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 which is proved equivalent to the original tracematch semantics but is better suited to static analysis. The static analysis is presented next, and is proved sound with respect to the semantics. The analysis computes precise local points-to sets and tracks the flow of individual objects, thereby enabling strong state updates. A fully 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 [27]. 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, 15–19, 21, 23]. 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 [1,9,17,23]. 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

*Technical Report CS-2007-49, extended version of a paper submitted to ECOOP 2008. This technical report was previously numbered CS-2007-18. The number was changed for administrative reasons with no changes to the contents of the report. 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(i);
9      }
10   }
11 }
```

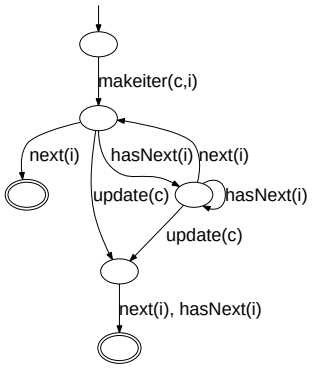

Figure 1: Tracematch example: iterator safety

time. For our analysis, tracematches have two advantages over similar formalisms. First, they are widely applicable because their semantics is intuitive and highly expressive compared to other regular-expression-based formalisms. A key issue in defining such formalisms is how to tease apart the interactions between operations on different objects; in some other systems, operations on different objects are not cleanly separated. Conceptually, a tracematch executes a separate copy of a finite automaton for every possible combination of runtime objects. While other systems require each automaton to bind all objects on the first state transition, tracematches do not have this restriction. Second, the semantics of tracematches has been formally specified, which allows us to formally prove that the static analysis soundly abstracts the semantics. The original tracematch paper motivates the design of a declarative semantics from the programmer's point of view, then proves it equivalent to an operational semantics better suited for implementation [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 typestate verification is that in a tracematch, typestate is associated not with a single object, but with a group of objects. Existing work on typestate verification (e.g. [15]) 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 similar to earlier work [10, 15, 20, 26]. 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 on the right checks that a list is not updated during iteration, and that every call to `next` on an iterator is preceded by a call to `hasNext`. A violation of

the property causes the automaton to enter one of the final states. The tracematch associated with this automaton 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` occur 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 [24] and IDE [25] frameworks which efficiently support fully context-sensitive interprocedural analyses. (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] defined a declarative semantics of how tracematches ought to work, as well as an operational semantics that they proved equivalent. We begin by reviewing their operational semantics and formalizing a few details that were left implicit. Then we define an equivalent operational semantics based on sets and lattices. Finally, we extend the operational semantics to a complete intermediate representation that includes instructions not directly related to tracematches.

2.1 Original Operational Semantics

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. Each operation in the tracematch is defined in terms of a *pointcut*, a predicate over instructions that may also bind objects involved in an instruction to tracematch parameters. In the abc compiler, a *matching* phase identifies the set of instructions that match each pointcut, and a *weaving* phase inserts additional code to update the tracematch state accordingly. We assume that these phases have already been performed. Thus, the input to our static analysis is the original code with additional tracematch *transition statements* that specify how to update the tracematch state.

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

Definition 1. *[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. We let $\langle Q, A, q_0, Q_f, \delta\rangle$ be a finite automaton that recognizes P. As is customary, Q is a finite set of states, A is a finite alphabet, in this case the set of tracematch symbols, $q_0 \in Q$ is the unique start state, $Q_f \subseteq Q$ is a set of final states, and $\delta \subseteq Q \times A \times Q$ is a transition relation. The automaton need not be deterministic.*

A transition statement comprises *transition elements*, which specify operations that cause the tracematch to change states.

Definition 2. *A* transition element *is a pair $\langle a, b\rangle$, where*

$a \in A$ *is a symbol of the tracematch, and*

$b : F \hookrightarrow \mathbf{Var}$ *is a partial map specifying the value to be bound to each of a subset of the tracematch parameters. When the transition element is executed, each parameter $f \in dom(b)$ is bound to the object currently pointed to by program variable $b(f)$.*

Definition 3. *A* transition statement ***tr***(T) *is a set T of transition elements.*

Because multiple pointcuts may match the same instruction, the tracematch semantics allows multiple transition elements in a single statement. When such a statement is executed, the tracematch non-deterministically follows the transitions specified by each transition element individually.

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 $\mathring{\sigma} : Q \to S$ 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 tracematch element $\langle a, b\rangle$ 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:

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

The formula for a set T of multiple transition elements is a disjunction of the formulas for the transition statements, since the tracematch non-deterministically executes all of the transition elements:

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

Recall that a tracematch state $\mathring{\sigma}$ 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. $\mathring{e}_0(b,\rho)$ is satisfied), or remains in its current state if its bindings are inconsistent (i.e. $\neg\mathring{e}_0(b,\rho)$ is satisfied). Thus, the transition function is defined [1,4] as:

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

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

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:

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

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

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. To avoid complicating the semantics of $\mathbf{tr}(T)$, we define a separate **body** statement to perform these tasks. In the intermediate representation, a **body** is inserted immediately following every transition statement $\mathbf{tr}(T)$. The semantics of **body** is to reset the formulas of all final states:

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

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.

2.2 A Lattice-Based Operational Semantics

To enable static analysis, we define an equivalent semantics using sets and lattices, which are easier to abstract statically. The core of the construction is a *binding lattice* which specifies that a tracematch parameter has been bound to some object (a *positive binding*) or that it has not yet been bound and may be bound later to any object except those in a given set (a *negative binding*).

The binding lattice $\langle \mathbf{Bind},\sqsubseteq\rangle$ is defined as follows. Its elements are $\mathbf{Bind} \triangleq \mathbf{Obj}\uplus\mathcal{P}(\mathbf{Obj})\uplus\{\bot\}$. An element $o\in\mathbf{Obj}$ indicates a positive binding while a set from $\mathcal{P}(\mathbf{Obj})$ indicates a negative binding. The element $\bot$ indicates an inconsistent binding that will never lead to a match. As a reminder that a set of values indicates negative bindings, we will always write such a set with a bar above it: $\overline{O}$. The bar is only a reminder; it has no semantic meaning. The partial order $\sqsubseteq$ is defined as the reflexive transitive closure of the following rules: $\bot\sqsubseteq d$ for any d; $\overline{O_1}\sqsubseteq\overline{O_2}$ if $O_1\supseteq O_2$; and $o_1\sqsubseteq\overline{O_2}$ if $o_1\notin O_2$. Intuitively, for an element higher in the order, there is more freedom to bind new objects than for a lower element. We use $\top$ as a synonym for the empty set of negative bindings, since $\top=\overline{\{\}}\sqsupseteq d$ for every d. The following proposition assures us that the binding lattice is indeed a lattice and provides a meet.

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

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

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

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

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

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

Case $o \in D$ and none of the above cases hold: In this case, D does not contain $\bot$ 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 $\bot$. Since $o \sqsupseteq \bot$, 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 $\bot$. All of these are less than $\bigcup_{\overline{O} \in D} O$. Thus the latter is the glb.

Since $\langle$**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]. □

Intuitively, if $d \in D$ is the current binding for a variable and we positively (resp. negatively) bind o to the same variable, the meet $d \sqcap o$ (resp. $d \sqcap \overline{\{o\}}$) gives the updated binding for the variable. The meet is extended pointwise on maps from F to **Bind**.

In the lattice-based semantics, the concrete runtime state of a tracematch is $\sigma \subseteq Q \times (F \to \textbf{Bind})$. That is, the state is a set of pairs each containing an automaton state and a map that associates an element of **Bind** with each tracematch parameter. We use **State** $\triangleq \mathcal{P}(Q \times (F \to \textbf{Bind}))$ to denote the domain of all possible tracematch states.

We begin defining the transition function by defining a binding map analogous to $\mathring{e}_0(b, \rho)$ which specifies that the tracematch parameters must be consistent with the objects bound in a transition element $\langle a, b \rangle$. Each parameter not bound is mapped to $\top$ to remain unrestricted.

$$e_0^+(b, \rho) \triangleq \lambda f. \begin{cases} \rho(b(f)) & \text{if } f \in \text{dom}(b) \\ \top & \text{otherwise} \end{cases}$$

For negative bindings, a similar map is defined. However, to be consistent, *all* tracematch parameters must be consistent, while to be inconsistent, *at least one* parameter need be inconsistent. Thus, the negative map is $\top$ for all but one tracematch parameter:

$$e_0^-(b, \rho, f) \triangleq \lambda f'. \begin{cases} \overline{\{\rho(b(f))\}} & \text{if } f = f' \\ \top & \text{otherwise} \end{cases}$$

At a transition statement, given a current state q and binding map m, we either transition the state along the automaton and update the binding map positively, or we stay in the current state and update the binding map negatively:

$$e^+[a, b, \rho](q, m) \triangleq \left\{ \left\langle q', m \sqcap e_0^+(b, \rho) \right\rangle : \delta(q, a, q') \right\}$$

$$e^-[b, \rho](q, m) \triangleq \left\{ \left\langle q, m \sqcap e_0^-(b, \rho, f) \right\rangle : f \in \text{dom}(b) \right\}$$

$$e[\{\langle a, b\rangle\}, \rho](q, m) \triangleq e^+[a, b, \rho](q, m) \cup e^-[b, \rho](q, m)$$

When a transition statement contains multiple transition elements, 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:

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

The tracematch transition statement performs the above operations 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:

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

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:

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

We have been careful to construct the lattice-based semantics to be equivalent to the original tracematch 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 $\mathring{\sigma}$ in the boolean-formula-based semantics.

$$s_d(\langle f, d\rangle) \triangleq \begin{cases} \text{false} & \text{if } d = \perp \\ (f = o) & \text{if } d \text{ is a positive binding } o \\ \bigwedge_{o \in \overline{O}} \neg(f = o) & \text{if } d \text{ is a negative binding } \overline{O} \end{cases}$$

$$s_m(m) \triangleq \bigwedge_{f \in F} s_d(\langle f, m(f)\rangle)$$

$$s_\sigma(\sigma) \triangleq \lambda q. \bigvee_{\langle q,m\rangle \in \sigma} s_m(m)$$

We have proven that the two semantics are bisimilar:

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

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

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 $d_1 = \bot$ **or** $d_2 = \bot$**:** Then $s_d(\langle f, d_1 \sqcap d_2 \rangle) = s_d(\langle f, \bot \rangle) =$ false. On the other side, $s_d(\langle f, d_1 \rangle) =$ false or $s_d(\langle f, d_2 \rangle =$ 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 $s_d(\langle f, d_1 \sqcap d_2 \rangle) = s_d(\langle f, \bot \rangle) =$ false. On the other side, $s_d(\langle f, d_1 \rangle) \wedge s_d(\langle f, d_2 \rangle) = (f = o_1) \wedge (f = o_2) =$ false since $o_1 \neq o_2$.

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

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

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

□

Lemma 2.

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

Proof.

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

□

Lemma 3.

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

Proof.

$$\begin{aligned}
s_m(e_+(b,\rho)) &= \bigwedge_{f\in F} s_d(\langle f, e_+(b,\rho)(f)\rangle) && \text{definition of } s_m\\
&= \bigwedge_{f\in dom(b)} s_d(\langle f, \rho(b(f))\rangle) \wedge \bigwedge_{f\notin dom(b)} s_d\left(\left\langle f, \overline{\emptyset}\right\rangle\right) && \text{definition of } e_+\\
&= \bigwedge_{f\in dom(b)} f = \rho(b(f)) \wedge \bigwedge_{f\notin dom(b)} \bigwedge_{o\in\emptyset} \neg(f = o) && \text{definition of } s_d\\
&= \mathring{e_0}(b,\rho) \wedge \text{true} && \text{empty conjunction}\\
&= \mathring{e_0}(b,\rho)
\end{aligned}$$

□

Lemma 4.

$$\bigvee_{f\in dom(b)} s_m(e_-(b,\rho,f)) = \neg\mathring{e_0}(b,\rho)$$

Proof.

$$\begin{aligned}
\bigvee_{f\in dom(b)} s_m(e_-(b,\rho,f)) &= \bigvee_{f\in dom(b)} \bigwedge_{f'\in F} s_d(\langle f', e_-(b,\rho,f')(f')\rangle)\\
&= \bigvee_{f\in dom(b)} \left(s_d\left(\left\langle f, \overline{\rho(b(f))}\right\rangle\right) \wedge \bigwedge_{f'\in\{F\setminus f\}} s_d\left(\left\langle f', \overline{\emptyset}\right\rangle\right)\right)\\
&= \bigvee_{f\in dom(b)} \left(\neg(f = \rho(b(f))) \wedge \bigwedge_{f'\in\{F\setminus f\}} \bigwedge_{o\in\emptyset} \neg(f' = o)\right)\\
&= \bigvee_{f\in dom(b)} \neg(f = \rho(b(f)))\\
&= \neg \bigwedge_{f\in dom(b)} (f = \rho(b(f)))\\
&= \neg\mathring{e_0}(b,\rho)
\end{aligned}$$

□

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

$$\bigvee_{\langle q,m\rangle\in e_-[b_n,\rho](\cdots(e_-[b_1,\rho](\sigma))\cdots)} s_m(m) = s_\sigma(\sigma)(q) \wedge \bigwedge_{1\le i\le n} \neg\mathring{e_0}(b_i,\rho)$$

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 $s_\sigma(\sigma)(q) \wedge$ true. These are equal by the definition of s_σ.

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

$$\bigvee_{\langle q,m\rangle\in\sigma'} s_m(m) = s_\sigma(\sigma)(q) \wedge \bigwedge_{1\le i\le n-1} \neg\mathring{e_0}(b_i,\rho)$$

then

$$\bigvee_{\langle q,m\rangle\in e_-[b_n,\rho](\sigma')} s_m(m) = s_\sigma(\sigma)(q) \wedge \bigwedge_{1\le i\le n} \neg\mathring{e_0}(b_i,\rho)$$

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

Case $\mathbf{dom}(b_n) \neq \emptyset$ 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 \mathrm{dom}(b_n)} s_m(m \sqcap e_-(b_n, \rho, f)) \\
=&\bigvee_{\langle q,m\rangle \in \sigma'} \bigvee_{f \in \mathrm{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 \mathrm{dom}(b_n)} \bigwedge_{f' \in F} s_d\left(\left\langle f', m[f \mapsto m(f) \sqcap \overline{\{\rho(b_n(f))\}}](f')\right\rangle\right) \\
=&\bigvee_{\langle q,m\rangle \in \sigma'} \bigvee_{f \in \mathrm{dom}(b_n)} \left(s_d\left(\left\langle f, m(f) \sqcap \overline{\{\rho(b_n(f))\}}\right\rangle\right) \wedge \bigwedge_{f' \in F\setminus\{f\}} s_d\left(\langle f', m(f')\rangle\right)\right) \\
=&\bigvee_{\langle q,m\rangle \in \sigma'} \bigvee_{f \in \mathrm{dom}(b_n)} \left(s_d\left(\langle f, m(f)\rangle\right) \wedge s_d\left(\left\langle f, \overline{\{\rho(b_n(f))\}}\right\rangle\right) \wedge \bigwedge_{f' \in F\setminus\{f\}} s_d\left(\langle f', m(f')\rangle\right)\right) \\
=&\bigvee_{\langle q,m\rangle \in \sigma'} \bigvee_{f \in \mathrm{dom}(b_n)} \left(s_d\left(\left\langle f, \overline{\rho(b_n(f))}\right\rangle\right) \wedge \bigwedge_{f' \in F} s_d\left(\langle f', m(f')\rangle\right)\right) \\
=&\bigvee_{\langle q,m\rangle \in \sigma'} \bigvee_{f \in \mathrm{dom}(b_n)} \neg(f = \rho(b_n(f))) \wedge s_m\left(m\right) \\
=&\left(\bigvee_{\langle q,m\rangle \in \sigma'} \bigvee_{f \in \mathrm{dom}(b_n)} \neg(f = \rho(b_n(f)))\right) \wedge \left(\bigvee_{\langle q,m\rangle \in \sigma'} \bigvee_{f \in \mathrm{dom}(b_n)} s_m\left(m\right)\right) \\
=&\left(\neg \bigwedge_{f \in \mathrm{dom}(b_n)} (f = \rho(b_n(f)))\right) \wedge \left(\bigvee_{\langle q,m\rangle \in \sigma'} s_m\left(m\right)\right) \\
=&\neg \mathring{e}_0(b_n, \rho) \wedge s_\sigma(\sigma)(q) \wedge \bigwedge_{1 \leq i \leq n-1} \neg \mathring{e}_0(b_i, \rho) \\
=&s_\sigma(\sigma)(q) \wedge \bigwedge_{1 \leq i \leq n} \neg \mathring{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 $\mathring{\sigma} \in Q \to S$, there exists a $\sigma \in$ **State** such that for all $q \in Q$, $\mathring{\sigma}(q) \iff s_\sigma(\sigma)(q)$.*

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

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

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

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

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

$$
\begin{aligned}
\mathring{\sigma}' &= \lambda q. \left(\bigvee_{a,j:\delta(j,a,q)} (\mathring{\sigma}[q_0 \mapsto \text{true}](j) \wedge \mathring{e}(a, \{\langle a_1, b_1\rangle \cdots \langle a_n, b_n\rangle\}, \rho)) \right) \vee \left(\mathring{\sigma}[q_0 \mapsto \text{true}](q) \wedge \bigwedge_{a \in A} \neg \mathring{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} \mathring{e_0}(b_i, \rho) \right) \right) \vee \left(s_\sigma(\sigma \cup \{\langle q_0, \lambda f.\top\rangle\})(q) \wedge \bigwedge_{1 \le i \le n} \neg \mathring{e_0}(b_i, \rho) \right) \\
&= \lambda q. \left(\bigvee_{a,j:\delta(j,a,q)} \left(\bigvee_{\langle q,m\rangle \in \sigma \cup \{\langle q_0, \lambda f.\top\rangle\}} s_m(m) \wedge \bigvee_{i:a_i = a} s_m(e_+(b_i, \rho)) \right) \right) \vee \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 \le i \le 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 \le i \le 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 \le i \le 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 \left(\bigcup_{1 \le i \le 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)} s_m(m) \\
&= s_\sigma \left(\left(\bigcup_{1 \le i \le 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}
$$

□

2.3 Full Intermediate Representation

So far, we have considered only the two statements that directly affect the tracematch state, $\mathbf{tr}(T)$ and **body**. We now extend the intermediate representation (IR) with the instructions necessary to express other interesting operations that occur in Java programs. The full set of instructions in the intraprocedural IR is:
$s ::= v_1 \leftarrow v_2 \mid v \leftarrow e \mid e \leftarrow v \mid v \leftarrow \textbf{new} \mid v \leftarrow \textbf{null} \mid \mathbf{tr}(T) \mid \textbf{body}$

In the instructions, v represents a local variable and e represents any expression accessing the heap, such as an access of a field or array element. In many similar analyses, different accesses to the heap are distinguished and the analysis reasons about the heap. We do not do this for two reasons. First, our focus is on the tracematch analysis rather than analysis of the object abstraction, which has been thoroughly studied in previous work. More precise object abstractions from previous work could be substituted. Second, as long as the program maintains *some* reference from a local variable to the object being tracked, that reference gives the analysis the most precise information possible about the identity of the object, because unlike heap locations, local variables cannot be aliased. Therefore, a tracematch analysis should model references from local variables as precisely as possible (including interprocedurally and context-sensitively), since local variables are the most likely source of precise information. Although information about heap references (such as the reference counts of [10] or the access paths of [15]) can be added to the analysis, our focus is on taking full advantage of the information available from local variables.

To the operational semantics, we add a set h containing all objects referenced from the heap. The instructions $\mathbf{tr}(T)$ and **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:

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

3 Static Abstraction

The static analysis is presented in two parts. We first define the object abstraction, then the abstraction of tracematch states.

3.1 Object Abstraction

The object abstraction represents an object by the set of local variables pointing to them. This is the same abstraction as the nodes in Sagiv et al.'s shape analysis [26]. However, the tracematch object abstraction is simpler than the shape analysis because it tracks only the nodes, not the pointer edges between objects. We define $\mathbf{Obj}^\sharp \triangleq \mathcal{P}(\mathbf{Var})$ as the set of all sets of variables. The function $\beta_o[\rho] : \mathbf{Obj} \rightarrow \mathbf{Obj}^\sharp$ gives for each concrete object o its abstract counterpart, the set of variables pointing to it:

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

The set of variables in the abstraction of each object is exact; it is neither a may-point-to nor a must-point-to approximation. In addition, every abstract object except the empty set $\emptyset$ represents at most one concrete object at any given point of execution, since a given variable only points to one object at a time. This enables very precise flow-sensitive analysis including strong updates.

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

Figure 2: Transfer function for the object abstraction

The analysis computes for each program point a set containing all abstract objects for which a concrete object may exist. Because the objects are represented by the local variables pointing to them, the set can be thought of as an abstraction of the concrete environment ρ. The abstraction function $\beta_\rho : \textbf{Env} \times \mathcal{P}(\textbf{Var}) \to \mathcal{P}(\textbf{Obj}^\sharp)$ for environments is defined as:

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

The set captures both may-alias and must-alias relationships between local variables. If variables x and y point to distinct objects, $\beta_\rho(\rho, h)$ will not contain any set containing both x and y. If variables x and y point to the same object, every set in $\beta_\rho(\rho, h)$ will contain either both x and y, or neither of them.

In addition to the sets of variables, the abstraction also tracks a subset $h^\sharp \subseteq \textbf{Obj}^\sharp$ of nodes that represent objects that may be pointed to from the heap, rather than only from local variables. The heap abstraction is defined by:

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

Finally, we combine β_ρ and β_h into a single 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 $\left\langle \rho_1^\sharp, h_1^\sharp \right\rangle \sqsubseteq \left\langle \rho_2^\sharp, h_2^\sharp \right\rangle$ if $\rho_1^\sharp \subseteq \rho_2^\sharp \wedge h_1^\sharp \subseteq h_2^\sharp$, which induces a join operator $\left\langle \rho_1^\sharp, h_1^\sharp \right\rangle \sqcup \left\langle \rho_2^\sharp, h_2^\sharp \right\rangle \triangleq \left\langle \rho_1^\sharp \cup \rho_2^\sharp, h_1^\sharp \cup h_2^\sharp \right\rangle$. The property that $\rho^\sharp \supseteq h^\sharp$ is always maintained.

The transfer function $[\![s]\!]_{\rho h^\sharp}$ on this combined abstraction is defined in Figure 2 in terms of several helper functions. The core helper function is $[\![s]\!]_{o^\sharp}$, which models the effect of each instruction on the objects represented by an abstract object $o^\sharp$. For statements that write to local variables, $o^\sharp$ is updated to contain the new set of variables pointing to the object. When the analysis encounters a heap load instruction, it is uncertain whether the object being loaded is represented by $o^\sharp$ and whether the destination variable v should therefore be added to $o^\sharp$. Thus, we leave $[\![v \leftarrow e]\!]_{o^\sharp}$ undefined. For all other statements, however, $[\![s]\!]_{o^\sharp}$ gives the exact new abstraction of every object. This property is key to tracking the flow of individual objects (as mentioned in the introduction) and is used again in the tracematch state abstraction. Formally:

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

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

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

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

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

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

Case $s = v \leftarrow$ **null:**

$$\begin{aligned}
\beta_o[\rho'](o) &= \beta_o[\rho[v \mapsto \bot]](o) \\
&= \{v' : \rho(v') = o\} \setminus \{v\} \\
&= \beta_o[\rho](o) \setminus \{v\} \\
&= [\![s]\!]_{o^\sharp}(\beta_o[\rho](o))
\end{aligned}$$

Case $v \leftarrow$ **new:**

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

Case $s \in \{e \leftarrow v, \mathbf{tr}(T), \mathbf{body}\}$**:** For these statements, $\rho' = \rho$ and $[\![s]\!]_{o^\sharp}$ is the identity. Thus $[\![s]\!]_{o^\sharp}(\beta_o[\rho](o)) = \beta_o[\rho](o) = \beta_o[\rho'](o)$.

□

To precisely handle the uncertainty in heap loads we use the materialization or focus operation from [10,15,20,26]. The abstract object $o^\sharp$ is split into two, one representing the single concrete object that may have been loaded, and the other representing all other objects previously represented by $o^\sharp$. This is done only if $o^\sharp \in h^\sharp$ (i.e. if $o^\sharp$ may represent an object referenced from the heap). Focus is important to regain the precision lost when an object is referenced only from the heap, in which case the analysis lumps it together with all other such objects. However, in order for a tracematch operation to be performed on the object, it must first be loaded into a variable, at which point focus separates it from the other objects. If multiple tracematch operations are performed on it, this uniqueness is necessary to guarantee that the operations are performed on the same concrete object.

Two additional special cases are handled. For an allocation instruction, a new abstract object containing only the destination variable v is created and added to $\rho^\sharp$. For a heap store instruction, all abstract objects containing the variable v being written are added to $h^\sharp$, the set of abstract objects that may be reachable from the heap.

On the object abstraction, we define the correctness relation $\langle \rho, h\rangle \, R_{\rho h} \, \langle \rho^\sharp, h^\sharp \rangle$ if $\beta_{\rho h}(\rho, h) \sqsubseteq \langle \rho^\sharp, h^\sharp \rangle$. The transfer function preserves the correctness relation:

Theorem 2. *If* $\langle s, \rho, h, \sigma\rangle \to \langle \rho', h', \sigma'\rangle$ *and* $\langle \rho, h\rangle \, R_{\rho h} \, \langle \rho^\sharp, h^\sharp \rangle$*, then* $\langle \rho', h'\rangle \, R_{\rho h} \, [\![s]\!]_{\rho h^\sharp}(\rho^\sharp, h^\sharp)$.

The following lemma is needed to prove the theorem.

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

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

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

Case $s = v \leftarrow e$**:** range(ρ') = range$(\rho[v \mapsto o])$ for some $o \in h$. Thus range$(\rho') \cup h'$ = range$(\rho[v \mapsto o]) \cup h \subseteq$ range$(\rho) \cup h$ since $o \in h$.

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

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

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

□

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

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

Case $s \in \{v_1 \leftarrow v_2, v \leftarrow$ **null**$, e \leftarrow v\}$**:**

$$
\begin{aligned}
[\![s]\!]_{\rho^\sharp}(\rho^\sharp, h^\sharp) &= [\![s]\!]_{O^\sharp}[h^\sharp](\rho^\sharp) \\
&= \{[\![s]\!]_{o^\sharp}(o^\sharp) : o^\sharp \in \rho^\sharp\} \\
&= \{[\![s]\!]_{o^\sharp}(\beta_o[\rho](o)) : o \in h \cup \text{range}(\rho) \setminus \{\bot\}\} \\
&\supseteq \{[\![s]\!]_{o^\sharp}(\beta_o[\rho](o)) : o \in h' \cup \text{range}(\rho') \setminus \{\bot\}\} \text{ (Using Lemma 7)} \\
&= \{\beta_o[\rho'](o) : o \in h' \cup \text{range}(\rho') \setminus \{\bot\}\} \text{ (Using Proposition 2)} \\
&= \beta_\rho(\rho', h')
\end{aligned}
$$

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

$$\begin{aligned}
[\![s]\!]_{\rho^\sharp}(\rho^\sharp, h^\sharp) &= [\![s]\!]_{O^\sharp}[h^\sharp](\rho^\sharp) \cup \{v\} \\
&= \{[\![s]\!]_{o^\sharp}(o^\sharp) : o^\sharp \in \rho^\sharp\} \cup \{v\} \\
&= \{[\![s]\!]_{o^\sharp}(\beta_o[\rho](o)) : o \in h \cup \text{range}(\rho) \setminus \{\bot\}\} \cup \{v\} \\
&= \{\beta_o[\rho'](o) : o \in h \cup \text{range}(\rho) \setminus \{\bot\}\} \cup \{v\} \text{ Proposition 2} \\
&= \{\beta_o[\rho'](o) : o \in h \cup \text{range}(\rho) \setminus \{\bot\} \cup o'\} \text{ since } \{v\} = \beta_o[\rho'](o') \\
&\supseteq \{\beta_o[\rho'](o) : o \in h' \cup \text{range}(\rho') \setminus \{\bot\}\} \\
&= \beta_\rho(\rho', h')
\end{aligned}$$

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

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

Case $s \in \{\textbf{tr}(T), \textbf{body}\}$**:** In this case, $\rho' = \rho$, $h' = h$, thus $[\![s]\!]_{\rho^\sharp}(\rho^\sharp, h^\sharp) = \rho^\sharp = \beta_\rho(\rho', h')$.

Next we prove $\beta_h(\rho', h') \subseteq [\![s]\!]_{h^\sharp}(\rho^\sharp, h^\sharp)$.

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

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

Case $s = e \leftarrow v$**:**

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

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

$$\begin{aligned}
[\![s]\!]_{h^\sharp}(\rho^\sharp, h^\sharp) &= [\![s]\!]_{O^\sharp}[h^\sharp](h^\sharp) \\
&= \bigcup_{o^\sharp \in h^\sharp} \mathit{focus}[h^\sharp](o^\sharp) \\
&= \bigcup_{o^\sharp \in h^\sharp} \{o^\sharp \setminus \{v\}, o^\sharp \cup \{v\}\} \\
&= \bigcup_{o \in h} \{\beta_o[\rho](o) \setminus \{v\}, \beta_o[\rho](o) \cup \{v\}\} \\
&\supseteq \bigcup_{o \in h} \{\beta_o[\rho[v \mapsto o']](o)\} \\
&= \{\beta_o[\rho'](o) : o \in h\} \\
&= \beta_h(\rho', h) \\
&= \beta_h(\rho', h')
\end{aligned}$$

Case $s \in \{\mathbf{tr}(T), \mathbf{body}\}$**:** In this case, $\rho' = \rho$, $h' = h$, thus $[\![s]\!]_{h^\sharp}(\rho^\sharp, h^\sharp) = h^\sharp = \beta_h(\rho', h')$.

This completes the proof for the special case when $\langle \rho^\sharp, h^\sharp \rangle = \beta_{\rho h}(\rho, h)$. In general, $\langle \rho^\sharp, h^\sharp \rangle \sqsupseteq \beta_{\rho h}(\rho, h)$. Since $[\![s]\!]_{\rho h^\sharp}$ is monotone, $[\![s]\!]_{\rho h^\sharp}(\rho^\sharp, h^\sharp) \sqsupseteq [\![s]\!]_{\rho h^\sharp}(\beta_{\rho h}(\rho, h))$, which we just proved is greater than $\beta_{\rho h}(\rho', h')$. Thus, the theorem holds in the general case. □

3.2 Tracematch Abstraction

We next define an abstraction of the tracematch state. An obvious abstraction would be to simply reuse the concrete state domain **State**, but replace each concrete object o in **Bind** with its abstract counterpart $\beta_o(o)$. Proposition 2 would guarantee that an object bound by the tracematch would be correctly propagated if we reused $[\![s]\!]_{o^\sharp}$ within the tracematch abstraction. We experimented with this abstraction but it did not scale. When an object is referenced from the heap, the focus operation splits it into two abstract objects. When the abstract object is part of an abstract tracematch state, focusing the object requires the abstract tracematch state to be split. Doing this many times resulted in excessive numbers of abstract tracematch states. In fact, there is little benefit to maintaining the precision provided by 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 operations is performed, we can be sure that they are performed on

the same concrete object. Thus, focus is needed for precise must-alias information at the transition statement where an object is bound. However, once an 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, to abstract the binding lattice, we replace the precise set of exactly the variables pointing to the object with an under- and over-approximation:

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

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 $\left\langle o^!, o^? \right\rangle$ 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:

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

We extend β_d pointwise to maps $F \to \mathbf{Bind}^\sharp$ and to the overall tracematch state $\mathbf{State}^\sharp \triangleq \mathcal{P}\left(Q \times \left(F \to \mathbf{Bind}^\sharp\right)\right)$ 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 **Bind**, is defined as the reflexive transitive closure of the following rules: $\bot \sqsubseteq x$ for any x; $\overline{V_1^\sharp} \sqsubseteq \overline{V_2^\sharp}$ if $V_1^\sharp \supseteq V_2^\sharp$; $\left\langle o^!, o^? \right\rangle \sqsubseteq \overline{V^\sharp}$ if $o^! \cap V^\sharp = \emptyset$; and $\left\langle o_1^!, o_1^? \right\rangle \sqsubseteq \left\langle o_2^!, o_2^? \right\rangle$ 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 **Bind** in $\mathbf{Bind}^\sharp$.

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

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

where pos $\left(o^!, o^?\right) \triangleq \begin{cases} \left\langle o^!, o^? \right\rangle & \textit{if } o^! \subseteq o^? \\ \bot & \textit{otherwise} \end{cases}$

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

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

A lower bound of two positive bindings $d_1^\sharp, d_2^\sharp$ can be either $\bot$ 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 $\bot$, 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 $\bot$.

The case of a meet of a positive binding and a negative binding is similar. Any lower bound must be either $\bot$ 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 $\bot$. Thus it is the glb. However, when it does not respect the subset restriction on positive bindings, only $\bot$ 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 $\bot$, 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. □

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 $d_1 = \bot$, $d_1^\sharp$ is also $\bot$, 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 $d_1^\sharp \sqsubseteq d_2^\sharp$. □

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

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

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 $\bot$ for any f. The correctness relation ensures that if this happens, the abstract state $\sigma^\sharp$ 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 3. The helper function $[\![s]\!]_{d^\sharp}$ is similar to $[\![s]\!]_{o^\sharp}$ 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 $[\![s]\!]_{m^\sharp}$ and $[\![s]\!]_{\sigma^\sharp}$. Since $[\![s]\!]_{d^\sharp}$ is so similar to $[\![s]\!]_{o^\sharp}$, we can prove an analogue of Proposition 2 for it:

Proposition 5. *If $\langle s, \rho \rangle \to \langle \rho' \rangle$ then $d \; R_d[\rho] \; d^\sharp \implies d \; R_d[\rho'] \; [\![s]\!]_{d^\sharp}(d^\sharp)$.*

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 $\bot$, 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 $o^! \subseteq \beta_o[\rho](o) \subseteq o^?$. □

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

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

Figure 3: Transfer functions for the tracematch state abstraction for $s \neq \textbf{tr}(T)$

Proof of Proposition 5. **Case** $d = \bot$**:** Then $\beta_d[\rho](d) = \bot \sqsubseteq [\![s]\!]_{d^\sharp}(d^\sharp)$, so $R_d[\rho']\ [\![s]\!]_{d^\sharp}(d^\sharp)$.

Case d **is a positive binding** o**:** By Lemma 8, $d^\sharp$ is either a negative binding or $\left\langle o^!, o^? \right\rangle$. If $d^\sharp$ is a negative binding, then so is $[\![s]\!]_{d^\sharp}(d^\sharp)$, so since $\beta_d[\rho](d)$ is less than any negative binding, $d\ R_d[\rho']\ [\![s]\!]_{d^\sharp}(d^\sharp)$. Thus, the remaining case is when $d^\sharp = \left\langle o^!, o^? \right\rangle$. By Lemma 8, $o^! \subseteq \beta_o[\rho](o^\sharp)o^?$.

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

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

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

Subcase $s = v_1 \leftarrow v_2 \wedge v_2 \notin o^! \wedge v_2 \in o^?$:

$$
\begin{aligned}
o\ R_d[\rho]\ \left\langle o^!, o^? \right\rangle &\implies o^! \subseteq \beta_o[\rho](o) \subseteq o^? \\
&\implies o^! \setminus \{v_1\} \subseteq \beta_o[\rho](o) \setminus \{v_1\} \subseteq \beta_o[\rho](o) \cup \{v_1\} \subseteq o^? \cup \{v_1\} \\
&\implies o^! \cup \{v_1\} \subseteq [\![s]\!]_{o^\sharp}(\beta_o[\rho](o)) \subseteq o^? \cup \{v_1\} && \text{definition of } [\![s]\!]_{o^\sharp} \\
&\implies o^! \cup \{v_1\} \subseteq \beta_o[\rho'](o) \subseteq o^? \cup \{v_1\} && \text{Proposition 2} \\
&\implies o\ R_d[\rho']\ \left\langle o^! \cup \{v\}, o^? \cup \{v\} \right\rangle && \text{definition of } R_d[\rho] \\
&\implies o\ R_d[\rho']\ [\![s]\!]_{d^\sharp}(\left\langle o^!, o^? \right\rangle) && \text{definition of } [\![s]\!]_{d^\sharp}
\end{aligned}
$$

Subcase $s = v_1 \leftarrow v_2 \wedge v_2 \notin o^! \wedge v_2 \notin o^?$:

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

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

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,

$$\begin{aligned}
o\ R_d[\rho]\ \langle o^!, o^? \rangle &\implies o^! \subseteq \beta_o[\rho](o) \subseteq o^? \\
&\implies o^! \setminus \{v\} \subseteq \beta_o[\rho](o) \setminus \{v\} \subseteq \beta_o[\rho](o) \cup \{v\} \subseteq o^? \cup \{v\} \\
&\implies o^! \setminus \{v\} \subseteq \beta_o[\rho'](o) \subseteq o^? \cup \{v\} \\
&\implies o\ R_d[\rho']\ \langle o^! \setminus \{v\}, o^? \cup \{v\} \rangle && \text{definition of } R_d[\rho] \\
&\implies o\ R_d[\rho']\ [\![s]\!]_{d^\sharp}(\langle o^!, o^? \rangle) && \text{definition of } [\![s]\!]_{d^\sharp}
\end{aligned}$$

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

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

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

$$\begin{aligned}
o\ R_d[\rho]\ \langle o^!, o^? \rangle &\implies o^! \subseteq \beta_o[\rho](o) \subseteq o^? \\
&\implies o^! \subseteq [\![s]\!]_{o^\sharp}(\beta_o[\rho](o)) \subseteq o^? && \text{definition of } [\![s]\!]_{o^\sharp} \\
&\implies o^! \subseteq \beta_o[\rho'](o) \subseteq o^? && \text{from Proposition 2} \\
&\implies o\ R_d[\rho']\ \langle o^!, o^? \rangle && \text{definition of } R_d[\rho] \\
&\implies o\ R_d[\rho']\ [\![s]\!]_{d^\sharp}(\langle o^!, o^? \rangle) && \text{definition of } [\![s]\!]_{d^\sharp}
\end{aligned}$$

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

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

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

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

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

Subcase $s = v_1 \leftarrow v_2 \wedge v_2 \notin \overline{V^\sharp}$: From the definition of $[\![s]\!]_{o^\sharp}$, it follows that $[\![v_1 \leftarrow v_2]\!]_{o^\sharp}(o^\sharp) \supseteq o^\sharp \setminus \{v_1\}$.

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

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

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

Subcase $s = v \leftarrow e$:

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

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

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

□

$$
\begin{aligned}
\mathit{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^? \\
\mathit{diff}(\langle o_1^!, o_1^? \rangle, \langle o_2^!, o_2^? \rangle) &\triangleq o_1^! \cap o_2^! = \emptyset \\
\mathit{compatible}(o_1^{!?}, o_2^{!?}) &\triangleq \mathit{same}(o_1^{!?}, o_2^{!?}) \vee \mathit{diff}(o_1^{!?}, o_2^{!?}) \\
\mathit{setcompat}(O^{!?}) &\triangleq \forall o_1^{!?}, o_2^{!?} \in O^{!?}.\mathit{compatible}(o_1^{!?}, o_2^{!?}) \\
\mathit{relevant}(O^\sharp, V) &\triangleq V \subseteq \cup_{o^\sharp \in O^\sharp} o^\sharp \wedge \forall o^\sharp \in O^\sharp.o^\sharp \cap V \neq \emptyset \\
\mathit{envs}(\rho^\sharp, O^{!?}, V) &\triangleq \left\{ O^\sharp \subseteq \rho^\sharp : \mathit{relevant}(O^\sharp, V) \wedge \mathit{setcompat}\left(\left\{\langle o^\sharp, o^\sharp \rangle : o^\sharp \in O^\sharp\right\} \cup O^{!?}\right)\right\} \\
\mathit{objs}\left(m^\sharp\right) &\triangleq \{\langle o^!, o^? \rangle \in \text{range}(m^\sharp)\} \\
n(O^\sharp, v) &\triangleq o^\sharp \in O^\sharp : v \in o^\sharp \\
e_0^{+\sharp}(b, O^\sharp) &\triangleq \lambda f. \begin{cases} \langle n(O^\sharp, b(f)), (O^\sharp, b(f)) \rangle & \text{if } f \in \text{dom}(b) \\ \top & \text{otherwise} \end{cases} \\
e^{+\sharp}[a, b, O^\sharp](q, m^\sharp) &\triangleq \left\{ \left\langle q', m^\sharp \sqcap e_0^{+\sharp}(b, O^\sharp) \right\rangle : \delta(q, a, q') \right\} \\
e_0^{-\sharp}(b, O^\sharp, f) &\triangleq \lambda f'. \begin{cases} \overline{n(O^\sharp, b(f))} & \text{if } f \triangleq f' \\ \top & \text{otherwise} \end{cases} \\
e^{-\sharp}[b, O^\sharp](q, m^\sharp) &\triangleq \left\{ \left\langle q, m^\sharp \sqcap e_0^{-\sharp}(b, O^\sharp, f) \right\rangle : f \in \text{dom}(b) \right\} \\
e^\sharp[\{\langle a, b \rangle\}, O^\sharp](q, m^\sharp) &\triangleq e^{+\sharp}[a, b, O^\sharp](q, m^\sharp) \cup e^{-\sharp}[b, O^\sharp](q, m^\sharp) \\
e^\sharp[\{\langle a_1, b_1 \rangle \cdots \langle a_n, b_n \rangle\}, O^\sharp](q, m^\sharp) &\triangleq \left(\left(\bigcup_{1 \leq i \leq n} e_+^\sharp[a_i, b_i, O^\sharp](q, m^\sharp) \right) \cup e_-^\sharp[b_1, O^\sharp](\cdots e_-^\sharp[b_n, O^\sharp](q, m^\sharp) \cdots) \right) \\
[\![\mathbf{tr}(T)]\!]_{m^\sharp}[\rho^\sharp](q, m^\sharp) &\triangleq \bigcup_{O^\sharp \in \mathit{envs}\left(\rho^\sharp, \mathit{objs}(m^\sharp), \bigcup_{\langle a, b \rangle \in T} \text{range}(b)\right)} e^\sharp[T, O^\sharp](q, m^\sharp) \\
[\![s]\!]_{\sigma^\sharp}[\rho^\sharp](\sigma^\sharp) &\triangleq \bigcup_{\langle q, m^\sharp \rangle \in \sigma^\sharp \cup \{\langle q_0, \lambda f. \top \rangle\}} [\![s]\!]_{m^\sharp}[\rho^\sharp](q, m^\sharp)
\end{aligned}
$$

Figure 4: Transfer function for the tracematch state abstraction for $s = \mathbf{tr}(T)$

The transfer function for transition statements is defined in Figure 4. At a high level, it mirrors the semantics of $\mathbf{tr}(T)$ presented in Section 2, but several helper functions are needed to abstract the semantics.

The main complication is variable lookup in the abstract environment. In the concrete semantics, a transition element identifies the objects pointed to by specified variables, and binds them in the tracematch state. Thus, the abstract transfer function must identify the abstract objects pointed to by the specified variables. However, the abstract object state $\rho^\sharp$ may contain multiple abstract objects pointed to by the same variable. For example, $\rho^\sharp$ 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, these two abstract objects are not *compatible*: at any instant at run time, it is not possible for two concrete objects with these two abstractions to coexist, since x cannot possibly point to both of them simultaneously (unless they are the same object, in which case y cannot both point and not point to it). The notion of compatibility is treated in detail by Sagiv et al. [26]. Every compatible subset of $O^\sharp \subseteq \rho^\sharp$ is guaranteed to contain no more than one abstract object pointed to by any variable. In addition, we would like $O^\sharp$ to be compatible with objects already bound in the tracematch, which are represented not precisely but as bindings with must and may sets. Thus, our definition of *compatible* extends Sagiv et al.'s definition to abstract objects represented by must and may sets. Furthermore, we want $O^\sharp$ to contain all the variables being bound by the transition statements, but we do not require it to contain irrelevant abstract objects not pointed to by these variables. The predicate *relevant* enforces this requirement. The set *envs* contains all compatible and relevant subsets of $\rho^\sharp$. Given such a subset $O^\sharp$, the helper function $n(O^\sharp, v)$ finds the unique abstract object pointed to by v.

Having defined abstract variable lookup, the abstract tracematch transition functions $e_0^{+\sharp}, e_0^{-\sharp}, e^{+\sharp}, e^{-\sharp}, e^\sharp$ are exactly like their concrete counterparts, but with abstract lookup $n(O^\sharp, v)$ substituted for concrete lookup in ρ. The overall transfer function $[\![\mathbf{tr}(T)]\!]_{m^\sharp}$ joins the results of $e^\sharp$ for all compatible and relevant abstract environments $O^\sharp \subseteq \rho^\sharp$. Finally, $[\![s]\!]_{\sigma^\sharp}$ extends $[\![s]\!]_{m^\sharp}$ to sets of abstract tracematch state pairs; it is the same as in Figure 3.

At control flow merge points, the join operator used on $\mathbf{State}^\sharp$ is set union.

We have proven that the transfer function $[\![s]\!]_{\sigma^\sharp}$ preserves the correctness relation:

Theorem 3. *If* $\langle s, \rho, h, \sigma\rangle \to \langle \rho', h', \sigma'\rangle$ *and* $\sigma\ R_\sigma[\rho]\ \sigma^\sharp$*, then* $\sigma'\ R_\sigma[\rho']\ [\![s]\!]_{\sigma^\sharp}[\rho^\sharp](\sigma^\sharp)$.

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* $\mathbf{tr}(T)$*, if* $\langle s, \rho, h, \sigma\rangle \to \langle \rho', h', \sigma'\rangle$ *and* $\sigma\ R_\sigma[\rho]\ \sigma^\sharp$*, then* $\sigma'\ R_\sigma[\rho']\ [\![s]\!]_{\sigma^\sharp}[\rho^\sharp](\sigma^\sharp)$.

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

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

□

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

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

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

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 $\mathit{setcompat}(\{o_i^{!?}\})$.

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 $\mathit{same}(o_1^{!?}, o_2^{!?})$.

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 4. *Given $\rho \in \mathcal{P}(\textbf{\textit{Var}}), V \subseteq \textbf{\textit{Var}}$ such that $\rho(v) \neq \bot$ for any $v \in V$, define $O^\sharp(\rho, V) \triangleq \{\beta_o[\rho](\rho(v)) : v \in V\}$.*

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

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

Proof. 1.

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

2.

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

Every $o^\sharp \in O^\sharp(\rho, V)$ 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 $v \in V$, $O^\sharp(\rho, V)$ contains $\beta_o[\rho](\rho(v))$. Also, $v \in \beta_o[\rho](\rho(v))$. Therefore, $\beta_o[\rho](\rho(v))$ satisfies the definition of $n(O^\sharp(\rho, V), v)$. Furthermore, $\beta_o[\rho](\rho(v))$ is the only such element of $O^\sharp(\rho, V)$, 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 range$(b) \subseteq V$. Then*

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

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

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

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

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

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

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

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

For $f \notin \mathrm{dom}(b)$, $e_-(b,\rho,f) = \top\ R_d[\rho]\ \top = e^\sharp_-(O^\sharp(\rho,V),b,f)$.

□

Lemma 15. *If $\langle \textbf{\textit{tr}}(T),\rho,h,\sigma\rangle \to \langle \rho',h',\sigma'\rangle$ and $\sigma\ R_\sigma[\rho]\ \sigma^\sharp$, then $\sigma'\ R_\sigma[\rho']\ [\![\textbf{\textit{tr}}(T)]\!][\rho^\sharp](\sigma^\sharp)$ for any $\rho^\sharp \sqsupseteq \beta_\rho(\rho,h)$.*

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

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

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

Therefore, for each

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

there exists

$$\langle q,m^\sharp\rangle \in \bigcup_{O^\sharp \in \textit{envs}(\rho^\sharp,\textit{objs}(m^\sharp),\mathrm{range}(b))} e^\sharp_+[a,b,O^\sharp](\sigma^\sharp \cup \{\langle q_0,\lambda f.\top\rangle\}) \cup e^\sharp_-[b,O^\sharp](\sigma^\sharp \cup \{\langle q_0,\lambda f.\top\rangle\}) = [\![\mathbf{tr}(\{\langle a,b\rangle\})]\!]_{\sigma^\sharp}[\rho^\sharp](\sigma^\sharp)$$

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

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

4 Context-Sensitive Interprocedural Analysis

We have implemented the analysis as an instance of the IFDS algorithm of Reps et al. [24] with small modifications which we explain in this section. The IFDS algorithm implements a fully context-sensitive interprocedural dataflow analysis provided that:

- the analysis domain is the powerset of a finite set **Dom**,
- the merge operator is set union, and
- the flow 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 algorithm tabulates two sets of dataflow functions for control-flow paths of increasing length. The first table, PathEdge, tabulates a flow function from the start node of each procedure to every other node in the same procedure. The second table, SummaryEdge, tabulates a flow function for each call site in the program.

The overall flow function $\lambda\rho^\sharp, h^\sharp, \sigma^\sharp.\left\langle [\![s]\!]_{\rho h^\sharp}(\rho^\sharp, h^\sharp), [\![s]\!]_{\sigma^\sharp}[\rho^\sharp](\sigma^\sharp)\right\rangle$ is not distributive. However, it can be separated into two analyses each satisfying the requirements of the IFDS algorithm. The first analysis computes the value abstraction, and the second analysis computes the tracematch abstraction.

The IFDS algorithm requires the transfer function $[\![s]\!] : \mathcal{P}(\mathbf{Dom}) \to \mathcal{P}(\mathbf{Dom})$ to be decomposed pointwise into a function acting separately on each element of $\mathbf{Dom} \cup \{0\}$, where $0 \notin \mathbf{Dom}$ is a special sentinel value. The pointwise function $[\![s]\!]_\bullet : \mathbf{Dom} \cup \{0\} \to \mathcal{P}(\mathbf{Dom})$ uniquely defines the overall transfer function as follows: $[\![s]\!](D) \triangleq \bigcup_{d\in D\cup\{0\}} [\![s]\!]_\bullet(d)$. A transfer function can be decomposed in this form if and only if it is distributive [24]. To implement the tracematch analysis in the IFDS framework, we show how to express the transfer functions $[\![s]\!]_{\rho h^\sharp}$ and $[\![s]\!]_{\sigma^\sharp}$ in this form.

4.1 Value Abstraction

For computing the value abstraction $\left\langle \rho^\sharp, h^\sharp\right\rangle$, we define the set **Dom** as two disjoint copies of $\mathbf{Obj}^\sharp = \mathcal{P}(\mathbf{Var})$, one copy for $\rho^\sharp$ and the other for $h^\sharp$. To distinguish elements of the two sets, we use the notation $\rho[o^\sharp]$ to mean $o^\sharp$ from the $\rho^\sharp$ copy of $\mathbf{Obj}^\sharp$, and $h[o^\sharp]$ to mean $o^\sharp$ from the $h^\sharp$ copy. Thus, a given pair $\left\langle \rho^\sharp, h^\sharp\right\rangle$ is represented using the set $\mathit{decomp}(\rho^\sharp, h^\sharp) \triangleq \{\rho[o^\sharp] : o^\sharp \in \rho^\sharp\} \cup \{h[o^\sharp] : o^\sharp \in h^\sharp\}$. The transfer function for individual elements of $\mathbf{Dom} \cup \{0\}$ is defined as follows:

$$[\![s]\!]_{\rho h^\sharp}(\rho[o^\sharp]) \triangleq \begin{cases} \{\rho[o^\sharp \setminus \{v\}]\} & \text{if } s = v \leftarrow e \\ \{\rho[o^\sharp], h[o^\sharp]\} & \text{if } s = e \leftarrow v \text{ and } v \in o^\sharp \\ \{\rho[o^\sharp]\} & \text{if } s = e \leftarrow v \text{ and } v \notin o^\sharp \\ \{\rho[[\![s]\!]_{o^\sharp}(o^\sharp)]\} & \text{otherwise} \end{cases}$$

$$[\![s]\!]_{\rho h^\sharp}(h[o^\sharp]) \triangleq \begin{cases} \{\rho[o^\sharp \setminus \{v\}], \rho[o^\sharp \cup \{v\}], h[o^\sharp \setminus \{v\}], h[o^\sharp \cup \{v\}]\} & \text{if } s = v \leftarrow e \\ \{h[[\![s]\!]_{o^\sharp}(o^\sharp)]\} & \text{otherwise} \end{cases}$$

$$[\![s]\!]_{\rho h^\sharp}(0) \triangleq \begin{cases} \{\rho[\{v\}]\} & \text{if } s = v \leftarrow \mathbf{new} \\ \emptyset & \text{otherwise} \end{cases}$$

The following proposition guarantees that when these pointwise transfer functions are composed, the result is isomorphic to the transfer function $[\![s]\!]_{\rho h^\sharp}$ from Section 3.

Proposition 6.

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

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

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

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

Therefore,

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

Also,

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

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

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

Therefore,

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

Also,

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

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

$$\begin{aligned}
[\![s]\!]_{\rho h^\sharp}(\rho[o^\sharp]) =& \{\rho[[\![s]\!]_{o^\sharp}(o^\sharp)]\} \\
[\![s]\!]_{\rho h^\sharp}(h[o^\sharp]) =& \{h[[\![s]\!]_{o^\sharp}(o^\sharp)]\} \\
[\![s]\!]_{\rho h^\sharp}(0) =& \{\rho[\{v\}]\}
\end{aligned}$$

Therefore,

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

Also,

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

Case s is any other statement: In this case,

$$\begin{aligned}
[\![s]\!]_{\rho h^\sharp}(\rho[o^\sharp]) =& \{\rho[[\![s]\!]_{o^\sharp}(o^\sharp)]\} \\
[\![s]\!]_{\rho h^\sharp}(h[o^\sharp]) =& \{h[[\![s]\!]_{o^\sharp}(o^\sharp)]\} \\
[\![s]\!]_{\rho h^\sharp}(0) =& \emptyset
\end{aligned}$$

Therefore,

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

Also,

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

□

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 **Dom** $\cup \{0\}$. The call flow function 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:

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

To define the flow out of procedure calls, 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 $\mathit{return} \circ [\![p]\!] \circ \mathit{call}$, where *call* is the call flow function, $[\![p]\!]$ 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 $[\![p]\!] \circ \mathit{call}$, directly constructs the SummaryEdge flow function. A similar modification is also used in the typestate analysis of Fink et al. [15]. 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 $\mathit{summ}_\bullet$ takes two arguments $d, d' \in \mathbf{Dom} \cup \{0\}$. The overall summary function is defined as:

$$\mathit{summ}(D) \triangleq \bigcup_{d \in D \cup \{0\}} \bigcup_{d' \in ([\![p]\!] \circ \mathit{call})(d) \cup \{0\}} \mathit{summ}_\bullet(d, d')$$

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 $\mathit{summ}_\bullet(d, d')$ 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 value 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 $o_c^\sharp$ in the caller before the call is being returned from the callee (i.e. $v_s \in o_r^\sharp$), then v_t is added to $o_c^\sharp$. If some other object is being returned, then v_t is removed from $o_c^\sharp$, 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 $o_c^\sharp$, since no variables of the caller pointed to the object before the call.

$$\mathit{rv}(o_c^\sharp, o_r^\sharp) \triangleq \begin{cases} o_c^\sharp & \text{if } p \text{ does not return a value} \\ o_c^\sharp \cup \{v_t\} & v_s \in o_r^\sharp \\ o_c^\sharp \setminus \{v_t\} & v_s \notin o_r^\sharp \end{cases}$$

$$\mathit{summ}_{\rho h^\sharp}\left(c_h^\rho[o_c^\sharp], r_h^\rho[o_r^\sharp]\right) \triangleq \left\{r_h^\rho[\mathit{rv}(o_c^\sharp, o_r^\sharp)]\right\} \text{ where each of } c_h^\rho, r_h^\rho \text{ is either } \rho \text{ or } h$$

$$\mathit{summ}_{\rho h^\sharp}\left(0, r_h^\rho[o_r^\sharp]\right) \triangleq \left\{r_h^\rho[\mathit{rv}(\emptyset, o_r^\sharp)]\right\}$$

4.2 Tracematch State Abstraction

The analysis for computing the tracematch abstraction operates on the set of possible tracematch state pairs $\mathbf{Dom} \triangleq Q \times (F \to \mathbf{Bind}^\sharp)$. The analysis uses the value abstraction $\rho^\sharp$ computed in an earlier pass. The tracematch transfer function from Section 3 is already in the decomposed form required by the IFDS algorithm:

$$[\![s]\!]_{\sigma^\sharp}[\rho^\sharp](\sigma^\sharp) \triangleq \bigcup_{\langle q, m^\sharp\rangle \in \sigma^\sharp \cup \{0\}} [\![s]\!][\rho^\sharp]_{m^\sharp}(q, m^\sharp) \qquad \text{where } [\![s]\!][\rho^\sharp]_{m^\sharp}(0) \triangleq [\![s]\!][\rho^\sharp]_{m^\sharp}(q_0, \lambda f.\top)$$

The call flow function for tracematch state is straightforward. It applies the $\mathit{update}[r]$ function that was defined for the value abstraction to each must, may, and negative binding set. Arguments are replaced by parameters, and non-arguments are removed.

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

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

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

4.3 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. When the analysis is used to optimize a dynamic tracematch implementation, all instrumentation can be removed in this case. However, the analysis may not be successful in ruling out *all* body statements. In this case, it is useful to compile a list of all transition statements that may contribute to a match at each body statement. Such a list is useful both for static verification and for optimizing a dynamic implementation. In static verification, this list helps the user identify the source of the bug, or to decide that the error report is a false positive. For example, if a collection is updated during iteration, the body statement is the failing `next` call on the iterator; more useful to the programmer would be the location of the collection update. In optimizing the dynamic tracematch implementation, all transition statements not leading to a potentially matching body statement can be removed, thereby reducing the runtime overhead of matching.

The analysis can be extended to keep track of relevant transition statements by replacing the IFDS algorithm with the IDE algorithm [25]. The IDE algorithm is an extension of the IFDS algorithm to analysis domains of the form $\mathbf{Dom} \to L$, where **Dom** satisfies the same conditions as for the IFDS algorithm and L is a lattice of finite height. Indeed, the IFDS algorithm is a special case of the IDE algorithm with L chosen as the two-point lattice $\bot \sqsubseteq \top$. The IFDS version of the tracematch analysis presented thus far determines only whether a given pair $\left\langle q, m^\sharp\right\rangle$ is $(\top)$ or is not $(\bot)$ present at each program point. To keep track of transition statements leading to a match, we keep the same set $\mathbf{Dom} = Q \times (F \to \mathbf{Bind}^\sharp)$, and define $L \triangleq \{\bot\} \uplus \mathcal{P}(\mathbf{Tr})$, where **Tr** is the set of all transition statements in the program. For each pair $\left\langle q, m^\sharp\right\rangle$ present at a program point, the IDE version of the analysis maintains the set of transition statements that may have contributed to its presence. The partial order on L is $l_1 \sqsubseteq l_2 \iff l_1 = \bot \vee l_1 \subseteq l_2$.

Like the IFDS algorithm, the IDE algorithm uses a decomposed transfer function. In the IDE algorithm, the pointwise transfer function has the form $[\![s]\!]_\bullet : (\mathbf{Dom} \cup \{0\}) \to \mathbf{Dom} \to L \to L$. Given a pair of elements d, d' from **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 $[\![s]\!] : (\mathbf{Dom} \to L) \to (\mathbf{Dom} \to L)$ as $[\![s]\!](f) \triangleq \lambda d'. \bigsqcup_{d \in \mathbf{Dom} \cup \{0\}} [\![s]\!]_\bullet(d)(d')(f(d))$.

The pointwise transfer function $[\![s]\!]_{m^\sharp}$ 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 $d' \in [\![s]\!]_{m^\sharp}(d)$ and the bottom function $\lambda l.\bot$ otherwise:

$$[\![s]\!]_{\sigma^{\sharp\{\}}}[\rho^\sharp](q, m^\sharp)(q', m'^\sharp) \triangleq \begin{cases} \lambda l.l & \text{if } \langle q', m'^\sharp \rangle \in [\![s]\!]_{m^\sharp}[\rho^\sharp](q, m^\sharp) \\ \lambda l.\bot & \text{otherwise} \end{cases}$$

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 $\langle q', m'^\sharp \rangle$.

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

In the second case above, when $\langle q, m^\sharp \rangle = \langle q', m'^\sharp \rangle$, 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* $\langle q_2, m_2 \rangle \in e^\sharp[T, \rho](q_1, m_1)$*;* $\langle q_1, m_1 \rangle \neq \langle q_2, m_2 \rangle$*;* $\rho^\sharp$ *overapproximates* ρ*; and* $\langle q_1, m_1 \rangle \ R_m[\rho] \ \langle q_1^\sharp, m_1^\sharp \rangle$*; then there exists* $\langle q_2^\sharp, m_2^\sharp \rangle \in [\![\mathbf{tr}(T)]\!]_{m^\sharp}[\rho^\sharp](q_1^\sharp, m_1^\sharp)$ *such that* $\langle q_1^\sharp, m_1^\sharp \rangle \neq \langle q_2^\sharp, m_2^\sharp \rangle$ *and* $\langle q_2, m_2 \rangle \ R_m[\rho] \ \langle q_2^\sharp, m_2^\sharp \rangle$.

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

Case $m_2(f) = m_1(f) \sqcap \rho(b(f))$ **and** $m_1^\sharp(f)$ **is a positive binding containing** $\rho(b(f))$ **in its must set:** In this case, since $m_1(f) \ R_d[\rho] \ m_1^\sharp(f)$, $m_1(f) = \rho(b(f))$, so $m_2(f) = m_1(f)$, a contradiction. Therefore this case cannot occur.

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

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

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

Case $m_2(f) = m_1(f) \sqcap \overline{\{\rho(b(f))\}}$ **and** $m_1^\sharp(f)$ **is a positive binding containing** $b(f)$ **in its may set:** In this case, $m_1^\sharp(f) \sqcap e_0^{+\sharp}(b, O^\sharp(\rho, \text{range}(b))) = m_1^\sharp(f) \sqcap \overline{n(O^\sharp(\rho, \text{range}(b)), b(f))}$, which is either $\bot$ 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 $[\![\mathbf{tr}(T)]\!]_{m^\sharp}[\rho^\sharp](q_1^\sharp, m_1^\sharp)$.

Case $m_2(f) = m_1(f) \sqcap \overline{\{\rho(b(f))\}}$ **and** $m_1^\sharp(f)$ **is a positive binding not containing** $b(f)$ **in its may set:** In this case, since $m_1(f)\ R_d[\rho]\ m_1^\sharp(f)$, $m_1(f)$ is a positive binding other than $\rho(b(f))$, so $m_2(f) = m_1(f)$, a contradiction. Therefore this case cannot occur.

Case $m_2(f) = m_1(f) \sqcap \overline{\{\rho(b(f))\}}$ **and** $m_1^\sharp(f)$ **is a negative binding containing** $b(f)$**:** In this case, since $m_1(f)\ R_d[\rho]\ m_1^\sharp(f)$, $m_1(f)$ is either a positive binding other than $\rho(b(f))$ or a negative binding containing $\rho(b(f))$. Either way, $m_2(f) = m_1(f)$, a contradiction. Therefore this case cannot occur.

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

□

It may happen that a transition statement in a loop changes the tracematch state in the first iteration but not in any subsequent iteration. When optimizing the dynamic tracematch implementation, it is desirable to 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 evaluated the precision of the analysis on the tracematches from Bodden et al. [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, all from the DaCapo benchmark suite, version 2006-10-MR2 [7]. Most of the benchmarks and the Java standard library use reflection. We instrumented the above benchmarks using ProBe [22] and *J [14] 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 tracematch/benchmark pairs contained a total of 5409 final transition statements, where final is defined as the transition symbol being the last symbol of some word in the language specified by the tracematch pattern. We count only 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.

36 of the 54 tracematch/benchmark pairs actually used the features described by the tracematch. These are represented by the 36 circles in Table 1. We define using the feature as containing enough tracematch symbols (i.e. operations) that it is possible to construct a word in the tracematch pattern P. Let $A' \subseteq A$ contain every tracematch symbol (i.e. operations) present in at least one transition statement in the benchmark; the feature is used if the language $A'^* \cap P$ is non-empty. This is the same as passing the quickcheck stage of Bodden et al. [8].

Of the 36 remaining pairs, the analysis proved that the tracematch cannot match at all in 15. These are represented by the white circles in Table 1. For the remaining 21 pairs, the white part of the circle is the fraction of final transition statements that were proved not to complete a match; the black part represents those that could complete a match.

4 of the 21 remaining pairs involve the HasNext and HasNextElem tracematches. In 1 pair (HasNext/pmd), all detected 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 pairs (in jython and in HasNext/bloat). In addition, the other pairs contain false positives due to iterators being stored only in fields and not local variables.

In 11 pairs 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. In many cases, the body of a loop iterating over a collection contains calls leading to very deep call

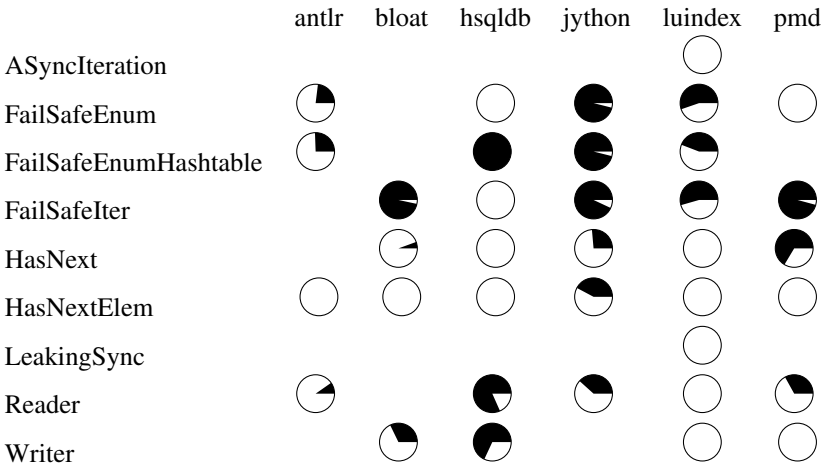

Table 1: Fraction of transition statements that may complete a match

chains comprising many methods, some of which update collections. The analysis is not able to prove that all these collections are distinct from the collection being iterated. May-point-to information may help rule out some of these cases. Also, 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. One of our future goals is to create a convenient user interface to help the programmer visualize the potential update locations and the call chains connecting them to the original loop.

The analysis found some false positives in 6 pairs involving the Reader and Writer tracematches. In every such case, the benchmark contained a loop repeatedly calling a helper method that used the reader or writer. Both the loop and the helper method contained a try block. An exception during the input/output operation would be caught in the helper, which would close the stream and re-throw the exception. The try block protecting the loop would catch the exception, thereby terminating the loop. Because the analysis does not distinguish between normal and exceptional returns, it conservatively assumed that the loop could continue iterating and therefore use the reader or writer after the stream was closed.

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. Earlier, Bodden et al. [8] implemented a static analysis based on flow-insensitive may-point-to information. The analysis comprises three stages. In the first stage (called quickcheck), only the set of tracematch symbols (i.e. operations) occurring in the program is examined; if it is impossible to make any word in the language specified by the tracematch pattern with only those symbols, the tracematch cannot match. The second stage considers the may-point-to sets of the variables bound at each transition statement. If a set of transitions is to lead to a match, they must have consistent bindings, and this is only possible if their points-to sets overlap. This stage was observed to reduce the number of instrumentation points (but not eliminate all of them) on five tracematch/benchmark pairs. The third stage attempts to track the tracematch state flow-sensitively. However, because it does not use must-aliasing information and does not track the flow of individual objects, it did not lead to any improvement over the second stage. In particular, it is ineffective for a tracematch requiring flow sensitivity such as HasNext, because it cannot be sure that the object on which a `hasNext` call occurs is the same object on which `next` will later be called. Our analysis is complementary to the use of may-point-to information in Bodden et al.'s second stage: while our analysis is very precise when local variable references to the bound objects are involved and when flow-sensitive information is required, it is less effective when the objects escape to the heap and local references are lost, such as in some cases of the FailSafe* tracematches. For these cases, it would help to extend our analysis to use may-point-to information.

Another similar analysis is Guyer and Lin's [18, 19] client-driven pointer analysis. Their analysis is based on an subset-based may-point-to analysis followed by flow-sensitive propagation of states on the abstract object represented

by each allocation site. However, 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 [15]. 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 at al.'s; it maintains reference counts from individual fields rather than full access paths. This object abstraction could also be substituted in the tracematch analysis.

The Metal system [21] is an unsound state-based bug finder for C. The core system does not worry about 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 recover some of this unsoundness. Despite being unsound, Metal was successful in finding many locking bugs in the Linux kernel.

An alternative to analyzing code with arbitrary aliasing is to require the code to conform to a specialized type system that restricts aliasing in ways that make it easier to prove correct. 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 if may be possible to infer annotations automatically. The Vault system [12] system uses *keys*, unique pointers to objects. The type system prevents duplication of keys, and each typestate change is correlated with a set of keys held at the point of the change. The same authors propose a system for specifying typestates 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 [16] is another system similar to but simpler than Vault. Bierhoff and Aldrich [5, 6] present a type system in which both aliasing and typestate 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 typestate 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 [24] and IDE [25] 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 experiment with these improvements in the future.

To make the analysis useful to programmers, as well as to ease our work of interpreting the analysis results, we plan to complement the analysis with a suitable user interface for presenting the analysis results and navigating the program and its call graph and control flow graph.

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. In *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. In *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. In *Proceedings of POPL '07*, pages 11–23, 2007.

[4] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *Proceedings of OOPSLA '07*, pages 589–608, 2007.

[5] K. Bierhoff and J. Aldrich. Lightweight object specification with typestates. In *Proceedings of ESEC/FSE-13*, pages 217–226, 2005.

[6] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *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. In *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. In *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. In *Proceedings of OOPSLA '07*, pages 569–588, 2007.

[10] S. Cherem and R. Rugina. Compile-time deallocation of individual objects. In *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. In *Proceedings of PLDI '01*, pages 59–69, 2001.

[13] R. DeLine and M. Fähndrich. Typestates for objects. In *Proceedings of ECOOP 2004*, volume 3086 of *LNCS*, pages 465–490, 2004.

[14] B. Dufour. Objective quantification of program behaviour using dynamic metrics. Master's thesis, McGill University, June 2004.

[15] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *Proceedings of ISSTA'06*, pages 133–144, 2006.

[16] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of PLDI '02*, pages 1–12, 2002.

[17] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Proceedings of OOPSLA '05*, pages 385–402, 2005.

[18] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *Proceedings of SAS '03*, volume 2694 of *LNCS*, pages 214–236, 2003.

[19] S. Z. Guyer and C. Lin. Error checking with client-driven pointer analysis. *Sci. Comput. Program.*, 58(1-2):83–114, 2005.

[20] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Proceedings of POPL '05*, pages 310–323, 2005.

[21] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of PLDI '02*, pages 69–82, 2002.

[22] O. Lhoták. Comparing call graphs. In *Proceedings of PASTE '07*, pages 37–42, 2007.

[23] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of OOPSLA '05*, pages 365–383, 2005.

[24] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of POPL '95*, pages 49–61, 1995.

[25] 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.

[26] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50, Jan. 1998.

[27] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.