=\; \{(l'_1, \sigma'_1), \ldots, (l'_n, \sigma'_n)\}}{\sigma \Vdash_{jump} \textbf{if } t(x^*) \textbf{ then } l_1 \textbf{ else } l_2; \Rightarrow \{(l'_1, \sigma'_1), \ldots, (l'_n, \sigma'_n)\}}$$

Transitions:

$$\frac{\Gamma(l) = l :\ al\ j \qquad \sigma \Vdash_{assigns} al \Rightarrow \sigma' \qquad \sigma' \Vdash_{jump} j \Rightarrow \{(l'_1, \sigma'_1), \ldots, (l'_n, \sigma'_n)\}}{\Sigma, A \Vdash_\Gamma (l, \sigma) \rightarrow (l'_i, \sigma'_i) \;\; \forall i \in \{1, \ldots, n\}}$$

Figure 3.2: Trace Semantics of Σ-programs with Respect to Σ-algebra A

Semantic Domains:

$$v \in \text{Values}[A] \qquad o_A \in \text{Operations}[A] \qquad t_A \in \text{Tests}[A]$$

$$\begin{array}{lcll} w & \in & \text{Spec-Values}[A] & = & \text{Values}[A] \times \text{Expressions}[\Sigma] \\ \sigma & \in & \text{Stores}[A] & = & \text{Variables}[\Sigma] \rightharpoonup \text{Values}[A] \\ l & \in & \text{Labels}[\Sigma] & = & \text{Block-Labels}[\Sigma] \cup \{\mathsf{halt}\} \\ \Gamma & \in & \text{Block-Maps}[\text{FCL}] & = & \text{Block-Labels}[\Sigma] \rightharpoonup \text{Blocks}[\Sigma] \end{array}$$

Expressions:

$$\frac{}{\sigma \vdash_{expr} x \Rightarrow \langle\, \sigma(x)\,,\, x \,\rangle}$$

$$\frac{\sigma \vdash_{expr} e_i \Rightarrow \langle\, v_i\,,\, e_i' \,\rangle \qquad o_A(v_1 \ldots v_n) = v \qquad v \in R}{\sigma \vdash_{expr} o(e_1 \ldots e_n) \Rightarrow \langle\, v\,,\, \mathit{lift}(v) \,\rangle}$$

$$\frac{\sigma \vdash_{expr} e_i \Rightarrow \langle\, v_i\,,\, e_i' \,\rangle \qquad o_A(v_1 \ldots v_n) = v \qquad v \notin R}{\sigma \vdash_{expr} o(e_1 \ldots e_n) \Rightarrow \langle\, v\,,\, o(e_1' \ldots e_n') \,\rangle}$$

Assignments:

$$\frac{\sigma \vdash_{expr} e \Rightarrow \langle\, v\,,\, e' \,\rangle}{\sigma \vdash_{assign} x := e; \Rightarrow \langle\, \sigma[x \mapsto v]\,,\, [x := e';] \,\rangle}$$

$$\frac{}{\sigma \vdash_{assigns} \cdot \Rightarrow \langle\, \sigma\,,\, [] \,\rangle}$$

$$\frac{\sigma \vdash_{assign} a \Rightarrow \langle\, \sigma'\,,\, al' \,\rangle \qquad \sigma' \vdash_{assigns} al \Rightarrow \langle\, \sigma''\,,\, al'' \,\rangle}{\sigma \vdash_{assigns} a\ al \Rightarrow \langle\, \sigma''\,,\, al' +\!\!+ al'' \,\rangle}$$

Figure 3.3: Abstraction-Based Specialization (part 1)

Jumps:

$$\frac{}{\sigma \vdash_{jump} \textbf{goto } l; \Rightarrow \langle\, \{(l,\sigma)\}\,,\, \textbf{goto } \pi(l,\sigma);\,\rangle}$$

$$\frac{}{\sigma \vdash_{jump} \textbf{return}; \Rightarrow \langle\, \{(\mathsf{halt},\sigma)\}\,,\, \textbf{return};\,\rangle}$$

$$\frac{t_A(x^*, l_1, l_2, \sigma) \;=\; \{(l_1, \sigma')\}}{\sigma \vdash_{jump} \textbf{if } t(x^*) \textbf{ then } l_1 \textbf{ else } l_2; \Rightarrow \langle\, \{(l_1,\sigma')\}\,,\, \textbf{goto } \pi(l_1,\sigma');\,\rangle}$$

$$\frac{t_A(x^*, l_1, l_2, \sigma) \;=\; \{(l_2, \sigma')\}}{\sigma \vdash_{jump} \textbf{if } t(x^*) \textbf{ then } l_1 \textbf{ else } l_2; \Rightarrow \langle\, \{(l_2,\sigma')\}\,,\, \textbf{goto } \pi(l_2,\sigma');\,\rangle}$$

$$\frac{t_A(x^*, l_1, l_2, \sigma) \;=\; \{(l_1, \sigma_1'), (l_2, \sigma_2')\}}{\begin{array}{l}\sigma \vdash_{jump} \textbf{if } t(x^*) \textbf{ then } l_1 \textbf{ else } l_2; \\ \qquad \Rightarrow \langle\, \{(l_1,\sigma_1'), (l_2,\sigma_2')\}\,,\, \textbf{if } t(x^*) \textbf{ then } \pi(l_1,\sigma_1') \textbf{ else } \pi(l_2,\sigma_2');\,\rangle\end{array}}$$

Blocks:

$$\frac{\begin{array}{c}\sigma \vdash_{assigns} al \Rightarrow \langle\, \sigma_1\,,\, al_1\,\rangle \\ \sigma_1 \vdash_{jump} j \Rightarrow \langle\, \{(l_{2_i}, \sigma_{2_i}) \;\mid\; i \in \{1, \ldots, n\}\}\,,\, j_2\,\rangle\end{array}}{\sigma \vdash_{block} l : al\; j \Rightarrow \langle\, \{(l_{2_i}, \sigma_{2_i}) \;\mid\; i \in \{1, \ldots, n\}\}\,,\, (l,\sigma) : al_1\; j_2\,\rangle}$$

Specialization steps:

$$\frac{\mathcal{C}_0(\iota) \vdash_{block} \Gamma(\pi^{-1}(\iota)) \Rightarrow \langle\, \{(l_i', \sigma_i') \;\mid\; i \in \{1, \ldots, n\}\}\,,\, b'\,\rangle}{\vdash_\Gamma \langle \mathcal{S}\,,\, \mathcal{C}_0\,,\, \Gamma_{\mathrm{R}}\rangle \longmapsto \langle \mathcal{S}_n\,,\, \mathcal{C}_n\,,\, \Gamma_{\mathrm{R}}[\iota \mapsto b']\rangle} \qquad \text{if } \iota^\circ = \mathit{first}(\mathcal{S}_0)$$

where

$$\iota_i \;=\; \pi(l_i', \sigma_i') \qquad \text{for } i \in \{1, \ldots, n\}$$

$$\mathcal{C}_i \;=\; \left\{\begin{array}{ll} \mathcal{C}_{i-1}[\iota_i \mapsto \theta(\sigma_i', \mathcal{C}_{i-1}(\iota_i))] & \text{if } \mathcal{C}_{i-1}(\iota_i)\!\downarrow \\ \mathcal{C}_{i-1}[\iota_i \mapsto \sigma_i'] & \text{if } \mathcal{C}_{i-1}(\iota_i)\!\uparrow \end{array}\right\} \qquad \text{for } i \in \{1, \ldots, n\}$$

$$\mathcal{S}_0 \;=\; \mathit{remove\text{-}arcs}(\mathit{mark}(\mathcal{S}, \iota), \iota)$$

$$\mathcal{S}_i \;=\; \left\{\begin{array}{ll} \mathit{make\text{-}arc}(\mathcal{S}_{i-1}, \iota, \iota_i^{\bullet}) & \text{if } \iota_i = \mathsf{halt} \\ \mathit{make\text{-}arc}(\mathcal{S}_{i-1}, \iota, \iota_i^{\circ}) & \text{if } \iota_i \text{ not in } \mathcal{S}_{i-1} \text{ and } \iota_i \neq \mathsf{halt} \\ \mathit{make\text{-}arc}(\mathcal{S}_{i-1}, \iota_i, \iota_i^{m}) & \text{if } \iota_i^{m'} \text{ in } \mathcal{S}_{i-1} \text{ and } \iota_i \neq \mathsf{halt} \text{ where } m = \circ \\ & \text{if } \mathcal{C}_0(\iota_i) \sqsubset \mathcal{C}_n(\iota_i) \text{ and} \\ & m = m' \text{ if } \mathcal{C}_0(\iota_i) = \mathcal{C}_n(\iota_i) \end{array}\right\} \qquad \text{for } i \in \{1, \ldots, n\}$$

Figure 3.4: Abstraction-Based Specialization (part 2)

where

- Σ is the signature of the program being specialized.
- Σ_{res} is the signature of the residual program. If abstract tokens (*e.g.*, `even`, `odd`) are being residualized, Σ_{res} differs from Σ (*e.g.*, Σ_{res} contains constants `even`, `odd`). As another example, the signature for the residual version of the Ada program of Figure 1.1 contains constants `zero` and `positive`, but these are not contained in the signature for the source program.
- A is a Σ-algebra with respect to which programs are specialized.
- π controls the degree of polyvariance by specifying which states are to be merged.
- θ is a widening operator used to merge stores.
- R is the set of values from Values[A] that can be residualized.
- *lift* generates code for values $v \in R$, *i.e.*, it maps a residualizable value v to a constant in Σ_{res}.

Each of these components is explained in the subsections below.

3.2.1 Residualization

Specialization transforms a Σ-program to a Σ_{res}-program. The constants from Σ and Σ_{res} may be different. For example, if we allow residualization of *even* and *odd* from previous examples, $\{even, odd\} \subseteq$ Constants[Σ_{res}]. Otherwise, the non-constant operations and tests must be the same in both Σ and Σ_{res}.

The definition of the set of residualizable values R and the definition of *lift* controls whether specialization will preserve the concrete semantics or only the abstract semantics of a program. Specialization preserves the semantics in expressions by using the code generation function $lift : R \rightarrow \text{Constants}[\Sigma_{res}]$. For example, to preserve concrete semantics, we define R_a and $lift_a$ as:

$$\begin{array}{rcll} R_c & = & \{0, 1, 2, \ldots\} & \\ lift_c(n) & = & \texttt{n} & \forall n \in R_c \end{array}$$

If we wanted to preserve the abstract semantics, however, we can define R_a and $lift_a$ as follows:

$$\begin{array}{rcll} R_a & = & \{even, odd, 0, 1, 2, \ldots\} & \\ lift_a(n) & = & \texttt{n} & \forall n \in \{0, 1, 2, \ldots\} \\ lift_a(even) & = & \texttt{even} & \\ lift_a(odd) & = & \texttt{odd} & \end{array}$$

3.2.2 Controlling Polyvariance

We saw earlier (Section 2.2) that abstract interpretation produces a series of states (l_i, σ_i). Maximally polyvariant specialization would produce a specialized basic block for each state. Usually, one does not want maximally polyvariance because this causes a large number of basic blocks (potentially infinite) to be specialized. Each variable in the σ's can have its full range of values and this causes the state explosion. For example, if there was a variable representing integers, a state could possibly be made for each value (0, 1, 2, ...) the integer can have.

To avoid this, specialization is parameterized by a projection operator $\pi : \text{States}[A] \rightarrow$ Indices where Indices is some unspecified set of tokens that depend on the definition of π. Each specialized block will be labeled with index ι, and $\pi(l, \sigma)$ yields the label of the residual block associated with (l, σ). This means the rules for jumps uses π to determine the label of the jump destination in the residual program. We assume

that Indices always includes an index halt and that for all stores σ, $\pi(\mathsf{halt}, \sigma) = \mathsf{halt}$.

The degree of polyvariance is controlled by mapping one or more states to the same index. For example, the following two definitions would yield a maximally polyvariant analysis and a monovariant analysis, respectively.

$$\pi(l,\sigma) \stackrel{\text{def}}{=} (l,\sigma) \quad \forall l \in \text{Block-Labels}[\Sigma], \forall \sigma \in \text{Stores}[A] \quad \text{and States}[A] \subseteq \text{Indices}$$
$$\pi(l,\sigma) \stackrel{\text{def}}{=} l \quad \forall l \in \text{Block-Labels}[\Sigma], \forall \sigma \in \text{Stores}[A] \quad \text{and Labels}[\Sigma] \subseteq \text{Indices}$$

If at least states (l,σ) and (l,σ') map to the same index, the associated residual block must be general enough to handle both σ and σ'.

There are many variations between the two extremes above (maximally polyvariant and monovariant) that can be done by π. For example, one might want to be polyvariant on the live variables in a basic block but monovariant on the dead variables in the block. Several default settings will be used in the full implementation.

3.2.3 Structuring the Residual Program

The specializer incrementally constructs a control-flow graph (program skeleton) $\mathcal{S}$ with nodes $n \in$ Indices to represent the structure of the residual program. Each node is annotated with a mark $m \in \{\circ, \bullet\}$. When a node is unmarked, ι°, it indicates that the associated block is pending specialization. When the node is marked, $\iota^{\bullet}$, the associated basic block has current information flowing into the block. New nodes are automatically unmarked, and the marked nodes can be unmarked if new information is formed for the block (widening). For example, if we have two states $s = (l,\sigma)$ and $s' = (l,\sigma')$ and π maps both states to the same index ι. If state s is encountered first in the specialization, a specialized block will be generated. Later, if s' is encountered, node ι will need to be unmarked so that it will be reprocessed.

The following operations are used to manipulate control flow graphs.

- *mark*$(\mathcal{S}, \iota)$: returns a graph identical to $\mathcal{S}$ except that the ι is now marked

(similarly for *unmark*$(\mathcal{S}, \iota)$).

- *make-arc*$(\mathcal{S}, \iota_1, \iota_2{}^m)$: returns a graph identical to $\mathcal{S}$ except that an arc from ι_1 to ι_2 is added. If the arc is already present, the set of arcs is unchanged. If node ι_2 is not already in the graph, then it is added with mark m. If node ι_2 is already in the graph, then its mark is changed to m.

- *remove-arcs*$(\mathcal{S}, \iota)$: returns a graph identical to $\mathcal{S}$ except all arcs leading out of ι are removed (the set of nodes is unchanged). If node ι is not in the graph, then $\mathcal{S}$ is returned unchanged.

Information that can flow into each block ι in the residual program is collected in a cache $\mathcal{C} \in \text{Indices} \rightharpoonup \text{Stores}[A]$. A cache is a partial function that maps indices to stores. It is a partial function because not all indices are defined for a cache. A block-map Γ_{R} maps a label (index ι) to the associated specialized block.

The control-flow graph $\mathcal{S}$ and cache $\mathcal{C}$ play roles similar to that of the "pending list" and "seen-before set", respectively, in conventional presentations of specializers [20]. The control-flow graph contains the information about which nodes are pending specialization (the unmarked nodes are pending), and the caches contains the program points that have been created. The reason the control-flow graph is used instead of a standard pending list is because of generalization. When a state is generalized, it may have dependent states waiting to be specialized (these would be in the pending list). Since generalization makes the information to a state be less precise, the dependent nodes need to be removed from the pending list because the information flowing in is out of date. The arcs in the control-flow graph give us the necessary dependency, and the marks indicate whether the corresponding cache value is up-to-date.

3.2.4 Specialization Steps

A specialization step begins with a topological sort on $\mathcal{S}$ to find an unmarked node with no unmarked predecessors. By choosing this node, this method ensures that the specializer does not waste time computing nodes that will later be changed. It also prevents non-terminating specialization (provided π and the widening operator θ are chosen appropriately).

After a node ι° is chosen, a specialized version b' of the source block is created using the store $\mathcal{C}(\iota)$ currently held in the cache for node ι. Information is propagated by processing the set of descendent states as follows.

For each state, an index ι_i is obtained and the cache entry for ι_i is updated by merging the new store σ_i' with the previously cached store for node ι (if it exists). The merging is parameterized on a widening operator θ.

The control-flow graph is updated by marking the node ι just processed. All the out-going arcs of ι are removed because the in-coming information can be less precise. For each descendent, the index ι_i is added to the children of ι. If $\iota_i = \mathsf{halt}$, then it is a terminal node and can be marked. Otherwise, if ι_i is not in the cache it is added to the cache. The index ι_i is unmarked, unless the in-coming store had not changed.

3.3 Illustrating ABPS

We illustrate ABPS by specializing the FCL program in Figure 3.5 using the even/odd abstraction A_{eo} defined earlier (see Section 3.2). Two specializations are formed: the first preserves concrete semantics by using lift_c of Section 3.2.1, and the second only preserves abstract semantics by using lift_a of Section 3.2.1.

The first specialization uses the following specialization structure

$$\Xi \;=\; (\Sigma_{num}, \Sigma_{num}, A_{\text{eo}}, \pi, \theta, R_c, \mathit{lift}_c).$$

Source program

```
(b1)
b1: if equal?(x,y) then b2 else b3;

b2: y := 10;
    z := *(z,3);
    goto b4;

b3: x := +(x,2);
    y := +(*(5,x),y);
    goto b4;

b4: if <(y,x) then b1 else b5;

b5: if even?(x) then b6 else b1;

b6: return;
```

Figure 3.5: Sample Program

Since $lift_c$ does not residualize any abstractions, the signature of the source and residual programs are both Σ_{num}. We choose the projection operator π so that it illustrates several concepts at once. Specialization is specified to be monovariant at some blocks and polyvariant (to various degrees) at other blocks. The program has three variables x,y, and z so the store will have the shape $\sigma = [x \mapsto v_x, y \mapsto v_y, z \mapsto v_z]$ (abbreviated $[v_x, v_y, v_z]$).

$$
\begin{array}{lcll}
\pi(b1, [v_x, v_y, v_z]) & = & (b1, [v_x, v_y]) & \textit{polyvariant on } x \textit{ and } y \\
\pi(b2, [v_x, v_y, v_z]) & = & b2 & \textit{monovariant} \\
\pi(b3, [v_x, v_y, v_z]) & = & (b3, [v_y]) & \textit{polyvariant on } y \\
\pi(b4, [v_x, v_y, v_z]) & = & b4 & \textit{monovariant} \\
\pi(b5, [v_x, v_y, v_z]) & = & b5 & \textit{monovariant} \\
\pi(b6, [v_x, v_y, v_z]) & = & b6 & \textit{monovariant}
\end{array}
$$

Thus, the abstract set of indices contains block labels, and pairs of block labels and stores. For widening, we simply define $\theta \equiv \sqcup$, where $\sqcup$ is the least upper bound operator.

Figure 3.6 shows some of the specialization steps that occurred when specializing

Initial configuration

○	$(b1, [0, \top])$	$(b1, [0, \top])$	$= [0, \top, even]$

After step 3

•	$(b1, [0, \top]) \rightarrow b2, (b3, [\top])$	$(b1, [0, \top])$	$= [0, \top, even]$
•	$b2 \rightarrow b4$	$b2$	$= [0, 0, even]$
•	$(b3, [\top]) \rightarrow b4$	$(b3, [\top])$	$= [0, \top, even]$
○	$b4$	$b4$	$= [even, \top, even]$

After step 4

•	$(b1, [0, \top]) \rightarrow b2, (b3, [\top])$	$(b1, [0, \top])$	$= [0, \top, even]$
•	$b2 \rightarrow b4$	$b2$	$= [0, 0, even]$
•	$(b3, [\top]) \rightarrow b4$	$(b3, [\top])$	$= [0, \top, even]$
•	$b4 \rightarrow (b1, [even, \top]), b5$	$b4$	$= [even, \top, even]$
○	$(b1, [even, \top])$	$(b1, [even, \top])$	$= [even, \top, even]$
○	$b5$	$b5$	$= [even, \top, even]$

After step 5

•	$(b1, [0, \top]) \rightarrow b2, (b3, [\top])$	$(b1, [0, \top])$	$= [0, \top, even]$
○	$b2 \rightarrow b4$	$b2$	$= [even, even, even]$
○	$(b3, [\top]) \rightarrow b4$	$(b3, [\top])$	$= [even, \top, even]$
•	$b4 \rightarrow (b1, [even, \top]), b5$	$b4$	$= [even, \top, even]$
•	$(b1, [even, \top]) \rightarrow b2, (b3, [\top])$	$(b1, [even, \top])$	$= [even, \top, even]$
○	$b5$	$b5$	$= [even, \top, even]$

After step 9

•	$(b1, [0, \top]) \rightarrow b2, (b3, [\top])$	$(b1, [0, \top])$	$= [0, \top, even]$
•	$b2 \rightarrow b4$	$b2$	$= [even, even, even]$
•	$(b3, [\top]) \rightarrow b4$	$(b3, [\top])$	$= [even, \top, even]$
•	$b4 \rightarrow (b1, [even, \top]), b5$	$b4$	$= [even, \top, even]$
•	$(b1, [even, \top]) \rightarrow b2, (b3, [\top])$	$(b1, [even, \top])$	$= [even, \top, even]$
•	$b5 \rightarrow b6$	$b5$	$= [even, \top, even]$
•	$b6 \rightarrow$ halt	$b6$	$= [even, \top, even]$
•	halt	halt	$= [even, \top, even]$

Figure 3.6: Specialization Steps for Example Program (excerpts)

Concrete residualization program

```
(b1,[0,T])
(b1,[0,T]): if equal?(x,y)
              then b2
              else (b3,[T]);

b2:         y := 10;
            z := *(z,3);
            goto b4;

(b3,[T]):   x := +(x,2);
            y := +(*(5,x),y);
            goto b4;

b4:         if <(y,x)
              then (b1,[even,T])
              else b5;

(b1,[even,T]): if equal?(x,y)
                 then b2
                 else (b3,[T]);

b5: goto b6;

b6: return;
```

Abstract residualization program

```
(b1,[0,T])
(b1,[0,T]): if equal?(x,y)
              then b2
              else (b3,[T]);

 b2:        y := 10;
            z := even;
            goto b4;

(b3,[T]):   x := even;
            y := +(even,y);
            goto b4;

 b4:        if <(y,x)
            then (b1,[even,T])
            else b5;

(b1,[even,T]): if equal?(x,y)
                 then b2
                 else (b3,[T]);

 b5: goto b6;

 b6: return;
```

Figure 3.7: Example Specialization Using ABPS

with an initial store of

$$\sigma_{init} = [x \mapsto 0, y \mapsto \top, z \mapsto even].$$

These steps illustrate the information propagation aspects by giving the current values of the control-flow graph (left) and the cache (right). The information propagated is the same no matter what choice for *lift* used. This means that the definition of *lift* affects only the code generation, and not the information propagated.

The left column of Figure 3.7 shows the code generated using $lift_c$. A block with index ι is specialized with respect to $\mathcal{C}\iota$. In this example, the only specialization takes

place is the resolution of the condition in $b5$. The assignment $y := 10;$ is residualized instead of being specialized away into the store, because y is generalized to $\top$ at $b4$.

The right column of Figure 3.7 gives the code generated from the steps above using the following specialization structure based on $lift_a$.

$$\Xi = (\Sigma_{num}, \Sigma_{num-eo}, A_{\text{eo}}, \pi, \theta, R_a, lift_a).$$

Σ_{num-eo} is identical to Σ_{num} except that it also contains constants *even* and *odd*.

To illustrate that an abstract trace can lose precision, we show what running the original and residual programs on the store,

$$\sigma = [x \mapsto 0, y \mapsto 1, z \mapsto 4].$$

The original trace and the concrete trace mirror each other. The abstract trace, however, diverges at $b4$. This is because the store at $b4$ is

$$\sigma = [even \mapsto 0, odd \mapsto 1, z \mapsto 4].$$

The concrete values have been lost and so the path cannot be chosen at the conditional.

3.4 Related work

The ABPS work relies heavily on previous works. Consel and Khoo [7] give a framework for abstraction-based partial evaluation of first-order functional languages. Jones [19] provides an elegant language-independent framework for describing partial evaluation and supercompilation. Other elements were inspired by Ashley's mechanisms for controlling polyvariance and generalization in flow analyses [3], Schmidt's presentation of abstract interpretation [27], and Sørensen and Glück's work on generalization for tree-structured data [29].

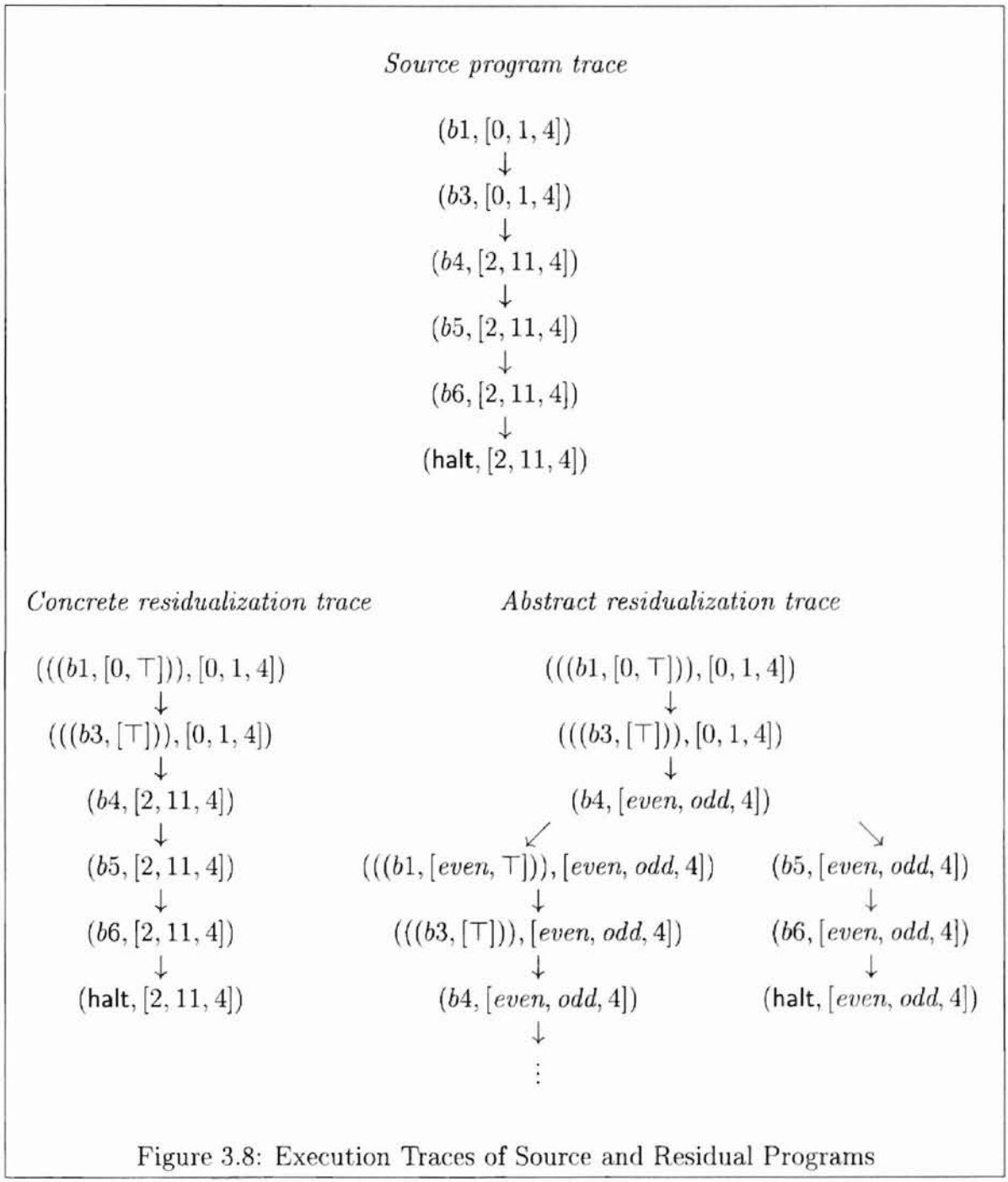


Figure 3.8: Execution Traces of Source and Residual Programs

CHAPTER IV

SCALING TO JAVA

The second stage of this work is to scale the current tools to Java. Many features such as integers and their operators can easily scale to Java from the FCL prototype, but others are more difficult. The FCL prototype does not deal with methods, arrays, dynamic allocation, and classes. All these constructs must be handled to get a full Java specializer running, but many are too difficult to include in this thesis.

4.1 System Overview

Figure 4.1 illustrates the overall view of the ABPS tools encompassed by this thesis. Specializing a method involves several steps. The method to be specialized is in a Java class file. This file is read by the Soot tools and converted to a Jimple representation, discussed in more depth in Section 4.2. The ABPS tools takes the Jimple representation and the definition of abstractions to be used and creates a specialized method in Jimple. This can then be converted back into a Java class file. This figure is similar to Figure 1.3 except the current figure shows the actual flow of information from Java Byte Code back to Java Byte Code. Tools are being created that can also translate to FSV tools. Jimplification is the process of converting Java to a Jimple representation.

The actual specialization process has a few steps. After jimplification, discussed in Section 4.2, the Jimple method needs to be converted to something the specializer

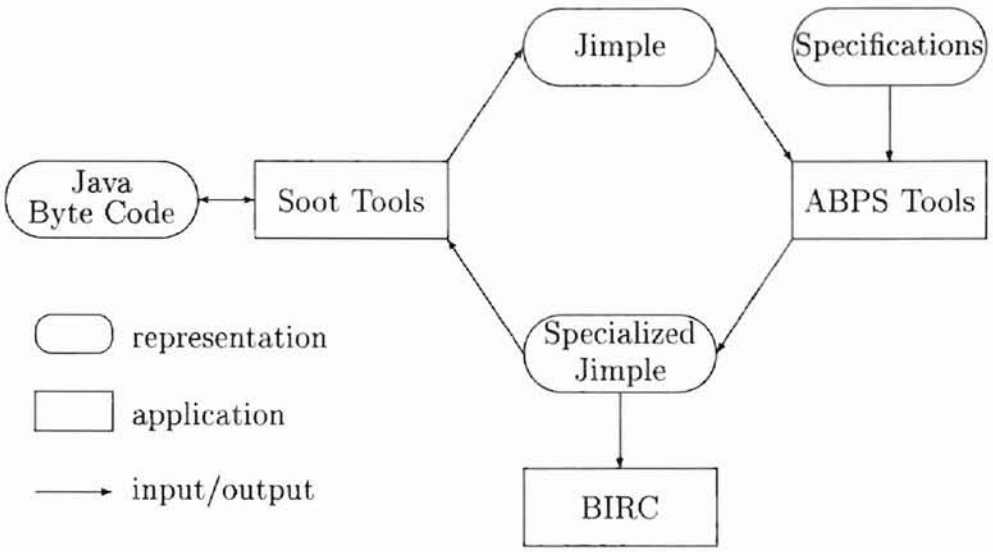


Figure 4.1: System Diagram

can understand and manipulate. For this step, the code is in-lined and broken up into basic blocks. Chapter 5 describes these processes. Chapters 7 and 8 discuss the actual specialization process and the information they require.

4.2 Jimplification

The Soot tools are being developed by Raja Vallee-Rai under Laurie Hendren's advisement at McGill University. The driving force for creating the Soot tools is to simplify analyses and transformations of Java Byte Code. As stated earlier, stack based models, such as the Java Byte Code, are difficult to analyze because an operation's effects might not be fully noticed until many instructions later. The Soot project's aims to reduce these obstacles by creating an intermediate representation of the Java Byte Code that does not rely on the stack.

The Jimple code is obtained by compiling the source file using javac, and then jimplifying the class file. Figure 4.2 presents an example program and the corre-

Java Code

```
public int power(int m, int n)
{
  int result = 1;

  for (;n > 0; n--)
    result = result * m;

  return result;
}
```

Jimple Code

```
public int power(int, int)
{
  int m, n, result;

  m := @parameter0;
  n := @parameter1;
  result = 1;
  goto label1;

 label0:
  result = result * m;
  n = n + -1;

 label1:
  if n > 0 goto label0;

  return result;
}
```

Figure 4.2: Java to Jimple Transformation

sponding Jimple code. A difference is the parameters are held in temporary variables (`@parameter0, @parameter1, ...`) and assigned explicitly to the formal parameters appearing in the source method. The Java Virtual Machine does a similar process when it executes a method. One cannot see from the example that expressions in the original code do not get recreated exactly in the Jimplified code. This happens partially because some of the information is lost in the transformation from Java source code to Java Byte Code and then to Jimple code and because the Jimple code is in three-address code [1].

4.2.1 Jimple Structures

The three key structures that are used in Jimple are methods, statements, and expressions.

4.2.1.1 Methods. Specialization works at the method level currently. This is to reduce the amount of analysis that is required and because the process of specializing whole systems is very difficult and beyond the scope of this thesis. Jimple methods contain five key components: method name, parameter types, return type, local variables, and the Jimple statements. The first two components are the method signature. The parameter types, local variables, and Jimple statements are all lists.

4.2.1.2 Statements. Statements are used to represent many of the constructs that occur in a method body. These include assignments, **goto**'s, **if**'s, and **return**'s. Statements also include constructs for handling exceptions and thread synchronization. Each of the handled statements has a particular structure.

- Assignment statements have two operands. The left operand represents the left side of an assignment and the right operand represents the right side of an assignment.

- **goto** statements have a pointer to the statement targeted by the **goto**.

- **if** statements have a conditional expression and a target statement. The target is the statement the **if** jumps to if the conditional expression evaluates to `TRUE`. If the expression evaluates to `FALSE`, the control goes to the next statement in the list. This is not quite like the **if**'s in FCL, which holds the destination of the false branch.

- **return** statements take two forms. The first is when the return type is `void`. In this case, the statement has no operands. The other case is when the return type is not `void`, the statement then has one operand, the value to be returned. This value can be a constant or as a variable.

Java Source	Byte Code (as Jimple)
	`if (y == z) goto label1;`
	`x = 0;`
`x = (y == z);`	`goto label2;`
	`label1:`
	`x = 1;`
	`label2:`

Figure 4.3: Test Example

4.2.1.3 Expressions. Expressions include constants, variables, operators, and method invocations.

- Variables and constant operators are basic expressions. Variables get their value from the store while constants contain their own value. All other expressions use variables and constants as the terminal end of the expression.

- Operators are expressions that are either unary or binary operators. Unary operators, such as not and negate, have a single operand. Binary, such as $+$, $-$, &, and $<$, operators have a left and right operand. Operators are used in both operators and test. The key distinguishing feature is the ones used for tests are operators that are used for comparison and Java places these in an **if**. This occurs even when the test is in an assignment statement, like that in Figure 4.3. Java does this breakdown so that all comparisons are the conditional expression of an **if**. When it is in an assignment statement, such as that in the same figure, the test is put in an **if** and then branches on the outcome to the correct assignment.

- Method invocations are expressions that invoke a method and yield the value returned by the method. These keep a pointer to the method so that the name, parameters, *etc.*, are easily accessed.

Java Byte Code	Plain Jimple
	`m := @parameter0;`
	`n := @parameter1;`
`0 iconst_1`	`op0 = 1;`
`1 istore_2`	`result = op0;`
`2 goto 12`	`goto label1;`
	`label0:`
`5 iload_2`	`op0 = result;`
`6 iload_0`	`op1 = m;`
`7 imul`	`op0 = op0 * op1;`
`8 istore_2`	`result = op0;`
`9 iinc 1 -1`	`n = n + -1;`
	`label1:`
`12 iload_1`	`op0 = n;`
`13 ifgt 5`	`if op0 > 0 goto label0;`
`16 iload_2`	`op0 = result;`
`17 ireturn`	`return op0;`

Figure 4.4: Comparison of Java Byte Code and Jimple

4.2.2 Jimplification

The process of jimplification includes many steps, from reading byte code to producing the final Jimple code.

1. Read Java Byte Code

   The first step of jimplification is to read the Java Byte Code from the class file. This is done with the Coffi component of the Soot tools. Coffi is a very low level representation of Java Byte Code. A Byte Code representation of the power method from Figure 4.2 is on the left side of Figure 4.4.

2. Remove Stack

   For the next step, Jimple removes the stack and converts the Coffi representation to a Jimple representation. A simple analysis finds stack positions of

the operands for each Java instruction. This is possible since the Java Virtual Machine's verifier enforces that an operator works with the same positions in the stack each time it is executed. This is done ensure that the stack has a fixed maximum size. Each instruction is changed to a Jimple statement. The references to the stack are changed to actual stack variables. This creates the first Jimple representation, but it is without types and leaves a large number of variables sitting around. The right side of Figure 4.4 shows what the Jimple code looks like after the initial jimplification.

3. Apply Types

   After the initial jimplification, types are applied to the code. This starts out with the initial types of values and iteratively works to a fix point.

4. Constant and Expression propagation

   The next step is constant and expression propagation. Constant propagation is very similar to that found in optimizing compilers. The expression propagation is similar also, but it must keep the code as three address code. The expression propagation is used to reduce the number of Jimple statements used to represent a section of code.

5. Pack Variables

   The final step is to pack the local variables. During the different processes, the number of local variables can explode. To correct this problem, they are packed as a final step. The packing algorithm is a common one used for register allocation.

After all the steps of jimplification, the resulting code looks similar to that found on the right side of Figure 4.2.

4.2.3 Features

There are many useful features in the Soot tools that are used in the implementation of ABPS. These are described in the following sections.

4.2.3.1 Class Manager. Soot contains a class manager. A class manager keeps track of all classes loaded so that access can be quick if a class is required more than once. Also, the class manager is responsible for reading a class.

4.2.3.2 Code Creation. Jimple has an easy mechanism for creating, accessing, and modifying structures used to represent classes, methods, statements, and expressions.

- Classes

  Classes for Jimple is the same for Soot, called Classes are presented in Jimple by the Soot representation `SootClass`. The constructor for `SootClass` is called to create a new instance. The arguments are the name of the new class and the access flags, *e.g.*, `public`, `abstract`. After creation, the super class and other features of a class can be changed. It is usually wise to add the class to the class manager so that it can be managed properly. Also, there are methods to make sure that the class is written out to the file system.

- Methods

  Creating a method is similar to creating a class, except that the method name, parameters, return type, and access flags are passed to the constructor. After this, the method is added to the correct class. The representation provides methods to get the statement list, add to the statement list, and to add local variables to the method.

- Statements

  Statements are created by calling the constructor for the statement representation class. For example, the call `Jimple.v().newGotoStmt(target)` creates a new goto statement. The part `Jimple.v()` is a static method that is the only way to get an instance of the `Jimple` class. The `target` is the target statement of the `goto`. Statements that have been created can be added to a statement list by calling the `add` method of the statement list, for example `stmtList.add(stmt)`.

- Expressions

  Expressions are always a part of a statement. Expressions are constructed and accessed like statements.

4.2.3.3 Switches. A common design pattern for an AST is called a visitor pattern. A visitor walks over an AST and performs some computation. Sometimes it is difficult to write visitors because of the many cases that must be included and the difficulty in matching a node with the correct functions. Jimple gives a graceful method to switch between different nodes, Switch classes. These define a method for each type of node. These different switches are divided into switches for statements, expressions, and types. Each node to be switched implements an apply method that takes an instance of a switch that corresponds to the category of the node. The apply method calls the corresponding method in the switch. For example, an AddExpr will call the caseAddExpr method in a switch. In the default switches, all methods call a default method that does nothing.

To implement a switch, one extends the default switch class and adds the methods for the nodes to be handled. There are some difficulties with passing arguments to the methods, however. Each method only takes the node as an argument. A newer

```
final List l = new List();
Stmt s = ...;
s.apply(new StmtSwitch() {
          public void caseAssignStmt(AssignStmt s)
          {
             // Code to handle assignment
             l.add(...);
          }

          // Other methods can be added
        });
```

Figure 4.5: Anonymous Class Example

feature of Java allows a nice way around this, called anonymous classes. Anonymous classes are basically a local class. One instantiates a normal class but adds methods to the class. An example using a StmtSwitch can be seen in Figure 4.5. With an anonymous class, variables defined when the class is instantiated are allowed in the class, such as the `List l` in Figure 4.5. This allows one to pass values to a method which instantiates the anonymous class and does the necessary analysis.

4.2.3.4 Utilities. The Soot tools require a set of useful utilities that are also implemented by the same people that implement Soot. The three most useful utilities are lists, maps, and iterators; all of these are interfaces. In Java, an interface defines the methods that must be implemented by a class.

- List

  A list has methods to access, add, and remove elements. There are also methods to search a list. Actual implementations of lists use arrays, vectors, and linked list. This way, one can pick the implementation that best fits the use. Each one uses the same interface so that code using the list does not need to know

the particular implementation of the list.

- Map

  A map associates an object to another object. There are methods to put and get associations, and to move through the values.

- Iterator

  An iterator goes through a list sequentially. There are three methods in an iterator: a method to check for more entries, a method to get the next entry, and a method to remove an entry. As with the other utilities, there are specific iterators for the different implementations of lists and maps..

4.3 Breakdown of the ABPS Tools

The ABPS tools can be broken down into four distinct parts.

1. Jimple Interface, described in Chapter V.
2. Data Structures, described in Chapter VI.
3. Specializer, described in Chapter VII.
4. Specifications, described in Chapter VIII.

CHAPTER V

JIMPLE INTERFACE

In Jimple, statements are held in a linear list. Jumps, like **if** and **goto**, are handled by having a pointer to the target statement. This works well when the code is being processed linearly, but ABPS collects information for blocks of statements. A basic block is therefore used by ABPS as the basic control structure. The control flow graph ABPS uses is maintained in basic blocks in a block map. ABPS also creates a group of basic blocks for the residual code and this is used to create the residual statement list for Jimple.

5.1 Control Flow

The control flow graph is a graph with one entry point and at least one exit point. Each node in the control flow graph is a basic block. These blocks are combined into a block map, which maps indices to the corresponding block. A statement list created by Jimple contains the information needed to make a control flow graph. Between the elements of the statement list, there are arcs corresponding to the jumps in the method. The information about the arcs and the entry and exit points are used to create the control flow graph and basic blocks.

5.1.1 Labels

The name of a basic block is a label, represented by an index. There are two types of indices used in ABPS. The first is `Index` class which gives the basic functionality of an index. The second is `StoreIndex` class which adds store information into the index.

5.1.1.1 `Index`. For a simple index, all that is kept is an integer. This allows for quick comparisons of an index, since integers have a quicker equality operation in Java than other types. `Index` has a constructor, equality operator, and a hash code generator. The equality operator checks types and then the index number. The hash code generator returns the index value as the hash code. The other method in `Index` is a method that returns the base index. In `Index`, the base index is itself. For other indices, the base index returns an instance of `Index` with the index number. This allows one to get the index other indices are based upon. For example, a residual block's index could be a combination of the original index and a store. If one needed to find the original block, one can get the base index of the residual block's index, which is the original block's index.

5.1.1.2 `StoreIndex`. `StoreIndex` is an extension of `Index`. It has the same methods, but the constructor takes either an integer and store or an `Index` and store. This is used to distinguish between different residual basic blocks that come from the same source basic block when there are differences in the input store.

5.1.2 Basic Block

A basic block is a node in the control flow graph. Each basic block consists of four parts: a list of predecessors, an index, a list of statements, and a list of successors.

Line Number	Explicit Jump	Implicit Jump
1 Predecessors	{2,0} ->	{1} ->
2 Index	1:	2:
3 Statement		result = result * m
4 Statement	if n > 0 goto 2	n = n + -1
5 Successors	-> {2,3}	-> {1}

Figure 5.1: Basic Block Examples

- The list of predecessors is used to know where control can come from. This helps for analysis on reverse control flow.

- The index gives the block a name (as described in the previous section) and can be thought of as a label. This is used to reference basic blocks in the predecessor and successor lists and for block maps.

- The statement list stores the list of statements the block represents.

- The list of successors is used to decide where control can go from this block.

A couple of example basic blocks are shown in Figure 5.1. The left column has line numbers and the type of information on the line. The other two columns are actual examples of a basic block from the power function from earlier examples. A basic block can have an explicit or an implicit jump. An explicit jump occurs when the statement list ends with a statement that jumps to some other point, such as **goto**'s and **if**'s. The successor list just reflects the successive block. In the case of an **if**, the first successor is the index of the block when the conditional evaluates to `TRUE`. The second is when it evaluates to `FALSE`. This is seen in the middle column of Figure 5.1. An implicit jump occurs when there is no jump at the end of the statement list. Control falls through to the next block. The next block is gotten from the successor list, which would have only one value. The right side of Figure 5.1 is an example of an implicit jump.

```
Entry: 0                          Exit: 3
{}->                              {1}->
0:                                2:
m := @parameter0                  result = result * m
n := @parameter1                  n = n + -1
result = 1                        -> {1}
goto <unnamed>
-> {1}                           {1}->
                                  3:
{2,0}->                           return result
1:                                -> {}
if n > 0 goto ?
-> {2,3}
```

Figure 5.2: Block Map Example

5.1.3 Block Map

The block map associates each index to a basic block. Figure 5.2 shows the block map for the power program of Figure 4.2. Each basic block contains the predecessor and successor information, so the block map is used to retrieve the predecessor and successor blocks. The block map structure keeps the entry and exit points, the local variables, and methods to retrieve and update information. The local variables are used to help reconstruct a method from the blocks.

The block map has a few features that determines the output created. The first is a flag to create a block level view or a statement level view. The block level creates blocks in the normal fashion where there can be any number of statements in a block. The statement level view forces each block to have only one statement. This gives a finer granularity of the statements but adds more overhead to keep track of and to move between blocks. The other feature is to produce a control flow graph that is in hammock form or not. Hammock form is where there is exactly one entry and one exit point. Certain types of analyses, such as slicing, require the graph to be in hammock form. Currently, the specializer does not make use of statement level mock

maps or hammock structure, however, the slicing component of the Bandera tool set (see Section 1.5) does.

5.2 Input Processing

There are two key steps for the input processing needed to be done before specialization: inline methods where possible and convert the statement list to a block map.

5.2.1 In-lining

It was decided to inline methods when possible to reduce the code complexity. A partial reason was to remove method calls, reducing the overhead to do the call. In-lining cannot be done for all methods. In Java, most methods are done in Java Byte Code but some are in code native to the machine it is running upon. This code cannot be in-lined. Because of the limited scope of the thesis, in-lining is currently only performed for static methods in the same class. For this reduced case, in-lining is not too difficult and follows a few easy steps.

1. Find a method call to a method that can be in-lined.

2. Replace the parameter references in the method to be in-lined with the actual expressions with which the method is called. For example, if `abs(b)` is a method call to be in-lined, the statement `x = @parameter0` in the in-lined method is replaced with `x = b`.

3. Replace **return**'s with an assignment of the return expression to the variable the method takes place in and a **goto** to the statement after the call. For example, if the method call is `a = abs(b); a = 2 * a;` then a `return x;` in the in-lined

method is replaced with `a = x; goto <a = 2 * a;>`. The `<a = 2 * a;>` is a pointer to the statement contained in the <>'s.

4. Add the code to the statement list. This can be added at the point of the call to reduce **goto**'s, or at the end. The current version places the code at the end only a single method call is required to add statements to the end of a list.

5. Replace the method call with a **goto** to the start code just inserted. This is not necessary, as mentioned earlier, if the code is inserted at the point of the call.

6. Repeat until there are no more method calls that can be in-lined.

5.2.2 Creating the Block Map

Jimple has a set of tools to create a control flow graph on the statements. This is used to determine the start and end points of basic blocks. Basic blocks take the statements and are put together and placed in the block map. The successors are updated during this process. Finally, after all blocks have been created, the list of blocks is processed to update the predecessor information.

5.3 Operation In-lining

The abstraction classes contain the information needed to implement the primitive integer operations, but they do not plug into Jimple easily. This problem is circumvented by having the abstraction classes inherit from the Jimple constant class. This solution allows Jimple to act as though abstractions are constants. This works well for tools the know about abstractions, but in many cases, especially for outside tools, the tools will not be extended to handle this functionality. To cope with this, the

```
x = y + 1

if (y == EVEN)
  x = ODD;
else
  if (y == ODD)
    x = EVEN;
  else
    x = CHOOSE(EVEN, ODD);
```

```
if (y == EVEN) goto label1:

if (y == EVEN)
  tmp = TRUE;
else
  if (y == ODD)
    tmp = FALSE;
  else
    tmp = CHOOSE(TRUE, FALSE);
if (tmp) goto label1;
```

Figure 5.3: Operation In-lining Examples

abstractions can be identified with an integer constant and the abstraction operations can be in-lined using regular Java code.

The integer constant used to identify the value of an abstraction can be obtained by the bit vector representing the set of tokens the abstraction represents. Section 6.2 discusses how these values are assigned and used. To in-line the abstraction operations, checks on these values are done and the corresponding result is used. Figure 5.3 shows a couple of examples of operation in-lining. The top lines represent the original code while the bottom section is the resulting code. `EVEN` and `ODD` used in the resulting code represent the integer value for `EVEN` and `ODD`. The expression `CHOOSE(...)` is an expression for nondeterministically choosing between the choices given and extends Jimple's expression class. This allows it to be incorporated into Jimple, much like the abstraction classes. The `CHOOSE` expression is for the case when $\top$ flows through the result in the residual code. The down stream tools that build model checker inputs implement the `CHOOSE` expression.

In Figure 5.3, the left side illustrates what happens when an operation occurs in an assignment statement. The abstraction being used in the example is the even odd abstraction. If y is known, the operation in-lining does not need to happen. For example, if y is *even*, then the result would be $x = odd$ and no operation in-lining is done. But if y is $\top$, then the inline happens. The code works by checking the possible

values of y and then doing the correct result. If both values are unknown, let us say $x = y + z$ and both y and z are $\top$, then both the values of y and z will need to be checked to ensure that x is assigned the correct value. The right side of the figure demonstrates how a conditional is handled. The result of the test is assigned to a temporary variable and then this is used to make the correct jump. An expression where both sides are unknown is handled as in the assignment statements.

This helps tools that do not know about abstractions to use the result from ABPS. The `CHOOSE` expression can be converted into a method call that decides between the choices.

5.4 Output Processing

Currently, the output processing consists of changing the block map back to a statement list. This is done by combining blocks that have implicit jumps where possible and basically putting the statements in the general order that they appear in the control flow graph. When implicit jumps cannot be placed together, a **goto** is inserted to ensure the jump to the correct block. The main requirement for putting the statements back together is that control flow is maintained.

CHAPTER VI

DATA STRUCTURES

This chapter summarizes the implementation of tokens, sets of tokens, lookup tables, states, stores, caches, control flow skeletons, and specialization values.

6.1 Tokens

Tokens are used to represent a single abstract value. For example, **even** from the even odd abstraction is a token. An actual token consists of a string, for the name, and an identification number. The number is used to do any comparisons and calculations using the token. The name is only used to print the token so it is easier to understand. The number is used for the calculations for speed because checks on strings are slow. Different tokens can use the same identification number as long as the tokens will not be mixed together.

6.2 Token Sets

A `TokenSet` represents a set of tokens. `TokenSet` uses a bit set to represent the set. This allows the common set functions, such as union, intersection, and difference, to be implemented using bitwise operators such as `or`, `and`, and `xor`. Using bitwise operators makes the manipulation much faster than iteratively going through a list to do the operators. An example of the negative zero positive set can be seen in Figure

Negative Zero Positive Set

Tokens

Token	ID Number
negative	001
zero	010
positive	100

Set Representations

Bit Set	Actual Set
000	{}
001	{**negative**}
110	{**zero**, **positive**}
101	{**positive**, **negative**}
111	{**negative**, **zero**, **positive**}

Figure 6.1: `TokenSet` Example

6.1. The left column is the table of tokens and their identification number. The right column represents a few bit set values and their corresponding sets.

A `TokenSet` takes an array of tokens representing the universe of tokens that the set can contain. It goes through the array and assigns an unique identification number to each token. The identification number for the token is the bit position that represents the token. Each different token set is given its own identification number. This is used to check to see if the sets represent the same universe. Because a token set has a unique identification number and an array of possible tokens, `TokenSet` has a function that returns a new empty set from the same universe as the original set.

6.3 Lookup Tables

A lookup table is a table that associates a value with a particular key. There are methods to add, update, and lookup values associated with keys. Lookup tables are similar to maps in Section 4.2.3.4, but hash codes are not used. The original ABPS prototype used lookup tables and many of the uses did not change for this ABPS implementation so they are used instead of maps in some cases. For updates and lookups, a lookup table iterates through the table until it finds the key and then does the necessary operation. The add places a new key at the end of the table. There are also functions to get a `Vector` of keys or values.

6.4 States

A state is a pair consisting of an index and a store. It it used to represent the current state in a program's execution by having the position in the program's execution, the index, and the current value of all the variables, the store.

6.5 Store

The store is a lookup table that maps a variable to its value. Because the current implementation does not deal with objects, this can be a simple mapping. In the future, when the implementation is expanded to include objects, the store will have to have additional functionality to deal with objects and pointers. A store keeps an array of variable names and an array of their associated values. A store has many of the same methods as a table. An additional function is a limit function which takes a list of variables and returns a new store with only the variables that are in the list. A little functionality is added to the store also. Because variables are added at the beginning of a program and their values need to be initialized, a value initializer can be specified. This takes the type of the variable and returns the corresponding initial value. Also, the equality method is modified to speed up an equality check. It uses only the list of values because it assumes that the two stores have the variables in the same order. This is the case with ABPS because the store has all variables added at the beginning of specialization and all copies of the store are copies of the original. Also, the limit functions keeps the variables in the same order, ignoring the order of the list that is used to limit the store.

6.6 Cache

`Cache` is the implementation of $\mathcal{C}$ for ABPS ($\mathcal{C}$ was defined in Section 3.2.3). It is a simple lookup table that associates stores to indices.

6.7 Control Flow Skeleton

In many partial evaluators, a pending list is kept (see Section 2.1.2). A problem with using a simple pending list is when information is updated for a state that modifies states already in the pending list. The updated state may need to be evaluated again with the new information and this can flow down the control flow graph. This state can be added to the pending list, but this may lead to problems since the other states in the list can be evaluated before the updated state. If the information from the updated state flows through the graph and makes the information at a state in the pending list obsolete, this needs to be known or extra computation is done that may not be used.

A control flow skeleton solves this problem. When the next node is required, the control flow skeleton does a topological search to find the first node that is not up-to-date. The topological search ensures that all nodes between the starting node to the current node is up-to-date. The actual structure of the control flow skeleton is a vector of nodes where each node contains the index of the basic block it represents, whether it is up-to-date, and the children node positions in the vector (as in Section 3.2.3). A node is up-to-date if it is marked, shown with the symbol $\bullet$, and not current if is unmarked, shown with the symbol $\circ$. Each new node is added onto the end of the vector. This makes the topological search become a sequential search for the first unmarked node. The cost might be improved with a more clever data structure.

The control flow skeleton reuses the code from the FCL prototype with only a

Skeleton Index	Block Index	Mark	Children*
Skeleton: Example 1			
0	0	•	1
1	1	○	2
2	2	•	1
Skeleton: Example 2			
0	0	•	1
1	1	•	2, 3
2	2	○	1
3	3	○	-

* Using the skeleton index.

Figure 6.2: Skeleton Example

small modification of the package and class names. It has the methods to add nodes, create arcs, and changed whether a node is marked or unmarked. It also has a method to get the next node to evaluate and it has a few methods that are the same as the steps needed by ABPS. For example, when a node is added to a parent, a node has to be created, unmarked, and placed at the end of the list of nodes. Also, the child needs to be added to the children list of the parent node. All these steps are included in a method `addDescendent`.

In Figure 6.2, the left most and right most column numbers are indices into the skeleton, but the block index column uses the indices associated with basic blocks. The two examples are successive updates and demonstrate how the skeleton can change. In example 1, there are only three nodes with node 1 getting evaluated. In example 2, a new node is added that is a child of node 1 and the children of node 1 get unmarked. This unmarks 2 because it is a child of 1 (following the definitions in Figure 3.4).

6.8 Specialization Values

The value returned from an expression is a specialization value. A specialization value originally consisted of the abstract value the expression evaluates to and a residual expression. This was used to decide what would get residualized by the specializer. If the abstract value can be residualized, it gets residualized. If, however, it cannot be residualized, the residual expression gets residualized.

This works well for most situations, but if the operations of the abstractions need to be in-lined (discussed in Section 5.3), more information needs to be returned. This information includes the statements needed to represent the operation. This prompted the creation of two other specialization values, one for assignments and one for if statements. In all cases, the value of the expression is returned, but the other data included in the specialization value depends on the context.

6.8.1 Expression Specialization Values

An expression specialization value contains the standard information of the expression's value and a residual expression.

6.8.2 Assignment Specialization Values

An assignment specialization value contains the standard value for the expression plus two lists. The first is the list of statements that is created to inline the expression. These look much like the examples in Figure 5.3 from Section 5.3. The second list is a list of boxes. A box is basically a Java pointer so that a value can be changed without recreating the whole statement or expression. These boxes point to the left hand side of the assignments that use the assigned variable. The exact variable is not

known during the creation of the code, so it must be plugged in by the assignment statement. The list of boxes gives an easy way of doing this. For example, the assignment $x = y + 1$ is being evaluated and the operation needs to be in-lined. The variable x is not passed down to the expression specializer so it has no knowledge of what the actual variable is. It therefore passes a list of boxes back where the variable needs to be plugged in and the assignment statement plugs the x into these boxes.

6.8.3 If Specialization Values

The if specialization value contains a large amount of information besides the value of the conditional. Like the expression specialization value, it keeps the residual expression of the condition. Next, it keeps a list of statements to handle in-lining of the test expressions and a list of boxes. It also keeps a separate store for each branch.

The if specialization value is used even when operation inline is not done. It is used to keep the stores for the different branches. This works by having the different pieces of code to be empty. The stores are updated with a new value of a variable if it can be deduced. For example, imagine the value of `x` is $\top$ in a store when the expression `x == 0` is encountered. If the abstraction for `x` is the Integer Abstraction, one if specialization value would be returned. In the specialization value, the `TRUE` store would be $\sigma[x \mapsto \top]$. The `FALSE` store would be unchanged. This is because when `x` evaluates to `0`, the true branch is followed and `x` must be `0` to go down the branch, but it remains $\top$ for the false branch.

CHAPTER VII

SPECIALIZER CORE

The specializer core does the necessary steps to specialize methods, basic blocks, statements, and expressions. Individual expressions are handled in the signature and abstractions. These are discussed in more depth in Chapter 8. Each level of specialization proceeds through the same general steps. First, the structure is separated into individual components. Each of these components are then specialized. Finally, the specialized components are combined.

7.1 Method Specialization

The steps to specialize a method are very similar to the steps in standard partial evaluation. These can be broken up into three separate section sections, consisting of initialization, specialization and creation of residual basic block, and updating the cache and control flow skeleton. The first step is done once; the second and third steps are done repeatedly.

7.1.1 Initialization

The initialization step consists of the initialization of the different structures that are used during specialization. The first is the creation of the initial store. This consists of adding all the local variables to the store and initializing the parameter values to

```
initialize(store);
state = new State(blockMap.initialLabel, store);
index = pi(state);
residualBlockMap.initialLabel = index;
C.update(index, theta(store, C.lookup(index)));
S.setStart(index);
```

Figure 7.1: Specialization Steps: Initialization

those that are passed into the specializer. Next, the initial state is created from the initial label of the block map and the initial store. Then, the residual index is created using the operator π (pi) (as defined in Section 3.2.2). This is used to set the initial label of the residual block map. Next, the store in the cache for the initial index is updated. Finally, the start of the control flow skeleton is set to the initial residual index. This can be seen in the pseudo-code in Figure 7.1.

7.1.2 Specialization and Creation of Residual Basic Block

After initialization is completed, the specializer repeatedly specializes blocks (step 2 above) and updates the cache and skeleton (step 3 above) until there are no unmarked nodes in the skeleton. During each cycle, the next index is retrieved from the skeleton. The index is used to look up the corresponding store from the cache. Next, the basic block labeled by index is retrieved from the block map. After that a new, empty residual basic block is created. This is passed to the basic block specializer along with the store. The specializer then adds residual statements to the new basic block, and the completed block is placed into the residual block map. The specializer also returns a list of states that are reached from the specialized basic block. Figure 7.2 represents the code for this step.

```
index = S.next();
store = C.lookup(index);
bb = blockMap.get(index.getOriginalLabel());
currentBB = new BasicBlock(index);
states = evalBasicBlock(bb, store, currentBB);
residual.put(index, currentBB);
```

Figure 7.2: Specialization Steps: Specialization and Creation

7.1.3 Updating of the Cache and Skeleton

The third step is the updating of the cache and control flow skeleton. First, the index is marked in the control flow skeleton. Next, the states resulting from processing the current basic block are processed. If the store (σ) in a state is not equal to the store (σ') associated with the state in the cache, a new store (σ_{new}) is created by merging σ and σ' using θ. The cache is updated with σ_{new} and an arc is made from the parent (the current basic block) to the child (a successor) in the skeleton. The child is set to unmarked. If the stores are the same, then an arc is made from the parent to the child and the child is left as marked or unmarked. The code for this is in Figure 7.3.

7.2 Basic Block Specialization

Basic block specialization requires specializing each statement in the block. A residual statement is created for each source statement during specialization, and the residual statement is added to the residual basic block. The statements from the source block are specialized in succession. Finally, when the last statement is reached, the successors are determined and set. For each jump construct (**goto**, **if**, and **return**) the successors of the residual basic block are determined using π and θ.

```
S.mark(index);
while (states.hasNext())
{
  state = states.next();
  if (!state.getStore().equals(C.lookup(pi(state))))
  {
    store = theta(state.getStore(), C.lookup(pi(state)));
    C.update(pi(state), store);
    S.makeArc(index, pi(state), false);
  }
  else
    S.makeArc(index, pi(state));
}
```

Figure 7.3: Specialization Steps: Updating the Cache and Skeleton

7.2.1 Goto's

These can be implicit or explicit. Implicit **goto**'s are in blocks that do not end with any type of jump and just has a successor to go to. Explicit **goto**'s are in blocks that have an actual **goto** at the end of the block. In either case, the basic block contains the successor and this is used to calculate the successor for the residual block.

7.2.2 If's

if's are one of the key points of specialization. If the conditional can be determined during specialization, it can be turned into a **goto**. During the specialization of an **if**, the possible targets are returned. A target is a state, which consists of the label or index of the original target and the current store. If the conditional can be determined, only the one target is returned. If the conditional cannot be determined, both the `TRUE` and `FALSE` targets are returned. In the first case, the target is used to calculate the successor and nothing is residualized for the statement, thus creating

an implicit **goto**. In the latter case, both targets are used to calculate successors and a new **if** statement is created.

7.2.3 Return's

All **return**'s must be residualized. There are, however, no successors to be set.

7.3 Statement Specialization

There are two results from specializing a statement. The first is the creation of a residual statement. The second is the return of any target blocks if the statement is a jump.

7.3.1 Assignment Statement

Assignment statements are made of two parts, a left and right hand side. Because currently only local variables that are integers are supported, the left hand side is just a variable. The right hand side, however, contains an expressions that needs specialized. The result of specializing an expression is a specialization value. There are two kinds of specialization values that apply to assignments: one for normal operation (an expression specialization value) and one for operation in-lining (an assignment specialization value). If the specialization value is one for normal operation, then the value of the expression is checked to see if it can be be residualized. If so, the assignment is residualized with the right hand side the same as the value. If not, the right hand side is the residualized expression. For example, imagine the left hand side being x and the specialization value of $< POS, y * 2 >$, where the POS is the value of the expression and the $y * 2$ is the residual expressions. If POS is residualizable then the residualized assignment would look something like $x = POS$.

If, however, POS is not residualizable, then the residualized assignment would look something like $x = y * 2$. More on expression specialization can be seen in Section 7.4. The case when the specialization value is one for operation in-lining is explained and demonstrated in Section 5.3.

7.3.2 Goto Statement

Goto statements are a special case. They do not get residualized, but are made implicit in the residual block. This allows the function that creates a new method from the block map to put **goto**'s wherever they are needed. Specializing the **goto** statement simply returns the target index.

7.3.3 If Statement

There are two cases for an if statement: the conditional can be evaluated to `TRUE` or `FALSE`, or the conditional evaluates to an unknown value. For the first two cases, specialization proceeds in the same way. If the condition evaluates to `TRUE` or `FALSE` then the **if** statement is not residualized, but instead it is converted to an implicit goto. The specializer returns the target index of the `TRUE` or `FALSE` branch depending on the value.

If the conditional evaluates to an unknown value, the residual expression returned in the specialization value is used for the conditional of the **if**. The specialization value returned in the second case is one for **if**'s (discussed in Section 6.8.3). This new **if** statement is residualized and the `TRUE` and `FALSE` targets are returned. The additional code in the specialization value is also residualized in its correct place.

The stores for the `TRUE` branch and the `FALSE` branch can be different by using conditional constant propagation. Conditional constant propagation is where the

form of a conditional expression is used to propagate values down different branches. For example, imagine that we had the expression $x == 0$ as a conditional and x is unknown. For the execution to follow the `TRUE` branch then x must equal 0, therefore 0 is propagated as the value for x down the `TRUE` branch. Down the `FALSE` branch, the value of x can be anything but 0, the value of x is left as unknown.

7.3.4 Return Statement

There are two types of return statements in Java. The first is a **return** that returns `void`. The second is a **return** that returns the value of the return expression. In the first case, the **return** is residualized. In the second case, the expression is specialized and residualized (along with the return) using the same process as with the assignment statement. An empty set of targets returned.

7.4 Expression Specialization

There are a wide range of expressions used in Java, but all of these can be broken down into two categories. The first are operators, *i.e.* $+$, $-$, $*$. This group also includes constants, variables, and method calls. The second group are tests, which consist of operations that return a boolean value, *i.e.* $<$, $==$, and $!=$. All these are dependent on the different abstractions so they are implemented in the Σ and the individual abstractions.

7.4.1 Operators

Operators return a specialization value, first mentioned in Section 6.8. Let us look at a few examples of specialization values for expressions. Examples of assignment and if specialization values are found in Section 5.3.

- Constants

  If one has a constant, such as 3, the value is 3 and the expression is also 3. The specialization value would be $< 3, 3 >$.

- Variables

  If one has a variable x then the value would be $\sigma(x)$ and the expression would be x. The specialization value would be $< \sigma(x), x >$.

- Operations

  These have subexpressions which are specialized individually and the specialization values are used to make the operation's specialization value. There are three cases of operations, one where the value is residualizable and two where the value is not residualizable. The difference between the two is whether the abstract operations are in-lined or not.

  - Value is residualizable

    An example of this is the expression $x + 2$ and x evaluates to 2. The value would then be 4 and the residual expression would be 4, thus making the specialization value to be $< 4, 4 >$.

  - Value is not residualizable and operations are not in-lined

    An example of this is the expression $x + 2$ and x evaluates to a value that cannot be residualized, such as $\top$. The value of the expression is $\top$ and the residual expression is $x + 2$. The specialization value is $< \top, x + 2 >$.

  - Value is not residualizable and operations are in-lined

    An example is similar to that above, but the specialization value is one for assignments. It would look something like $< \top, code, [] >$. The code is

similar to that in Figure 5.3 and the [] is the list of boxes that need to be filled with the variable x.

7.4.2 Tests

Test are similar to operations. There are two cases for tests. The first is when the conditional can be determined. This creates an expression specialization value like the ones for operations. The second case is when the conditional cannot be determined, an if specialization value is returned. This works whether operations are in-lined or not. When operations are not in-lined, the additional code sections are left empty, but when operations are in-lined, the additional code sections are filled in. One of the key parts of the if specialization value is it keeps stores for the `TRUE` and `FALSE` branches. This is used for conditional constant propagation (Section 7.3.3).

CHAPTER VIII

SPECIALIZER OPERATORS

The specializer takes in a number of operators that determine different aspects of the specialization process. These include the abstractions, the signature Σ, the projection operator π, and the widening operator θ.

8.1 Abstractions

The abstractions are one of the key controlling points of the specializer. Each abstraction is defined in a special abstraction class. These classes have methods to do the different operators and tests. They also include methods to create the residual code for the expression, to abstract values, and to lift and merge values of the abstraction. This way, each abstraction can handle the operations and functions differently. For example, in the int abstraction, the addition of two integers is the normal addition operator for integers. For the even odd abstraction, however, the addition operator depends on the abstract value. So if the expression *even*+ *even* is evaluated the result should return *even*. The expression *even*+ *odd*, however, should return *odd*. All these different cases must be implemented in the abstraction.

Currently, the abstraction classes are constructed by hand, but it is planned to make their construction more automatic. The abstractions that have been created so far have started off in a simple abstraction specification language and then hand compiled to Java code. The Bandera abstraction specification language (BASL) has

Specification

```
tokens:  NEG, ZERO, POS, T
```

Code

```
public final static Token NEG_TOKEN = new Token("NEG");
public final static Token ZERO_TOKEN = new Token("ZERO");
public final static Token POS_TOKEN = new Token("POS");
public final static Token T_TOKEN = new Token("T");

protected final static Token tokens[] = {NEG_TOKEN, ZERO_TOKEN,
                                         POS_TOKEN, T_TOKEN};
public final static TokenSet TOKENSET = new TokenSet(tokens);

public final static SignsAbstraction NEG =
                              new SignsAbstraction(NEG_TOKEN);
public final static SignsAbstraction ZERO =
                              new SignsAbstraction(ZERO_TOKEN);
public final static SignsAbstraction POS =
                              new SignsAbstraction(POS_TOKEN);
public final static SignsAbstraction T =
                              new SignsAbstraction(T_TOKEN);
```

Figure 8.1: Header

not been formally defined yet. However, in this chapter example BASL specifications are given as they are expected to be written once the language is formalized. Also, the code expected to be produced by the BASL compiler is shown. The BASL compiler automates the encoding of the abstractions in Java. There are seven sections to a BASL specification. The following sections discuss each of these using the signs abstraction as an example. The signs abstraction keeps track of the signs of the integer, but not the actual magnitude. The three tokens are **NEG**, **ZERO**, and **POS**. **NEG** represents values less than zero and **POS** represents values greater than zero.

Specification

```
abstract
   (n < 0)  -> NEG
   (n == 0) -> ZERO
   (n > 0)  -> POS
```

Code

```
public static SignsAbstraction abs(int n)
{
  if (n < 0)
    return NEG;
  else
    if (n == 0)
      return ZERO;
    else
      if (n > 0)
        return POS;
      else
        return T;
}
```

Figure 8.2: Abstraction Function

8.1.1 Header

This section specifies the tokens of the abstraction. For example, in the signs abstraction the tokens are **NEG**, **ZERO**, **POS**, and $\top$. The BASL header section appears in the top of Figure 8.1. The code produced would look like that in the bottom of Figure 8.1.

8.1.2 Abstract Function

The abstract function section defines how to abstract the values. This is called to abstract constants and create a new instance of the abstraction. For example, if the constant 2 was abstracted for the signs abstraction, the result would be **POS**. An

Specification

```
merge
   POS  POS  -> POS
   ZERO ZERO -> ZERO
   NEG  NEG  -> NEG
   default   -> T
```

Code

```
public static Abstraction merge(SignsAbstraction i1,
                                SignsAbstraction i2)
{
  if (i1.same(POS) && i2.same(POS))
    return POS;
  else
    if (i1.same(ZERO) && i2.same(ZERO))
      return ZERO;
    else
      if (i1.same(NEG) && i2.same(NEG))
        return NEG;
      else
        return T;
}
```

Figure 8.3: Merge Function

example of the specification and resulting code can be seen in Figure 8.2.

8.1.3 Merge Function

The merge function is used to merge two values from the same abstraction. In many cases, this is the least upper bound operator. The specification defines a set of cases on what to return. The code uses an if-then-else approach. This can be seen in Figure 8.3.

Specification

```
lift NEG, ZERO, POS
```

Code

```
public Value lift()
{
  if (same(NEG))
    return NEG;
  else
    if (same(ZERO))
      return ZERO;
    else
      if (same(POS))
        return POS;
      else
        return null;
}
```

Figure 8.4: Lift Function

8.1.4 Lift Function

The lift function returns a Jimple expression if the value can be residualized and lifted, otherwise it returns null. In the case of the signs abstraction, we want to be able to residualize **NEG**, **ZERO**, and **POS** but not $\top$ in our example. Figure 8.4 contains the appropriate lift function.

8.1.5 Operators

The BASL specifications for operations and tests use patterns to defined the different cases. Cases are processed from top to bottom and the result is given by the first pattern matched. The specification section of Figure 8.5 show a few new features used to define an abstraction. The first is the wildcard pattern (*). The second is the `op1` and `op2` values. These represent the left and right operands, respectively. These

Specification

```
operator +
   POS  POS  -> POS
   NEG  NEG  -> NEG
   *    ZERO -> op1
   ZERO *    -> op2
   default   -> T
```

Code

```
public static SpecValue add(ExprSpecValue op1, ExprSpecValue op2)
{
  Abstraction v1 = op1.getValue();
  Abstraction v2 = op2.getValue();

  if (v1.same(POS) && v2.same(POS))
    return new SpecValue(POS);
  else
    if (v1.same(NEG) && v2.same(NEG))
      return new SpecValue(NEG);
    else
      if (v1.same(ZERO))
        return op2;
      else
        if (v2.same(ZERO))
          return op1;
        else
          return new SpecValue(T);
}
```

Figure 8.5: Addition Example

are used in this example because anything added to zero is itself. The value returned is a simple specialization value. The residual expression is added by the signature. This is done in one of three ways.

- Value is liftable

  The residual value is created by lifting the value when it is liftable.

- Value is not liftable and operations are not in-lined

  When the value is not liftable and operations are not in-lined, the signature creates the residual value by making a new operand with the residual values of the arguments.

- Value is not liftable and operations are in-lined

  When operations are in-lined, another call to the abstractions makes the residual code that replaces the operand and is passed back in an assignment specialization value.

The extra method in the abstraction matches the patterns for the rules and creates the code for the particular case. For example, if the expression was $x + 2$ for the rules in Figure 8.5 and x was $\top$, then all the rules where the right operand is **POS** apply. These are the first, fourth, and fifth rules. The code for each check is appended together to take care of all the cases. This creation of code is actually handled by a method in the base `Abstraction` class. This method is used in all the abstractions currently. This makes the creation of the different methods easy to do, either automatically or by hand.

```
test ==
   POS  POS  -> T     (-, -)   (-, -)
   NEG  NEG  -> T     (-, -)   (-, -)
   ZERO ZERO -> TRUE  (-, -)   (-, -)
   T    *    -> T     (op2, -) (-, -)
   *    T    -> T     (-, op1) (-, -)
   default   -> FALSE (-, -)   (-, -)
```

Figure 8.6: Equality Example: Specification

8.1.6 Tests

Tests work similar to operators but there are a few key differences. First, they are used to determine a branch in the execution. Second, if they cannot be determined, different values can flow down different paths. This made the creation of the specification a little more difficult. The solution was to create a spot for each possible value flowing down each branch. These are the values of the arguments to the test. For example, in the expression $x == y$, x and y are the arguments. These are the variables that can have values flow down different branches. If the y was a 2, then only the x could have its value changed. Also, there is only two branches. This made the specification for a test to have the two values of the pattern to match, a boolean value or $\top$ as the result, and two pair of the values that can flow down the branches.

Figure 8.6 shows the specification for the equality test. The values returned are tokens for the boolean abstraction. The items in parenthesis represent the new values for the operator arguments in the true branch and false branch respectively. The - is an empty token which represents that on operator does not change its value. If you look at the fourth pattern, the true branch is `(op2, -)`. This says that the left operand gets the same value as the right operand (`op2`). Figure 8.7 shows the code created for the specification.

```
public static SpecValue eq(ExprSpecValue op1, ExprSpecValue op2,
                           Store trueStore, Store falseStore)
{
  Abstraction v1 = op1.getValue();
  Abstraction v2 = op2.getValue();

  if (v1.same(POS) && v2.same(POS))
    return new IfSpecValue(BooleanAbstraction.T,
                           new ArrayList(), new ArrayList(),
                           trueStore, falseStore);
  else
    if (v1.same(NEG) && v2.same(NEG))
      return new IfSpecValue(BooleanAbstraction.T,
                             new ArrayList(), new ArrayList(),
                             trueStore, falseStore);
    else
      if (v1.same(ZERO) && v2.same(ZERO))
        return new IfSpecValue(BooleanAbstraction.TRUE,
                               new ArrayList(), new ArrayList(),
                               trueStore, falseStore);
      else
        if (v1.same(T))
          {
            trueStore.update(EVariable.convert(op1.getExpr()),
                             op2.getValue());
            return new IfSpecValue(BooleanAbstraction.T,
                                   new ArrayList(), new ArrayList(),
                                   trueStore, falseStore);
          }
        else
          if (v2.same(T))
            {
              trueStore.update(EVariable.convert(op1.getExpr()),
                               op1.getValue());
              return new IfSpecValue(BooleanAbstraction.T,
                                     new ArrayList(), new ArrayList(),
                                     trueStore, falseStore);
            }
          else
            return new IfSpecValue(BooleanAbstraction.FALSE,
                                   new ArrayList(), new ArrayList(),
                                   trueStore, falseStore);
}
```

Figure 8.7: Equality Example: Code

Tests are also like operators because there are two methods in each abstraction for each test. The second method, like for operators, is for when operator in-lining is done. It creates the code necessary to in-line the tests and ensure that the correct jump point is reached. It also ensures that the residual code makes the correct assignment. As in the previous example, when the left operand gets assigned the value of the right operand, this assignment must be residualized when the operations are in-lined to ensure correct transformation.

8.1.7 Other Functions

There are other functions associated with abstractions. These include retrieving the bit set that represents the set. This is also the unique number used when operation in-lining is done (see Section 5.3). There are also methods to create a new instance of the abstraction which represents the empty set. This is used when a new instance of an abstraction is needed. Finally, there is a method that creates the `CHOICE` expression with the values the abstraction represents.

8.2 Signature - Σ

The signature determines which method gets called for a particular operator or test. It extends Jimple's `ValueSwitch` which works as a switch between each expression. It works by having a separate method for each expression that needs to be implemented. Each method goes through the same set of steps.

1. Evaluate Subexpressions

   This step calls upon the signature to evaluate subexpressions. Some types of expressions do not need to do this step because they do not have any subexpressions, *i.e.* variable lookup and constants.

```
if (v1.ID == SignsAbstraction.absID &&
    v2.ID == SignsAbstraction.absID)
  res = SignsAbstraction.add(sv1, sv2);
else
  if (v1.ID == ZeroPosAbstraction.absID &&
      v2.ID == ZeroPosAbstraction.absID)
    res = ZeroPosAbstraction.add(sv1, sv2);
```

Figure 8.8: Determining the Abstraction

2. Evaluate Expression

   For simple expressions, such as variable lookup and constants, the signature evaluates the expression itself. For more complicated expressions, such as addition, subtraction, *etc.*, the values of the subexpressions are used to determine what abstraction is used to handle the expressions. For example, if both subexpressions evaluate to even odd abstractions and the expression is addition, then the addition method in the even odd abstraction will be called. An example of this can be seen in Figure 8.8. Each possible combination of abstractions must be handle for each possible operation. When two different abstractions come together, one must have a special method to handle the combination or convert one abstraction to the other and then call the corresponding method. All this can get difficult to implement by hand, but much of the code can be reused. It is planned to make this automatic in the future to remove much of the difficulty.

   The signature can also determine when to change the abstraction of a particular value. The only case currently used is for an expression such as $x + 2$. The x will be the abstraction assigned to it but the two will be a concrete integer value. This needs to be converted to the same abstraction (in many cases) as

x. A check is made to see if a value is a concrete integer and then, if so, it is abstracted to the correct abstraction. In the future, this will be used to convert from one abstraction to another when it makes sense to do so.

3. Create Residual Expression

   The result of the operation is next lifted to an expression. If the value is residualizable, then this step is finished. If the value is not residualizable, then a new expression must be created. For example, if the value of $x * 2$ is *even* and *even* is residualizable, an expression representing *even* is returned by the lift function and it is done. If, however, *even* is not residualizable, then the expression must be recreated with the residualized subexpressions put together with the operator, *i.e.* $x * 2$. This is handled by the signature when operation in-lining is not done. When operation in-lining is done, however, a separate method is called in the abstraction that creates the correction specialization value with the correct code to represent the expression.

4. Creation of the Specialization Value

   A specialization value is created for an operation expression by the signature when operation in-lining does not occur. In all other cases, the specialization values are created by the methods in the abstraction class. This makes the work of the signature stay basically a test for the types of abstractions to determine the correct abstraction and method to call.

8.3 Pi - π

The operator π is used to create an index from a state. Because a state contains an index and a store, these can be used in different combinations to create the new index.

π is implemented in its own class, and to create a new instance of π one extends the class. The key function to implement is the pi function:

```
public Index pi(State s)
```

This function is what is called to create a new index. This function can be configured in a number of ways but the three most common are

- Monovariant Analysis - Only one residual basic block for each original basic block. This works by returning the index of the original basic block.

  ```
  return s.getIndex();
  ```

- Maximally Polyvariant Analysis - A residual basic block for each different store and index pair. This works by creating a new index that combines the index and store.

  ```
  return new StoreIndex(s.getIndex(), s.getStore());
  ```

- Variable Polyvariant Analysis - This is where polyvariance is done only on a select group of variables. It limits, or reduces, the store to the number of variables and then combines it with the index.

  ```
  return new StoreIndex(s.getIndex(), s.getStore().limit(vars));
  ```

More complicated notions of polyvariance can also be encoded. For example, if a live variable analysis was done for each block, one might want the ABPS to be polyvariant on the live variables for a particular basic block. This would make the implementation do a different limit of the store depending on the index of the original basic block. The call might look something like

```
return new StoreIndex(s.getIndex(),
                      s.getStore().limit(liveVars(s.getIndex())));
```

where the function call `liveVars(s.getIndex())` returns the live variables for a given basic block.

8.4 Theta - θ

The θ operator performs a component-wise merge on two stores. The merge function is usually defined in the class representing the abstraction of the particular value. The main user defined function in the operator determines which function to call to merge the two values. For example, if the two values are from the even odd abstraction, then it would call merge in the even odd abstraction. An example of the merge function can be seen in Figure 8.9. This checks to see if two values are either both int abstractions (`IntAbstraction`) or even odd abstractions (`EOAbstraction`).

Other features can be added to the merge function. One common occurance is where the value is an int abstraction in one store and an even odd abstraction in the other. In the code in Figure 8.9, an exception would be thrown. To solve this, the code in Figure 8.10 would be put into the if-then-else statements.

```
public class SimpleTheta extends Theta
{
  public Abstraction merge (Object o1, Object o2)
  {
    if (o1 instanceof IntAbstraction &&
        o2 instanceof IntAbstraction)
      return IntAbstraction.merge((IntAbstraction)o1,
                                  (IntAbstraction)o2);
    else
      if (o1 instanceof EOAbstraction &&
          o2 instanceof EOAbstraction)
        return EOAbstraction.merge((EOAbstraction)o1,
                                   (EOAbstraction)o2);
      else
        throw new RuntimeException("Unhandled case in merge:  " +
                                   o1 + "\t" + o2);
  }
}
```

Figure 8.9: θ Example

```
if (o1 instanceof EOAbstraction &&
    o2 instanceof IntAbstraction)
  return EOAbstraction.merge((EOAbstraction)o1,
                             EOAbstraction.abs(
                                  ((IntAbstraction)o2).value));
else
  if (o1 instanceof IntAbstraction &&
      o2 instanceof EOAbstraction)
    return EOAbstraction.merge(EOAbstraction.abs(
                                   ((IntAbstraction)o1).value)
                               (EOAbstraction)o2);
```

Figure 8.10: Addition Code for θ

CHAPTER IX

RESULTS

The ABPS tools processes a small subset of Java. This includes integers and their operators. The tools can only specialize a method so this limits the application, but there are still some interesting examples that can be treated. This chapter consists of examples using a concurrent readers/writers control code.

9.1 Reader/Writer Controller

A controller for concurrent readers/writers keeps track of the number of reader processes and the number of writer processes. This is to ensure that there are no readers while there is a writer process. Also, that there can be no more than one writer process. The example has been modified to keep only the features needed for this line of tests. This is a common procedure in FSV to reduce the state space and the complexity of the problem.

Figure 9.1 shows the Java source code for the reader writer controller. The code loops until it gets a request to stop. `Model.choose(4)` nondeterministicly chooses a value between 0 and 4. The value is assigned to `req`. Figure 9.2 shows the actions represented by the request values. The resulting Jimple code is in Figures 9.3 and 9.4.

```
public static void controller()
{
  boolean writerPresent = false;
  int req = 1;
  int activeReaders = 0;
  boolean errorFlag = false;

  while (req!=0)
    {
      req = Model.choose(4);  // controller@7

      if (req == 1)
        {
          if (!writerPresent)  // startRead()
            ++activeReaders;   // controller@13  START_READ
        }
      else
        if (req == 2) // else of case 1
          {
            if (activeReaders>0)
              {
                activeReaders = activeReaders - 1;
                if (writerPresent) errorFlag = true;
              }
          }
        else
          if (req == 3) // else of case 2
            {
              if (activeReaders==0 && !writerPresent)
                writerPresent = true;  // controller@19  START_WRITE
            }
          else
            if (req == 4) //else of case 3
              if (writerPresent)
                {
                  writerPresent = false;  // controller@21  STOP_WRITE
                  if (activeReaders>0) errorFlag = true;
                }
    }
}
```

Figure 9.1: Reader Writer Java Code

req	Step
0	Ends the loop
1	Add a new reader
2	Removes a reader
3	Adds a new writer
4	Removes the writer

Figure 9.2: Reader Writer Requests

9.2 Abstractions

There were three different abstractions used for these examples. For all the abstractions, only the operators and tests that are needed are implemented.

9.2.1 Boolean Abstraction

A boolean abstraction keeps track of the boolean values `TRUE` and `FALSE`. They can also have an unknown, $\top$. These were used for the `writerpresent` variable.

9.2.2 Zero Positive Abstraction

The zero-positive abstraction is similar to the signs abstraction. The only notable difference is the lack of a negative token. It is used for the `activereaders` variable which keeps a count of the number of active readers. It is safe to use the zero-positive abstraction because the number of active readers can never fall below 0. Also, all tests on active readers need to know only if the value is 0 or greater than zero.

```
public static void controller()
{
    int i0, writerPresent, req, activeReaders;

    writerPresent = 0;
    req = 1;
    activeReaders = 0;
    goto label4;

 label0:
    i0 = Model.choose(4);
    req = i0;
    if req != 1 goto label1;

    if writerPresent != 0 goto label4;

    activeReaders = activeReaders + 1;
    goto label4;

 label1:
    if req != 2 goto label2;

    if activeReaders <= 0 goto label4;

    i0 = activeReaders - 1;
    activeReaders = i0;
    if writerPresent == 0 goto label4;

    goto label4;

 label2:
    if req != 3 goto label3;

    if activeReaders != 0 goto label4;

    if writerPresent != 0 goto label4;

    writerPresent = 1;
    goto label4;
```

Figure 9.3: Reader Writer Jimple Code (Part 1)

```
label3:
    if req != 4 goto label4;

    if writerPresent == 0 goto label4;

    writerPresent = 0;
    if activeReaders <= 0 goto label4;

label4:
    if req != 0 goto label0;

    return;
}
```

Figure 9.4: Reader Writer Jimple Code (Part 2)

9.2.3 Range 0-4 Abstraction

The last abstraction used is a range abstraction. A range abstraction has tokens to represent values in a particular range, in this case between 0 and 4. This is used for the `req` variable because that is the range of its values.

9.3 Signature and Theta

ABPS also requires specification of the signature and theta. These are straight forward because they basically choose between which abstraction to call. Signatures were discussed in Section 8.2 and theta in Section 8.4.

9.4 Running the ABPS Tools

Unfortunately, the current implementation only takes a few command line arguments. These include the class and method names and three possible options. The class

names is the fully qualified class name. This is the combination of the package and class name in Java. ABPS currently does not distinguish between overloaded methods. It will simply specialize the first method with the given name. The two options are `-poly` and `-inline`. The `-poly` option chooses polyvariant analysis instead of the default monovariant analysis. To do operation in-lining, the option `-inline` is used. The other options must be implemented by changing the code in `Main.java` or the different specializer operators.

Two changes can be made by modifying the code in the `Main` class. The first is the arguments passed as the parameters of the method. These arguments are passed as an array, one entry for each parameter. One can change the value and the abstraction used for these arguments. Also, there is a table passed to the specializer that associates variables with abstractions. One can add new entries into this table. In the future, these will be retrieved from the command line or from an options file.

9.5 Results

The following section presents the results of running ABPS in several different modes: monovariant, polyvariant, and monovariant with operation in-lining.

9.5.1 Monovariant Case

The monovariant case is the easiest to understand. For each label in the source program, there is exactly one label in the residual program. A table to show the relationship between the source labels and residual labels is in Figure 9.5. The resulting code can be seen in in Appendix C starting on page 111.

Figure 9.6 contains a piece of the residual code that corresponds to `label1` in Figure 9.6. One of the first noticeable differences between the code in Figure 9.3

Source Labels	Monovariant Case	Polyvariant Case
label0	label1	label1, label5, label7, label10, label11, label17, label22, label23, label29, label32, label34, label39, label41, label43, label44, label47, label51
label1	label2	label3, label4, label13, label15, label18, label30, label46
label2	label3	label9, label12, label20, label24, label26, label36, label48
label3	label4	label16, label19, label28, label31, label33, label40, label49
label4	label0	label25, label27, label35, label37, label38, label45, label50

Figure 9.5: Source and Residual Labels

```
label2:
 if req != R2 goto label3;

 if activeReaders <= ZERO goto label0;

 i1 = POS - POS;
 activeReaders = i1;
 if writerPresent == FALSE goto label0;

 goto label0;
```

Figure 9.6: Excerpt of Monovariant Results

and the source code in Figure 9.3 is the inclusion of tokens. The tokens represents abstractions which are considered Jimple constants (see Section 5.3). Recall from Section 3.2.1, that the user can control which tokens are residualized and which tokens are not residualize.

Another difference is the conversion of the expression `activeReaders - 1` to `POS - POS` in Figure 9.6. This conversion occurs because the result of **POS – POS** is $\top$ (subtracting two positive natural numbers can give either a zero or a positive number). The expression was residualized because $\top$ is not residualizable.

9.5.2 Polyvariant Case

The polyvariant case is probably the most difficult example to follow (see Appendix C on page 112). Part of the reason is the code explosion that occurred. The code explosion is a result of the creation of a new residual basic block for the combination of each source basic block and variable values. Figure 9.7 is an excerpt of residual code that demonstrates multiple residual blocks. These pieces of code is some of the specialized versions of the code found under `label1` in Figure 9.3. Each label, except `label14`, represents the specialized entry point to `label1`. `label14` is the body of the same block. Figure 9.8 helps explain why each block is different. The figure shows the entering values at each entry point. With these values, each case can be explained.

- `label3`: The number of active readers is **ZERO** in this case. This means the check `activeReaders <= ZERO` is true and the rest of the block is skipped.

- `label4`: The number of active readers is **POS**. The check is false and so the rest of the block is specialized.

- `label13`: The number of active readers is $\top$. The check is unknown, so each

```
label3:
 if req != R2 goto label9;

 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label3;

 goto label0;

label4:
 if req != R2 goto label12;

 i0 = POS - POS;
 activeReaders = i0;

           :

label13:
 if req != R2 goto label20;

 if activeReaders <= ZERO goto label5;

label14:
 i0 = POS - POS;
 activeReaders = i0;
 goto label5;

label15:
 if req != R2 goto label24;

 goto label14;
```

Figure 9.7: Excerpt of Polyvariant Results

label	req	activeReaders
label3	$\top$	**ZERO**
label4	$\top$	**POS**
label13	$\top$	$\top$
label14	$\top$	**POS**
label15	$\top$	**POS**

Figure 9.8: Variable Values Upon Block Entry

possible branch must be taken. To follow the false branch, the number of active readers must be positive. This information is propagated down the path and is an example of conditional constant propagation (see Section 7.3.3). The control then flows down into the body of the block.

- `label15`: The number of active readers is **POS**. The check is false so the rest of the block is specialized. The values flowing into the body match those flowing into the body from `label13`, allowing the two blocks to be merged.

The differences between the cases where the number of active readers is **POS** is from the different values of the other variables. These are not the only residual blocks created for the source blocks. The rest can be seen in Figure 9.5.

9.5.3 Monovariant Case with Operation In-Lining

The examples for monovariant specialization with operation in-lining uses some different code than the other examples because the reader/writers code is not a good example of what can happened during operation in-lining. The example code used for operation in-lining is in Figure 9.9. It contains an assignment where a positive value is added to the variable x. ABPS creates the residual code in Figure 9.9 when operation in-lining is set and x is $\top$.

Source Code

```
i0 = 3 + x;
```

Residual Code

```
if x != POS goto label0;

i0 = POS;
goto label2;

label0:
   if x != ZERO goto label1;

   i0 = POS;
   goto label2;

label1:
   i0 = Model.choose({NEG,ZERO,POS});

label2:
```

Figure 9.9: Monovariant with In-lining Example

When x is **POS** or **ZERO**, the result is **POS**. When x is **NEG**, the result is $\top$., which gets residualized as a nondeterministic choice. All these cases needs to be handled when operation in-lining is done in this case. To do this, ABPS creates code that checks for each possible value of x that does not result in $\top$. For each of these cases, it creates code to assign the correct value for the assignment. When the result is $\top$, ABPS creates a nondeterministic assignment, `i0 = Model.choose({NEG,ZERO,POS})`.

9.5.4 BIRC Output

One of the main goals of ABPS was for the output to be processed by BIRC, the translator that converts Jimple code to code for a FSV. Figure 9.10 is a sample

```
:: atomic { ((req == R1) && (writerPresent == FALSE)) ->
  req = {R0, R1, R2, R3, R4};
  activeReaders = POS;
  goto loc_2; }
:: atomic { ((req == R2) && (activeReaders > ZERO) &&
              (writerPresent == FALSE)) ->
  req = {R0, R1, R2, R3, R4};
  activeReaders = {POS, ZERO};
  goto loc_2; }
:: atomic { ((req == R3) && (activeReaders != ZERO)) ->
  req = {R0, R1, R2, R3, R4};
  goto loc_2; }
:: atomic { ((req == R3) && (activeReaders == ZERO) &&
              (writerPresent != FALSE)) ->
  req = {R0, R1, R2, R3, R4};
  goto loc_2; }
:: atomic { ((req != R3) && (activeReaders == ZERO) &&
              (writerPresent == FALSE)) ->
  req = {R0, R1, R2, R3, R4};
  writerPresent = TRUE;
  goto loc_2; }
```

Figure 9.10: Promela Code from BIRC

of the output created by BIRC for the readers/writers example. This output has been optimized by hand to make it more readable. Trivial tests, *i.e.* $0 == 0$ and $1! = 0$, have been removed. Also, the equations have been rearranged. For example, $(!(activeReaders! = ZERO))$ was changed to $(activerReaders == ZERO)$. Finally, extraneous parentheses were removed. The output was obtained by putting the residual method into a Java class that BIRC then processed. Spin was used to check the readers/writers with the promela output. The results were not what was expected because the current version of BIRC did not allow static invoke expressions in the Jimple code.

CHAPTER X

CONCLUSION

This work has demonstrated that partial evaluation and abstract interpretation can be combined to obtain a tool that automatically generates abstract models of simple software systems. These models can be fed into existing finite state verification tools. The ABPS has been integrated into the larger Bandera verification system. An initial release of the Bandera tool set (including ABPS) is being tested by researchers at NASA Ames, Stanford, University of Massachusetts, and University of Hawaii.

10.1 Assessment

- Usability

  In its current state, the system is not too difficult to use if the abstractions remain simple and the Java code lies in the subset of what is implemented. However, there are several limitations that can be improved immediately. There are not many options that can be passed on the command line. One can see some nice examples, but if a different method with different parameters or variables is to specialized, one must change some ABPS code by hand. Also, abstractions and the signature are still difficult to implement. This is because they must be done by hand. This makes it tedious to add much functionality, or even a new abstraction, to the current code. The structure was designed so

that it would be relatively easy for a tool to automatically construct these from a specification file.

- Effectiveness

  The effectiveness seems to be pretty good for the examples it was tried on. The results matched up to models that would be constructed by hand. To be fully effective, objects and arrays need to be handled, plus a way to specialize whole systems. The system is currently not as efficient as it could be but this will be addressed in the future.

- Technical Challenges

  Creating modular abstractions became the biggest technical challenge. The ABPS tools needed to be able to change abstractions without much hassle. Also, in the future, they need to be created automatically. This meant that they needed a regular structure that would be easy to specify and easy to use. The structure of abstractions and the signature changed over the course of working on the implementation. Each one came closer to the desired goal of being more automatic, but there are still changes that can be made to make it more so.

10.2 Future Work

There is much to be done in the future on this system. The two primary goals are to make it more automated, and to scale it up to a wider range of Java features. Many aspects of the first goal can be handled without much difficulty. To make the creation of abstractions and signatures more automated, an abstraction specification language needs to be developed. Some work has been done for this, and the specification language looks much like the sample specifications presented in Chapter VIII.

After this, a compiler needs to be written that compiles the specifications to Java source code that can be used by the system. Further work also needs to be done on the structures of the abstractions to aid in the creation of the signature and other operations passed to the ABPS tools. These include structures that hold the operators and tests implemented along with their method names. This would allow the signature to be created automatically from the names of the abstractions to be used. Other information could also be included to determine if it is possible to change one abstraction into another.

The degree of automation can also be increased by passing more options to ABPS. Options can be supplied on the command line or in an options file. With these techniques, one can remove the need to change actual ABPS code for different configurations of the system.

Scaling up the system to include additional Java features will be more challenging. This is because specialization will occur on objects, not just methods and variables. There is little information in the literature on the ways to scale up to a object oriented language. There are four main areas to be included to scale to a full Java systems, each with its own plan.

1. Composite Data

   Composite data includes arrays and structures. In both, different parts of each can have different binding times. For example, a structure with variables x and y can have x and y static, x static and y dynamic, x dynamic and y static, or both could be dynamic. This problem is discussed in the literation and possible solutions are presented [2, 20]. More research needs to be done to include composite data into the system.

2. Object Flow

   Object flow plays a major roll in object oriented languages. This is because methods can be inherited or overridden from the parent class. When a method call is made on a particular object, if the exact object type is not known, a virtual method call has to be made. For a virtual method call, the type of the object has to be determined at runtime and then the corresponding method. In Java, virtual method calls are considered slow. An object flow analysis is being developed for Jimple [30]. To understand the possible analyses and to determine the best course, the Jimple analysis, Mossin's PhD Thesis [24], and other possible sources need to be studied and the best course of action implemented.

3. Partial Evaluation of Object Oriented Languages

   Partial evaluation of object oriented languages is new and many of the problems do not have good solutions. Studying the literature [4, 9, 28] and developing methods to incorporate object oriented features into partial evaluation will need to be done.

4. Partial Evaluation of Languages with Concurrent Features

   Java includes primitives and functions for concurrent software. A look at the literature for partial evaluation of concurrent languages has been reported in the literature [22, 23] and more research needs to be done to be able to partially evaluate Java.

More research and study of the literature needs to be done in each of the areas to develop a plan to scale up to the other features to Java.

REFERENCES

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers - Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, USA, 1986.

[2] Andersen, L. O. *Program Analysis and Specialization for the C Programming Language.* Ph.D. Thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1994. DIKU Report 94-19.

[3] Ashley, J. M. A practical and flexible flow analysis for higher-order languages. *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Programming Languages* (St. Petersburg Beach, FL, Jan. 1996), pp. 184–194.

[4] Braux, M. Speeding up the meta-level processing of Java through partial evaluation. *Workshop on Reflective Programming in C++ and Java at OOPSLA* (Center for Computational Physics, University of Tsukuba, Japan, Oct. 1998), J.-C. Fabre and S. Chiba, Eds.

[5] Cattel, T. Process control design using spin. *Proceedings of the First SPIN Workshop* (Montreal, Quebec, Oct. 1995).

[6] Cimatti, A., Giunchiglia, F., Mongardi, G., and Romano, D. Model checking safety-critical software with SPIN: An application to a railway interlocking system. *Lecture Notes in Computer Science 1516* (1998), 284–300.

[7] Consel, C., and Khoo, S. C. Parameterized partial evaluation. *ACM Trans. Program. Lang. Syst. 15*, 3 (1993), 463–493.

[8] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages* (Los Angeles, California, Jan. 1977), pp. 238–252.

[9] Cowan, C., Black, A., Krasikand, C., Pu, C., Walpole, J., Consel, C., and Volanschi, E.-N. Specialization classes: An object framework for specialization. *Fifth IEEE International Workshop on Object Orientation in Operating Systems* (Seattle, WA, Oct. 1996).

[10] Dwyer, M., Carr, V., and Hines, L. Model checking graphical user interfaces using abstractions. *Software Engineering—-ESEC/FSE '97: Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Zurich, Switzerland, Sept. 1997), M. Jazayeri and H. Schauer, Eds., Vol. 1301 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 244–261.

[11] Dwyer, M., and Wallentine, V. Object-oriented coordination abstractions for parallel software. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)* (Las Vegas, NV, June 1997).

[12] Dwyer, M. B., Craig, M. J., and Runquist, E. An application-independent concurrency skeleton in Ada 95. *Proceedings of the TRI-Ada Conference* (New York, NY, Dec. 1996), ACM Press, pp. 179–192.

[13] Dwyer, M. B., and Pasareanu, C. S. Filter-based model checking of partial systems. *Sixth International Symposium on the Foundation of Software Engineering* (Lake Buena Vista, FL, Nov. 1998).

[14] Elseaidy, W. M., Cleaveland, R., and Baugh Jr, J. W. Modeling and verifying active structural control systems. *Science of Computer Programming 29*, 1–2 (July 1997), 99–122.

[15] Gomard, C. K., and Jones, N. D. Compiler generation by partial evaluation. *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress* (1989), G. X. Ritter, Ed., IFIP, North-Holland, pp. 1139–1144.

[16] Hankin, C., Nielson, F., and Nielson, H. R. Advanced course on the principles of program analysis, Nov. 1998. Professional Development Course at Schloss Dagstuhl, Event No. 98451. Dagstuhl, Germany.

[17] Hatcliff, J., Dwyer, M., and Laubach, S. Staging static analyses using abstraction-based program specialization. *Lecture Notes in Computer Science 1490* (1998), 134–151.

[18] Holzmann, G. The model checker spin. *IEEE Transactions on Software Engineering 23*, 5 (May 1997), 279–294.

[19] Jones, N. D. The essence of program transformation by partial evaluation and driving. *Logic, Language and Computation, a Festschrift in honor of Satoru Takasu*, M. S. Neil D. Jones, Masami Hagiya, Ed. Springer-Verlag, Berlin/Heidelberg, Germany, Apr. 1994, pp. 206–224.

[20] Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, London, UK, 1993.

[21] Jones, N. D., and Neilson, F. Abstract interpretation: a semantics-based tool for program analysis. *Handbook of Logic in Computer Science.* Oxford University Press, Oxford, UK, 1994, pp. 527–629.

[22] Marinescu, M., and Goldberg, B. Partial-evaluation techniques for concurrent programs. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97)* (New York, NY, June 12–13 1997), Vol. 32, 12 of *ACM SIGPLAN Notices*, ACM Press, pp. 47–62.

[23] Masuhara, H., and Yonezawa, A. Design and partial evaluation of meta-objects for a concurrent reflective language. *Lecture Notes in Computer Science 1445* (1998), 418–439.

[24] Mossin, C. *Flow Analysis of Typed Higher-Order Programs.* Ph.D. Thesis, DIKU, Computer Science Department, University of Copenhagen, January (revised August) 1997.

[25] Naumovich, G. N., Clarke, L. A., and Osterweil, L. J. Verification of communication protocols using data flow analysis. *Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering* (New York, NY, Oct.16–18 1996), Vol. 21 of *ACM Software Engineering Notes*, ACM Press, pp. 93–105.

[26] Personal Communication, Jay Corbett, February, 1999.

[27] Schmidt, D. A. Trace-based abstract interpretation of operational semantics. *Journal Lisp and Symbolic Computation 10*, 3 (1998), 237–271.

[28] Schultz, U. P., Lawall, J., Consel, C., and Muller, G. Toward automatic specialization of Java programs. *13th European Conference on Object-Oriented Programming* (Lisbon, Portugal, June 1999).

[29] Sørensen, M. H., and Glück, R. An algorithm of generalization in positive supercompilation. *Logic Programming: Proceedings of the 1995 International Symposium* (Portland, OR, 1995), J. Lloyd, Ed., MIT Press, pp. 465–479.

[30] Sundaresan, V., Razafimahefa, C., Vallee-Rai, R., and Hendren, L. J. Practical virtual method call resolution for Java. Tech. Rep. tr-1998-7, McGill University, Computer Science Department, Montreal, Quebec, November 1998.

[31] Vallee-Rai, R., and Hendren, L. J. Jimple: Simplifying Java bytecode for analyses and transformations. Tech. Rep. tr-1998-4, McGill University, Computer Science Department, Montreal, Quebec, July 1998.

[32] Wing, J. M., and Vaziri-Farahani, M. Model Checking Software Systems: A Case Study. *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Washington, D.C., Oct. 1995), pp. 128–139.

[33] Wulf, W. A., Shaw, M., Hilfinger, P., and Flon, L. *Fundamental Structures of Computer Science.* Addison-Wesley, Reading, MA, 1981.

APPENDICES

APPENDIX A

GLOSSARY

ABPS	Abstraction-Based Program Specialization.
AI	Abstract Interpretation.
AST	Abstract Syntax Tree.
Abstract Interpretation	Interpretation of a program using abstract values.
Abstract Interpreter	A rigorous methodology for static program analysis by manipulating abstract tokens.
BASL	Bandera Abstraction Specification Language. A language that is used to define abstractions for ABPS. It is currently in the development stages.
BIRC	A part of the Bandera toolset that translates Java into FSV input.
Basic Block	Sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end [1].
Box	A reference to an expression in Jimple that allows one to change its value without affecting the rest of the structure.
CI	Concrete Interpretation.
Concrete Interpretation	Interpretation of a program using concrete values.
Conditional Constant Propagation	Where the values of a conditional expression are used to propagate values down different branches.
Dynamic	Unknown.

FCL	Flow Chart Language.
FSV	Finite State Verification.
Finite State Verification	Model-based techniques that can be used to check that a system satisfies certain properties.
Hammock Form	Form where there is only one entry and exit point.
IR	Intermediate Representation.
In-lining	Process of moving a method body into another method to eliminate the overhead of call the moved method.
Jimple	Intermediate representation of Java produced by the Sable group at McGill University in Canada (see Section 1.4.2).
Jimplification	The process of converting Java to a Jimple representation.
Label	A program point.
Lift	A method that converts a value to a corresponding expression.
Maximally Polivariant	Polyvariance where each state in the program has a corresponding residual basic block.
Monovariant	Analysis where there is only one residual basic block for each original basic block.
Off-line Partial Evaluation	Partial evaluation where the analysis is done before the evaluator runs.
On-line Partial Evaluation	Partial evaluation where the analysis is done while the evaluator is running.
PE	Partial Evaluator.
Partial Evaluator	A technique for specializing programs based on information known about the environment or expected patterns of use.
Pending List	List of states waiting to be analyzed.
Polyvariance	Analysis where there can be more than one residual basic block for each original basic block.

Residualize	Place the code into the output code.
Seen-Before Set	Set of states that have been analyzed.
Signature	Operator that parameterizes the syntax. It contains operations, tests, and constants.
State	A combination of a label and a store.
Static	Known.
Store	The representation of memory, it is a lookup table for variables and their values.
Three Address Code	A representation where each statement has at most three operands. Method invocation is an exception to this.

APPENDIX B

NOTATION

$\circ$	Representation for an unmarked node.
$\bullet$	Representation for a marked node.
$\rightharpoonup$	Partial function.
$\to$	Transition between states or a total function.
$\mapsto$	Used in representation of stores to associate a variable to a value.
$\subseteq$	Subset or equal.
$\sqsubseteq$	Partial order.
Ξ	Specialization structure.
π	A function that maps a state to an index.
σ	Store.
θ	A function that merges two stores.
$\top$	Upper bound, represents all possible values or unknown.
Σ	Signature, contains the operators and tests.
Γ	Block-map, maps a label to a basic block.
Γ_{R}	Residual block-map.
$[\![\cdot]\!]_A^\Sigma$	Map symbols in Σ to operations in A.
A	Algebra, contains values, operations, and tests.

A_{eo}	The algebra for the even/odd abstraction.
$AbsVal$	Abstract value.
β	Abstraction function.
b	Basic block.
$\mathcal{C}$	Cache, maps indices to stores.
$dom(\sigma)$	Set of defined variables for the store.
e	Even abstract value.
ι	Index.
ι°	An unmarked index pending specialization.
$\iota^{\bullet}$	A marked index that is up-to-date.
l	Label.
lift	Maps a residualizable value to a constant.
m	A mark, either $\circ$ or $\bullet$.
n	Node.
o	Odd abstract value.
R	Set of values that can be residualized.
$\mathcal{S}$	Program skeleton.
s	State.
Val	Concrete value.

APPENDIX C

SPECIALIZATION OUTPUT

This appendix contains the output of ABPS with the readers/writers. This output is descussed in more depth in Chapter IX.

- Monovariant Output - the output created when monovariance is chosen for the readers/writers.
- Polyvariant Output - the output created when polyvariance is chosen for the readers/writers.

Monovariant Output

```
void controller$abps_mono()
{
  int i1, i0, activeReaders, writerPresent, req;

  writerPresent = FALSE;
  req = R1;
  activeReaders = ZERO;

 label0:
  if req != R0 goto label1;

  return;

 label1:
  i0 = Model.choose({R0,R1,R2,R3,R4});
  req = i0;
  if req != R1 goto label2;

  if writerPresent != FALSE goto label0;
```

```
  activeReaders = POS;
  goto label0;

 label2:
  if req != R2 goto label3;

  if activeReaders <= ZERO goto label0;

  i1 = POS - POS;
  activeReaders = i1;
  if writerPresent == FALSE goto label0;

  goto label0;

 label3:
  if req != R3 goto label4;

  if activeReaders != ZERO goto label0;

  if writerPresent != FALSE goto label0;

  writerPresent = TRUE;
  goto label0;

 label4:
  if req != R4 goto label0;

  if writerPresent == FALSE goto label0;

  writerPresent = FALSE;
  if activeReaders <= ZERO goto label0;

  goto label0;
}
```

Polyvariant Output

```
void controller$abps_poly()
{
  int i1, i0, activeReaders, writerPresent, req;

  writerPresent = FALSE;
  req = R1;
  activeReaders = ZERO;
  i1 = Model.choose({R0,R1,R2,R3,R4});
  req = i1;
  if req != R1 goto label3;

 label0:
  activeReaders = POS;
```

```
label1:
 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label4;

label2:
 activeReaders = POS;
 goto label1;

label3:
 if req != R2 goto label9;

 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label3;

 goto label0;

label4:
 if req != R2 goto label12;

 i0 = POS - POS;
 activeReaders = i0;

label5:
 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label13;

label6:
 activeReaders = POS;

label7:
 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label15;

label8:
 activeReaders = POS;
 goto label7;

label9:
 if req != R3 goto label16;

 writerPresent = TRUE;

label10:
 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label18;

label11:
```

```
 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label18;

 goto label11;

label12:
 if req != R3 goto label19;

 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label4;

 goto label2;

label13:
 if req != R2 goto label20;

 if activeReaders <= ZERO goto label5;

label14:
 i0 = POS - POS;
 activeReaders = i0;
 goto label5;

label15:
 if req != R2 goto label24;

 goto label14;

label16:
 if req != R4 goto label25;

label17:
 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label3;

 goto label0;

label18:
 if req != R2 goto label26;

 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label18;

 goto label11;

label19:
 if req != R4 goto label27;

 i1 = Model.choose({R0,R1,R2,R3,R4});
```

```
 req = i1;
 if req != R1 goto label4;

 goto label2;

label20:
 if req != R3 goto label28;

 if activeReaders != ZERO goto label29;

label21:
 writerPresent = TRUE;

label22:
 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label30;

label23:
 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label30;

 goto label23;

label24:
 if req != R3 goto label31;

 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label15;

 goto label8;

label25:
 if req != R0 goto label32;

 return;

label26:
 if req != R3 goto label33;

 goto label10;

label27:
 if req != R0 goto label34;

 return;

label28:
 if req != R4 goto label35;

 i1 = Model.choose({R0,R1,R2,R3,R4});
```

```
 req = i1;
 if req != R1 goto label13;

 goto label6;

label29:
 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label13;

 goto label6;

label30:
 if req != R2 goto label36;

 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label30;

 goto label23;

label31:
 if req != R4 goto label37;

 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label15;

 goto label8;

label32:
 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label3;

 goto label0;

label33:
 if req != R4 goto label38;

 writerPresent = FALSE;
 goto label17;

label34:
 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label4;

 goto label2;

label35:
 if req != R0 goto label39;
```

```
 return;

label36:
 if req != R3 goto label40;

 goto label22;

label37:
 if req != R0 goto label43;

 return;

label38:
 if req != R0 goto label44;

 return;

label39:
 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label13;

 goto label6;

label40:
 if req != R4 goto label45;

 writerPresent = FALSE;

label41:
 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label46;

label42:
 activeReaders = POS;
 goto label7;

label43:
 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label15;

 goto label8;

label44:
 i1 = Model.choose({R0,R1,R2,R3,R4});
 req = i1;
 if req != R1 goto label18;

 goto label11;

label45:
```

```
  if req != R0 goto label47;

  return;

 label46:
  if req != R2 goto label48;

  i1 = Model.choose({R0,R1,R2,R3,R4});
  req = i1;
  if req != R1 goto label46;

  goto label42;

 label47:
  i1 = Model.choose({R0,R1,R2,R3,R4});
  req = i1;
  if req != R1 goto label30;

  goto label23;

 label48:
  if req != R3 goto label49;

  goto label21;

 label49:
  if req != R4 goto label50;

  goto label41;

 label50:
  if req != R0 goto label51;

  return;

 label51:
  i1 = Model.choose({R0,R1,R2,R3,R4});
  req = i1;
  if req != R1 goto label46;

  goto label42;
}
```

APPENDIX D

PROGRAM LISTING

The program files are presented in this appendix. The following files contain code written in Java. The files with extension .java are Java files. The files succeeded by a * is not included in the appendix. The order of the files in the following pages is given below:

edu.ksu.cis.bandera.util

Table.java *

Token.java *

TokenSet.java *

edu.ksu.cis.bandera.prog

BasicBlock.java *

BlockMap.java *

CFGSkel.java *

Cache.java *

EValue.java *

EVariable.java *

Index.java *

Inline.java *

State.java *

Store.java *

StoreIndex.java *

ValueInitializer.java *

edu.ksu.cis.bandera.jext

ChooseExpr.java *

edu.ksu.cis.bandera.abps

ABPS.java

ABPSArgs.java *

Abstraction.java *

ExprSpecValue.java *

IfSpecValue.java *

IntAbstraction.java *

Main.java

Pi.java *

Signature.java

SpecType.java *

SpecValue.java *

StmtSpecValue.java *

Theta.java *

edu.ksu.cis.bandera.abps.lib

BooleanAbstraction.java *

ConcreteInt.java *

Range04Abstraction.java *

SignsAbstraction.java *

SimplePi.java *

SimpleSignature.java *

SimpleTheta.java *

ZeroPosAbstraction.java *

```
//ABPS.java
package edu.ksu.cis.bandera.abps;

import java.io.*;

import ca.mcgill.sable.soot.*;
import ca.mcgill.sable.soot.jimple.*;
import ca.mcgill.sable.util.*;

import edu.ksu.cis.bandera.abps.lib.*;
import edu.ksu.cis.bandera.prog.*;
import edu.ksu.cis.bandera.util.Table;

/**
 * This is the core of the specializer.  It does the specialization of
 * methods, blocks, statemetns, and the initial call for expressions.
 *
 * @author <a href="mailto:laubach@cis.ksu.edu">Shawn Laubach</a>
 *
 * @version 0.1
 */
public class ABPS
{
  /**
   * The Soot Class Manager that is used for everything.
   */
  public static SootClassManager cm = new SootClassManager();

  protected SootMethod method;  // The method being specialized
  protected SootClass cls;      // The class the method is in

  protected BlockMap bm;        // The block map
  protected BlockMap residual = new BlockMap(); // The residual block map
  protected Store initStore;    // The initial store

  protected Cache c = new Cache(); // The cache
  protected CFGSkel S = new CFGSkel(); // The control flow skeleton

  protected Abstraction result; // The result of the method

  // A map of the residual statements to original statements
  protected Map residualToOriginalStmts;
  // A map of the residual labels to the original labels
  protected Map residualToOriginalLabels;

  /**
   * Option to inline the operations.
```

```
 */
public static boolean inlineAbstractions = false;

/**
 * Option for monovariant.
 */
public static boolean monovariant = true;

/**
 * This constructs a new specializer that loads in the class and
 * method and make the block map.
 *
 * @param clsStr the name of the class
 * @param met the name of the method
 */
public ABPS(String clsStr, String met)
{
  //      SootClass cls;
  Object m[];
  int i;

  cls = cm.getClass(clsStr);
  cls.resolveIfNecessary();
  m = cls.getMethods().toArray();
  for (i = 0; i < m.length &&
         !((SootMethod)m[i]).getName().equals(met); i++);
  BuildAndStoreBody bd =
    new BuildAndStoreBody(Jimple.v(),
                          new StoredBody(ClassFile.v()),
                          BuildJimpleBodyOption.NO_PACKING);

  if (i == m.length)
    {
      System.out.println("Method not found:  " + met +
                         " in " + clsStr);
      System.exit(0);
    }

  method = (SootMethod)m[i];
  bd.resolveFor(method);
  method = Inline.inline(method);
  method.getBody(Jimple.v()).printTo(new PrintWriter(System.out,
                                                     true),
                                     BuildJimpleBodyOption.NO_PACKING);
  bm = new BlockMap(method, 0);
  residual.setLocals(bm.getLocals());
  residualToOriginalStmts = new HashMap();
```

```
}

/**
 * This constructs a new specializer that uses the class and
 * method and make the block map.
 *
 * @param c the class
 * @param met the name of the method
 */
public ABPS(SootClass c, String met)
{
  Object m[];
  int i;

  cls = c;
  cls.resolveIfNecessary();
  m = cls.getMethods().toArray();
  for (i = 0; i < m.length &&
         !((SootMethod)m[i]).getName().equals(met); i++);
  BuildAndStoreBody bd =
    new BuildAndStoreBody(Jimple.v(),
                          new StoredBody(ClassFile.v()),
                          BuildJimpleBodyOption.NO_PACKING);

  if (i == m.length)
    {
      System.out.println("Method not found:  " + met +
                         " in " + c.getName());
      System.exit(0);
    }

  method = (SootMethod)m[i];
  bd.resolveFor(method);
  method = Inline.inline(method);
  method.getBody(Jimple.v()).printTo(new PrintWriter(System.out, true),
                                     BuildJimpleBodyOption.NO_PACKING);
  bm = new BlockMap(method, 0);
  residual.setLocals(bm.getLocals());
  residualToOriginalStmts = new HashMap();
}

/**
 * Evaluates the method that the abps is specializing.
 *
 * @param args array of abstractions for the parameters of the
 * method
 * @param sign the signature to be used
```

```
 * @param pi the pi to use
 * @param theta the theta to use
 * @param init the table of initial abstractions
 *
 * @return The result of the method.
 */
public Abstraction eval(Abstraction args[], Signature sign,
Pi pi, Theta theta, Table init)
{
  BasicBlock bb;
  int i;
  Index ind;
  Iterator vars;
  ABPSArgs evalArgs = new ABPSArgs();
  List list;
  State state;
  Store store;
  BasicBlock currentBB;

  // Sets up the arguments to be used during specialization
  evalArgs.abps = this;
  evalArgs.signature = sign;
  evalArgs.pi = pi;
  evalArgs.theta = theta;

  // Initializes the store
  {
    store = new Store();

    for (i = 0; i < method.getParameterCount(); i++)
      {
        store.add(new EVariable("@parameter" + i,
                                method.getParameterType(i)),
                  args[i]);
      }

    vars = bm.getLocals().iterator();
    for (; vars.hasNext();)
      {
        EVariable v = new EVariable((Local)vars.next());
        if (init.lookup(v.getName()) != null)
        store.add(v, (EValue)init.lookup(v.getName()));
        else
          store.add(v);
      }

    System.out.println(store.vars());
```

```
  System.out.println(store);
}

// Create the initial state and index
state = new State(bm.getInit(), store.copy());
ind = pi.pi(state);

// Initializes the cache and skeleton
if (!state.getStore().equals(c.lookup(pi.pi(state))))
  {
    residual.setInit(ind);
    c.update(ind, theta.theta(state.getStore().copy(), c.lookup(ind)));
    S.start(ind);
  }

//      System.out.println(S);
//      System.out.println(c);
//System.out.println(bm);

// While there are still nodes to specialize
while ((ind = S.next()) != null)
  {
    store = c.lookup(ind).copy(); // Get the store
    bb = (BasicBlock)bm.get(ind.baseIndex()); // Get the block
    currentBB = new BasicBlock(ind); // Create the residual block
    evalArgs.basicblock = currentBB; // Set the arguments
    list = evalBasicBlock(bb, store, evalArgs); // Evaluate the block

    residual.put(ind, currentBB); // Put the residual block in the map

    //   System.out.println("Residual Block\n" + currentBB);

    //   System.out.println("Next states:  " + list);

    // For all the states returned, update the cache, skeleton
    Iterator states = list.iterator();
    while (states.hasNext())
      {
        state = (State)states.next();
        store = theta.theta(state.getStore(),
                            c.lookup(pi.pi(state)));


        if (!state.getStore().equals(c.lookup(pi.pi(state))))
          {
            c.update(pi.pi(state), store);
            S.makeArc(ind, pi.pi(state), false);
```

```
            }
          else
            S.makeArc(ind, pi.pi(state));
        }

      S.mark(ind);

      //System.out.println(c);
      //System.out.println(S);
      //System.out.println(currentBB);
    }

  //      System.out.println(residual);

  return null;
}

/**
 * Specializes a basic block.
 *
 * @param bb the basic block to specialize
 * @param store the store to use
 * @param the arguments
 *
 * @return The list of next states.
 */
protected List evalBasicBlock(BasicBlock bb, Store store, ABPSArgs args)
{
  int i, op, np;
  Iterator iterator;
  Stmt s;
  List list = null;
  List stmtResult = null;
  List stores = new VectorList();
  State state;
  Pi pi = args.pi;
  BasicBlock currentBB = args.basicblock;

  op = 0;

  // For all the statements
  for (i = 0; i < bb.size(); i++)
    {
      s = bb.get(i);          // Get the statement
      //   System.out.println("Evaluating statement:  " + s);
      stmtResult = evalStmt(s, store, bb, args);// Evaluate it
      np = currentBB.get().size(); // update the maps of res to orig
```

```
    for (; op < np; op++)
      residualToOriginalStmts.put(currentBB.get(op), s);
  }

// If there are no next states
if (stmtResult == null)
  {
    list = new VectorList(); // Make an empty list
    list.add(new State(bb.getSuccs(0), store.copy()));
    if (bb.size() == 0)
      {
        if (bb.getSuccs().size() > 0)
          {
            currentBB.addSuccs(pi.pi(new State(bb.getSuccs(0),
                                               store.copy())));
          }
      }
  }
else
  {
    // Else make them all states
    iterator = stmtResult.iterator();
    while (iterator.hasNext())
      {
        state = (State)iterator.next();
        if (state.getIndex() != null)
          {
            if (list == null)
              list = new VectorList();
            list.add(state);
          }
      }

    // If the results are null then make add the stuff
    if (list == null)
      {
        list = new VectorList();
        if (bb.getSuccs().size() > 0)
          {
            list.add(new State(bb.getSuccs(0), store.copy()));
            currentBB.addSuccs(pi.pi(new State(bb.getSuccs(0),
                                               store.copy())));
          }
      }
  }

return list;
```

```
}

/**
 * Evaluates an expression.
 *
 * @param v the expression to specialize.
 * @param store the store to use
 * @param the arguments
 *
 * @return A specialization value for the expression.
 */
public SpecValue evalExpr(Value v, Store store, final ABPSArgs args)
{
  Signature sign = args.signature;
  args.store = store;
  Signature ns = sign.newSignature(args);
  //      System.out.println("\tEvaluating:  " + v);
  v.apply(ns);                // Call the signature to specialize
  //      System.out.println("\t" + ns.getResult());
  return (SpecValue)ns.getResult();
}

/**
 * Gets the l expression
 *
 * @param v the expression
 *
 * @return The variable it represents
 */
protected EVariable evalLExpr(Local v)
{
  return EVariable.convert(v);
}

/**
 * Specializes a statement.
 *
 * @param s the statement to specialize
 * @param store the store to use
 * @param bb the basic block it is from
 * @param args the arguments
 *
 * @return A list of next states.
 */
protected List evalStmt(Stmt s, final Store store,
                        final BasicBlock bb, final ABPSArgs args)
{
```

```
AbstractStmtSwitch sw;
final Signature sign = args.signature;
final Pi pi = args.pi;
final Theta theta = args.theta;
final BasicBlock currentBB = args.basicblock;

args.abstraction = null;

sw = new AbstractStmtSwitch()
  {
    // The identent statement.  Works much like an assignment.
    public void caseIdentityStmt(IdentityStmt s)
      {
        EVariable l = evalLExpr((Local)s.getLeftOp());
        List list = new VectorList();
        SpecValue sv = evalExpr(s.getRightOp(), store, args);
        Value v;

        if (sv instanceof ExprSpecValue)
          {
            ExprSpecValue esv = (ExprSpecValue)sv;
            v = esv.getValue().lift();
            if (v == null)
              {
                currentBB.addStmt(Jimple.v()
                                  .newIdentityStmt((Local)s.getLeftOp(),
                                                   esv.getExpr()));
              }
            else
              {
                currentBB.addStmt(Jimple.v()
                                   .newAssignStmt((Local)s.getLeftOp(),
                                                  v));
              }
          }
        else
          if (sv instanceof StmtSpecValue)
            {
              StmtSpecValue ssv = (StmtSpecValue)sv;
              Iterator it = ssv.getStmts().iterator();
              while (it.hasNext())
                currentBB.addStmt((Stmt)it.next());

              it = ssv.getBoxes().iterator();
              while (it.hasNext())
                ((ValueBox)it.next()).setValue(s.getLeftOp());
            }
```

```
    list.add(new State(null, store.update(l, sv.getValue())));
    setResult(list);
  }

// Assignment statement.  It first specializes the expression
// and then checks the result.  If it is an expression
// specialization value, then it make the residual assignment
// with the residual expression and updates the store.  If
// operation inlining is done, it adds the code and fills the
// boxes with the variable assigned to.  It then updates the
// store.
public void caseAssignStmt(AssignStmt s)
  {
    EVariable l = evalLExpr((Local)s.getLeftOp());
    args.abstraction = (Abstraction)store.lookup(l);
    List list = new VectorList();
    SpecValue sv = evalExpr(s.getRightOp(), store, args);
    Value v;

    if (sv instanceof ExprSpecValue)
      {
        ExprSpecValue esv = (ExprSpecValue)sv;
        v = esv.getValue().lift();
        if (v == null)
          {
            currentBB.addStmt(Jimple.v().
                              newAssignStmt((Local)s.getLeftOp(),
                                            esv.getExpr()));
          }
        else
          {
            currentBB.addStmt(Jimple.v()
                              .newAssignStmt((Local)s.getLeftOp(),
                                             v));
          }
      }
    else
      if (sv instanceof StmtSpecValue)
        {
          StmtSpecValue ssv = (StmtSpecValue)sv;
          Iterator it = ssv.getStmts().iterator();
          while (it.hasNext())
            currentBB.addStmt((Stmt)it.next());

          it = ssv.getBoxes().iterator();
          while (it.hasNext())
```

```
            ((ValueBox)it.next()).setValue(s.getLeftOp());
        }
      if (store.lookup(l) != null &&
          ((Abstraction)store.lookup(l)).ID != sv.getValue().ID)
        throw new RuntimeException("Assignment from one " +
                                   "abstraction to another in "
                                   + s + " with " +
                                   store.lookup(l).getClass() +
                                   " and " +
                                   sv.getValue().getClass() +
                                   ".");
      else
        list.add(new State(null, store.update(l, sv.getValue())));
      setResult(list);
    }

  // Goto statement.  It just updates the next list.
  public void caseGotoStmt(GotoStmt s)
    {
      List list = new VectorList();
      State state = new State(bb.getSuccs(0), store);

      // Check out the goto
      currentBB.addSuccs(pi.pi(state));

      list.add(state);
      setResult(list);
    }

  // If statement.  It evaluates the conditional and gets an if
  // specialization value back.  It then fills the boxes and
  // adds the code.  It then uses the value to determine whether
  // to the next state(s) and returns them.
  public void caseIfStmt(IfStmt s)
    {
      List list = new VectorList();
      SpecValue sv = evalExpr(s.getCondition(), store, args);
      State state;
      if (sv instanceof IfSpecValue)
        {
          IfSpecValue ssv = (IfSpecValue)sv;
          Iterator it = ssv.getStmts().iterator();
          Local tmp = args.abps.getLocal("bool$tmp$");
          while (it.hasNext())
            currentBB.addStmt((Stmt)it.next());

          it = ssv.getBoxes().iterator();
```

```
        while (it.hasNext())
          ((ValueBox)it.next()).setValue(tmp);

        if (ssv.getValue().same(BooleanAbstraction.TRUE))
          {
            state = new State(bb.getSuccs(0),
                              ssv.getTrueStore());
            list.add(state);
            currentBB.addSuccs(pi.pi(state));
          }
        else
          if (ssv.getValue().same(BooleanAbstraction.FALSE))
            {
              state = new State(bb.getSuccs(1),
                                ssv.getFalseStore());
              list.add(state);
              currentBB.addSuccs(pi.pi(state));
            }
          else
            {
              list.add(new State(bb.getSuccs(0),
                                 ssv.getTrueStore()));
              list.add(new State(bb.getSuccs(1),
                                 ssv.getFalseStore()));

              currentBB.addSuccs(pi.pi((State)list.get(0)));
              currentBB.addSuccs(pi.pi((State)list.get(1)));
            }
      }
    else
      throw new RuntimeException("Unhandled spec value in if."
                                 + sv + " " + sv.getClass());

    setResult(list);
  }

// Return void.  Adds the residual and sends back an empty
// next list.
public void caseReturnVoidStmt(ReturnVoidStmt s)
  {
    currentBB.addStmt(Jimple.v().newReturnVoidStmt());
    setResult(new VectorList());
  }

// Return expr.  Specializes the expression and creates a
// residual return and sets no next states
public void caseReturnStmt(ReturnStmt s)
```

```
        {
          ExprSpecValue sv =
            (ExprSpecValue)evalExpr(s.getReturnValue(), store,
                                    args);

          mergeResult(sv.getValue(), theta);

          currentBB.addStmt(Jimple.v().newReturnStmt(sv.getExpr()));
          setResult(new VectorList());
        }

      // Default case.
      public void defaultCase(Object s)
        {
          throw new RuntimeException("??\t" + s + "\t" + s.getClass());
        }
    };
  s.apply(sw);                // Apply the different statemens

  return (List)sw.getResult();
}

/**
 * Gets the initial index from the block map.
 */
public Index getInitIndex()
{
  return bm.getInit();
}

/**
 * Creates a new method from the residual block map.
 */
public SootMethod getMethod()
{
  SootMethod res = residual.createMethod(method.getName() + "$abps",
                                         method.getParameterTypes(),
                                         method.getReturnType());
  if (cls.declaresMethod(res.getName(),
                         res.getParameterTypes()))
    cls.removeMethod(cls.getMethod(res.getName(),
                                   res.getParameterTypes()));
  cls.addMethod(res);

  setupTable((JimpleBody)method.getBody(Jimple.v()),
             (JimpleBody)res.getBody(Jimple.v()));
```

```
  return res;
}

/**
 * Gets the list of residual statements.
 */
public List getResidualStatements()
{
  return residual.collapse();
}

/**
 * Gets the residual initial index.
 */
public Index getResInitIndex()
{
  return residual.getInit();
}

/**
 * Merges the results of the method.
 */
protected void mergeResult(Abstraction a, Theta theta)
{
  if (result == null)
    result = a;
  else
    result = theta.merge(result, a);
}

/**
 * Gets the specialized result of the method.
 */
public Abstraction result()
{
  return result;
}

/**
 * Sets the initial store.
 *
 * @param p the store to set to.
 */
public void setStore(Store p)
{
  initStore = p;
}
```

```
/**
 * Gets the particular local from the residual method.
 *
 * @param name the name of the local
 */
public Local getLocal(String name)
{
  return residual.getLocal(name);
}

/**
 * Sets up the table between the original label and residual label.
 *
 * @param o the original jimple body
 * @param r the residual jimple body
 */
public void setupTable(JimpleBody o, JimpleBody r)
{
  Map ostn = new HashMap();
  Map rstn = new HashMap();

  // Create statement name table
  {
    Iterator boxIt = o.getUnitBoxes().iterator();

    Set labelStmts = new HashSet();

    // Build labelStmts
    {
      while(boxIt.hasNext())
        {
          UnitBox box = (UnitBox) boxIt.next();
          Stmt stmt = (Stmt) box.getUnit();

          labelStmts.add(stmt);
        }
    }

    // Traverse the stmts and assign a label if necessary
    {
      int labelCount = 0;

      Iterator stmtIt = o.getStmtList().iterator();

      while(stmtIt.hasNext())
        {
```

```
        Stmt s = (Stmt) stmtIt.next();

        if(labelStmts.contains(s))
          ostn.put(s, "label" + (labelCount++));
      }
  }
}

// Create statement name table
{
  Iterator boxIt = r.getUnitBoxes().iterator();

  Set labelStmts = new HashSet();

  // Build labelStmts
  {
    while(boxIt.hasNext())
      {
        UnitBox box = (UnitBox) boxIt.next();
        Stmt stmt = (Stmt) box.getUnit();

        labelStmts.add(stmt);
      }
  }

  // Traverse the stmts and assign a label if necessary
  {
    int labelCount = 0;

    Iterator stmtIt = r.getStmtList().iterator();

    while(stmtIt.hasNext())
      {
        Stmt s = (Stmt) stmtIt.next();

        if(labelStmts.contains(s))
          rstn.put(s, "label" + (labelCount++));
      }
  }
}

residualToOriginalLabels = new HashMap();

Iterator it = rstn.keySet().iterator();
while (it.hasNext())
  {
    Object o1, o2;
```

```
        o1 = it.next();
        o2 = residualToOriginalStmts.get(o1);
        if (o2 != null &&
            ostn.get(o2) != null)
          residualToOriginalLabels.put(rstn.get(o1),
                                       ostn.get(o2));
      }

  }

  /**
   * Prints the table.
   */
  public void printTable()
  {
    int i, count = residualToOriginalLabels.keySet().size();

    for (i = 0; i < count; i++)
      if (residualToOriginalLabels.get("label" + i) == null)
        count++;
      else
        System.out.println("label" + i + "\t" +
                           residualToOriginalLabels.get("label" + i));
  }

  public String toString()
  {
    return initStore + "\n" +
      c + "\n" + S + "\n" +
      residual + "\n" + result;
  }
}


//Main.java
package edu.ksu.cis.bandera.abps;

import java.io.*;
import edu.ksu.cis.bandera.abps.*;
import edu.ksu.cis.bandera.abps.lib.*;
import edu.ksu.cis.bandera.jext.*;
import edu.ksu.cis.bandera.util.*;
import ca.mcgill.sable.soot.jimple.Jimple;
import ca.mcgill.sable.soot.StoredBody;

/**
 * Main class and method to run ABPS.  It takes in the class name and
```

```
 * method name on the command line.  It can also take options to
 * determine polyvariant analysis, operation inlining, and whether the
 * output should be pure Jimple.
 *
 * @author <a href="mailto:laubach@cis.ksu.edu">Shawn Laubach</a>
 *
 * @version 0.1
 */
public class Main
{

  public static void main(String args[])
  {
    String cls, method;
    ABPS abps;
    Abstraction ABPSargs[];
    int i;
    Table table = new Table();

    if (args.length < 2)
      {
System.out.println("Main class method [options]");
return;
      }

    cls = args[0];
    method = args[1];

    for (i = 2; i < args.length; i++)
      if (args[i].equals("-inline"))
ABPS.inlineAbstractions = true;
      else
        if (args[i].equals("-poly"))
          ABPS.monovariant = false;
        else
          System.out.println("Unknown option:  " + args[i]);

    try {
    // ABPS takes the class and method
    // This can be changed to take a sootclass, abps takes both
    abps = new ABPS(cls, method);

    // This is where you populate the method parameters
    // First set of the size of the array.  It must equal the number
    // of parameters.
    ABPSargs = new Abstraction[0];
    //    ABPSargs[0] = SignsAbstraction.T;
```

```
    // Then put in the arguments into the positions of an array.  The
    // first one is an example of declaring an actual token from the
    // abstraction.  The second is how to have it abstract a value.
    //    ABPSargs[0] = SignsAbstraction.T;
    //    ABPSargs[1] = SignsAbstraction.empty().abs(4);

    //  You can initialize other variables in the ccde.  Just call add
    //  on the table with adds a name / value pair into the table.
    //  This is looked up when the initial store is created.
    table.add("writerPresent", BooleanAbstraction.empty().abs(0));
    table.add("req", Range04Abstraction.empty().abs(0));
    table.add("activeReaders", ZeroPosAbstraction.empty().abs(0));

    // Call eval with the arguments, a signature, a pi, and theta.
    // The only changes to theta you'll want to make is to add more
    // abstractions if you are adding abstractions.  Pi has three
    // options that you can change by hand in the constructor between
    // monovariant, maximally polyvariant, and limited polyvariance on
    // certain variables.  You could add other features but all this
    // is currently done by hand.  The signature was computer
    // generated with the addition of the package name and the
    // handling of static invoking, which currently returns T.
    abps.eval(ABPSargs, new SimpleSignature(),
              new SimplePi(), new SimpleTheta(),
              table);

    // abps.getMethod() gets the new method created by abps.  It is
    // named method + "$abps".  You could add it back to the class
    // file or just pass it on depending on the information you need.
    // The rest gets the body and prints it out to the screen
    abps.getMethod().getBody(Jimple.v())
      .printTo(new PrintWriter(System.out,
                               true), 0);
    abps.printTable();
    } catch (Exception e) {
      System.out.println(e);
      e.printStackTrace();
    }
  }
}


//Signature.java
package edu.ksu.cis.bandera.abps;

import ca.mcgill.sable.soot.jimple.*;
```

```
import ca.mcgill.sable.soot.*;
import ca.mcgill.sable.util.*;

import edu.ksu.cis.bandera.prog.*;

/**
 * This is the base class for all signatures.  It makes sure they
 * inherit from the proper switches and then has the proper methods.
 *
 * @author <a href="mailto:laubach@cis.ksu.edu">Shawn Laubach</a>
 *
 * @version 0.1
 */
public class Signature extends AbstractJimpleValueSwitch
  implements Cloneable
{
  protected volatile ABPSArgs args; // The arguments being passed around

  /**
   * The current working class.
   */
  public static SootClass workingClass;

  /**
   * Creates a new copy of the signature.
   */
  protected Object clone()
  {
    try {
      return super.clone();
    } catch (Exception e) {
      e.printStackTrace();
      return null;
    }
  }

  /**
   * This handles the default case.  It throws an exception because it
   * must be overridden.
   */
  public void defaultCase(Object v)
  {
    throw new RuntimeException("Unhandle expression:  " + v + " " +
                               v.getClass());
  }

  /**
```

```
    * Creates a new signature with the arguments set.
    *
    * @param a the arguments
    *
    * @return A new signature.
    */
   public Signature newSignature(ABPSArgs a)
   {
     Signature sig = (Signature)clone();
     sig.args = a;
     return sig;
   }
}
```

VITA

Shawn M. Laubach

Candidate for the Degree of

Master of Science

Thesis: ABSTRACTION-BASED PROGRAM SPECIALIZATION

Major Field: Computer Science

Biographical:

Personal Data: Born in Landstuhl, West Germany on October 21, 1974, the son of Paul M. and Rhonda K. Laubach, and husband of Angela R. Laubach.

Education: Graduated form Okeene High School, Okeene, Oklahoma and Oklahoma School of Science and Mathematics, Oklahoma City, Oklahoma, in May of 1993; received Bachelor of Science degree in Computer Science at the Computer Science Department at Oklahoma State University, Stillwater, Oklahoma in May 1997; completed the requirements for the Master of Science degree in Computer Science at the Computer Science Department at Oklahoma State University in July 1999.

Experience: Employed by Oklahoma State University as a Graduate Assistant re-engineering software for Tinker Air Force Base from October 1996 to July 1997; employed by Oklahoma State University as a Research Assistant from August 1997 to July 1998; employed by Oklahoma State University as a Teaching Assistant from August 1998 to December 1998; employed by Kansas State University as a Research Assistant from August 1998 to July 1999.