

Verification of temporal properties involving multiple interacting objects

by

Nomair A. Naeem

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2013

© Nomair A. Naeem 2013

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Defects that arise due to violating a prescribed order for executing statements or executing a disallowed sequence of statements can be hard to detect since the sequence is often spread over multiple functions and source code files. In this dissertation, we develop a verification tool which uses a sound and precise static analysis to verify temporal specifications that can involve multiple objects.

Statically analyzing properties that involve multiple objects requires two separate abstractions; one that abstracts the objects in the program and the second which abstracts the state of a group of objects. We present two such abstractions. Objects are abstracted using a storeless heap abstraction. This provides flow-sensitive tracking of individual objects along control flow paths and precise may-alias information. The state abstraction leverages the object abstraction to abstract the state of a group of related objects. We prove these abstractions to be sound with respect to the concrete operational semantics of tracematches, the AspectJ construct that we use to specify temporal properties.

We use the IFDS algorithm, an interprocedural, context-sensitive and flow-sensitive data flow analysis algorithm, to implement an analysis that computes the object and state abstractions. Since the original IFDS algorithm is not directly suitable for domains involving objects and pointers, we develop four extensions to the original IFDS algorithm. We present results of an empirical study to measure the precision of the analysis. For the selected benchmarks and tracematches, the analysis successfully confirms 42% of the test cases to not violate the specified property. Overall, the analysis guarantees that 89% of statements of interest that could be violations are not.

The performance of the analysis is improved through the use of two types of method summaries. Callee summaries guarantee that using the summary instead of flow-sensitive analysis of the callee does not degrade the precision of the abstraction at the callsite for the callee. For further performance gains, caller summaries that make conservative assumptions for aliasing between parameters of a function call are used. We present results from empirically evaluating the use of these summaries for the object analysis. The results show that although caller summaries may theoretically reduce precision, empirically they do not. Furthermore, on average, using callee and caller summaries reduces the running time of the object analysis by 27% and 96%, respectively.

Finally, to make the analysis practical for use in the development life cycle, we present a verification tool to configure the analysis and visualize the results. The tool provides a number of configuration options to run the analysis. The analysis results are presented in a list displaying statements flagged as possible violations of a property and, for each violation, the sequence of events (statements) that lead to this violation.

Acknowledgements

First, I would like to thank my advisor, Professor Ondrej Lhotak. Ondrej has been a guide, a mentor and someone I could always rely on to share my concerns. I could not have asked for a better advisor. His insights into complex technical problems and his ability to explain concepts never ceases to amaze me. His flexibility in letting me pursue my teaching interests and his concern for my well-being and success have been a great strength. It is indeed my honour to be his first graduating PhD.

I owe much, in terms of my academic achievements, to Professor Laurie Hendren from McGill University. Thank you for introducing me to the wonderful world of programming languages and compilers. It is incredible to think that a last minute decision to take her Compiler Design course would lead to a PhD in this field.

I would like to thank Professor Gordon Cormack, Professor Frank Tip, Professor Krzysztof Czarnecki and Professor Atanas (Nasko) Rountev (Ohio State University) for serving on my thesis committee. Their valuable feedback and suggestions has greatly helped in improving the quality of this work. I would also like to thank the department's administrative staff (especially Margaret, Paula, Wendy, Jessica and Helen) for always helping me despite short notices. A special thanks to my colleagues at the PLG Lab with whom I have spent countless hours chatting, ranting and discussing things of great and no importance.

I would also like to thank Dr. Steffen Roller and Tim Lehan from OpenText for giving me the opportunity to work on, and build, an industrial programming language and compiler. I have fond memories of the time spent with my R&D team at OpenText.

I must also thank Professor Khwaja Masud, my high school mathematics teacher. Though he is not amongst us anymore, he lit a spark in me that eventually led to this degree.

On a personal note, one person that has always stood rock solid beside me is my wife, Mariam. Her unconditional support and love made this journey much easier. I would also like to thank my in-laws, Arzoo Fatima and Mati-ur-Rasool, for letting their daughter marry a graduate student. My parents are my inspiration. Both holding PhDs, they never pushed me and just asked me to “try my best and forget the rest”. My mom's prayers, her joy at hearing of completion of my PhD milestones and her love was all the encouragement I ever needed. My dad's words of wisdom, his tactful ways of giving advice without making it sound like advice, and his valuable life lessons made me the person I am. My sisters, Muznah, Anika and Maliha have always been the best sisters in the world (well most of

the time). And finally, my son Mekaeel, who brings me immense joy and makes me look forward to the future.

My life would not have been the same without my truly great friends. Ahmar K, Ahmer A., Bilal, Farhan, Farheen, Hassan, Kamran, Mariam A., Nabeel A., Nabeel B., Omar Z., Raqib, Rehan, Sajjad, Salman, Sumair, Mohsin, Umar F., Umar S., Usman, Umair, Uzma, Waqqas, Zara and Zunaira, thank you for being such an important part of my life. I will forever remember and cherish the wonderful times we have shared.

This work was financially supported by NSERC.

Dedication

This thesis is dedicated to my parents,
Dr. Naeem Tariq (baba)
Dr. Shahida Naeem (amaa)

my wife,
Mariam Rasool

my son,
Mekaeel Naeem

and my sisters,
Muznah, Anika and Maliha

Table of Contents

List of Tables	x
List of Figures	xi
1 Introduction	1
1.1 Challenges	3
1.2 Contributions	6
1.3 Organization	7
2 Tracematches	9
2.1 Declarative Semantics	10
2.2 Original Operational Semantics	11
2.3 A Lattice-Based Operational Semantics	16
2.4 Summary	23
3 Static Abstraction	25
3.1 Intermediate Representation	25
3.2 Object Abstraction	26
3.3 Tracematch Abstraction	32
3.4 Related Work	42
3.5 Summary	46

4	Extensions to IFDS	47
4.1	The Original IFDS Algorithm	48
4.2	Running Example: Type Analysis	52
4.3	Demand Construction of the Supergraph	55
4.3.1	Eliminating the SummaryEdge Table	58
4.3.2	Empirical Evaluation	61
4.4	Return Flow Functions	62
4.5	Static Single Assignment (SSA) Form	65
4.5.1	Example of precision loss	67
4.6	Exploiting Structure in the Set Dom	68
4.6.1	Empirical Evaluation	70
4.7	Related Work	71
4.8	Using the Extended IFDS Algorithm for Analyzing Tracematches	75
4.9	Concluding Remarks	76
5	Implementation	77
5.1	Collecting Useful Update Shadows	83
5.2	Empirical Evaluation	85
5.2.1	Discussion of Results	87
6	Optimizations	90
6.1	Alias Set Analysis	92
6.2	Callee Summaries	95
6.2.1	Computing Callee Summaries	96
6.2.2	Using Callee Summaries	101
6.3	Caller Summaries	104
6.4	Experiments	106
6.4.1	Shadow Statistics	107

6.4.2	Efficiency	108
6.4.3	Tracematch Analysis Precision	110
6.4.4	Fine-grained Precision Metrics	110
6.5	Related Work	112
6.6	Concluding Remarks	113
7	Presenting Analysis Output	115
7.1	Motivation	115
7.2	TMAalysis: an Eclipse plugin	116
7.2.1	Configuration	116
7.2.2	Running the Analysis	120
7.2.3	Visualization of Results	120
7.3	Conclusion	122
8	Conclusion and Future Work	123
8.1	Abstractions	123
8.2	Static Analysis	124
8.3	Precision	124
8.4	Efficiency	125
8.5	Verification Tool	125
8.6	Future Work	126
	APPENDICES	128
	A Proofs	129
	References	156
	Index of Symbol Definitions	162

List of Tables

4.1	Benchmark Characteristics	62
6.1	Statically reachable methods and time to compute callee summaries	107
6.2	Percentage of reachable methods in M^* and S	108
6.3	Effect of using summaries on the time to compute the alias set analysis	109
6.4	Fine-grained Precision metrics	111

List of Figures

1.1	Tracematch example: iterator safety	3
1.2	Tracematch source code	4
2.1	Example: Declarative semantics of tracematches	12
2.2	Tracematch source code	12
2.3	Example: Operational semantics of tracematches	17
2.4	Concrete Binding Lattice Bind	18
2.5	Example automaton	20
2.6	Transfer function for $\mathbf{tr} \langle a, b \rangle$ in lattice-based operational semantics	21
2.7	Example: Lattice-based operational semantics for tracematches	22
3.1	Transfer functions for $s \neq \{\mathbf{tr}, \mathbf{body}\}$ in lattice-based operational semantics	26
3.2	Transfer function for the object abstraction	29
3.3	Example statement sequence to illustrate transfer functions	30
3.4	Abstract Binding Lattice Bind [#]	33
3.5	Transfer functions for the tracematch state abstraction for $s \neq \mathbf{tr}(T)$	37
3.6	Computing reduced environments	39
3.7	Generalized compatibility predicate.	39
3.8	Transfer function for tracematch state abstraction for $s = \mathbf{tr}(T)$	40
3.9	Example: State abstraction for tracematches	43
4.1	Compact representation of functions and their composition	49

4.2	Original IFDS Algorithm reproduced.	50
4.3	Intraprocedural flow functions for the Variable Type Analysis	54
4.4	Extended IFDS Algorithm	56
4.5	Extended IFDS Algorithm without computing summary edges	59
4.6	Comparing the Exploded Supergraph and its reachable subgraph	63
4.7	Example: mapping information from caller back to caller	64
4.8	The effect on precision due to the choice of merge strategy at ϕ nodes.	67
4.9	Extended Propagate Procedure	70
4.10	Effect of taking advantage of subsumption relationships in D	72
5.1	Precision of IFDS on Dacapo Benchmark Suite	88
6.1	Sample code illustrating the use of callee and caller summaries	91
6.2	Interprocedural control flow graph with <i>call</i> , <i>return</i> and <i>CallFlow</i> edges.	93
6.3	Transfer functions for the alias set abstraction	94
6.4	Callee Summary for a callsite with target method m	96
6.5	Example illustrating the effect of a method call on alias sets in the caller.	97
6.6	Algorithm to compute callee escape summary (α_{esc}) for a method m	98
6.7	getSeededWorklist: algorithm to obtain escaped variables	99
6.8	Algorithm to compute the return value summary (α_{ret}) for a method m	100
6.9	Tracking whether the returned object has escaped	101
6.10	allEscaped: algorithm to compute variables that escape from a method m	102
6.11	Modified transfer functions for the alias set analysis using callee summaries.	103
6.12	Transfer functions using callee and caller summaries.	106
7.1	Configuration Screen	117
7.2	Example Input	120
7.3	Example Input	121
7.4	Visualization Screen	122

Chapter 1

Introduction

Software defects often occur when a chain of events leads the program into an undesirable state. Such undesirable temporal sequences of events can often be specified in the form of a temporal specification that the programmer must strive to avoid. For an object, a temporal specification can be expressed using tpestate [64]. At any time, the object is in some state, and the state changes when an operation is performed on the object. Many programming errors can be detected by checking whether undesirable states are reachable. A multitude of tpestate checking tools, both dynamic and static, have been developed [4, 8, 9, 15, 22, 23, 29, 30, 31, 32, 34, 35, 37, 47]. Temporal specifications can also be applied to express constraints on the interactions between software components. In this case, the specified protocol may involve multiple interacting objects from different components. Additionally, even within a software component, an object is not isolated; it interacts with other objects. Some newer specification mechanisms can express temporal properties of multiple objects [4, 15, 47, 32]. These formalisms are mainly intended for dynamic checking. This work focuses on developing a technique to formulate and implement the static analysis of such multi-object temporal specifications.

Such a static analysis has two classes of applications. First, it can be used for sound static program verification. The analysis is intended to be precise: in the ideal case, all possible violations are ruled out statically, and the program is therefore guaranteed to observe the specified protocol. However, it is not always possible to rule out all violations statically. In this case, the program can be instrumented with dynamic checks that report violations at run time. The second application of the static analysis is to reduce the overhead of these dynamic checks. If the analysis proves that some instrumentation points cannot possibly lead to a violation, no instrumentation is required at those points. Thus, the runtime overhead at those program points is reduced.

We have chosen tracematches [4] as the formalism for specifying the temporal properties to be checked. A tracematch specifies which operations are relevant to the specification, how the operations identify the objects involved, the sequence of operations leading to an undesirable state, and what should be done when a violation is detected at run time. For our analysis, tracematches have two advantages over similar formalisms. First, they are widely applicable because their semantics is intuitive and highly expressive compared to other regular-expression-based formalisms. A key issue in defining such formalisms is how to tease apart the interactions between operations on different objects; in some other systems, operations on different objects are not cleanly separated. Conceptually, a tracematch executes a separate copy of a finite automaton for every possible combination of runtime objects. While other systems require each automaton to bind all objects on the first state transition, tracematches do not have this restriction. Second, the semantics of tracematches has been formally specified, which allows us to formally prove that the static analysis soundly abstracts the semantics. The original tracematch paper motivates the design of a declarative semantics from the programmer’s point of view, then proves it equivalent to an operational semantics better suited for implementation [4]. The operations, and how they bind objects, are specified using AspectJ pointcuts, which are in widespread use and have a formal specification [6].

While the operational tracematch semantics is convenient for a dynamic implementation, it is difficult to abstract statically because it is defined in terms of manipulating and simplifying boolean formulas, a relatively complicated concrete domain. Thus, we have defined a new, equivalent semantics based on sets and lattices, which are more convenient to reason about and to abstract. We have proven the two semantics bisimilar. The static analysis uses a provably sound abstraction of the lattice-based semantics.

The formal definitions and correctness proofs are important because reasoning about interacting objects is subtle. Allan et al. wrote this about their dynamic implementation:

In our experience it is very hard to get the implementation correct, and indeed, we got it wrong several times before we formally showed the equivalence of the declarative and operational semantics.[4]

Similar pitfalls apply when defining a static analysis.

A key difference between our analysis and previous work on tpestate verification is that in a tracematch, tpestate is associated not with a single object, but with a group of objects. Existing work on tpestate verification (e.g. [29, 30]) generally uses some abstraction of objects and adds the current state to each abstract object. This approach cannot be applied when there is no single object to which the state can be attached. Thus,

our analysis uses two separate abstractions: the first models individual objects and the second models tracematch state of related groups of objects.

1.1 Challenges

The example in Figure 1.1 illustrates the kind of property that the analysis verifies. The method `flatten` takes a list of lists `in`, and adds all of their elements to the list `out`. The automaton besides the code checks that a list is not updated during iteration, and that every call to `next` on an iterator is preceded by a call to `hasNext`. A violation of the property causes the automaton to enter one of the final states. The tracematch associated with this automaton (shown in Figure 1.2) has two parameters, the list (c) and the iterator (i). The `next` and `hasNext` operations bind the iterator i , `update` binds the list c , and `makeiter` binds both. According to the declarative tracematch semantics, a copy of the automaton is made for every possible runtime pair of list and iterator. Each operation causes a transition in those automata consistent with the bindings. For example, the `update(c)` operation on runtime list object o_c causes an update transition in all automaton copies having o_c as their list c .

```

1 void flatten(List in, List out) {
2   Iterator it = in.iterator();
3   while(it.hasNext()) {
4     List l = (List) it.next();
5     Iterator it2 = l.iterator();
6     while(it2.hasNext()) {
7       Object o = it2.next();
8       out.add(o);
9     }
10  }
11 }

```

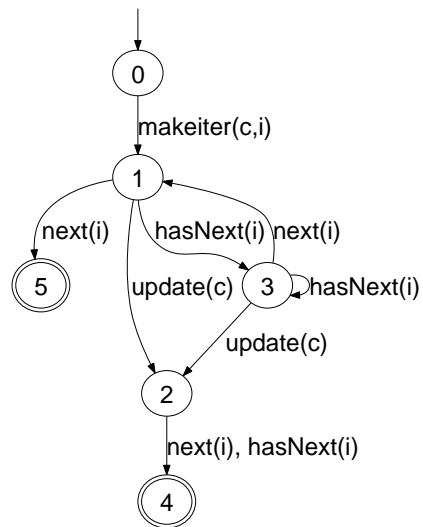


Figure 1.1: Tracematch example: iterator safety

```

1  tracematch(Collection c, Iterator i) {
2      sym makeiter after returning(i):
3          call(* Collection+.iterator()) && target(c);
4      sym next before: call(* Iterator+.next()) && target(i);
5      sym hasNext before: call(* Iterator+.hasNext()) && target(i);
6      sym update after : (call(* Collection+.add*(..)) ||
7          call(* Collection+.clear()) ||
8          call(* Collection+.remove*(..)) ) && target(c);
9
10     makeiter (hasNext+ next)* ( next | hasNext* update+ (next | hasNext) )
11     {
12         throw new RuntimeException(“Violated safety property.”);
13     }
14 }

```

Figure 1.2: Tracematch source code

Abstraction

Consider what information a static analysis needs to prove the absence of a violation. First, it needs precise may-alias information to determine that the list `out` updated in line 8 is not aliased with the list `in` or any of the lists it contains, over which the loops iterate. Interprocedural information is necessary because aliases may be made elsewhere; for example, the caller of the method could pass in the same list as both `in` and `out`. In fact, since the method could be called several times on different lists, context sensitivity is useful. In addition, the analysis must ensure that each call to `hasNext` occurs on the same iterator as the subsequent call to `next`. Although some have suggested using must-alias analysis, proving this fact requires more than just knowing that a pair of variables must be aliased. A must-alias analysis can prove that whenever execution reaches a given program point, two variables point to the same object. A must-alias analysis does not say anything about the values of variables at *different* times during execution. It would be difficult to extend the notion of must-aliasing to an unambiguous definition of the relationship between variables at different times. For example, it is *not* true that `it2` in line 6 always points to the same object as `it2` in line 7. When control flows from line 6 to line 7, `it2` continues to point to the same iterator, but when control flows from line 7 around the outer loop and back to line 6, the object to which `it2` points changes. Thus, a statement about the relationship between `it2` at line 6 and `it2` at line 7 would be ambiguous unless it somehow considered specific control flow paths between the two points. Instead, in order to reason about the objects pointed to by variables at different points in time, the analysis must

track the flow of individual objects along specific control flow paths.

Implementation

The requirements for context-sensitivity, precise may-aliasing and flow-sensitive tracking of individual objects make any implementation of the analysis computationally expensive. Therefore, in order to mitigate the high computational demands, an efficient and precise algorithm is required. One such algorithm is the Interprocedural Finite Distributive Subset (IFDS) algorithm [54]. IFDS is an efficient and precise, context-sensitive and flow-sensitive dataflow analysis algorithm for the class of problems that satisfy its restrictions. These include the classic bit-vector dataflow problems. Unfortunately the algorithm cannot be directly used for other interesting problems for which context- and flow-sensitivity would be useful, particularly problems involving objects and pointers. This is due to the following four restrictions in the original algorithm:

1. The analysis requires as input a graph where the number of nodes in the graph is approximately $|\text{Inst}| \times |\mathbf{Dom}|$ where Inst is the set of instructions in the program and \mathbf{Dom} is the analysis domain. Therefore, a practical restriction is that \mathbf{Dom} must be small. Domains for analyses modelling objects and pointers are often theoretically very large, making the original algorithm impractical to use due to the size of the graph that would be required as input.
2. The original IFDS algorithm provides limited information to functions modelling return flow from a procedure. In particular, information about the state before the procedure was called is not available within the return flow function. Many analyses require this information to map dataflow facts from the callee back to the caller.
3. Many dataflow analysis algorithms, IFDS included, can be less precise on a program in Static Single Assignment (SSA) form [20] than on the original non-SSA form of the program. This imprecision arises due to imprecisely modelling the semantics of ϕ instructions in SSA form.
4. In many analysis domains, particularly those for objects and pointers, elements of \mathbf{Dom} often subsume others. The original IFDS algorithm does not take advantage of such properties to reduce analysis time.

Performance

An abstraction for the objects in the program with the properties mentioned above is expensive to compute irrespective of how efficient the dataflow algorithm is. This is primarily due to two requirements. First, the analysis must be flow sensitive. Flow sensitivity takes

into account the order of instructions in the program to compute a result for each program point. This is useful so that precise results are available at program points that are of interest for verifying a property. For instance, in the example illustrated in Figure 1.1, precise information is needed at program points which involve either a `makeIter`, `next`, `hasNext` or `update` operation. Whereas precise information is required at these program points, it is not needed at all program points. In fact, there might be long segments of code where the results of the analysis (and therefore the precision of the results) are of no use. However, typical dataflow algorithms are either flow-sensitive or not i.e. there is no way to make the algorithm selectively flow sensitive even though that is exactly what is needed to improve performance.

The second requirement is that the abstraction be computed using an interprocedural dataflow algorithm. This is necessary to ensure that already computed precise information regarding the parameters and receiver of a function call are passed into the called function at a callsite. For example, only an interprocedural analysis would be able to ascertain whether the two parameters `in` and `out` are aliased when the `flatten` function from Figure 1.1 is called. Without the use of an interprocedural analysis, conservative worst case assumptions need to be made at method entry. In situations, where the running time of the analysis is a bigger concern an intraprocedural analysis could be substituted. However, this would cause a theoretical decrease in precision of the abstraction which might or might-not ripple into the analysis to verify temporal properties.

1.2 Contributions

This dissertation contributes to the use of static program analysis to verify temporal properties of objects. We use the `tracematch` construct to specify temporal properties. The inventors of the construct provided a declarative semantics and an operational semantics based on boolean formulas. Since boolean formulas are a difficult domain to abstract, we have developed a new operational semantics based on sets and lattices. We have proved the two operational semantics to be bisimilar.

We have developed two abstractions for the analysis. First, we compute an abstraction of the objects in the program and second, we use the object abstraction to compute an abstraction that models the state of objects. The object abstraction satisfies the following requirements that were highlighted in Section 1.1:

1. precise may-alias information,

2. precise context-sensitive interprocedural information, and
3. flow-sensitive tracking of individual objects along control flow paths.

To overcome the restrictions to the original IFDS algorithm highlighted in the previous section, we have made four extensions. First, we remove the restriction on the size of **Dom** by enabling the algorithm to compute only the reachable part of the graph. By dynamically computing just the reachable subset of the graph, IFDS can be used for domains which are theoretically large, but only a small subset of the domain is encountered during the analysis. Second, we extend the IFDS algorithm to expose the state before a procedure was called at the return sites of the procedure. Third, we extend the algorithm to handle ϕ statements in a semantically correct manner. Fourth, we provide the ability to exploit any structure in **Dom** to reduce analysis time.

We have successfully formulated the verification problem as an IFDS problem. The result of this analysis is simply an indication of how many possible violations of the specified temporal property exist for a given program. More useful would be the exact sequence of operations that lead to a possible violation of the property. To obtain these sequences of operations, we have implemented another analysis as an instance of the Interprocedural Distributive Environment (IDE) [60] algorithm.

We have used method summaries to improve the analysis time for computing the object abstraction. Callee summaries guarantee that foregoing flow-sensitive analysis of the callee will not degrade precision in the caller. For even better performance, caller summaries further reduce the number of methods analyzed flow-sensitively although with a theoretical loss in precision.

We have developed an Eclipse plugin that can be used to configure and execute our analysis for verifying temporal properties. Once the analysis has been executed, the plugin lists the potential violations found and provides a mechanism for users to navigate the sequence of operations leading to a violation.

1.3 Organization

The remainder of this dissertation is organized as follows. In Chapter 2 we provide a background of the tracematch construct, including the original declarative semantics and the operational semantics. We also present the new lattice-based operational semantics. Chapter 3 discusses the formulation of the object and state abstractions. The extensions to the IFDS algorithm and an experimental evaluation of these extensions on an example

Variable Type Analysis are reported in Chapter 4. The implementation of the abstractions using the IFDS algorithm and of the analysis to compute the sequences of operations leading to a violation using the IDE algorithm are discussed in Chapter 5. We also present an empirical evaluation of our analysis and a comparison with existing work. The optimization using method summaries is discussed in Chapter 6. Results from an evaluation of the effect of caller summaries on the precision of the analysis, and the use of summaries on the running time are also presented. Chapter 7 discusses the Eclipse plugin for the analysis. In Chapter 8 we conclude and mention promising future avenues of research. Existing related work falls into three categories: analyzing properties of programs by abstracting objects, the IFDS dataflow analysis algorithm and the use of summaries to improve performance. We include separate related work sections for each of these categories in Sections 3.4, 4.7 and 6.5.

Chapter 2

Tracematches

We have chosen tracematches [4] as the formalism for specifying the temporal properties to be checked. The tracematch construct was introduced in the *abc* compiler for AspectJ as a mechanism to perform runtime verification. To verify that a program conforms to a certain property, the property is first specified as a tracematch. Then, the program and the tracematch are compiled using the *abc* compiler. The compiler generates instrumented bytecode which can be executed to verify that the program does not violate the property at runtime.

For statically analyzing a property specified by a tracematch, the instrumented bytecode is the ideal starting point since the compiler has already identified, and instrumented, relevant operations and objects. The static analysis can detect this instrumentation and thereby determine relevant operations and objects. In this work, by using the tracematch construct, we were able to focus on the static analysis while relying on the *abc* compiler to catch the operations and objects relevant to the property.

Allan et al. [4] define a tracematch as follows:

Definition 1 *A tracematch is a triple $\langle F, A, P \rangle$, where*

F is a finite set of tracematch parameters,

A is a finite alphabet of symbols (operations), and

P is a regular language over A .

Figure 1.2 shows the source code that a programmer would write to define the example tracematch discussed in Chapter 1. This tracematch has two parameters, a `Collection c` and an `Iterator i`. Lines 2-8 define the four tracematch symbols. Each symbol is accompanied by an AspectJ pointcut that specifies where in the base code the symbol occurs. A pointcut may also bind objects from the base code to tracematch parameters. For example, the `makeIter` pointcut binds the target of the call (the collection) to `c` and the returned iterator to `i`. Line 10 defines the regular language of the tracematch and lines 11-13 provide the code to be executed when the tracematch matches at run time.

When writing a tracematch, the programmer specifies P using a regular expression. Internally within the *abc* compiler, P is represented as a non-deterministic finite automaton accepting the same language. To refer to this NFA, we use the customary notation $\langle Q, A, q_0, Q_f, \delta \rangle$, where Q is a finite set of states, A is the finite alphabet of tracematch symbols, $q_0 \in Q$ is the start state, $Q_f \subseteq Q$ is a set of final states, and $\delta \subseteq Q \times A \times Q$ is a transition relation.

A tracematch is applied to a program in an existing language such as Java or AspectJ. The program executes according to the semantics of the base language, but the dynamic tracematch implementation maintains additional state to keep track of the configuration of the tracematch. Allan et al. defined a declarative semantics of how tracematches ought to work, as well as an operational semantics that they proved equivalent [4].

Next, we review both of these semantics, formalizing a few details that were left implicit. We then define a new operational semantics based on sets and lattices which is more amenable to static analysis. Finally, we formally prove that the lattice-based operational semantics is bisimilar to the operational semantics of Allan et al. Thus, all three semantics are equivalent.

2.1 Declarative Semantics

The essential part of a tracematch is a regular expression over operations of interest (symbols). The dynamic tracematch implementation checks, for each suffix of the program trace, whether the suffix is a word in the language specified by the regular expression. Each such word is a *match* and causes the tracematch body to be executed. When a tracematch defines a safety property, each violation of the specified property is a match of the tracematch. Checking each suffix of the program trace ensures that the match (violation) is detected as soon as it occurs in the program.

Much of the expressive power of tracematches comes from their parameters, to which

symbols can bind specific objects. The tracematch body executes for each suffix of the trace that matches the specified regular expression with a consistent set of object bindings. The declarative semantics makes this precise: a separate *version* of the tracematch automaton is considered to be instantiated for each possible set of objects that could be bound to the tracematch parameters. These automaton versions run independently of each other. An automaton version makes a transition on an event in the trace if the parameters bound by the event are bound to the same objects that are associated with that automaton version. Whenever an automaton version reaches an accepting state, the tracematch body is executed; at that point, the automaton version is discarded.

We illustrate with an example. Figure 2.1 shows a possible trace of the events declared in the tracematch from Figure 1.2 which we have reproduced in Figure 2.2 for ease of reference. Each `hasNext` and `next` event binds an iterator object, `update` binds a list object and `makeiter` binds both a list and an iterator. We assume the program creates two list objects `x` and `y` and two iterator objects `f` and `g`. Thus, there are four possible ways in which these objects could be bound to the parameters, which correspond to the four automaton versions shown as columns in Figure 2.1. Each column includes only those events from the trace that are consistent with the object bindings of each version. The example trace results in matches of two automaton versions: the version with `c=x` and `i=f`, and the version with `c=y` and `i=g`. The first of these signals that the collection was modified while it was being iterated. The second signals two consecutive `next` events without an intervening `hasNext` event on the same iterator.

2.2 Original Operational Semantics

In the declarative semantics, the number of automaton versions that must be maintained is unbounded because the number of objects that could be created by the program is unbounded. This unboundedness hinders a practical dynamic implementation. Therefore, Allan et al. defined an equivalent operational semantics, which we now discuss.

The *abc* compiler includes a transformation that implements tracematch semantics at run time. This is done by inserting additional code, which we call *transition statements*, at each point in the base program where a tracematch symbol could match. In the dynamic implementation, the effect of each transition statement is to update the tracematch state to reflect the corresponding state transition and parameter bindings. The operational semantics is defined on the code that results after transition statements have been inserted.

The two instructions directly relevant to tracematches are `tr` $\langle a, b \rangle$ (*transition statement*) and `body` (*body statement*). Each transition statement contains a pair a, b where

Trace	c=x i=f	c=x i=g	c=y i=f	c=y i=g
makeiter(x,f)	makeIter			
hasNext(f)	hasNext		hasNext	
makeiter(y,g)				makeIter
next(f)	next		next	
hasNext(g)		hasNext		hasNext
update(x)	update	update		
next(g)		next		next
next(f)	next		next	
next(g)		next		next
	match	no	no	match

Figure 2.1: Example: Declarative semantics of tracematches. Column 1 shows the program trace. Columns 2 to 5 show automaton versions for different runtime objects bound to tracematch parameters.

```

1  tracematch(Collection c, Iterator i) {
2      sym makeiter after returning(i):
3          call(* Collection+.iterator()) && target(c);
4      sym next before: call(* Iterator+.next()) && target(i);
5      sym hasNext before: call(* Iterator+.hasNext()) && target(i);
6      sym update after : (call(* Collection+.add*(..)) ||
7          call(* Collection+.clear()) ||
8          call(* Collection+.remove*(..)) ) && target(c);
9
10     makeiter (hasNext+ next)* ( next | hasNext* update+ (next | hasNext) )
11     {
12         throw new RuntimeException(“Violated safety property.”);
13     }
14 }

```

Figure 2.2: Tracematch source code

$a \in A$ is one of the symbols of the tracematch and $b : F \hookrightarrow \mathbf{Var}$ is a partial map specifying the object to be bound to each tracematch parameter. The map b binds a subset of the parameters; any of the parameters may be left unbound. When $\mathbf{tr} \langle a, b \rangle$ is executed, each automaton version whose object bindings are consistent with the objects currently pointed to by the variables specified by b performs a transition on the symbol a .

In fact, Allan et al. [4] allow each transition statement to contain *multiple* transitions, each consisting of a pair $\langle a, b \rangle$. This is necessary because the placement of transition statements is determined according to AspectJ pointcuts, and it can happen that the pointcuts of multiple symbols match in the same place. When such a statement is executed, the tracematch non-deterministically follows the transitions specified by each individual pair. The semantics and our analysis fully handle this general though rare case. For clarity, we use the term *transition element* when referring to a pair $\langle a, b \rangle$ from a transition statement.

A body statement is generated immediately after every transition statement which contains a transition element $\langle a, b \rangle$ in which a is a symbol on which the tracematch automaton contains a transition into an accepting state. The effect of **body** is to find each automaton version that is in an accepting state, execute the tracematch body for it, and discard it.

The semantics of transition statements is defined in terms of a set \mathbf{Var} of variables in the base language and a set $\mathbf{Obj} \cup \{\perp\}$ of values that those variables can take. The symbol \perp denotes the special null value and \mathbf{Obj} denotes the set of all non-null values. We assume the presence of an environment $\rho : \mathbf{Env} \triangleq \mathbf{Var} \rightarrow \mathbf{Obj} \cup \{\perp\}$ that gives the value of each variable at each (dynamic) program point.

The operational semantics expresses tracematch state using boolean formulas. The literals of these formulas are true, false, and $(f = o)$, where $f \in F$ is any tracematch parameter and $o \in \mathbf{Obj}$ is any runtime value. A formula is constructed from these literals using the boolean connectives \wedge , \vee , and \neg . Let S denote the set of all formulas that can be expressed in this way. The concrete runtime state $\hat{\sigma} : Q \rightarrow S$ of a tracematch maintains one such formula for each state of the tracematch automaton¹. Intuitively, the formula associated with a state q is a predicate on tracematch bindings which is satisfied by the bindings of exactly those copies of the automaton that are in state q .

When a transition statement containing a single transition element $\langle a, b \rangle$ is executed in environment ρ , a boolean formula is generated that evaluates to true for tracematch

¹We use the ring superscript to indicate concepts in the original operational semantics, and we will use the same identifiers without the ring for the corresponding concepts in a new operational semantics that we propose in the next section.

bindings that are consistent with the objects bound in the transition element:

$$\dot{e}_0(b, \rho) \triangleq \bigwedge_{f \in \text{dom}(b)} (f = \rho(b(f)))$$

In [4], the notation $e(a)$ is used with the same meaning as $\dot{e}_0(b, \rho)$.

When the transition statement contains a set T of transition elements, the formula is a disjunction of the formulas for each element, since the tracematch non-deterministically executes all of the transition elements:

$$\dot{e}_T(T, \rho) \triangleq \bigvee_{b: \langle a, b \rangle \in T} \dot{e}_0(b, \rho)$$

Recall that a tracematch state $\dot{\sigma}$ conceptually represents the state of different automata with different bindings. At a transition, each automaton performs a transition if its bindings are consistent with the objects bound in the transition (i.e. $\dot{e}_0(b, \rho)$ is satisfied), or remains in its current state if its bindings are inconsistent (i.e. $\neg \dot{e}_0(b, \rho)$ is satisfied). Thus, the transition function is defined [4, 7] as:

$$\dot{e}[T, \rho](\dot{\sigma}) \triangleq \lambda i. \left(\bigvee_{a, j: \delta(j, a, i)} \dot{\sigma}(j) \wedge \dot{e}_T(T_a, \rho) \right) \vee \left(\dot{\sigma}(i) \wedge \bigwedge_{a \in A} \neg \dot{e}_T(T_a, \rho) \right)$$

where $T_a \triangleq \{b : \langle a, b \rangle \in T\}$.

The first clause on the right hand side of the function above deals with the case when a transition element $\langle a, b \rangle$ binds objects that are consistent with the bindings for some automaton in state j . Then, the automaton transitions to some state i as defined by the transition function for the tracematch state machine for state j and symbol a . Additionally, since the bindings for the automaton must be consistent with objects bound in the transition element, $\dot{e}_0(b, \rho)$ must be satisfied. When multiple transition elements are consistent with the bindings of the automaton, then, since the tracematch state represents each such automaton separately, the result is a disjunction of such clauses. The second clause deals with inconsistent bindings. In this case, the automaton stays in the current state i only if the bindings for the automaton are *not* consistent with the objects bound by the transition element, i.e., for a single transition element $\langle a, b \rangle$ the automaton stays in state i if $\neg \dot{e}_0(b, \rho)$ is satisfied. If there are multiple such transition elements, the automaton stays in the current state only if the bindings for the automaton are not consistent with the objects bounds in any of the transition elements.

Finally, a tracematch is defined to match when any suffix of the sequence of operations executed matches the specification. Thus, every automaton is considered to potentially be in the initial state at all times. Therefore, the transfer function for transition statements in the operational semantics is:

$$\langle \mathbf{tr}(T), \rho, \hat{\sigma} \rangle \xrightarrow{\circ} \hat{e}[T, \rho](\hat{\sigma}[q_0 \mapsto \text{true}])$$

where $\hat{\sigma}[q_0 \mapsto \text{true}]$ maps q_0 to true and every other state q to $\hat{\sigma}(q)$.

At the beginning of program execution, the tracematch state is initialized to false for all states $q \in Q$.

After every transition statement, if the formula for any final state is not false, the tracematch is said to *match* and its body is executed. When this happens, the formula is reset to false. These effects are expressed in the semantics of the **body** statement:

$$\langle \mathbf{body}, \rho, \hat{\sigma} \rangle \xrightarrow{\circ} \lambda q. \begin{cases} \hat{\sigma}(q) & \text{if } q \notin Q_f \\ \text{false} & \text{if } q \in Q_f \end{cases}$$

We illustrate with an example. Consider the program trace from Figure 2.1. The first event `makeiter(x, f)` is represented by the transition statement $\mathbf{tr} \langle \text{makeiter}, [c \mapsto x, i \mapsto f] \rangle$. Assume the presence of the following environment ρ , mapping variables to values:

$$\rho = [x \mapsto o_1, y \mapsto o_2, f \mapsto o_3, g \mapsto o_4]$$

We compute $\hat{e}_0(b, \rho)$ as:

$$\begin{aligned} \hat{e}_0(b, \rho) &= (c = \rho(x) \wedge i = \rho(f)) \\ &= (c = o_1 \wedge i = o_3) \end{aligned}$$

Since the transition statement $\mathbf{tr} \langle \text{makeiter}, [c \mapsto x, i \mapsto f] \rangle$ only contains one transition element, the transfer function simplifies to:

$$\hat{e}[\langle a, b \rangle, \rho](\hat{\sigma}) \triangleq \lambda i. (\hat{\sigma}(j) \wedge \hat{e}_0(b, \rho)) \vee (\hat{\sigma}(i) \wedge \neg \hat{e}_0(b, \rho))$$

where $j : \delta(j, a, i)$. Using this transfer function, we compute the tracematch state $\hat{\sigma}$ by computing the predicate for each state of the automaton, q_1 to q_5 ². This state labelling is

² q_0 is always true

obtained from the automaton for the tracematch as presented in Figure 1.1.

$$\begin{aligned}
q_1 &\mapsto (\overset{\circ}{\sigma}(q_0) \wedge c = o_1 \wedge i = o_3) \vee (\overset{\circ}{\sigma}(q_1) \wedge \neg(c = o_1 \wedge i = o_3)) \\
&\mapsto (\text{true} \wedge c = o_1 \wedge i = o_3) \vee \text{false} \\
&\mapsto c = o_1 \wedge i = o_3 \\
q_2 &\mapsto \overset{\circ}{\sigma}(q_2) \wedge \neg(c = o_1 \wedge i = o_3) = \text{false} \\
q_3 &\mapsto \overset{\circ}{\sigma}(q_3) \wedge \neg(c = o_1 \wedge i = o_3) = \text{false} \\
q_4 &\mapsto \overset{\circ}{\sigma}(q_4) \wedge \neg(c = o_1 \wedge i = o_3) = \text{false} \\
q_5 &\mapsto \overset{\circ}{\sigma}(q_5) \wedge \neg(c = o_1 \wedge i = o_3) = \text{false}
\end{aligned}$$

The above tracematch state indicates that the configuration $c = o_1$ and $i = o_2$ is in state q_1 . Since o_1 is pointed to by variable \mathbf{x} and o_3 by variable \mathbf{f} , this indicates that the automaton version for variables \mathbf{x} and \mathbf{f} is in state q_1 . This is equivalent to our discussion of Figure 2.1 where the event `makeiter(x, f)` was applied only to those automaton versions where the object bindings of the version were consistent with the event. Figure 2.3 shows the tracematch state as computed by the operational semantics for the complete trace from figure 2.1. The first match occurs after statement 12. The predicate for state q_4 is satisfied for the collection o_1 (pointed to by variable \mathbf{x}) and the iterator o_3 (pointed to by variable \mathbf{f}). This represents the violation that the collection was modified while it was being iterated. The following **body** statement executes the tracematch body and sets state q_4 to false. The next violation occurs after statement 14 is executed since the predicate for q_5 is satisfied. This represents the occurrence of two consecutive **next** events without an intervening **hasNext** event on the same iterator. The **body** statement 15 executes the tracematch body and clears q_5 .

Allan et al. [4] proved that this operational semantics is equivalent to the declarative semantics defined in terms of operations on a multitude of automata, one for each possible set of objects bound to tracematch parameters. This makes a dynamic implementation of tracematches practical, because the implementation only has to manipulate one automaton with boolean formulas on its states, rather than an unbounded collection of automata. However, boolean formulas are a difficult domain to abstract.

2.3 A Lattice-Based Operational Semantics

The operational semantics just presented is suitable for a dynamic implementation of trace-matches, but boolean formulas are a difficult concrete domain to abstract. We therefore

	Instructions executed	Change in tracematch state
1	tr $\langle \text{makeiter}, [c \mapsto x, i \mapsto f] \rangle$	$\dot{\sigma}[q_1 \mapsto (c = o_1 \wedge i = o_3)]$
2	tr $\langle \text{hasNext}, [i \mapsto f] \rangle$	$\dot{\sigma}[q_3 \mapsto (c = o_1 \wedge i = o_3)]$
3	body	No change to $\dot{\sigma}$
4	tr $\langle \text{makeiter}, [c \mapsto y, i \mapsto g] \rangle$	$\dot{\sigma}[q_1 \mapsto (c = o_2 \wedge i = o_4), q_3 \mapsto (c = o_1 \wedge i = o_3)]$
5	tr $\langle \text{next}, [i \mapsto f] \rangle$	$\dot{\sigma}[q_1 \mapsto (c = o_2 \wedge i = o_4) \vee (c = o_1 \wedge i = o_3)]$
6	body	No change to $\dot{\sigma}$
7	tr $\langle \text{hasNext}, [i \mapsto g] \rangle$	$\dot{\sigma}[q_1 \mapsto (c = o_1 \wedge i = o_3), q_3 \mapsto (c = o_2 \wedge i = o_4)]$
8	body	No change to $\dot{\sigma}$
9	tr $\langle \text{update}, [c \mapsto x] \rangle$	$\dot{\sigma}[q_2 \mapsto (c = o_1 \wedge i = o_3), q_3 \mapsto (c = o_2 \wedge i = o_4)]$
10	tr $\langle \text{next}, [i \mapsto g] \rangle$	$\dot{\sigma}[q_1 \mapsto (c = o_2 \wedge i = o_4), q_2 \mapsto (c = o_1 \wedge i = o_3)]$
11	body	No change to $\dot{\sigma}$
12	tr $\langle \text{next}, [i \mapsto f] \rangle$	$\dot{\sigma}[q_1 \mapsto (c = o_2 \wedge i = o_4), q_4 \mapsto (c = o_1 \wedge i = o_3)]$
13	body	$\dot{\sigma}[q_1 \mapsto (c = o_2 \wedge i = o_4)]$
14	tr $\langle \text{next}, [i \mapsto g] \rangle$	$\dot{\sigma}[q_5 \mapsto (c = o_2 \wedge i = o_4)]$
15	body	$\dot{\sigma}[q_0 \mapsto \text{true}]$

Figure 2.3: Example: Operational semantics of tracematches. Column 1 shows the same instructions as in Figure 2.1 but using the IR. Column 2 shows the change in tracematch state after each statement. Since state q_0 is always true it is not shown. Similarly, any state which has false as its formula is also not shown.

define a different but equivalent operational semantics based on sets and lattices that is well suited for static analysis.

The core construction of our semantics is a *binding lattice*. Figure 2.4 illustrates a sample binding lattice for a program with three objects o_1, o_2, o_3 ; in general, the binding lattice is defined analogously for the unbounded number of objects that the program may allocate. Thus, the binding lattice is infinite. In Section 3, we will define a finite abstraction of the binding lattice for use in the static analysis. The binding lattice comprises the element \perp , positive bindings (which are a single object), and negative bindings (which contain zero or more objects). The interpretation of each element of the binding lattice is a set of objects: \perp represents the empty set, a positive binding represents a single object, and a negative binding represents the set of all objects other than those in the binding. We write $\overline{\top}$ as a synonym for the empty set of negative bindings (which represents all objects). The lattice order corresponds to the subset order on sets of objects: for any pair of bindings $d_1 \sqsubseteq d_2$, every object in the set represented by d_1 is also in the set represented by d_2 . As a reminder that a set of objects indicates negative bindings, we will always write such a set with a bar above it: \overline{O} . The bar is only a reminder; it has no semantic meaning.

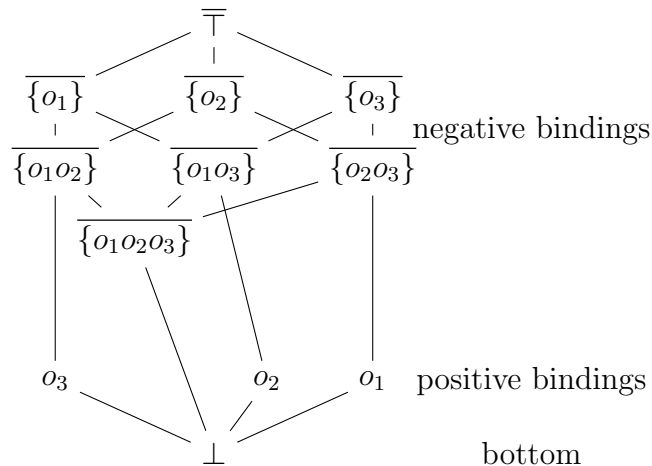


Figure 2.4: Concrete Binding Lattice **Bind**

Formally, the binding lattice $\langle \mathbf{Bind}, \sqsubseteq \rangle$ is defined as follows. Its elements are $\mathbf{Bind} \triangleq \mathbf{Obj} \uplus \overline{\mathcal{P}(\mathbf{Obj})} \uplus \{\perp\}$. The partial order \sqsubseteq is defined as the reflexive transitive closure of the following rules: $\perp \sqsubseteq d$ for any d ; $\overline{O_1} \sqsubseteq \overline{O_2}$ if $O_1 \supseteq O_2$; and $o_1 \sqsubseteq \overline{O_2}$ if $o_1 \notin O_2$.

The following proposition assures us that the binding lattice is indeed a lattice and provides a meet.

Proposition 1. $\langle \mathbf{Bind}, \sqsubseteq \rangle$ is a complete lattice with meet operator defined as:

$$\prod D \triangleq \begin{cases} \perp & \text{if } D \text{ contains } \perp \text{ or } D \text{ contains two positive bindings } o_1, o_2 \text{ that} \\ & \text{are distinct } (o_1 \neq o_2) \text{ or } D \text{ contains a positive binding } o_1 \text{ and a} \\ & \text{negative binding } \overline{O_2} \text{ with } o_1 \in O_2 \\ o & \text{if the above case does not hold and } o \in D \\ \bigcup_{\overline{O} \in D} \overline{O} & \text{otherwise} \end{cases}$$

The proof for the proposition can be found in the Appendix.

For example, using the concrete binding lattice shown in Figure 2.4, $o_1 \sqcap o_2 = \perp$ since $o_1 \neq o_2$. Similarly, $o_3 \sqcap \{\overline{o_1, o_3}\} = \perp$ as $o_3 \in \{o_1, o_3\}$. Also, $o_3 \sqcap \{\overline{o_1, o_2}\}$ is o_3 as the first case in the definition above is not applicable.

We extend the binding lattice pointwise to the space of functions that map each tracematch parameter to an element of the binding lattice. We say that a mapping $m \in F \rightarrow \mathbf{Bind}$ is *consistent* with a given automaton version if the object it associates with each parameter f is in the set represented by $m(f)$. Thus, each mapping m can be interpreted as a set of automaton versions. For example, consider the mapping $c \mapsto \mathbf{x}, i \mapsto \{\overline{\mathbf{g}}\}$. Of the automaton versions shown in Figure 2.1, only the one corresponding to $\mathbf{c}=\mathbf{x}$ and $\mathbf{i}=\mathbf{f}$ is consistent with this mapping. Again, the lattice order on $F \rightarrow \mathbf{Bind}$ corresponds to the subset order on automaton versions.

The runtime state of a tracematch is then defined as a set σ of pairs $\langle q, m \rangle$, where q is a tracematch state, and $m \in F \rightarrow \mathbf{Bind}$. Each pair $\langle q, m \rangle$ indicates that all automaton versions consistent with m are in the state q .

When execution begins, the initial tracematch state is the single pair $\langle q_0, \lambda f. \top \rangle$. The binding map $\lambda f. \top$ is consistent with every version of the automaton, and q_0 indicates that all these versions are in the initial state.

Whenever a transition statement executes, some automaton versions change state and others keep their old state. A mapping m in the runtime state must be refined to distinguish the versions whose state changes from those whose state remains the same. In both cases, this refinement is done using the meet operator of the lattice.

For example, consider a tracematch with a single parameter c and the automaton in Figure 2.5, and suppose that the transition $\langle a, c \mapsto o_1 \rangle$ occurs. The automaton version for o_1 should move to state qa and all others should remain in state q . From the initial map $\lambda f. \top$, we perform meets with $c \mapsto o_1$ and $c \mapsto \{\overline{o_1}\}$ to obtain the desired pairs $\langle qa, [c \mapsto o_1] \rangle$ and $\langle q, [c \mapsto \{\overline{o_1}\}] \rangle$. Suppose the transition $\langle b, c \mapsto o_2 \rangle$ occurs next. We

again perform the meets of the existing states with both $c \mapsto o_2$ and $c \mapsto \overline{\{o_2\}}$ to obtain $\langle qab, [c \mapsto \perp] \rangle, \langle qa, [c \mapsto o_1] \rangle, \langle qb, [c \mapsto o_2] \rangle, \langle q, [c \mapsto \overline{\{o_1 o_2\}}] \rangle$. Since the binding in the first pair is \perp , it is not consistent with any automaton version and can be discarded. The next two pairs correspond to the two automaton versions for o_1 and o_2 in states qa and qb , respectively, and the final pair corresponds to all other automaton versions still in the initial state.

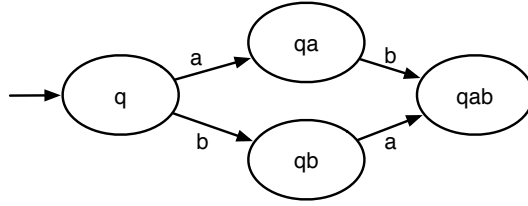


Figure 2.5: Example automaton

In the general case of a tracematch with multiple parameters, there is an additional difference between negative and positive bindings. In the declarative semantics, only the automaton versions consistent in *all* the parameters bound by the transition statement change state; if an automaton version is inconsistent in *any* parameter, its state remains the same. Thus, for the automaton versions that change state, the new map is computed by replacing each $m(f)$ with the meet $m(f) \sqcap o$, where o is the object bound to f by the transition statement. However, for the automaton versions that do not change state, multiple maps must be computed, one for each parameter bound by the transition statement. The map computed for each parameter f reflects the condition that the object bound to f in the automaton version differs from the object bound to f by the transition statement. Thus, the new map for parameter f is constructed by replacing only $m(f)$ with $m(f) \sqcap \overline{\{o\}}$, where o is the object bound to f by the transition statement.

Figure 2.6 shows the complete transfer function e that is applied to each pair $\langle q, m \rangle$ in the tracematch state at every transition statement $\mathbf{tr} \langle a, b \rangle$ that contains a single transition element $\langle a, b \rangle$.

When a transition statement contains multiple transition elements $\langle a, b \rangle$, we apply all the associated positive updates to the original state independently. We only remain in the current state if none of the transitions are taken; therefore, all of the negative updates are applied in sequence:
$$e[\{\langle a_1, b_1 \rangle \cdots \langle a_n, b_n \rangle\}, \rho](q, m) \triangleq (\bigcup_{1 \leq i \leq n} e^+[a_i, b_i, \rho](q, m)) \cup e^-[b_1, \rho](\cdots (e^-[b_n, \rho](q, m)) \cdots)$$

$$\begin{aligned}
e_0^+(b, \rho) &\triangleq \lambda f. \begin{cases} \rho(b(f)) & \text{if } f \in \text{dom}(b) \\ \top & \text{otherwise} \end{cases} \\
e_0^-(b, \rho, f) &\triangleq \lambda f'. \begin{cases} \overline{\{\rho(b(f))\}} & \text{if } f = f' \\ \top & \text{otherwise} \end{cases} \\
e^+[a, b, \rho](q, m) &\triangleq \{\langle q', m \sqcap e_0^+(b, \rho) \rangle : \delta(q, a, q')\} \\
e^-[b, \rho](q, m) &\triangleq \{\langle q, m \sqcap e_0^-(b, \rho, f) \rangle : f \in \text{dom}(b)\} \\
e[a, b, \rho](q, m) &\triangleq e^+[a, b, \rho](q, m) \cup e^-[b, \rho](q, m)
\end{aligned}$$

Figure 2.6: Transfer function, in the Lattice-based operational semantics, for $\mathbf{tr} \langle a, b \rangle$ in local variable environment ρ , which is applied to each pair $\langle q, m \rangle$ in the tracematch state.

The tracematch transition statement performs the above operation on each pair in the set describing the current tracematch state, as well as on the pair $\langle q_0, \lambda f. \top \rangle$ that describes the initial state:

$$\langle \mathbf{tr}(T), \rho, \sigma \rangle \rightarrow \bigcup_{\langle q, m \rangle \in \sigma \cup \{\langle q_0, \lambda f. \top \rangle\}} e[T, \rho](q, m)$$

The **body** statement executes the tracematch body when σ contains a pair $\langle q, m \rangle$ such that q is a final state and $m(f)$ is not \perp for any f . When this happens, all such pairs are removed from the tracematch state:

$$\langle \mathbf{body}, \rho, \sigma \rangle \rightarrow \{\langle q, m \rangle \in \sigma : q \notin Q_f\}$$

In Figure 2.7 we compute the tracematch state σ for the same program trace used in Figures 2.1 and 2.3. As before, we assume the environment ρ to contain the same mapping from variables to values:

$$\rho = [x \mapsto o_1, y \mapsto o_2, f \mapsto o_3, g \mapsto o_4]$$

For transition statements, the first $\langle q, m \rangle$ (shown in bold in Figure 2.7) represents the pair in which the automaton versions consistent with the parameters bound by the transition statements changed state i.e. $e^+[a, b, \rho](q, m)$. In Figure 2.7, the state numbering q_0 to q_5 is obtained from the labelling of states from the automaton defined in Figure 1.1. As seen in the boolean-formula-based operational semantics, the first match occurs at

Instructions executed	Change in tracematch state
1. tr $\langle \text{makeiter}, [c \mapsto x, i \mapsto f] \rangle$	$\langle \mathbf{q}_1, [c \mapsto \mathbf{o}_1, i \mapsto \mathbf{o}_3] \rangle, \langle q_0, [c \mapsto \overline{\{o_1\}}, i \mapsto \top] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{o_3\}}] \rangle$
2. tr $\langle \text{hasNext}, [i \mapsto f] \rangle$	$\langle \mathbf{q}_3, [c \mapsto \mathbf{o}_1, i \mapsto \mathbf{o}_3] \rangle, \langle q_0, [c \mapsto \overline{\{o_1\}}, i \mapsto \overline{\{o_3\}}] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{o_3\}}] \rangle$
3. body	No change to σ
4. tr $\langle \text{makeiter}, [c \mapsto y, i \mapsto g] \rangle$	$\langle \mathbf{q}_1, [c \mapsto \mathbf{o}_2, i \mapsto \mathbf{o}_4] \rangle, \langle q_3, [c \mapsto o_1, i \mapsto o_3] \rangle, \langle q_0, [c \mapsto \overline{\{o_1, o_2\}}, i \mapsto \overline{\{o_3\}}] \rangle,$ $\langle q_0, [c \mapsto \overline{\{o_1\}}, i \mapsto \overline{\{o_3, o_4\}}] \rangle, \langle q_0, [c \mapsto \overline{\{o_2\}}, i \mapsto \overline{\{o_3\}}] \rangle,$ $\langle q_0, [c \mapsto \top, i \mapsto \overline{\{o_3, o_4\}}] \rangle$
5. tr $\langle \text{next}, [i \mapsto f] \rangle$	$\langle \mathbf{q}_1, [c \mapsto \mathbf{o}_1, i \mapsto \mathbf{o}_3] \rangle, \langle q_1, [c \mapsto o_2, i \mapsto o_4] \rangle, \langle q_0, [c \mapsto \overline{\{o_1, o_2\}}, i \mapsto \overline{\{o_3\}}] \rangle,$ $\langle q_0, [c \mapsto \overline{\{o_1\}}, i \mapsto \overline{\{o_3, o_4\}}] \rangle, \langle q_0, [c \mapsto \overline{\{o_2\}}, i \mapsto \overline{\{o_3\}}] \rangle,$ $\langle q_0, [c \mapsto \top, i \mapsto \overline{\{o_3, o_4\}}] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{o_3\}}] \rangle$
6. body	No change to σ
7. tr $\langle \text{hasNext}, [i \mapsto g] \rangle$	$\langle \mathbf{q}_3, [c \mapsto \mathbf{o}_2, i \mapsto \mathbf{o}_4] \rangle, \langle q_1, [c \mapsto o_1, i \mapsto o_3] \rangle,$ $\langle q_0, [c \mapsto \overline{\{o_1, o_2\}}, i \mapsto \overline{\{o_3, o_4\}}] \rangle, \langle q_0, [c \mapsto \overline{\{o_1\}}, i \mapsto \overline{\{o_3, o_4\}}] \rangle,$ $\langle q_0, [c \mapsto \overline{\{o_2\}}, i \mapsto \overline{\{o_3, o_4\}}] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{o_3, o_4\}}] \rangle$
8. body	No change to σ
9. tr $\langle \text{update}, [c \mapsto x] \rangle$	$\langle \mathbf{q}_2, [c \mapsto \mathbf{o}_1, i \mapsto \mathbf{o}_3] \rangle, \langle q_3, [c \mapsto o_2, i \mapsto o_4] \rangle, \langle q_0, [c \mapsto \overline{\{o_1\}}, i \mapsto \top] \rangle,$ $\langle q_0, [c \mapsto \overline{\{o_1\}}, i \mapsto \overline{\{o_3, o_4\}}] \rangle, \langle q_0, [c \mapsto \overline{\{o_1, o_2\}}, i \mapsto \overline{\{o_3, o_4\}}] \rangle$
10. tr $\langle \text{next}, [i \mapsto g] \rangle$	$\langle \mathbf{q}_1, [c \mapsto \mathbf{o}_2, i \mapsto \mathbf{o}_4] \rangle, \langle q_2, [c \mapsto o_1, i \mapsto o_3] \rangle,$ $\langle q_0, [c \mapsto \overline{\{o_1\}}, i \mapsto \overline{\{o_3, o_4\}}] \rangle, \langle q_0, [c \mapsto \overline{\{o_1\}}, i \mapsto \overline{\{o_4\}}] \rangle,$ $\langle q_0, [c \mapsto \top, i \mapsto \overline{\{o_4\}}] \rangle, \langle q_0, [c \mapsto \overline{\{o_1, o_2\}}, i \mapsto \overline{\{o_3, o_4\}}] \rangle$
11. body	No change to σ
12. tr $\langle \text{next}, [i \mapsto f] \rangle$	$\langle \mathbf{q}_4, [c \mapsto \mathbf{o}_1, i \mapsto \mathbf{o}_3] \rangle, \langle q_1, [c \mapsto o_2, i \mapsto o_4] \rangle,$ $\langle q_0, [c \mapsto \overline{\{o_1\}}, i \mapsto \overline{\{o_3, o_4\}}] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{o_3, o_4\}}] \rangle,$ $\langle q_0, [c \mapsto \top, i \mapsto \overline{\{o_3\}}] \rangle, \langle q_0, [c \mapsto \overline{\{o_1, o_2\}}, i \mapsto \overline{\{o_3, o_4\}}] \rangle$
13. body	$\langle q_1, [c \mapsto o_2, i \mapsto o_4] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{o_3, o_4\}}] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{o_3\}}] \rangle,$ $\langle q_0, [c \mapsto \overline{\{o_1, o_2\}}, i \mapsto \overline{\{o_3, o_4\}}] \rangle, \langle q_0, [c \mapsto \overline{\{o_1\}}, i \mapsto \overline{\{o_3, o_4\}}] \rangle$
14. tr $\langle \text{next}, [i \mapsto g] \rangle$	$\langle \mathbf{q}_5, [c \mapsto \mathbf{o}_2, i \mapsto \mathbf{o}_4] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{o_3, o_4\}}] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{o_4\}}] \rangle,$ $\langle q_0, [c \mapsto \overline{\{o_1, o_2\}}, i \mapsto \overline{\{o_3, o_4\}}] \rangle, \langle q_0, [c \mapsto \overline{\{o_1\}}, i \mapsto \overline{\{o_3, o_4\}}] \rangle$
15. body	$\langle q_0, [c \mapsto \overline{\{o_1\}}, i \mapsto \overline{\{o_3, o_4\}}] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{o_4\}}] \rangle,$ $\langle q_0, [c \mapsto \overline{\{o_1, o_2\}}, i \mapsto \overline{\{o_3, o_4\}}] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{o_3, o_4\}}] \rangle,$

Figure 2.7: Computing the tracematch state using the lattice-based operational semantics for the same sequence of instructions as in Figure 2.1 and 2.3.

statement 12 due to the occurrence of the pair $\langle q_4, [c \mapsto o_1, i \mapsto o_3] \rangle$, which represents the situation where the collection o_1 was modified while iterating over it using the iterator o_3 . As per the operational semantics of the **body** statement, this pair is removed from σ . The tracematch state after statement 14 contains the pair $\langle q_5, [c \mapsto o_2, i \mapsto o_4] \rangle$ which represents the occurrence of two consecutive **next** events without an intervening **hasNext** on the same iterator. This pair is removed from σ in the following **body** statement. The lattice-based semantics is equivalent to the declarative semantics. The following function s_σ makes this precise by defining a translation from a state σ in the lattice-based semantics to an equivalent state $\dot{\sigma}$ in the boolean-formula-based semantics.

$$\begin{aligned}
s_d(\langle f, d \rangle) &\triangleq \begin{cases} \text{false} & \text{if } d = \perp \\ (f = o) & \text{if } d \text{ is a positive binding } o \\ \bigwedge_{o \in \bar{O}} \neg(f = o) & \text{if } d \text{ is a negative binding } \bar{O} \end{cases} \\
s_m(m) &\triangleq \bigwedge_{f \in F} s_d(\langle f, m(f) \rangle) \\
s_\sigma(\sigma) &\triangleq \lambda q. \bigvee_{\langle q, m \rangle \in \sigma} s_m(m)
\end{aligned}$$

We have proven that the lattice-based semantics is bisimilar to the original tracematch semantics:

Theorem 1. *The transition relations $\overset{\circ}{\rightarrow}$ and \rightarrow are bisimilar with bisimulation relation $\dot{\sigma} R \sigma \triangleq s_\sigma(\sigma)(q) \iff \dot{\sigma}(q)$. That is,*

- *for every σ there exists $\dot{\sigma}$ with $s_\sigma(\sigma)(q) \iff \dot{\sigma}(q)$ such that $\langle \mathbf{tr}(T), \sigma \rangle \rightarrow \langle \sigma' \rangle \implies \langle \mathbf{tr}(T), \dot{\sigma} \rangle \overset{\circ}{\rightarrow} \langle \sigma' \rangle \wedge \dot{\sigma}'(q) \iff s_\sigma(\sigma')(q)$, and conversely,*
- *for every $\dot{\sigma}$ there exists σ with $s_\sigma(\sigma)(q) \iff \dot{\sigma}(q)$ such that $\langle \mathbf{tr}(T), \dot{\sigma} \rangle \rightarrow \langle \sigma' \rangle \implies \langle \mathbf{tr}(T), \sigma \rangle \rightarrow \langle \sigma' \rangle \wedge \dot{\sigma}'(q) \iff s_\sigma(\sigma')(q)$.*

We direct the interested reader to the Appendix A for the proof.

2.4 Summary

In this chapter, we have provided background information on the tracematch construct that we use to specify temporal specifications to be verified. We have presented the original

declarative semantics and the original operational semantics based on boolean formulas. Since boolean formulas are a difficult domain to abstract, we have also presented a new operational semantics based on sets and lattices. We also proved that the newly proposed lattice-based operational semantics are bisimilar to the original operational semantics. In the next chapter, we create an abstraction for tracematch states and prove it to be correct with respect to the concrete lattice-based operational semantics formalized in this chapter.

Chapter 3

Static Abstraction

In this chapter, we present the abstractions we have designed to statically analyze temporal specifications involving multiple objects as specified by tracematches. We first discuss the intermediate representation used by the static analysis in Section 3.1. The abstraction is presented in two parts. The first abstraction computes object aliasing relationships and is presented in Section 3.2. This information is needed to determine which objects are pointed to by the variables in each transition statement. The second abstraction is presented in Section 3.3 and models the tracematch state. Using this abstraction, the analysis can prove that at certain body statements, the tracematch cannot be in an accepting state. We also discuss some related work in Section 3.4.

3.1 Intermediate Representation

Before performing the static analysis, we simplify the code to an intermediate representation (IR) containing only instructions relevant to tracematch semantics. The intraprocedural instructions in the IR are:

$$s ::= \mathbf{tr} \langle a, b \rangle \mid \mathbf{body} \mid v_1 \leftarrow v_2 \mid v \leftarrow \mathbf{h} \mid \mathbf{h} \leftarrow v \mid v \leftarrow \mathbf{null} \mid v \leftarrow \mathbf{new}$$

In addition, the IR contains method call and return instructions. In the IR, v can be any variable from the set \mathbf{Var} of local variables of the current method. For the properties of tracematches, we do not intend to distinguish individual heap locations. Therefore, we use the symbol \mathbf{h} to represent any heap location, such as a field of an object or an array element. The copy instruction $v_1 \leftarrow v_2$ copies object references between variables.

Therefore, after the copy, variable v_1 points to the same object that v_2 points to. The load instruction, $v \leftarrow \mathbf{h}$, copies the reference for an object from the heap into variable v , so that after the instruction executes, v points to the loaded object. The instruction $\mathbf{h} \leftarrow v$ escapes the object pointed to by v by storing it in a location within the heap. After the **null** assignment instruction $v \leftarrow \mathbf{NULL}$, variable v does not point to any object. The instruction $v \leftarrow \mathbf{new}$, creates a new object and assigns a reference to that object to v .

In Section 2.3, we already gave transfer functions for the **tr** $\langle a, b \rangle$ and **body** instructions in the lattice-based operational semantics. To extend the operational semantics to statements other than these, we add to it a set h abstracting all objects referenced from the heap. The instructions **tr** $\langle a, b \rangle$ and **body** do not change the environment ρ or the heap h . The operational semantics of the remaining instructions are shown in Figure 3.1. For the copy instruction $v_1 \leftarrow v_2$, the environment ρ is updated so that v_1 is mapped to the same value as v_2 , i.e., $\rho(v_2)$. The effect of the load instruction $v \leftarrow \mathbf{h}$ is non-deterministic, because we do not know which specific object from h is loaded. The transfer function updates v to reference any of the objects that could be referenced from the heap. The instruction $\mathbf{h} \leftarrow v$, does not change the environment ρ , but since the object referenced by v has escaped, we add it ($\rho(v)$) to the set h that abstracts all objects referenced from the heap. The instruction $v \leftarrow \mathbf{new}$ creates a new value o . Therefore, the environment ρ is updated with v mapped to the new value, o . For the instruction $v \leftarrow \mathbf{null}$, the environment ρ is updated with v mapped to the special null value, \perp .

$$\begin{aligned}
\langle v_1 \leftarrow v_2, \rho, h, \sigma \rangle &\rightarrow \langle \rho[v_1 \mapsto \rho(v_2)], h, \sigma \rangle \\
\langle v \leftarrow \mathbf{h}, \rho, h, \sigma \rangle &\rightarrow \langle \rho[v \mapsto o], h, \sigma \rangle \text{ for every } o \in h \\
\langle \mathbf{h} \leftarrow v, \rho, h, \sigma \rangle &\rightarrow \langle \rho, h \cup \{\rho(v)\}, \sigma \rangle \\
\langle v \leftarrow \mathbf{new}, \rho, h, \sigma \rangle &\rightarrow \langle \rho[v \mapsto o], h, \sigma \rangle \text{ with } o \text{ fresh} \\
\langle v \leftarrow \mathbf{null}, \rho, h, \sigma \rangle &\rightarrow \langle \rho[v \mapsto \perp], h, \sigma \rangle
\end{aligned}$$

Figure 3.1: Transfer functions for $s \neq \{\mathbf{tr} \langle a, b \rangle, \mathbf{body}\}$ in the lattice-based operational semantics

3.2 Object Abstraction

In a world where an object could only be assigned to a single, never changing, variable, abstracting objects would be trivial; the variable pointing to an object would represent

its abstraction as it would uniquely identify the object at all program points. However, real world programs contain pointers, and pointers cause aliasing. Therefore, statically abstracting the objects in a program is really an exercise in inferring which pointers point to what objects.

Inferring properties of pointers created and manipulated by programs has been the subject of intense research [38, 59]. A large spectrum of pointer analyses, ranging from efficient points-to analyses to highly precise shape analyses, have been developed. A shape analysis emphasizes individual concrete objects and the relationships between them, whereas a pointer analysis emphasizes the pointers, and often models multiple concrete objects using the same abstract representative (e.g. an allocation site). A useful tradeoff, and an increasingly used abstraction, is alias set analysis. This analysis uses a storeless heap abstraction [42, 24] and has been used effectively in earlier work [61, 36, 17, 29, 30]. The abstraction combines certain aspects of both pointer and shape abstractions. Unlike a shape analysis which emphasizes the precise relationships between objects, and is expensive to model, an alias set analysis, like a pointer abstraction, focuses on local pointers to objects. This makes computing the alias set abstraction faster than shape analyses. However, since the analysis is flow-sensitive and inter-procedural it is still considerably slower than most points-to analyses. In Chapter 6 we propose two ways to further speed-up the alias set analysis.

We use the alias set abstraction to model the objects in the program. The abstraction represents each concrete object by the set of local variables pointing to it. This is the same abstraction as the nodes in Sagiv et al.’s shape analysis [61]. However, our abstraction is simpler in that it tracks only the nodes, not the pointer edges between objects.

The set of variables in the abstraction of each object is exact; it is neither a may-point-to nor a must-point-to approximation. Since it may not be known statically whether a given pointer points to the object, the analysis maintains a set ρ^\sharp of abstract objects. This set is an over-approximation of all possible objects. That is, if it is possible for some concrete object to be pointed to by the set of variables o^\sharp , then the set o^\sharp must be an element of ρ^\sharp . Conversely, the presence of o^\sharp in ρ^\sharp indicates that there may exist zero or more concrete objects which are pointed to by the variables in o^\sharp and no others. For example, consider a concrete environment in which variables x and y point to distinct objects and z may be either null or point to the same object as x . The abstraction of this environment would be the set $\{\{x\}, \{x, z\}, \{y\}\}$. Formally, define $\mathbf{Obj}^\sharp \triangleq \mathcal{P}(\mathbf{Var})$ as the set of all sets of variables. The function $\beta_o[\rho] : \mathbf{Obj} \rightarrow \mathbf{Obj}^\sharp$ gives for each concrete object o its abstract counterpart, the set of variables pointing to it:

$$\beta_o[\rho](o) \triangleq \{v \in \mathbf{Var} : \rho(v) = o\}$$

The set of abstract objects can be thought of as an abstraction of the possible concrete environments ρ . Thus, the abstraction function $\beta_\rho : \mathbf{Env} \times \mathcal{P}(\mathbf{Var}) \rightarrow \mathcal{P}(\mathbf{Obj}^\#)$ for environments is defined as:

$$\beta_\rho(\rho, h) \triangleq \{\beta_o[\rho](o) : o \in \text{range}(\rho) \cup h \setminus \{\perp\}\}$$

where h is the set of all objects referenced from the heap.

The alias set abstraction subsumes both may- and must-alias relationships. Pointer analyses generally use one of two abstractions. The first are points-to pairs (p, o) , indicating that the pointer p may point to one of the concrete objects represented by the abstract object o . The second are may- or must-alias pairs (p_1, p_2) , indicating that the pointers p_1 and p_2 may or must point to the same object. In comparison, the alias set abstraction associates with each program point a set of alias sets, each of the form $\{p_1, \dots, p_n\}$. The presence of the set $\{p_1, \dots, p_n\}$ indicates that there may exist an object pointed to by all of the pointers p_1, \dots, p_n and no others. The presence of an alias set containing both p_1 and p_2 at a given program point implies that p_1 and p_2 may be aliased at that point. On the other hand, if every alias set at a given program point contains either both p_1 and p_2 or neither of them, then p_1 and p_2 must be aliased at that point. For example, consider an abstraction $\rho^\#$ that contains only the alias sets $\{z\}$, $\{x, y\}$ and $\{x, y, z\}$. From this abstraction, we can infer that variables x and y must be aliased since all alias sets in $\rho^\#$ either contain both these variables or contain neither x nor y . Similarly, we can also infer that z may alias variables x and y since it is present in some (but not all) alias sets containing x or y . Notice that we cannot make the stronger inference that z must alias x (or y) since not all alias sets in $\rho^\#$ contain z whenever they contain x (or y). If information about allocation sites is needed, an alias set could be augmented with an allocation site, and thus represent only those objects pointed to by the pointers in the set and allocated at the given allocation site.

Each alias set except the empty set represents at most one concrete object at any given instant at run time. For example, consider the alias set $\{x\}$. At run time, the pointer x can only point to one concrete object o at a time; thus at that instant, the alias set $\{x\}$ represents only o and no other concrete objects. This property to statically pinpoint a runtime object enables very precise transfer functions for individual alias sets, with strong updates. This makes the abstraction suitable for a wide variety of analyses that track individual objects [17, 52, 30]. In particular, for the tracematch analysis, the ability to precisely determine which object a variable points to is essential since only then the state of objects can be transitioned based on variables bound in a transition statement.

The transfer function $\llbracket s \rrbracket_{o^\#}$ shown in Figure 3.2 computes the effect of any statement s in the IR except a heap load ($v \leftarrow \mathbf{h}$) on $o^\#$, the set of variables pointing to a given concrete

object o . After a copy statement, $v_1 \leftarrow v_2$, both variables point to the same object. If this object is abstracted by o^\sharp , i.e., v_2 is in the set o^\sharp , then variable v_1 is added to o^\sharp . However, if v_2 is not in o^\sharp , then o^\sharp does not represent the abstraction of the object pointed to by v_2 . Therefore, v_1 can also not point to the object abstracted by o^\sharp and must not be in this set. If v is assigned the value **null**, $v \leftarrow \mathbf{null}$, then v no longer points to the object represented by the set o^\sharp and is removed from this set. Similarly, if s is $v \leftarrow \mathbf{new}$, then, since v points to a new value, it cannot possibly point to some object that is abstracted by o^\sharp . Hence v is removed from o^\sharp . The statements $\mathbf{h} \leftarrow v$, $\mathbf{tr}(T)$ and \mathbf{body} do not change the variables pointing to an object and therefore do not effect o^\sharp .

$$\begin{aligned}
\llbracket s \rrbracket_{o^\sharp}(o^\sharp) &\triangleq \begin{cases} o^\sharp \cup \{v_1\} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \in o^\sharp \\ o^\sharp \setminus \{v_1\} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \notin o^\sharp \\ o^\sharp \setminus \{v\} & \text{if } s \in \{v \leftarrow \mathbf{null}, v \leftarrow \mathbf{new}\} \\ o^\sharp & \text{if } s \in \{\mathbf{h} \leftarrow v, \mathbf{tr}(T), \mathbf{body}\} \\ \text{undefined} & \text{if } s = v \leftarrow \mathbf{h} \end{cases} \\
\mathit{focus}[h^\sharp](v, o^\sharp) &\triangleq \begin{cases} \{o^\sharp \setminus \{v\}\} & \text{if } o^\sharp \notin h^\sharp \\ \{o^\sharp \setminus \{v\}, o^\sharp \cup \{v\}\} & \text{if } o^\sharp \in h^\sharp \end{cases} \\
\llbracket s \rrbracket_{O^\sharp}[h^\sharp](O^\sharp) &\triangleq \begin{cases} \{\llbracket s \rrbracket_{o^\sharp}(o^\sharp) : o^\sharp \in O^\sharp\} & \text{if } s \neq v \leftarrow \mathbf{h} \\ \bigcup_{o^\sharp \in O^\sharp} \mathit{focus}[h^\sharp](v, o^\sharp) & \text{if } s = v \leftarrow \mathbf{h} \end{cases} \\
\llbracket s \rrbracket_{\rho^\sharp}(\rho^\sharp, h^\sharp) &\triangleq \begin{cases} \llbracket s \rrbracket_{O^\sharp}[h^\sharp](\rho^\sharp) \cup \{\{v\}\} & \text{if } s = v \leftarrow \mathbf{new} \\ \llbracket s \rrbracket_{O^\sharp}[h^\sharp](\rho^\sharp) & \text{otherwise} \end{cases} \\
\llbracket s \rrbracket_{h^\sharp}(\rho^\sharp, h^\sharp) &\triangleq \llbracket s \rrbracket_{O^\sharp}[h^\sharp] \left(\begin{cases} h^\sharp \cup \{o^\sharp \in \rho^\sharp : v \in o^\sharp\} & \text{if } s = \mathbf{h} \leftarrow v \\ h^\sharp & \text{otherwise} \end{cases} \right) \\
\llbracket s \rrbracket_{\rho h^\sharp}(\rho^\sharp, h^\sharp) &\triangleq \langle \llbracket s \rrbracket_{\rho^\sharp}(\rho^\sharp, h^\sharp), \llbracket s \rrbracket_{h^\sharp}(\rho^\sharp, h^\sharp) \rangle
\end{aligned}$$

Figure 3.2: Transfer function for the object abstraction

To precisely handle the uncertainty in heap loads we use the materialization or focus operation [61, 36, 17, 29]. The abstract object o^\sharp is split into two, one representing the single concrete object that was loaded, and the other representing all other objects previously represented by o^\sharp . Focus is important to regain the precision lost when an object is no longer referenced from any local variables, in which case the analysis lumps it together with all other such objects. In order for a tracematch operation to be performed on such an object, the object must first be loaded into a variable. At the load, the focus operation separates the loaded object from the other objects. If multiple tracematch operations are

then performed on the object, the analysis knows that they are performed on the same concrete object as long as the local variable continues to point to it.

In addition to the set $\rho^\#$ of possible abstract objects, the analysis tracks a subset $h^\# \subseteq \rho^\#$ of abstract objects which may have escaped to the heap. Formally, the heap abstraction is defined by:

$$\beta_h(\rho, h) \triangleq \{\beta_o[\rho](o) : o \in h\}$$

The focus operation is performed only on these escaped abstract objects. Since focus splits one abstract object into two, it can theoretically lead to exponential growth in the abstraction. The escape information was necessary and sufficient to control this growth in the benchmarks that we evaluated.

We illustrate the effect of the transfer functions using the example statement sequence shown in Figure 3.3. Statement 1 creates a new concrete object and assigns it to variable x . Correspondingly, the transfer function $\llbracket s \rrbracket_{\rho^\#}$ creates the abstract object $\{x\}$. Statement 2 assigns the value of x to some pointer in the heap. The transfer function $\llbracket s \rrbracket_{h^\#}$ adds the abstract object $\{x\}$ to $h^\#$ since the concrete object represented by this abstract object has been assigned to a heap location. The value of x is then assigned to a local variable w in statement 3. The transfer function $\llbracket s \rrbracket_{o^\#}$ adds the variable w to the abstract object $\{x\}$ since after statement 3 executes, w and x point to the same concrete object. Statement 4 creates a new concrete object and assigns it to y . Like in statement 1, a new abstract

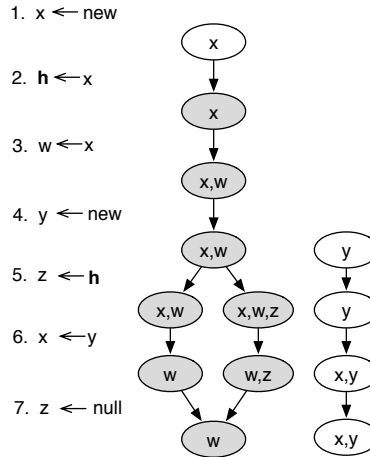


Figure 3.3: Example statement sequence to illustrate transfer functions. Shaded ovals represent abstract objects that are in both $\rho^\#$ and $h^\#$.

object, $\{y\}$, is added to ρ^\sharp . Statement 5 is a load from the heap. The transfer function $\llbracket s \rrbracket_{o^\sharp}$ applies the focus operation to both $\{y\}$ and $\{x, w\}$. Since the abstract object $\{y\}$ is not in h^\sharp , $\text{focus}[h^\sharp](z, \{y\})$ is simply $\{\{y\}\}$. However, since $\{x, w\} \in h^\sharp$, this abstract object is split into two: $\{x, w, z\}$ and $\{x, w\}$. After statement 5, ρ^\sharp contains three abstract objects: $\{x, w\}$, $\{y\}$, and $\{x, w, z\}$. Statement 6 assigns y to x . The transfer function $\llbracket s \rrbracket_{o^\sharp}$ is applied to each of the three abstract objects, yielding $\{w\}$, $\{x, y\}$, and $\{w, z\}$. Statement 7 assigns **null** to z . This changes $\{w, z\}$ to simply $\{w\}$, yielding the abstract environment $\{w\}, \{x, y\}$.

The example illustrates the key characteristic of the object abstraction. The transfer functions flow sensitively track the effect of statements on variables. Each path in the figure represents what happens to a particular concrete object as statements execute; all that changes is the set of variables that point to the same object at different program points. Specifically, if s is any statement in the IR except a heap load, and if o^\sharp is the set of variables pointing to a given concrete object o , then the transfer function $\llbracket s \rrbracket_{o^\sharp}$ from Figure 3.2 computes the exact set of variables which will point to o after the execution of s . This property enables the analysis to flow-sensitively track individual objects along control flow paths; this was one of the three requirements motivated in the introduction. The following proposition formalizes the property:

Proposition 2. *If s is any statement except $v \leftarrow \mathbf{h}$, and $\langle s, \rho, h, \sigma \rangle \rightarrow \langle \rho', h', \sigma' \rangle$, then for any concrete object o that exists prior to the execution of s ,*

$$\llbracket s \rrbracket_{o^\sharp}(\beta_o[\rho](o)) = \beta_o[\rho'](o)$$

We prove this proposition in Appendix A.

Formal definitions of the transfer functions for ρ^\sharp and h^\sharp in terms of $\llbracket s \rrbracket_{o^\sharp}$ and the focus operation are given in Figure 3.2. We combine ρ^\sharp and h^\sharp into a single abstraction and define the combined abstraction function $\beta_{\rho h}(\rho, h) \triangleq \langle \beta_\rho(\rho, h), \beta_h(\rho, h) \rangle$. On the combined abstraction, we define the partial order $\langle \rho_1^\sharp, h_1^\sharp \rangle \sqsubseteq \langle \rho_2^\sharp, h_2^\sharp \rangle$ if $\rho_1^\sharp \subseteq \rho_2^\sharp \wedge h_1^\sharp \subseteq h_2^\sharp$, which induces a join operator $\langle \rho_1^\sharp, h_1^\sharp \rangle \sqcup \langle \rho_2^\sharp, h_2^\sharp \rangle \triangleq \langle \rho_1^\sharp \cup \rho_2^\sharp, h_1^\sharp \cup h_2^\sharp \rangle$. The property that $\rho^\sharp \supseteq h^\sharp$ is always maintained by the transfer functions. On the combined object abstraction, the correctness relation $R_{\rho h}$ is defined as $\langle \rho, h \rangle R_{\rho h} \langle \rho^\sharp, h^\sharp \rangle \triangleq \beta_{\rho h}(\rho, h) \sqsubseteq \langle \rho^\sharp, h^\sharp \rangle$. This ensures that for any concrete object o occurring at run time, its abstract counterpart o^\sharp is included in ρ^\sharp , as well as in h^\sharp if o is referenced from the heap. We have proven that the transfer function for ρ^\sharp and h^\sharp preserves the correctness relation:

Theorem 2. *If $\langle s, \rho, h, \sigma \rangle \rightarrow \langle \rho', h', \sigma' \rangle$ and $\langle \rho, h \rangle R_{\rho h} \langle \rho^\sharp, h^\sharp \rangle$, then $\langle \rho', h' \rangle R_{\rho h} \llbracket s \rrbracket_{\rho h^\sharp}(\rho^\sharp, h^\sharp)$.*

A proof of the theorem is presented in Appendix A.

3.3 Tracematch Abstraction

Typestate associates a state with each runtime object. Existing typestate analyses (e.g. [64, 30]) model each runtime object using an abstraction similar to the one defined in the previous section. The typestate analysis models the state of a runtime object by maintaining a set of possible states for each abstract object. A runtime object o can only be in state q if the abstract object $o^\#$ representing o has q in its set of possible states. When the analysis encounters an instruction that changes the state of an object, it updates the possible states of the appropriate abstract objects.

In our setting, a state is not associated with any single object, but with multiple objects. Thus, we cannot just add the state to any given object abstraction. Therefore, our analysis uses a second abstraction to represent the tracematch state. Each such abstract tracematch state contains within it the abstractions of the objects bound by the tracematch.

We begin by presenting a simple but inefficient abstraction of the tracematch state, then discuss the refined version that we have implemented in our analysis. Thanks to the lattice-based design of our tracematch semantics, a basic tracematch state abstraction would be straightforward to define. Recall that a concrete tracematch state is a set of pairs $\langle q, m \rangle$, where m maps each tracematch parameter to an element of the **Bind** lattice. An abstraction of this state could be defined by replacing all concrete objects in the **Bind** lattice with their abstract counterparts as defined in the previous section. The resulting abstract lattice **Bind**[#] has the same structure as **Bind**, but each positive binding is an *abstract* object, and each negative binding is a set of *abstract* objects. The overall abstraction is a set of pairs $\langle q, m^\# \rangle$, where $m^\#$ maps each tracematch parameter to an element of **Bind**[#]. After working out some details, we defined a transfer function on this domain, proved that it correctly abstracts the semantics, and implemented it. However, on tracematches with multiple parameters, the implementation did not scale to large benchmarks. The key reason for this is that the focus operation was applied to every abstract object bound by a tracematch state. Since each focus splits the state into two, the growth was exponential in the number of abstract objects appearing in the tracematch state.

In fact, there is little benefit to performing the focus operation once the object has been bound in a tracematch state. The benefit of the focus operation is that it singles out one object, so that if a sequence of transition statements occurs, we know that they occur on the same concrete object. Thus, focus is needed for precise aliasing information

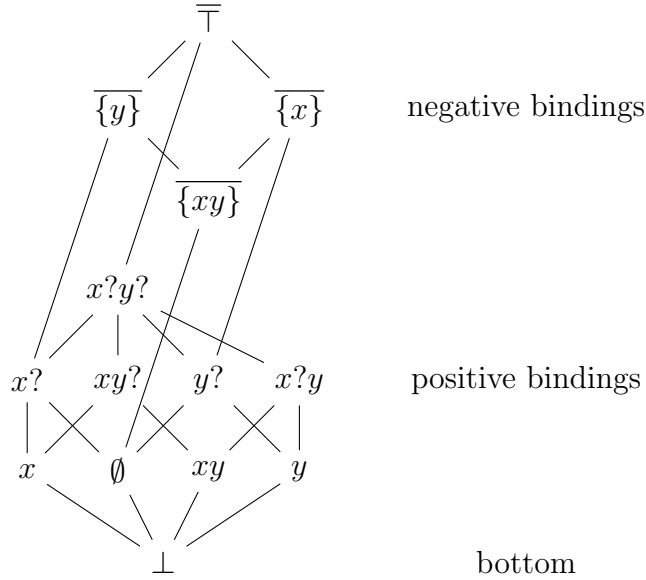


Figure 3.4: Abstract Binding Lattice \mathbf{Bind}^\sharp

at the transition statement before an object is bound. However, after the object is bound, focusing it simply causes both resulting objects to appear in two separate tracematch states, and does not improve precision of the tracematch abstraction.

Therefore, in the tracematch state, we replaced the object abstraction (the precise set of variables pointing to the object) with an under- and over-approximation: a pair of a must set $o^!$ and may set $o^?$ represents every concrete object pointed to by all variables in $o^!$ and only by variables in $o^?$. In the special case when the must and may sets are equal, we recover the precise set of variables pointing to the object. The resulting abstract lattice \mathbf{Bind}^\sharp is illustrated for two variables x, y in Figure 3.4. We use the notation $x^?$ to say that the variable x is in the may set but not the must set, and x to say that it is in both sets. Suppose that a tracematch state has bound an object pointed to by x and a heap load to y occurs. Instead of focusing the bound object to x and xy , we instead use the join of these two, namely $xy^?$, to represent both possibilities. Thus, we avoid focusing objects already bound in the tracematch state.

Efficiency can be further improved for negative bindings. It turns out that the transfer function is independent of the may sets of negatively-bound objects; thus, we need only maintain the must sets. This is because a negative binding indicates that some object o' is not the object o bound by a given automaton version; knowing that a given variable v

may not point to o' gives no information about the identity of o , since v could still point to some other object o'' that is also not o . In addition, although a concrete negative binding is a *set* of objects, all the must sets representing these objects can be replaced with their union without affecting precision of the analysis. Thus, the \mathbf{Bind}^\sharp lattice illustrated in Figure 3.4 represents a negative binding as simply a set of variables that definitely point to every concrete object that may have been negatively bound.

Formally, the object abstraction used in the tracematch state abstraction is given by:

$$\mathbf{Bind}^\sharp \triangleq \{\perp\} \uplus \{\langle o^!, o^? \rangle \in \mathcal{P}(\mathbf{Var})^2 : o^! \subseteq o^?\} \uplus \overline{\mathcal{P}(\mathbf{Var})}$$

As a result, when we do not know whether a variable points to some object, instead of requiring two precise abstract objects, we need only one in which the variable appears in the may set $o^?$ but not the must set $o^!$. Informally, a positive binding $\langle o^!, o^? \rangle$ represents an object o for which $o^! \subseteq \beta_o(o) \subseteq o^?$. A negative binding \overline{V}^\sharp represents a set \overline{O} of negatively bound objects for which $V^\sharp \subseteq \bigcup_{o \in \overline{O}} \beta_o(o)$.

We illustrate the abstraction for positive bindings using the following example. Suppose that a tracematch state has bound an object pointed to by x . In the object abstraction, this object is abstracted by the alias set $\{x\}$. In the tracematch state, we represent the same object as a pair $\langle \{x\}, \{x\} \rangle$. Now, if a heap load to y occurs ($y \leftarrow \mathbf{h}$), the object abstraction would focus $\{x\}$ resulting in the two alias sets, $\{x\}$ and $\{x, y\}$. In the tracematch state we represent the bound object, after the heap load, by the pair $\langle \{x\}, \{x, y\} \rangle$.

Let us now also consider an example for negative bindings. Assume that a tracematch state negatively binds object o_1 and o_2 which are represented in the object abstraction as the alias sets $\{a\}$ and $\{b\}$ respectively. If we had simply chosen the same abstraction as for positive bindings, the negative bindings would have been represented using a set containing both the pairs $\langle \{a\}, \{a\} \rangle$ and $\langle \{b\}, \{b\} \rangle$. This would have been sufficient to make the abstraction efficient when encountering loads from the heap (as discussed earlier in the case of positive bindings). However, for negative bindings, we chose a more efficient abstraction which collects the must sets of the abstractions for objects o_1 and o_2 in a single set of variables. Hence, for our chosen abstraction, the negative binding of o_1 and o_2 is represented by the set $\overline{\{a, b\}}$.

The function β_d is defined as the most precise abstraction of an element of the concrete binding lattice:

$$\beta_d[\rho](d) \triangleq \begin{cases} \perp & \text{if } d = \perp \\ \langle \beta_o[\rho](o), \beta_o[\rho](o) \rangle & \text{if } d \text{ is a positive binding } o \in \mathbf{Obj} \\ \overline{\bigcup_{o \in \overline{O}} \beta_o[\rho](o)} & \text{if } d \text{ is a negative binding } \overline{O} \subseteq \mathbf{Obj} \end{cases}$$

We extend β_d pointwise to maps $F \rightarrow \mathbf{Bind}^\sharp$ and to the overall tracematch state $\mathbf{State}^\sharp \triangleq \mathcal{P}(Q \times (F \rightarrow \mathbf{Bind}^\sharp))$ as follows:

$$\begin{aligned}\beta_m[\rho](m) &\triangleq \lambda f. \beta_d[\rho](m(f)) \\ \beta_\sigma[\rho](\sigma) &\triangleq \{\langle q, \beta_m[\rho](m) \rangle : \langle q, m \rangle \in \sigma\}\end{aligned}$$

A partial order on \mathbf{Bind}^\sharp , coinciding with the partial order on \mathbf{Bind} , is defined as the reflexive transitive closure of the following rules: $\perp \sqsubseteq x$ for any x ; $\overline{V_1^\sharp} \sqsubseteq \overline{V_2^\sharp}$ if $V_1^\sharp \supseteq V_2^\sharp$; $\langle o^!, o^? \rangle \sqsubseteq \overline{V^\sharp}$ if $o^! \cap V^\sharp = \emptyset$; and $\langle o_1^!, o_1^? \rangle \sqsubseteq \langle o_2^!, o_2^? \rangle$ if $o_1^! \supseteq o_2^!$ and $o_1^? \subseteq o_2^?$.

The following propositions ensure that \mathbf{Bind}^\sharp is a finite lattice and that the abstraction function β_d preserves the partial order from \mathbf{Bind} in \mathbf{Bind}^\sharp i.e. it is *monotone*.

Proposition 3. $\langle \mathbf{Bind}^\sharp, \sqsubseteq \rangle$ is a finite lattice with meet operator defined as:

$$\begin{aligned}\perp \sqcap x &= x \sqcap \perp \triangleq \perp \text{ for any } x \\ \langle o_1^!, o_2^? \rangle \sqcap \langle o_2^!, o_2^? \rangle &\triangleq \text{pos}(o_1^! \cup o_2^!, o_1^? \cap o_2^?) \\ \langle o^!, o^? \rangle \sqcap \overline{V^\sharp} &= \overline{V^\sharp} \sqcap \langle o^!, o^? \rangle \triangleq \text{pos}(o^!, o^? \setminus V^\sharp) \\ \overline{V_1^\sharp} \sqcap \overline{V_2^\sharp} &\triangleq \overline{V_1^\sharp \cup V_2^\sharp}\end{aligned}$$

$$\text{where } \text{pos}(o^!, o^?) \triangleq \begin{cases} \langle o^!, o^? \rangle & \text{if } o^! \subseteq o^? \\ \perp & \text{otherwise} \end{cases}$$

Proposition 4. The abstraction function $\beta_d[\rho]$ is monotone. That is, $d_1 \sqsubseteq d_2 \implies \beta_d[\rho](d_1) \sqsubseteq \beta_d[\rho](d_2)$.

Proofs of the propositions can be found in the Appendix.

A correctness relation relating concrete and abstract binding lattice elements is defined in terms of the partial order, and is extended pointwise to maps $F \rightarrow \mathbf{Bind}^\sharp$ and the overall abstract tracematch state \mathbf{State}^\sharp :

$$\begin{aligned}d \ R_d[\rho] \ d^\sharp & \text{ if } \beta_d[\rho](d) \sqsubseteq d^\sharp \\ \langle q, m \rangle \ R_m[\rho] \ \langle q, m^\sharp \rangle & \text{ if } \forall f \in F. m(f) \ R_d[\rho] \ m^\sharp(f) \\ \sigma \ R_\sigma[\rho] \ \sigma^\sharp & \text{ if } \forall \langle q, m \rangle \in \sigma. \exists \langle q, m^\sharp \rangle \in \sigma^\sharp. \langle q, m \rangle \ R_m[\rho] \ \langle q, m^\sharp \rangle\end{aligned}$$

An abstract state σ^\sharp soundly approximates a concrete state σ if for every pair $\langle q, m \rangle$ in σ , there is a corresponding pair $\langle q, m^\sharp \rangle$ in σ^\sharp that soundly approximates it. A pair $\langle q, m^\sharp \rangle$ soundly approximates $\langle q, m \rangle$ if for every tracematch parameter f , $m^\sharp(f)$ is higher in the binding lattice than the abstraction of $m(f)$ obtained by replacing each concrete object with the set of variables that point to it. Recall that a **body** statement completes a match only if the concrete state contains a pair $\langle q, m \rangle$ such that q is a final state and $m(f)$ is not \perp for any f . The correctness relation ensures that if this happens, the abstract state σ^\sharp must also contain a pair $\langle q, m^\sharp \rangle$ satisfying the same conditions. In the absence of such a pair in the abstract state, the analysis concludes that the **body** statement cannot complete a match.

The transfer function for the tracematch state abstraction for all statements except transition statements is shown in Figure 3.5. We again draw a bar over each negative binding like we did for the concrete tracematch lattice. The helper function $\llbracket s \rrbracket_{d^\sharp}$ is similar to $\llbracket s \rrbracket_{o^\sharp}$ from the object abstraction, but it updates both the must and may set of each abstract binding. On a heap load instruction, it introduces uncertainty into the binding instead of focusing it. The transfer function is extended pointwise to maps of bindings by $\llbracket s \rrbracket_{m^\sharp}$ and to sets of abstract state pairs by $\llbracket s \rrbracket_{\sigma^\sharp}$. Like for $\llbracket s \rrbracket_{o^\sharp}$, we have proven that the adapted function $\llbracket s \rrbracket_{d^\sharp}$ also tracks each concrete object flow-sensitively along control flow paths:

Proposition 5. *If $\langle s, \rho \rangle \rightarrow \langle \rho' \rangle$ then $d R_d[\rho] d^\sharp \implies d R_d[\rho'] \llbracket s \rrbracket_{d^\sharp}(d^\sharp)$.*

We prove this proposition in the Appendix.

The transfer function for transition statements is more complicated. Recall from our discussion in Section 2.2 that a transition statement $\mathbf{tr}(T)$ is inserted into the code at each point in the program where a tracematch symbol could match. In the operational semantics, all variables mentioned in each transition statement are looked up in the concrete environment. How should this lookup be performed in the abstract domain? A sound but imprecise and therefore costly approach is to consider that each variable v could point to any abstract object containing v , and to handle all possible combinations of variable values independently. We use a more precise approach that considers *compatibility* [61], the notion that some abstract objects cannot possibly correspond to concrete objects in the same execution. For example, the abstract environment may contain both $\{x\}$ and $\{x, y\}$ if the object pointed to by x is also pointed to by y in some but not all executions. However, at any given instant at run time, y cannot both point and not point to the object pointed to by x ; thus, the two abstract objects are incompatible. The analysis therefore considers *reduced* environments, which are subsets of the abstract environment ρ^\sharp satisfying the following constraints:

$$\begin{aligned}
\llbracket s \rrbracket_{d^\#}(\perp) &\triangleq \perp \text{ for all statements } s \\
\llbracket s \rrbracket_{d^\#}(\langle o^!, o^? \rangle) &\triangleq \begin{cases} \langle o^! \cup \{v_1\}, o^? \cup \{v_1\} \rangle & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \in o^! \\ \langle o^! \setminus \{v_1\}, o^? \cup \{v_1\} \rangle & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \notin o^! \wedge v_2 \in o^? \\ \langle o^! \setminus \{v_1\}, o^? \setminus \{v_1\} \rangle & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \notin o^! \wedge v_2 \notin o^? \\ \langle o^! \setminus \{v\}, o^? \setminus \{v\} \rangle & \text{if } s \in \{v \leftarrow \mathbf{null}, v \leftarrow \mathbf{new}\} \\ \langle o^! \setminus \{v\}, o^? \cup \{v\} \rangle & \text{if } s = v \leftarrow \mathbf{h} \\ \langle o^!, o^? \rangle & \text{if } s \in \{\mathbf{h} \leftarrow v, \mathbf{body}\} \end{cases} \\
\llbracket s \rrbracket_{d^\#}(\overline{V^\#}) &\triangleq \begin{cases} \overline{V^\# \cup \{v_1\}} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \in \overline{V^\#} \\ \overline{V^\# \setminus \{v_1\}} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \notin \overline{V^\#} \\ \overline{V^\# \setminus \{v\}} & \text{if } s \in \{v \leftarrow \mathbf{null}, v \leftarrow \mathbf{new}, v \leftarrow \mathbf{h}\} \\ \overline{V^\#} & \text{if } s \in \{\mathbf{h} \leftarrow v, \mathbf{body}\} \end{cases} \\
\llbracket s \rrbracket_{m^\#}(q, m^\#) &\triangleq \{ \langle q, \lambda f. \llbracket s \rrbracket_{d^\#}(m^\#(f)) \rangle \} \\
\llbracket s \rrbracket_{\sigma^\#}(\sigma^\#) &\triangleq \bigcup_{\langle q, m^\# \rangle \in \sigma^\# \cup \{ \langle q_0, \lambda f. \top \rangle \}} \llbracket s \rrbracket_{m^\#}(q, m^\#)
\end{aligned}$$

Figure 3.5: Transfer functions for the tracematch state abstraction for $s \neq \mathbf{tr}(T)$.

- The objects must all be *compatible* with each other, and with all objects in the tracematch state being updated.
- The objects must be *relevant*: each object must be pointed to by some variable in the transition statement.
- The subset must contain some object pointed to by each variable in the transition statement.

These constraints guarantee that each variable points to a unique abstract object, so every variable can be looked up in the reduced abstract environment. In addition, the constraints reduce the otherwise possibly exponential number of subsets of the abstract environment to a small number, usually only one. To be sound, the analysis considers all reduced environments satisfying the constraints.

Consider, for example, a transition statement binding x and y to two tracematch parameters. Suppose that the abstract environment contains abstract objects $\{x\}, \{y\}, \{x, y\}$

and $\{z\}$. The subsets $\{\{x\}, \{y\}\}$ and $\{\{x, y\}\}$ satisfy the constraints of the reduced environment. The subsets $\{\{x\}, \{x, y\}\}$ and $\{\{y\}, \{x, y\}\}$ are not compatible. The subset $\{\{x\}, \{y\}, \{z\}\}$ is compatible but not relevant since the transition statement does not bind z . The subset $\{\{x\}\}$ is not in the reduced environment because it does not contain any object pointed to by y .

It would be expensive to construct the reduced environment by considering all subsets of the abstract environment and selecting those that satisfy the constraints. Instead, we use the algorithm in Figure 3.6, which, by construction, only generates environments satisfying the constraints. The algorithm works as follows: at each step, it chooses some abstract object o^\sharp to remove from the abstract environment ρ^\sharp , and calls itself recursively to construct all reduced environments not containing o^\sharp and all reduced environments containing o^\sharp . The set of all reduced environments not containing o^\sharp is simply the set of all reduced environments of the smaller abstract environment $\rho^\sharp \setminus \{o^\sharp\}$. A reduced environment can contain o^\sharp only if o^\sharp is relevant and compatible with other abstract objects in the environment. To check that o^\sharp is relevant, the algorithm checks that $o^\sharp \cap \text{relevantVars}$ is non-empty. To ensure that o^\sharp is compatible with other abstract objects in the environment, the algorithm uses a parameter called *forbiddenVars* to keep track of variables which already appear in some abstract object. When it calls itself recursively to construct the reduced environments to which o^\sharp will be added, it adds all the variables in o^\sharp to *forbiddenVars*. Thus, the abstract objects in the environments returned by the recursive call cannot contain any of the variables in o^\sharp , so they are compatible with o^\sharp . To each of the reduced environments returned by the recursive call, the algorithm adds o^\sharp , and the environments are then returned. In the base case, when ρ^\sharp is empty, the algorithm returns either the empty environment if every relevant variable has already been included in some abstract object, or no environments if some relevant variable remains.

Since Sagiv et al.’s notion of compatibility [61] is defined only for the precise object abstraction, we generalized it for the must-may abstraction. The generalized *compatible* predicate and the computation of reduced environments are formally defined in Figure 3.7. In order for two abstract objects to be compatible, they must either be abstractions of distinct concrete objects, or of the same concrete object. In the former case, the two must sets need to be disjoint. In the latter case, the must set of each abstract object needs to be a subset of the may set of the other. Before computing the reduced environments using Algorithm 3.6, we use the generalized compatibility predicate to remove from the abstract environment any abstract objects that are incompatible with an abstract object already bound in the tracematch state.

At a high level, the transfer function for transition statements mirrors the operational semantics of $\text{tr}(T)$ presented in Section 2.3. We first define the transfer function e^\sharp (Fig-

```

reducedEnvs( $\rho^\#$ :  $\mathcal{P}(\mathbf{Var})$ , relevantVars:  $\mathbf{Var}$ , forbiddenVars:  $\mathbf{Var}$ )
1   if  $\rho^\# \neq \emptyset$ 
2     Let  $o^\# \in \rho^\#$ 
3      $r1 = \text{reducedEnvs}(\rho^\# \setminus \{o^\#\}, \text{relevantVars}, \text{forbiddenVars})$ 
4     if ( $o^\# \cap \text{relevantVars} \neq \emptyset$ )  $\wedge$  ( $o^\# \cap \text{forbiddenVars} = \emptyset$ )
5        $r2 = \text{reducedEnvs}(\rho^\# \setminus \{o^\#\}, \text{relevantVars} \setminus o^\#, \text{forbiddenVars} \cup o^\#)$ 
6        $r3 = \{\rho'^\# \cup \{o^\#\} : \rho'^\# \in r2\}$ 
7       return  $r1 \cup r3$ 
8     else
9       return  $r1$ 
10    fi
11  else
12    if  $\text{relevantVars} = \emptyset$  then return  $\{\{\}\}$ 
13    else return  $\{\}$ 
14  fi

```

Figure 3.6: Computing reduced environments

$$\begin{aligned}
\text{same}(\langle o_1^!, o_1^? \rangle, \langle o_2^!, o_2^? \rangle) &\triangleq o_1^! \subseteq o_2^? \wedge o_2^! \subseteq o_1^? \\
\text{diff}(\langle o_1^!, o_1^? \rangle, \langle o_2^!, o_2^? \rangle) &\triangleq o_1^! \cap o_2^! = \emptyset \\
\text{compatible}(o_1^{!?}, o_2^{!?}) &\triangleq \text{same}(o_1^{!?}, o_2^{!?}) \vee \text{diff}(o_1^{!?}, o_2^{!?}) \\
\text{setcompat}(O^{!?}) &\triangleq \forall o_1^{!?}, o_2^{!?} \in O^{!?} \text{ compatible}(o_1^{!?}, o_2^{!?}) \\
\text{relevant}(O^\#, V) &\triangleq V \subseteq \cup_{o^\# \in O^\#} o^\# \wedge \forall o^\# \in O^\# o^\# \cap V \neq \emptyset \\
\text{red-envs}(\rho^\#, O^{!?}, V) &\triangleq \{O^\# \subseteq \rho^\# : \text{relevant}(O^\#, V) \wedge \\
&\quad \text{setcompat}(\{\langle o^\#, o^\# \rangle : o^\# \in O^\#\} \cup O^{!?})\}
\end{aligned}$$

Figure 3.7: Generalized compatibility predicate.

ure 3.8) that is applied for each pair $\langle q, m^\# \rangle$ on a transition element $\langle a, b \rangle$ for a transition statement $\mathbf{tr}(T)$. Having defined abstract variable lookup, the abstract tracematch transition functions $e_0^{+\#}, e_0^{-\#}, e^{+\#}, e^{-\#}, e^\#$ are exactly like their concrete counterparts, but with abstract lookup $lookup(O^\#, v)$ substituted for concrete lookup in ρ .

$$\begin{aligned}
objs(m^\#) &\triangleq \{\langle o^!, o^? \rangle \in \text{range}(m^\#)\} \\
lookup(O^\#, v) &\triangleq o^\# \in O^\# : v \in o^\# \\
e_0^{+\#}(b, O^\#) &\triangleq \lambda f. \begin{cases} \langle o^\#, o^\# \rangle \text{ where } o^\# = \text{lookup}(O^\#, b(f)) & \text{if } f \in \text{dom}(b) \\ \top & \text{otherwise} \end{cases} \\
e^{+\#}[a, b, O^\#](q, m^\#) &\triangleq \left\{ \left\langle q', m^\# \sqcap e_0^{+\#}(b, O^\#) \right\rangle : \delta(q, a, q') \right\} \\
e_0^{-\#}(b, O^\#, f) &\triangleq \lambda f'. \begin{cases} \overline{\text{lookup}(O^\#, b(f))} & \text{if } f = f' \\ \top & \text{otherwise} \end{cases} \\
e^{-\#}[b, O^\#](q, m^\#) &\triangleq \left\{ \left\langle q, m^\# \sqcap e_0^{-\#}(b, O^\#, f) \right\rangle : f \in \text{dom}(b) \right\} \\
e^\#[a, b, O^\#](q, m^\#) &\triangleq e^{+\#}[a, b, O^\#](q, m^\#) \cup e^{-\#}[b, O^\#](q, m^\#)
\end{aligned}$$

Figure 3.8: Transfer function for a transition statement $\mathbf{tr}(T)$, which is applied to each pair $\langle q, m^\# \rangle$ in the tracematch state abstraction.

As we did in the case of the operational semantics for $\mathbf{tr}(T)$, if the transition statement contains multiple pairs $\langle a, b \rangle$ we apply all the associated positive updates to the original state independently. Also as before, all the negative updates are applied in sequence since we only remain in the current state if none of the transitions are taken:

$$e^\#[\{\langle a_1, b_1 \rangle \cdots \langle a_n, b_n \rangle\}, O^\#](q, m^\#) \triangleq \left(\bigcup_{1 \leq i \leq n} e^{+\#}[a_i, b_i, O^\#](q, m^\#) \right) \cup e^{-\#}[b_1, O^\#](\cdots e^{-\#}[b_n, O^\#](q, m^\#) \cdots)$$

Then, the overall transfer function $\llbracket \mathbf{tr}(T) \rrbracket_{m^\#}$ is defined as:

$$\llbracket \mathbf{tr}(T) \rrbracket_{m^\#}[\rho^\#](q, m^\#) \triangleq \bigcup_{O^\# \in \text{red-envs}(\rho^\#, \text{objs}(m^\#), \bigcup_{\langle a, b \rangle \in T} \text{range}(b))} e^\#[T, O^\#](q, m^\#)$$

The transfer function joins the results of $e^\#$ for all reduced abstract environments $O^\# \subseteq \rho^\#$. Finally, $\llbracket s \rrbracket_{\sigma^\#}$ extends $\llbracket s \rrbracket_{m^\#}$ to sets of abstract tracematch state pairs; it is the same as in Figure 3.5. At control flow merge points, the join operator used on sets of tracematch state pairs is set union.

We illustrate the effect of the tracematch state transfer function using our continued example from previous chapters. The safety property dealing with collections and iterators

was specified in Chapter 1 and the actual tracematch is stated in Figure 1.2. Previously, we have used the same sequence of events to illustrate the declarative semantics (Figure 2.1) and the computation of the tracematch state using the operational semantics (boolean-based in Figure 2.3 and lattice-based in Figure 2.7). The example assumed the presence of a runtime environment ρ to contain the following mapping for variables to values:

$$\rho = [x \mapsto o_1, y \mapsto o_2, f \mapsto o_3, g \mapsto o_4]$$

We assume that object o_1 is represented by the abstract object $\{x\}$, o_2 by $\{y\}$, o_3 by $\{f\}$ and object o_4 is abstracted by the set $\{g\}$. Hence the abstract environment ρ^\sharp is:

$$\rho^\sharp = [x \mapsto \{x\}, y \mapsto \{y\}, f \mapsto \{f\}, g \mapsto \{g\}]$$

For conciseness, we will use the precise object abstraction and not the one which uses may and must sets. Also, for the sake of simplicity, objects are not aliased.

The first event $\mathbf{makeiter}(\mathbf{x}, \mathbf{f})$ is represented by the transition statement $\mathbf{tr} \langle \mathbf{makeiter}, [c \mapsto x, i \mapsto f] \rangle$. Since this is the first transition statement, σ^\sharp does not contain any pair $\langle q, m^\sharp \rangle$. The only pair on which we must apply $\llbracket \mathbf{tr}(T) \rrbracket$ is $\langle q_0, \lambda f. \top \rangle$. Since the tracematch has two parameters, c and i , this is equivalent to $\langle q_0, [c \mapsto \top, i \mapsto \top] \rangle$.

$$\begin{aligned} \llbracket \mathbf{tr}(T) \rrbracket_{m^\sharp}[\rho^\sharp](\langle q_0, [c \mapsto \top, i \mapsto \top] \rangle) = \\ \bigcup_{O^\sharp \in \text{red-envs}(\rho^\sharp, \text{objs}(m^\sharp), \bigcup_{(a,b) \in T} \text{range}(b))} e^\sharp[T, O^\sharp](\langle q_0, [c \mapsto \top, i \mapsto \top] \rangle) \end{aligned}$$

As the transition statement has one transition element $\langle \mathbf{makeiter}, [c \mapsto x, i \mapsto f] \rangle$, $\bigcup_{(a,b) \in T} \text{range}(b) = \{x, f\}$. Also as $m^\sharp = [c \mapsto \top, i \mapsto \top]$, $O^\sharp = \text{objs}(m^\sharp) = \{\}$. Therefore:

$$\begin{aligned} \llbracket \mathbf{tr}(T) \rrbracket_{m^\sharp}[\rho^\sharp](\langle q_0, [c \mapsto \top, i \mapsto \top] \rangle) = \\ \bigcup_{O^\sharp \in \text{red-envs}(\rho^\sharp, \{\}, \{x, f\})} e^\sharp[\mathbf{makeiter}, [c \mapsto x, i \mapsto f], O^\sharp](\langle q_0, [c \mapsto \top, i \mapsto \top] \rangle) \end{aligned}$$

Computing $\text{red-envs}(\rho^\sharp, \{\}, \{x, f\})$ gives us that the only reduced environment possible is the environment $\{\{x\}, \{f\}\}$ since $\{y\}$ and $\{g\}$ are not relevant.

$$\begin{aligned} \llbracket \mathbf{tr}(T) \rrbracket_{m^\sharp}[\rho^\sharp](\langle q_0, [c \mapsto \top, i \mapsto \top] \rangle) = \\ e^\sharp[\mathbf{makeiter}, [c \mapsto x, i \mapsto f], \{\{x\}, \{f\}\}](\langle q_0, [c \mapsto \top, i \mapsto \top] \rangle) \end{aligned}$$

We compute $e^{+\sharp}$ and $e^{-\sharp}$:

$$\begin{aligned} e^{+\sharp} &= \{\langle q_1, [c \mapsto \{x\}, i \mapsto \{f\}] \rangle\} \\ e^{-\sharp} &= \left\{ \left\langle q_0, [c \mapsto \overline{\{x\}}, i \mapsto \top] \right\rangle, \left\langle q_0, [c \mapsto \top, i \mapsto \overline{\{f\}}] \right\rangle \right\} \end{aligned}$$

Above, the pair $\langle q, m^\sharp \rangle$ obtained from $e^{+\sharp}$ represents the pair in which the automaton has changed state from q_0 to q_1 with a consistent binding between the parameters bound

by the transition element and the automaton. The two pairs generated by $e^{-\#}$ represent automaton versions which stay in state q_0 since the parameters bound by the transition element are inconsistent with the automaton.

Therefore:

$$\sigma^\# = \{\langle q_1, [c \mapsto \{x\}, i \mapsto \{f\}] \rangle, \langle q_0, [c \mapsto \overline{\{x\}}, i \mapsto \top] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{f\}}] \rangle\}$$

When we computed the tracematch state using the lattice-based operational semantics (Figure 2.7), the state σ after the first transition statement was:

$$\sigma = \langle q_1, [c \mapsto o_1, i \mapsto o_3] \rangle, \langle q_0, [c \mapsto \overline{\{o_1\}}, i \mapsto \top] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{o_3\}}] \rangle$$

Comparing the two, we see that the states are equivalent except that the abstract state contains the abstract representation of each runtime object. Figure 3.9 computes the abstract tracematch state after each instruction in the sample program. We find, as per our decision, the abstract state mimics the concrete state from Figure 2.7 with the only difference being that for a positive binding each object is replaced by its abstraction and the variables in the abstract objects representing negative bindings are all merged into a single set.

We have proven that the transfer function $\llbracket s \rrbracket_{\sigma^\#}$ preserves the correctness relation:

Theorem 3. *If $\langle s, \rho, h, \sigma \rangle \rightarrow \langle \rho', h', \sigma' \rangle$ and $\sigma R_\sigma[\rho] \sigma^\#$, then $\sigma' R_\sigma[\rho'] \llbracket s \rrbracket_{\sigma^\#}[\rho^\#](\sigma^\#)$.*

We prove this theorem in Appendix A.

3.4 Related Work

When tracematches were introduced, space and time overhead of their dynamic implementation was a concern [4]. In general, the overhead varied widely depending on the tracematch and the number of dynamic updates to the tracematch state that must be performed; in many cases, the overhead was prohibitive.

One approach to reduce the overhead has been to improve the dynamic tracematch implementation [7]. In this approach, the tracematch automaton (but not the base code to which it is applied) is analyzed statically to generate more efficient matching code. Specific attention has been paid to freeing bindings as soon as possible to reduce memory

Instructions executed	Change in tracematch state
1. tr $\langle \text{makeiter}, [c \mapsto x, i \mapsto f] \rangle$	$\langle \mathbf{q}_1, [c \mapsto \{\mathbf{x}\}, i \mapsto \{\mathbf{f}\}] \rangle, \langle q_0, [c \mapsto \overline{\{x\}}, i \mapsto \top] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{f\}}] \rangle$
2. tr $\langle \text{hasNext}, [i \mapsto f] \rangle$	$\langle \mathbf{q}_3, [c \mapsto \{\mathbf{x}\}, i \mapsto \{\mathbf{y}\}] \rangle, \langle q_0, [c \mapsto \overline{\{x\}}, i \mapsto \overline{\{f\}}] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{f\}}] \rangle$
3. body	No change to σ
4. tr $\langle \text{makeiter}, [c \mapsto y, i \mapsto g] \rangle$	$\langle \mathbf{q}_1, [c \mapsto \{\mathbf{y}\}, i \mapsto \{\mathbf{g}\}] \rangle, \langle q_3, [c \mapsto \{x\}, i \mapsto \{y\}] \rangle, \langle q_0, [c \mapsto \overline{\{x, y\}}, i \mapsto \overline{\{f\}}] \rangle,$ $\langle q_0, [c \mapsto \overline{\{x\}}, i \mapsto \overline{\{f, g\}}] \rangle, \langle q_0, [c \mapsto \overline{\{y\}}, i \mapsto \overline{\{f\}}] \rangle,$ $\langle q_0, [c \mapsto \top, i \mapsto \overline{\{f, g\}}] \rangle$
5. tr $\langle \text{next}, [i \mapsto f] \rangle$	$\langle \mathbf{q}_1, [c \mapsto \{\mathbf{x}\}, i \mapsto \{\mathbf{y}\}] \rangle, \langle q_1, [c \mapsto \{y\}, i \mapsto \{g\}] \rangle, \langle q_0, [c \mapsto \overline{\{x, y\}}, i \mapsto \overline{\{f\}}] \rangle,$ $\langle q_0, [c \mapsto \overline{\{x\}}, i \mapsto \overline{\{f, g\}}] \rangle, \langle q_0, [c \mapsto \overline{\{y\}}, i \mapsto \overline{\{f\}}] \rangle,$ $\langle q_0, [c \mapsto \top, i \mapsto \overline{\{f, g\}}] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{f\}}] \rangle$
6. body	No change to σ
7. tr $\langle \text{hasNext}, [i \mapsto g] \rangle$	$\langle \mathbf{q}_3, [c \mapsto \{\mathbf{y}\}, i \mapsto \{\mathbf{g}\}] \rangle, \langle q_1, [c \mapsto \{x\}, i \mapsto \{y\}] \rangle,$ $\langle q_0, [c \mapsto \overline{\{x, y\}}, i \mapsto \overline{\{f, g\}}] \rangle, \langle q_0, [c \mapsto \overline{\{x\}}, i \mapsto \overline{\{f, g\}}] \rangle,$ $\langle q_0, [c \mapsto \overline{\{y\}}, i \mapsto \overline{\{f, g\}}] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{f, g\}}] \rangle$
8. body	No change to σ
9. tr $\langle \text{update}, [c \mapsto x] \rangle$	$\langle \mathbf{q}_2, [c \mapsto \{\mathbf{x}\}, i \mapsto \{\mathbf{y}\}] \rangle, \langle q_3, [c \mapsto \{y\}, i \mapsto \{g\}] \rangle, \langle q_0, [c \mapsto \overline{\{x\}}, i \mapsto \top] \rangle,$ $\langle q_0, [c \mapsto \overline{\{x\}}, i \mapsto \overline{\{f, g\}}] \rangle, \langle q_0, [c \mapsto \overline{\{x, y\}}, i \mapsto \overline{\{f, g\}}] \rangle$
10. tr $\langle \text{next}, [i \mapsto g] \rangle$	$\langle \mathbf{q}_1, [c \mapsto \{\mathbf{y}\}, i \mapsto \{\mathbf{g}\}] \rangle, \langle q_2, [c \mapsto \{x\}, i \mapsto \{y\}] \rangle,$ $\langle q_0, [c \mapsto \overline{\{x\}}, i \mapsto \overline{\{f, g\}}] \rangle, \langle q_0, [c \mapsto \overline{\{x\}}, i \mapsto \overline{\{g\}}] \rangle,$ $\langle q_0, [c \mapsto \top, i \mapsto \overline{\{g\}}] \rangle, \langle q_0, [c \mapsto \overline{\{x, y\}}, i \mapsto \overline{\{f, g\}}] \rangle$
11. body	No change to σ
12. tr $\langle \text{next}, [i \mapsto f] \rangle$	$\langle \mathbf{q}_4, [c \mapsto \{\mathbf{x}\}, i \mapsto \{\mathbf{y}\}] \rangle, \langle q_1, [c \mapsto \{y\}, i \mapsto \{g\}] \rangle,$ $\langle q_0, [c \mapsto \overline{\{x\}}, i \mapsto \overline{\{f, g\}}] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{f, g\}}] \rangle,$ $\langle q_0, [c \mapsto \top, i \mapsto \overline{\{f\}}] \rangle, \langle q_0, [c \mapsto \overline{\{x, y\}}, i \mapsto \overline{\{f, g\}}] \rangle$
13. body	$\langle q_1, [c \mapsto \{y\}, i \mapsto \{g\}] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{f, g\}}] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{f\}}] \rangle,$ $\langle q_0, [c \mapsto \overline{\{x, y\}}, i \mapsto \overline{\{f, g\}}] \rangle, \langle q_0, [c \mapsto \overline{\{x\}}, i \mapsto \overline{\{f, g\}}] \rangle$
14. tr $\langle \text{next}, [i \mapsto g] \rangle$	$\langle \mathbf{q}_5, [c \mapsto \{\mathbf{y}\}, i \mapsto \{\mathbf{g}\}] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{f, g\}}] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{g\}}] \rangle,$ $\langle q_0, [c \mapsto \overline{\{x, y\}}, i \mapsto \overline{\{f, g\}}] \rangle, \langle q_0, [c \mapsto \overline{\{x\}}, i \mapsto \overline{\{f, g\}}] \rangle$
15. body	$\langle q_0, [c \mapsto \overline{\{x\}}, i \mapsto \overline{\{f, g\}}] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{g\}}] \rangle,$ $\langle q_0, [c \mapsto \overline{\{x, y\}}, i \mapsto \overline{\{f, g\}}] \rangle, \langle q_0, [c \mapsto \top, i \mapsto \overline{\{f, g\}}] \rangle,$

Figure 3.9: Computing the tracematch state abstraction for the same sequence of instructions as in Figure 2.1, 2.3 and 2.7.

requirements and to detect statically when a tracematch may lead to unbounded space overhead. Freeing bindings early has the additional benefit of reducing the time required to find the binding requiring update when a transition statement is encountered. This time can be reduced further by maintaining suitable indexes on the binding set. On some realistic tracematches, these techniques yield speed improvements of multiple orders of magnitude. Thus, these techniques are necessary for a practical dynamic implementation of tracematches. A similar indexing technique is also applied in JavaMOP [15].

A second approach, of which our work is an example, is to use static analysis to reduce the number of transition statements that must be instrumented. Another example is the work of Bodden et al. [12], which we discuss in more detail in Section 5.2. In follow-on work, Bodden et al. [13] have augmented the analysis with a suite of intraprocedural flow-sensitive analyses. The analyses combine local alias information with inexpensive whole program summary information. In their benchmark suite, tracematches described mostly local patterns, thus a careful combination of these analyses could detect many violations and with few false positives. Guyer and Lin’s [34, 35] client-driven pointer analysis is also related. Their analysis is based on a subset-based may-point-to analysis followed by flow-sensitive propagation of states on the abstract object represented by each allocation site. When a property cannot be proven, the analysis iteratively refines the context-sensitivity of the points-to analysis in order to improve precision and hopefully verify the property. Dwyer and Purandare [28] also use static analysis to reduce the cost of dynamic typestate verification by proving that certain transitions need not be instrumented because they cannot lead to a violation.

The static abstraction most closely related to ours is one used by Fink et al.’s typestate analysis [29, 30]. Their analysis also uses an object abstraction in which an abstract object represents at most one concrete object, and it uses the focus operation to achieve this. Their object abstraction is more precise but more costly than ours because it tracks access paths through fields, rather than only references from local variables. In addition, the object abstraction contains the allocation site of each object, which provides the same information as a subset-based may-point-to analysis. It would be possible to replace the object abstraction in our tracematch analysis with that of Fink et al. to improve precision. Unlike tracematches, typestate applies only to a single object. Therefore, rather than requiring a separate tracematch abstraction, Fink et al. simply augment the abstraction of each object with its typestate.

Another object abstraction similar to ours is used by Cherem and Rugina [17] to statically insert free instructions to deallocate some objects earlier than the garbage collector can get to them. This application makes use of the property that the abstract object corresponding to a given concrete object can be traced through the control flow graph. The

object abstraction is also more precise than ours, but less so than Fink’s; it maintains reference counts from individual fields rather than full access paths. This object abstraction could also be substituted in the tracematch analysis.

Multiobject temporal constraints have been studied by Jaspan and Aldrich [40, 41] in the context of plugins for object-oriented frameworks. The motivation for their work is that since large frameworks introduce complex constraints that are difficult to understand and document, it is difficult for programmers to develop plugins which conform to the constraints laid down by the framework. They present a lightweight specification system which allows the framework developers to specify runtime interactions between objects and the framework constraints that depend on these interactions. A static analysis is presented that uses these specifications and analyzes the plugin code for any violations of the constraints laid down by the framework developer. The analysis has been shown to work on real world examples from the ASP.NET and Eclipse framework.

Torlak et al. [66] present Tracker, a tool that leverages static analysis to find resource leaks in Java Programs. The analysis takes in specifications of procedures that acquire and release system resources and aims to establish whether the allocation of a resource is followed by its release on all execution paths. This is achieved by symbolically tracking each resource through paths in the Control Flow Graph until either it is released or it becomes unreachable without being released and is classified to have leaked. As is the case for all analyses that track objects, to be effective the analysis computes precise inter-procedural must-alias information. However, instead of a whole-program alias analysis, Tracker performs efficient must-alias reasoning using selective equality predicates. This is achieved by computing a set of must-access-paths to a tracked resource. It is also worth mentioning that Tracker pays special attention to exceptional flow since programmers have been known to incorrectly rely on try-catch-finally blocks to release resources.

Ramalingam et al. [53] present a verification technique for checking that a client program follows conventions required by an API. The effects and requirements of the API methods are specified using a declarative language. The system constructs a predicate abstraction of the API internals from the specification. The predicate abstraction is used to prove that a client program satisfies the requirements. The system was used to check correct usage of iterators in client programs of up to 2396 LOC.

The Metal system [37] is an unsound state-based bug finder for C. The core system does not consider aliasing; instead an automaton is maintained for each variable, regardless of the object to which it may be pointing. It uses heuristics such as *synonyms* (an unsound variation of must-alias analysis) to partially recover from this unsoundness. Metal was successful in finding many locking bugs in the Linux kernel.

An alternative to analyzing arbitrary aliasing is to use a specialized type system to restrict aliasing. An advantage of this approach is modularity: a violation of the type system is local, as are violations of the tpestate property when the aliasing restrictions are obeyed. A disadvantage is that it is difficult to apply to existing, unannotated code, although sometimes annotations can be inferred automatically. The Vault system [22] uses *keys*, unique pointers to objects. The type system prevents duplication of keys, and each tpestate change is correlated with a set of keys held at the point of the change. The same authors propose a system for specifying tpestates of object-oriented programs, focusing especially on object-oriented features such as subtyping [23]. To handle aliasing, they allow objects to be either unaliased and updateable, or possibly aliased and non-updateable. CQual [31] is another system similar to but simpler than Vault. Bierhoff and Aldrich [8, 9] present a type system in which both aliasing and tpestate information are specified using types. A key innovation of their system are *access permissions*, which specify whether a pointer is unique or whether it is aliased but with fine-grained restrictions on which aliases may read or write to the object. Access permissions can be split for multiple aliases and later recombined, making them more flexible than earlier aliasing control mechanisms.

3.5 Summary

The static analysis of properties that involve multiple objects cannot be represented using a single abstraction. Two abstractions are required: one to represent the objects in the program, and a second to represent the state of a group of abstract objects. In this chapter, we presented the alias set abstraction as a precise way of representing objects. This object abstraction supports precise may-alias information as well as the flow sensitive tracking of objects along control flow paths. We also presented a state abstraction which uses the alias set abstraction for objects to associate a state to a group of relevant objects. Additionally, we specified a correctness relation and proved that the transfer functions for the two abstractions preserve the correctness relation. In Chapter 5, we use a dataflow analysis algorithm to compute the abstractions presented in this chapter. The resulting abstraction is then used for the verification of the specified temporal property.

Chapter 4

Extensions to IFDS

In Chapter 1 we highlighted three properties that an analysis for tracematches requires. Two of them, precise aliasing information and the ability to flow sensitively track individual objects along control flow paths, have been shown to hold for the abstraction as discussed in Chapter 3. Any standard dataflow analysis algorithm which can compute precise context-sensitive interprocedural information can be used to compute this flow-sensitive abstraction. We have chosen to use the Interprocedural Finite Distributive Subset (IFDS) algorithm [54]. This is an efficient and precise, context-sensitive and flow-sensitive dataflow analysis algorithm for the class of problems that satisfy its restrictions. Although this class includes the classic bit-vector dataflow problems, the original IFDS algorithm is not directly suitable for more interesting problems for which context- and flow-sensitivity would be useful, particularly problems involving objects and pointers. However, the algorithm can be extended to solve this larger class of problems, and in this chapter, we present four such extensions. In the following chapter, we discuss using this extended IFDS algorithm to implement the tracematch analysis.

The IFDS algorithm is an efficient dynamic programming instantiation of the functional approach to interprocedural analysis [62]. The fundamental restrictions of the algorithm, which we do not seek to eliminate, are that the analysis domain must be a powerset of some finite set **Dom**, and that the dataflow functions must be distributive. We present a detailed overview of the IFDS algorithm in Section 4.1, and further illustrate the algorithm with a running example variable type analysis in Section 4.2.

A more practical restriction is that the set **Dom** must be small, because the algorithm requires as input a so-called exploded supergraph, and the number of nodes in this supergraph is approximately the product of the size of **Dom** and the number of instructions in

the program. Our first extension, presented in Section 4.3, removes the restriction on the size of **Dom** by enabling the algorithm to compute only those parts of the supergraph that are actually reached in the analysis. This allows the algorithm to be used for problems in which **Dom** is theoretically large, but only a small subset of **Dom** is encountered during the analysis, which is typical of analyses modelling objects and pointers such as our object abstraction from Section 3.2.

A second practical restriction of the original IFDS algorithm is that it provides limited information to flow functions modelling return flow from a procedure. For many analyses, mapping dataflow facts from the callee back to the caller requires information about the state before the procedure was called. In Section 4.4, we extend the IFDS algorithm to provide this information to the return flow function.

A third limitation of many standard dataflow analysis algorithms, IFDS included, is that they can be less precise on a program in Static Single Assignment (SSA) form [20] than on the original non-SSA form of the program. When an instruction has multiple control flow predecessors, incoming dataflow facts are merged before the flow function is applied; this imprecisely models the semantics of ϕ instructions in SSA form. In Section 4.5, we present an example that exhibits this imprecision, and we extend the IFDS algorithm to avoid it, so that it is equally precise on SSA form as on non-SSA form programs. SSA form is not only a convenience; SSA form can be used to improve running time and space requirements of analyses such as alias set analysis [50].

Finally, the IFDS algorithm does not take advantage of any structure in the set **Dom**. In many analyses of objects and pointers, some elements of **Dom** subsume others. In Section 4.6, we present an extension that exploits such structure to reduce analysis time.

4.1 The Original IFDS Algorithm

The IFDS algorithm of Reps et al. [54] is a dynamic programming algorithm that computes a merge-over-all-valid paths solution to interprocedural, finite, distributive, subset problems. The merge is over *valid* paths in that procedure calls and returns are correctly matched (i.e. the analysis is context sensitive). The algorithm requires that the domain of dataflow facts be the powerset of a finite set **Dom**, with set union as the merge operator. The data flow functions must be distributive over set union: $f(a) \cup f(b) = f(a \cup b)$.

The algorithm follows the summary function approach to context-sensitive interprocedural analysis [62], in that it computes functions in $\mathcal{P}(\mathbf{Dom}) \rightarrow \mathcal{P}(\mathbf{Dom})$ that summarize the effect of ever-longer sections of code on any given subset of **Dom**. The al-

gorithm uses $O(E|\mathbf{Dom}|^3)$ time in the worst case, where E is the number of control-flow edges in the program. The key to the efficiency of the algorithm is the compact representation of functions, made possible by their distributivity. For example, suppose the set $S = \{a, b, c\}$ is a subset of \mathbf{Dom} . By distributivity, $f(S)$ can be computed as $f(S) = f(\{\}) \cup f(\{a\}) \cup f(\{b\}) \cup f(\{c\})$. Thus every distributive function in $\mathcal{P}(\mathbf{Dom}) \rightarrow \mathcal{P}(\mathbf{Dom})$ is uniquely defined by its value on the empty set and on every singleton subset of \mathbf{Dom} . Equivalently, the function can be defined by a bipartite graph $\langle \mathbf{Dom} \cup \{\mathbf{0}\}, \mathbf{Dom}, E \rangle$, where E is a set of edges from elements of $\mathbf{Dom} \cup \{\mathbf{0}\}$ to elements of (a second copy of) \mathbf{Dom} , i.e., the function can be efficiently represented as a graph with at most $(|\mathbf{Dom}| + 1)^2$ edges [54, Section 3]. The graph contains an edge from d_1 to d_2 if and only if $d_2 \in f(\{d_1\})$. The special $\mathbf{0}$ vertex represents the empty set: the edge $\mathbf{0} \rightarrow d$ indicates that $d \in f(\{\})$. The function represented by the graph is defined to be $f(S) = \{b : (a, b) \in E \wedge (a = \mathbf{0} \vee a \in S)\}$. For example, the graph in Figure 4.1(a) represents the function $g(S) = \{x : x \in \{b, c\} \vee (x = d \wedge d \in S)\}$, which can be written more simply as $g(S) = (S \setminus \{a\}) \cup \{b, c\}$.

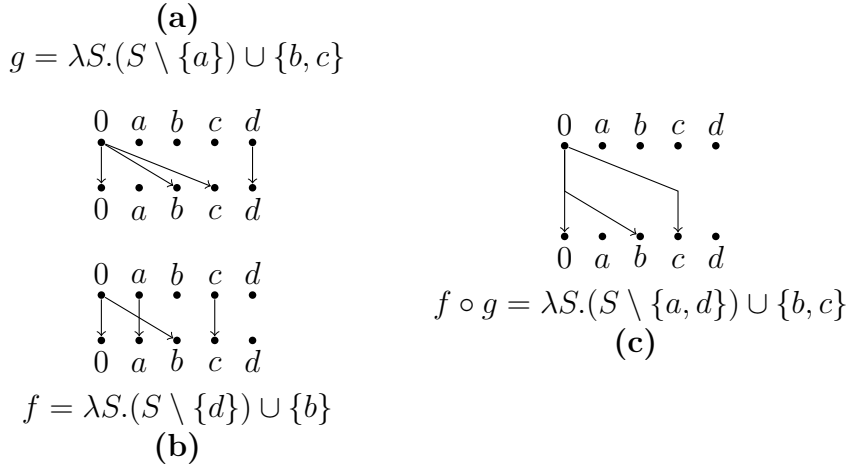


Figure 4.1: Compact representation of functions and their composition

The composition $f \circ g$ of two functions can be computed by combining their graphs, merging the nodes of the range of g with the corresponding nodes of the domain of f , then computing reachability from the nodes of the domain of g to the nodes of the range of f . That is, a relational product of the sets of edges representing the two functions gives a set of edges representing their composition. An example is shown in Figure 4.1. The graph in Figure 4.1(c), representing $f \circ g$, contains an edge from x to y whenever there is an edge

from x to some z in the representation of g in Figure 4.1(a) and an edge from the same z to y in the representation of f in Figure 4.1(b).

We have reproduced the original IFDS algorithm [54] in Figure 4.2. The input to the algorithm is a so-called exploded supergraph ($G_{IP}^\#$) that represents both the program being analyzed and the dataflow functions. The supergraph is constructed from the interprocedural control flow graph (ICFG) of the program by replacing each instruction with the graph representation of its flow function. Thus the vertices of the supergraph are pairs $\langle l, d \rangle$, where l is a label in the program and $d \in \mathbf{Dom} \cup \{\mathbf{0}\}$. The supergraph contains an edge $\langle l, d \rangle \rightarrow \langle l', d' \rangle$ if the ICFG contains an edge $l \rightarrow l'$ and $d' \in f(\{d\})$ (or $d' \in f(\{\})$ when $d = \mathbf{0}$), where f is the flow function of the instruction at l . For each interprocedural call or return edge in the ICFG, the supergraph contains a set of edges representing the flow function associated with the call or return. The flow function on the call edge typically maps facts about actuals in the caller to facts about formals in the callee. The merge-over-all-valid paths solution at label l contains exactly the elements d of \mathbf{Dom} for which there exists a valid path from $\langle s, \mathbf{0} \rangle$ to $\langle l, d \rangle$ in the supergraph. The dataflow analysis therefore reduces to valid-path reachability on the supergraph.

```

declare PathEdge, WorkList, SummaryEdge: global edge set
algorithm Tabulate( $G_{IP}^\#$ )
begin
1   Let ( $N^\#, E^\#$ ) =  $G_{IP}^\#$ 
2   PathEdge := {  $\langle s_{main}, \mathbf{0} \rangle \rightarrow \langle s_{main}, \mathbf{0} \rangle$  }
3   WorkList := {  $\langle s_{main}, \mathbf{0} \rangle \rightarrow \langle s_{main}, \mathbf{0} \rangle$  }
4   SummaryEdge :=  $\emptyset$ 
5   ForwardTabulateSLRPs()
6   foreach  $n \in N^\#$  do
7      $X_n := \{ d_2 \in \mathbf{Dom} \mid \exists d_1 \in (\mathbf{Dom} \cup \{\mathbf{0}\})$ 
           s.t.  $\langle s_{procOf(n)}, d_1 \rangle \rightarrow \langle n, d_2 \rangle \in \text{PathEdge} \}$ 
8   od
end
procedure Propagate( $e$ )
begin
9   if  $e \notin \text{PathEdge}$  then Insert  $e$  into PathEdge; Insert  $e$  into WorkList; fi
end

           continued ...

```

Figure 4.2: Original IFDS Algorithm reproduced.

```

procedure ForwardTabulateSLRPs()
begin
10   while WorkList  $\neq \emptyset$  do
11     Select and remove an edge  $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$  from WorkList
12     switch  $n$ 
13       case  $n \in Call_p$  :
14         foreach  $d_3$  s.t.  $\langle n, d_2 \rangle \rightarrow \langle s_{calledProc(n)}, d_3 \rangle \in E^\#$  do
15           Propagate( $\langle s_{calledProc(n)}, d_3 \rangle \rightarrow \langle s_{calledProc(n)}, d_3 \rangle$ )
16         od
17         foreach  $d_3$  s.t.  $\langle n, d_2 \rangle \rightarrow \langle returnSite(n), d_3 \rangle \in (E^\# \cup SummaryEdge)$  do
18           Propagate( $\langle s_p, d_1 \rangle \rightarrow \langle returnSite(n), d_3 \rangle$ )
19         od
20       end case
21       case  $n \in e_p$  :
22         foreach  $c \in callers(p)$  do
23           foreach  $d_4, d_5$  s.t.  $\langle c, d_4 \rangle \rightarrow \langle s_p, d_1 \rangle \in E^\#$  and
24              $\langle e_p, d_2 \rangle \rightarrow \langle returnSite(c), d_5 \rangle \in E^\#$  do
25               if  $\langle c, d_4 \rangle \rightarrow \langle returnSite(c), d_5 \rangle \notin SummaryEdge$  then
26                 Insert  $\langle c, d_4 \rangle \rightarrow \langle returnSite(c), d_5 \rangle$  into SummaryEdge
27                 foreach  $d_3$  s.t.  $\langle s_{procOf(c)}, d_3 \rangle \rightarrow \langle c, d_4 \rangle \in PathEdge$  do
28                   Propagate( $\langle s_{procOf(c)}, d_3 \rangle \rightarrow \langle returnSite(c), d_5 \rangle$ )
29                 od
30               fi
31             od
32           od
33       end case
34       case  $n \in (N_p - Call_p - \{e_p\})$  :
35         foreach  $\langle m, d_3 \rangle$  s.t.  $\langle n, d_2 \rangle \rightarrow \langle m, d_3 \rangle \in E^\#$  do
36           Propagate( $\langle s_p, d_1 \rangle \rightarrow \langle m, d_3 \rangle$ )
37         od
38       end case
39     end switch
od
end

```

Figure 4.2: Original IFDS Algorithm reproduced(contd.)

The IFDS algorithm works by incrementally constructing two tables, PathEdge and SummaryEdge, representing the flow functions of ever longer sequences of code. The PathEdge table contains triples $\langle d, l, d' \rangle$, indicating that there is a path from $\langle s_p, d \rangle$ to $\langle l, d' \rangle$, where s_p is the start node of the procedure containing l . These triples are often written in the form $\langle s_p, d \rangle \rightarrow \langle l, d' \rangle$ for clarity, but the start node s_p is uniquely determined by l , so it is not stored in an actual implementation. The SummaryEdge table contains triples $\langle c, d, d' \rangle$, where c is the label of a call site. Such a triple indicates that $d' \in f(\{d\})$, where f is a flow function summarizing the effect of the procedure called at c . These triples are often written $\langle c, d \rangle \rightarrow \langle r, d' \rangle$, where r is the instruction following c . For convenience, Reps’s presentation of the IFDS algorithm [54] assumes that in the ICFG, every call site c has a single successor, a no-op “return site” node r .

The PathEdge and SummaryEdge tables are interdependent. Consider the edge $\langle s_p, d_1 \rangle \rightarrow \langle e_p, d_2 \rangle$ added to PathEdge, in which e_p is the exit node of some procedure p . This edge means that $d_2 \in f_p(\{d_1\})$, where f_p is the flow function representing the effect of the entire procedure p . As a result, for every call site c calling procedure p , a corresponding triple must be added to SummaryEdge indicating the newly-discovered effect at that call site. In fact, several such triples may be needed for a single edge added to PathEdge, since the effect of a procedure at c is represented not just by f_p , but by the composition $f_r \circ f_p \circ f_c$, where f_c and f_r are the flow functions representing the function call and return. This composition is computed by combining the graphs representing f_c and f_r from the supergraph with the newly discovered edge $\langle d_1, d_2 \rangle$ of f_p . That is, for each d_4 and d_5 such that $\langle d_4, d_1 \rangle \in f_c$ and $\langle d_2, d_5 \rangle \in f_r$, $\langle c, d_4, d_5 \rangle$ is added to SummaryEdge. This is performed in lines 23 to 25 of the algorithm.

Conversely, consider a triple $\langle c, d_4, d_5 \rangle$ added to SummaryEdge, indicating a new effect of the call at c . As a result, for each d_3 such that there is a path from $\langle s, d_3 \rangle$ to $\langle c, d_4 \rangle$, where s is the start node of the procedure containing c , there is now a valid path from $\langle s, d_3 \rangle$ to $\langle r, d_5 \rangle$, where r is the successor of c . Thus $\langle s, d_3 \rangle \rightarrow \langle r, d_5 \rangle$ must be added to PathEdge. This is performed in lines 26 to 28 of the algorithm.

4.2 Running Example: Type Analysis

The extensions we have made to the IFDS algorithm were motivated by the alias set analysis (Section 3.2) and the tracematch state analysis (Section 3.3). The same extensions are applicable to many other kinds of analyses. In order to not clutter the discussion with the complexity of these domains, we will use a much simpler analysis domain to illustrate the extensions.

The example analysis is a variation of Variable Type Analysis (VTA) [65] for Java. The analysis computes the set of possible types for each variable. This information can be used to construct a call graph or to check the validity of casts. At each program point p , the analysis computes a subset of **Dom**, where **Dom** is defined as the set of all pairs $\langle v, t \rangle$, where v is a variable in the program and t is a class in the program. The presence of the pair $\langle v, t \rangle$ in the subset indicates that the variable v may point to an object of type t .

For the sake of the example, we would like the analysis to analyze only the application code and not the large standard library. The analysis therefore makes conservative assumptions about the unanalyzed code based on statically declared types. For example, if $m()$ is in the library, the analysis assumes that $m()$ could return an object of the declared return type of $m()$ or any of its subtypes. To this end we amend the meaning of a pair $\langle v, t \rangle$ to indicate that v may point to an object of type t or any of its subtypes.

The unanalyzed code could write to fields in the heap, either directly or by calling back into application code. To keep the analysis sound yet simple, we make the conservative assumption that a field can point to any object whose type is consistent with its declared type. We model a field read $\mathbf{x} = \mathbf{y}.f$ with the pair $\langle \mathbf{x}, t \rangle$, where t is the declared type of f . We make these simplifications because the analysis is intended to illustrate the extensions to the IFDS algorithm, not necessarily as a practical analysis.

When the declared type of a field is an interface, the object read from it could be of any class that implements the interface. For a read from such a field, we generate multiple pairs $\langle \mathbf{x}, t_i \rangle$, where the t_i are all classes that implement the interface. If class B extends A and both implement the interface, it is redundant to include $\langle \mathbf{x}, B \rangle$ since $\langle \mathbf{x}, A \rangle$ already includes all subclasses of A , including B . For efficiency, we generate only those pairs $\langle \mathbf{x}, t_i \rangle$ where t_i implements the interface and its superclass does not.

The analysis is performed on an intermediate representation comprising the following kinds of instructions, in addition to procedure calls and returns: $s ::= x \leftarrow y \mid y.f \leftarrow x \mid x \leftarrow y.f \mid x \leftarrow \mathbf{null} \mid x \leftarrow \mathbf{new} T \mid x \leftarrow (T)y$. The instructions copy pointers between variables, store and load objects to and from fields, assign **null** to variables, create new objects and cast objects to a given type, respectively. We use $\llbracket s \rrbracket_P : \mathcal{P}(\mathbf{Dom}) \rightarrow \mathcal{P}(\mathbf{Dom})$ to denote the transfer function for the type analysis. The IFDS algorithm requires the transfer function to be decomposed into its effect on each individual element of **Dom** and on the empty set. We decompose it as $\llbracket s \rrbracket : \mathbf{Dom} \cup \{\mathbf{0}\} \rightarrow \mathcal{P}(\mathbf{Dom})$ and define $\llbracket s \rrbracket_P(P) \triangleq \llbracket s \rrbracket(\mathbf{0}) \cup \bigcup_{d \in P} \llbracket s \rrbracket(d)$. The decomposed transfer function $\llbracket s \rrbracket$ is defined in Figure 4.3.

The flow function for a copy instruction $(x \leftarrow y)$ applied to a pair $\langle v, t \rangle$ requires three cases. When v is the same as y , the pair $\langle v, t \rangle$ is preserved and, since the value of y is copied to x , a new pair $\langle x, t \rangle$ is created. If v is neither x nor y , the value of v is unaffected by the

$$\begin{aligned}
\llbracket x \leftarrow y \rrbracket(\langle v, t \rangle) &\triangleq \begin{cases} \{\langle x, t \rangle, \langle y, t \rangle\} & \text{if } v = y \\ \{\langle v, t \rangle\} & \text{if } v \neq y \text{ and } v \neq x \\ \emptyset & \text{if } v \neq y \text{ and } v = x \end{cases} \\
\llbracket y.f \leftarrow x \rrbracket(\langle v, t \rangle) &\triangleq \{\langle v, t \rangle\} \\
\llbracket x \leftarrow \mathbf{null} | \mathbf{new } T | y.f \rrbracket(\langle v, t \rangle) &\triangleq \begin{cases} \{\langle v, t \rangle\} & \text{if } v \neq x \\ \emptyset & \text{otherwise} \end{cases} \\
\llbracket x \leftarrow \mathbf{new } T \rrbracket(\mathbf{0}) &\triangleq \{\langle x, T \rangle\} \\
\llbracket x \leftarrow y.f \rrbracket(\mathbf{0}) &\triangleq \{\langle x, c \rangle : c \in \text{implClasses}(\text{type}(f))\} \\
\llbracket x \leftarrow (T)y \rrbracket(\langle v, t \rangle) &\triangleq \bigcup_{c \in \text{implClasses}(T)} \text{cast}(x, y, c)(\langle v, t \rangle) \\
\text{cast}(x, y, t_2)(\langle v, t_1 \rangle) &\triangleq \begin{cases} \{\langle v, t_1 \rangle\} & \text{if } v \neq x \text{ and } v \neq y \\ \emptyset & \text{if } v = x \text{ and } v \neq y \\ \{\langle x, t_1 \rangle, \langle y, t_1 \rangle\} & \text{if } v = y \text{ and } t_1 <: t_2 \\ \{\langle x, t_2 \rangle, \langle y, t_2 \rangle\} & \text{if } v = y \text{ and } t_2 <: t_1 \\ \emptyset & \text{if } v = y \text{ and } t_1 \text{ and } t_2 \text{ are unrelated} \end{cases} \\
\llbracket s \rrbracket(\mathbf{0}) &\triangleq \emptyset \text{ if } s \neq x \leftarrow y.f \text{ and } s \neq x \leftarrow \mathbf{new}
\end{aligned}$$

Figure 4.3: Intraprocedural flow functions for the Variable Type Analysis

copy and the pair is therefore preserved. If v is x , and x and y are distinct, then since the existing value of x is overwritten by the new value, the existing pair $\langle v, t \rangle$ describing the old value of v is discarded, and the result is the empty set.

The store instruction $(v.f \leftarrow x)$ has no effect on the values of local variables, and its flow function is therefore the identity. The flow function for an assignment to x via a load, **new** or **null** does not affect $\langle v, t \rangle$, unless v is x , in which case the existing value of x is overwritten, so the pair is dropped from the set. An allocation instruction $x \leftarrow \mathbf{new } T$ generates the new pair $\langle x, T \rangle$. A load instruction $x \leftarrow y.f$ creates the pair $\langle x, t \rangle$, if the type of the field f is a class t , or the set of pairs $\langle x, t_i \rangle$, if the type of the field f is an interface, where the t_i are all of the classes implementing the interface, as explained earlier. The helper function $\text{implClasses}(t)$ computes this set of classes.

The most interesting case is the cast instruction $(x \leftarrow (T)y)$. The first complication is that T could be an interface. Such a cast is treated as casts to all classes implementing T . The flow function is the union of the flow functions modelling casts to these classes,

reflecting the fact that the cast to the interface type succeeds if the cast to at least one of the implementing classes succeeds. For the simpler case of a cast to a type t_2 that is a class, not an interface, there are still several cases. The cast instruction has no effect on $\langle v, t_1 \rangle$ when v is neither x nor y . When v is x , the pair is dropped because the cast overwrites the existing value of x . When v is y and $t_1 <: t_2$, indicating that we already know that y points to an object whose type is a subtype of t_2 , the cast acts as a copy and the new pair $\langle x, t_1 \rangle$ is generated. When v is y and $t_2 <: t_1$, indicating that y is being cast to a more restrictive type than the type it is already known to point to, we generate the new pair $\langle x, t_2 \rangle$, indicating that x must point to a subtype of the more restrictive cast type. The original pair $\langle y, t_1 \rangle$ can also be changed to the more precise pair $\langle y, t_2 \rangle$, since if control flow proceeds after the cast, the cast must have succeeded, and therefore y must point to an object whose type is a subtype of the cast type. For the purposes of the example, we assume that a failing cast terminates the program rather than being caught by an exception handler; catching class cast exceptions is rare in practice.

4.3 Demand Construction of the Supergraph

The number of nodes in the exploded supergraph $G^\#$ is $|\text{Inst}| \times (|\mathbf{Dom}| + 1)$, where $|\text{Inst}|$ is the number of instructions in the program and $|\mathbf{Dom}|$ is the size of \mathbf{Dom} . In many analyses, \mathbf{Dom} , though finite, is very large. For example, in the alias set analysis that we intend to compute for the tracematch analysis, \mathbf{Dom} is a union of the powersets of the sets of variables of all procedures, and therefore exponential in the number of variables in a procedure. In our example variable type analysis, $\mathbf{Dom} = |\text{Var}| \times |\text{Class}|$, where Var is the set of all variables in the program and Class is the set of all classes in the program, so $|\mathbf{Dom}|$ is over one million even for a moderate program with a thousand variables and a thousand classes. Constructing and storing a graph that is a million times larger than the ICFG is not practical. In practice, only a small subgraph of $G^\#$ is reachable by valid paths from $\langle s_{main}, \mathbf{0} \rangle$ and therefore explored by the algorithm. Unfortunately, we cannot know exactly which subgraph this is before running the IFDS algorithm, since determining which nodes are reachable is exactly what the IFDS algorithm does. Therefore, our first extension to the IFDS algorithm modifies it to request only those parts of the supergraph that it encounters, instead of requiring the whole supergraph as input.

The extended IFDS algorithm with all four of our extensions is shown in Figure 4.4. Parts of the algorithm that were changed from the original or added are underlined.

```

declare PathEdge, WorkList, SummaryEdge, Incoming, EndSummary: global
algorithm Tabulate(flow, passArgs, returnVal, callFlow)
    ⋮
procedure ForwardTabulateSLRPs()
begin
10     while WorkList  $\neq \emptyset$  do
11         Select and remove an edge  $\langle s_p, d_1 \rangle \xrightarrow{\pi} \langle n, d_2 \rangle$  from WorkList
12         switch  $n$ 
13             case  $n \in Call_p$  :
14                 foreach  $d_3 \in passArgs(\langle n, d_2 \rangle)$  do
15                     Propagate  $(\langle s_{calledProc(n)}, d_3 \rangle \xrightarrow{0} \langle s_{calledProc(n)}, d_3 \rangle)$ 
15.1                 Incoming  $[\langle s_{calledProc(n)}, d_3 \rangle] \cup = \langle n, d_2 \rangle$ 
15.2                 foreach  $\langle e_p, d_4 \rangle \in EndSummary[\langle s_{calledProc(n)}, d_3 \rangle]$  do
15.3                     foreach  $d_5 \in returnVal(\langle e_p, d_4 \rangle, \langle n, d_2 \rangle)$  do
15.4                         Insert  $\langle n, d_2 \rangle \rightarrow \langle returnSite(n), d_5 \rangle$  into SummaryEdge
15.5                     od
15.6                 od
16             od
17             foreach  $d_3$  s.t.  $d_3 \in callFlow(\langle n, d_2 \rangle)$  or
                 $\langle n, d_2 \rangle \rightarrow \langle returnSite(n), d_3 \rangle \in SummaryEdge$  do
18                 Propagate  $(\langle s_p, d_1 \rangle \xrightarrow{n} \langle returnSite(n), d_3 \rangle)$ 
19             od
20         end case
                continued ...

```

Figure 4.4: Extended IFDS Algorithm

```

21         case  $n \in e_p$  :
21.1         EndSummary [ $\langle s_p, d_1 \rangle$ ]  $\cup = \langle e_p, d_2 \rangle$ 
22         foreach  $\langle c, d_4 \rangle \in \text{Incoming}[\langle s_p, d_1 \rangle]$  do
23             foreach  $d_5 \in \text{returnVal}(\langle e_p, d_2 \rangle, \langle c, d_4 \rangle)$  do
24                 if  $\langle c, d_4 \rangle \rightarrow \langle \text{returnSite}(c), d_5 \rangle \notin \text{SummaryEdge}$  then
25                     Insert  $\langle c, d_4 \rangle \rightarrow \langle \text{returnSite}(c), d_5 \rangle$  into SummaryEdge
26                     foreach  $d_3$  s.t.  $\langle s_{\text{procOf}(c)}, d_3 \rangle \rightarrow \langle c, d_4 \rangle \in \text{PathEdge}$  do
27                         Propagate( $\langle s_{\text{procOf}(c)}, d_3 \rangle \xrightarrow{c} \langle \text{returnSite}(c), d_5 \rangle$ )
28                     od
29                 fi
30             od
31         od
32     end case
33     case  $n \in (N_p - \text{Call}_p - \{e_p\})$  :
34         foreach  $m, d_3$  s.t.  $n \rightarrow m \in \text{CFG}$  and  $d_3 \in \text{flow}(\langle n, d_2 \rangle, \pi)$  do
35             Propagate( $\langle s_p, d_1 \rangle \xrightarrow{n} \langle m, d_3 \rangle$ )
36         od
37     end case
38 end switch
39 od
end

```

Figure 4.4: Extended IFDS Algorithm (contd.)

The input to the extended algorithm is a function that, given a supergraph node n^\sharp , computes all of the edges leaving that node (i.e. the flow function of the desired analysis). For clarity of presentation, we have split this function into four separate functions:

- $\text{flow}(n^\sharp)$ computes all intraprocedural edges.¹
- $\text{passArgs}(n^\sharp)$ computes call-to-start edges when n^\sharp is at a call site.
- $\text{returnVal}(n^\sharp)$ computes exit-to-return-site edges when n^\sharp is at the exit of a procedure.²

¹In Figure 4.4, flow has a second parameter π , which will be explained in Section 4.5.

²In Figure 4.4, returnVal has a second node parameter, which will be explained in Section 4.4.

- `callFlow(n^\sharp)` computes call-to-return-site edges when n^\sharp is at a call site. These edges model procedure-local information that is not affected by the called procedure.

The original IFDS algorithm queries the edges of the supergraph E^\sharp in five places. The queries on lines 14, 17 and 34, and the second query on line 23 can simply be replaced by calls to `passArgs`, `callFlow`, `flow`, and `returnVal`, respectively.

However, the first query on line 23 asks to evaluate the inverse of the flow function: find all call nodes $\langle c, d_4 \rangle$ from which an edge leads to the procedure start node $\langle s_p, d_1 \rangle$. This would require computing the inverse of the flow function, which can be difficult for many analyses. Moreover, even though $\langle s_p, d_1 \rangle$ is reachable in G^\sharp , many of its predecessors in E^\sharp may not be, and enumerating them may be intractable. For example, for an alias set analysis, the number of predecessors for most nodes is $2^{|\text{Var}|-1}$, where $|\text{Var}|$ is the number of variables in the calling procedure. The extended algorithm therefore maintains a set `Incoming`[$\langle s_p, d_1 \rangle$] that records nodes that the analysis has observed to be reachable and predecessors of $\langle s_p, d_1 \rangle$. Whenever the call to `passArgs`($\langle n, d_2 \rangle$) in line 14 returns $\langle s_p, d_3 \rangle$, $\langle n, d_2 \rangle$ is added in line 15.1 to `Incoming`($\langle s_p, d_3 \rangle$).

An obvious issue with querying the set of nodes already observed to be predecessors of $\langle s_p, d_1 \rangle$ is what must be done when a new predecessor is observed later. The solution is to keep track of exit nodes for which a given value of `Incoming` has been queried (line 21.1). Then, whenever a new predecessor is observed, those exit nodes are reprocessed to reflect the new predecessor. A simple way to reprocess the exit nodes correctly is to add them to the worklist. However, this approach is very inefficient, because whenever a new predecessor is added at one call site, the effect of the procedure is reprocessed for all predecessors at all call sites of the procedure. This intuitively poor performance was confirmed by our experience with the initial implementation of the algorithm.

A better way to reprocess the exit node is to recognize that when a new predecessor of $\langle s_p, d_1 \rangle$ is observed, the predecessor tells us the relevant call site. Instead of adding the corresponding exit node to the worklist, we can immediately process that exit node, but do only the work necessary for that one predecessor. Concretely, we duplicate the effect of lines 24 through 29 after line 15.1. The effect of lines 24, 25 and 29, adding the appropriate edge to `SummaryEdge`, is done in lines 15.3 through 15.5. The effect of lines 26 through 28 is already done by lines 17 through 19 of the original algorithm.

4.3.1 Eliminating the SummaryEdge Table

Once the algorithm has been modified to record nodes corresponding to callers of a procedure and the appropriate modifications have been made to reprocess the exit nodes as

discussed above, another opportunity to optimize the algorithm arises. At line 15.4 of the extended algorithm, the summary edge e representing the effect of calling the procedure p on the domain element d_2 at the call site n (node $\langle n, d_2 \rangle$) is added to the SummaryEdge table. Then later, at lines 17-19, this summary edge is retrieved and composed with the path edge from a start node $\langle s_p, d_1 \rangle$ and leading to $\langle n, d_2 \rangle$ to produce an edge e' from the start node to the return site of n . This step, first adding e into the SummaryEdge table only to immediately retrieve it, is unnecessary. Instead, the path edge e' can be created directly and propagated (Line 15.4 in Figure 4.5 shown with a wavy underline). Since all SummaryEdges originating at $\langle n, d_2 \rangle$ would already have been propagated at Line 15.4, there is no longer any need to repeat this, so the second loop condition at Line 17

```

declare PathEdge, WorkList, Incoming, EndSummary: global
algorithm Tabulate(flow, passArgs, returnVal, callFlow)
    :
procedure ForwardTabulateSLRPs()
begin
10   while WorkList  $\neq \emptyset$  do
11     Select and remove an edge  $\langle s_p, d_1 \rangle \xrightarrow{\pi} \langle n, d_2 \rangle$  from WorkList
12     switch  $n$ 
13       case  $n \in Call_p$  :
14         foreach  $d_3 \in passArgs(\langle n, d_2 \rangle)$  do
15           Propagate( $\langle s_{calledProc(n)}, d_3 \rangle \xrightarrow{0} \langle s_{calledProc(n)}, d_3 \rangle$ )
15.1         Incoming [ $\langle s_{calledProc(n)}, d_3 \rangle$ ]  $\cup = \langle n, d_2 \rangle$ 
15.2         foreach  $\langle e_p, d_4 \rangle \in EndSummary[\langle s_{calledProc(n)}, d_3 \rangle]$  do
15.3           foreach  $d_5 \in returnVal(\langle e_p, d_4 \rangle, \langle n, d_2 \rangle)$  do
15.4             Propagate( $\langle s_p, d_1 \rangle \xrightarrow{n} \langle returnSite(n), d_5 \rangle$ ).
15.5           od
15.6         od
16       od
17       foreach  $d_3$  s.t.  $d_3 \in callFlow(\langle n, d_2 \rangle)$  do
18         Propagate( $\langle s_p, d_1 \rangle \xrightarrow{n} \langle returnSite(n), d_3 \rangle$ )
19       od
20     end case

```

Figure 4.5: Extended IFDS Algorithm without computing summary edges

```

21         case  $n \in e_p$  :
21.1         EndSummary [ $\langle s_p, d_1 \rangle$ ]  $\cup = \langle e_p, d_2 \rangle$ 
22         foreach  $\langle c, d_4 \rangle \in \text{Incoming}[\langle s_p, d_1 \rangle]$  do
23             foreach  $d_5 \in \text{returnVal}(\langle e_p, d_2 \rangle, \langle c, d_4 \rangle)$  do
24
25
26                 foreach  $d_3$  s.t.  $\langle s_{\text{procOf}(c)}, d_3 \rangle \rightarrow \langle c, d_4 \rangle \in \text{PathEdge}$  do
27                     Propagate( $\langle s_{\text{procOf}(c)}, d_3 \rangle \xrightarrow{c} \langle \text{returnSite}(c), d_5 \rangle$ )
28                 od
29             od
30         od
31     od
32 end case
33 case  $n \in (N_p - \text{Call}_p - \{e_p\})$  :
34     foreach  $m, d_3$  s.t.  $n \rightarrow m \in \text{CFG}$  and  $d_3 \in \text{flow}(\langle n, d_2 \rangle, \pi)$  do
35         Propagate( $\langle s_p, d_1 \rangle \xrightarrow{n} \langle m, d_3 \rangle$ )
36     od
37 end case
38 end switch
39 od
end

```

Figure 4.5: Extended IFDS Algorithm without computing summary edges (contd.)

of Figure 4.4 is removed. Once these two changes are made, the SummaryEdge table is never queried, so the table can be removed from the algorithm. Therefore, the creation of SummaryEdges at Lines 24, 25 and 29 is also removed. To summarize, when an exit node of a procedure p is processed, path edges from the start nodes of all currently known callers of p to the return site of the call to p are computed. Similarly, when a call node within a procedure p is processed then, apart from processing the callee, path edges are produced, directly from the start nodes in p to the successor of the call, using all known exit nodes of the callee.

4.3.2 Empirical Evaluation

We have performed an empirical evaluation, using the variable type analysis, to answer the question: How large is the supergraph, and what fraction of it is reachable along valid paths?

We implemented the extended IFDS algorithm and the example type analysis in Scala [51] using Soot [67] as a front-end to parse and convert Java classes into 3-address code and construct a control flow graph (CFG). Both normal Java control flow and control flow due to exceptions was represented by edges in the CFG. We ran the extended algorithm on the DaCapo Benchmark Suite, version 2006-10-MR2 [10]. Since most of the benchmarks use reflection, we provided Soot with summaries of uses of reflection obtained by instrumenting the benchmarks using ProBe [45] and *J [27].³ Statistics about the benchmarks are presented in Table 4.1. The Methods column shows the number of methods in the part of the call graph analyzed by the IFDS analysis; since the type analysis does not analyze the library, we cut off the call graph at any call into the library.⁴ The Variables column shows the number of SSA variables in the analyzed methods. The Instructions column shows the number of instructions after conversion to the intermediate representation presented in Section 4.2. The Possible Types column shows the number of concrete classes in the benchmark. These are the classes that could appear as the type associated with a variable in the analysis results.

We first measured the size of the complete exploded supergraph. In general, the number of nodes in the exploded graph is given by $|\text{Inst}| \times (|\mathbf{Dom}| + 1)$ where $\mathbf{Dom} = \mathbf{Var} \times \text{Class}$, \mathbf{Var} is the set of all variables in the program and Class is the set of all classes. However, when analyzing a given method, only the local variables of that method need to be tracked. Thus a much smaller exploded supergraph can be constructed of size $\sum_{m \in \text{Methods}} |\mathbf{Var}_m| \times |\text{Class}| \times |\text{Inst}_m|$, where $|\mathbf{Var}_m|$ and $|\text{Inst}_m|$ are the numbers of variables and instructions in method m . We measured the size of this smaller, more reasonable exploded supergraph. In addition to the number of nodes, we computed the number of edges in the exploded supergraph. To do this, we applied the flow function to every node of the exploded supergraph and counted the number of outgoing edges. The sizes of the exploded supergraph are shown using squares in Figure 4.6. The sizes range from 138 million to 21 billion nodes. On average (geometric mean), each exploded supergraph has 1.16 times as many edges as nodes. The largest exploded supergraphs took over 24 hours to

³We excluded the Eclipse benchmark because it makes such heavy use of reflection that Soot is unable to process it.

⁴Not analyzing the library is a characteristic of our example analysis, and not a limitation of the IFDS algorithm in general.

Benchmark	Methods	Variables	Instructions	Possible Types
antlr	949	10839	16621	257
bloat	3142	33727	46550	623
chart	9419	91280	129850	2292
fop	13556	131901	185129	3400
hsqldb	768	8004	11552	443
python	5487	56090	74031	1079
luindex	1306	12519	18131	617
lusearch	1633	14850	21368	676
pmd	3643	33945	49640	998
xalan	786	7708	11084	451

Table 4.1: Benchmark Characteristics

enumerate. This is another reason why we did not use the alias set analysis abstraction as an example to illustrate our extensions to IFDS. Enumerating the exploded supergraph for that abstraction would take an incredibly long time, considering the larger domain **Dom** for that abstraction.

We also measured the sizes of the reachable part of the supergraph that is explored when the IFDS algorithm has been extended with demand supergraph construction. These sizes are shown as diamond shapes in Figure 4.6. The number of edges in the reachable part of the supergraph is 1.09 times the number of nodes. On average (geometric mean), the complete supergraph contains 2081 times as many nodes as the reachable part of the supergraph. Constructing the supergraph on demand rather than exhaustively is key to analyzing benchmarks of this size in reasonable time and memory bounds.

4.4 Return Flow Functions

In the original IFDS algorithm, the return flow function is modelled by interprocedural edges in the exploded supergraph that lead from the exit of a procedure to the call site that called the procedure. In the callee, each flow fact is represented in terms of the local scope of the callee. For many analyses, it is necessary to map information in the callee back to the caller. For example, for the code in Figure 4.7(a), the cast inside `ensureCircle` succeeds only if the object pointed to by `z`, which is also pointed to by `x` and `y`, is of type `Circle` or its subtype. Therefore, if `ensureCircle` returns normally, we know that `x` cannot point to

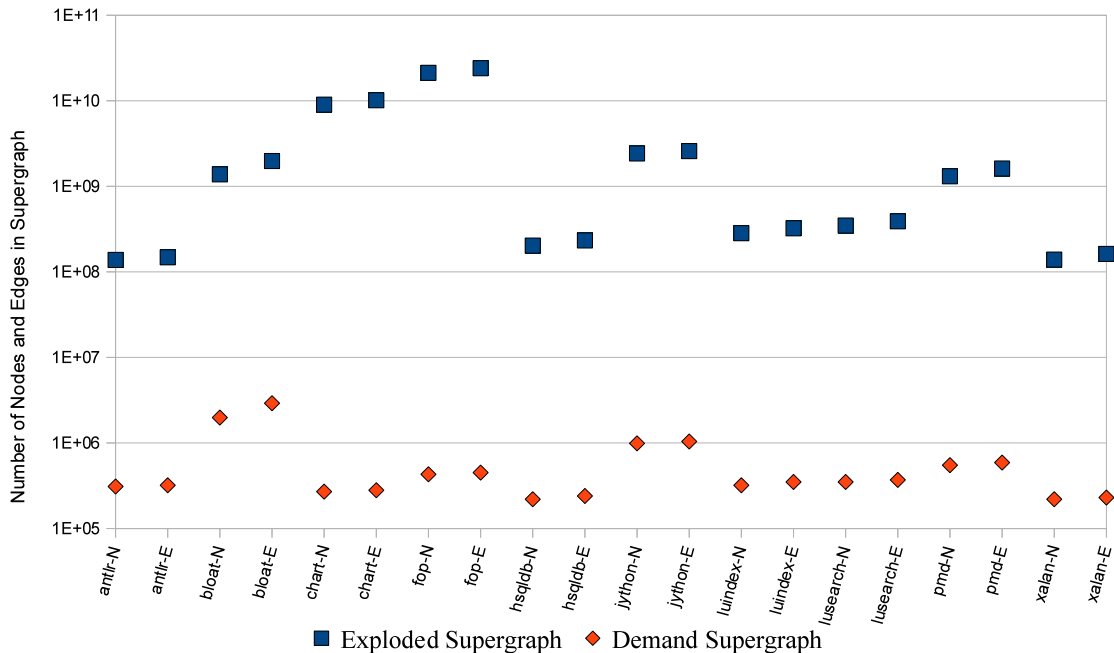


Figure 4.6: Number of nodes and edges in the exploded supergraph and its reachable subgraph. The letters N and E after each benchmark designate nodes and edges, respectively.

an arbitrary `Shape`, but only to a `Circle`. However, the original IFDS algorithm cannot discover this fact: although it determines that at the exit of `ensureCircle`, `z` points to an object of type `Circle`, there is no way in the supergraph to associate `z` in the callee with `x` in the caller.

Yet with a small extension, this reverse mapping can be recovered. The fact that `z` points to a subtype of `Circle` is expressed by the edge $\langle s_{\text{ensureCircle}}, \langle y, \text{Shape} \rangle \rangle \rightarrow \langle e_{\text{ensureCircle}}, \langle z, \text{Circle} \rangle \rangle$ in `PathEdge`. In Figure 4.7(b), which shows the demand supergraph constructed by the algorithm, this edge is labeled e_1 . This edge means that at the beginning of the procedure, there was an object pointed to by `y`, and at the exit of the procedure, the same object is pointed to by `z` and we know it is of type `Circle`. In addition, `Incoming`[$\langle s_{\text{ensureCircle}}, \langle y, \text{Shape} \rangle \rangle$] contains $\langle c, \langle x, \text{Shape} \rangle \rangle$. This means that the object passed in through `y` from the call site `c` was pointed to by `x` in the caller scope. This is represented by the edge e_2 in the figure. We can combine the context information

provided by Incoming with the intraprocedural information computed in PathEdge to determine that the object pointed to by x at the call site is known to be of type `Circle` after the call. In the figure, this is represented by the edge labeled e_3 .

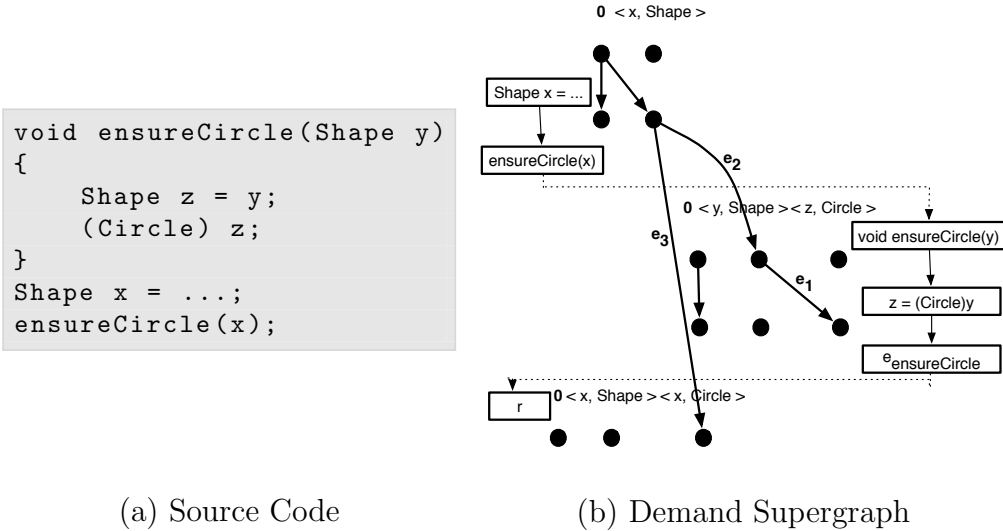


Figure 4.7: Example: mapping information from caller back to caller

This extension appears in the extended algorithm in Figure 4.5 on line 23. The `returnVal` function takes, in addition to the node d_2 at the exit instruction e_p , a second node d_4 at the call site c . These arguments indicate not only that the node d_2 is reachable at e_p , but that it is reachable from some node d_1 at the start instruction s_p of the procedure, and that a `passArgs` edge leads to the latter node from node d_4 at the call site c . Thus the `returnVal` function can use the caller-side state from the time the procedure was invoked to map the callee-side state at the exit of the procedure back to the caller-side context.

This extension is not merely an extension of the IFDS algorithm, but an extension of the exploded supergraph abstraction that the algorithm is based on. In the supergraph, for every pair of nodes d_2 at an exit node and d_5 at a return site, there either is or is not an edge from d_2 to d_5 ; if there is such an edge, the algorithm adds a `SummaryEdge` from $\langle c, d_4 \rangle$ to $\langle returnSite(c), d_5 \rangle$ for every call site c calling the procedure and for every reachable node d_4 at c . However, the extended algorithm gives the analysis designer more flexibility, in that the decision to add the `SummaryEdge` is additionally dependent on the specific call-site node $\langle c, d_4 \rangle$ being considered. It is as if the supergraph edge $\langle e_p, d_2 \rangle \rightarrow \langle returnSite(c), d_5 \rangle$ can both exist and not exist, depending on which call site node $\langle c, d_4 \rangle$ is being taken on the path used to reach $\langle e_p, d_2 \rangle$.

4.5 Static Single Assignment (SSA) Form

Static Single Assignment (SSA) form [20] is a popular intermediate representation that makes many program analyses simpler and more efficient. Standard dataflow analysis algorithms such as the original IFDS can be applied unchanged to programs in SSA form, but without appropriate extensions, such an analysis may be less precise than when the same analysis is done on the original, non-SSA version of the program. In this section, we discuss the reasons for the precision loss and propose an extension to the IFDS algorithm that fully restores the lost precision. The extended algorithm analyzes a program in SSA form as precisely as in its original, non-SSA form.

The defining feature of SSA form is that every variable is written to in only one instruction in the program. To convert a program to SSA form, every variable is renamed at each of its definitions, so each definition writes to a fresh, unique variable. Every use of a variable must also be renamed to match the reaching definition. A problem arises when multiple definitions reach a use: to which of the new names should the variable at the use be renamed? The solution is to add ϕ pseudo-instructions to select the reaching definition based on the control flow path taken. A ϕ instruction at a control flow merge point defines a new variable whose value is selected from among the reaching definitions depending on the edge taken into the merge point. Thus only the ϕ definition of the variable reaches the instructions following the merge.

The ϕ pseudo-instruction differs from normal instructions in two ways. First, if multiple variables require ϕ assignments at a given merge point, the ϕ assignments are performed simultaneously, in parallel. The set of ϕ instructions at the merge point defines, for each incoming control-flow edge, a permutation of the variables. Thus it is clearer to group all of the ϕ instructions at a given merge point into a single multi-variable ϕ instruction. Multiple instructions in sequence would suggest that the operations are performed one after the other, which is an incorrect semantics for ϕ instructions.

Second, unlike other instructions, the effect of a ϕ depends on the control-flow edge taken to reach the instruction. This causes many dataflow analysis algorithms, including the original IFDS, to lose precision when analyzing a program in SSA form, compared to analyzing the same program in its original non-SSA form. We will present an example program that exhibits such precision loss in Section 4.5.1. In most dataflow analyses, at a control flow merge point, the analysis first merges the dataflow facts from the incoming edges, then passes the merged value to the flow function of the instruction after the merge (i.e. $out[s] = f_s(\bigcup_{p \in pred(s)} out[p])$). Merging before applying the flow function reflects the structure of the control flow graph, and is appropriate when the merge is followed by a

non- ϕ instruction. When the merge is followed by a ϕ instruction, however, the merge preceding the flow function application makes it impossible for the flow function f_s to depend on the control flow predecessor that its input came from, since the inputs from all the predecessors have been merged into a single dataflow value. Most dataflow analyses treat a ϕ instruction such as $x_3 = \phi(x_1, x_2)$ as an assignment from both x_1 and x_2 to x_3 , ignoring the control flow edges on which those values of x_1 and x_2 arrived.

To analyze SSA-form code as precisely as non-SSA-form code, the merge must be delayed until *after* the ϕ instruction. That is, the ϕ flow function is applied separately to the dataflow value on each incoming control flow path, and the merge is performed on the *outputs* of the ϕ flow function, not on its input. As a result, the incoming control flow edge associated with each dataflow value can be made available to the flow function f_ϕ modelling the ϕ instruction. Formally, $out[\phi] = \bigcup_{p \in pred(\phi)} f_\phi(p, out[p])$.

Extending the IFDS algorithm to perform dataflow merges after ϕ instructions instead of before them requires two modifications. First, every edge added to PathEdge is annotated with a control flow predecessor. The edge $\langle s_p, d_1 \rangle \xrightarrow{n} \langle m, d_2 \rangle$ indicates that there is a path in the supergraph starting at the dataflow fact d_1 at the start node s_p , leading to the dataflow fact d_2 at node m , and that the second-last node on the path is at node n . In other words, the dataflow fact d_2 reaches m along the incoming control flow edge from n . Two PathEdge edges that differ only in the control flow predecessor are considered to be distinct. The PathEdge edges created in lines 18, 27, and 35 of the algorithm are annotated with the control flow predecessor, shown above the arrow. The PathEdge edge created in line 15 corresponds to the empty path from $\langle s, d_3 \rangle$ to itself, so there is no control flow predecessor to record. We therefore use a dummy predecessor, which we write as 0. However, the target of this edge is the start node of the procedure, which is never a ϕ instruction, so the predecessor will never be needed for this node.

Second, the flow function is extended with a second parameter, and when the function is called in line 34, the control-flow predecessor π of the PathEdge edge currently being processed is passed in. Thus the flow function for the ϕ instruction can depend on the control-flow predecessor π associated with the dataflow value d_2 reaching n .

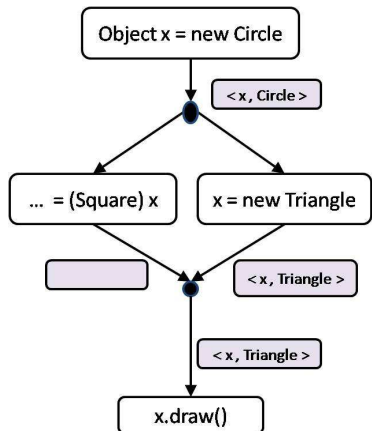
An obvious optimization is to annotate only those edges $\langle s_p, d_1 \rangle \rightarrow \langle m, d_2 \rangle$ in which m is a ϕ instruction, and leave all other edges unannotated. We do this in our implementation, but have not shown it in Figure 4.5 to avoid cluttering the algorithm.

```

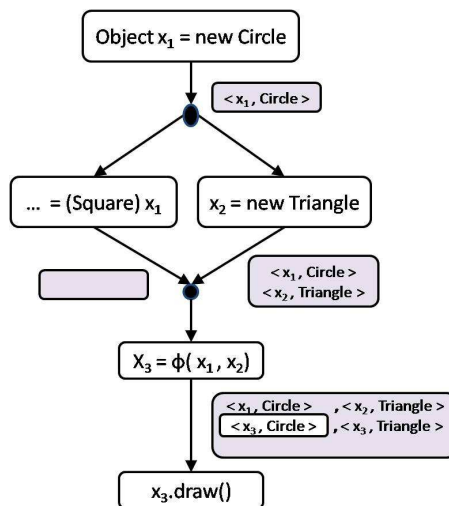
Object x = new Circle
if (cond) ... = (Square) x;
else x = new Triangle;
x.draw();

```

(a) Original Source Code



(b) Non-SSA Type Results



(c) SSA Type Results

Figure 4.8: The effect on precision due to the choice of merge strategy at ϕ nodes.

4.5.1 Example of precision loss

An example of how merging dataflow information before rather than after a ϕ instruction reduces precision is shown in Figure 4.8. The original non-SSA source code of the example program is in Figure 4.8(a). A variable x is initialized as a `Circle`. In the left branch of the conditional, x is cast to `Square`. In the right branch, x is redefined as a `Triangle`. Figure 4.8(b) shows the results of running the type analysis on the code. The flow function for the cast operation kills the flow fact $\langle x, \text{Circle} \rangle$, since a `Circle` cannot be successfully cast to a `Square`. Therefore, the type analysis indicates that the only possible type for receiver x at instruction `x.draw()` is `Triangle`. This is sound since the cast operation can never succeed and therefore a program executing the left branch can never reach the `draw` call. Conversely, if the program reaches the `draw` call it must have taken the right branch and the receiver must be a `Triangle`.

Figure 4.8(c) shows the same code after SSA conversion. The receiver x_3 for the call `x3.draw()` is x_1 when the path follows the left branch and x_2 when the path follows the right branch, as reflected in the ϕ function. The left predecessor of the ϕ function has

no flow facts because the cast kills $\langle x_1, \text{Circle} \rangle$ as before. The right predecessor has the facts $\langle x_1, \text{Circle} \rangle$ and $\langle x_2, \text{Triangle} \rangle$. The original IFDS algorithm would first merge the incoming flow facts from the two branches, then apply the flow function that models the ϕ as a copy from both x_1 and x_2 . At the call to $x_3.\text{draw}()$, the analysis would compute the facts $\langle x_3, \text{Circle} \rangle$ and $\langle x_3, \text{Triangle} \rangle$, which is less precise than the non-SSA version of the analysis that was able to rule out x being a **Circle**.

In the extended IFDS algorithm, the merge is not performed before the flow function of the ϕ instruction, so the flow function has information about the control flow edge on which each dataflow fact arrives. For facts coming in from the left edge, it models a copy from x_1 to x_3 ; for facts coming in from the right edge, it models a copy from x_2 to x_3 . Thus only the fact $\langle x_2, \text{Triangle} \rangle$ coming from the right edge leads to a new fact $\langle x_3, \text{Triangle} \rangle$. The fact $\langle x_1, \text{Circle} \rangle$ does not give rise to $\langle x_3, \text{Circle} \rangle$, as it did before, because it comes in from the right edge, which is not associated with a copy from x_1 to x_3 . Thus the extended IFDS algorithm achieves the same precision on the SSA-form version of the program as on the original non-SSA-form version.

4.6 Exploiting Structure in the Set Dom

The IFDS algorithm requires that the dataflow domain be the powerset of a finite set **Dom**. The elements of **Dom** are treated independently and equally. The algorithm does not assume or take advantage of any relationships between the elements of **Dom**. This is appropriate for bit-vector dataflow problems. For example, the liveness of variable x at some program point implies nothing about the liveness of a different variable y .

However, some domains have more structure in the form of subsumption relationships between elements. In the example type analysis, the fact $\langle x, \text{Circle} \rangle$ subsumes the fact $\langle x, \text{Shape} \rangle$, since knowing that x points to an object whose type is some subtype of **Circle** implies that its type is also a subtype of **Shape**. Therefore, if the analysis computes, for some program point, the set $\{\langle x, \text{Circle} \rangle, \langle x, \text{Shape} \rangle\}$, which means that x points to a subtype of **Circle** or that x points to a subtype of **Shape**, then this set provides no additional information compared to the smaller set $\{\langle x, \text{Shape} \rangle\}$ that could have been computed; the two sets are equivalent.

Formally, we can define for an arbitrary analysis the partial order $a \leq b$, meaning that a subsumes b (for example, $\langle x, \text{Circle} \rangle \leq \langle x, \text{Shape} \rangle$). We require all of the dataflow functions to be monotone in the partial order: $a \leq b \implies \text{flow}(a) \leq \text{flow}(b)$. We consider two sets computed by the analysis to be equivalent, written $D_1 \sim D_2$, if every element of

each set is subsumed by some element of the other set:

$$\begin{aligned}
 D_1 \leq D_2 &\iff \forall d_1 \in D_1 \exists d_2 \in D_2 \text{ s.t. } d_1 \leq d_2 \\
 D_1 \sim D_2 &\iff D_1 \leq D_2 \wedge D_2 \leq D_1
 \end{aligned}$$

The original IFDS algorithm handles analyses in which **Dom** has structure correctly but not as efficiently as possible. Because it ignores the subsumption relationship, it compute $\{\langle x, \text{Circle} \rangle, \langle x, \text{Shape} \rangle\}$ instead of the equivalent smaller set $\{\langle x, \text{Shape} \rangle\}$. We have extended the algorithm to use subsumption relationships in **Dom** to find smaller equivalent sets. The extension reduces the size not only of the final result, but of the intermediate sets during execution of the algorithm. The performance improvement is cumulative since smaller intermediate sets require less further processing.

The extended algorithm is as precise as the original IFDS algorithm in the sense that if the algorithms compute dataflow facts \mathbf{Dom}_{ext} and \mathbf{Dom}_{orig} , respectively, for a given program point, then $\mathbf{Dom}_{ext} \sim \mathbf{Dom}_{orig}$.

Extending the algorithm to exploit subsumption requires two steps. First, the Propagate function is changed to only add an edge to PathEdge if it does not subsume any already existing edge, as shown in Figure 4.9.⁵ Any edges in PathEdge and in the WorkList subsuming the newly-added edge are redundant and can be removed in line 9.1. Removing a subsuming edge from the WorkList would be an expensive operation. Therefore, in our implementation, we remove subsuming edges only from PathEdge. Since PathEdge is a set, this can be achieved efficiently. Since a subsuming edge is not removed from the WorkList, at some point in the algorithm the edge will be dequeued (Line 11 in the IFDS algorithm). At this point, our implementation checks if this edge is still in PathEdge and ignores the edge if it is not found. This ensures that the algorithm does not process any edge that has been removed from PathEdge without having to remove such edges from the WorkList. We have not presented these implementation details in the algorithm so as to not clutter the algorithm.

Second, the worklist is modified so that subsumed elements are processed before subsuming ones. Without an appropriate worklist ordering, the algorithm might do the work of constructing the full sets and only afterwards discover an element that the existing elements subsume, making the existing elements unnecessary. Thus only after all of the work was done would the algorithm discover that the work was not necessary.

⁵Though it may seem counterintuitive, it is correct to only add elements that do not subsume an existing element, rather than elements not themselves subsumed by an existing element. The interpretation of the PathEdge set is a disjunction of the possible types for each variable: any element in the set is a possible abstraction of runtime behaviour. If a subsumes b , then adding a to a disjunction already containing b does not change the meaning of the disjunction.

```

procedure Propagate( $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ )
begin
9   if  $\exists \langle s_p, d_1 \rangle \rightarrow \langle n, d_3 \rangle \in \text{PathEdge}$  s.t.  $d_2 \leq d_3$  then
      Insert  $e$  into PathEdge; Insert  $e$  into WorkList;
      fi
9.1 Remove all edges  $\langle s_p, d_1 \rangle \rightarrow \langle n, d_3 \rangle$  s.t.  $d_3 \leq d_2$  from PathEdge
      and from WorkList
end

```

Figure 4.9: Extended Propagate Procedure

To define a suitable worklist ordering, we define an estimate function mapping each element of **Dom** to an integer with the property that $d_1 \leq d_2 \implies \text{estimate}(d_1) \leq \text{estimate}(d_2)$. For all analyses we have encountered, we have found it easy to define such an estimate. For the example type analysis, we use the following estimate: the class **Object** has estimate 0, and the estimate of each other class is one less than the estimate of its superclass. For a given estimate function, the worklist is implemented as a priority queue that makes the algorithm process edges with the highest estimate first.

This ordering heuristic does not completely guarantee that the algorithm will never call Propagate with an edge that makes a previous edge unnecessary, but it does ensure this property in most cases, and works well in practice. Recall that each flow function is monotonic, so that $a \leq b \implies \text{flow}(a) \leq \text{flow}(b)$. We can be sure to compute $\text{flow}(b)$ and $\text{flow}(a)$ in the correct order (that is, $\text{flow}(b)$ first) by following the ordering heuristic to remove b from the worklist before a . However, at a control flow merge point, it is possible that a and b appear at two different control flow predecessors p, p' , which are modelled by different flow functions. There is no guarantee that $a \leq b \implies \text{flow}_p(a) \leq \text{flow}_{p'}(b)$, so we cannot guarantee that it is more efficient to compute $\text{flow}_{p'}(b)$ before $\text{flow}_p(a)$.

4.6.1 Empirical Evaluation

Using the same experimental setup as in Section 4.3.2 we set out to answer the question:

How does taking advantage of subsumption relationships in **Dom** reduce the number of dataflow facts that must be processed and the running time of the IFDS algorithm?

We ran the type analysis three times. In the first run, the subsumption extension was turned off, so all dataflow facts were propagated regardless of their subsumption relation-

ships. In the second run, the subsumption extension was turned on, but the original first-in first-out (FIFO) worklist was used. In the third run, both the subsumption extension and the subsumption-aware worklist ordering were used. For each case, we measured the running time of the analysis and the total number of pairs $\langle v, t \rangle$ computed (i.e. the sum over all instructions of the number of $\langle v, t \rangle$ pairs for that instruction). The results are shown in Figure 4.10 both in tabular form and as bar charts. Empty cells in the table indicate that the analysis did not complete within 10000 seconds of CPU time and 10 GB of memory. These are marked as DNT on the bar charts indicating that the analysis Did Not Terminate within the allotted resources. The subsumption-extended analysis completed on all of the benchmarks, but the unextended analysis completed on only five benchmarks within these time and memory limits. Columns 2 and 3 show the number of $\langle v, t \rangle$ pairs without and with the subsumption extension (this number is independent of the worklist ordering). The same result is shown as a bar chart in the bottom left part of the figure. Columns 4, 5, and 6 show the running time of the three runs of the analysis (also shown graphically in the bottom right corner of the figure). On the five benchmarks on which all algorithms ran to completion, the unextended analysis had to compute 6.3 times as many pairs as the extended analysis, so the unextended analysis took 55 times as long as the extended analysis (geometric mean). In the extended analysis, the subsumption-aware priority queue worklist reduced the running time by 10% (geometric mean over all benchmarks). Extremes were *jython*, where the reduction was 43%, and *antlr*, where the running time increased by 2% due to the higher cost of maintaining a priority queue compared to a FIFO list. The subsumption extension presented in this section is very important for the speed of the analysis and for its ability to analyze programs of significant size.

4.7 Related Work

Sharir and Puneli [62] extended Kildall’s framework of intraprocedural dataflow analysis [44, 43] to two frameworks of context-sensitive interprocedural dataflow analysis, which they called the *call-strings* approach and the *functional* approach. The two frameworks compute a merge-over-all-valid-paths solution, where a valid path is one in which procedure calls and returns are correctly matched. The call-strings approach treats calls and returns from a procedure like all other control flow but restricts propagation to valid paths by tagging propagated dataflow facts with a call string (an abstraction of the active call stack). In the functional approach, the effects of each procedure are summarized by a summary function $f_p : \mathbf{Dom} \rightarrow \mathbf{Dom}$, where \mathbf{Dom} is the dataflow analysis domain. The summary function is then used at each call site of the procedure to model the effect of

Benchmark	Facts ($\times 10^3$)		Time (s)		
	w/o subs.	w subs.	w/o subs.	w subs., w/o PQ	w subs., w PQ
antlr	546	309	179	44	45
bloat		2037		1544	1518
chart		2817		3377	3197
fop		4408		3247	2847
hsqldb	1758	224	4720	60	60
kython		1015		1225	697
luindex	2900	326	9860	75	70
lusearch	3432	356	9776	78	68
pmd		556		241	211
xalan	1809	218	4813	61	60

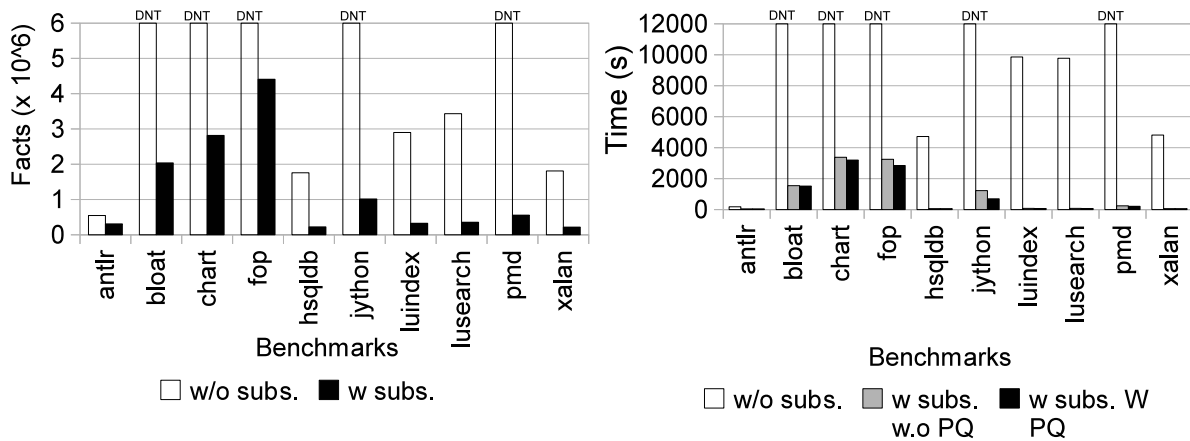


Figure 4.10: Effect of taking advantage of subsumption relationships in D .

the call. The key operation in the functional approach is function composition. For example, to compute the summary function f_r of a caller procedure that contains a call site to a callee procedure, the summary function f_e of the callee procedure must be composed with functions representing the intraprocedural effects of the caller procedure. Although the functional approach has the potential to be more precise and more efficient than the call strings approach, a key challenge is devising efficient representations of the summary functions that are amenable to function composition.

The IFDS framework [54] provides such an efficient representation of summary functions for the functional approach, as discussed in Section 4.1. When the dataflow domain is $\mathcal{P}(\mathbf{Dom})$ for a finite set \mathbf{Dom} and all of the dataflow functions are distributive, they can be compactly represented using bipartite graphs with $O(\mathbf{Dom})$ nodes. Function composition can be computed efficiently in this representation, and the composition of distributive functions is also distributive. Thus the IFDS algorithm makes the functional approach practical for the class of dataflow analyses satisfying these restrictions. The IFDS algorithm has been used to solve both locally separable problems such as reaching definitions, available expressions and live variables, and non-locally-separable problems such as uninitialized variables and copy-constant propagation.

The IDE [60] algorithm⁶ generalizes IFDS to a wider class of dataflow analyses. Whereas in IFDS, the dataflow facts are elements of $\mathcal{P}(\mathbf{Dom})$, the IDE algorithm allows dataflow facts that are maps drawn from $\mathbf{Dom} \rightarrow L$, where \mathbf{Dom} is a finite set and L is a finite-height semi-lattice.⁷ The IDE algorithm has been used to express copy-constant propagation and linear constant propagation [60]. The IDE literature calls elements of $\mathbf{Dom} \rightarrow L$ environments, so the flow functions that are composed in the algorithm are environment transformers drawn from $(\mathbf{Dom} \rightarrow L) \rightarrow (\mathbf{Dom} \rightarrow L)$. Provided these transformers are distributive, they can be represented efficiently using graphs similar to those used in the IFDS algorithm, with additional labels on the edges of the graph describing the effect of the edge on elements of L . Whereas the IFDS problem computes reachability along valid paths, the IDE algorithm additionally evaluates functions $L \rightarrow L$ along those paths. The overall structure of both algorithms is very similar, however. All of the extensions presented in this chapter are equally applicable to the IDE algorithm as well as to the IFDS algorithm. We have implemented the extensions in both algorithms.

Demand-driven variations of the IFDS and IDE algorithms have been thoroughly studied [55, 39, 60, 25, 26]. These algorithms differ from the exhaustive algorithms in that

⁶We use the IDE algorithm in Section 5.1 to compile a list of all transition statements that may contribute to a match at each body statement.

⁷The domain $\mathcal{P}(\mathbf{Dom})$ is isomorphic to $\mathbf{Dom} \rightarrow L$ if L is chosen to be the two-point lattice.

rather than computing all nodes reachable from the start node, they determine whether a given node n is reachable. These algorithms can be faster when only a small number of nodes are queried. The algorithms work by exploring reverse paths along the supergraph from the given node n , by evaluating inverses of the dataflow functions. The demand-driven computation of reachability implemented by these algorithms is distinct from and complementary to the demand-driven exploded supergraph construction that we presented in Section 4.3. The purpose of demand supergraph construction is to avoid constructing the whole supergraph, which may be much larger than its reachable subgraph; the demand-driven reachability algorithms do require the whole exploded supergraph to be constructed ahead of time. Although our extended IFDS algorithm constructs the exploded supergraph on demand, it then exhaustively computes all nodes reachable along valid paths, rather than answering reachability queries for specific nodes. An interesting direction for future work would be to combine demand supergraph construction with demand-driven reachability queries. Such an algorithm appears to be challenging to design and to tune, however. The key difficulty that we had to overcome in constructing the exploded supergraph on demand was the need, on line 23 of the original IFDS algorithm, to evaluate the inverse of the dataflow function. The demand-driven supergraph reachability algorithms require much more evaluation of inverse dataflow functions.

Others have noticed limitations of the original IFDS algorithm, and mention implementing extensions similar to some of those that we have presented here [29, 30, 56, 63, 69]. Fink et al. [29, 30] used the IFDS algorithm to verify tpestate properties of objects. To verify that an object respects a temporal property, they build precise abstractions of the objects in the program and aliasing between them. The analysis computes an object abstraction containing sets of access paths that must or must-not reference an object. This abstraction is computed using the IFDS algorithm with extensions for exceptional control flow and polymorphic dispatch. Though their presentation focuses on the tpestate analysis rather than specifics of their extensions to the IFDS algorithm, their implementation depends on constructing the exploded supergraph on demand, providing call-site information to return flow functions, and exploiting subsumption between elements of D . Shoham et al. [63] apply the infrastructure of Fink et al. [29, 30], along with its IFDS extensions, to statically extract finite-state automata of sequences of API calls.

Some shape analyses that have been implemented as instances of the IFDS algorithm construct the supergraph on demand for scalability. Rinetzky et al. [56] present an efficient shape analysis for the class of cutpoint-free programs, in which at each procedure call, the subgraph of the heap reachable in the callee can only be reached in the caller through arguments of the call. Yang et al. [69] present a different shape analysis that works for general programs. Both of these analyses are instances of the IFDS algorithm, and both

implementations construct only the reachable part of the supergraph.

Several of the analyses just mentioned [29, 30, 63, 56, 69] use partial joins, an extension similar to subsumption in the analysis domain D that we discussed in Section 4.6. Whereas a partial join enables the analysis designer to sacrifice precision for efficiency, exploiting subsumption does not change analysis precision. A partial join may make the analysis output depend on the order of exploration; exploiting subsumption does not. A partial join operator $\dot{\sqcup}$ is a partial function $\dot{\sqcup} : D \times D \dashrightarrow D$ with the property that if $a \dot{\sqcup} b = d$, then each of a and b subsume d . Whenever the partial join IFDS algorithm encounters both a and b in a given set, it replaces them with d , reducing the size of the set. This operation is sound, since if each of a and b subsume d , then so does their disjunction. However, it may reduce precision. For example, if we also define $a \dot{\sqcup} c = d$, it becomes impossible for the analysis to distinguish $\{a, b\}$ from $\{a, c\}$, even though neither set subsumes the other (i.e. $\{a, b\} \not\prec \{a, c\}$). Our subsumption extension can be implemented using the following definition of a partial join: if $a \leq b$, then $a \dot{\sqcup} b = b \dot{\sqcup} a = b$, else $a \dot{\sqcup} b$ is undefined.

4.8 Using the Extended IFDS Algorithm for Analyzing Tracematches

The extensions to the original IFDS algorithm that we presented in this chapter were born out of necessity. In this section, we discuss how each extension relates to the tracematch analysis. The following chapter discusses the implementation in more detail.

In the original IFDS algorithm, the exploded supergraph contains $|\text{Inst}| \times (|\mathbf{Dom}| + 1)$ nodes. For the object abstraction, the domain is the set of sets of variables. This makes the exploded supergraph extremely expensive to compute, and the overall analysis infeasible on any program of reasonable size. Our extension to the algorithm, to only compute the reachable subset of the supergraph, alleviates this restriction and allows us to use the IFDS algorithm for computing the object abstraction on reasonably sized programs.

Our second extension extends the return flow function by making available, to the end nodes of the callee, caller-side information at the time of the procedure call. The tracematch analysis relies on this extension. An object is abstracted by the set of local variables that point to it. Therefore, within a callee, all variables contained in an abstract object are local to the callee. At end nodes, the analysis must map abstract objects, containing variables local to the callee, to abstract objects that contain variables local to the caller. In the unextended IFDS algorithm, the return flow function is defined only in terms of the facts computed for the end nodes of the callee and is not sufficient to perform

this mapping. Our extension makes the mapping possible by providing the function that mapped caller-side variables to callee-side variables at the time of the call.

As discussed earlier, correctly handling ϕ nodes within a SSA-based representation is important to ensure that the analysis does not lose precision compared to their non-SSA counterparts. Since the tracematch analysis is performed on an intermediate representation that is in SSA form, it benefits from the semantically correct handling of ϕ nodes.

The last extension deals with exploiting the structure in the analysis domain to improve performance by computing a small final result and intermediate sets. This is achieved by only adding an edge to PathEdge if it does not subsume any already existing edge. Such a subsumption relationship is easy to define for the object abstraction. An abstract object o_1 subsumes an abstract object o_2 ($o_1 \leq o_2$) if o_2 is a subset of o_1 . The object abstraction implementation discussed in the following chapter uses this subsumption relationship between abstract objects to improve the performance of the analysis.

4.9 Concluding Remarks

In this chapter, we presented four extensions to the IFDS algorithm that make it applicable to a wider class of interprocedural dataflow analysis problems, in particular analyses of objects and pointers. The extended algorithm does not require an exploded supergraph as input, but builds it on demand, only for those dataflow facts for which it is actually needed. The extended algorithm provides caller-side context information from before a procedure call to the flow function that maps callee-side state back to the caller after the call. The extended algorithm analyzes programs in SSA form as precisely as programs not in SSA form. The extended algorithm takes advantage of structure in the dataflow analysis domain to significantly speed up analyses exhibiting such structure. We illustrated our extensions on a variation of variable type analysis. In the next chapter, we leverage the extended IFDS analysis to implement the object and state abstractions we discussed in Chapter 3.

Chapter 5

Implementation

To formulate the tracematch analysis as an IFDS problem, we must define the set **Dom** and the transfer functions on individual elements of **Dom**. This cannot be done for the overall flow function, $\lambda \rho^\#, h^\#, \sigma^\#. \langle \llbracket s \rrbracket_{\rho h^\#}(\rho^\#, h^\#), \llbracket s \rrbracket_{\sigma^\#}[\rho^\#](\sigma^\#) \rangle$, that computes both the object and tracematch abstractions because it is not distributive. This is because the tracematch state depends on abstractions of multiple objects, which could come from different control flow paths. Individually, however, each of the transfer functions for the object abstraction and for the tracematch state abstraction is distributive. Thus, we can first perform the object analysis as one instance of IFDS, then use the result to perform the tracematch state analysis as a second instance of IFDS. Moreover, the decomposition into transfer functions on individual elements of a finite set **Dom** comes naturally from the definition of the overall transfer functions. For the object abstraction, **Dom** is two copies of the set of all possible abstract objects, one copy to represent each of $\rho^\#$ and $h^\#$. Thus, the decomposed transfer function specifies the effect of an instruction on a single abstract object at a time. For the tracematch state abstraction, **Dom** is the set of all possible pairs $\langle q, m^\# \rangle$. Thus, the decomposed transfer function specifies the effect of an instruction on one pair at a time.

Let us now formally define the decomposed transfer functions. As mentioned earlier, $\rho^\#$ and $h^\#$ are both defined as $\mathbf{Obj}^\# = \mathcal{P}(\mathbf{Var})$. Therefore, the value abstraction $\langle \rho^\#, h^\# \rangle$ is defined as two sets of $\mathbf{Obj}^\#$. To distinguish elements of the two sets, we use the notation $\rho[o^\#]$ to mean $o^\#$ from $\rho^\#$, and $h[o^\#]$ to mean $o^\#$ from $h^\#$. Thus, a given pair $\langle \rho^\#, h^\# \rangle$ is represented using the set $decomp(\rho^\#, h^\#) \triangleq \{\rho[o^\#] : o^\# \in \rho^\#\} \cup \{h[o^\#] : o^\# \in h^\#\}$. The transfer

function for individual elements of $\mathbf{Dom} \cup \{0\}$ is defined as follows:

$$\begin{aligned} \llbracket s \rrbracket_{\rho h^\#}(\rho[o^\#]) &\triangleq \begin{cases} \{\rho[o^\# \setminus \{v\}]\} & \text{if } s = v \leftarrow e \\ \{\rho[o^\#], h[o^\#]\} & \text{if } s = e \leftarrow v \text{ and } v \in o^\# \\ \{\rho[o^\#]\} & \text{if } s = e \leftarrow v \text{ and } v \notin o^\# \\ \{\rho[\llbracket s \rrbracket_{o^\#}(o^\#)]\} & \text{otherwise} \end{cases} \\ \llbracket s \rrbracket_{\rho h^\#}(h[o^\#]) &\triangleq \begin{cases} \{\rho[o^\# \setminus \{v\}], \rho[o^\# \cup \{v\}], h[o^\# \setminus \{v\}], h[o^\# \cup \{v\}]\} & \text{if } s = v \leftarrow e \\ \{h[\llbracket s \rrbracket_{o^\#}(o^\#)]\} & \text{otherwise} \end{cases} \\ \llbracket s \rrbracket_{\rho h^\#}(0) &\triangleq \begin{cases} \{\rho[\{v\}]\} & \text{if } s = v \leftarrow \mathbf{new} \\ \emptyset & \text{otherwise} \end{cases} \end{aligned}$$

The transfer function above uses $\llbracket s \rrbracket_{o^\#}$ as defined earlier for the object abstraction in Figure 3.2. We reproduce it below for ease of reference:

$$\llbracket s \rrbracket_{o^\#}(o^\#) \triangleq \begin{cases} o^\# \cup \{v_1\} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \in o^\# \\ o^\# \setminus \{v_1\} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \notin o^\# \\ o^\# \setminus \{v\} & \text{if } s \in \{v \leftarrow \mathbf{null}, v \leftarrow \mathbf{new}\} \\ o^\# & \text{if } s \in \{e \leftarrow v, \mathbf{tr}(T), \mathbf{body}\} \\ \text{undefined} & \text{if } s = v \leftarrow e \end{cases}$$

The following proposition guarantees that when these pointwise transfer functions are composed, the result is isomorphic to the transfer function $\llbracket s \rrbracket_{\rho h^\#}$ from Section 3.2

Proposition 6.

$$\begin{aligned} \llbracket s \rrbracket_{\rho^\#}(\rho^\#, h^\#) &= \left\{ o^\# : \rho[o^\#] \in \bigcup_{d \in \text{decomp}(\rho^\#, h^\#) \cup \{0\}} \llbracket s \rrbracket_{\rho h^\#}(d) \right\} \\ \llbracket s \rrbracket_{h^\#}(\rho^\#, h^\#) &= \left\{ o^\# : h[o^\#] \in \bigcup_{d \in \text{decomp}(\rho^\#, h^\#) \cup \{0\}} \llbracket s \rrbracket_{\rho h^\#}(d) \right\} \end{aligned}$$

A proof of the proposition is presented in Appendix A.

Recall that the IFDS algorithm (Tabulate in Figure 4.4) expects the functions $flow(n^\#)$, $passArgs(n^\#)$, $returnVal(n_1^\#, n_2^\#)$ and $callFlow(n^\#)$ where $n^\#$ is a node in the ICFG and

has the form $\langle l, d \rangle$ with l being a label in the program and $d \in \mathbf{Dom} \cup \{0\}$. Using the decomposed transfer functions for the value abstraction we define $flow(\langle l, d \rangle)$ as: $flow(\langle l, d \rangle) \triangleq \llbracket l \rrbracket_{\rho h^\#}(d)$

The analysis for computing the tracematch abstraction operates on the set of possible tracematch state pairs $\mathbf{Dom} \triangleq Q \times (F \rightarrow \mathbf{Bind}^\#)$. The analysis uses the value abstraction $\rho^\#$ computed in an earlier pass. The tracematch transfer function from Section 3.3 is already in the decomposed form required by the IFDS algorithm:

$$\llbracket s \rrbracket_{\sigma^\#}[\rho^\#](\sigma^\#) \triangleq \bigcup_{\langle q, m^\# \rangle \in \sigma^\# \cup \{0\}} \llbracket s \rrbracket[\rho^\#]_{m^\#}(q, m^\#) \quad \text{where } \llbracket s \rrbracket[\rho^\#]_{m^\#}(0) \triangleq \llbracket s \rrbracket[\rho^\#]_{m^\#}(q_0, \lambda f. \top)$$

In addition, the IFDS algorithm requires functions describing the flow into (`passArgs`) and out of (`returnVal`) procedure calls. These flow functions are also decomposed into functions acting on individual elements of $\mathbf{Dom} \cup \{0\}$. The `passArgs` function for the object abstraction is straightforward to define. Within each variable set representing an abstract object, each argument is replaced with the corresponding parameter, and all other variables are removed.

Given a substitution r that maps each argument to its corresponding parameter, the function is defined as:

$$\begin{aligned} update_{o^\#}[r](o^\#) &\triangleq \{r(v) : v \in o^\# \cap \text{dom}(r)\} \\ call_{\rho h^\#}[r](\rho[o^\#]) &\triangleq \{\rho[update[r](o^\#)]\} \\ call_{\rho h^\#}[r](h[o^\#]) &\triangleq \{h[update[r](o^\#)]\} \\ call_{\rho h^\#}[r](0) &\triangleq \emptyset \\ passArgs(\langle l, d \rangle) &\triangleq call_{\rho h^\#}[r](d) \end{aligned}$$

Let us now define the flow out of procedure calls for the object abstraction. In the unextended IFDS algorithm (Figure 4.2), the return flow function is defined only in terms of the flow facts computed for the end node of the callee. The difficulty is that in the callee, each abstract object is represented by a set of variables local to the callee, and it is unknown which caller variables point to the object. However, the only place where the algorithm uses the return flow function is when computing a `SummaryEdge` flow function for a given call site by composing $return \circ \llbracket p \rrbracket \circ call$, where $call$ is the call flow function, $\llbracket p \rrbracket$ is the summarized flow function of the callee, and $return$ is the return flow function. The original formulation of the algorithm assumes a fixed return flow function $return$ for each

call site. In the extended algorithm we use a `returnVal` function that, given a call site and the computed flow function $\llbracket p \rrbracket \circ \text{call}$, directly constructs the `SummaryEdge` flow function.

This summary flow function is also specified pointwise. The pointwise function summ_\bullet takes two arguments $d, d' \in \mathbf{Dom} \cup \{0\}$. The overall summary function is defined as:

$$\text{summ}(D) \triangleq \bigcup_{d \in D \cup \{0\}} \bigcup_{d' \in (\llbracket p \rrbracket \circ \text{call})(d) \cup \{0\}} \text{summ}_\bullet(d, d')$$

Intuitively, d is the caller-side abstraction of an object existing before the call, d' is one possible callee-side abstraction of the same object at the return site, and $\text{summ}_\bullet(d, d')$ ought to yield the set of possible caller-side abstractions of the object after the call. An object newly created within the callee is handled by the case $d = 0$.

The summary flow function for the object abstraction is defined as follows, where v_s is the callee variable being returned and v_t is the caller variable to which the returned value is assigned. If the object that was represented by $o_c^\#$ in the caller before the call is being returned from the callee (i.e. $v_s \in o_r^\#$), then v_t is added to $o_c^\#$. If some other object is being returned, then v_t is removed from $o_c^\#$, since v_t gets overwritten by the return value. In the case of an object newly created within the callee, the empty set is substituted for $o_c^\#$, since no variables of the caller pointed to the object before the call.

$$\begin{aligned} rv(o_c^\#, o_r^\#) &\triangleq \begin{cases} o_c^\# & \text{if } p \text{ does not return a value} \\ o_c^\# \cup \{v_t\} & v_s \in o_r^\# \\ o_c^\# \setminus \{v_t\} & v_s \notin o_r^\# \end{cases} \\ \text{summ}_{\rho h^\#}(c_h^\rho[o_c^\#], r_h^\rho[o_r^\#]) &\triangleq \{r_h^\rho[rv(o_c^\#, o_r^\#)]\} \text{ where each of } c_h^\rho, r_h^\rho \text{ is either } \rho \text{ or } h \\ \text{summ}_{\rho h^\#}(0, r_h^\rho[o_r^\#]) &\triangleq \{r_h^\rho[rv(\emptyset, o_r^\#)]\} \\ \text{returnVal}(\langle e_p, d_1 \rangle, \langle n, d_2 \rangle) &\triangleq \text{summ}_{\rho h^\#}(d_2, d_1) \end{aligned}$$

The `passArgs` function for `tracematch` state applies the `update[r]` function that was defined for the object abstraction to each `must`, `may`, and `negative` binding set. Arguments are replaced by parameters, and non-arguments are removed.

$$\begin{aligned}
update_{d^\#}[r](\langle o^!, o^? \rangle) &\triangleq \langle update_{o^\#}[r](o^!), update_{o^\#}[r](o^?) \rangle \\
update_{d^\#}[r](\overline{V^\#}) &\triangleq \overline{update_{o^\#}[r](V^\#)} \\
call_{m^\#}[r](q, m^\#) &\triangleq \{ \langle q, \lambda f. update_{d^\#}[r](m^\#(f)) \rangle \} \\
call_{m^\#}[r](0) &\triangleq \emptyset \\
passArgs(\langle l, d \rangle) &\triangleq call_{m^\#}[r](d)
\end{aligned}$$

Next, we must define the returnVal function for the tracematch abstraction. We first define a function $rv_{d^\#}(b_c^\#, b_r^\#)$, where $b_c^\# \in \mathbf{Bind}^\#$ represents the binding for some tracematch parameter f at the call and $b_r^\# \in \mathbf{Bind}^\#$ represents the binding for the same parameter f at the return site. The function $rv_{d^\#}$ computes a binding for f after the return. Notice that both $b_c^\#$ and the resulting binding from $rv_{d^\#}$ contain variables local to the caller, whereas $b_r^\#$ contains variables local to the callee. The function $rv_{d^\#}$ is defined analogously to the function rv for the object abstraction. As before, we use v_s to specify the callee variable being returned and v_t for the caller variable to which the returned value is assigned.

Elements of $\mathbf{Bind}^\#$ are either a positive or negative binding. For the state abstraction, a positive binding abstracts a runtime object using a pair $\langle o^!, o^? \rangle$ of a must set $o^!$ and a may set $o^?$. A negative binding on the other hand is a set of variables that do not reference the object bound by the tracematch parameter. When both $b_c^\#$ and $b_r^\#$ represent abstract objects (positive bindings), if the object that was represented by o_c in the caller before the call is being returned from the callee ($v_s \in o_r^!$), then v_t points to the object after the call, i.e. v_t is added to both $o_c^!$ and $o_c^?$. If the object being returned might have been represented by o_c before the call, i.e. v_s is in $o_r^?$ but not in $o_r^!$, then we introduce the same uncertainty in o_c by adding v_t to $o_c^?$ and removing it from $o_c^!$. If some other object is being returned, then v_t is removed from both $o_c^!$ and $o_c^?$. Formally:

$$rv_{d^\#}(\langle o_c^!, o_c^? \rangle, \langle o_r^!, o_r^? \rangle) \triangleq \begin{cases} \langle o_c^!, o_c^? \rangle & \text{if } p \text{ does not return a value} \\ \langle o_c^! \cup \{v_t\}, o_c^? \cup \{v_t\} \rangle & \text{if } v_s \in o_r^! \\ \langle o_c^! \setminus \{v_t\}, o_c^? \cup \{v_t\} \rangle & \text{if } v_s \notin o_r^! \wedge v_s \in o_r^? \\ \langle o_c^! \setminus \{v_t\}, o_c^? \setminus \{v_t\} \rangle & \text{if } v_s \notin o_r^! \wedge v_s \notin o_r^? \end{cases}$$

If both $b_c^\#$ and $b_r^\#$ are negative bindings, then, $b_c^\#$ is a set $\overline{V_c^\#}$ that represents the set of caller side variables that reference objects not bound by f , and $b_r^\#$ is a set $\overline{V_r^\#}$ that represents the set of callee side variables that reference objects not bound by f . Therefore, if the

returned variable v_s is in $\overline{V_r^\sharp}$, v_s does not reference the bound object. Hence, after the return, the assigned variable v_t does not reference the bound object either; v_t is added to the caller-side negative binding. Alternately, if v_s is not in $\overline{V_r^\sharp}$, then v_t should not be in the caller-side negative binding after the return. The function for negative bindings behaves exactly the same as the function rv used in the object abstraction.

A last case to handle in function rv_{d^\sharp} is when b_c^\sharp is a negative binding and b_r^\sharp is a positive binding¹. The result must be a positive binding $\langle o^!, o^? \rangle$. The set $o^!$ represents the set of caller side variables that reference the bound object. Before the call, no variables are known to reference the bound object, whereas at the return site, variables in $o_r^!$ must reference the bound object. Hence, $rv(\emptyset, o_r^!)$ computes the set of caller side variables that must reference the bound object after the call. To compute the set of caller side variables that might reference the bound object, recall that before the call, parameter f was mapped to a negative binding $\overline{V_c^\sharp}$. Therefore, any caller side variable other than variables in $\overline{V_c^\sharp}$ might reference the bound object. Therefore, $rv(\mathbf{Var}_{\text{caller}} \setminus \overline{V_c^\sharp}, o_r^?)$ computes the caller side variables that might reference the bound object after the call. Formally:

$$\begin{aligned} rv_{d^\sharp}(\overline{V_c^\sharp}, \overline{V_r^\sharp}) &\triangleq \overline{rv(V_c^\sharp, V_r^\sharp)} \\ rv_{d^\sharp}(\overline{V_c^\sharp}, \langle o_r^!, o_r^? \rangle) &\triangleq \langle rv(\emptyset, o_r^!), rv(\mathbf{Var}_{\text{caller}} \setminus \overline{V_c^\sharp}, o_r^?) \rangle \end{aligned}$$

Having defined rv_{d^\sharp} , we now define the returnVal function using the summary function $summ_{m^\sharp}(d_2, d_1)$, where d_2 is the caller-side state pair before the call and d_1 represents a callee-side pair at the return site for the same automaton version that was abstracted by d_2 at the call. The summary function returns a caller-side state pair after the call. Recall that each state pair is an element from the domain $Q \times (F \rightarrow \mathbf{Bind}^\sharp)$ i.e. a pair containing the state and a map from each tracematch parameter to an element of \mathbf{Bind}^\sharp for the automaton version this pair abstracts.

The state of the generated caller-side pair after the call is the state from d_1 , the pair at the return site. In the formal definition below, this is state q_r . For each tracematch parameter f , we apply the function $rv_{d^\sharp}(m_c^\sharp(f), m_r^\sharp(f))$, where the parameters respectively represent the caller and callee side elements from \mathbf{Bind}^\sharp for parameter f . For an automaton

¹The case that b_c^\sharp is a positive binding and b_r^\sharp is a negative binding is not possible since positive bindings are lower than negative bindings in the bind lattice \mathbf{Bind}^\sharp .

version that is newly created in the procedure, we substitute d_2 with 0 and $m_c^\#(f)$ with \top .

$$\begin{aligned} \text{returnVal}(\langle e_p, d_1 \rangle, \langle n, d_2 \rangle) &\triangleq \text{summ}_{m^\#}(d_2, d_1) \\ \text{summ}_{m^\#}(0, \langle q_r, m_r^\# \rangle) &\triangleq \{ \langle q_r, \lambda f. rv_{d^\#}(\top, m_r^\#(f)) \rangle \} \\ \text{summ}_{m^\#}(\langle q_c, m_c^\# \rangle, \langle q_r, m_r^\# \rangle) &\triangleq \{ \langle q_r, \lambda f. rv_{d^\#}(m_c^\#(f), m_r^\#(f)) \rangle \} \end{aligned}$$

This completes the definition of the returnVal function for the tracematch abstraction.

Finally, the $\text{callFlow}(n^\#)$ function for the IFDS algorithm is defined as the empty set for both the object and tracematch state abstraction.

5.1 Collecting Useful Update Shadows

The analysis presented thus far can prove that the tracematch will never be in an accepting state at a given body statement. If this can be proved for all body statements in the program, the property expressed by the tracematch has been fully verified statically, and all dynamic instrumentation can be removed. However, the analysis may not be successful in ruling out *all* body statements. In this case, it is useful to compile a list of all transition statements that may contribute to a match at each body statement. Such a list is useful both for static verification and for optimizing a dynamic implementation. In static verification, this list helps the user identify the source of the bug, or to decide that the error report is a false positive. For example, if a collection is updated during iteration, the body statement is the failing `next` call on the iterator; more useful to the programmer would be the location of the collection update. In optimizing the dynamic tracematch implementation, all transition statements not leading to a potentially matching body statement can be removed, thereby reducing the runtime overhead of matching.

The analysis can be extended to keep track of relevant transition statements by using the Interprocedural Distributive Environment [60] (IDE) algorithm instead of IFDS. The IDE algorithm is an extension of IFDS to analysis domains of the form $\mathbf{Dom} \rightarrow L$, where \mathbf{Dom} satisfies the same conditions as for IFDS and L is a lattice of finite height. Indeed, IFDS is a special case of IDE with L chosen as the two-point lattice $\perp \sqsubseteq \top$. The IFDS version of the tracematch analysis presented thus far determines only whether a given pair $\langle q, m^\# \rangle$ is (\top) or is not (\perp) present at each program point. To keep track of transition statements leading to a match, we keep the same set $\mathbf{Dom} = Q \times (F \rightarrow \mathbf{Bind}^\#)$, and define L to contain \perp along with all subsets of the set of all transition statements. For each pair

$\langle q, m^\# \rangle$ present at a program point, the IDE version of the analysis maintains the set of transition statements that may have contributed to its presence.

The IDE transfer functions are extensions of the IFDS transfer functions that we have already presented. The transfer function for every statement other than a transition statement keeps the set of relevant transition statements for each tracematch state pair unchanged. The transfer function for a transition statement adds the current transition statement to the set of relevant transition statements for each tracematch state pair. There is one exception: when the transition statement transforms a tracematch state pair $\langle q, m^\# \rangle$ to itself, the transition statement is not added to the set of relevant transition statements for that pair. A transition statement that does not change the *concrete* tracematch state at run time is not considered relevant because removing it would not change the program behaviour. Such a statement occurs when the tracematch regular expression contains a subexpression of the form a^* , which causes a self-loop in the finite automaton. In what follows, we formally define the IDE transfer functions.

Like the IFDS algorithm, the IDE algorithm uses a decomposed transfer function. In the IDE algorithm, the pointwise transfer function has the form $\llbracket s \rrbracket_\bullet : (\mathbf{Dom} \cup \{0\}) \rightarrow \mathbf{Dom} \rightarrow L \rightarrow L$. Given a pair of elements d, d' from \mathbf{Dom} , the pointwise transfer function yields a transformer from L to L to be used to transform the lattice value associated with d to a lattice value to be associated with d' . The pointwise transfer function uniquely defines the overall transfer function $\llbracket s \rrbracket : (\mathbf{Dom} \rightarrow L) \rightarrow (\mathbf{Dom} \rightarrow L)$ as $\llbracket s \rrbracket(f) \triangleq \lambda d'. \bigsqcup_{d \in \mathbf{Dom} \cup \{0\}} \llbracket s \rrbracket_\bullet(d)(d')(f(d))$. Given an element d' , the transfer function applies the pointwise transfer function $\llbracket s \rrbracket_\bullet$ to each element of $\mathbf{Dom} \cup \{0\}$.

The pointwise transfer function $\llbracket s \rrbracket_{m^\#}$ from Section 3.3 can be re-used to implement the tracematch state analysis within the IDE framework. Statements other than $\mathbf{tr}(T)$ do not change the set of transition statements relevant to a match, so the transfer function yields the identity when $d' \in \llbracket s \rrbracket_{m^\#}(d)$ and the bottom function $\lambda l. \perp$ otherwise:

$$\llbracket s \rrbracket_{\sigma^\# \cup \{\rho^\#\}}(q, m^\#)(q', m'^\#) \triangleq \begin{cases} \lambda l. l & \text{if } \langle q', m'^\# \rangle \in \llbracket s \rrbracket_{m^\#}[\rho^\#](q, m^\#) \\ \lambda l. \perp & \text{otherwise} \end{cases}$$

The call and return flow functions are generalized in the same way from those used in the IFDS version of the algorithm.

The transfer function for a transition statement is similar, but in addition, its label ℓ is added to the set of relevant transition statements associated with each generated pair $\langle q', m'^\# \rangle$.

$$\llbracket \ell : \mathbf{tr}(T) \rrbracket_{\sigma^{\#\{\}}[\rho^{\#}]}(q, m^{\#})(q', m'^{\#}) \triangleq \begin{cases} \lambda l.l \sqcup \{\ell\} & \text{if } \langle q', m'^{\#} \rangle \in \llbracket \mathbf{tr}(T) \rrbracket_{m^{\#}[\rho^{\#}]}(q, m^{\#}) \wedge \langle q, m^{\#} \rangle \neq \langle q', m'^{\#} \rangle \\ \lambda l.l & \text{if } \langle q', m'^{\#} \rangle \in \llbracket \mathbf{tr}(T) \rrbracket_{m^{\#}[\rho^{\#}]}(q, m^{\#}) \wedge \langle q, m^{\#} \rangle = \langle q', m'^{\#} \rangle \\ \lambda l.\perp & \text{otherwise} \end{cases}$$

In the second case above, when $\langle q, m^{\#} \rangle = \langle q', m'^{\#} \rangle$, the label is not added. As discussed earlier, a transition statement that does not change the *concrete* tracematch state is not considered relevant because removing it would not change the program behaviour. To soundly exclude such a transition statement, we must ensure that it does not change the *concrete* state. The following proposition assures us that this is the case when the transition statement does not change the *abstract* state.

Proposition 7. *If $\langle q_2, m_2 \rangle \in e^{\#}[T, \rho](q_1, m_1)$; $\langle q_1, m_1 \rangle \neq \langle q_2, m_2 \rangle$; $\rho^{\#}$ overapproximates ρ ; and $\langle q_1, m_1 \rangle R_m[\rho] \langle q_1^{\#}, m_1^{\#} \rangle$; then there exists $\langle q_2^{\#}, m_2^{\#} \rangle \in \llbracket \mathbf{tr}(T) \rrbracket_{m^{\#}[\rho^{\#}]}(q_1^{\#}, m_1^{\#})$ such that $\langle q_1^{\#}, m_1^{\#} \rangle \neq \langle q_2^{\#}, m_2^{\#} \rangle$ and $\langle q_2, m_2 \rangle R_m[\rho] \langle q_2^{\#}, m_2^{\#} \rangle$.*

A proof of the proposition is provided in Appendix A.

It may happen that a transition statement in a loop changes the tracematch state in the first iteration but not in any subsequent iteration. An optimized dynamic implementation should execute the first, relevant transition, but should avoid executing the redundant transitions in subsequent iterations of the loop. This can be achieved by peeling one iteration of every loop containing a transition statement prior to performing the IDE analysis. The analysis will mark the transition as relevant in the peeled iteration and unnecessary in the remaining loop.

5.2 Empirical Evaluation

We empirically evaluated the precision of our analysis and compared it to Bodden et al.'s existing tracematch analysis [12], which uses may-point-to information to rule out possibly matching transition statements. The evaluation was performed on the tracematches from [12] plus one new one (FailSafeEnumHashtable), summarized below:

ASyncIteration: A synchronized collection should not be iterated over without owning its lock.

FailSafeEnum: A vector should not be updated while enumerating it.

FailSafeEnumHashtable: A hashtable should not be updated while enumerating its keys or values.

FailSafeIter: A collection should not be updated while iterating over it.

HasNext: The `hasNext` method should be called prior to every call to `next` on an iterator.

HasNextElem: The `hasNextElem` method should be called prior to every call to `nextElement` on an enumeration.

LeakingSync: A synchronized collection should only be accessed through its synchronized wrapper.

Reader: A `Reader` should not be used after its `InputStream` has been closed.

Writer: A `Writer` should not be used after its `OutputStream` has been closed.

We applied the above tracematches to the benchmarks `antlr`, `bloat`, `hsqldb`, `luindex`, `lython`, and `pmd` from the DaCapo benchmark suite, version 2006-10-MR2 [10]. Most of the benchmarks use reflection to load key classes. We instrumented the benchmarks using ProBe [45] and *J [27] to record actual uses of reflection at run time, and provided the resulting reflection summary to the static analysis. The `lython` benchmark generates code at run time which it then executes; for this benchmark, we made the unsound assumption that the generated code has no effect on aliasing or tracematch state.

Each of the 6 benchmarks was analyzed with each of the 9 tracematches, a total of 54 cases (tracematch/benchmark pairs). The 54 cases evaluated contained a total of 5409 final transition statements. We define a transition statement $\langle a, b \rangle$ as *final* if the tracematch automaton contains a transition to an accepting state on a . Thus, a match can be completed only at a final transition statement and implies a violation of the specified property. We count only final transition statements in the reachable part of the call graph. Of these, our analysis proved that 4815 (89 %) will never complete a match. Thus, a programmer wishing to check the tracematch properties need only examine 11 % of the uses of the features checked by the tracematches.

Bodden’s analysis comprises three stages. The first stage (QC) considers only the set of tracematch symbols present in the program; if every word satisfying the tracematch pattern contains a given symbol and that symbol does not appear anywhere in the program, the tracematch cannot match and hence the safety property, cannot be violated. The second

stage (FI) considers the may-point-to sets of the variables in each transition statement. If a sequence of transitions is to lead to a violation, they must have consistent bindings, which is possible only if their points-to sets overlap. Bodden observed this stage to reduce the number of matching transition statements in seven of nine cases (tracematch/benchmark pairs); in one case, it completely eliminated all possibility of a match. The third stage (FS) considers the order in which symbols occur during execution, but does not coordinate this order with the flow of individual objects; Bodden observed no precision improvement over FI. Since our analysis subsumes QC and the precision of FI and FS is equivalent in practice, the evaluation in this section compares our analysis with FI.

Of the 54 cases, 36 actually used the features described by the tracematch, in the sense that QC did not rule out a match. These cases contained 1509 final transition statements, and our analysis proved that 915 will never complete a match and hence do not violate the tracematch property. Each of the 36 cases is represented by a circle in Figure 5.1. Beside each circle is a fraction giving the number of transition statements at which a match could not be ruled out and the total number of final transition statements. In 15 of the 36 cases, our analysis ruled out all matches; i.e. it successfully verified that the benchmark is free of any violations of the property specified by the tracematch. These cases are represented by the 15 fully white circles. In comparison, the FI analysis ruled out all matches in only 1 of the 36 cases where QC was unsuccessful (LeakingSync/Luindex).

However, the two analyses are complementary in that they are successful on *different* transition statements. Our analysis fares better when the temporal order in which events occur is relevant in ruling out the match. When the feature monitored by the tracematch is used in many distinct ways in different parts of the program, like iterators, FI is sometimes better at distinguishing the different uses based on the allocation sites of the objects involved. More specific examples are discussed in the rest of this section. The two analyses can be run together, and the combination is more precise than each analysis on its own.

5.2.1 Discussion of Results

In this section we take a closer look at some of the results from Figure 5.1. Of the 21 remaining cases in which all violations could not be removed, 4 involve the HasNext and HasNextElem tracematches. In one case (HasNext/pmd), all possible matches are actual violations of the tracematch pattern. The code uses `isEmpty` to ensure that a collection is not empty, then calls `next` on an iterator without calling `hasNext` first.² Similar violations

²The HasNext tracematch could be extended to check if either `hasNext` or `isEmpty` has been called. However, `isEmpty` must be invoked on the collection being iterated. This requires changing the tracematch

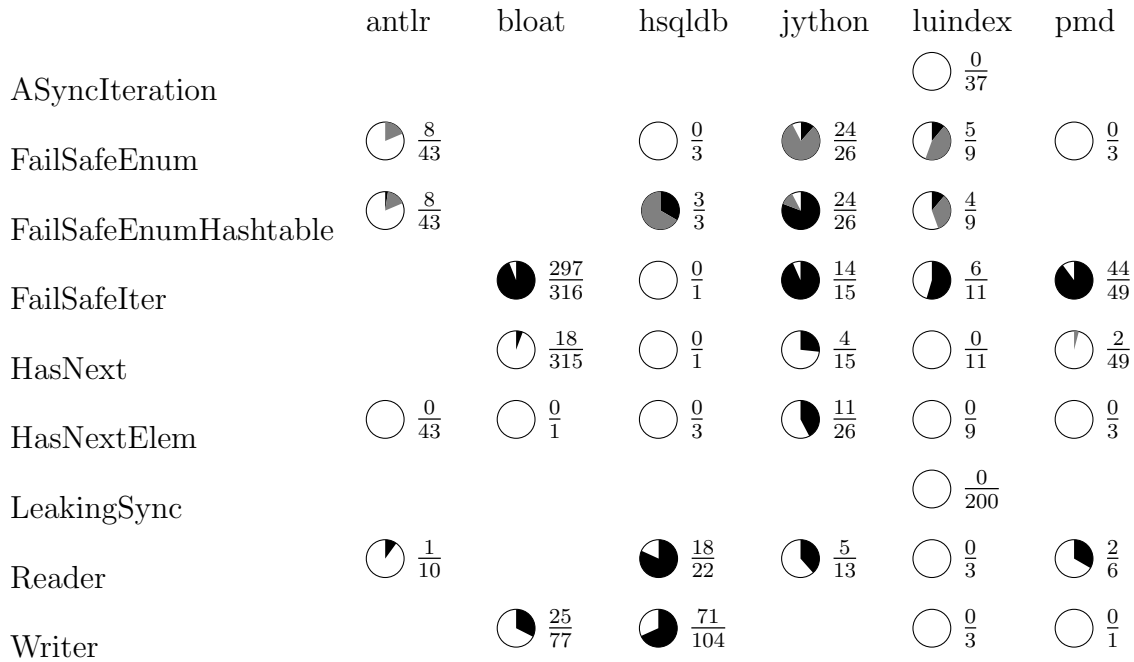


Figure 5.1: Fraction of final transition statements that may complete a match. The white part of each circle represents those that cannot complete a match. The black part represents those at which a match cannot be ruled out, due either to analysis imprecision or an actual violation. The gray part represents those at which a violation is known to exist.

occur in the other three cases (in jython and in HasNext/bloat). In addition, these cases contain false positives due to iterators stored only in fields and not local variables. In the HasNext and HasNextElem tracematches, flow-sensitive tracking of individual objects is crucial to ensure that the `hasNext` call occurs on the same object as the calls to `next`. Thus, while our analysis ruled out matches at 441 of the 476 final transition statements, FI could not rule out a match at any of them.³

In 11 cases involving the FailSafe* tracematches, the analysis found both violations and likely false positives due to aliasing. Some collections, such as `java.util.Hashtable`, keep a singleton enumeration and iterator which are reused every time the collection is empty.

to bind both the iterator and the collection, making the tracematch contain multiple parameters hence making the tracematch another example of a temporal property that involve multiple interacting objects.

³Some transition statements were ruled out in [12] because they were determined to be in code that could not be reached at run time. Our evaluation considers only reachable code.

This violates the tracematch because an iterator is being used even though a collection with which it was previously associated has since been updated. This accounts for many but not all of the detected matches; the associated transition statements are shown in gray in Figure 5.1.

At many of the other transition statements, a match cannot be ruled out because a loop iterating over a collection contains calls leading to very deep call chains comprising many methods, some of which update collections. The analysis is not able to prove that all these collections are distinct from the collection being iterated. In some of these loops, may point-to information would help: FI ruled out matches at 19 transition statements in 3 cases that our analysis did not. On the other hand, our analysis ruled out matches at 54 transition statements in 2 cases that FI did not. Since so many methods are transitively called from the loop, it is difficult to examine them all by hand to determine whether any of the updated collections may in fact alias the iterated collection.

The cases involving the Reader and Writer tracematches can be classified into three categories. The first category includes readers/writers of files, which are closed after the last access. In these instances, our analysis proved all accesses occur before the close, thereby ruling out a violation. Since FI ignores the order of the events, it could not rule out a violation. The second category includes readers/writers of the standard input/output streams. These are never closed, and the FI analysis proves this fact, thus ruling out a match. These streams are often referenced only by their static field in the System class, and not by any local variables. Therefore, our analysis cannot distinguish them from other readers/writers on which close is called, and cannot rule out a match. The third category includes readers/writers for which neither analysis can rule out a violation. We noticed the following pattern in several benchmarks. A loop repeatedly calls a helper method that uses the reader/writer. Both the loop and the helper method contain a try block. An exception during the input/output operation is caught in the helper, which closes the stream and re-throws the exception. The try block protecting the loop catches the exception, thereby terminating the loop and preventing any further use of the reader/writer. Because our analysis does not distinguish normal and exceptional returns, it conservatively assumes that the loop could continue iterating and therefore use the reader/writer after the stream was closed. Overall, our analysis proves three Reader/Writer cases correct compared to two for FI, but FI rules out slightly more final transition statements than our analysis.

In summary, although our analysis is often more precise than FI, the two are complementary in that each is more effective than the other on certain code patterns. In many practical cases, our analysis is precise enough to rule out a match. However, there remain cases where the abstraction loses all local variable references to an object. Thus, our analysis would benefit from some information about pointers from within the heap.

Chapter 6

Optimizations

As discussed in Section 3.2 we use the alias set analysis to create a static abstraction for the objects in the program. A key reason for our choice was that unlike a shape analysis which emphasizes the precise relationships between objects, and is expensive to model, an alias set analysis, like a pointer abstraction, focuses on local pointers to objects. This makes computing the alias set abstraction faster than shape analyses. However, since the analysis is flow-sensitive and inter-procedural it is still considerably slower than most points-to analyses, despite the use of the efficient IFDS algorithm. In this chapter we describe two ways to further speed-up the alias set analysis; callee summaries providing effect and return value information and caller summaries that make conservative assumptions at method entry.

Flow sensitive analyses take into account the order of instructions in the program and compute a result for each program point. Although typically more precise than those that are insensitive to program flow, flow-sensitive analyses often have longer execution times than their flow-insensitive counterparts. Computing such precise information for each program point is often overkill; clients of the analysis need precise results only at specific places. Long segments of code might exist where a client neither queries the analysis nor cares about its precision. As an example, consider the code in Figure 6.1 in light of the example tracematch discussed in Chapter 1. The tracematch analysis is a client of the alias set analysis as it requires flow-sensitive tracking of individual objects to statically determine runtime objects, in this case involved in operations on lists and iterators. Notice that precise alias sets are required only when operations of interest occur. For the example, these are the two calls to `next` at lines 7 and 10 and the call to `add` at line 11. On the other hand, a typical alias set analysis computes flow-sensitive results for all program points even though it is likely to be queried only at a few places.

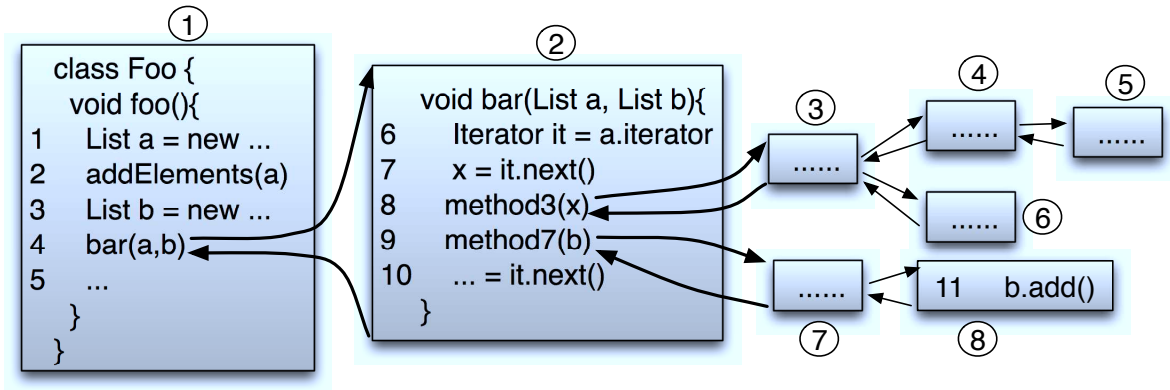


Figure 6.1: Sample code illustrating the use of callee and caller summaries

In such situations, we propose the use of a selectively flow-sensitive alias set analysis that uses callee method summaries as a cheaper option. Only methods that contain a point of interest (which we call *shadows*), or that transitively call methods containing shadows, are analyzed flow-sensitively¹. For all other methods, callee summaries providing effect information for the parameters of a method invocation and the possible return value are used. If callee summaries were available, only methods 1, 2, 7 and 8 from Figure 6.1 would have to be analyzed flow-sensitively since they contain shadows or call methods containing shadows. For the entire segment of code represented by methods 3-6, flow-sensitive information is not required and callee summaries can be used instead. In particular, while analyzing method 2 the alias set analysis need not propagate the analysis into method 3 at line 8 and instead its callee summary can be used. From the client’s perspective this is acceptable since it does not query any program point within methods 3-6. In fact, as long as callee summaries contain sufficient information so that foregoing flow-sensitive analysis of methods without shadows does not affect alias set precision in methods with shadows, the client’s precision will be unaffected. Details of the construction of callee summaries and their use in the alias set analysis are given in Section 6.2.

The advantage any static analysis derives from interprocedurally analyzing a program is that the analysis need not make conservative worst case assumptions at method entry. This certainly holds true for the alias set analysis. At a callsite, the analysis ensures an appropriate mapping from the caller scope arguments to the callee scope parameters so that alias sets in the callee precisely represent aliasing at the start of the method.

¹For the tracematch analysis, the shadows are the transition statements.

However, when efficiency is a bigger concern, we propose the use of caller summaries which are conservative and sound approximations of incoming alias sets. A direct benefit of using such summaries at method entries is that methods that were previously analyzed flow-sensitively only to obtain precise entry mappings for methods containing shadows no longer require flow-sensitive analysis. For example, since methods 1 and 7 in Figure 6.1 were analyzed flow-sensitively only because they contain calls to methods 2 and 8, with the added use of caller summaries this is no longer required. Only methods 2 and 8 will be analyzed flow-sensitively with caller summaries used to seed their initial alias sets and callee summaries used at all callsites.

Unlike callee summaries, caller summaries can affect the precision of the alias set abstraction since important aliasing information available at a particular callsite might not be propagated into the callee and instead some conservative assumption is made. The degree to which the use of caller summaries affects precision is dependent on the choice of caller summary as well as the client analysis.

The remainder of this chapter is organized as follows:

- In Section 6.1 we summarize the alias set analysis as discussed in Chapters 3 and 5.
- In Section 6.2 we describe callee method summaries for the alias set analysis which provide sufficient information at a method callsite to forego flow-sensitive analysis of the callee without a loss of precision in the caller. We present algorithms to compute such summaries and a transfer function that employs the computed summary.
- Section 6.3, introduces the simplest caller summary as a proof of concept to using such summaries to flow-sensitively analyze even fewer methods. A transfer function for the alias set abstraction that uses both callee and caller summaries is also presented.
- We empirically evaluate the effect of caller summaries on the precision of the trace-match analysis and present, in Section 6.4, precision metrics for the alias set abstraction. The effect on the running time of different incarnations of the alias set analysis is discussed.
- We end with some concluding remarks in Section 6.6.

6.1 Alias Set Analysis

We assume that the program has been converted into an SSA-based intermediate representation containing the following kinds of instructions:

$$s ::= Copy(v_1 \leftarrow v_2) \mid Store(\mathbf{h} \leftarrow v) \mid Load(v \leftarrow \mathbf{h}) \mid \\ Null(v \leftarrow \mathbf{null}) \mid New(v \leftarrow \mathbf{new}) \mid Call(m(p_0 \cdots p_k))$$

The interprocedural control flow graph is created in the standard way; nodes represent instructions and edges specify predecessor and successor relationships. Each procedure begins with a unique *Start* node and ends at a unique *Exit* node. By construction, a call instruction is divided into two nodes; call and return. A call edge connects the call node in the caller with the start node in the callee. A return edge connects the exit node in the callee with the return node in the caller. A *CallFlow* edge connects a call node to its return node completely bypassing the callee (Figure 6.2). This edge is parameterized with the method it bypasses and the variable the return from the call is assigned to.

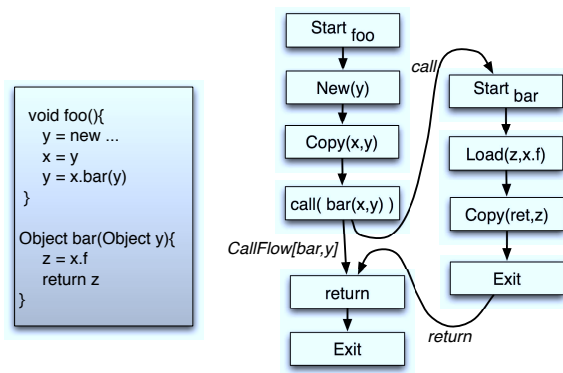


Figure 6.2: Interprocedural control flow graph with *call*, *return* and *CallFlow* edges.

The intra-procedural transfer functions for the alias set abstraction were presented in Figure 3.2 which we reproduce in Figure 6.3. In Figure 3.3 we gave a detailed example illustrating the effect of the transfer functions on a sequence of statements. In Chapter 5 we extended these transfer functions in terms of the IFDS functions for flow into (passArgs) and out of (returnVal) procedure calls. The same functions are reproduced in Figure 6.3 but this time in terms of the transfer functions for *call* and *return* where the overall effect of calling a function m is $\llbracket return \rrbracket \circ \llbracket m \rrbracket \circ \llbracket call \rrbracket$ for each possible callee. The function $\llbracket call \rrbracket_{o^\#}$ and $\llbracket return \rrbracket_{o^\#}$ are the same as $update_{o^\#}[r](o^\#)$ and $rv(o_c^\#, o_r^\#)$ from Chapter 5.

$$\begin{aligned}
\llbracket s \rrbracket_{o^\#}^1(o^\#) &\triangleq \begin{cases} o^\# \cup \{v_1\} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \in o^\# \\ o^\# \setminus \{v_1\} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \notin o^\# \\ o^\# \setminus \{v\} & \text{if } s \in \{v \leftarrow \mathbf{null}, v \leftarrow \mathbf{new}\} \\ o^\# & \text{if } s = \mathbf{h} \leftarrow v \\ \text{undefined} & \text{if } s = v \leftarrow \mathbf{h} \end{cases} \\
\text{focus}[h^\#](v, o^\#) &\triangleq \begin{cases} \{o^\# \setminus \{v\}\} & \text{if } o^\# \notin h^\# \\ \{o^\# \setminus \{v\}, o^\# \cup \{v\}\} & \text{if } o^\# \in h^\# \end{cases} \\
\llbracket s \rrbracket_{O^\#}^1[h^\#](O^\#) &\triangleq \begin{cases} \bigcup_{o^\# \in O^\#} \llbracket s \rrbracket_{o^\#}^1(o^\#) & \text{if } s \neq v \leftarrow \mathbf{h} \\ \bigcup_{o^\# \in O^\#} \text{focus}[h^\#](v, o^\#) & \text{if } s = v \leftarrow \mathbf{h} \end{cases} \\
\llbracket s \rrbracket_{\rho^\#}^1(\rho^\#, h^\#) &\triangleq \llbracket s \rrbracket_{\text{gen}}^1 \cup \llbracket s \rrbracket_{O^\#}^1[h^\#](\rho^\#) \\
\llbracket s \rrbracket_{\text{gen}}^1 &\triangleq \begin{cases} \{\{v\}\} & \text{if } s = v \leftarrow \mathbf{new} \\ \emptyset & \text{otherwise} \end{cases} \\
\llbracket s \rrbracket_{h^\#}^1(\rho^\#, h^\#) &\triangleq \llbracket s \rrbracket_{O^\#}^1[h^\#] \left(\begin{cases} h^\# \cup \{o^\# \in \rho^\# : v \in o^\#\} & \text{if } s = \mathbf{h} \leftarrow v \\ h^\# & \text{otherwise} \end{cases} \right) \\
\llbracket s \rrbracket_{\rho h^\#}^1(\rho^\#, h^\#) &\triangleq \langle \llbracket s \rrbracket_{\rho^\#}^1(\rho^\#, h^\#), \llbracket s \rrbracket_{h^\#}^1(\rho^\#, h^\#) \rangle \\
\llbracket \text{call} \rrbracket_{o^\#}^1(o^\#) &\triangleq \{r(v) : v \in o^\# \cap \text{dom}(r)\} \\
rv(o_c^\#, o_r^\#) &\triangleq \begin{cases} o_c^\# & \text{if } p \text{ does not return a value} \\ o_c^\# \cup \{v_t\} & v_s \in o_r^\# \\ o_c^\# \setminus \{v_t\} & v_s \notin o_r^\# \end{cases} \\
\llbracket \text{return} \rrbracket_{o_c^\#}^1(o_c^\#) &\triangleq \{rv(o_c^\#, o_r^\#) : o_r^\# \in \llbracket m \rrbracket \circ \llbracket \text{call} \rrbracket\}
\end{aligned}$$

Figure 6.3: Transfer functions on individual alias sets. The superscript¹ identifies the version of the transfer function; we will present modified versions of the transfer functions later. We illustrated the effect of the transfer functions using the example statement sequence shown in Figure 3.3.

6.2 Callee Summaries

Although precise, the alias set analysis in its original form is expensive to compute. Using efficient data structures [50] and algorithms [54] only improves the efficiency to some extent. In situations where a faster running time is desired, we propose the use of method summaries. In this section, we discuss the use of callee summaries that decrease the computation load without any effect on a client analysis.

The key insight is that clients of a flow-sensitive whole program analysis often need precise information at a small subset of program points. On the other hand, a flow-sensitive program analysis computes precise information at all program points and therefore computes a lot more information than required. Computing this unnecessary information is wasteful and should be avoided. We use callee summaries to achieve this.

Before we explain the contents of a callee summary, let us see how the alias set analysis can use such summaries. Consider a callsite, with a target method m . If an oracle predicts that a client of the alias set analysis never queries any program point within m or any methods transitively called by m , then computing flow-sensitive alias results for all methods in the transitive closure of m is unnecessary. Instead a callee summary, which provides information regarding the parameters and return value, could be used. For many client analyses such an oracle exists. For example, in the case of the tracematch analysis, the shadows are transition statements, i.e., operations that change the state an object is in and are statically known ahead of time.

The key requirement we put on a callee summary is that it should enable the analysis to bypass flow-sensitively analyzing a method without impacting precision in the caller, i.e., the alias set abstraction computed after a callsite should be the same irrespective of whether the method was analyzed flow-sensitively or whether a callee summary was used. Figure 6.4 provides a summary of the contents of such a summary. The summary is divided into escape (α_{esc}) and return value (α_{ret}) information.

To determine the contents of a callee summary, one must understand the effect of a method call on the alias set abstraction. First, the callee might escape the receiver or arguments of the call. This might occur *directly*, when a callee’s parameter is stored in a field, or *indirectly*, when a parameter is copied to a local reference which is then stored. In Figure 6.5, the function `foo` escapes both its parameters, p directly via a store to field f of class `Foo` and q indirectly by first copying the reference to y and then storing in `Foo.f`. Therefore, a callee summary analysis must track such copies and ultimately provide a list of all parameters that might have escaped.

Second, the return value from the callee might be assigned to a reference in the caller.

Escape Information (α_{esc})	
params	set of parameters (including receiver) that may be stored into the heap by m or procedures transitively called by m
Return Value Information (α_{ret})	
params	set of parameters (including receiver) that might be returned by m .
heap	might an object loaded from the heap be returned?
fresh	might a newly created object be returned?
escaped	might a newly created object be stored in the heap before being returned?
null	might a null reference be returned?

Figure 6.4: Callee Summary for a callsite with target method m

To see how this might affect aliasing in the caller, consider once again the example in Figure 6.5. The function `foo` returns the pointer y which is a copy of q , one of `foo`'s parameters. Therefore, the returned reference is the argument which is mapped to q , in this case variable b . At run time, the effect of calling `foo` is that after the call, a and b must point to the same object. Let us examine the effect on the abstraction at the callsite if the interprocedural transfer functions from Figure 6.3 were used. $\llbracket call \rrbracket$ determines that b and q point to the same location and $\llbracket foo \rrbracket$ determines that q and y point to the same location. This leads $\llbracket return \rrbracket$ to infer that since b and y point to the same location and y is assigned to a , b and a must point to the same location after the call; an alias set containing both a and b is created in the caller. In order to forego flow-sensitive analysis of `foo` in favour of a callee summary, the summary must specify which of the callee's parameters might be returned so that similar updates can be made at the callsite. Other possible returned references include references to newly created objects or those loaded from the heap.

6.2.1 Computing Callee Summaries

The algorithm to compute the set of parameters that escape (α_{esc}) from a method m is presented in Figure 6.6. The algorithm takes as input a SSA-based control flow graph of the method and returns a set of indices which refer to the positions of parameters in the method's signature which might have escaped². At Line 1 the algorithm invokes the

²Recall from Section 6.1 that we write a function call as $m(p_0, \dots p_k)$ where p_0 denotes the receiver of the call and p_1 to p_k are the arguments.

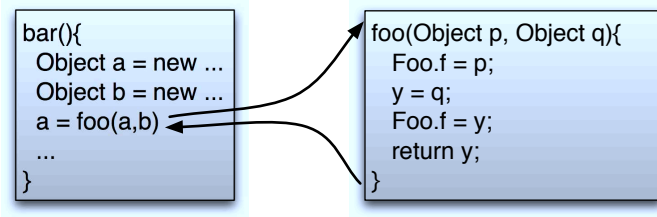


Figure 6.5: Example illustrating the effect of a method call on alias sets in the caller.

helper function `getSeededWorklist` (Figure 6.7) which populates and returns a worklist with variables that either escaped through a store or through a function call from within m . The algorithm then proceeds through each variable v in the worklist. Using the SSA property that each variable has a single reaching definition, the algorithm retrieves the unique definition def of v (line 6). If def represents the `Start` node, then v is a receiver or a parameter and the appropriate index is added to the `mayEscape` set. For a copy instruction $v \leftarrow s$, s is added to the worklist, since v and s both point to the same escaped object. Notice that the order between the instruction that escapes v and the copy from s to v does not matter, since in SSA-form once a variable is defined its value remains unchanged. If variable v is assigned the return value from a function call, then all arguments corresponding to the parameters that might be returned are added to the worklist since these might have escaped (lines 10-14). A SSA ϕ instruction acts as a multi-variable copy statement.

Figure 6.8 presents the algorithm to compute the return value summary for a function m . The algorithm maintains a worklist of variables that might be returned. The worklist is seeded with the unique return variable of m . For each variable v in the worklist, depending on its unique definition, the return value summary and the worklist are updated. In lines 10-12, if v is defined at the `Start` node, then, since a `Start` node defines the receiver or parameters of method m , the corresponding index of the parameter is stored in `params`. This represents the situation when the receiver or a parameter to m might be returned. Lines 13-21 update the return value summary if v is assigned the return value at a callsite. The return value summaries of all possible target methods at the callsite are consulted and the `fresh`, `heap` and `null` fields of the summary of m appropriately updated. If any of the return value summaries indicate that a receiver or parameter might be returned, the corresponding argument is added to the worklist. *Copy* and *Phi* instructions add sources of assignments to the worklist. *Load*, *New* and *Null* instructions require an update to the corresponding `heap`, `fresh` and `null` fields of the return value summary.

```

input: SSA-based CFG of method  $m$ 
output: mayEscape
declare mayEscape : Set[Int], WorkList: FIFOWorklist[Var], seen : Set[Var]
1   WorkList: getSeededWorklist(cfg)
2   while WorkList not empty
3     Select and Remove variable  $v$  from WorkList
4     if seen contains  $v$  then continue fi
5     add  $v$  to seen
6      $def = \text{uniqueDef}(cfg, v)$ 
7     switch  $def$ 
8       case  $def = \text{Start}(p_0 \cdots p_k)$ : mayEscape += {  $i : p_i = v$  } end case
9       case  $def = \text{Copy}(v, s)$ : add  $s$  to WorkList end case
10      case  $def = \text{CallSite}(args, \text{retval})$ :
11        foreach  $tgt \in \text{callees}(def)$  do
12          WorkList += {  $args(i) : i \in \text{RetValSummaries}(tgt).params$  }
13        od
14      end case
15      case  $def = \text{Phi}$  :
16        foreach  $\text{Copy}(v, s) \in \text{phi.defs}(v)$  do add  $s$  to WorkList od
17      end case
18    end switch
19  od

```

Figure 6.6: Algorithm to compute callee escape summary (α_{esc}) for a method m

The algorithm maintains a set $seen$ that is updated (Line 7) every time a variable v is processed. This ensures that we process a variable that makes its way into the WorkList only once. Once the WorkList is empty, the $seen$ set contains all variables that flow to the return value. The last line (Line 30) of the algorithm uses the $seen$ set to update the escaped information for the return value summary, i.e., whether the object being returned has escaped or not. The set of $seen$ variables is intersected with the return from the helper function `allEscaped`. If the intersection is non-empty `escaped` is set to true for the method m . To understand why this is necessary, let us consider how objects escape. In our IR, an object escapes if the reference v that points to the object is the source of a `Store(h ← v)` instruction. This might occur *directly* where the reference v itself is stored before being returned or *indirectly* as is the case in Figure 6.9(a). Alternately, it is possible that an object escaped because a reference to that object was passed as an argument to another method which escaped it (via a `Store`). Again, this could occur directly (where the

```

input: SSA-based CFG of method  $m$ 
output: WorkList
declare WorkList: FIFOWorklist[Var]
1   foreach instruction  $inst \in \text{cfg}$  do
2     switch  $inst$ 
3       case  $inst = \text{Store}(v)$  : add  $v$  to WorkList end case
4       case  $inst = \text{CallSite}(\text{args}, \text{retval})$  :
5         foreach  $tgt \in \text{callees}(inst)$  do
6           WorkList += {  $\text{args}(i) : i \in \text{EscapeSummaries}(tgt)$  }
7         od
8       end case
9     end switch
10  od

```

Figure 6.7: getSeededWorklist: algorithm to obtain escaped variables

returned reference was also an argument to some method call that escaped the argument) or indirectly as shown in Figure 6.9(b). The algorithm in Figure 6.8 could have caught the first case by simply checking whether any of the set of variables in *seen* occur as a source of a *Store* instruction. For example, in Figure 6.9(a) the set *seen* is $\{x,y\}$ and y is the source in a *Store* instruction. The second case, when called methods escape an argument, could be caught by checking whether any of the variables in *seen* were passed as arguments and whether the argument escaped. In Figure 6.9(b) the *seen* set is $\{x,y\}$ and y is an argument to a call to *foo* where it escapes. Unfortunately, the way objects escape is not limited to these two types of examples. Consider for instance Figure 6.9(c). The *seen* set is $\{x,y\}$ and neither of these are the source in a *Store* instruction. Yet, the object pointed to by these references has escaped via the copy to z and the *Store* of z . Therefore, on Line 30 of the algorithm we find whether the escaped predicate should be set to true by intersecting the *seen* set with the output of the helper function *allEscaped* shown in Figure 6.10. The function maintains a worklist of variables from method m that are known to have escaped. The Worklist is initialized through the use of the helper function *getSeededWorklist* which returns all variables of method m that have escaped either through a *Store* or through a function call. At each step, an escaped variable from the worklist is dequeued and its definition traced backwards. If the unique definition is a *CallSite*, i.e., the escaped variable was assigned the return from a call, then the return value summary of each call target is queried to see if any of its parameters are returned. If yes, corresponding arguments are added to the worklist since these too have escaped (Lines 8-13). For a variable whose definition comes from a copy, the source variable is added to the worklist, since it too has

```

input: SSA-based CFG of method  $m$ 
output: retValSum
declare WorkList: FIFOWorklist[Var], seen : Set[Var]
1   retValSum = { params: Set[Int], heap = fresh = escaped = null = false }
2   if  $m.isVoid$  then return retValSum fi
3   Insert unique return variable into WorkList
4   while WorkList not empty
5     Select and remove variable  $v$  from WorkList
6     if seen contains  $v$  then continue fi
7     add  $v$  to seen
8      $def = uniqueDef(cfg, v)$ 
9     switch  $def$ 
10    case  $def = Start(p_0 \cdots p_k)$  :
11      retValSum.params += {  $i : p_i = v$  }
12    end case
13    case  $def = CallSite( args, retval )$  :
14      foreach  $tgt \in callees(def)$  do
15        calleeRetValSum = RetValSummaries( $tgt$ )
16        if calleeRetValSum.fresh then retValSum.fresh = true fi
17        if calleeRetValSum.heap then retValSum.heap = true fi
18        if calleeRetValSum.null then retValSum.null = true fi
19        WorkList += {  $args(i) : i \in calleeRetValSum.params$  }
20      od
21    end case
22    case  $def = Copy(v, s)$  : add  $s$  to WorkList end case
23    case  $def = Phi$  :
24      foreach  $Copy(v, s) \in phi.defs(v)$  do add  $s$  to WorkList od
25    end case
26    case  $def = Load$  : retValSum.heap = true end case
27    case  $def = New$  : retValSum.fresh = true end case
28    case  $def = Null$  : retValSum.null = true end case
29  end switch
30  if  $seen \cap allEscaped(cfg) \neq \{\}$  then retValSum.escaped = true fi

```

Figure 6.8: Algorithm to compute the return value summary (α_{ret}) for a method m

escaped. The end result of computing `allEscaped` for method m is a set of variables which have escaped. Line 30 of the algorithm in Figure 6.8 intersects this set with the set of *seen* variables to find if the object being returned has escaped.

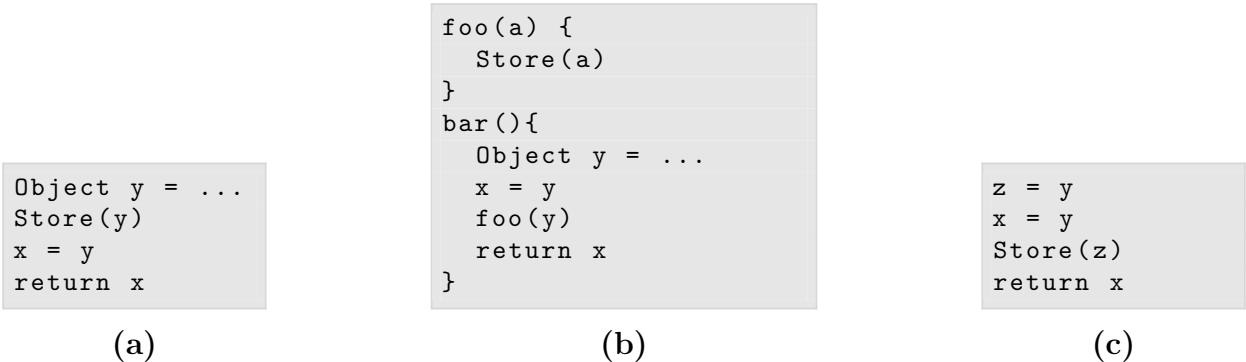


Figure 6.9: Tracking whether the returned object has escaped

Since the callee summary of a function m depends on summaries of functions called by m , the algorithms presented must be wrapped in an interprocedural fixed-point computation. A worklist keeps track of all functions whose summaries may need to be recomputed. Whenever the summary of a function changes, all of its callers are added to the worklist. The computation iterates until the worklist becomes empty. It is interesting to note that the algorithms presented in this section are computing reachability along dataflow paths and could be represented as IFDS problems.

6.2.2 Using Callee Summaries

To leverage callee summaries in the alias set analysis, the transfer functions from Figures 6.3 are modified. These modifications are presented in Figure 6.11. We denote the set of methods that contain shadows or that transitively call methods containing shadows by M^* . The function $\llbracket call \rrbracket_{o\#}^1$ is modified so that arguments in the caller are mapped to parameters in the callee only for methods in M^* , the methods that are still analyzed flow-sensitively. Since return instructions are only encountered in methods not using callee summaries, no change is required to the return function $\llbracket return \rrbracket_{o\#}^1$.

For methods not in M^* , we define the transfer function *CallFlow* for the similarly named edge connecting a call node to a return node in the caller. The *CallFlow* function uses two helper functions `mustReturn` and `mightReturn` which employ the return value summary α_{ret} to update alias sets by simulating the effect of analyzing the callee. The

```

input: SSA-based CFG of method  $m$ 
output: allEscape
declare WorkList: FIFOWorklist[Var], allEscape : Set[Var]
1   WorkList: getSeededWorklist(cfg)
2   while WorkList not empty
3     Select and remove variable  $v$  from WorkList
4     if allEscape contains  $v$  then continue fi
5     add  $v$  to allEscape
6      $def = \text{uniqueDef}(cfg, v)$ 
7     switch  $def$ 
8       case  $def = \text{CallSite}(args, \text{retval})$  :
9         foreach  $tgt \in \text{callees}(def)$  do
10          calleeRetValSum = RetValSummaries( $tgt$ )
11          WorkList += {  $args(i) : i \in \text{calleeRetValSum.params}$  }
12        od
13      end case
14      case  $def = \text{Copy}(v, s)$  : add  $s$  to WorkList end case
15      case  $def = \text{Phi}$  :
16        foreach  $\text{Copy}(v, s) \in \text{phi.defs}(v)$  do add  $s$  to WorkList od
17      end case
18    end switch
19  od

```

Figure 6.10: allEscaped: algorithm to compute variables that escape from a method m

$$\begin{aligned}
\llbracket call \rrbracket_{o^\#}^2(o^\#) &\triangleq \begin{cases} \llbracket call \rrbracket_{o^\#}^1(o^\#) & \text{if } s = call \wedge target(call) \in M^* \\ \emptyset & \text{otherwise} \end{cases} \\
mustReturn(o^\#, \alpha_{ret}) &\triangleq \begin{cases} true & \text{if } !\alpha_{ret}.fresh \wedge !\alpha_{ret}.heap \wedge !\alpha_{ret}.null \wedge \\ & \forall p, p \in o^\# : p \in \bar{r}(\alpha_{ret}.params) \\ false & \text{otherwise} \end{cases} \\
mightReturn[h^\#](o^\#, \alpha_{ret}) &\triangleq \begin{cases} true & \text{if } (o^\# \in h^\# \wedge \alpha_{ret}.heap) \vee \\ & \exists p, p \in o^\# : p \in \bar{r}(\alpha_{ret}.params) \\ false & \text{otherwise} \end{cases} \\
CallFlow_{o^\#}^2[h^\#, m, v](o^\#) &\triangleq \begin{cases} \emptyset & \text{if } m \in M^* \\ \{o^\# \cup v\} & \text{if } m \notin M^* \wedge \\ & mustReturn(o^\#, m.\alpha_{ret}) \\ \{o^\# \setminus v, o^\# \cup v\} & \text{if } m \notin M^* \wedge \\ & mightReturn[h^\#](o^\#, m.\alpha_{ret}) \\ \{o^\# \setminus v\} & \text{otherwise} \end{cases} \\
\llbracket s \rrbracket_{ret}^2 &\triangleq \begin{cases} \{\{v\}\} & \text{if } s = CallFlow[m, v] \wedge m \notin M^* \wedge \\ & m.\alpha_{ret}.fresh \\ \emptyset & \text{otherwise} \end{cases} \\
\llbracket s \rrbracket_{O^\#}^2[h^\#](O^\#) &\triangleq \begin{cases} \bigcup_{o^\# \in O^\#} \llbracket s \rrbracket_{o^\#}^1(o^\#) \\ & \text{if } s \notin \{v \leftarrow \mathbf{h}, CallFlow[m, v]\} \\ \bigcup_{o^\# \in O^\#} CallFlow_{o^\#}^2[h^\#, m, v](o^\#) \\ & \text{if } s = CallFlow[m, v] \\ \bigcup_{o^\# \in O^\#} focus[h^\#](v, o^\#) & \text{if } s = v \leftarrow \mathbf{h} \end{cases} \\
\llbracket s \rrbracket_{\rho^\#}^2(\rho^\#, h^\#) &\triangleq \llbracket s \rrbracket_{gen}^1 \cup \llbracket s \rrbracket_{ret}^2 \cup \llbracket s \rrbracket_{O^\#}^1[h^\#](\rho^\#) \\
escape(\rho^\#, h^\#, m) &\triangleq h^\# \cup \{o^\# \in \rho^\# : \exists p, p \in o^\# : p \in \bar{r}(m.\alpha_{esc}.params)\} \\
\llbracket s \rrbracket_{esc}^2(m) &\triangleq \begin{cases} \{\{v\}\} & \text{if } m.\alpha_{ret}.fresh \wedge m.\alpha_{ret}.escaped \\ \emptyset & \text{otherwise} \end{cases} \\
\llbracket s \rrbracket_{h^\#}^2(\rho^\#, h^\#) &\triangleq \begin{cases} \llbracket s \rrbracket_{O^\#}^2[h^\#](h^\# \cup \{o^\# \in \rho^\# : v \in o^\#\}) & \text{if } s = \mathbf{h} \leftarrow v \\ \llbracket s \rrbracket_{esc}^2(m) \cup \llbracket s \rrbracket_{O^\#}^2[h^\#]escape(\rho^\#, h^\#, m) \\ & \text{if } s = CallFlow[m, v] \wedge m \notin M^* \\ \llbracket s \rrbracket_{O^\#}^2h^\# & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 6.11: Modified transfer functions for the alias set analysis using callee summaries.

function `mustReturn` is true only when the object represented by o^\sharp before the call is returned by the callee. Therefore the null, fresh and heap flags of the return value summary should be false since a non-null object, which was not allocated in the callee nor loaded from the heap, should be returned. Additionally, o^\sharp must contain the corresponding arguments of all parameters that might be returned by the callee. Parameters that might be returned are given by $\alpha_{ret}.params$ and the corresponding arguments are retrieved through the inverse function \bar{r} , where r is the function mapping arguments to parameters.

The helper function `mightReturn` determines whether o^\sharp might be returned. This is true if o^\sharp represents an escaped object ($o^\sharp \in h^\sharp$) and an object loaded from the heap might be returned. An object representing o^\sharp might also be returned if at least one parameter, whose corresponding argument is in o^\sharp , might be returned. To handle the uncertainty when `mightReturn` is true, *CallFlow* accounts for both possibilities similarly to the *focus* operation. If the returned object must not be o^\sharp then the variable assigned from the return of the callee cannot possibly point to o^\sharp after the call.

The callee escape summary α_{esc} is utilized to update alias sets representing objects that might escape due to the function call (the function *escape* in Figure 6.11). If any of the corresponding arguments to parameters that escape ($\alpha_{esc}.params$) are in an alias set in ρ^\sharp , the alias set is added to h^\sharp since the function call escapes the parameter. The transfer function must also handle situations when the callee allocates and returns a new object. If $\alpha_{ret}.fresh$ is true and the return from the callee is assigned to variable v , an alias set containing only v is added to ρ^\sharp . If the freshly created object might have been stored in the heap before being returned ($\alpha_{ret}.escaped$ is true) a similar alias set is added to h^\sharp .

6.3 Caller Summaries

In this section we present caller summaries as a mechanism to speed up the interprocedural context-sensitive alias set analysis. Although static analyses that infer properties of pointers can be useful even when the analysis is carried out locally on individual methods, such analyses shine most when computed interprocedurally. The added ability to carry forward computed pointer and aliasing information from a caller into a callee by mapping arguments to parameters can significantly improve precision. However, when efficiency is a bigger concern, a natural trade-off is to forego some precision by conservatively assuming initial pointer and aliasing relationships for the parameters of a method.

Using caller summaries improves efficiency because it decreases the number of the methods that must be analyzed flow-sensitively. Let us revisit the example in Figure 6.1.

Using callee summaries enables the alias set analysis to discard flow-sensitive analysis of methods 3-6 since they do not contain any shadows. However, even though methods 1 and 7 do not contain shadows, they are analyzed flow-sensitively to ensure that at a callsite to a method containing a shadow, precise information can be mapped into the callee. In an analysis that uses caller summaries to make conservative assumptions at every method entry, flow-sensitively analyzing such methods is un-needed since the precise information computed at the callsite will never be propagated into the callee.

We have implemented a conservative mechanism for computing caller summaries. In our summaries, initial alias sets are created for the parameters of a method such that the abstraction at the start of the method specifies that any two parameters might be aliased. In our intermediate representation the method `bar` in the example from Figure 6.1 has three parameters; the `this` receiver and the two `List` references `a` and `b`. The caller summary for this method contains the following sets: $\{\}$, $\{\text{this}\}$, $\{a\}$, $\{b\}$, $\{\text{this},a\}$, $\{\text{this},b\}$, $\{a,b\}$ and $\{\text{this},a,b\}$. Notice that given these alias sets, the only conclusion that can be drawn is that the three parameters might be aliased, *.*, no must or must-not relationships exist between the parameters. This is overly conservative. First, the caller summary does not take into account any type information. Although `a` and `b` are both references to a `List` data structure, the receiver `this` is of type `Foo` and, unless `Foo` is declared a supertype of `List`, a reference of type `Foo` can never point to a `List` object. Second, the caller summaries do not leverage any pointer information. For example, subset-based points-to analyses that use allocation sites as their object abstraction are often performed at onset for constructing a callgraph. Using this type of pointer analysis could potentially improve the precision of the caller summary in situations where the pointer analysis can specify that parameters `a` and `b` were created at different allocation sites.

Our reason for using a naive caller summary was to investigate the maximum precision degradation due to such summaries. Whereas the callee summaries presented in the preceding section do not affect precision, caller summaries do. As an example, let us look more closely at the example in Figure 6.1. The method `bar` receives two `List` references, `a` and `b`. An alias set analysis which does not utilize caller summaries is able to differentiate between the two references. In particular, at the start of method `bar` the analysis infers that `a` and `b` must-not alias (two separate lists were created at lines 1 and 3 and assigned to `a` and `b` respectively, and a reference of one is never copied to the other). However, the naive caller summary assumes that `a` and `b` could be aliased. Hence the precision of the alias set analysis degrades, *i.e.*, fewer must-not facts are computed.

This decrease in precision can cascade into client analyses. For example suppose a client of the alias set analysis is a verification tool for the property that an iterator's underlying list structure has not been modified when its `next` method is invoked (executing such

code results in a runtime exception). If caller summaries are not used, the analysis infers that the iterator’s underlying list, i.e., the list referenced by a , is never modified since a and b must-not point to the same object and the code only modifies the list referenced by b . Hence, the client analysis can prove that line 10 is not a violation of the property. However, when caller summaries are used, the client analysis infers that the list pointed to by reference a might be modified (the caller summary suggests that a and b might be aliased and an element is added at line 11 to the list pointed to by reference b). Hence the client analysis loses precision since it can no longer prove that the `next` operation at line 10 is safe w.r.t. the property being verified. We empirically evaluate the loss in precision of using caller summaries on the alias set analysis and a client analysis in Section 6.4.

$$\begin{aligned}
\llbracket call \rrbracket_{o^\#}^3(o^\#) &\triangleq \text{callerSummaries}(\text{target}(\text{call})) \\
\text{CallFlow}_{o^\#}^3[h^\#, m, v](o^\#) &\triangleq \begin{cases} \{o^\# \cup v\} & \text{if } \text{mustReturn}(o^\#, m.\alpha_{ret}) \\ \{o^\# \setminus v, o^\# \cup v\} & \text{if } \text{mightReturn}[h^\#](o^\#, m.\alpha_{ret}) \\ \{o^\# \setminus v\} & \text{otherwise} \end{cases} \\
\llbracket s \rrbracket_{h^\#}^3(\rho^\#, h^\#) &\triangleq \begin{cases} \llbracket s \rrbracket_{O^\#}^2[h^\#](h^\# \cup \{o^\# \in \rho^\# : v \in o^\#\}) & \text{if } s = e \leftarrow v \\ \llbracket s \rrbracket_{\text{esc}}^2(m) \cup \llbracket s \rrbracket_{O^\#}^2[h^\#]\text{escape}(\rho^\#, h^\#, m) & \text{if } s = \text{CallFlow}[m, v] \\ \llbracket s \rrbracket_{O^\#}^2h^\# & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 6.12: Transfer functions using callee and caller summaries.

Modifying the alias set analysis to use caller summaries is straightforward. Figure 6.12 shows those transfer functions which have been modified from their earlier version (Figure 6.11). First, the `call` function is modified. Instead of mapping arguments to parameters, the caller summary provides the set of alias sets to seed the callee’s analysis. Second, callee summaries are used for all methods that do not contain shadows.

6.4 Experiments

In this section, we discuss experimental results in evaluating the effect of using callee and caller summaries. The questions that we wanted to answer were:

- What is the effect of using callee and caller summaries on the running time of the analysis?

- What effect does the use of caller summaries have on the precision of the alias set analysis?
- Does the resulting decrease in precision of the alias set analysis due to the use of summaries degrade the precision of the client analysis? And if yes, by how much?

Our experiments were conducted using the same benchmark suite and tracematches as discussed in Section 5.2, where we discussed the precision of our analysis. To give an indication of the size of these benchmarks, we computed the number of methods statically reachable in the control flow graph created by Soot and present these in Table 6.1. Time taken to pre-compute the callee summaries using the algorithms discussed in Section 6.2 is also shown.

Benchmark	antlr	bloat	chart	fop	hsqldb	jython	luindex	lusearch	pmd	xalan
Reachable Methods	4452	5955	14912	27408	11418	14437	7358	7821	9365	14961
Callee SummaryTime(s)	8	12	29	72	28	50	10	10	15	29

Table 6.1: Number of statically reachable methods and the time to precompute callee summaries.

The tracematch analysis is an ideal analysis for evaluating the effect of callee and caller summaries on the alias set analysis. First, each temporal property specifies its own points of interest; only events that transition the state machine of that property are considered shadows. By choosing different properties, we ensure a varying set M^* , the set of methods for which callee summaries are used. Second, the tracematch analysis cleanly teases apart the computation of the alias set abstraction and its use in computing the state abstraction. This enables us to measure the precision and efficiency of the alias set abstraction in a real-world scenario.

We call a pair containing a benchmark and temporal property a *test case*. Since not all benchmarks exercise all temporal properties, we have chosen to present results only for test cases when a temporal property is applicable for a benchmark e.g. the antlr benchmark never uses a Writer and hence the corresponding temporal property is inapplicable.

6.4.1 Shadow Statistics

In Section 6.2 we proposed the use of callee summaries for methods not in M^* . Later, in Section 6.3, we proposed the use of caller summaries for all methods, thereby requiring flow-sensitive analysis of only methods containing shadows (S). We measured the percentage of

reachable methods that are in M^* and S and present these in Table 6.2. The maximum percentage of methods in M^* was for the test case jython-FSI, where 59.9% of the methods are in M^* . Notice that only 0.6% of the methods for jython-FSI contain shadows, indicating that most methods are in M^* since they call methods containing shadows. On average (geometric mean), M^* contains 11.9% of the reachable methods implying that callee summaries are used for the remaining 88.1%. When using both callee and caller summaries, a mere 0.3% of reachable methods (average of set S) require flow-sensitive analysis.

	antlr		bloat		chart		fop		hsqldb	
	M^*	S	M^*	S	M^*	S	M^*	S	M^*	S
FSE	56.3	0.7					47.6	0.1	1.6	0.1
FSEH	56.3	1.1							2.8	0.1
FSI			56.3	4.2	50.6	0.6	47.7	0.5	54.0	0.1
HN			56.0	2.6	50.5	0.4	47.6	0.1	54.0	0.1
HNE	56.3	0.7	0.1	0.1					1.6	0.1
R	7.5	0.2							4.2	0.3
W			55.8	0.5			47.6	0.3	5.7	0.6
	jython		luindex		lusearch		pmd		xalan	
	M^*	S	M^*	S	M^*	S	M^*	S	M^*	S
FSE	59.8	0.4	1.6	0.4	1.1	0.2	9.5	0.1	50.0	0.6
FSEH	59.8	0.4	1.1	0.3	0.8	0.2			49.9	0.3
FSI	59.9	0.6	53.1	0.4	52.4	0.6	52.4	1.0	50.0	0.5
HN	59.8	0.2	53.1	0.2	52.3	0.3	52.2	0.5	0.1	0.1
HNE	59.8	0.2	0.5	0.2	0.3	0.1	6.6	0.1	49.9	0.2
R	59.8	0.2	0.9	0.1	2.3	0.3	7.7	0.1	49.9	0.1
W			0.8	0.1	1.7	0.3	0.2	0.1	49.9	0.4

Table 6.2: Percentage of reachable methods that contain shadows or transitively call methods with shadows (M^*) and methods that contain shadows (S)

6.4.2 Efficiency

To measure the effect of summaries on the time required to compute the alias set abstraction, we computed the abstractions using the three versions of the transfer functions. In Table 6.3, we show the running time of the original alias set abstraction (ORIG), the alias set abstraction which uses only callee summaries (CS) and the abstraction using both callee

and caller summaries (CCS). The times for CS and CCS include the time for computing the callee summary and CCS also includes the time to compute the caller summary.

For all test cases, the time required to compute the abstraction is reduced when callee summaries are used for methods not in M^* . The greatest reduction is for pmd-W, which takes 99.6% less time to compute (6670 vs 29 seconds). The reason for this is quite obvious; for pmd-W, M^* contains only 12 methods out of the 9365 reachable methods. On average, the use of callee summaries reduces the time to compute the alias set abstraction by 27%. Introducing caller summaries has a more significant impact; an average reduction of 96% is witnessed over the entire test set.

	antlr	bloat	chart	fop	hsqldb	jython	luindex	lusearch	pmd	xalan
FSE-ORIG	484			5934	1351	2390	1017	580	2638	5052
FSE-CS	349			5422	220	1524	118	151	37	4484
FSE-CCS	15			108	40	71	19	18	26	112
FSEH-ORIG	500				1426	2020	1054	576		6395
FSEH-CS	386				226	1397	120	133		5834
FSEH-CCS	18				39	77	17	17		51
FSI-ORIG		1810	1653	4057	1316	2335	1057	553	3685	5100
FSI-CS		1683	1022	4051	651	1671	391	407	2069	5099
FSI-CCS		563	72	109	37	69	17	17	29	119
HN-ORIG		1601	1665	3735	1406	2225	1019	485	5273	5262
HN-CS		1556	1035	3289	714	1390	393	388	5031	164
HN-CCS		455	44	115	41	70	18	17	29	45
HNE-ORIG	457	1607			1450	2233	1098	562	5182	4361
HNE-CS	358	119			220	1481	137	121	32	3588
HNE-CCS	16	18			40	77	18	17	26	44
R-ORIG	511				1416	2205	1097	563	4348	3280
R-CS	55				238	1487	123	123	35	3172
R-CCS	13				39	73	16	16	27	48
W-ORIG		1551		3840	1450		1067	607	6670	3468
W-CS		1411		3553	318		120	146	29	3323
W-CCS		19		447	37		17	18	28	66

Table 6.3: Time taken to compute the alias set abstraction for the original transfer functions (ORIG), the transfer functions leveraging Callee Summaries (CS) and the transfer functions employing both Callee and Caller Summaries (CCS)

6.4.3 Tracematch Analysis Precision

Callee summaries do not affect the precision of the tracematch analysis. To evaluate the effect of using caller summaries on the precision of the tracematch analysis, we executed the tracematch analysis using the ORIG and CCS abstractions for the alias set analysis. As per our discussion in Section 6.3, we expected a decrease in precision since caller summaries cause the alias set abstraction to compute fewer aliasing facts. However, the results surprised us; none of the 54 test cases showed any degradation in the tracematch analysis and the results were exactly the same as discussed in Section 5.2 and illustrated in Figure 5.1. The CCS abstraction contained sufficient must and must-not aliasing at each shadow of a test case to produce the same transitions on the abstract state machine.

Our conclusion from this experiment is that even though caller summaries cause a theoretical decrease in precision, this does not automatically translate into precision loss for the client analysis. The tracematch analysis is one such example where the benefits of using caller summaries outweigh the slight chance of losing precision.

6.4.4 Fine-grained Precision Metrics

Although the tracematch analysis did not show a loss of precision, we know that caller summaries have the potential of reducing precision of the alias set analysis e.g. they prevent any must-not alias information to be forwarded into a method from a callsite. To measure this loss of precision, we developed a fine-grained metric for evaluating precision of alias sets. Using the alias set abstraction, we compute must and must-not alias pairs for variables live at the shadows of each test case. Then we sum the alias pairs for all shadows in a test case to give us two precision metrics: MA the aggregated must-alias pairs and MNA the aggregated must-not alias pairs. As expected, the metric values for ORIG and CS are identical indicating that no precision is lost by using callee summaries. Table 6.4 presents the results of ORIG (alternately CS) vs CCS. For each test case, the MA and MNA values for ORIG are presented. Below this is a number indicating the number of alias pairs that are lost with CCS. For example, the MA value for jython-FSE is 51 indicating that 51 different alias-pairs were identified at the shadows of this test case. The absence of a number below indicates no decrease in precision when using caller summaries. The MNA value for jython-FSE is 189. The -7 below indicates that 7 must-not alias pairs were lost when caller summaries were used.

Of the 54 test cases, only 9 showed a degradation in the MA precision metric. The four highest degradations were for luindex-R (75%), luindex-W (73%), lusearch-R (22%) and

	antlr		bloat		chart		fop		hsqldb	
	MA	MNA	MA	MNA	MA	MNA	MA	MNA	MA	MNA
FSE	1	81					57	141	0	21
FSEH	1	80							0	21
FSI			1152	18858 -611	3344	6244 -212 -254	357	876 -28	0	21
HN			606	7584 -328	704	1529 -14 -214	136	336 -26	0	21
HNE	2	135	0	21					0	21
R	133	338							46	189 -7
W			7	306 -55			53	439 -12	163	339 -4
	jython		luindex		lusearch		pmd		xalan	
	MA	MNA	MA	MNA	MA	MNA	MA	MNA	MA	MNA
FSE	51	189 -7	6	98 -2	18	91 -1	0	35 -1	371	1180 -64
FSEH	930 -21	1546 -19	4	63 -4	1	17			179	384 -2
FSI	404	583 -32	76	226 -1	77	233 -1	459 -3	1505 -79	350	1042 -22
HN	322	386	95	202	58	112 -1	127	839 -53	0	13
HNE	11	133 -10	8	91 -2	0	13	0	41	45	194 -1
R	253 -9	427 -14	59 -44	80	203 -44	222	56	130 -7	671	1395 -20
W			56 -41	83	200 -41	219	0	22	524	799 -21

Table 6.4: Alias set abstraction precision in terms of aggregated must aliasing (MA) and must not aliasing (MNA) metrics computed at the shadows for each test case.

lusearch-W (21%). The average (geometric mean) degradation for the 9 test cases was 8%. 31 of the 54 test cases also noted a decrease in the MNA metric. The maximum decrease was 17% for bloat-W with an average decrease of 4%.

6.5 Related Work

As discussed in Chapter 4, the IFDS and IDE algorithms are an example of Sharir and Pnueli’s [62] functional approach for context-sensitive interprocedural dataflow analysis. The crux of this approach is that the effect of a procedure is computed by composing functions representing the effect on individual instructions in the procedure. Once a procedure’s summary has been computed, it is used at each call site of the procedure to model the effect of the call.

Whole-program analysis of even relatively small applications can be expensive since the entire program, including the library, must be analyzed. A possible optimization is to not analyze the library and instead use pre-computed summaries. If a library did not call back into the application, the IFDS and IDE algorithms could be used to pre-compute such summaries. This would enable a whole-program analysis to just analyze the application and use the pre-computed summaries for library procedures. Rountev et al. [57] present a framework to summarize the effects of libraries even in the presence of call backs. Their extension to the IFDS and IDE algorithms to enable pre-analyzing the library independently of any application code. For a library procedure p that contains a call back to a procedure n , the extended algorithm summarizes the effect of p by computing a summary from the start node of p to the call site for n and then from the return site of the call to n to the exit node of p . The missing portion, the effect of n , is computed when the application code becomes available. This enables efficient whole-program analysis since the analysis of an application can use pre-computed summaries for a library. Building on this, Rountev et al. [58] evaluated the use of such summaries on a client analysis expressed as an IDE problem. Their experiments showed a 51% saving in time and a 33% saving in memory.

Other frameworks for computing procedure summaries have also been proposed. Gulwani and Tiwari [33] developed procedure summaries in the form of constraints that must be satisfied for some generic assertion to hold at the end of the procedure. Their key insight was to use weakest preconditions of such generic assertions. Furthermore, for efficiency, they used strengthening and simplification of these preconditions to ensure early termination. The approach has been used to compute two useful abstractions; unary uninterpreted functions and linear arithmetic. Recently, Yorsh et al. [70] introduced an algorithm which

also computes weakest preconditions and relies on simplification for termination. They describe a class of complex abstract domains (including the class of problems solvable using IFDS) for which they can generate concise and precise procedure summaries. Their approach uses symbolic composition of the transfer functions for the instructions in the program to obtain a compact representation for the possibly infinite calling contexts.

In contrast to the related work discussed above, we propose a technique to reduce the number of methods that must be analyzed using any of the approaches discussed above (our implementation uses the IFDS algorithm [54] to compute the alias set abstraction). Under certain conditions, instead of computing expensive procedure summaries through IFDS, our analysis uses cheaper callee summaries without a loss of precision.

Cherem and Rugina [18] present a flow-insensitive, unification-based context-sensitive analysis to construct method summaries that describe heap effects. The analysis is parameterized for specifying the depth of the heap to analyze (k) and the number of fields to track per object (b). Varying the values for k and b results in different method summaries; smaller values produce lightweight summaries whereas larger values result in increased precision. Method summaries were shown to significantly improve a client analysis that infers uniqueness of variables i.e. when a variable holds the only reference to an object.

Also related are analyses which traverse the program callgraph (mostly bottom-up but some top-down analyses have also been proposed) and compute a summary function for each procedure [68, 14, 16]. This summary function is then used when analyzing the callers.

Escape analysis has been widely studied [19, 11, 1, 68] and used in a variety of applications ranging from allocating objects on the stack to eliminating unnecessary synchronization in Java programs. To determine whether an object can be allocated on the stack and whether it is accessed by a single thread, Choi et al. [19] compute object escape information using *connected graphs*. A connected graph summarizes a method and helps identify non-escaping objects in different calling contexts. In their work on inferring aliasing and encapsulation properties for Java [46], Ma and Foster present a static analysis for demand-driven predicate inference. Their analysis computes predicates such as checking for uniqueness of pointers (only reference to an object), parameters that are lent (callee does not change uniqueness) and those that do not escape a callee.

6.6 Concluding Remarks

This chapter presented callee and caller summaries as a means to improve the efficiency of the alias set analysis. We described the information required from a callee summary

to ensure that their use does not decrease precision at a callsite. Through experimental evidence, we showed that a client analysis and alias set precision metrics are unaffected by the use of callee summaries. On average a 27% reduction in the running time to compute the abstraction was witnessed.

In situations where some loss of precision is acceptable in favour of larger gains in efficiency, we showed how caller summaries that make assumptions about pointer and aliasing relationships at method entry can be employed. In order to gauge the maximum decrease in precision, we chose to use a conservative caller summary which assumes that any two parameters of a method might be aliased. Empirical evaluation of the effect of using caller summaries on the precision of the client analysis revealed no decrease in the abilities of the client analysis. For a fine-grained evaluation of precision, two metrics deriving aggregated must and must-not aliasing between variables were calculated. The average decrease was 8% for the must- and 4% for the must-not alias metric. The running time for computing the alias set abstraction decreases by 96% on average if both callee and caller summaries are used.

Chapter 7

Presenting Analysis Output

7.1 Motivation

The IFDS-based analysis attempts to prove that a program does not violate a property by determining that the state machine representing the property never reaches an accepting state. However, in case the analysis is not successful in ruling out all violations, it outputs a list of potential violations. Each violation is a transition statement which, based on the transitions already taken, would transition the state machine to an accepting state. Although finding these violations is useful as it gives an indication of how many possible violations there are in the program, more useful is obtaining a list of transition statements that may contribute to a match at the final transition statement. The IDE analysis of Section 5.1 computes this list for each possible violation.

In our experience, interpreting the raw results of the verification analysis is difficult since all we have is a list of possible violations, and for each violation a list of transition statements that led to the violation. These transition statements are not necessarily within one method and can be interspersed in a large segment of the program. To make the analysis practical for use during the development life cycle, we found it important to develop a Graphical User Interface (GUI) to run the analysis and visualize the output. Another reason for developing the TMEclipse plugin is that the analysis can be configured in many ways. Again, for practicality of use, it is useful to be able to configure the analysis via a GUI rather than having to deal with many command line options and parameters. In this chapter, we highlight the tool that we created for configuring, running and analyzing the output from our verification analysis.

7.2 TMAalysis: an Eclipse plugin

We developed the verification tool as a plugin to the Eclipse IDE. The tool provides a configuration screen, progress indicator and a visualization window. We highlight key features of these functionalities in the following three sections:

7.2.1 Configuration

Figure 7.1 shows the configuration panel which opens when the developer selects the Plugin’s Configure option. The developer can choose any of the projects open within their Eclipse workspace from a drop down list. The developer must specify the main class for the project. This is used by the analysis to determine the starting point of the program (in order to generate the callgraph). The next input field (Choose Application Classes) allows the user to select which portions of the code are to be verified (i.e. belong to the application).

Since Java programs can use reflection to load classes, the configure panel provides the Dynamic Class List field to specify which, if any, classes are loaded via reflection. The field expects a path to a file which contains each of the dynamically loaded classes in a separate line. In Section 5.2 we mentioned our use of Probe and *J to record actual uses of reflection. These uses can be stored in a file and supplied to the analysis via the Dynamic Class List field.

The next input field is a drop down list to select a tracematch. The plugin only supports verifying one property at a time and hence the list allows selecting one tracematch. The tracematches discussed in this work (Section 5.2) come built-in with the plugin. For these built-in tracematches a description page that provides the actual code for the tracematch, a graphical representation of the state machine and a description of the tracematch are provided. Developers can also specify their own property by choosing the “Create Custom Tracematch” option from the drop-down list and specifying the location of a Java file containing the tracematch specification using AspectJ syntax.

The output folder field lets developers specify a place where the analysis outputs the results in XML format. The Output XML field lets the developer specify the name of the output file that will be generated.

The default behaviour of the tool is to first execute the IFDS-based analysis which determines whether there are any violations of the property in the program. If any violations are found, then the IDE-based analysis is automatically executed to find the transition

statements that contribute to each violation. If the developer's intent is only to verify the program for conformance to a selected tracematch, then, to save time, there is no need to run the IDE analysis. The tool provides a checkbox which can be unchecked to disable the IDE-analysis. In this case the tool only executes the IFDS-based analysis and therefore functions purely as a verification tool .

The additional options field supports a number of checkbox fields that can be used to modify the behaviour of the analysis. Most are useful for debugging the analysis. More

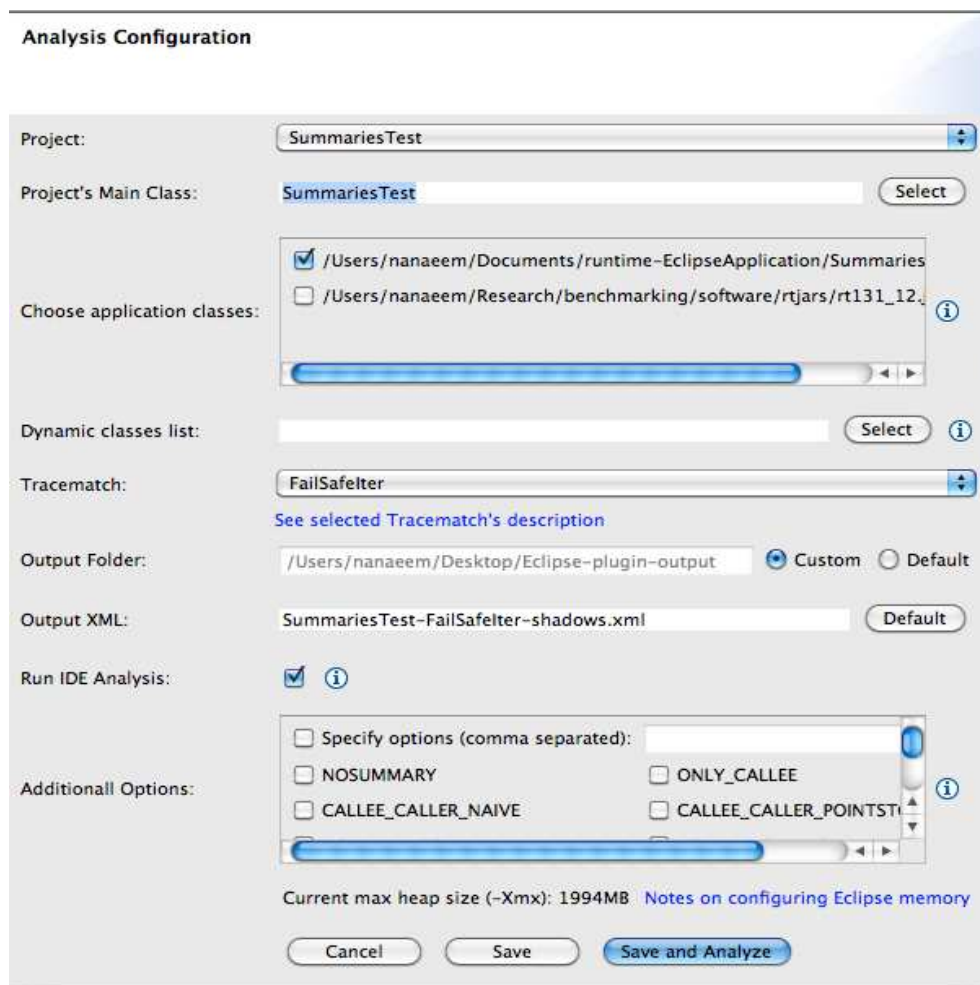


Figure 7.1: Configuration Screen

details can be found by pressing the information button next to the input field on the configure screen.

Using summaries for the object abstraction analysis

These options are used to control which method summaries approaches (that were discussed in Chapter 6) are used.

NOSUMMARY: This option disables the use of any summary causing all methods to be analyzed flow-sensitively.

ONLY_CALLEE: This option enables the use of callee summaries but not caller summaries. Therefore, only methods that contain shadows or call methods that contain shadows are analyzed flow-sensitively.

CALLEE_CALLER_NAIVE: This option enables the use of both callee and caller summaries. Therefore, methods that call methods containing shadows are not analyzed flow-sensitively. Additionally, methods that contain shadows are analyzed flow-sensitively with an initial naive approximation for the parameters of the method.

For a developer, selecting one of `ONLY_CALLEE` or `CALLEE_CALLER_NAIVE` is the only useful option. Since running the analysis with no summaries and with only callee summaries has the same precision guarantees, there is no reason not to use callee summaries. When precision is of utmost importance, it is good to use the `ONLY_CALLEE` option since `CALLEE_CALLER_NAIVE` does cause some loss of precision in the alias set abstraction.

The options also serve for debugging the alias set analysis. The key use is that multiple summary options can be selected within the same run. This causes the tool to run the Alias Set Analysis (object analysis) multiple times, once for each type of summary. Once all abstractions have been computed, the tool will output to the Console, the aggregated must- and must-not alias inferences that were made. This is useful to compare the precision of the alias set abstraction created by using (or not using) summaries. For example, to empirically see that the use of callee summaries does not effect the abstraction's precision, one can select both `NOSUMMARY` and `ONLY_CALLEE`. The aggregated must- and must-not alias inferences will be identical for the two abstractions.

Computing the state abstraction

As discussed in Chapter 3, the analysis relies on two abstractions: abstracting the objects in the program and the abstraction of the state that these objects are in. By default the state abstraction is computed for each type of object abstraction that is computed (depending on the summary options mentioned above). This is a useful debugging feature

so that the effect on the precision of the tracematch analysis can be monitored with respect to the use of different summaries. However, if this is not the intent, then the following options can be used to disable the state abstraction computations. For finer control, three separate debugging options are provided which individually turn off the state computation for each of the summary options discussed above.

NOT_TM_NOSUMMARY: This disables the computation of the state abstraction using the object abstraction that did not use any summaries.

NOT_TM_ONLY_CALLEE: This disables the computation of the state abstraction using the object abstraction that only used callee summaries.

NOT_TM_CALLEE_CALLER_NAIVE: This disables the computation of the the state abstraction using the object abstraction that used the callee and the naive caller summaries.

For example running the tool with the options `NOSUMMARY`, `ONLY_CALLEE` and `CALLEE_CALLER_NAIVE` allows us to infer if the alias-set abstraction for `NOSUMMARY` and `ONLY_CALLEE` are the same (as discussed above). Since this is always going to be true, then there is no need to separately compute the state abstractions for both the `NOSUMMARY` and `ONLY_CALLEE` object abstractions (they are going to be the same). We can additionally set the `NOT_TM_NOSUMMARY` option. The analysis will then compute the state abstraction only twice; once using the object abstraction when using callee summaries only and once using the object abstraction that used both summaries. When multiple state abstractions are computed, the analysis outputs to the console the total number of violations found for each abstraction. This is useful for debugging the effect of summaries on a client analysis. To only compute the object abstraction and no state abstraction, the `NOT_TM_*` counterparts to the summary options must be selected.

Intraprocedural Control Flow Graph for Debugging

The `OUTPUTDOT` option can be selected to output DOT files for all methods containing useful shadows for each type of object analysis (using different summaries options). Similar output is generated for IFDS and IDE analysis. The destination is the specified output folder. DOT is a plain text graph description language that is especially useful for describing directed graphs. The tool produces a separate DOT file for each method and contains the description of the control flow graph for that method. Each instruction in the method appears as a node and outbound edges from a node represent control flow successors. Additionally, each node is decorated with useful debug information such as alias-sets at that node and any bindings for the state abstraction. The DOT file can be processed by any program that understands the DOT language e.g. `dot` from the Graphviz package, `GVEdit` etc. to render the graph.

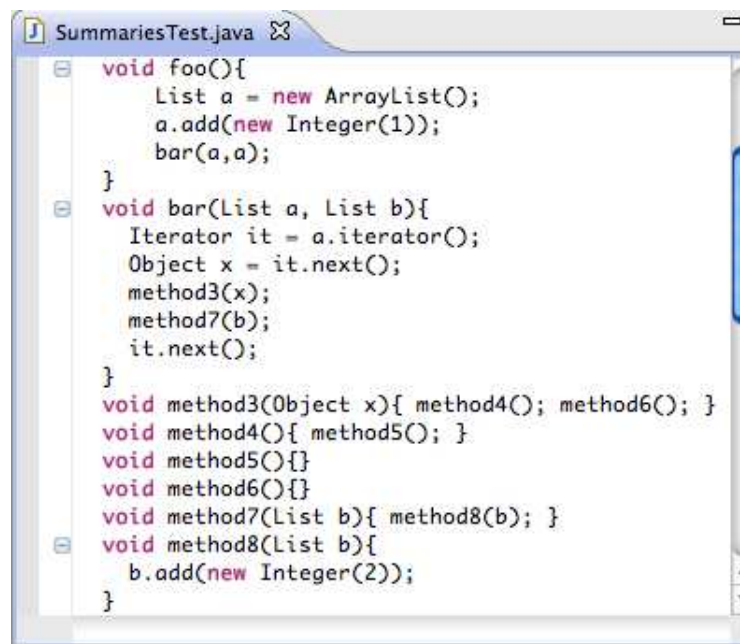
7.2.2 Running the Analysis

When the “Save and Analyze” button on the Configure screen is pressed, the current configuration is saved and the analysis executed. Depending on the options selected, the analysis can take a considerable time to complete. To ensure that the tool does not become unresponsive, a “Run in Background” option is provided which allows the user to continue to work within the Eclipse IDE while the analysis executes as a background task. A Progress Indicator displays the current stage of the analysis. Once the analysis finishes, the user is notified by opening the Tracematch Analysis View discussed below.

7.2.3 Visualization of Results

The visualization component displays a list of possible violations (body shadows) that the analysis has identified. Each violation can be selected to display a list of transition statements (update shadows) that contribute to a possible match.

To illustrate the visualization, we use the example discussed in Figure 6.1, but with



```
SummariesTest.java
void foo(){
    List a = new ArrayList();
    a.add(new Integer(1));
    bar(a,a);
}
void bar(List a, List b){
    Iterator it = a.iterator();
    Object x = it.next();
    method3(x);
    method7(b);
    it.next();
}
void method3(Object x){ method4(); method6(); }
void method4(){ method5(); }
void method5(){
}
void method6(){
}
void method7(List b){ method8(b); }
void method8(List b){
    b.add(new Integer(2));
}
```

Figure 7.2: Example Input

one change. The original example illustrated the benefits of using summaries and did not violate the safe iteration property discussed in Section 1.1. To highlight the tool’s capabilities, we intentionally modify the example to introduce a violation by replacing Line 4 in the code from Figure 6.1 with the function call `bar(a,a)`, i.e., the function `bar` is called with the two parameters aliased. The actual code is shown in Figure 7.2. Even for this contrived example, it is non-trivial to trace this violation. The violation can be determined by tracing the `bar` function. `bar` calls `method7` with `b` as an argument. `Method7` then calls `method8` which updates the list. When the call to `method7` from within `bar` returns, the iterator for the list pointed to by `a` is incremented (via `next`). Since `a` and `b` were aliased, this violates the property that the underlying data structure should not be updated while iterating over it.

The analysis is able to detect the violation and provides an easy way to visualize the result. The visualization comprises of two parts: annotations within the editor window and a custom view. We show the analysis supplied annotations for the above example in Figure 7.3. Transition statements relevant to the possible violation appear highlighted and a marker appears on the left border for ease of navigation. The custom view is shown in

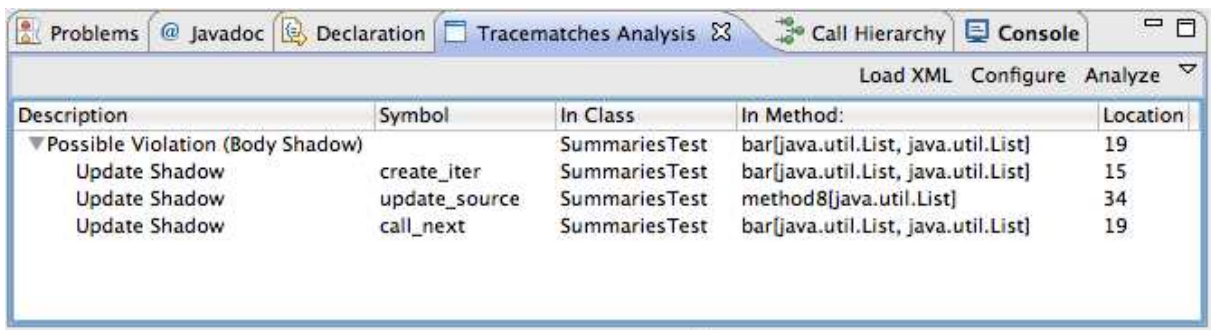
```

SummariesTest.java
void foo(){
    List a = new ArrayList();
    a.add(new Integer(1));
    bar(a,a);
}
void bar(List a, List b){
    TA Iterator it = a.iterator();
    Object x = it.next();
    method3(x);
    method7(b);
    TA it.next();
}
void method3(Object x){ method4(); method6(); }
void method4(){ method5(); }
void method5(){
}
void method6(){
}
void method7(List b){ method8(b); }
void method8(List b){
    TA b.add(new Integer(2));
}

```

Figure 7.3: Example Input

Figure 7.4. The view is a table with five columns: Description, Symbol, Class, Method and location. Each of the possible violations are listed with the name of the class and method the statement occurs in and the offending line of code. Clicking on any of these body shadows opens a sub-level containing the transition elements that contribute to this violation. For the example, there is one possible violation which occurs at line 19 within the SummariesTest file in method `bar`. This corresponds to the invocation of the `next` method on `it`. The sequence of events that lead to this violation are a match of the symbol `create_iter` in method `bar` at line 15, followed by the symbol `update_source` in `method8` at line 34 and followed by the symbol `next` at line 19 in method `bar`.



Description	Symbol	In Class	In Method:	Location
▼ Possible Violation (Body Shadow)		SummariesTest	bar[java.util.List, java.util.List]	19
Update Shadow	create_iter	SummariesTest	bar[java.util.List, java.util.List]	15
Update Shadow	update_source	SummariesTest	method8[java.util.List]	34
Update Shadow	call_next	SummariesTest	bar[java.util.List, java.util.List]	19

Figure 7.4: Visualization Screen

7.3 Conclusion

This chapter presented a verification tool that we developed to configure and execute the analysis to verify temporal properties of programs. The configuration screen allows users to easily configure the analysis and specify which property to verify on which program. The tool has the ability to run in the background, thereby releasing the Eclipse IDE for use by the developer during execution. A list-based and a source-code based visualization of results are provided. All violations and related transition statements are presented in the custom list view. Each entry in this list can be clicked to reach the relevant line in the source code editor. Additionally, relevant source code lines are highlighted and marked for ease of navigation and visualization.

Chapter 8

Conclusion and Future Work

Verifying the conformance of a program to specified usage requirements is important to ensure the reliability of the software. Requirements are often in the form of sequences of operations that must (or must not) be performed. These requirements often involve multiple objects that interact with each other. We have designed static abstractions for the objects in the program and the state they are in. Additionally, we have leveraged inter-procedural, context-sensitive dataflow analysis algorithms to compute these abstractions to determine if a program violates specified properties. Furthermore, we have significantly improved the performance of the analysis by designing and employing two types of method summaries. Finally, we have packaged the static analysis as an Eclipse plug-in that provides developers with an easy-to-use temporal property verification tool.

8.1 Abstractions

We have presented two abstractions that can be used to analyze temporal specifications of multiple interacting objects expressed via tracematches. The first abstraction models individual objects using a storeless heap abstraction. This alias-set abstraction provides precise may-alias information and flow-sensitive tracking of individual objects along control flow paths. The second abstraction models the tracematch state for a group of related objects. The abstraction can be viewed as a tuple containing the state and the abstract objects corresponding to the runtime objects that would cause the tracematch to be in the given state. Transition statements (events of interest) modify the state of only those tuples that contain the abstract objects corresponding to the runtime object on which the

event occurs. We have proven the abstractions to be sound with respect to the tracematch semantics.

8.2 Static Analysis

We have implemented fully context-sensitive and inter-procedural versions of the abstractions as instances of the IFDS and IDE algorithms. The IFDS-based analysis is used to determine transition statements that are potential violations of a specified temporal property. The IDE-based analysis provides more useful information by providing a sequence of transition statements that could eventually lead to a violation.

The IFDS and IDE algorithms in their original form were not directly suitable for problems involving objects and pointers. We have presented four extensions to the IFDS algorithm that make it applicable to a wider class of interprocedural dataflow analysis problems, in particular analyses of objects and pointers. These include not requiring an exploded supergraph as input and instead building one on demand, extending the return-flow function to expose dataflow facts available before the call, more precise handling of programs in SSA form, and leveraging subsumption properties of the analysis domain to speed up the dataflow analysis. Although the extensions were discussed in terms of the IFDS algorithm they are equally applicable to the more general IDE algorithm and we have implemented them in both.

8.3 Precision

The precision of the analysis was evaluated using the tracematches of Bodden et al. [12] on the Dacapo Benchmark Suite. The results showed that the analysis is very precise. Of the 36 tracematch/benchmark pairs in which the benchmark used features checked by the tracematch, the analysis fully verified 15 to contain no possible violations. This gives the analysis a 42% success rate in fully verifying a program for a given property without requiring any manual intervention.

Overall, the analysis ruled out the possibility of a violation at 89% of the final transition statements in the benchmarks. From a practical standpoint, this leaves 11% of final transition statements that could not be verified. To complete the verification of the program, this much smaller subset of transition statements can be checked manually to determine

whether they are indeed violations or false positives. Alternately, these transition statements and associated update shadows can be instrumented to perform runtime monitoring of the temporal property.

8.4 Efficiency

A quick runtime was essential to make the analysis viable as a tool to be used within the development lifecycle. We have made the analysis efficient by implementing an optimization that uses method summaries. We designed and implemented callee summaries with the design principle that foregoing flow-sensitive analysis of a method in favour of using its callee summary should not decrease the precision of the computed abstraction at a callsite. Through experimental evidence, we showed that the precision of the tracematch analysis and alias set precision metrics are unaffected by the use of callee summaries. Empirically, the use of callee summaries reduced the running time for computing the abstraction by 27% on average.

When efficiency is even a bigger concern and some loss of precision is acceptable, we have designed and leveraged caller summaries to make the analysis even faster. To determine the maximum decrease in precision, we used the most conservative caller summary that assumes that any two parameters of a method might be aliased. Even with this worst case assumption, empirical evaluation revealed that the tracematch analysis's precision is not affected by using caller summaries. Since there is a decrease in the precision of the abstraction itself, we computed fine-grained alias-set precision metrics. These metrics showed an average decrease of 8% in the must-aliasing information that can be inferred from the abstraction, and a decrease of 4% on average for the must-not alias information that can be inferred. Empirical evaluation of the running time when using both summaries on the alias set abstraction showed a significant 96% decrease, which makes the tracematch analysis quite suitable for use in a typical development and quality assurance life cycle.

8.5 Verification Tool

We have developed a verification tool in the form of an Eclipse plugin to make the analysis practical to use for developers. The tool provides a configuration panel which can be used to specify inputs to the analysis. These include specifying which part of a project's source to analyze and specifying any code that is loaded through reflection. Options to select use of method summaries and many debugging and analysis options are also exposed. The tool

provides two features for visualizing the results. A list view provides a list of transition statements flagged as possible violations. Below each violation is the sequence of transition statements that could lead to this violation. The location of each statement in the source code is specified. Clicking on any statement opens a Java editor window containing the class which contains the statement. The editor has been customized to contain specialized markers in the left browsing panel to provide quick visual references to where possible violations and related transition statements are. These lines of code are also highlighted.

8.6 Future Work

A key observation while comparing the precision of our analysis with that of Bodden et al. was that the analyses are complementary to each other as they are successful on different transition statements. Therefore, the result of running the two analyses together is more precise than either analysis on its own. Our analysis relies on precise information in the form of sets of local variables pointing to an object. In cases where no local references to an object remain, the analysis cannot track that object in the heap and makes worst case assumptions. A situation which was encountered often in the benchmarks was when instead of passing an object as an argument, it was stored in a field of another object which was made available to the called method. Since there is no local reference to the object, our analysis cannot track it precisely. A possible solution for this imprecision is to use a flow-insensitive subset-based pointer analysis, using allocation sites as the object abstraction, to track the set of objects stored in a particular field. In cases where the analysis has no local references to an object, the may-points-to information can be used. If the may-points-to information guarantees that the field can never point to a particular object, i.e., the may-points-to set for the field does not contain the allocation site of the given object, then this is more precise than the current worst-case assumption that the analysis makes for fields.

A second source of imprecision is due to imprecise handling of interprocedural exceptional control flow. By making suitable modifications to the IFDS and IDE algorithms, it should be possible to improve the precision of how exceptions are handled, which would improve the precision of the analysis.

A significant portion of the running time of the analysis is spent creating a sound and precise whole-program call graph, including the Java standard library. Recently, Ali et al. [2, 3] have developed the Averroes tool that generates a placeholder library which over-approximates the behaviour of the standard library. This library can be constructed quickly, is smaller in size and can be used in place of the standard Java library. The

advantage is that by using the placeholder library, the construction time for the whole-program call graph can be improved by a factor of 4.3x to 12x. Additionally, Averroes makes it easier to handle reflection soundly. Since the call graphs built with Averroes have been shown to be as precise as the framework we used to build our call graph, using the Averroes generated library promises to lead to further improvements in the running time of the analysis.

APPENDICES

Appendix A

Proofs

Proposition 1. $\langle \mathbf{Bind}, \sqsubseteq \rangle$ is a complete lattice with meet operator defined as:

$$\prod D \triangleq \begin{cases} \perp & \text{if } D \text{ contains } \perp \text{ or } o_1, o_2 \text{ with } o_1 \neq o_2 \text{ or } o_1, \overline{O_2} \text{ with } o_1 \in O_2 \\ o & \text{if the above case does not hold and } o \in D \\ \bigcup_{\overline{O} \in D} O & \text{otherwise} \end{cases}$$

Proof. We first show that the meet as defined is the greatest lower bound of D .

Case $\perp \in D$: In this case, $\perp \sqsubseteq d$ by definition for all $d \in D$, and \perp is the only lower bound of \perp , so \perp is the glb.

Case $o_1, o_2 \in D$ with $o_1 \neq o_2$: In this case, \perp is the only lower bound of both o_1 and o_2 , so \perp is the glb.

Case $o_1, \overline{O_2} \in D$ with $o_1 \in O_2$: In this case, \perp is the only lower bound of both o_1 and $\overline{O_2}$, so \perp is the glb.

Case $o \in D$ and none of the above cases hold: In this case, D does not contain \perp or any positive bindings other than o . Thus D only contains o and negative bindings. None of the negative bindings contain o . Therefore o is a lower bound of each negative binding. Thus o is a lower bound of D . The only elements that can be lower bounds of a positive binding are the positive binding itself or \perp . Since $o \sqsupseteq \perp$, o is the glb.

Case none of the above cases hold: In this case, D contains only negative bindings. Their union contains all of them and is therefore a lower bound. Other lower bounds

are other sets that contain all of them, positive bindings not contained in any of the negative bindings in D , and \perp . All of these are less than $\bigcup_{O \in D} O$. Thus the latter is the glb.

Since $\langle \mathbf{Bind}, \sqsubseteq \rangle$ has a meet for arbitrary subsets, it is a complete meet semi-lattice. Thus it is a complete lattice [21, Theorem 2.16]. \square

Theorem 1. *The transition relations $\dot{\rightarrow}$ and \rightarrow are bisimilar with bisimulation relation $\dot{\sigma} R \sigma \triangleq s_\sigma(\sigma)(q) \iff \dot{\sigma}(q)$. That is,*

- for every σ there exists $\dot{\sigma}$ with $s_\sigma(\sigma)(q) \iff \dot{\sigma}(q)$ such that $\langle \mathbf{tr}(T), \sigma \rangle \rightarrow \langle \sigma' \rangle \implies \langle \mathbf{tr}(T), \dot{\sigma} \rangle \dot{\rightarrow} \langle \dot{\sigma}' \rangle \wedge \dot{\sigma}'(q) \iff s_\sigma(\sigma')(q)$, and conversely,
- for every $\dot{\sigma}$ there exists σ with $s_\sigma(\sigma)(q) \iff \dot{\sigma}(q)$ such that $\langle \mathbf{tr}(T), \dot{\sigma} \rangle \rightarrow \langle \dot{\sigma}' \rangle \implies \langle \mathbf{tr}(T), \sigma \rangle \rightarrow \langle \sigma' \rangle \wedge \dot{\sigma}'(q) \iff s_\sigma(\sigma')(q)$.

The following lemmas are needed to prove the theorem.

Lemma 1.

$$s_d(\langle f, d_1 \sqcap d_2 \rangle) = s_d(\langle f, d_1 \rangle) \wedge s_d(\langle f, d_2 \rangle)$$

Proof. Using case analysis on d_1 and d_2 .

Case $d_1 = \perp$ or $d_2 = \perp$: Then $s_d(\langle f, d_1 \sqcap d_2 \rangle) = s_d(\langle f, \perp \rangle) = \text{false}$. On the other side, $s_d(\langle f, d_1 \rangle) = \text{false}$ or $s_d(\langle f, d_2 \rangle) = \text{false}$, so their conjunction is false.

Case $d_1 = d_2 = o$: Then $s_d(\langle f, d_1 \sqcap d_2 \rangle) = s_d(\langle f, o \rangle) = s_d(\langle f, o \rangle) \wedge s_d(\langle f, o \rangle) = s_d(\langle f, d_1 \rangle) \wedge s_d(\langle f, d_2 \rangle)$.

Case $d_1 = o_1$ and $d_2 = o_2$ where $o_1 \neq o_2$: Then $s_d(\langle f, d_1 \sqcap d_2 \rangle) = s_d(\langle f, \perp \rangle) = \text{false}$. On the other side, $s_d(\langle f, d_1 \rangle) \wedge s_d(\langle f, d_2 \rangle) = (f = o_1) \wedge (f = o_2) = \text{false}$ since $o_1 \neq o_2$.

Case $d_1 = \overline{O_1}$ and $d_2 = o_2$ where $o_2 \in \overline{O_1}$: Then $s_d(\langle f, d_1 \sqcap d_2 \rangle) = s_d(\langle f, \perp \rangle) = \text{false}$. On the other side, $s_d(\langle f, d_1 \rangle) \wedge s_d(\langle f, d_2 \rangle) = \bigwedge_{o \in \overline{O_1}} \neg(f = o) \wedge (f = o_2) = \bigwedge_{o \in \overline{O_1}} \neg(f = o) \wedge \neg(f = o_2) \wedge (f = o_2) = \text{false}$ since $o_2 \in \overline{O_1}$.

Case $d_1 = \overline{O_1}$ and $d_2 = o_2$ where $o_2 \notin \overline{O_1}$: Then $s_d(\langle f, d_1 \sqcap d_2 \rangle) = s_d(\langle f, o_2 \rangle) = (f = o_2)$. On the other side, $s_d(\langle f, d_1 \rangle) \wedge s_d(\langle f, d_2 \rangle) = \bigwedge_{o \in \overline{O_1}} \neg(f = o) \wedge (f = o_2) = (f = o_2)$ since $(f = o_2) \implies \neg(f = o)$ for all $o \neq o_2$, and $o_2 \notin \overline{O_1}$.

Case $d_1 = \overline{O_1}$ and $d_2 = \overline{O_2}$: Then $s_d(\langle f, d_1 \sqcap d_2 \rangle) = s_d(\langle f, O_1 \cup O_2 \rangle) = \bigwedge_{o \in O_1 \cup O_2} \neg(f = o)$. On the other side, $s_d(\langle f, d_1 \rangle) \wedge s_d(\langle f, d_2 \rangle) = \bigwedge_{o \in O_1} \neg(f = o) \wedge \bigwedge_{o \in O_2} \neg(f = o) = \bigwedge_{o \in O_1 \cup O_2} \neg(f = o)$.

□

Lemma 2.

$$s_m(m_1 \sqcap m_2) = s_m(m_1) \wedge s_m(m_2)$$

Proof.

$$\begin{aligned}
s_m(m_1 \sqcap m_2) &= \bigwedge_{f \in F} s_d(\langle f, (m_1 \sqcap m_2)(f) \rangle) && \text{definition of } s_m \\
&= \bigwedge_{f \in F} s_d(\langle f, m_1(f) \sqcap m_2(f) \rangle) && \text{definition of } \sqcap_{F \rightarrow \mathbf{Bind}} \\
&= \bigwedge_{f \in F} s_d(\langle f, m_1(f) \rangle) \wedge s_d(\langle f, m_2(f) \rangle) && \text{Lemma 1} \\
&= \bigwedge_{f \in F} s_d(\langle f, m_1(f) \rangle) \wedge \bigwedge_{f \in F} s_d(\langle f, m_2(f) \rangle) \\
&= s_m(m_1) \wedge s_m(m_2) && \text{definition of } s_m
\end{aligned}$$

□

Lemma 3.

$$s_m(e^+(b, \rho)) = \dot{e}_0(b, \rho)$$

Proof.

$$\begin{aligned}
s_m(e^+(b, \rho)) &= \bigwedge_{f \in F} s_d(\langle f, e^+(b, \rho)(f) \rangle) && \text{definition of } s_m \\
&= \bigwedge_{f \in \text{dom}(b)} s_d(\langle f, \rho(b(f)) \rangle) \wedge \bigwedge_{f \notin \text{dom}(b)} s_d(\langle f, \bar{\emptyset} \rangle) && \text{definition of } e^+ \\
&= \bigwedge_{f \in \text{dom}(b)} f = \rho(b(f)) \wedge \bigwedge_{f \notin \text{dom}(b)} \bigwedge_{o \in \emptyset} \neg(f = o) && \text{definition of } s_d \\
&= \dot{e}_0(b, \rho) \wedge \text{true} && \text{empty conjunction} \\
&= \dot{e}_0(b, \rho)
\end{aligned}$$

□

Lemma 4.

$$\bigvee_{f \in \text{dom}(b)} s_m(e^-(b, \rho, f)) = \neg \dot{e}_0(b, \rho)$$

Proof.

$$\begin{aligned} \bigvee_{f \in \text{dom}(b)} s_m(e^-(b, \rho, f)) &= \bigvee_{f \in \text{dom}(b)} \bigwedge_{f' \in F} s_d(\langle f', e^-(b, \rho, f')(f') \rangle) \\ &= \bigvee_{f \in \text{dom}(b)} \left(s_d(\langle f, \overline{\rho(b(f))} \rangle) \wedge \bigwedge_{f' \in \{F \setminus f\}} s_d(\langle f', \bar{\emptyset} \rangle) \right) \\ &= \bigvee_{f \in \text{dom}(b)} \left(\neg(f = \rho(b(f))) \wedge \bigwedge_{f' \in \{F \setminus f\}} \bigwedge_{o \in \emptyset} \neg(f' = o) \right) \\ &= \bigvee_{f \in \text{dom}(b)} \neg(f = \rho(b(f))) \\ &= \neg \bigwedge_{f \in \text{dom}(b)} (f = \rho(b(f))) \\ &= \neg \dot{e}_0(b, \rho) \end{aligned}$$

□

Lemma 5. For all $q \in Q$,

$$\bigvee_{\langle q, m \rangle \in e^-[b_n, \rho](\dots(e^-[b_1, \rho](\sigma))\dots)} s_m(m) = s_\sigma(\sigma)(q) \wedge \bigwedge_{1 \leq i \leq n} \neg \dot{e}_0(b_i, \rho)$$

Proof. We use induction on n .

In the base case, $n = 0$, so the left-hand side is $\bigvee_{\langle q, m \rangle \in \sigma} s_m(m)$ and the right-hand side is $s_\sigma(\sigma)(q) \wedge \text{true}$. These are equal by the definition of s_σ .

For the inductive case, let $\sigma' = e^-[b_{(n-1)}, \rho](\dots(e^-[b_1, \rho](\sigma))\dots)$. We will show that if

$$\bigvee_{\langle q, m \rangle \in \sigma'} s_m(m) = s_\sigma(\sigma)(q) \wedge \bigwedge_{1 \leq i \leq n-1} \neg \dot{e}_0(b_i, \rho)$$

then

$$\bigvee_{\langle q, m \rangle \in e^-[b_n, \rho](\sigma')} s_m(m) = s_\sigma(\sigma)(q) \wedge \bigwedge_{1 \leq i \leq n} \neg \dot{e}_0(b_i, \rho)$$

Case $\text{dom}(b_n) = \emptyset$: In this case, $e^-[b_n, \rho](\sigma') = \emptyset$, so $\bigvee_{\langle q, m \rangle \in e^-[b_n, \rho](\sigma')} s_m(m) = \text{false}$, and $\neg \dot{e}_0(b_n, \rho) = \neg \text{true} = \text{false}$, so the right-hand side is also false.

Case $\nexists \langle q, m \rangle \in \sigma'$: In this case, $e^-[b_n, \rho](\sigma') = \emptyset$, so $\bigvee_{\langle q, m \rangle \in e^-[b_n, \rho](\sigma')} s_m(m) = \text{false}$, and $s_\sigma(\sigma')(q) = \text{false}$, so the right-hand side is also false.

Case $\text{dom}(b_n) \neq \emptyset$ and $\exists \langle q, m \rangle \in \sigma'$:

$$\begin{aligned}
& \bigvee_{\langle q, m \rangle \in e^-[b_n, \rho](\sigma')} s_m(m) \\
= & \bigvee_{\langle q, m \rangle \in \sigma'} \bigvee_{f \in \text{dom}(b_n)} s_m(m \sqcap e^-(b_n, \rho, f)) \\
= & \bigvee_{\langle q, m \rangle \in \sigma'} \bigvee_{f \in \text{dom}(b_n)} s_m(m[f \mapsto m(f) \sqcap \overline{\{\rho(b_n(f))\}}]) \\
= & \bigvee_{\langle q, m \rangle \in \sigma'} \bigvee_{f \in \text{dom}(b_n)} \bigwedge_{f' \in F} s_d(\langle \langle f', m[f \mapsto m(f) \sqcap \overline{\{\rho(b_n(f))\}}] \rangle \rangle (f')) \\
= & \bigvee_{\langle q, m \rangle \in \sigma'} \bigvee_{f \in \text{dom}(b_n)} \left(s_d(\langle \langle f, m(f) \sqcap \overline{\{\rho(b_n(f))\}} \rangle \rangle) \wedge \bigwedge_{f' \in F \setminus \{f\}} s_d(\langle \langle f', m(f') \rangle \rangle) \right) \\
= & \bigvee_{\langle q, m \rangle \in \sigma'} \bigvee_{f \in \text{dom}(b_n)} \left(s_d(\langle \langle f, m(f) \rangle \rangle) \wedge s_d(\langle \langle f, \overline{\{\rho(b_n(f))\}} \rangle \rangle) \wedge \bigwedge_{f' \in F \setminus \{f\}} s_d(\langle \langle f', m(f') \rangle \rangle) \right) \\
= & \bigvee_{\langle q, m \rangle \in \sigma'} \bigvee_{f \in \text{dom}(b_n)} \left(s_d(\langle \langle f, \overline{\{\rho(b_n(f))\}} \rangle \rangle) \wedge \bigwedge_{f' \in F} s_d(\langle \langle f', m(f') \rangle \rangle) \right) \\
= & \bigvee_{\langle q, m \rangle \in \sigma'} \bigvee_{f \in \text{dom}(b_n)} \neg(f = \rho(b_n(f))) \wedge s_m(m) \\
= & \left(\bigvee_{\langle q, m \rangle \in \sigma'} \bigvee_{f \in \text{dom}(b_n)} \neg(f = \rho(b_n(f))) \right) \wedge \left(\bigvee_{\langle q, m \rangle \in \sigma'} \bigvee_{f \in \text{dom}(b_n)} s_m(m) \right) \\
= & \left(\neg \bigwedge_{f \in \text{dom}(b_n)} (f = \rho(b_n(f))) \right) \wedge \left(\bigvee_{\langle q, m \rangle \in \sigma'} s_m(m) \right) \\
= & \neg \dot{e}_0(b_n, \rho) \wedge s_\sigma(\sigma)(q) \wedge \bigwedge_{1 \leq i \leq n-1} \neg \dot{e}_0(b_i, \rho) \\
= & s_\sigma(\sigma)(q) \wedge \bigwedge_{1 \leq i \leq n} \neg \dot{e}_0(b_i, \rho)
\end{aligned}$$

□

Lemma 6. *Every tracematch state in the original semantics has an equivalent in the lattice-based semantics. Formally, for every $\dot{\sigma} \in Q \rightarrow S$, there exists a $\sigma \in \mathbf{State}$ such that for all $q \in Q$, $\dot{\sigma}(q) \iff s_\sigma(\sigma)(q)$.*

Proof. Let $\dot{\sigma}(q)$ be an arbitrary boolean formula. It has an equivalent formula in disjunctive normal form as a disjunction of conjunctions of literals of the forms $(f = o)$ and $\neg(f = o)$. Simplify the DNF formula using the following identities:

- Replace $(f = o_1) \wedge (f = o_2)$ with false if $o_1 \neq o_2$.
- Replace $(f = o) \wedge \neg(f = o)$ with false.
- Replace $(f = o_1) \wedge \neg(f = o_2)$ with just $(f = o_1)$ if $o_1 \neq o_2$.
- Remove true from any conjunction in which it appears.
- Eliminate any conjunctions containing false.

Then each resulting conjunction contains, for each $f \in F$, either a single literal $(f = o)$, or a set of literals $\neg(f = o)$. In the former case define $m(f) \triangleq o$. In the latter case define $m(f) \triangleq \{\overline{o : \neg(f = o)} \text{ is a literal in the conjunction}\}$. Then $s_m(m)$ is exactly the conjunction. Define s_σ as the set of all pairs $\langle q, m \rangle$ such that $s_m(m)$ is a conjunction in the formula normalized from $\mathring{\sigma}(q)$. Then $s_\sigma(\sigma)(q) \iff \mathring{\sigma}(q)$ for all q as required. \square

Having proved the lemmas, we now give a proof of Theorem 1.

Proof (Proof of Theorem 1). For every σ we can define $\mathring{\sigma} \triangleq s_\sigma(\sigma)$, and this definition ensures that $\mathring{\sigma}(q) \iff s_\sigma(\sigma)(q)$. Conversely, for every $\mathring{\sigma}$, Lemma 6 constructs a σ such that the same property holds. It remains to show that if the property holds and $\langle \mathbf{tr}(T), \sigma \rangle \rightarrow \langle \sigma' \rangle$ and $\langle \mathbf{tr}(T), \mathring{\sigma} \rangle \overset{\circ}{\rightarrow} \langle \mathring{\sigma}' \rangle$, then $\mathring{\sigma}' = s_\sigma(\sigma')$.

$$\begin{aligned}
\dot{\sigma}' &= \lambda q. \left(\bigvee_{a,j:\delta(j,a,q)} (\dot{\sigma}[q_0 \mapsto \text{true}](j) \wedge \dot{e}(a, \{\langle a_1, b_1 \rangle \cdots \langle a_n, b_n \rangle\}, \rho)) \right) \vee \\
&\quad \left(\dot{\sigma}[q_0 \mapsto \text{true}](q) \wedge \bigwedge_{a \in A} \neg \dot{e}(a, \{\langle a_1, b_1 \rangle \cdots \langle a_n, b_n \rangle\}, \rho) \right) \\
&= \lambda q. \left(\bigvee_{a,j:\delta(j,a,q)} \left(s_\sigma(\sigma \cup \{\langle q_0, \lambda f. \top \rangle\})(j) \wedge \bigvee_{i:a_i=a} \dot{e}_0(b_i, \rho) \right) \right) \\
&\quad \left(s_\sigma(\sigma \cup \{\langle q_0, \lambda f. \top \rangle\})(q) \wedge \bigwedge_{1 \leq i \leq n} \neg \dot{e}_0(b_i, \rho) \right) \\
&= \lambda q. \left(\bigvee_{a,j:\delta(j,a,q)} \left(\bigvee_{\langle q,m \rangle \in \sigma \cup \{\langle q_0, \lambda f. \top \rangle\}} s_m(m) \wedge \bigvee_{i:a_i=a} s_m(e^+(b_i, \rho)) \right) \right) \vee \\
&\quad \left(\bigvee_{\langle q,m \rangle \in e^-[b_n, \rho] (\cdots (e^-[b_1, \rho] (\sigma \cup \{\langle q_0, \lambda f. \top \rangle\})) \cdots)} s_m(m) \right) \\
&= \lambda q. \left(\bigvee_{a,j:\delta(j,a,q)} \bigvee_{\langle q,m \rangle \in \sigma \cup \{\langle q_0, \lambda f. \top \rangle\}} \bigvee_{i:a_i=a} (s_m(m) \wedge s_m(e^+(b_i, \rho))) \right) \vee \\
&\quad \left(\bigvee_{\langle q,m \rangle \in e^-[b_n, \rho] (\cdots (e^-[b_1, \rho] (\sigma \cup \{\langle q_0, \lambda f. \top \rangle\})) \cdots)} s_m(m) \right) \\
&= \lambda q. \left(\bigvee_{1 \leq i \leq n} \bigvee_{j:\delta(j,a_i,q)} \bigvee_{\langle q,m \rangle \in \sigma \cup \{\langle q_0, \lambda f. \top \rangle\}} s_m(m \sqcap e^+(b_i, \rho)) \right) \vee \\
&\quad \left(\bigvee_{\langle q,m \rangle \in e^-[b_n, \rho] (\cdots (e^-[b_1, \rho] (\sigma \cup \{\langle q_0, \lambda f. \top \rangle\})) \cdots)} s_m(m) \right) \\
&= \lambda q. \left(\bigvee_{1 \leq i \leq n} \bigvee_{\langle q,m \rangle \in e^+[a_i, b_i, \rho] (\sigma \cup \{\langle q_0, \lambda f. \top \rangle\})} s_m(m) \right) \vee \\
&\quad \left(\bigvee_{\langle q,m \rangle \in e^-[b_n, \rho] (\cdots (e^-[b_1, \rho] (\sigma \cup \{\langle q_0, \lambda f. \top \rangle\})) \cdots)} s_m(m) \right)
\end{aligned}$$

$$\begin{aligned}
\delta' &= \lambda q. \left(\bigvee_{1 \leq i \leq n} \bigvee_{\langle q, m \rangle \in e^+[a_i, b_i, \rho](\sigma \cup \{ \langle q_0, \lambda f. \top \rangle \})} s_m(m) \right) \vee \\
&\quad \left(\bigvee_{\langle q, m \rangle \in e^-[b_n, \rho](\dots(e^-[b_1, \rho](\sigma \cup \{ \langle q_0, \lambda f. \top \rangle \})) \dots)} s_m(m) \right) \\
&= \lambda q. \bigvee_{\langle q, m \rangle \in (\bigcup_{1 \leq i \leq n} e^+[a_i, b_i, \rho](\sigma \cup \{ \langle q_0, \lambda f. \top \rangle \})) \cup e^-[b_1, \rho](\dots(e^-[b_n, \rho](\sigma \cup \{ \langle q_0, \lambda f. \top \rangle \})) \dots)} s_m(m)} \\
&= s_\sigma \left(\left(\bigcup_{1 \leq i \leq n} e^+[a_i, b_i, \rho](\sigma \cup \{ \langle q_0, \lambda f. \top \rangle \}) \right) \cup e^-[b_1, \rho](\dots(e^-[b_n, \rho](\sigma \cup \{ \langle q_0, \lambda f. \top \rangle \})) \dots) \right) \\
&= s_\sigma(\sigma')
\end{aligned}$$

□

Proposition 2. *If s is any statement except $v \leftarrow \mathbf{h}$, and $\langle s, \rho, h, \sigma \rangle \rightarrow \langle \rho', h', \sigma' \rangle$, then for any concrete object o that exists prior to the execution of s ,*

$$\llbracket s \rrbracket_{o\#}(\beta_o[\rho](o)) = \beta_o[\rho'](o)$$

Proof. **Case $s = v_1 \leftarrow v_2$ and $\rho(v_2) = o$:**

$$\begin{aligned}
\beta_o[\rho'](o) &= \beta_o[\rho[v_1 \mapsto \rho(v_2)]](o) \\
&= \beta_o[\rho[v_1 \mapsto o]](o) \\
&= \{v : \rho[v_1 \mapsto o](v) = o\} \\
&= \{v : \rho(v) = o\} \cup \{v_1\} \\
&= \beta_o[\rho](o) \cup \{v_1\} \\
\llbracket s \rrbracket_{o\#}(\beta_o[\rho](o)) &= \beta_o[\rho](o) \cup \{v_1\} \quad \text{since } v_2 \in \beta_o[\rho](o)
\end{aligned}$$

Case $s = v_1 \leftarrow v_2$ and $\rho(v_2) \neq o$:

$$\begin{aligned}
\beta_o[\rho'](o) &= \beta_o[\rho[v_1 \mapsto \rho(v_2)]](o) \\
&= \beta_o[\rho[v_1 \mapsto o' : o \neq o']](o) \\
&= \{v : \rho(v) = o\} \setminus \{v_1\} \\
&= \beta_o[\rho](o) \setminus \{v_1\} \\
\llbracket s \rrbracket_{o\#}(\beta_o[\rho](o)) &= \beta_o[\rho](o) \setminus \{v_1\} \quad \text{since } v_2 \notin \beta_o[\rho](o)
\end{aligned}$$

Case $s = v \leftarrow \text{null}$:

$$\begin{aligned}
\beta_o[\rho'](o) &= \beta_o[\rho[v \mapsto \perp]](o) \\
&= \{v' : \rho(v') = o\} \setminus \{v\} \\
&= \beta_o[\rho](o) \setminus \{v\} \\
&= \llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o))
\end{aligned}$$

Case $v \leftarrow \text{new}$:

$$\begin{aligned}
\beta_o[\rho'](o) &= \beta_o[\rho[v \mapsto o']](o) && \text{with } o' \text{ fresh} \\
&= \{v' : \rho(v') = o\} \setminus \{v\} && \text{since } o \neq o' \\
&= \beta_o[\rho](o) \setminus \{v\} \\
&= \llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o))
\end{aligned}$$

Case $s \in \{\mathbf{h} \leftarrow v, \text{tr}(T), \text{body}\}$: For these statements, $\rho' = \rho$ and $\llbracket s \rrbracket_{o^\#}$ is the identity. Thus $\llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o)) = \beta_o[\rho](o) = \beta_o[\rho'](o)$.

□

Theorem 2. *If $\langle s, \rho, h, \sigma \rangle \rightarrow \langle \rho', h', \sigma' \rangle$ and $\langle \rho, h \rangle R_{\rho h} \langle \rho^\#, h^\# \rangle$, then $\langle \rho', h' \rangle R_{\rho h} \llbracket s \rrbracket_{\rho h^\#}(\rho^\#, h^\#)$.*

The following lemma is needed to prove the theorem.

Lemma 7. *If s is any statement except $v \leftarrow \text{new}$, and $\langle s, \rho, h, \sigma \rangle \rightarrow \langle \rho', h', \sigma' \rangle$, then $\text{range}(\rho) \cup h \setminus \{\perp\} \supseteq \text{range}(\rho') \cup h' \setminus \{\perp\}$.*

Proof. Since $x \supseteq x'$ implies $x \setminus \{\perp\} \supseteq x' \setminus \{\perp\}$ for any x, x' , for all but the last case, we show that $\text{range}(\rho) \cup h \supseteq \text{range}(\rho') \cup h'$.

Case $s = v_1 \leftarrow v_2$: $\text{range}(\rho') = \text{range}(\rho[v_1 \mapsto \rho(v_2)]) \subseteq \text{range}(\rho)$. Also, $h' = h$. Thus $\text{range}(\rho) \cup h \supseteq \text{range}(\rho') \cup h'$.

Case $s = v \leftarrow \mathbf{h}$: $\text{range}(\rho') = \text{range}(\rho[v \mapsto o])$ for some $o \in h$. Thus $\text{range}(\rho') \cup h' = \text{range}(\rho[v \mapsto o]) \cup h \subseteq \text{range}(\rho) \cup h$ since $o \in h$.

Case $s = \mathbf{h} \leftarrow v$: $\text{range}(\rho') \cup h' = \text{range}(\rho) \cup h \cup \{\rho(v)\} = \text{range}(\rho) \cup h$.

Case $s \in \{\mathbf{body}, \mathbf{tr}(T)\}$: Since $\rho' = \rho$ and $h' = h$, $\text{range}(\rho') \cup h' = \text{range}(\rho) \cup h$.

Case $s = v \leftarrow \mathbf{null}$: $\text{range}(\rho') = \text{range}(\rho[v \mapsto \perp]) \subseteq \text{range}(\rho) \cup \{\perp\}$. Since $h = h'$, this implies that $\text{range}(\rho) \cup h \setminus \{\perp\} \supseteq \text{range}(\rho') \cup h' \setminus \{\perp\}$.

□

Proof (Proof of Theorem 2). We first prove the theorem for the special case when $\langle \rho^\#, h^\# \rangle = \beta_{\rho h}(\rho, h)$.

By the definitions of $R_{\rho h}$ and \sqsubseteq , the conclusion of the theorem is equivalent to $\beta_\rho(\rho', h') \subseteq \llbracket s \rrbracket_{\rho^\#}(\rho^\#, h^\#) \wedge \beta_h(\rho', h') \subseteq \llbracket s \rrbracket_{h^\#}(\rho^\#, h^\#)$. We first prove $\beta_\rho(\rho', h') \subseteq \llbracket s \rrbracket_{\rho^\#}(\rho^\#, h^\#)$.

Case $s \in \{v_1 \leftarrow v_2, v \leftarrow \mathbf{null}, \mathbf{h} \leftarrow v\}$:

$$\begin{aligned}
\llbracket s \rrbracket_{\rho^\#}(\rho^\#, h^\#) &= \llbracket s \rrbracket_{o^\#}[h^\#](\rho^\#) \\
&= \{\llbracket s \rrbracket_{o^\#}(o^\#) : o^\# \in \rho^\#\} \\
&= \{\llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o)) : o \in h \cup \text{range}(\rho) \setminus \{\perp\}\} \\
&\supseteq \{\llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o)) : o \in h' \cup \text{range}(\rho') \setminus \{\perp\}\} \text{ (Using Lemma 7)} \\
&= \{\beta_o[\rho'](o) : o \in h' \cup \text{range}(\rho') \setminus \{\perp\}\} \text{ (Using Proposition 2)} \\
&= \beta_\rho(\rho', h')
\end{aligned}$$

Case $s = v \leftarrow \mathbf{new}$: Let o' be the newly created object. Since $h' = h$ and $\rho' = \rho[v \mapsto o']$, $\text{range}(\rho) \cup \{o'\} \cup h \supseteq \text{range}(\rho') \cup h'$.

$$\begin{aligned}
\llbracket s \rrbracket_{\rho^\#}(\rho^\#, h^\#) &= \llbracket s \rrbracket_{o^\#}[h^\#](\rho^\#) \cup \{v\} \\
&= \{\llbracket s \rrbracket_{o^\#}(o^\#) : o^\# \in \rho^\#\} \cup \{v\} \\
&= \{\llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o)) : o \in h \cup \text{range}(\rho) \setminus \{\perp\}\} \cup \{v\} \\
&= \{\beta_o[\rho'](o) : o \in h \cup \text{range}(\rho) \setminus \{\perp\}\} \cup \{v\} \text{ Proposition 2} \\
&= \{\beta_o[\rho'](o) : o \in h \cup \text{range}(\rho) \setminus \{\perp\} \cup o'\} \text{ since } \{v\} = \beta_o[\rho'](o') \\
&\supseteq \{\beta_o[\rho'](o) : o \in h' \cup \text{range}(\rho') \setminus \{\perp\}\} \\
&= \beta_\rho(\rho', h')
\end{aligned}$$

Case $s = v \leftarrow \mathbf{h}$: Let $o' \in h$ be the object such that $\rho' = \rho[v \mapsto o']$ (the object being loaded).

$$\begin{aligned}
\llbracket s \rrbracket_{\rho^\sharp}(\rho^\sharp, h^\sharp) &= \llbracket s \rrbracket_{o^\sharp}[h^\sharp](\rho^\sharp) \\
&= \bigcup_{o^\sharp \in \rho^\sharp} \text{focus}[h^\sharp](o^\sharp) \\
&= \bigcup_{o \in h \cup \text{range}(\rho) \setminus \{\perp\}} \text{focus}[h^\sharp](\beta_o[\rho](o)) \\
&= \bigcup_{o \in \text{range}(\rho) \setminus \{\perp\} \setminus h} \text{focus}[h^\sharp](\beta_o[\rho](o)) \cup \\
&\quad \bigcup_{o \in h \setminus \{\perp\}} \text{focus}[h^\sharp](\beta_o[\rho](o)) \\
&= \{\beta_o[\rho](o) \setminus \{v\} : o \in \text{range}(\rho) \setminus \{\perp\} \setminus h\} \cup \\
&\quad \bigcup_{o \in h \setminus \{\perp\}} \{\beta_o[\rho](o) \setminus \{v\}, \beta_o[\rho](o) \cup \{v\}\} \\
&\supseteq \{\beta_o[\rho[v \mapsto o']](o) : o \in \text{range}(\rho) \setminus \{\perp\} \setminus h\} \cup \\
&\quad \bigcup_{o \in h \setminus \{\perp\}} \{\beta_o[\rho[v \mapsto o']](o)\} \\
&= \{\beta_o[\rho'](o) : o \in \text{range}(\rho) \cup h \setminus \{\perp\}\} \\
&\supseteq \{\beta_o[\rho'](o) : o \in \text{range}(\rho') \cup h' \setminus \{\perp\}\} \\
&= \beta_\rho(\rho', h')
\end{aligned}$$

Case $s \in \{\text{tr}(T), \text{body}\}$: In this case, $\rho' = \rho$, $h' = h$, thus $\llbracket s \rrbracket_{\rho^\sharp}(\rho^\sharp, h^\sharp) = \rho^\sharp = \beta_\rho(\rho', h')$.

Next we prove $\beta_h(\rho', h') \subseteq \llbracket s \rrbracket_{h^\sharp}(\rho^\sharp, h^\sharp)$.

Case $s \in \{v_1 \leftarrow v_2, v \leftarrow \mathbf{null}, v \leftarrow \mathbf{new}\}$:

$$\begin{aligned}
\llbracket s \rrbracket_{h^\sharp}(\rho^\sharp, \beta_h(\rho, h)) &= \llbracket s \rrbracket_{h^\sharp}(\rho^\sharp, h^\sharp) \\
&= \llbracket s \rrbracket_{o^\sharp}h^\sharp \\
&= \{\llbracket s \rrbracket_{o^\sharp}(o^\sharp) : o^\sharp \in h^\sharp\} \\
&= \{\llbracket s \rrbracket_{o^\sharp}(\beta_o[\rho](o)) : o \in h\} \\
&= \{\llbracket s \rrbracket_{o^\sharp}(\beta_o[\rho](o)) : o \in h'\} \text{ (Since } h = h'\text{)} \\
&= \{\beta_o[\rho'](o) : o \in h'\} \text{ (Using Proposition 2)} \\
&= \beta_h(\rho', h')
\end{aligned}$$

Case $s = \mathbf{h} \leftarrow v$:

$$\begin{aligned}
\llbracket s \rrbracket_{h^\sharp}(\rho^\sharp, h^\sharp) &= \llbracket s \rrbracket_{o^\sharp}[h^\sharp](h^\sharp \cup \{o^\sharp \in \rho^\sharp : v \in o^\sharp\}) \\
&= \{\llbracket s \rrbracket_{o^\sharp}(o^\sharp) : o^\sharp \in h^\sharp \cup \{o^\sharp \in \rho^\sharp : v \in o^\sharp\}\} \\
&= \{\llbracket s \rrbracket_{o^\sharp}(o^\sharp) : o^\sharp \in h^\sharp\} \cup \{\llbracket s \rrbracket_{o^\sharp}(o^\sharp) : o^\sharp \in \rho^\sharp \wedge v \in o^\sharp\} \\
&= \{\llbracket s \rrbracket_{o^\sharp}(\beta_o[\rho](o)) : o \in h\} \cup \{o^\sharp \in \rho^\sharp : v \in o^\sharp\} \\
&= \{\beta_o[\rho](o) : o \in h\} \cup \{\beta_o[\rho](\rho(v))\} \\
&= \{\beta_o[\rho](o) : o \in h \cup \{\rho(v)\}\} \\
&= \{\beta_o[\rho'](o) : o \in h'\} \\
&= \beta_h(\rho', h')
\end{aligned}$$

Case $s = v \leftarrow \mathbf{h}$: Let $o' \in h$ be the object such that $\rho' = \rho[v \mapsto o']$ (the object being

loaded).

$$\begin{aligned}
\llbracket s \rrbracket_{h^\#}(\rho^\#, h^\#) &= \llbracket s \rrbracket_{o^\#}h^\# \\
&= \bigcup_{o^\# \in h^\#} \text{focus}[h^\#](o^\#) \\
&= \bigcup_{o^\# \in h^\#} \{o^\# \setminus \{v\}, o^\# \cup \{v\}\} \\
&= \bigcup_{o \in h} \{\beta_o[\rho](o) \setminus \{v\}, \beta_o[\rho](o) \cup \{v\}\} \\
&\supseteq \bigcup_{o \in h} \{\beta_o[\rho[v \mapsto o']](o)\} \\
&= \{\beta_o[\rho'](o) : o \in h\} \\
&= \beta_h(\rho', h) \\
&= \beta_h(\rho', h')
\end{aligned}$$

Case $s \in \{\text{tr}(T), \text{body}\}$: In this case, $\rho' = \rho$, $h' = h$, thus $\llbracket s \rrbracket_{h^\#}(\rho^\#, h^\#) = h^\# = \beta_h(\rho', h')$.

This completes the proof for the special case when $\langle \rho^\#, h^\# \rangle = \beta_{\rho h}(\rho, h)$. In general, $\langle \rho^\#, h^\# \rangle \supseteq \beta_{\rho h}(\rho, h)$. Since $\llbracket s \rrbracket_{\rho h^\#}$ is monotone, $\llbracket s \rrbracket_{\rho h^\#}(\rho^\#, h^\#) \supseteq \llbracket s \rrbracket_{\rho h^\#}(\beta_{\rho h}(\rho, h))$, which we just proved is greater than $\beta_{\rho h}(\rho', h')$. Thus, the theorem holds in the general case. \square

Proposition 3. $\langle \mathbf{Bind}^\#, \sqsubseteq \rangle$ is a finite lattice with meet operator defined as:

$$\begin{aligned}
\perp \sqcap x &= x \sqcap \perp \triangleq \perp \text{ for any } x \\
\langle o_1^!, o_2^? \rangle \sqcap \langle o_2^!, o_2^? \rangle &\triangleq \text{pos}(o_1^! \cup o_2^!, o_1^? \cap o_2^?) \\
\langle o^!, o^? \rangle \sqcap \overline{V}^\# &= \overline{V}^\# \sqcap \langle o^!, o^? \rangle \triangleq \text{pos}(o^!, o^? \setminus V^\#) \\
\overline{V}_1^\# \sqcap \overline{V}_2^\# &\triangleq \overline{V}_1^\# \cup \overline{V}_2^\#
\end{aligned}$$

where $\text{pos}(o^!, o^?) \triangleq \begin{cases} \langle o^!, o^? \rangle & \text{if } o^! \subseteq o^? \\ \perp & \text{otherwise} \end{cases}$

Proof. $\mathbf{Bind}^\#$ is finite by construction because \mathbf{Var} is finite.

The bottom element \perp is a lower bound of every element, and is the only lower bound of itself. Therefore, it is the glb of any pair containing \perp .

A lower bound of two positive bindings $d_1^\#, d_2^\#$ can be either \perp or a positive binding whose must set is a superset of their must sets and whose may set is a subset of their may sets. Of the positive bindings, the one whose must set is the union of the must sets of $d_1^\#$ and $d_2^\#$ and whose may set is the intersection is greater than all others. It is also greater than \perp , so it is the glb. However, every positive binding must respect the restriction $o^! \subseteq o^?$. When this restriction cannot be respected the only and therefore greatest lower bound is \perp .

The case of a meet of a positive binding and a negative binding is similar. Any lower bound must be either \perp or a positive binding whose must set is a superset of the original must set, and whose may set is a subset of the original may set but disjoint from the negative binding. The positive binding $\langle o^!, o^? \setminus V^\# \rangle$ satisfies these restrictions and is greater than all other positive bindings that do. It is also greater than \perp . Thus it is the glb. However, when it does not respect the subset restriction on positive bindings, only \perp is a lower bound and is therefore the glb.

The meet of two negative bindings, if it is a negative binding, must be a superset of both. Their union is greater than any other such negative binding, and it is greater than any positive binding and \perp , so it is the glb.

Since $\mathbf{Bind}^\#$ is finite, it is a complete meet semi-lattice. Therefore it is a complete, finite lattice. \square

Proposition 4. *The abstraction function $\beta_d[\rho]$ is monotone. That is, $d_1 \sqsubseteq d_2 \implies \beta_d[\rho](d_1) \sqsubseteq \beta_d[\rho](d_2)$.*

Proof. For conciseness, define $d_1^\# \triangleq \beta_d[\rho](d_1)$ and $d_2^\# \triangleq \beta_d[\rho](d_2)$.

When $d_1 = \perp$, $d_1^\#$ is also \perp , so the conclusion holds.

When d_1 is a positive binding o_1 , d_2 is either also o_1 or a negative binding $\overline{O_2}$ with $o_1 \notin O_2$. In the former case, the conclusion holds trivially. In the latter case, since $o_1 \notin O_2$, none of the variables pointing to o_1 point to any object in O_2 . Thus $\beta_o(o_1)$ is disjoint from every $\beta_o(o)$ for any $o \in O_2$. Thus $\beta_d[\rho](\overline{O_2})$ is disjoint from the must set of $\beta_d[\rho](o_1)$. Therefore $\beta_d[\rho](d_1) \sqsubseteq \beta_d[\rho](d_2)$.

When d_1 is a negative binding $\overline{O_1}$, d_2 can only be a negative binding $\overline{O_2}$ with $O_1 \supseteq O_2$. Therefore $d_1^\# = \bigcup_{o \in O_1} \beta_o(o) \supseteq \bigcup_{o \in O_2} \beta_o(o) = d_2^\#$, so $d_1^\# \sqsubseteq d_2^\#$. \square

Proposition 5. *If $\langle s, \rho \rangle \rightarrow \langle \rho' \rangle$ then $d \ R_d[\rho] \ d^\# \implies d \ R_d[\rho'] \ \llbracket s \rrbracket_{d^\#} (d^\#)$.*

We use the following lemmas to prove the proposition.

Lemma 8. *If $o R_d[\rho] d^\sharp$, then d^\sharp is either a negative binding, or $d^\sharp = \langle o^1, o^2 \rangle$ and $o^1 \subseteq \beta_o[\rho](o) \subseteq o^2$.*

Proof. Since $o R_d[\rho] d^\sharp$, $\beta_d[\rho](o) = \langle \beta_o[\rho](o), \beta_o[\rho](o) \rangle \sqsubseteq d^\sharp$. Therefore d^\sharp cannot be \perp , so it must be a negative or positive binding. If it is a positive binding, it must be greater than $\langle \beta_o[\rho](o), \beta_o[\rho](o) \rangle$, which is defined to mean $o^1 \subseteq \beta_o[\rho](o) \subseteq o^2$. \square

Lemma 9. *If $\bar{O} R_d[\rho] d^\sharp$, then d^\sharp is a negative binding $d^\sharp = \bar{V}^\sharp \subseteq \bigcup_{o \in \bar{O}} \beta_o[\rho](o)$.*

Proof. Since $\bar{O} R_d[\rho] d^\sharp$, $\beta_d[\rho](\bar{O}) = \overline{\bigcup_{o \in \bar{O}} \beta_o[\rho](o)} \sqsubseteq d^\sharp$. Only negative bindings are greater than a negative binding, so d^\sharp must be a negative binding. Also, to be greater, d^\sharp must be a subset of $\bigcup_{o \in \bar{O}} \beta_o[\rho](o)$. \square

Proof (Proof of Proposition 5).

Case $d = \perp$: Then $\beta_d[\rho](d) = \perp \sqsubseteq \llbracket s \rrbracket_{d^\sharp}(d^\sharp)$, so $R_d[\rho'] \llbracket s \rrbracket_{d^\sharp}(d^\sharp)$.

Case d is a positive binding o : By Lemma 8, d^\sharp is either a negative binding or $\langle o^1, o^2 \rangle$.

If d^\sharp is a negative binding, then so is $\llbracket s \rrbracket_{d^\sharp}(d^\sharp)$, so since $\beta_d[\rho](d)$ is less than any negative binding, $d R_d[\rho'] \llbracket s \rrbracket_{d^\sharp}(d^\sharp)$. Thus, the remaining case is when $d^\sharp = \langle o^1, o^2 \rangle$. By Lemma 8, $o^1 \subseteq \beta_o[\rho](o^\sharp) \subseteq o^2$.

Subcase $s = v_1 \leftarrow v_2 \wedge v_2 \in o^1$:

Since $v_2 \in o^1$ this means $v_2 \in \beta_o[\rho](o)$ and $v_2 \in o^2$.

$$\begin{aligned}
& o R_d[\rho] \langle o^1, o^2 \rangle \\
& \implies o^1 \subseteq \beta_o[\rho](o) \subseteq o^2 \\
& \implies o^1 \cup \{v_1\} \subseteq \beta_o[\rho](o) \cup \{v_1\} \subseteq o^2 \cup \{v_1\} \\
& \implies o^1 \cup \{v_1\} \subseteq \llbracket s \rrbracket_{o^\sharp}(\beta_o[\rho](o)) \subseteq o^2 \cup \{v_1\} \quad \text{defn. of } \llbracket s \rrbracket_{o^\sharp} \text{ when } v_2 \in \beta_o[\rho](o) \\
& \implies o^1 \cup \{v_1\} \subseteq \beta_o[\rho'](o) \subseteq o^2 \cup \{v_1\} \quad \text{Proposition 2} \\
& \implies o R_d[\rho'] \langle o^1 \cup \{v_1\}, o^2 \cup \{v_1\} \rangle \quad \text{defn. of } R_d[\rho] \\
& \implies o R_d[\rho'] \llbracket s \rrbracket_{d^\sharp}(\langle o^1, o^2 \rangle) \quad \text{defn. of } \llbracket s \rrbracket_{d^\sharp}
\end{aligned}$$

Subcase $s = v_1 \leftarrow v_2 \wedge v_2 \notin o^1 \wedge v_2 \in o^2$:

$$\begin{aligned}
& o R_d[\rho] \langle o^1, o^2 \rangle \\
& \implies o^1 \subseteq \beta_o[\rho](o) \subseteq o^2 \\
& \implies o^1 \setminus \{v_1\} \subseteq \beta_o[\rho](o) \setminus \{v_1\} \subseteq \beta_o[\rho](o) \cup \{v_1\} \subseteq o^2 \cup \{v_1\} \\
& \implies o^1 \setminus \{v_1\} \subseteq \llbracket s \rrbracket_{o^\sharp}(\beta_o[\rho](o)) \subseteq o^2 \cup \{v_1\} \quad \text{defn. of } \llbracket s \rrbracket_{o^\sharp} \\
& \implies o^1 \setminus \{v_1\} \subseteq \beta_o[\rho'](o) \subseteq o^2 \cup \{v_1\} \quad \text{Proposition 2} \\
& \implies o R_d[\rho'] \langle o^1 \setminus \{v_1\}, o^2 \cup \{v_1\} \rangle \quad \text{defn. of } R_d[\rho] \\
& \implies o R_d[\rho'] \llbracket s \rrbracket_{d^\sharp}(\langle o^1, o^2 \rangle) \quad \text{defn. of } \llbracket s \rrbracket_{d^\sharp}
\end{aligned}$$

Subcase $s = v_1 \leftarrow v_2 \wedge v_2 \notin o^1 \wedge v_2 \notin o^2$:

Since $v_2 \notin o^2$ this means $v_2 \notin \beta_o[\rho](o)$.

$$\begin{aligned}
& o R_d[\rho] \langle o^1, o^2 \rangle \\
& \implies o^1 \subseteq \beta_o[\rho](o) \subseteq o^2 \\
& \implies o^1 \setminus \{v_1\} \subseteq \beta_o[\rho](o) \setminus \{v_1\} \subseteq o^2 \setminus \{v_1\} \\
& \implies o^1 \setminus \{v_1\} \subseteq \llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o)) \subseteq o^2 \setminus \{v_1\} \quad \text{defn. of } \llbracket s \rrbracket_{o^\#} \text{ when } v_2 \notin \beta_o[\rho](o) \\
& \implies o^1 \setminus \{v_1\} \subseteq \beta_o[\rho'](o) \subseteq o^2 \setminus \{v_1\} \quad \text{Proposition 2} \\
& \implies o R_d[\rho'] \langle o^1 \setminus \{v_1\}, o^2 \setminus \{v_1\} \rangle \quad \text{defn. of } R_d[\rho] \\
& \implies o R_d[\rho'] \llbracket s \rrbracket_{d^\#}(\langle o^1, o^2 \rangle) \quad \text{defn. of } \llbracket s \rrbracket_{d^\#}
\end{aligned}$$

Subcase $s = v \leftarrow \mathbf{h}$:

Let the object loaded from the heap be o' . Then $\rho' = \rho[v \mapsto o']$.

If $o' = o$, then $\rho'(v) = o$, so

$$\begin{aligned}
\beta_o[\rho'](o) &= \{v' : \rho'(v') = o\} \\
&= \{v' : \rho(v') = o\} \cup \{v\} \\
&= \beta_o[\rho](o) \cup \{v\}
\end{aligned}$$

If $o' \neq o$, then $\rho'(v) \neq o$, so

$$\begin{aligned}
\beta_o[\rho'](o) &= \{v' : \rho'(v') = o\} \\
&= \{v' : \rho(v') = o\} \setminus \{v\} \\
&= \beta_o[\rho](o) \setminus \{v\}
\end{aligned}$$

In either case,

$$\begin{aligned}
& o R_d[\rho] \langle o^1, o^2 \rangle \\
& \implies o^1 \subseteq \beta_o[\rho](o) \subseteq o^2 \\
& \implies o^1 \setminus \{v\} \subseteq \beta_o[\rho](o) \setminus \{v\} \subseteq \beta_o[\rho](o) \cup \{v\} \subseteq o^2 \cup \{v\} \\
& \implies o^1 \setminus \{v\} \subseteq \beta_o[\rho'](o) \subseteq o^2 \cup \{v\} \\
& \implies o R_d[\rho'] \langle o^1 \setminus \{v\}, o^2 \cup \{v\} \rangle \quad \text{defn. of } R_d[\rho] \\
& \implies o R_d[\rho'] \llbracket s \rrbracket_{d^\#}(\langle o^1, o^2 \rangle) \quad \text{defn. of } \llbracket s \rrbracket_{d^\#}
\end{aligned}$$

Subcase $s \in \{v \leftarrow \mathbf{null}, v \leftarrow \mathbf{new}\}$:

$$\begin{aligned}
& o R_d[\rho] \langle o^!, o^? \rangle \\
& \implies o^! \subseteq \beta_o[\rho](o) \subseteq o^? \\
& \implies o^! \setminus \{v\} \subseteq \beta_o[\rho](o) \setminus \{v\} \subseteq o^? \setminus \{v\} \\
& \implies o^! \setminus \{v\} \subseteq \llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o)) \subseteq o^? \setminus \{v\} && \text{defn. of } \llbracket s \rrbracket_{o^\#} \\
& \implies o^! \setminus \{v\} \subseteq \beta_o[\rho'](o) \subseteq o^? \setminus \{v\} && \text{Proposition 2} \\
& \implies o R_d[\rho'] \langle o^! \setminus \{v\}, o^? \setminus \{v\} \rangle && \text{defn. of } R_d[\rho] \\
& \implies o R_d[\rho'] \llbracket s \rrbracket_{d^\#}(\langle o^!, o^? \rangle) && \text{defn. of } \llbracket s \rrbracket_{d^\#}
\end{aligned}$$

Subcase $s \in \{\mathbf{h} \leftarrow v, \mathbf{body}\}$:

$$\begin{aligned}
& o R_d[\rho] \langle o^!, o^? \rangle \\
& \implies o^! \subseteq \beta_o[\rho](o) \subseteq o^? \\
& \implies o^! \subseteq \llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o)) \subseteq o^? && \text{defn. of } \llbracket s \rrbracket_{o^\#} \\
& \implies o^! \subseteq \beta_o[\rho'](o) \subseteq o^? && \text{Proposition 2} \\
& \implies o R_d[\rho'] \langle o^!, o^? \rangle && \text{defn. of } R_d[\rho] \\
& \implies o R_d[\rho'] \llbracket s \rrbracket_{d^\#}(\langle o^!, o^? \rangle) && \text{defn. of } \llbracket s \rrbracket_{d^\#}
\end{aligned}$$

Case d is a negative binding \bar{O} : Then by Lemma 9, $d^\#$ is a negative binding $d^\# = \bar{V}^\# \subseteq \bigcup_{o \in \bar{O}} \beta_o[\rho](o)$.

Subcase $s = v_1 \leftarrow v_2 \wedge v_2 \in \bar{V}^\#$:

$$\bar{O} R_d[\rho] \bar{V}^\# \implies \bar{V}^\# \subseteq \bigcup_{o \in \bar{O}} \beta_o[\rho](o)$$

Therefore, there is some $o' \in \bar{O}$ for which $v_2 \in \beta_o[\rho](o')$.

So $\beta_o[\rho'](o') = \llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o')) = \beta_o[\rho](o') \cup \{v_1\}$.

$$\begin{aligned}
& \bar{V}^\# \subseteq \bigcup_{o \in \bar{O}} \beta_o[\rho](o) \\
& \implies \bar{V}^\# \subseteq \left(\bigcup_{o \in \bar{O}} \beta_o[\rho](o) \right) \cup \beta_o[\rho](o') \\
& \implies \bar{V}^\# \cup \{v_1\} \subseteq \left(\bigcup_{o \in \bar{O}} \beta_o[\rho](o) \right) \cup \beta_o[\rho](o') \cup \{v_1\} \\
& \implies \bar{V}^\# \cup \{v_1\} \subseteq \left(\bigcup_{o \in \bar{O}} \beta_o[\rho](o) \setminus \{v_1\} \right) \cup \beta_o[\rho](o') \cup \{v_1\} \\
& \implies \bar{V}^\# \cup \{v_1\} \subseteq \left(\bigcup_{o \in \bar{O}} \llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o)) \right) \cup \llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o')) \\
& \implies \bar{V}^\# \cup \{v_1\} \subseteq \bigcup_{o \in \bar{O}} \llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o)) \\
& \implies \bar{V}^\# \cup \{v_1\} \subseteq \bigcup_{o \in \bar{O}} \beta_o[\rho'](o) \\
& \implies \bar{O} R_d[\rho'] \bar{V}^\# \cup \{v_1\} && \text{defn. of } R_d[\rho] \\
& \implies \bar{O} R_d[\rho'] \llbracket s \rrbracket_{d^\#}(\bar{V}^\#) && \text{defn. of } \llbracket s \rrbracket_{d^\#}
\end{aligned}$$

Subcase $s = v_1 \leftarrow v_2 \wedge v_2 \notin \bar{V}^\#$:

From the definition of $\llbracket s \rrbracket_{o^\#}$, it follows that $\llbracket v_1 \leftarrow v_2 \rrbracket_{o^\#}(o^\#) \supseteq o^\# \setminus \{v_1\}$.

$$\begin{aligned}
& \overline{O} R_d[\rho] \overline{V^\#} \\
& \implies \overline{V^\#} \subseteq \bigcup_{o \in \overline{O}} \beta_o[\rho](o) \\
& \implies \overline{V^\#} \setminus \{v_1\} \subseteq \bigcup_{o \in \overline{O}} \beta_o[\rho](o) \setminus \{v_1\} \\
& \implies \overline{V^\#} \setminus \{v_1\} \subseteq \bigcup_{o \in \overline{O}} \llbracket s \rrbracket_{o^\#}(o^\#) \\
& \implies \overline{V^\#} \setminus \{v_1\} \subseteq \bigcup_{o \in \overline{O}} \beta_o[\rho'](o) \\
& \implies \overline{O} R_d[\rho'] \overline{V^\#} \setminus \{v_1\} && \text{defn. of } R_d[\rho] \\
& \implies \overline{O} R_d[\rho'] \llbracket s \rrbracket_{d^\#}(\overline{V^\#}) && \text{defn. of } \llbracket s \rrbracket_{d^\#}
\end{aligned}$$

Subcase $s \in \{v \leftarrow \mathbf{null}, v \leftarrow \mathbf{new}\}$:

$$\begin{aligned}
& \overline{O} R_d[\rho] \overline{V^\#} \\
& \implies \overline{V^\#} \subseteq \bigcup_{o \in \overline{O}} \beta_o[\rho](o) \\
& \implies \overline{V^\#} \setminus \{v\} \subseteq \bigcup_{o \in \overline{O}} (\beta_o[\rho](o) \setminus \{v\}) \\
& \implies \overline{V^\#} \setminus \{v\} \subseteq \bigcup_{o \in \overline{O}} \llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o)) && \text{defn. of } \llbracket s \rrbracket_{o^\#} \\
& \implies \overline{V^\#} \setminus \{v\} \subseteq \bigcup_{o \in \overline{O}} \beta_o[\rho'](o) && \text{Proposition 2} \\
& \implies \overline{O} R_d[\rho'] \overline{V^\#} \setminus \{v\} && \text{defn. of } R_d[\rho] \\
& \implies \overline{O} R_d[\rho'] \llbracket s \rrbracket_{d^\#}(\overline{V^\#}) && \text{defn. of } \llbracket s \rrbracket_{d^\#}
\end{aligned}$$

Subcase $s = v \leftarrow \mathbf{h}$:

As in the case for positive bindings, $\llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o))$ is either $\beta_o[\rho](o) \cup \{v\}$ or $\beta_o[\rho](o) \setminus \{v\}$. Either way, $\llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o)) \supseteq \beta_o[\rho](o) \setminus \{v\}$. Thus, the same reasoning as in the preceding subcase applies.

Subcase $s \in \{\mathbf{h} \leftarrow v, \mathbf{body}\}$:

$$\begin{aligned}
& \overline{O} R_d[\rho] \overline{V^\#} \\
& \implies \overline{V^\#} \subseteq \bigcup_{o \in \overline{O}} \beta_o[\rho](o) \\
& \implies \overline{V^\#} \subseteq \bigcup_{o \in \overline{O}} \llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o)) && \text{defn. of } \llbracket s \rrbracket_{o^\#} \\
& \implies \overline{V^\#} \subseteq \bigcup_{o \in \overline{O}} \beta_o[\rho'](o) && \text{Proposition 2} \\
& \implies \overline{O} R_d[\rho'] \overline{V^\#} && \text{defn. of } R_d[\rho] \\
& \implies \overline{O} R_d[\rho'] \llbracket s \rrbracket_{d^\#}(\overline{V^\#}) && \text{defn. of } \llbracket s \rrbracket_{d^\#}
\end{aligned}$$

□

Theorem 3. *If $\langle s, \rho, h, \sigma \rangle \rightarrow \langle \rho', h', \sigma' \rangle$ and $\sigma R_\sigma[\rho] \sigma^\#$, then $\sigma' R_{\sigma'}[\rho'] \llbracket s \rrbracket_{\sigma^\#}[\rho^\#](\sigma^\#)$.*

For ease of reference we restate the correctness relation:

$$\begin{array}{ll}
d R_d[\rho] d^\# & \text{if } \beta_d[\rho](d) \sqsubseteq d^\# \\
\langle q, m \rangle R_m[\rho] \langle q, m^\# \rangle & \text{if } \forall f \in F. m(f) R_d[\rho] m^\#(f) \\
\sigma R_\sigma[\rho] \sigma^\# & \text{if } \forall \langle q, m \rangle \in \sigma. \exists \langle q, m^\# \rangle \in \sigma^\#. \langle q, m \rangle R_m[\rho] \langle q, m^\# \rangle
\end{array}$$

Where $\beta_d[\rho](d)$ is defined as:

$$\beta_d[\rho](d) \triangleq \begin{cases} \perp & \text{if } d = \perp \\ \langle \beta_o[\rho](o), \beta_o[\rho](o) \rangle & \text{if } d \text{ is a positive binding } o \in \mathbf{Obj} \\ \bigcup_{o \in \overline{O}} \beta_o[\rho](o) & \text{if } d \text{ is a negative binding } \overline{O} \subseteq \mathbf{Obj} \end{cases}$$

We divide the proof of the theorem into the following six lemmas. The theorem is the combination of Lemmas 10 and 15.

Lemma 10. *For all statements except $\mathbf{tr}(T)$, if $\langle s, \rho, h, \sigma \rangle \rightarrow \langle \rho', h', \sigma' \rangle$ and $\sigma R_\sigma[\rho] \sigma^\#$, then $\sigma' R_\sigma[\rho'] \llbracket s \rrbracket_{\sigma^\#}[\rho^\#](\sigma^\#)$.*

Proof. Notice that all statements except $\mathbf{tr}(T)$ leave the tracematch state abstraction unchanged. This means that $\sigma = \sigma'$.

$$\begin{array}{ll}
\sigma R_\sigma[\rho] \sigma^\# & \\
\implies \forall \langle q, m \rangle \in \sigma. \exists \langle q, m^\# \rangle \in \sigma^\#. \langle q, m \rangle R_m[\rho] \langle q, m^\# \rangle & \\
\implies \forall \langle q, m \rangle \in \sigma. \exists \langle q, m^\# \rangle \in \sigma^\#. \forall f \in F. m(f) R_d[\rho] m^\#(f) & \\
\implies \forall \langle q, m \rangle \in \sigma. \exists \langle q, m^\# \rangle \in \sigma^\#. \forall f \in F. m(f) R_d[\rho'] \llbracket s \rrbracket_{d^\#}(m^\#(f)) & \text{Proposition 5} \\
\implies \forall \langle q, m \rangle \in \sigma. \exists \langle q, m^\# \rangle \in \sigma^\#. \langle q, m \rangle R_m[\rho'] \langle q, \lambda f. \llbracket s \rrbracket_{d^\#}(m^\#(f)) \rangle & \\
\implies \forall \langle q, m \rangle \in \sigma. \exists \langle q, m^\# \rangle \in \llbracket s \rrbracket_{\sigma^\#}[\rho^\#](\sigma^\#). \langle q, m \rangle R_m[\rho'] \langle q, m^\# \rangle & \\
\implies \sigma R_\sigma[\rho'] \llbracket s \rrbracket_{\sigma^\#}[\rho^\#](\sigma^\#) & \text{defn. of } \llbracket s \rrbracket_{\sigma^\#} \\
\implies \sigma' R_\sigma[\rho'] \llbracket s \rrbracket_{\sigma^\#}(\sigma^\#) & \text{since } \sigma' = \sigma \\
& \square
\end{array}$$

Lemma 11. *If $d_1 R_d[\rho] d_1^\#$ and $d_2 R_d[\rho] d_2^\#$, then $d_1 \sqcap d_2 R_d[\rho] d_1^\# \sqcap d_2^\#$.*

Proof. Since $d_1 \sqcap d_2 \sqsubseteq d_1$, by Proposition 4, $\beta_d[\rho](d_1 \sqcap d_2) \sqsubseteq \beta_d[\rho](d_1) \sqsubseteq d_1^\#$. Similarly, $\beta_d[\rho](d_1 \sqcap d_2) \sqsubseteq d_2^\#$. Therefore, $\beta_d[\rho](d_1 \sqcap d_2) \sqsubseteq d_1^\# \sqcap d_2^\#$. Thus, $d_1 \sqcap d_2 R_d[\rho] d_1^\# \sqcap d_2^\#$. \square

Lemma 12. *Let o_1, o_2 be two concrete objects existing simultaneously at any state in the program execution with environment ρ . If $o_1 R_d[\rho] o_1^\#$ and $o_2 R_d[\rho] o_2^\#$, then*

1. $o_1 = o_2 \implies \text{same}(o_1^{!?}, o_2^{!?})$
2. $o_1 \neq o_2 \implies \text{diff}(o_1^{!?}, o_2^{!?})$
3. In either case, $\text{compatible}(o_1^{!?}, o_2^{!?})$.

As a corollary, for any set $\{o_1 \cdots o_n\}$ of concrete objects, if $o_i \text{ Rd}[\rho] o_i^{!?}$ for all i , then $\text{setcompat}(\{o_i^{!?}\})$.

- Proof.*
1. From the correctness relation, $o_i^! \subseteq \beta_o[\rho](o_i) \subseteq o_i^?$ for $i \in \{1, 2\}$. Since $o_1 = o_2$, $o_1^! \subseteq \beta_o[\rho](o_i) \subseteq o_2^?$. Similarly, $o_2^! \subseteq \beta_o[\rho](o_i) \subseteq o_1^?$. This is the definition of $\text{same}(o_1^{!?}, o_2^{!?})$.
 2. If $o_1 = \rho(v)$, then $o_2 \neq \rho(v)$, and vice versa. Therefore, $\beta_o[\rho](o_1) \cap \beta_o[\rho](o_2) = \emptyset$. Since $o_i^! \subseteq \beta_o[\rho](o_i)$ for $i \in \{1, 2\}$, $o_1^! \cap o_2^! \subseteq \emptyset$.
 3. Immediate from the above two cases and the definition of *compatible*.

□

Definition 2 Given $\rho \in \mathcal{P}(\mathbf{Var})$, $V \subseteq \mathbf{Var}$ such that $\rho(v) \neq \perp$ for any $v \in V$, define $O^\sharp(\rho, V) \triangleq \{\beta_o[\rho](\rho(v)) : v \in V\}$.

Lemma 13. Let $\rho^\sharp \sqsupseteq \beta_\rho(\rho, h)$ and $V \subseteq \mathbf{Var}$. Then

1. $O^\sharp(\rho, V) \subseteq \rho^\sharp$
2. $\text{relevant}(O^\sharp(\rho, V), V)$
3. $\text{lookup}(O^\sharp(\rho, V), v) = \beta_o[\rho](\rho(v))$ for all $v \in V$

Proof. 1.

$$\begin{aligned}
O^\sharp(\rho, V) &= \{\beta_o[\rho](o) : v \in V \wedge \rho(v) = o\} \\
&\subseteq \{\beta_o[\rho](o) : o \in \text{range}(\rho) \cup h\} \\
&= \beta_\rho(\rho, h) \\
&\subseteq \rho^\sharp
\end{aligned}$$

2.

$$\begin{aligned}
\bigcup_{o^\# \in O^\#(\rho, V)} o^\# &= \bigcup_{v \in V} \beta_o[\rho](\rho(v)) \\
&\supseteq \bigcup_{v \in V} \{v\} \\
&= V
\end{aligned}$$

Every $o^\# \in O^\#(\rho, V)$ is $\beta_o[\rho](\rho(v))$ for some $v \in V$. By definition of β_o , $v \in \beta_o[\rho](\rho(v))$. Therefore, $v \in \beta_o[\rho](\rho(v)) \cap V$, so this intersection is not empty.

3. For all $v \in V$, $O^\#(\rho, V)$ contains $\beta_o[\rho](\rho(v))$. Also, $v \in \beta_o[\rho](\rho(v))$. Therefore, $\beta_o[\rho](\rho(v))$ satisfies the definition of $lookup(O^\#(\rho, V), v)$. Furthermore, $\beta_o[\rho](\rho(v))$ is the only such element of $O^\#(\rho, V)$, since for any other object $o' \neq \rho(v)$, $v \notin \beta_o[\rho](o')$. \square

Lemma 14. *Let V be any set of variables such that $range(b) \subseteq V$. Then*

1. $e^+(b, \rho) R_m[\rho] e^{+\#}(b, O^\#(\rho, V))$
2. $e^-(b, \rho, f) R_m[\rho] e^{-\#}(b, O^\#(\rho, V), f)$

Proof. 1. For $f \in \text{dom}(b)$,

$$\begin{aligned}
e^{+\#}(b, O^\#(\rho, V))(f) &= \langle lookup(O^\#(\rho, V), b(f)), lookup(O^\#(\rho, V), b(f)) \rangle && \text{definition of } e^{+\#} \\
&= \langle \beta_o[\rho](\rho(b(f))), \beta_o[\rho](\rho(b(f))) \rangle && \text{Lemma 13} \\
&= \beta_d[\rho](\rho(b(f))) && \text{definition of } \beta_d
\end{aligned}$$

Therefore, $e^+(b, \rho) = \rho(b(f)) R_d[\rho] \beta_d[\rho](\rho(b(f))) = e^{+\#}(b, O^\#(\rho, V))(f)$.

For $f \notin \text{dom}(b)$, $e^+(b, \rho)(f) = \top R_d[\rho] \top = e^{+\#}(b, O^\#(\rho, V))(f)$.

2. For $f \in \text{dom}(b)$,

$$\begin{aligned}
e^{-\#}(b, O^\#(\rho, V), f) &= \overline{lookup(O^\#(\rho, V), b(f))} && \text{definition of } e^{-\#} \\
&= \overline{\beta_o[\rho](\rho(b(f)))} && \text{Lemma 13} \\
&= \beta_d[\rho](\{\rho(b(f))\}) && \text{definition of } \beta_d
\end{aligned}$$

Therefore, $e^-(b, \rho, f) = \overline{\{\rho(b(f))\}} R_d[\rho] \beta_d[\rho](\{\rho(b(f))\}) = e^{-\#}(b, O^\#(\rho, V), f)$.

For $f \notin \text{dom}(b)$, $e^-(b, \rho, f) = \top R_d[\rho] \top = e^{-\#}(b, O^\#(\rho, V), f)$. \square

Lemma 15. *If $\langle \mathbf{tr}(T), \rho, h, \sigma \rangle \rightarrow \langle \rho', h', \sigma' \rangle$ and $\sigma R_\sigma[\rho] \sigma^\sharp$, then $\sigma' R_\sigma[\rho'] \llbracket \mathbf{tr}(T) \rrbracket[\rho^\sharp](\sigma^\sharp)$ for any $\rho^\sharp \sqsupseteq \beta_\rho(\rho, h)$.*

Proof. For any $V \supseteq \text{range}(b)$, from Lemmas 11 and 14 and from the premise that $\sigma R_\sigma[\rho] \sigma^\sharp$, it follows that for every $\langle q, m \rangle \in \sigma \cup \{\langle q_0, \lambda f. \top \rangle\}$ there is a $\langle q, m^\sharp \rangle \in \sigma^\sharp \cup \{\langle q_0, \lambda f. \top \rangle\}$ such that:

$$\begin{aligned} e^+[a, b, \rho](\langle q, m \rangle) R_\sigma[\rho] e^{+\sharp}[a, b, O^\sharp(\rho, V)](\langle q, m^\sharp \rangle) \\ e^-[b, \rho](\langle q, m \rangle) R_\sigma[\rho] e^{-\sharp}[b, O^\sharp(\rho, V)](\langle q, m^\sharp \rangle) \end{aligned}$$

By Lemma 12, $\text{setcompat}(\text{objs}(m^\sharp) \cup \{\langle o^\sharp, o^\sharp \rangle : o^\sharp \in O^\sharp(\rho, V)\})$. By Lemma 13, $O^\sharp(\rho, V) \subseteq \rho^\sharp$ and $\text{relevant}(O^\sharp(\rho, V), V)$. Thus, $O^\sharp(\rho, V) \in \text{red-enuvs}(\rho^\sharp, \text{objs}(m^\sharp), V)$.

Therefore, for each

$$\langle q, m \rangle \in \sigma' = e^+[a, b, \rho](\sigma \cup \{\langle q_0, \lambda f. \top \rangle\}) \cup e^-[b, \rho](\sigma \cup \{\langle q_0, \lambda f. \top \rangle\})$$

there exists $\langle q, m^\sharp \rangle \in$

$$\bigcup_{O^\sharp \in \text{red-enuvs}(\rho^\sharp, \text{objs}(m^\sharp), \text{range}(b))} e^{+\sharp}[a, b, O^\sharp](\sigma^\sharp \cup \{\langle q_0, \lambda f. \top \rangle\}) \cup e^{-\sharp}[b, O^\sharp](\sigma^\sharp \cup \{\langle q_0, \lambda f. \top \rangle\})$$

$= \llbracket \mathbf{tr}(\{\langle a, b \rangle\}) \rrbracket_{\sigma^\sharp}[\rho^\sharp](\sigma^\sharp)$ such that $\langle q, m \rangle R_m[\rho] \langle q, m^\sharp \rangle$. The same correspondence holds for the case when T contains multiple transition elements.

This is the definition of $\sigma' R_\sigma[\rho] \llbracket \mathbf{tr}(T) \rrbracket[\rho^\sharp](\sigma^\sharp)$. Since $\rho' = \rho$, $\sigma' R_\sigma[\rho'] \llbracket \mathbf{tr}(T) \rrbracket[\rho^\sharp](\sigma^\sharp)$. \square

Proposition 6.

$$\begin{aligned} \llbracket s \rrbracket_{\rho^\sharp}(\rho^\sharp, h^\sharp) &= \left\{ o^\sharp : \rho[o^\sharp] \in \bigcup_{d \in \text{decomp}(\rho^\sharp, h^\sharp) \cup \{0\}} \llbracket s \rrbracket_{\rho h^\sharp}(d) \right\} \\ \llbracket s \rrbracket_{h^\sharp}(\rho^\sharp, h^\sharp) &= \left\{ o^\sharp : h[o^\sharp] \in \bigcup_{d \in \text{decomp}(\rho^\sharp, h^\sharp) \cup \{0\}} \llbracket s \rrbracket_{\rho h^\sharp}(d) \right\} \end{aligned}$$

Proof. **Case $s = v \leftarrow \mathbf{h}$:** In this case,

$$\begin{aligned} \llbracket s \rrbracket_{\rho h^\#}(\rho[o^\#]) &= \{\rho[o^\# \setminus \{v\}]\} \\ \llbracket s \rrbracket_{\rho h^\#}(h[o^\#]) &= \{\rho[o^\# \setminus \{v\}], \rho[o^\# \cup \{v\}], h[o^\# \setminus \{v\}], h[o^\# \cup \{v\}]\} \\ \llbracket s \rrbracket_{\rho h^\#}(0) &= \emptyset \end{aligned}$$

Therefore,

$$\begin{aligned} \left\{ o^\# : \rho[o^\#] \in \bigcup_{d \in \text{decomp}(\rho^\#, h^\#) \cup \{0\}} \llbracket s \rrbracket_{\rho h^\#}(d) \right\} &= \bigcup_{o^\# \in \rho^\#} \{o^\# \setminus \{v\}\} \cup \bigcup_{o^\# \in h^\#} \{o^\# \setminus \{v\}, o^\# \cup \{v\}\} \\ &= \bigcup_{o^\# \in \rho^\# \cup h^\#} \text{focus}[h^\#](v, o^\#) \\ &= \bigcup_{o^\# \in \rho^\#} \text{focus}[h^\#](v, o^\#) = \llbracket s \rrbracket_{\rho^\#}(\rho^\#, h^\#) \end{aligned}$$

Also,

$$\begin{aligned} \left\{ o^\# : h[o^\#] \in \bigcup_{d \in \text{decomp}(\rho^\#, h^\#) \cup \{0\}} \llbracket s \rrbracket_{\rho h^\#}(d) \right\} &= \bigcup_{o^\# \in h^\#} \{o^\# \setminus \{v\}, o^\# \cup \{v\}\} \\ &= \bigcup_{o^\# \in h^\#} \text{focus}[h^\#](v, o^\#) = \llbracket s \rrbracket_{h^\#}(\rho^\#, h^\#) \end{aligned}$$

Case $s = \mathbf{h} \leftarrow v$: In this case,

$$\begin{aligned} \llbracket s \rrbracket_{\rho h^\#}(\rho[o^\#]) &= \begin{cases} \{\rho[o^\#], h[o^\#]\} & \text{if } v \in o^\# \\ \{\rho[o^\#]\} & \text{if } v \notin o^\# \end{cases} \\ \llbracket s \rrbracket_{\rho h^\#}(h[o^\#]) &= \{h[\llbracket s \rrbracket_{o^\#}(o^\#)]\} \\ \llbracket s \rrbracket_{\rho h^\#}(0) &= \emptyset \end{aligned}$$

Therefore,

$$\begin{aligned} \left\{ o^\# : \rho[o^\#] \in \bigcup_{d \in \text{decomp}(\rho^\#, h^\#) \cup \{0\}} \llbracket s \rrbracket_{\rho h^\#}(d) \right\} &= \bigcup_{o^\# \in \rho^\#} \{o^\#\} = \bigcup_{o^\# \in \rho^\#} \{\llbracket s \rrbracket_{o^\#}(o^\#)\} \\ &= \{\llbracket s \rrbracket_{o^\#}(o^\#) : o^\# \in \rho^\#\} = \llbracket s \rrbracket_{\rho^\#}(\rho^\#, h^\#) \end{aligned}$$

Also,

$$\begin{aligned}
\left\{ o^\# : h[o^\#] \in \bigcup_{d \in \text{decomp}(\rho^\#, h^\#) \cup \{0\}} \llbracket s \rrbracket_{\rho h^\#}(d) \right\} &= \bigcup_{o^\# \in h^\#} \{ \llbracket s \rrbracket_{o^\#}(o^\#) \} \cup \bigcup_{o^\# \in \rho^\# : v \in o^\#} \{ o^\# \} \\
&= \bigcup_{o^\# \in h^\# \cup \{ o^\# \in \rho^\# : v \in o^\# \}} \{ \llbracket s \rrbracket_{o^\#}(o^\#) \} \\
&= \{ \llbracket s \rrbracket_{o^\#}(o^\#) : o^\# \in h^\# \cup \{ o^\# \in \rho^\# : v \in o^\# \} \} \\
&= \llbracket s \rrbracket_{h^\#}(\rho^\#, h^\#)
\end{aligned}$$

Case $s = v \leftarrow \text{new}$: In this case,

$$\begin{aligned}
\llbracket s \rrbracket_{\rho h^\#}(\rho[o^\#]) &= \{ \rho[\llbracket s \rrbracket_{o^\#}(o^\#)] \} \\
\llbracket s \rrbracket_{\rho h^\#}(h[o^\#]) &= \{ h[\llbracket s \rrbracket_{o^\#}(o^\#)] \} \\
\llbracket s \rrbracket_{\rho h^\#}(0) &= \{ \rho[\{v\}] \}
\end{aligned}$$

Therefore,

$$\begin{aligned}
\left\{ o^\# : \rho[o^\#] \in \bigcup_{d \in \text{decomp}(\rho^\#, h^\#) \cup \{0\}} \llbracket s \rrbracket_{\rho h^\#}(d) \right\} &= \bigcup_{o^\# \in \rho^\#} \{ \llbracket s \rrbracket_{o^\#}(o^\#) \} \cup \{ \{v\} \} \\
&= \{ \llbracket s \rrbracket_{o^\#}(o^\#) : o^\# \in \rho^\# \} \cup \{ \{v\} \} \\
&= \llbracket s \rrbracket_{\rho^\#}(\rho^\#, h^\#)
\end{aligned}$$

Also,

$$\begin{aligned}
\left\{ o^\# : h[o^\#] \in \bigcup_{d \in \text{decomp}(\rho^\#, h^\#) \cup \{0\}} \llbracket s \rrbracket_{\rho h^\#}(d) \right\} &= \bigcup_{o^\# \in h^\#} \{ \llbracket s \rrbracket_{o^\#}(o^\#) \} \\
&= \{ \llbracket s \rrbracket_{o^\#}(o^\#) : o^\# \in h^\# \} = \llbracket s \rrbracket_{h^\#}(\rho^\#, h^\#)
\end{aligned}$$

Case s is any other statement: In this case,

$$\begin{aligned}
\llbracket s \rrbracket_{\rho h^\#}(\rho[o^\#]) &= \{ \rho[\llbracket s \rrbracket_{o^\#}(o^\#)] \} \\
\llbracket s \rrbracket_{\rho h^\#}(h[o^\#]) &= \{ h[\llbracket s \rrbracket_{o^\#}(o^\#)] \} \\
\llbracket s \rrbracket_{\rho h^\#}(0) &= \emptyset
\end{aligned}$$

Therefore,

$$\left\{ o^\sharp : \rho[o^\sharp] \in \bigcup_{d \in \text{decomp}(\rho^\sharp, h^\sharp) \cup \{0\}} \llbracket s \rrbracket_{\rho h^\sharp}(d) \right\} = \bigcup_{o^\sharp \in \rho^\sharp} \{ \llbracket s \rrbracket_{o^\sharp}(o^\sharp) \} \\ = \{ \llbracket s \rrbracket_{o^\sharp}(o^\sharp) : o^\sharp \in \rho^\sharp \} = \llbracket s \rrbracket_{\rho^\sharp}(\rho^\sharp, h^\sharp)$$

Also,

$$\left\{ o^\sharp : h[o^\sharp] \in \bigcup_{d \in \text{decomp}(\rho^\sharp, h^\sharp) \cup \{0\}} \llbracket s \rrbracket_{\rho h^\sharp}(d) \right\} = \bigcup_{o^\sharp \in h^\sharp} \{ \llbracket s \rrbracket_{o^\sharp}(o^\sharp) \} \\ = \{ \llbracket s \rrbracket_{o^\sharp}(o^\sharp) : o^\sharp \in h^\sharp \} = \llbracket s \rrbracket_{h^\sharp}(\rho^\sharp, h^\sharp)$$

□

Proposition 7. *If $\langle q_2, m_2 \rangle \in e^\sharp[T, \rho](q_1, m_1)$; $\langle q_1, m_1 \rangle \neq \langle q_2, m_2 \rangle$; ρ^\sharp overapproximates ρ ; and $\langle q_1, m_1 \rangle R_m[\rho] \langle q_1^\sharp, m_1^\sharp \rangle$; then there exists $\langle q_2^\sharp, m_2^\sharp \rangle \in \llbracket \mathbf{tr}(T) \rrbracket_{m^\sharp}[\rho^\sharp](q_1^\sharp, m_1^\sharp)$ such that $\langle q_1^\sharp, m_1^\sharp \rangle \neq \langle q_2^\sharp, m_2^\sharp \rangle$ and $\langle q_2, m_2 \rangle R_m[\rho] \langle q_2^\sharp, m_2^\sharp \rangle$.*

Proof. From the definition of the correctness relation $R_m[\rho]$, $q_1 = q_1^\sharp$ and $q_2 = q_2^\sharp$. If $q_1 \neq q_2$, the conclusion is immediate. Suppose instead that all the q_i, q_i^\sharp are equal, and call this common state q . Then $m_1 \neq m_2$. From the definition of e^\sharp , $\langle q, m_2 \rangle$ is in either $e^+[a, b, \rho](q, m_1)$ or in $e^-[b, \rho](q, m_1)$ for some $\langle a, b \rangle \in T$. Thus there exists an $f \in F$ such that $m_1(f) \neq m_2(f)$ and either $m_2(f) = m_1(f) \sqcap \rho(b(f))$ or $m_2(f) = m_1(f) \sqcap \{\rho(b(f))\}$. Also, $m_1(f) \neq \perp$, since then $m_2(f)$ would also have to be \perp .

Case: $m_2(f) = m_1(f) \sqcap \rho(b(f))$ and $m_1^\sharp(f)$ is a positive binding containing $\rho(b(f))$ in its must set

In this case, since $m_1(f) R_d[\rho] m_1^\sharp(f)$, $m_1(f) = \rho(b(f))$, so $m_2(f) = m_1(f)$, a contradiction. Therefore this case cannot occur.

Case: $m_2(f) = m_1(f) \sqcap \rho(b(f))$ and $m_1^\sharp(f)$ is a positive binding not containing $\rho(b(f))$ in its must set

In this case,

$$m_1^\sharp(f) \sqcap e_0^{+\sharp}(b, O^\sharp(\rho, \text{range}(b))) \\ = m_1^\sharp(f) \sqcap \langle n(O^\sharp(\rho, \text{range}(b)), b(f)), n(O^\sharp(\rho, \text{range}(b)), b(f)) \rangle$$

which contains $\rho(b(f))$ in its must set and is therefore distinct from $m_1^\sharp(f)$. Therefore $m_1^\sharp \sqcap e_0^{+\sharp}(b, O^\sharp(\rho, \text{range}(b)))$ is a correct abstraction of m_2 , is distinct from m_1^\sharp , and is contained in $\llbracket \mathbf{tr}(T) \rrbracket_{m^\sharp}[\rho^\sharp](q_1^\sharp, m_1^\sharp)$.

Case: $m_2(f) = m_1(f) \sqcap \rho(b(f))$ and $m_1^\sharp(f)$ is a negative binding containing $b(f)$

In this case,

$$\begin{aligned} & m_1^\sharp(f) \sqcap e_0^{+\sharp}(b, O^\sharp(\rho, \text{range}(b))) \\ &= m_1^\sharp(f) \sqcap \langle n(O^\sharp(\rho, \text{range}(b)), b(f)), n(O^\sharp(\rho, \text{range}(b)), b(f)) \rangle = \perp \end{aligned}$$

which is distinct from $m_1^\sharp(f)$. Therefore $m_1^\sharp \sqcap e_0^{+\sharp}(b, O^\sharp(\rho, \text{range}(b)))$ is a correct abstraction of m_2 , is distinct from m_1^\sharp , and is contained in $\llbracket \mathbf{tr}(T) \rrbracket_{m^\sharp}[\rho^\sharp](q_1^\sharp, m_1^\sharp)$.

Case: $m_2(f) = m_1(f) \sqcap \rho(b(f))$ and $m_1^\sharp(f)$ is a negative binding not containing $b(f)$

In this case,

$$\begin{aligned} & m_1^\sharp(f) \sqcap e_0^{+\sharp}(b, O^\sharp(\rho, \text{range}(b))) \\ &= m_1^\sharp(f) \sqcap \langle n(O^\sharp(\rho, \text{range}(b)), b(f)), n(O^\sharp(\rho, \text{range}(b)), b(f)) \rangle \end{aligned}$$

which is a positive binding and is therefore distinct from $m_1^\sharp(f)$. Therefore $m_1^\sharp \sqcap e_0^{+\sharp}(b, O^\sharp(\rho, \text{range}(b)))$ is a correct abstraction of m_2 , is distinct from m_1^\sharp , and is contained in $\llbracket \mathbf{tr}(T) \rrbracket_{m^\sharp}[\rho^\sharp](q_1^\sharp, m_1^\sharp)$.

Case: $m_2(f) = m_1(f) \sqcap \overline{\{\rho(b(f))\}}$ and $m_1^\sharp(f)$ is a positive binding containing $b(f)$ in its may set:

In this case,

$$\begin{aligned} & m_1^\sharp(f) \sqcap e_0^{+\sharp}(b, O^\sharp(\rho, \text{range}(b))) \\ &= m_1^\sharp(f) \sqcap \overline{n(O^\sharp(\rho, \text{range}(b)), b(f))} \end{aligned}$$

which is either \perp or a positive binding not containing $\rho(b(f))$ in its may set. Either way, it is distinct from $m_1^\sharp(f)$. Therefore $m_1^\sharp \sqcap e_0^{+\sharp}(b, O^\sharp(\rho, \text{range}(b)))$ is a correct abstraction of m_2 , is distinct from m_1^\sharp , and is contained in $\llbracket \mathbf{tr}(T) \rrbracket_{m^\sharp}[\rho^\sharp](q_1^\sharp, m_1^\sharp)$.

Case: $m_2(f) = m_1(f) \sqcap \overline{\{\rho(b(f))\}}$ and $m_1^\sharp(f)$ is a positive binding not containing $b(f)$ in its may set

In this case, since $m_1(f) R_d[\rho] m_1^\sharp(f)$, $m_1(f)$ is a positive binding other than $\rho(b(f))$, so $m_2(f) = m_1(f)$, a contradiction. Therefore this case cannot occur.

Case: $m_2(f) = m_1(f) \sqcap \overline{\{\rho(b(f))\}}$ and $m_1^\sharp(f)$ is a negative binding containing $b(f)$

In this case, since

$m_1(f) R_d[\rho] m_1^\sharp(f)$, $m_1(f)$ is either a positive binding other than $\rho(b(f))$ or a negative binding containing $\rho(b(f))$. Either way, $m_2(f) = m_1(f)$, a contradiction. Therefore this case cannot occur.

Case: $m_2(f) = m_1(f) \sqcap \overline{\{\rho(b(f))\}}$ and $m_1^\sharp(f)$ is a negative binding not containing $b(f)$ In this case,

$m_1^\sharp(f) \sqcap e_0^{+\sharp}(b, O^\sharp(\rho, \text{range}(b))) = m_1^\sharp(f) \sqcap \overline{n(O^\sharp(\rho, \text{range}(b)), b(f))}$, which is a negative binding containing $b(f)$ and is therefore distinct from $m_1^\sharp(f)$. Therefore $m_1^\sharp \sqcap e_0^{+\sharp}(b, O^\sharp(\rho, \text{range}(b)))$ is a correct abstraction of m_2 , is distinct from m_1^\sharp , and is contained in $\llbracket \mathbf{tr}(T) \rrbracket_{m^\sharp}[\rho^\sharp](q_1^\sharp, m_1^\sharp)$.

□

References

- [1] J. Aldrich, C. Chambers, E. G. Sirer, and S. J. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. *Proceedings of SAS '99*, pages 19–38, 1999.
- [2] K. Ali and O. Lhotk. Application-only call graph construction. *Proceedings of ECOOP '12*, pages 688–712, 2012.
- [3] K. Ali and O. Lhotk. Averroes: Whole-program analysis without the whole program. *Proceedings of ECOOP '13*, pages 378–400, 2013.
- [4] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. *Proceedings of OOPSLA '05*, pages 345–364, 2005.
- [5] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *abc* : An extensible AspectJ compiler. *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *LNCS*, pages 293–334. 2006.
- [6] P. Avgustinov, E. Hajiyev, N. Ongkingco, O. de Moor, D. Sereni, J. Tibble, and M. Verbaere. Semantics of static pointcuts in AspectJ. *Proceedings of POPL '07*, pages 11–23, 2007.
- [7] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. *Proceedings of OOPSLA '07*, pages 589–608, 2007.
- [8] K. Bierhoff and J. Aldrich. Lightweight object specification with tpestates. *Proceedings of ESEC/FSE-13*, pages 217–226, 2005.
- [9] K. Bierhoff and J. Aldrich. Modular tpestate checking of aliased objects. *Proceedings of OOPSLA '07*, pages 301–320, 2007.

- [10] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. *Proceedings of OOPSLA '06*, 2006.
- [11] B. Blanchet. Escape analysis for object-oriented languages: application to Java. *Proceedings of OOPSLA '99*, pages 20–34, 1999.
- [12] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. *Proceedings of ECOOP 2007*, volume 4609 of *LNCS*, pages 525–549, 2007.
- [13] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. *Proceedings of FSE '08*, pages 36–47, 2008.
- [14] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. *Proceedings of POPL '99*, pages 133–146, 1999.
- [15] F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. *Proceedings of OOPSLA '07*, pages 569–588, 2007.
- [16] B.-C. Cheng and W.-M. W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. *Proceedings of PLDI '00*, pages 57–69, 2000.
- [17] S. Chereem and R. Rugina. Compile-time deallocation of individual objects. *Proceedings of ISMM '06*, pages 138–149, 2006.
- [18] S. Chereem and R. Rugina. A practical escape and effect analysis for building lightweight method summaries. *Proceedings of CC'07*, pages 172–186, 2007.
- [19] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. *Proceedings of OOPSLA '99*, pages 1–19, 1999.
- [20] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. *Proceedings of POPL '89*, pages 25–35, 1989.
- [21] B. A. Davey and H. A. Priestly. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks. Cambridge University Press, first edition, 1990.

- [22] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. *Proceedings of PLDI '01*, pages 59–69, 2001.
- [23] R. DeLine and M. Fähndrich. Typestates for objects. *Proceedings of ECOOP 2004*, volume 3086 of *LNCS*, pages 465–490, 2004.
- [24] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. *4th International Conference on Computer Languages*, pages 2–13. IEEE Computer Society Press, 1992.
- [25] E. Duesterwald, R. Gupta, and M. L. Soffa. Demand-driven computation of interprocedural data flow. *Proceedings of POPL '95*, pages 37–48, 1995.
- [26] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Trans. Program. Lang. Syst.*, 19(6):992–1030, 1997.
- [27] B. Dufour. Objective quantification of program behaviour using dynamic metrics. Master's thesis, McGill University, June 2004.
- [28] M. Dwyer and R. Purandare. Residual dynamic typestate analysis : Exploiting static analysis results to reformulate and reduce the cost of dynamic analysis. *Proceedings of ASE'07*, pages 124–133, 2007.
- [29] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *Proceedings of ISSTA '06*, pages 133–144, 2006.
- [30] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):1–34, 2008.
- [31] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. *Proceedings of PLDI '02*, pages 1–12, 2002.
- [32] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. *Proceedings of OOPSLA '05*, pages 385–402, 2005.
- [33] S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. *Proceedings of ESOP '07*, pages 253–267, 2007.
- [34] S. Z. Guyer and C. Lin. Client-driven pointer analysis. *Proceedings of SAS '03*, volume 2694 of *LNCS*, pages 214–236, 2003.

- [35] S. Z. Guyer and C. Lin. Error checking with client-driven pointer analysis. *Sci. Comput. Program.*, 58(1-2):83–114, 2005.
- [36] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. *Proceedings of POPL '05*, pages 310–323, 2005.
- [37] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. *Proceedings of PLDI '02*, pages 69–82, 2002.
- [38] M. Hind. Pointer analysis: haven't we solved this problem yet? *Proceedings of PASTE '01*, pages 54–61, 2001.
- [39] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. *Proceedings of FSE '95*, pages 104–115, 1995.
- [40] C. Jaspan and J. Aldrich. Checking temporal relations between multiple objects. Technical Report CMU-ISR-08-119, School of Computer Science, Carnegie Mellon University, 2008. <http://reports-archive.adm.cs.cmu.edu/anon/usr0/ftp/home/anon/usr/anon/isr2008/CMU-ISR-08-119.pdf>.
- [41] C. Jaspan and J. Aldrich. Checking framework interactions with relationships. *Proceedings of ECOOP '09*, pages 27–51, 2009.
- [42] H. B. M. Jonkers. Abstract storage structures. de Bakker and van Vliet, editors, *Algorithmic Languages*, pages 321–343. IFIP, North Holland, 1981.
- [43] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317, 1977.
- [44] G. A. Kildall. A unified approach to global program optimization. *Proceedings of POPL '73*, pages 194–206, 1973.
- [45] O. Lhoták. Comparing call graphs. *Proceedings of PASTE '07*, pages 37–42, 2007.
- [46] K.-K. Ma and J. S. Foster. Inferring aliasing and encapsulation properties for Java. *Proceedings of OOPSLA '07*, pages 423–440, 2007.
- [47] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. *Proceedings of OOPSLA '05*, pages 365–383, 2005.

- [48] N. A. Naeem and O. Lhoták. Extending tpestate analysis to multiple interacting objects. Technical Report CS-2008-04, D. R. Cheriton School of Computer Science, University of Waterloo, 2008. <http://www.cs.uwaterloo.ca/research/tr/2008/CS-2008-04.pdf>.
- [49] N. A. Naeem and O. Lhoták. Tpestate-like analysis of multiple interacting objects. *Proceedings of OOPSLA '08*, pages 347–366, 2008.
- [50] N. A. Naeem and O. Lhoták. Efficient alias set analysis using SSA form. *Proceedings of ISMM '09*, pages 79–88, 2009.
- [51] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Press, 2008.
- [52] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. *Proceedings of SAS '06*, pages 405–424. Springer, 2006.
- [53] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. *Proceedings of PLDI '02*, pages 83–94, 2002.
- [54] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. *Proceedings of POPL '95*, pages 49–61, 1995.
- [55] T. W. Reps. Solving demand versions of interprocedural analysis problems. *Proceedings of CC'94*, pages 389–403, 1994.
- [56] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. *Proceedings of SAS 2005*, pages 284–302, 2005.
- [57] A. Rountev, S. Kagan, and T. Marlowe. Interprocedural dataflow analysis in the presence of large libraries. *Proceedings of CC'06*, pages 2–16, 2006.
- [58] A. Rountev, M. Sharp, and G. Xu. Ide dataflow analysis in the presence of large object-oriented libraries. *Proceedings of CC'08*, pages 53–68, 2008.
- [59] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. G. Hedin, editor, *Proceedings of CC '03*, pages 126–137, 2003.
- [60] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 1996.

- [61] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50, Jan. 1998.
- [62] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.
- [63] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. *Proceedings of ISSSTA '07*, pages 174–184, 2007.
- [64] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.
- [65] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. *Proceedings of OOPSLA '00*, pages 264–280, 2000.
- [66] E. Torlak and S. Chandra. Effective interprocedural resource leak detection. *Proceedings of ICSE'10*, pages 535–544, 2010.
- [67] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: is it feasible? *Proceedings of CC'00*, pages 18–34, 2000.
- [68] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. *Proceedings of OOPSLA '99*, pages 187–206, 1999.
- [69] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. *Proceedings of CAV '08*, pages 385–398, 2008.
- [70] G. Yorsh, E. Yahav, and S. Chandra. Generating precise and concise procedure summaries. *Proceedings of POPL '08*, pages 221–234, 2008.

Index of Symbol Definitions

- Bind**, 18
- Bind[#]**, 34
- $call_{\rho h^\#}[r](0)$, 79
- $call_{\rho h^\#}[r](\rho[o^\#])$, 79
- $call_{\rho h^\#}[r](h[o^\#])$, 79
- $call_{m^\#}[r](0)$, 80
- $call_{m^\#}[r](q, m^\#)$, 80
- compatible*, 38
- $decomp(\rho^\#, h^\#)$, 77
- Dom**, 48
- red-envs*, 38
- h**, 25
- lookup*, 40
- objs*, 40
- Obj**, 13
- Obj[#]**, 27
- diff*, 38
- same*, 38
- relevant*, 38
- setcompat*, 38
- State[#]**, 35
- $update_{d^\#}[r](\overline{V^\#})$, 80
- $update_{d^\#}[r](\langle o^!, o^? \rangle)$, 80
- $update_{o^\#}[r](o^\#)$, 79
- Var**, 13
- $\beta_\rho(\rho, h)$, 28
- $\beta_d[\rho](d)$, 34
- $\beta_m[\rho](m)$, 35
- $\beta_o[\rho](o)$, 27
- $\beta_\sigma[\rho](\sigma)$, 35
- $callFlow(n^\#)$, 57
- $flow(n^\#)$, 57
- $\dot{\sigma}$, 13
- $\dot{e}_0(b, \rho)$, 13
- $\dot{e}_T(T, \rho)$, 14
- $passArgs(n^\#)$, 57
- ϕ , 65
- $returnVal(n^\#)$, 57
- ρ , 13
- $\rho^\#$, 27
- $rv(o_c^\#, o_r^\#)$, 80
- σ , 23
- $\sigma^\#$, 35
- $summ(D)$, 80
- $summ_{\rho h^\#}(0, r_h^\rho[o_r^\#])$, 80
- $summ_{\rho h^\#}(c_h^\rho[o_c^\#], r_h^\rho[o_r^\#])$, 80
- $e(T, \rho)$, 20
- $e[a, b, \rho](q, m)$, 20
- $e^+[a, b, \rho](q, m)$, 20
- $e^-[b, \rho](q, m)$, 20
- $e^\#[a, b, O^\#](q, m^\#)$, 40
- $e^{+\#}[a, b, O^\#](q, m^\#)$, 40
- $e^{-\#}[b, O^\#](q, m^\#)$, 40
- $e_0^+(b, \rho)$, 20
- $e_0^-(b, \rho, f)$, 20
- $e_0^{+\#}(b, O^\#)$, 40
- $e_0^{-\#}(b, O^\#, f)$, 40
- $passArgs(\langle l, d \rangle)$, 79, 80
- $returnVal(\langle e_p, d_1 \rangle, \langle n, d_2 \rangle)$, 80