

Extending Typestate Analysis to Multiple Interacting Objects*

Nomair A. Naeem Ondřej Lhoták
D. R. Cheriton School of Computer Science
University of Waterloo, Canada
{nanaeem,olhotak}@uwaterloo.ca

Abstract

This paper extends static typestate analysis to temporal specifications of groups of interacting objects, which are expressed using tracematches. Unlike typestate, a tracematch state may change due to operations on any of a set of objects bound by the tracematch. The paper proposes a lattice-based operational semantics equivalent to the original tracematch semantics but better suited to static analysis. The paper defines a static analysis that computes precise local points-to sets and tracks the flow of individual objects, thereby enabling strong state updates of the tracematch state. The analysis has been proved sound with respect to the semantics. A context-sensitive version of the analysis has been implemented as instances of the IFDS and IDE algorithms. The analysis was evaluated on tracematches used in earlier work and found to be very precise. Remaining imprecisions could be eliminated with more precise modeling of references from the heap and of exceptional control flow.

1 Introduction

An object is not isolated; it interacts with other objects. For an object, a temporal specification can be expressed using typestate [30]. 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 typestate checking tools, both dynamic and static, have been developed [1, 5, 6, 9, 12, 13, 16–20, 22, 25]. Temporal specifications can 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. Some newer specification mechanisms can express temporal properties of multiple objects [1, 9, 18, 25]. These formalisms are mainly intended for dynamic checking. In this paper, we extend techniques from static typestate verification to formulate and implement a static analysis of such multi-object temporal specifications.

The 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 [1] 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

*Technical Report CS-2008-04, extended version of a paper submitted to OOPSLA 2008. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

```

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 }

```

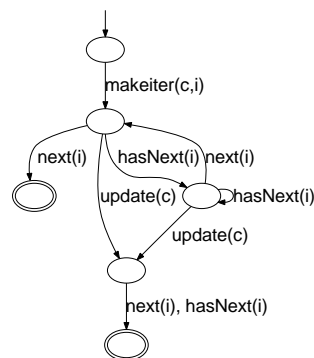


Figure 1: Tracematch example: iterator safety

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 [1]. The operations, and how they bind objects, are specified using AspectJ pointcuts, which are in widespread use and have a formal specification [3].

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. [1]

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. [16]) 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. The first analysis uses a storeless heap abstraction [14, 23] similar to earlier work [10, 16, 21, 29]. The focus of the paper is on the second analysis, which is novel. Indeed, we present a specific object analysis only for the sake of concreteness; the object analysis could be replaced with more precise or cheaper variants if necessary for a particular application.

The example in Figure 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

```

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

```

Figure 2: Tracematch source code

of the property causes the automaton to enter one of the final states. The tracematch associated with this automaton (shown in Figure 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 .

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. Precise must-alias information is necessary to ensure that each call to `hasNext` occurs on the same iterator as the subsequent call to `next`. In fact, we need to know more than that a pair of variables are aliased. For example, it is *not* true that `it2` in lines 6 and 7 always point to the same object. 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, the blanket statement that `it2` at line 6 is must-aliased to `it2` at line 7 is false. Instead, the analysis must track the flow of objects along control flow paths. To summarize, the analysis requires:

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

The analysis presented in this paper satisfies all three requirements.

The main contributions of this paper are:

1. We define a lattice-based operational semantics of tracematches which is better suited to static analysis than the original semantics of Allan et al. [1]. We have proven that the two semantics are bisimilar. (Section 2)
2. We define a precise static abstraction of the lattice-based operational semantics. The first part is an abstraction of the runtime objects occurring in the program. The second part is an abstraction of tracematch states. We have proven that the overall abstraction is sound with respect to the operational semantics. (Section 3)

3. We express the static analysis as instances of the IFDS [27] and IDE [28] frameworks which efficiently support context-sensitive interprocedural analysis. (Section 4)
4. We report experimental results from our implementation of the static analysis. We implemented the analysis in Scala, using the tracematch implementation in the abc compiler [1, 2] to provide the intermediate representation to be analyzed. (Section 5)

2 Tracematch Semantics

Allan et al. [1] 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 2 shows the source code that a programmer would write to define the example tracematch discussed in Section 1. This tracematch has two parameters, a `Collection c` and an `Iterator i`. Lines 2-7 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 9 defines the regular language of the tracematch and lines 10-12 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 [1].

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 of Tracematches

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.

Much of the expressive power comes from tracematch 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 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 each event in the trace if the

| Trace | c=x i=a | c=x i=b | c=y i=a | c=y i=b |
|---------------|------------|------------|------------|------------|
| makeIter(x,a) | makeIter | | | |
| hasNext(a) | hasNext | | hasNext | |
| makeIter(y,b) | | | | makeIter |
| next(a) | next | | next | |
| hasNext(b) | | hasNext | | hasNext |
| update(x) | update | update | | |
| next(b) | | next | | next |
| next(a) | next | | next | |
| next(b) | | next | | next |
| | match | no | no | match |

Figure 3: 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.

parameters bound by the event are bound to the objects associated with that automaton version. The tracematch body is executed whenever an automaton version reaches an accepting state; at that point, the automaton version is discarded.

We illustrate with an example. Figure 3 shows a possible trace of the events declared in the tracematch from Figure 2. 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 `a` and `b`. 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 3. 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=a`, and the version with `c=y` and `i=b`. 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

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.

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. The constant \mathbf{h} represents any heap location, such as a field of an object or an array element.

The two instructions directly relevant to tracematches are \mathbf{tr} (*transition statement*) and \mathbf{body} (*body statement*). Each transition statement contains a pair a, b where $a \in A$ 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. [1] allow each transition statement to contain *multiple* transitions, each 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.

A body statement is generated immediately after every transition statement $\mathbf{tr} \langle a, b \rangle$ 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 in an accepting state, execute the tracematch body for it, and discard it.

The remaining IR instructions are self-explanatory: they copy object references between variables and the heap, and create new objects.

The semantics of transition statements is defined in terms of a set **Var** of variables in the base language and a set **Obj** \cup $\{\perp\}$ of values that those variables can take. The symbol \perp denotes the special null value and **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 automaton. 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 $\mathbf{tr} \langle a, b \rangle$ is executed in environment ρ , a boolean formula is generated that evaluates to true for tracematch bindings consistent with the objects bound in the transition element:

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

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

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

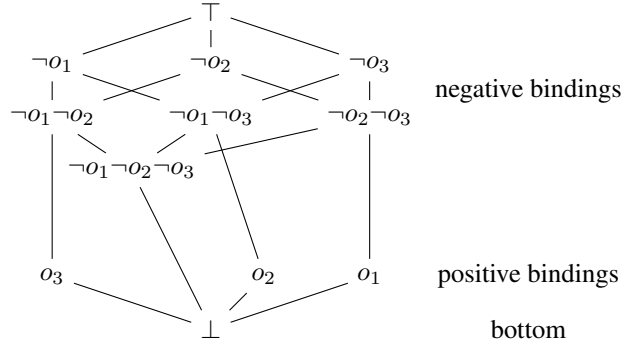
Recall that a tracematch state $\hat{\sigma}$ conceptually represents the state of different automata with different bindings. At a transition, each automaton performs a transition if its bindings are consistent with the objects bound in the transition (i.e. $\hat{e}_0(b, \rho)$ is satisfied), or remains in its current state if its bindings are inconsistent (i.e. $\neg \hat{e}_0(b, \rho)$ is satisfied). Thus, the transition function is defined [1, 4] as:

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

where $T_a \triangleq \{b : \langle a, b \rangle \in T\}$. In [1], the notation $e(a)$ is used with the same meaning as $\hat{e}_T(T_a, \rho)$.

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 transition function for transition statements in the operational semantics is:

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

Figure 4: Binding Lattice **Bind**

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:

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

Allan et al. [1] 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 it only has to manipulate one automaton with boolean formulas on its states, rather than an unbounded collection of automata. However, boolean formulas are not well suited to static abstraction.

2.3 A Lattice-Based Operational Semantics

The operational semantics just presented is suitable for a dynamic implementation of tracematches, but boolean formulas are a difficult concrete domain to abstract. We therefore define a different but equivalent operational semantics based on sets and lattices that is well suited for static analysis.

The core construction of the semantics is a *binding lattice*. Figure 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. 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 \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 .

Formally, the binding lattice $(\mathbf{Bind}, \sqsubseteq)$ is defined as follows. Its elements are $\mathbf{Bind} \triangleq \mathbf{Obj} \uplus \mathcal{P}(\mathbf{Obj}) \uplus \{\perp\}$. As a reminder that a set of values indicates negative bindings, we will always write such a set with a bar above it: \bar{O} . The bar is only a reminder; it has no semantic meaning. The partial order \sqsubseteq is defined as the reflexive transitive closure of the following rules: $\perp \sqsubseteq d$ for any d ; $\bar{O}_1 \sqsubseteq \bar{O}_2$ if $O_1 \supseteq O_2$; and $o_1 \sqsubseteq \bar{O}_2$ 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:

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

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 $o \not\sqsubseteq \perp$, 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 [11, Theorem 2.16]. \square

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 $c \mapsto x$, $i \mapsto \neg b$. Of the automaton versions shown in Figure 3, only the one corresponding to $c=x$ and $i=a$ 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 f and the automaton in Figure 5, and suppose that the transition $\langle a, f \mapsto o_1 \rangle$ 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 $\lambda f. o_1$ and $\lambda f. \neg o_1$ to obtain the desired pairs $\langle qa, \lambda f. o_1 \rangle$ and $\langle q, \lambda f. \neg o_1 \rangle$. Suppose the transition $\langle b, f \mapsto o_2 \rangle$ occurs next. We again perform the meets of the existing states with both $\lambda f. o_2$ and $\lambda f. \neg o_2$ to obtain $\langle qab, \lambda f. \perp \rangle$, $\langle qa, \lambda f. o_1 \rangle$, $\langle qb, \lambda f. o_2 \rangle$, $\langle q, \lambda f. \neg o_1 \neg o_2 \rangle$. 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.

In the general case of a tracematch with multiple parameters, there is an additional difference between negative and positive bindings. The automaton versions that change state must match in *all* the parameters bound by the transition statement, while the versions that remain the same need only differ in *some* parameter. Thus, the new map for the

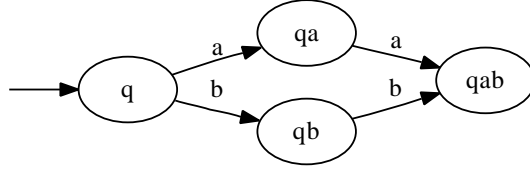


Figure 5: Example automaton

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

Figure 6: Transition function for transition statements

changing versions 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. The nonchanging versions are represented by multiple maps, one for each parameter bound by the tracematch: each of these maps is obtained by replacing only the one $m(f)$ with $m(f) \sqcap \neg o$, where o is the object bound to f by the transition statement.

Figure 6 shows the complete transition function e that is applied to each pair $\langle q, m \rangle$ in the tracematch state at transition statement $\mathbf{tr}\langle a, b \rangle$.

When a transition statement contains multiple pairs $\langle a, b \rangle$, 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:

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

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:

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

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 $\hat{\sigma}$ 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 $\hat{\sigma} R \sigma \triangleq s_\sigma(\sigma)(q) \iff \hat{\sigma}(q)$. That is,*

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

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 $d_1 = \bar{O}_1$ and $d_2 = o_2$ where $o_2 \in \bar{O}_1$: Then $s_d(\langle f, d_1 \sqcap d_2 \rangle) = s_d(\langle f, \perp \rangle) = \text{false}$. On the other side, $s_d(\langle f, d_1 \rangle) \wedge s_d(\langle f, d_2 \rangle) = \bigwedge_{o \in \bar{O}_1} \neg(f = o) \wedge (f = o_2) = \bigwedge_{o \in \bar{O}_1} \neg(f = o) \wedge \neg(f = o_2) \wedge (f = o_2) = \text{false}$ since $o_2 \in \bar{O}_1$.

Case $d_1 = \bar{O}_1$ and $d_2 = o_2$ where $o_2 \notin \bar{O}_1$: 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, $s_d(\langle f, d_1 \rangle) \wedge s_d(\langle f, d_2 \rangle) = \bigwedge_{o \in \bar{O}_1} \neg(f = o) \wedge (f = o_2) = (f = o_2)$ since $(f = o_2) \implies \neg(f = o)$ for all $o \neq o_2$, and $o_2 \notin \bar{O}_1$.

Case $d_1 = \bar{O}_1$ and $d_2 = \bar{O}_2$: 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, $s_d(\langle f, d_1 \rangle) \wedge s_d(\langle f, d_2 \rangle) = \bigwedge_{o \in \bar{O}_1} \neg(f = o) \wedge \bigwedge_{o \in \bar{O}_2} \neg(f = o) = \bigwedge_{o \in O_1 \cup O_2} \neg(f = o)$.

□

Lemma 2.

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

Proof.

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

□

Lemma 3.

$$s_m(e_+(b, \rho)) = \mathring{e}_0(b, \rho)$$

Proof.

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

□

Lemma 4.

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

Proof.

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

□

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

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

Proof. We use induction on n . In the base case, $n = 0$, so the right-hand side is $\bigvee_{\langle q, m \rangle \in \sigma} s_m(m)$ and the left-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

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

then

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

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 $\neg \mathring{e}_0(b_n, \rho) = \neg \text{true} = \text{false}$, so the right-hand side is also false.

Case $\exists \langle q, m \rangle \in \sigma'$: 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 $\exists \langle q, m \rangle \in \sigma'$:

$$\begin{aligned}
& \bigvee_{\langle q, m \rangle \in e_-[b_n, \rho](\sigma')} s_m(m) \\
= & \bigvee_{\langle q, m \rangle \in \sigma'} \bigvee_{f \in \text{dom}(b_n)} s_m(m \sqcap e_-(b_n, \rho, f)) \\
= & \bigvee_{\langle q, m \rangle \in \sigma'} \bigvee_{f \in \text{dom}(b_n)} s_m(m[f \mapsto m(f) \sqcap \overline{\{\rho(b_n(f))\}}]) \\
= & \bigvee_{\langle q, m \rangle \in \sigma'} \bigvee_{f \in \text{dom}(b_n)} \bigwedge_{f' \in F} s_d(\langle f', m[f \mapsto m(f) \sqcap \overline{\{\rho(b_n(f))\}}](f') \rangle) \\
= & \bigvee_{\langle q, m \rangle \in \sigma'} \bigvee_{f \in \text{dom}(b_n)} \left(s_d(\langle f, m(f) \sqcap \overline{\{\rho(b_n(f))\}} \rangle) \wedge \bigwedge_{f' \in F \setminus \{f\}} s_d(\langle f', m(f') \rangle) \right) \\
= & \bigvee_{\langle q, m \rangle \in \sigma'} \bigvee_{f \in \text{dom}(b_n)} \left(s_d(\langle f, m(f) \rangle) \wedge s_d(\langle f, \overline{\{\rho(b_n(f))\}} \rangle) \wedge \bigwedge_{f' \in F \setminus \{f\}} s_d(\langle f', m(f') \rangle) \right) \\
= & \bigvee_{\langle q, m \rangle \in \sigma'} \bigvee_{f \in \text{dom}(b_n)} \left(s_d(\langle f, \overline{\{\rho(b_n(f))\}} \rangle) \wedge \bigwedge_{f' \in F} s_d(\langle f', m(f') \rangle) \right) \\
= & \bigvee_{\langle q, m \rangle \in \sigma'} \bigvee_{f \in \text{dom}(b_n)} \neg(f = \rho(b_n(f))) \wedge s_m(m) \\
= & \left(\bigvee_{\langle q, m \rangle \in \sigma'} \bigvee_{f \in \text{dom}(b_n)} \neg(f = \rho(b_n(f))) \right) \wedge \left(\bigvee_{\langle q, m \rangle \in \sigma'} \bigvee_{f \in \text{dom}(b_n)} s_m(m) \right) \\
= & \left(\neg \bigwedge_{f \in \text{dom}(b_n)} (f = \rho(b_n(f))) \right) \wedge \left(\bigvee_{\langle q, m \rangle \in \sigma'} s_m(m) \right) \\
= & \neg e_0(b_n, \rho) \wedge s_\sigma(\sigma)(q) \wedge \bigwedge_{1 \leq i \leq n-1} \neg e_0(b_i, \rho) \\
= & s_\sigma(\sigma)(q) \wedge \bigwedge_{1 \leq i \leq n} \neg e_0(b_i, \rho)
\end{aligned}$$

□

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

Proof. Let $\tilde{\sigma}(q)$ 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 $m(f) \triangleq \{o : \neg(f = o) \text{ is a literal in the conjunction}\}$. 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 $\hat{\sigma}(q)$. Then $s_\sigma(\sigma)(q) \iff \hat{\sigma}(q)$ for all q as required. \square

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

Proof of Theorem 1. For every σ we can define $\hat{\sigma} \triangleq s_\sigma(\sigma)$, and this definition ensures that $\hat{\sigma}(q) \iff s_\sigma(\sigma)(q)$. Conversely, for every $\hat{\sigma}$, 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 $\langle \mathbf{tr}(T), \hat{\sigma} \rangle \rightarrow \langle \hat{\sigma}' \rangle$, then $\hat{\sigma}' = s_\sigma(\sigma')$.

$$\begin{aligned}
\hat{\sigma}' &= \lambda q. \left(\bigvee_{a,j:\delta(j,a,q)} (\hat{\sigma}[q_0 \mapsto \mathbf{true}](j) \wedge \hat{e}(a, \{\langle a_1, b_1 \rangle \cdots \langle a_n, b_n \rangle\}, \rho)) \right) \vee \left(\hat{\sigma}[q_0 \mapsto \mathbf{true}](q) \wedge \bigwedge_{a \in A} \neg \hat{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} e_0^{\circ}(b_i, \rho) \right) \right) \vee \left(s_\sigma(\sigma \cup \{\langle q_0, \lambda f. \top \rangle\})(q) \wedge \bigwedge_{1 \leq i \leq n} \neg e_0^{\circ}(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 \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 \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 \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 \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 \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. \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] \cdots (e_-[b_n, \rho](\sigma \cup \{\langle q_0, \lambda f. \top \rangle\})) \cdots} 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] \cdots (e_-[b_n, \rho](\sigma \cup \{\langle q_0, \lambda f. \top \rangle\})) \cdots \right) \\
&= s_\sigma(\sigma')
\end{aligned}$$

\square

To extend the operational semantics to statements other than $\mathbf{tr} \langle a, b \rangle$ and \mathbf{body} , we add to it a set h containing all objects referenced from the heap. The instructions $\mathbf{tr} \langle a, b \rangle$ and \mathbf{body} defined earlier do not change the environment

ρ or the heap h . The operational semantics of the remaining instructions is unsurprising, except that the effect of the load instruction $v \leftarrow e$ is non-deterministic, because we do not know which specific object from h is loaded:

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

3 Static Abstraction

The abstraction is presented in two parts. The first abstraction computes object aliasing relationships. This information is needed to determine which objects are pointed to by the variables in each transition statement. The second abstraction models the tracematch state. Using this abstraction, the analysis can prove that at certain body statements, the tracematch cannot be in an accepting state.

3.1 Object Abstraction

The object 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 [29]. 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 $\rho^\#$ of abstract objects. This set is an overapproximation of all possible objects. That is, if it is possible for some concrete object to be pointed to by the set of variables $o^\#$, then the set $o^\#$ must be an element of $\rho^\#$. Conversely, the presence of $o^\#$ in $\rho^\#$ indicates that there may exist zero or more concrete objects which are pointed to by the variables in $o^\#$ 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 $\mathbf{Obj}^\# \triangleq \mathcal{P}(\mathbf{Var})$ as the set of all sets of variables. The function $\beta_o[\rho] : \mathbf{Obj} \rightarrow \mathbf{Obj}^\#$ 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:

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

The abstraction subsumes both may-alias and must-alias relationships. If variables x and y point to distinct objects, $\rho^\#$ will not contain any set containing both x and y . If variables x and y point to the same object, every set in $\rho^\#$ will contain either both x and y , or neither of them.

At run time, a variable cannot point to more than one object at a time. Thus, every abstract object except the empty set \emptyset represents at most one concrete object at any given point of execution. This enables precise flow-sensitive analysis including strong updates.

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

Figure 7: Transfer function for the object abstraction

Specifically, if s is any statement in the IR except a heap load, and if $o^\#$ is the set of variables pointing to a given concrete object o , then it is possible to compute the exact set of variables which will point to o after the execution of s . The transfer function $\llbracket s \rrbracket_{o^\#}$ that performs this computation is shown in Figure 7. 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 $v \leftarrow e$, 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 $\rho(v_2) = o$:**

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

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

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

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

To precisely handle the uncertainty in heap loads we use the materialization or focus operation from [10,16,21,29]. The abstract object $o^\#$ is split into two, one representing the single concrete object that was loaded, and the other representing all other objects previously represented by $o^\#$. 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.

Formal definitions of the transfer functions for $\rho^\#$ and $h^\#$ in terms of $\llbracket s \rrbracket_{o^\#}$ and the focus operation are given in Figure 7. We combine $\rho^\#$ and $h^\#$ 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 $\langle \rho_1^\#, h_1^\# \rangle \sqsubseteq \langle \rho_2^\#, h_2^\# \rangle$ if $\rho_1^\# \subseteq \rho_2^\# \wedge h_1^\# \subseteq h_2^\#$, which induces a join operator $\langle \rho_1^\#, h_1^\# \rangle \sqcup \langle \rho_2^\#, h_2^\# \rangle \triangleq \langle \rho_1^\# \cup \rho_2^\#, h_1^\# \cup h_2^\# \rangle$. The property that $\rho^\# \supseteq h^\#$ is always maintained by the transfer functions. On the combined object abstraction, the correctness relation $R_{\rho h}$ is defined as $\langle \rho, h \rangle R_{\rho h} \langle \rho^\#, h^\# \rangle \triangleq \beta_{\rho h}(\rho, h) \sqsubseteq \langle \rho^\#, h^\# \rangle$. We have proven that the transfer function preserves the correctness relation:

Theorem 2. *If $\langle s, \rho, h, \sigma \rangle \rightarrow \langle \rho', h', \sigma' \rangle$ and $\langle \rho, h \rangle R_{\rho h} \langle \rho^\#, h^\# \rangle$, 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 $s = v \leftarrow e$: $\text{range}(\rho') = \text{range}(\rho[v \mapsto o])$ 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 $s = e \leftarrow v$: $\text{range}(\rho') \cup h' = \text{range}(\rho) \cup h \cup \{\rho(v)\} = \text{range}(\rho) \cup h$.

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

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

□

Proof 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 $s \in \{v_1 \leftarrow v_2, v \leftarrow \text{null}, e \leftarrow v\}$:

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

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

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

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

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

Case $s = e \leftarrow v$:

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

Case $s = v \leftarrow e$: 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, $\langle \rho^\#, h^\# \rangle \sqsupseteq \beta_{\rho h}(\rho, h)$. Since $\llbracket s \rrbracket_{\rho h^\#}$ is monotone, $\llbracket s \rrbracket_{\rho h^\#}(\rho^\#, h^\#) \sqsupseteq \llbracket s \rrbracket_{\rho h^\#}(\beta_{\rho h}(\rho, h))$, which we just proved is greater than $\beta_{\rho h}(\rho', h')$. Thus, the theorem holds in the general case. \square

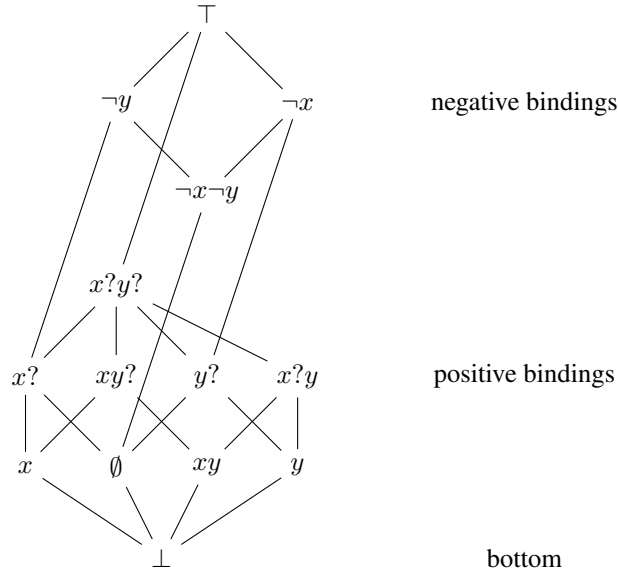
3.2 Tracematch Abstraction

In existing work that associates typestate with a single object (e.g. [16]), the static analysis simply adds the typestate to the static abstraction of each object. In our setting, a state is associated with multiple objects, so 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.

Thanks to the design of our tracematch semantics, a basic tracematch state abstraction is 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 can 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 must-alias information 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

Figure 8: Abstract Binding Lattice \mathbf{Bind}^\sharp

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 8. 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. In addition, although a concrete negative binding is a *set* of objects, all the must sets representing these objects can be replaced with their intersection without affecting precision of the analysis. Thus, the \mathbf{Bind}^\sharp lattice illustrated in Figure 8 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 $V^\sharp \subseteq \bigcup_{o \in \overline{O}} \beta_o(o)$.

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

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

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 ; $V_1^\sharp \sqsubseteq V_2^\sharp$ 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 lattice and that the abstraction function β_d preserves the partial order from \mathbf{Bind} in \mathbf{Bind}^\sharp .

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

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

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

Proof. \mathbf{Bind}^\sharp 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^\sharp, d_2^\sharp 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^\sharp and d_2^\sharp 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 $\langle o^!, o^? \setminus V^\sharp \rangle$ 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}^\sharp 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 $d_1^\sharp \triangleq \beta_d[\rho](d_1)$ and $d_2^\sharp \triangleq \beta_d[\rho](d_2)$.

When $d_1 = \perp$, d_1^\sharp 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 $d_1^\sharp = \bigcup_{o \in O_1} \beta_o(o) \supseteq \bigcup_{o \in O_2} \beta_o(o) = d_2^\sharp$, so $d_1^\sharp \sqsubseteq d_2^\sharp$. \square

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

Figure 9: Transfer functions for the tracematch state abstraction for $s \neq \mathbf{tr}(T)$

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 defined in Figure 9. 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 sets 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 and to \mathbf{State}^\sharp by $\llbracket s \rrbracket_{m^\sharp}$ and $\llbracket s \rrbracket_{\sigma^\sharp}$. Since $\llbracket s \rrbracket_{d^\sharp}$ is so similar to $\llbracket s \rrbracket_{o^\sharp}$, we can prove an analogue of Proposition 2 for it:

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 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 $d^\sharp = \langle o^!, o^? \rangle$ and $o^! \subseteq \beta_o[\rho](o) \subseteq o^?$.*

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 $o^! \subseteq \beta_o[\rho](o) \subseteq o^?$. \square

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

Proof. Since $\overline{O} R_d[\rho] d^\sharp$, $\beta_d[\rho](\overline{O}) = \overline{\bigcup_{o \in \overline{O}} \beta_o[\rho](o)} \subseteq d^\sharp$. 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 $\bigcup_{o \in \overline{O}} \beta_o[\rho](o)$. \square

Proof of Proposition 5. Case $d = \perp$: Then $\beta_d[\rho](d) = \perp \subseteq \llbracket s \rrbracket_{d^\sharp}(d^\sharp)$, 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, $o^1 \subseteq \beta_o[\rho](o^\sharp) o^2$.

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 $v_2 \in o^2$.

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

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

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

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 $v_2 \notin \beta_o[\rho](o)$.

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

Subcase $s = v \leftarrow e$:

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,

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

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

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

Subcase $s \in \{e \leftarrow v, \text{body}\}$:

$$\begin{aligned}o R_d[\rho] \langle o^1, o^2 \rangle &\implies o^1 \subseteq \beta_o[\rho](o) \subseteq o^2 \\ &\implies o^1 \subseteq \llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o)) \subseteq o^2 && \text{definition of } \llbracket s \rrbracket_{o^\#} \\ &\implies o^1 \subseteq \beta_o[\rho'](o) \subseteq o^2 && \text{from Proposition 2} \\ &\implies o R_d[\rho'] \langle o^1, o^2 \rangle && \text{definition of } R_d[\rho] \\ &\implies o R_d[\rho'] \llbracket s \rrbracket_{d^\#}(\langle o^1, o^2 \rangle) && \text{definition of } \llbracket s \rrbracket_{d^\#}\end{aligned}$$

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

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

Therefore, there is some $o' \in \overline{O}$ for which $v_2 \in \beta_o[\rho](o')$. So $\beta_o[\rho'](o') = \llbracket s \rrbracket_{o^\#}(\beta_o[\rho](o')) = \beta_o[\rho](o') \cup \{v_1\}$.

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

Subcase $s = v_1 \leftarrow v_2 \wedge v_2 \notin \overline{V^\#}$: From the definition of $\llbracket s \rrbracket_{o^\#}$, it follows that $\llbracket v_1 \leftarrow v_2 \rrbracket_{o^\#}(o^\#) \supseteq o^\# \setminus \{v_1\}$.

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

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

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

Subcase $s = v \leftarrow e$:

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 $s \in \{e \leftarrow v, \text{body}\}$:

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

□

The transfer function for transition statements is more complicated. 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* [29], 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 $\rho^\#$ satisfying the following constraints:

- 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.

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

Figure 10: Generalized compatibility predicate.

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 .

Since Sagiv et al.'s notion of compatibility [29] 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 10. 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.

The transfer function for transition statements is defined in Figure 11. At a high level, it mirrors the semantics of $\text{tr}(T)$ presented in Section 2. 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 $\text{lookup}(O^\#, v)$ substituted for concrete lookup in ρ . The overall transfer function $\llbracket \text{tr}(T) \rrbracket_{m^\#}$ joins the results of $e^\#$ for all compatible and relevant 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 9. At control flow merge points, the join operator used on $\text{State}^\#$ is set union.

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

Lemma 10. *For all statements except $\text{tr}(T)$, 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 $\text{tr}(T)$ leave the tracematch state abstraction σ unchanged. This means that

$$\begin{aligned}
\text{objs}(m^\#) &\triangleq \{\langle o^!, o^? \rangle \in \text{range}(m^\#)\} \\
\text{lookup}(O^\#, v) &\triangleq o^\# \in O^\# : v \in o^\# \\
e_0^{+\#}(b, O^\#) &\triangleq \lambda f. \begin{cases} \langle \text{lookup}(O^\#, b(f)), \text{lookup}(O^\#, b(f)) \rangle & \text{if } f \in \text{dom}(b) \\ \perp & \text{otherwise} \end{cases} \\
e^{+\#}[a, b, O^\#](q, m^\#) &\triangleq \left\{ \left\langle q', m^\# \sqcap e_0^{+\#}(b, O^\#) \right\rangle : \delta(q, a, q') \right\} \\
e_0^{-\#}(b, O^\#, f) &\triangleq \lambda f'. \begin{cases} \text{lookup}(O^\#, b(f)) & \text{if } f = f' \\ \perp & \text{otherwise} \end{cases} \\
e^{-\#}[b, O^\#](q, m^\#) &\triangleq \left\{ \left\langle q, m^\# \sqcap e_0^{-\#}(b, O^\#, f) \right\rangle : f \in \text{dom}(b) \right\} \\
e^\#[a, b, O^\#](q, m^\#) &\triangleq e^{+\#}[a, b, O^\#](q, m^\#) \cup e^{-\#}[b, O^\#](q, m^\#) \\
e^\#\{\{\langle a_1, b_1 \rangle \cdots \langle a_n, b_n \rangle\}, O^\#\}(q, m^\#) &\triangleq \left(\left(\bigcup_{1 \leq i \leq n} e_+^\#[a_i, b_i, O^\#](q, m^\#) \right) \cup e_-^\#[b_1, O^\#](\cdots e_-^\#[b_n, O^\#](q, m^\#) \cdots) \right) \\
\llbracket \mathbf{tr}(T) \rrbracket_{m^\#}[\rho^\#](q, m^\#) &\triangleq \bigcup_{O^\# \in \text{red-envs}(\rho^\#, \text{objs}(m^\#), \bigcup_{(a,b) \in T} \text{range}(b))} e^\#[T, O^\#](q, m^\#)
\end{aligned}$$

Figure 11: Transfer function for the tracematch state abstraction for $s = \mathbf{tr}(T)$

$\sigma = \sigma'$.

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

□

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, $\beta_d[\rho](d_1 \sqcap d_2) \sqsubseteq \beta_d[\rho](d_2) \sqsubseteq d_2^\#$. Therefore, $\beta_d[\rho](d_1 \sqcap d_2) \sqsubseteq d_1^\# \sqcap d_2^\#$. Thus, $d_1 \sqcap d_2 R_d[\rho] d_1^\# \sqcap d_2^\#$. □

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 $o_i R_d[\rho] o_i^{!?}$ 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 $O^\#(\rho, V) \triangleq \{\beta_o[\rho](\rho(v)) : v \in V\}$.

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

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

Proof. 1.

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

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 $n(O^\#(\rho, V), v)$. Furthermore, $\beta_o[\rho](\rho(v))$ is the only such element of $O^\#(\rho, V)$, since for any other object $o' \neq \rho(v)$, $v \notin \beta_o[\rho](o')$. □

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\langle q, m^\# \rangle \in \bigcup_{O^\# \in \text{red-envs}(\rho^\#, \text{objs}(m^\#), \text{range}(b))} e_+^\#[a, b, O^\#](\sigma^\# \cup \{\langle q_0, \lambda f. \top \rangle\}) \cup e_-^\#[b, O^\#](\sigma^\# \cup \{\langle q_0, \lambda f. \top \rangle\}) = \llbracket \text{tr}(\{\langle a, b \rangle\}) \rrbracket_{\sigma^\#}[\rho^\#](\sigma^\#)$$

such that $m R_m[\rho] m^\#$. The same correspondence holds for the case when T contains multiple transition elements.

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

4 Context-Sensitive Interprocedural Analysis

We implemented the analysis as an instance of the IFDS¹ algorithm of Reps et al. [27] with some small modifications which we explain in this section. The IFDS algorithm implements a fully context-sensitive interprocedural dataflow analysis provided that:

¹interprocedural finite distributive subset

- the analysis domain is the powerset of a finite set **Dom**,
- the merge operator is union, and
- the transfer function is distributive.

IFDS is an efficient dynamic programming algorithm that 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 reason for its efficiency is that it evaluates transfer functions on each individual element of **Dom** at a time, rather than on a subset of **Dom** at a time (recall that each element of the analysis domain is a subset of **Dom**). As a result, any distributive function $f : \mathcal{P}(\mathbf{Dom}) \rightarrow \mathcal{P}(\mathbf{Dom})$ can be efficiently represented as a graph with at most $(|\mathbf{Dom}| + 1)^2$ edges [27, Section 3]. The IFDS algorithm starts with a graph representation of the transfer function for each instruction in the program, and works by composing them into transfer functions for ever longer sequences of instructions. Specifically, it uses a typical worklist algorithm to complete two tables of transfer functions: the PathEdge table gives the transfer function from the start node of each procedure to every other node in the same procedure, and the SummaryEdge table gives the transfer function that summarizes the effect of each call site in the program.

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 for the value abstraction $\langle \rho^\#, h^\# \rangle$, we define the set **Dom** as two copies of $\mathbf{Obj}^\# = \mathcal{P}(\mathbf{Var})$, one copy for $\rho^\#$ and the other for $h^\#$. To distinguish elements of the two sets, we use the notation $\rho[o^\#]$ to mean $o^\#$ from the $\rho^\#$ copy of $\mathbf{Obj}^\#$, and $h[o^\#]$ to mean $o^\#$ from the $h^\#$ copy. 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 **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 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.

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 $s = v \leftarrow e$:** In this case,

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

Therefore,

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

Also,

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

Case $s = e \leftarrow v$: In this case,

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

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,

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

Also,

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

□

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 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 and out of procedure calls. These flow functions are also decomposed into functions acting on individual elements of $\mathbf{Dom} \cup \{0\}$. The call flow 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:

$$\begin{aligned} \text{update}_{o^\#}[r](o^\#) &\triangleq \{r(v) : v \in o^\# \cap \text{dom}(r)\} \\ \text{call}_{\rho h^\#}[r](\rho[o^\#]) &\triangleq \{\rho[\text{update}[r](o^\#)]\} \\ \text{call}_{\rho h^\#}[r](h[o^\#]) &\triangleq \{h[\text{update}[r](o^\#)]\} \\ \text{call}_{\rho h^\#}[r](0) &\triangleq \emptyset \end{aligned}$$

To define the flow out of procedure calls for the object abstraction, a small modification to the IFDS algorithm is necessary. In the original algorithm, 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 $\text{return} \circ \llbracket p \rrbracket \circ \text{call}$, 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. It is straightforward to modify the algorithm to instead use a function that, given a call site and the computed flow function $\llbracket p \rrbracket \circ \text{call}$, directly constructs the SummaryEdge flow function. A similar modification is also used in the tpestate analysis of Fink et al. [16]. Indeed, the general modification is likely to be useful in other instantiations of the IFDS algorithm.

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.

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

The call flow function for tracematch state applies the $\text{update}[r]$ 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.

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

The summary flow function for the tracematch abstraction is defined as follows:

$$\begin{aligned}
\text{summ}_{m^\#}(0, \langle q_r, m_r^\# \rangle) &\triangleq \{ \langle q_r, \lambda f. \text{rv}_{d^\#}(\top, m_r^\#(f)) \rangle \} \\
\text{summ}_{m^\#}(\langle q_c, m_c^\# \rangle, \langle q_r, m_r^\# \rangle) &\triangleq \{ \langle q_r, \lambda f. \text{rv}_{d^\#}(m_c^\#(f), m_r^\#(f)) \rangle \} \\
\text{rv}_{d^\#}(\langle o_c^!, o_c^? \rangle, \langle o_r^!, o_r^? \rangle) &\triangleq \begin{cases} \langle o_c^!, o_c^? \rangle & \text{if } p \text{ does not return a value} \\ \langle o_c^! \cup \{v_t\}, o_c^? \cup \{v_t\} \rangle & \text{if } v_s \in o_r^! \\ \langle o_c^! \setminus \{v_t\}, o_c^? \cup \{v_t\} \rangle & \text{if } v_s \notin o_r^! \wedge v_s \in o_r^? \\ \langle o_c^! \setminus \{v_t\}, o_c^? \setminus \{v_t\} \rangle & \text{if } v_s \notin o_r^! \wedge v_s \notin 1o_r^? \end{cases} \\
\text{rv}_{d^\#}(\overline{V_c^\#}, \overline{V_r^\#}) &\triangleq \overline{\text{rv}(V_c^\#, V_r^\#)} \\
\text{rv}_{d^\#}(\overline{V_c^\#}, \langle o_r^!, o_r^? \rangle) &\triangleq \langle \text{rv}(\emptyset, o_r^!), \text{rv}(\mathbf{Var}_{\text{caller}} \setminus V_c^\#, o_r^?) \rangle
\end{aligned}$$

4.1 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. We are currently developing an Eclipse plugin to present this information to the programmer. 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 IDE² [28] 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.

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 $\llbracket s \rrbracket(f) \triangleq \lambda d'. \bigsqcup_{d \in \mathbf{Dom} \cup \{0\}} \llbracket s \rrbracket_\bullet(d)(d')(f(d))$.

²interprocedural distributive environment

The pointwise transfer function $\llbracket s \rrbracket_{m^\#}$ from Section 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:

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

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. A transition statement that does not change the *concrete* tracematch state 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. 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 $\langle q_2, m_2 \rangle R_m[\rho] \langle q_2^\#, m_2^\# \rangle$.*

Proof. From the definition of the correctness relation $R_m[\rho]$, $q_1 = q_1^\#$ and $q_2 = q_2^\#$. If $q_1 \neq q_2$, the conclusion is immediate. Suppose instead that all the $q_i, q_i^\#$ are equal, and call this common state q . Then $m_1 \neq m_2$. From the definition of $e^\#$, $\langle q, m_2 \rangle$ 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 $m_1^\#(f)$ is a positive binding containing $\rho(b(f))$ in its must set: In this case, since $m_1(f) R_d[\rho] m_1^\#(f)$, $m_1(f) = \rho(b(f))$, 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 $m_1^\#(f)$ is a positive binding not containing $\rho(b(f))$ in its must set: In this case, $m_1^\#(f) \sqcap e_0^{+\#}(b, O^\#(\rho, \text{range}(b))) = m_1^\#(f) \sqcap \langle n(O^\#(\rho, \text{range}(b)), b(f)), n(O^\#(\rho, \text{range}(b)), b(f)) \rangle$, which contains $\rho(b(f))$ in its must set and is therefore distinct from $m_1^\#(f)$. Therefore $m_1^\# \sqcap e_0^{+\#}(b, O^\#(\rho, \text{range}(b)))$ is a correct abstraction of m_2 , is distinct from $m_1^\#$, and is contained in $\llbracket \mathbf{tr}(T) \rrbracket_{m^\#}[\rho^\#](q_1^\#, m_1^\#)$.

Case $m_2(f) = m_1(f) \sqcap \rho(b(f))$ and $m_1^\#(f)$ is a negative binding containing $b(f)$: In this case, $m_1^\#(f) \sqcap e_0^{+\#}(b, O^\#(\rho, \text{range}(b))) = m_1^\#(f) \sqcap \langle n(O^\#(\rho, \text{range}(b)), b(f)), n(O^\#(\rho, \text{range}(b)), b(f)) \rangle = \perp$, which is distinct from $m_1^\#(f)$. Therefore $m_1^\# \sqcap e_0^{+\#}(b, O^\#(\rho, \text{range}(b)))$ is a correct abstraction of m_2 , is distinct from $m_1^\#$, and is contained in $\llbracket \mathbf{tr}(T) \rrbracket_{m^\#}[\rho^\#](q_1^\#, m_1^\#)$.

Case $m_2(f) = m_1(f) \sqcap \rho(b(f))$ and $m_1^\#(f)$ is a negative binding not containing $b(f)$: In this case, $m_1^\#(f) \sqcap e_0^{+\#}(b, O^\#(\rho, \text{range}(b))) = m_1^\#(f) \sqcap \langle n(O^\#(\rho, \text{range}(b)), b(f)), n(O^\#(\rho, \text{range}(b)), b(f)) \rangle$, which is a positive binding and is therefore distinct from $m_1^\#(f)$. Therefore $m_1^\# \sqcap e_0^{+\#}(b, O^\#(\rho, \text{range}(b)))$ is a correct abstraction of m_2 , is distinct from $m_1^\#$, and is contained in $\llbracket \mathbf{tr}(T) \rrbracket_{m^\#}[\rho^\#](q_1^\#, m_1^\#)$.

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, $m_1^\sharp(f) \sqcap e_0^{+\sharp}(b, O^\sharp(\rho, \text{range}(b))) = m_1^\sharp(f) \sqcap n(O^\sharp(\rho, \text{range}(b)), b(f))$, 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 $\llbracket \text{tr}(T) \rrbracket_{m^\sharp}[\rho^\sharp](q_1^\sharp, m_1^\sharp)$.

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, $m_1^\sharp(f) \sqcap e_0^{+\sharp}(b, O^\sharp(\rho, \text{range}(b))) = m_1^\sharp(f) \sqcap n(O^\sharp(\rho, \text{range}(b)), b(f))$, 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 $\llbracket \text{tr}(T) \rrbracket_{m^\sharp}[\rho^\sharp](q_1^\sharp, m_1^\sharp)$.

□

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 avoid executing the redundant transitions. This can be achieved by peeling one iteration of every loop containing a transition statement prior to performing the analysis. The analysis will mark the transition as relevant in the peeled iteration and unnecessary in the remaining loop.

5 Empirical Evaluation

We empirically evaluated the precision of our analysis and compared it to Bodden et al.’s existing tracematch analysis [8], which uses may-point-to information to rule out possibly matching transition statements. 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. 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 match, 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 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 compares our analysis with FI.

Of the cases where QC is unsuccessful, our analysis rules out a match in 15 cases, compared to 1 case for FI. 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.

The evaluation was performed on the tracematches from [8] 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`, `jython`, and `pmd` from the DaCapo benchmark suite, version 2006-10-MR2 [7]. Most of the benchmarks use reflection to load key classes. We instrumented the benchmarks using ProBe [24] and *J [15] to record actual uses of reflection at run time, and provided the resulting reflection summary to the static analysis. The `jython` 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.

The 54 cases (tracematch/benchmark pairs) 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. We count only final transition statements in the reachable part of the call graph. Of these, the analysis proved that 4796 (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.

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 the analysis proved that 896 will never complete a match. Each of the 36 cases is represented by a circle in Table 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. The 15 fully white circles indicate cases for which the analysis proved that the tracematch cannot match at all.

Of the 21 remaining cases, 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. Similar violations occur in the other three cases (in `jython` and in `HasNext/bloat`). In addition, the other cases contain false positives due to iterators stored only in fields and not local variables. In these two 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. 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 Table 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

³Some transition statements were ruled out in [8] because they were determined to be in code that could not be reached at run time. Our evaluation considers only reachable code.

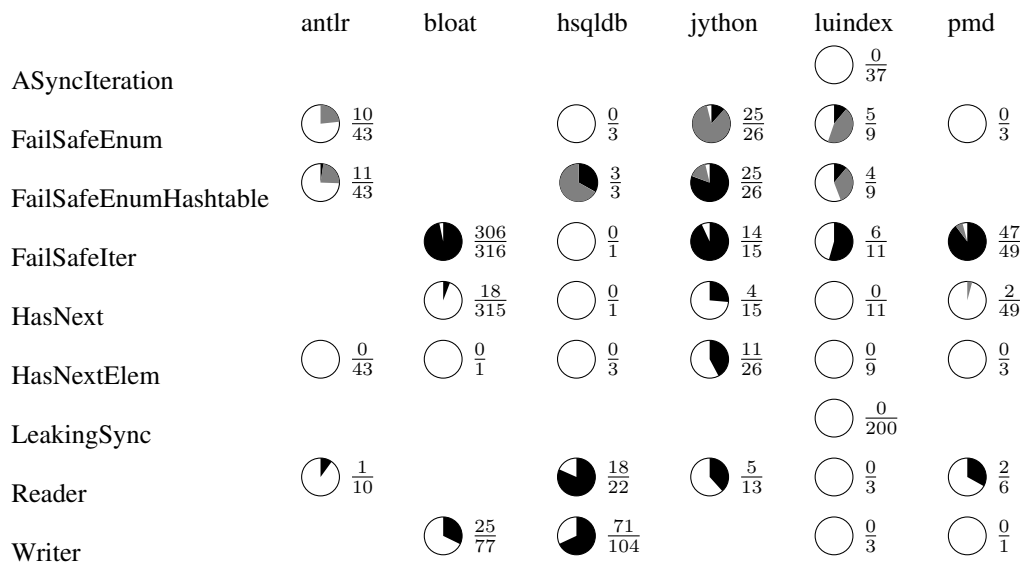


Table 1: Fraction of final transition statements that may complete a match. The white part of each circle represents those that cannot complete a match. The black part represents those at which a match cannot be ruled out, due either to analysis imprecision or an actual violation. The gray part represents those at which a violation is known to exist.

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. We are working on a convenient user interface to visualize the potential update locations and the call chains connecting them to the original loop.

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 match. 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 no local variable references remain to an object. Thus, our analysis would

benefit from some information about pointers from within the heap. We will investigate augmenting the abstraction with such information in future work.

6 Related Work

When tracematches were introduced, space and time overhead of their dynamic implementation was a concern [1]. 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 [4]. 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 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 [9].

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. [8], which we discussed in Section 5. Guyer and Lin's [19, 20] 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.

The static analysis most closely related to our analysis is Fink et al.'s typestate analysis [16]. 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 [10] 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.

Ramalingam et al. [26] 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 [22] 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 typestate 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 [12] 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 [13]. To handle aliasing, they allow objects to be either unaliased and updateable, or possibly aliased and non-updateable. CQual [17] is another system similar to but simpler than Vault. Bierhoff and Aldrich [5,6] 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.

7 Conclusions and Future Work

The analysis we have presented extends static tpestate checking to checking temporal specifications of multiple interacting objects expressed using tracematches. The analysis has been proven sound with respect to the tracematch semantics. A fully context-sensitive version of the analysis has been implemented as two instances of IFDS [27] and IDE [28] algorithms. The analysis was evaluated on the tracematches of Bodden et al. [8] and found to be very precise.

Remaining imprecisions are mainly due to two factors. First, the analysis loses precision when all local variable references to an object are lost. This can be remedied either by making use of may-point-to information, or by adding more precise information about heap references to the object abstraction. Even type information may help in some cases. Second, the analysis fails to verify some tracematches due to imprecise handling of interprocedural exceptional control flow. The precision of exceptional control flow can be improved with suitable modifications to the IFDS and IDE algorithms. We plan to investigate these improvements in the future.

To make the analysis useful to programmers, and to ease our work of interpreting the results, we are developing an Eclipse plugin for presenting the analysis results and for navigating the call graph and control flow graph of the program.

References

- [1] 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.
- [2] 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.
- [3] 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.
- [4] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. *Proceedings of OOPSLA '07*, pages 589–608, 2007.
- [5] K. Bierhoff and J. Aldrich. Lightweight object specification with tpestates. *Proceedings of ESEC/FSE-13*, pages 217–226, 2005.
- [6] K. Bierhoff and J. Aldrich. Modular tpestate checking of aliased objects. *Proceedings of OOPSLA '07*, pages 301–320, 2007.

- [7] 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.
- [8] 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.
- [9] F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. *Proceedings of OOPSLA '07*, pages 569–588, 2007.
- [10] S. Chereem and R. Rugina. Compile-time deallocation of individual objects. *Proceedings of ISMM '06*, pages 138–149, 2006.
- [11] B. A. Davey and H. A. Priestly. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks. Cambridge University Press, first edition, 1990.
- [12] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. *Proceedings of PLDI '01*, pages 59–69, 2001.
- [13] R. DeLine and M. Fähndrich. Typestates for objects. *Proceedings of ECOOP 2004*, volume 3086 of *LNCS*, pages 465–490, 2004.
- [14] 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.
- [15] B. Dufour. Objective quantification of program behaviour using dynamic metrics. Master’s thesis, McGill University, June 2004.
- [16] 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.
- [17] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. *Proceedings of PLDI '02*, pages 1–12, 2002.
- [18] S. Goldsmith, R. O’Callahan, and A. Aiken. Relational queries over program traces. *Proceedings of OOPSLA '05*, pages 385–402, 2005.
- [19] S. Z. Guyer and C. Lin. Client-driven pointer analysis. *Proceedings of SAS '03*, volume 2694 of *LNCS*, pages 214–236, 2003.
- [20] S. Z. Guyer and C. Lin. Error checking with client-driven pointer analysis. *Sci. Comput. Program.*, 58(1-2):83–114, 2005.
- [21] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. *Proceedings of POPL '05*, pages 310–323, 2005.
- [22] 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.
- [23] H. B. M. Jonkers. Abstract storage structures. de Bakker and van Vliet, editors, *Algorithmic Languages*, pages 321–343. IFIP, North Holland, 1981.
- [24] O. Lhoták. Comparing call graphs. *Proceedings of PASTE '07*, pages 37–42, 2007.

- [25] 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.
- [26] 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.
- [27] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. *Proceedings of POPL '95*, pages 49–61, 1995.
- [28] 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.
- [29] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50, Jan. 1998.
- [30] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.