Scalable Parametric Static Analysis

Xin Zhang
Georgia Institute of Technology
xin.zhang@gatech.edu

Mayur Naik
Georgia Institute of Technology
naik@cc.gatech.edu

Hongseok Yang
University of Oxford
hongseok.yang@cs.ox.ac.uk

Abstract

Parametric static analysis allows choosing a parameter value to balance the precision and cost of the instantiated analysis. We propose an efficient approach to either find a cheapest parameter value to prove a given query or show that no such parameter value exists. Our approach is based on refinement, as in CEGAR (counterexample-guided abstraction refinement), but applies a novel meta-analysis to abstract counterexample traces to efficiently find parameter values that are incapable of proving the query. We formalize our approach in a generic framework and apply it to two parametric analyses: a thread-escape analysis and a type-state analysis. The thread-escape analysis is implemented and applied to eight Java benchmarks comprising 2.5 MLOC. Our experiments show that our approach is effective in practice: for our four largest benchmarks, searching 2^{9K} parameter values for each of 10K queries on average per benchmark, it finds a cheapest one for proving 46% queries and shows that none exists for 37% queries, in one minute per query on average.

1. Introduction

A key problem in static analysis concerns how to balance its precision and cost. Ideally, the analysis should compute an abstraction of a given program that discards all the unnecessary details for proving a given *query*, e.g., an assertion at a program point. Query-oblivious analyses use a fixed abstraction independent of the query. Query-guided analyses, on the other hand, exploit any gain in precision and reduction in cost by tailoring the abstraction to the query.

This paper presents a new approach to build efficient query-guided static analyses. Our approach assumes a parametric static analysis that takes as inputs a program s, a query q, and a parameter value $\pi \in \mathbb{P}$, and either proves the query or produces an *abstract counterexample trace* as a witness of its failure to prove the query. The set $\mathbb{P}$ of all parameter values is finite and preordered by the cost of the instantiated analysis: $\pi \preceq \pi'$ means performing the analysis using π is at least as cheap as performing it using π'. Our goal is then to solve the *parametric static analysis problem* which asks to efficiently compute one of the following:

1. **A Minimum-Cost Proof:** A π such that (1) the analysis proves the query using π and (2) whenever the analysis does the same using a different parameter value π', we have $\pi \preceq \pi'$.
2. **An Impossibility Result:** There does not exist any π that enables the analysis to prove the query.

Many static analyses can be naturally parameterized. For instance, software model checkers such as SLAM are usually parametric in program predicates to be used in the *predicate abstraction* [6], and shape analyses are parametric in *instrumentation predicates* that dictate how to abstract concrete heap cells to summary nodes [17]. Cloning-based pointer analyses are parametric in the degree of cloning $\pi(h) \geq 0$ to use for each call site and each object allocation site h in the program [11]. In all these cases, the parameter dictates the cost of the instantiated analysis to a first approximation. For instance, in cloning-based pointer analysis, it is reasonable to say $\pi \preceq \pi'$ holds if $\Sigma_h \pi(h) \leq \Sigma_h \pi'(h)$.

A naive solution to our parametric static analysis problem is to run the analysis using each parameter value in $\mathbb{P}$, from the cheapest to the most expensive, until we obtain a parameter value that proves the query, failing which we get an impossibility result. However, $\mathbb{P}$ is very large in practice, typically exponential in the size of the program. For instance, with k degrees of cloning and a program with N sites, we have $|\mathbb{P}| = (k+1)^N$.

This paper presents a new approach to solve the parametric static analysis problem. Our approach is based on refinement as in CEGAR (counterexample-guided abstraction refinement) but differs radically in how it analyzes an abstract counterexample trace: it computes a sufficient condition for the failure of the analysis to prove the query along the trace. The condition represents a set of parameter values $\Pi \subseteq \mathbb{P}$ such that the analysis instantiated using any $\pi \in \Pi$ is guaranteed to fail to prove the query. Our approach then discards all parameter values in Π as unviable and picks the next cheapest parameter value from $\mathbb{P} \setminus \Pi$ as dictated by $\preceq$. If the set of viable parameter values becomes empty then our approach declares impossibility; otherwise, it returns the parameter value that was used to instantiate the analysis in the final iteration as the cheapest one that proves the query.

Our approach finds unviable parameter values by performing a backward analysis on the trace. It is a *meta-analysis* that must be proven sound with respect to the abstract semantics of the forward analysis. Scalability of backward analyses is typically hindered by exploring program states that are unreachable from the initial state. Our backward meta-analysis, in contrast, is guided by the trace provided by the forward analysis. Moreover, this trace enables our backward analysis to perform underapproximation while still guaranteeing to find a non-empty set of unviable parameter values.

Like the forward analysis, the backward meta-analysis is a static analysis, and the performance of our overall approach depends heavily on how this meta-analysis balances its own precision/cost tradeoff. If it performs underapproximation too aggressively, it analyzes the trace efficiently but finds only the current parameter value unviable, and requires more iterations to converge. On the other hand, if it performs underapproximation too passively, it analyzes the trace inefficiently but finds many more parameter values unviable, and thereby requires fewer iterations to converge.

We present a generic framework for developing an efficient backward meta-analysis, which involves choosing an abstract domain, devising (backward) transfer functions, and proving the soundness of these functions with respect to the forward analysis. Our framework suggests a domain built from a DNF representation of the formulas, and provides several effective optimizations for scaling the backward meta-analysis, including double-negation, simplification by clause subsumption, and beam search. We show the versatility of the framework by applying it to two parametric analyses in the literature: the thread-escape analysis from [13] and a simplified type-state analysis based on [5]. These analyses are

```
while (*) {
    u = new h1;
    v = new h2;
    S;
pc: local(u)?
    u.start();
}
```

$$\text{Choices for } S \;=\; \begin{cases} (1)\ \texttt{skip} \\ (2)\ \texttt{g = u} \\ (3)\ \texttt{v.f = u} \end{cases}$$

Figure 1. Example program.

selected because they use very different heap abstractions and they control these abstractions using different types of parameter values.

We implemented our approach for parametric static analyses of Java programs and evaluated it using our thread-escape analysis as a candidate analysis on a suite of eight Java benchmarks comprising 2.5 MLOC. This analysis is a fully flow- and context-sensitive, top-down, summary-based shape analysis with 2^N possible parameter values for each query, where N is the number of object allocation sites in the program. Our experiments show that our approach is effective in practice: for our four largest benchmarks, each containing on average 9K such sites, 10K queries, and 350 KLOC, it finds a cheapest parameter value for proving 46% queries and shows that none exists for 37% queries, in one minute per query on average.

We summarize the key contributions of this paper:

- We formulate the parametric static analysis problem of choosing a suitable parameter value to prove a given query. The formulation seeks a cheapest parameter value that proves the query or an impossibility result that none exists.
- We present a new refinement-based approach to solve the problem. The central insight of our approach is a novel meta-analysis that operates on abstract counterexample traces to find parameter values that are incapable of proving the query.
- We present a generic framework to design the meta-analysis, along with a DNF representation for the formulas tracked by it, and several effective optimizations for scaling it. We apply the framework to two parametric analyses in the literature.
- We present extensive empirical evaluation showing the efficacy of our approach on a parametric thread-escape analysis for a suite of several large real-world Java benchmarks.

2. Informal Description

We describe our approach informally using the thread-escape analysis. To simplify presentation, here we focus on what the approach does, using later sections to illustrate details of how it works.

Parametric Thread-Escape Analysis. A heap object in a multi-threaded shared-memory program is *thread-local* when it is reachable only from at most a single thread. For instance, consider the program in Figure 1, which repeatedly allocates two heap objects at sites labeled `h1` and `h2`, binds them with local variables `u` and `v`, executes an unspecified code snippet S, and starts a new thread on the object `u` by calling `u.start()` from `java.lang.Thread`. When S is `skip` or `v.f = u`, local variable `u` at program point `pc` points to a thread-local object, because this object is reachable only from the current thread. On the other hand, when S is `g = u` that assigns `u` to a global variable `g` (which corresponds to a static field in Java), the object is not thread-local, because it is reachable from threads besides the current thread via global variable `g`. In this case, we say that the object is thread-escaping. Determining the thread locality of heap objects is useful because it helps various other analyses to reason about concurrent properties accurately or efficiently.

Our thread-escape analysis conservatively answers queries on thread locality. It is a flow-sensitive analysis about the heap and employs a very coarse heap abstraction. In this abstraction, all heap objects are summarized to one of only two summary nodes `L` and `E`. The node `E` includes all thread-escaping objects and possibly some thread-local ones, whereas `L` summarizes only thread-local objects. For instance, the abstract state $[\texttt{u} \mapsto \texttt{L}, \texttt{v} \mapsto \texttt{E}]$ expresses that the local variable `u` definitely points to a thread-local object but `v` may point to a thread-escaping object. Hence, if this is the analysis result at `pc` in Figure 1, we can conclude that object `u` at `pc` is local to the current thread. On the other hand, if the analysis result is $[\texttt{u} \mapsto \texttt{E}, \texttt{v} \mapsto \texttt{E}]$, we cannot make the same conclusion because the result allows `u` to point to a thread-escaping object.

One interesting aspect of our thread-escape analysis is that its abstract semantics is parameterized, which opens the door for tuning the analysis to a given query over a given program. The parameter is a map π from each object allocation site to `L` or `E`, and it determines the abstract semantics of object allocation. If $\pi(h)$ is `L`, all objects allocated at h are summarized by `L`. Otherwise, they are summarized by `E`. For instance, there are four possible parameter values for the program in Figure 1 with $S = \texttt{skip}$:

$$\pi_0 = [\texttt{h1} \mapsto \texttt{E}, \texttt{h2} \mapsto \texttt{E}], \quad \pi_1 = [\texttt{h1} \mapsto \texttt{L}, \texttt{h2} \mapsto \texttt{E}],$$
$$\pi_2 = [\texttt{h1} \mapsto \texttt{E}, \texttt{h2} \mapsto \texttt{L}], \quad \pi_3 = [\texttt{h1} \mapsto \texttt{L}, \texttt{h2} \mapsto \texttt{L}].$$

If we run the analysis with π_0, the heap objects allocated at `h1` and `h2` are summarized by the `E` node, so that we get the abstract state $[\texttt{u} \mapsto \texttt{E}, \texttt{v} \mapsto \texttt{E}]$ at `pc`. This analysis result does not ensure that `u` points to a thread-local object, so we cannot prove the query at `pc`. On the other hand, running the analysis with π_1 leads to a proof that object `u` is thread-local, because it produces the abstract state $[\texttt{u} \mapsto \texttt{L}, \texttt{v} \mapsto \texttt{E}]$ at `pc`.

Parameter Searching by Iterative Refinement. As a reader might have guessed by now, in order to use the thread-escape analysis most effectively, we need a mechanism for selecting a parameter value π appropriate for a given query. Trying all possible π's is not an option, because the space of parameter values is huge: there are 2^N different parameter values for a program with N allocation sites. To make matters more complicated, the cost of running the analysis varies depending on parameter values: although we do not go into the details here, using a parameter value with more `E`-mapped sites generally makes the analysis faster.

The iterative refinement algorithm in this paper provides an effective solution to this parameter selection problem. It repeatedly runs the thread-escape analysis with different parameter values, until it proves the given query or infers that proving the query with the analysis is impossible for any parameter value. In each iteration, if the analysis with a parameter value π fails to prove the query, our algorithm generates a counterexample trace, which is analyzed by our algorithm to find out other parameter values π' that would lead to a similar verification failure.

We illustrate the algorithm using the program in Figure 1. Assume that `skip` is chosen for S in the program, and that we wish to answer query `local(u)?` at `pc`. The algorithm first runs the thread-escape analysis with the cheapest parameter value $\pi_0 = [\texttt{h1} \mapsto \texttt{E}, \texttt{h2} \mapsto \texttt{E}]$. As we noted before, this analysis run leads to the abstract state $d_0 = [\texttt{u} \mapsto \texttt{E}, \texttt{v} \mapsto \texttt{E}]$ at `pc`, and fails to prove the query. At this point, our algorithm generates the counterexample trace:

```
u = new h1 ; v = new h2 ; skip ; local(u)?
```

whose abstract semantics under π_0 produces the abstract state d_0. Then, it analyzes this trace backward, and infers a *sufficient* condition on π's that makes the analysis *fail* to prove the query over the same trace. The inference step here involves a meta-reasoning on the behavior of the thread-escape analysis under various parameter values, so we call it meta-analysis. In this example, the inferred condition on parameter values π is $\pi(\texttt{h1}) = \texttt{E}$. In order to prove the query, we should pick π outside of the above set. Our algorithm chooses a cheapest such parameter value $\pi_1 = [\texttt{h1} \mapsto \texttt{L}, \texttt{h2} \mapsto \texttt{E}]$,

```
      u = create();        create() {
      v = create();            return new h;
      g = v;               }
  pc: local(u)?
```

Figure 2. Example program to illustrate analysis incompleteness.

and runs the analysis again. This time the analysis computes the invariant $[\mathtt{u} \mapsto \mathtt{L}, \mathtt{v} \mapsto \mathtt{E}]$ at pc, and proves the query.

Impossibility. Some queries cannot be proved by our thread-escape analysis no matter which parameter value is used with the analysis. There are two reasons for this impossibility. The first is that a heap object in concern is actually thread-escaping, so that the query is false. The second more interesting case is that the query holds but it cannot be proved by the analysis because of the inherent incompleteness of the analysis.

The program in Figure 1 with $\mathtt{g} = \mathtt{u}$ chosen for S illustrates the first type of impossibility. In this case, object u is escaping at pc, so the query `local(u)?` does not hold. Our algorithm finds this impossibility as follows. It runs the thread-escape analysis with $\pi_0 = [\mathtt{h1} \mapsto \mathtt{E}, \mathtt{h2} \mapsto \mathtt{E}]$, which computes the abstract state $[\mathtt{u} \mapsto \mathtt{E}, \mathtt{v} \mapsto \mathtt{E}]$ at pc and fails to prove the query. From the failed analysis run, the algorithm generates the counterexample trace:

```
u = new h1 ; v = new h2 ; g = u ; local(u)?
```

which the meta-analysis analyzes, so as to compute the condition $\pi(\mathtt{h1}) = \mathtt{E} \vee \pi(\mathtt{h1}) = \mathtt{L}$ for making the analysis fail to prove the query. But this condition always holds, implying that the analysis will always fail to prove the query no matter which parameter value is used. Our approach detects this, and returns impossibility. Note that although there are four parameter values, our analysis detects this impossibility in a single iteration just by generalizing the reason of failure of the first analysis run.

We use the program in Figure 2 to illustrate the second type of impossibility, which occurs due to the incompleteness of the analysis. The program in the figure first allocates two objects u and v by calling a wrapper function `create()` to the allocation command, and then assigns v to g, making the second object v thread-escaping. A query is given right after this assignment, and it asks whether u points to a thread-local object. The object u is thread-local so the query holds in the concrete semantics. However, the thread-escape analysis cannot prove the query. For this program, there are two possible parameter values, $\pi_0 = [\mathtt{h} \mapsto \mathtt{E}]$ and $\pi_1 = [\mathtt{h} \mapsto \mathtt{L}]$, but using either of them leads to abstract state $[\mathtt{u} \mapsto \mathtt{E}, \mathtt{v} \mapsto \mathtt{E}]$ at pc and makes the analysis fail to prove the query. For instance, when the analysis is run with π_1, it computes the abstract state $[\mathtt{u} \mapsto \mathtt{L}, \mathtt{v} \mapsto \mathtt{L}]$ right before the assignment $\mathtt{g} = \mathtt{v}$, but the assignment transforms this state to $[\mathtt{u} \mapsto \mathtt{E}, \mathtt{v} \mapsto \mathtt{E}]$, because it needs to consider the possibility that u and v are aliased and both get affected by the assignment. Our algorithm finds this impossibility. As usual, it starts by running the analysis with π_0 above, fails to prove the query, and obtains the counterexample trace:

```
u = create() ; v = create() ; g = v ; local(u)?
```

The trace is analyzed by the meta-analysis, which computes the condition $\pi(\mathtt{h}) = \mathtt{E} \vee \pi(\mathtt{h}) = \mathtt{L}$. From this condition, the algorithm concludes that the query is impossible to prove.

3. Preliminaries

This section describes a formal setting used throughout the paper.

3.1 Programming Language

We present our results using a simple imperative language:

$$\begin{array}{rl} \text{(atomic command)} & a ::= \ldots \\ \text{(program)} & s ::= a \mid s\,;\,s' \mid s + s' \mid s^* \end{array}$$

$$\begin{array}{rcl} \mathsf{trace}(a) &=& \{a\} \\ \mathsf{trace}(s + s') &=& \mathsf{trace}(s) \cup \mathsf{trace}(s') \\ \mathsf{trace}(s\,;\,s') &=& \{\tau\,\tau' \mid \tau \in \mathsf{trace}(s) \wedge \tau' \in \mathsf{trace}(s')\} \\ \mathsf{trace}(s^*) &=& \mathsf{leastFix}\,\lambda T.\,\{\epsilon\} \cup \{\tau;\tau' \mid \tau \in T \wedge \tau' \in \mathsf{trace}(s)\} \end{array}$$

Figure 3. Traces of a program s. Symbol ϵ denotes an empty trace.

$$\begin{array}{rcl} F_\pi[s] &:& 2^{\mathbb{D}} \to 2^{\mathbb{D}} \\ F_\pi[a](D) &=& \{[\![a]\!]_\pi(d) \mid d \in D\} \\ F_\pi[s\,;\,s'](D) &=& (F_\pi[s'] \circ F_\pi[s])(D) \\ F_\pi[s + s'](D) &=& F_\pi[s](D) \cup F_\pi[s'](D) \\ F_\pi[s^*](D) &=& \mathsf{leastFix}\,\lambda D_0.\,D \cup F_\pi[s](D_0) \\ \\ F_\pi[\tau] &:& \mathbb{D} \to \mathbb{D} \\ F_\pi[\epsilon](d) &=& d \\ F_\pi[a](d) &=& [\![a]\!]_\pi(d) \\ F_\pi[\tau\,;\,\tau'](d) &=& F_\pi[\tau'](F_\pi[\tau](d)) \end{array}$$

Figure 4. Abstract semantics. In the case of loop, we take the least fixpoint with respect to the subset order in the powerset domain $2^{\mathbb{D}}$.

The language includes a (unspecified) set of atomic commands. Examples are assignments $v = w.f$ and assume statements $\mathsf{assume}(e)$, which filter out executions where e evaluates to false. The language also contains the standard compound constructs: sequential composition, non-deterministic choice, and iteration.

A ***trace*** τ is a finite sequence of atomic commands $a_1 a_2 \ldots a_n$. It records the steps taken during one execution of a program. Function $\mathsf{trace}(s)$ in Figure 3 shows a standard method for generating all possible traces of a program s.

3.2 Parametric Static Analysis

We consider static analyses whose transfer functions for atomic commands are parameterized. The parameter controls the precision/cost tradeoff of the analysis. By choosing it carefully, we can tailor the analysis to a given program and a given query.

Our parametric analyses are specified by the following data:

1. A partially ordered set $(\mathbb{P}, \preceq)$ specified by a preorder $\preceq$ (i.e., $\preceq$ is reflexive and transitive). Elements $\pi \in \mathbb{P}$ are parameter values, and decide the degree of abstraction used by the analysis. The preorder $\preceq$ on π's approximates the cost of running the analysis in terms of speed. Hence, running the analysis with a smaller π correlates the lower cost of the analysis.

   We require that every nonempty subset $P \subseteq \mathbb{P}$ has a *minimum* element $\pi \in P$ (i.e., $\pi \preceq \pi'$ for every $\pi' \in P$).
2. A finite set $\mathbb{D}$ of abstract states. Our analysis uses a set of abstract states to approximate reachable concrete states at each program point. Formally, this means the analysis is disjunctive.
3. A transfer function $[\![a]\!]_\pi : \mathbb{D} \to \mathbb{D}$ for each atomic command a. The function is parameterized by $\pi \in \mathbb{P}$.

A parametric analysis analyzes a program in a standard way, except that it requires a parameter value to be provided before the analysis starts. The abstract semantics in Figure 4 describes the behavior of the analysis formally. In the figure, a program s denotes a transformer $F_\pi[s]$ on sets of abstract states, which is parameterized by $\pi \in \mathbb{P}$. Note that the parameter π is used when atomic commands are interpreted. Hence, π controls the analysis by changing the transfer functions for atomic commands. Other than this parameterization, all the defining clauses are standard.

We remind the reader of a well-known result on disjunctive program analyses, which applies to our parametric analyses.

LEMMA 1. *For all programs s, parameter values π, and abstract states d, we have that $F_\pi[s](\{d\}) = \{F_\pi[\tau](d) \mid \tau \in \mathsf{trace}(s)\}$, where $F_\pi[\tau]$ is the result of analyzing trace τ as shown in Figure 4.*

The lemma ensures that for all final abstract states $d' \in F_\pi[s](\{d\})$, we can construct a trace τ transforming d to d'. This trace does not have a loop, and is significantly simpler than the original program s. We exploit this simplicity of generated traces in our approach, as will be shown later.

Example: Thread-Escape Analysis. The thread-escape analysis [13] is an example of a parametric static analysis. Let $\mathbb{H}$ be a finite set of allocation sites, $\mathbb{L}$ that of local variables, and $\mathbb{F}$ a finite set of object fields. The analysis uses the following domains for parameter values and abstract states:

$$\mathbb{P} = \mathbb{H} \to \{\mathtt{L}, \mathtt{E}\}, \qquad \mathbb{D} = (\mathbb{L} \cup \mathbb{F}) \to \{\mathtt{L}, \mathtt{E}, \mathtt{N}\},$$
$$\pi \preceq \pi' \Leftrightarrow |\{h \in \mathbb{H} \mid \pi(h) = \mathtt{L}\}| \le |\{h \in \mathbb{H} \mid \pi'(h) = \mathtt{L}\}|.$$

Here $\mathtt{N}$ means the null value, and $\mathtt{L}$ and $\mathtt{E}$ are abstract locations, representing disjoint sets of heap objects, except that both $\mathtt{L}$ and $\mathtt{E}$ include the null value. The abstract location $\mathtt{E}$ summaries null, all the thread-escaping objects, and possibly some of thread-local ones. On the other hand, $\mathtt{L}$ denotes all the other heap objects and null. Hence it means only thread-local objects, although it might miss some such objects. The analysis maintains an invariant that $\mathtt{E}$-summarized objects are closed under pointer reachability: if a heap object is summarized by $\mathtt{E}$, following any of its fields always gives null or $\mathtt{E}$-summarized objects, but not $\mathtt{L}$-summarized ones.

A parameter value π determines for each site $h \in \mathbb{H}$ whether $\mathtt{L}$ or $\mathtt{E}$ should be used to summarize objects allocated at h. An abstract element d is a function from local variables or fields of $\mathtt{L}$-summarized objects to abstract locations or $\mathtt{N}$. It records the values of local variables and those of the fields of $\mathtt{L}$-summarized heap objects. For instance, the abstract state $[v \mapsto \mathtt{L}, f_1 \mapsto \mathtt{E}, f_2 \mapsto \mathtt{E}]$ means that a local variable v points to some heap object summarized by $\mathtt{L}$, and fields f_1, f_2 of every $\mathtt{L}$ object point to those summarized by $\mathtt{E}$. Since all the thread-escaping objects are summarized by $\mathtt{E}$, this abstract state implies that the heap object v is thread-local.

We order parameter values $\pi \preceq \pi'$ based on how many sites are mapped to $\mathtt{L}$ by π and π'. The number of $\mathtt{L}$-mapped sites usually correlates the performance of the analysis. Hence, it is desirable to run the analysis with a small parameter value when we attempt to prove a given query.

The thread-escape analysis is developed to prove properties of heap-manipulating programs, so it includes transfer functions of the following heap-manipulating commands:

$$a \ ::= \ v = \mathsf{new}\, h \mid g = v \mid v = g \mid v = \mathtt{null} \mid v = v' \mid v = v'.f \mid v.f = v'$$

Here g and v are global and local variables, respectively, and $h \in \mathbb{H}$ is an allocation site. The transfer functions of these commands simulate their standard meanings but on abstract states, rather than on concrete states. They are shown in Figure 5.

The function $[\![v = \mathsf{new}\, h]\!]_\pi(d)$ makes the variable v point to the abstract location $\pi(h)$, which simulates the allocation of a $\pi(h)$-summarized heap object and the binding of this object with v. The transfer function for $g = v$ models that if v points to a thread-local object, this assignment makes the object escaping, because it exposes the object's reference to other threads via the global variable g. When v points to $\mathtt{L}$, the transfer function calls $\mathsf{esc}(d)$, which sets all the local variables to $\mathtt{E}$ (unless they have the $\mathtt{N}$ value) and resets all the fields to $\mathtt{N}$. This means that after calling $\mathsf{esc}(d)$, the analysis loses most of the information about thread locality, and concludes that all the variables point to potentially thread-escaping objects. This dramatic information loss is inevitable because if v points to $\mathtt{L}$ beforehand, the assignment $g = v$ can lead to the escaping of any object summarized by $\mathtt{L}$. The transfer functions

$$\begin{array}{rcl}
\mathsf{esc} & : & \mathbb{D} \to \mathbb{D} \\
\mathsf{esc}(d) & = & \lambda u.\,\text{if } (d(u) = \mathtt{N} \vee u \in \mathbb{F}) \text{ then } \mathtt{N} \text{ else } \mathtt{E} \\[6pt]
[\![a]\!]_\pi & : & \mathbb{D} \to \mathbb{D} \\
[\![v = \mathsf{new}\, h]\!]_\pi(d) & = & d[v \mapsto \pi(h)] \\
[\![g = v]\!]_\pi(d) & = & \text{if } (d(v) = \mathtt{L}) \text{ then } \mathsf{esc}(d) \text{ else } d \\
[\![v = g]\!]_\pi(d) & = & d[v \mapsto \mathtt{E}] \\
[\![v = \mathtt{null}]\!]_\pi(d) & = & d[v \mapsto \mathtt{N}] \\
[\![v = v']\!]_\pi(d) & = & d[v \mapsto d(v')] \\
[\![v = v'.f]\!]_\pi(d) & = & \begin{cases} d[v \mapsto d(f)] & \text{if } (d(v') = \mathtt{L}) \\ d[v \mapsto \mathtt{E}] & \text{otherwise} \end{cases} \\
[\![v.f = v']\!]_\pi(d) & = &
\end{array}$$
$$\begin{cases}
\mathsf{esc}(d) & \text{if } d(v) = \mathtt{E} \wedge d(v') = \mathtt{L} \\
d & \text{if } (d(v) = \mathtt{E} \wedge d(v') \neq \mathtt{L}) \vee d(v) = \mathtt{N} \\
\mathsf{esc}(d) & \text{if } d(v) = \mathtt{L} \wedge \{d(f), d(v')\} = \{\mathtt{L}, \mathtt{E}\} \\
d[f \mapsto \mathtt{L}] & \text{if } d(v) = \mathtt{L} \wedge \{d(f), d(v')\} = \{\mathtt{N}, \mathtt{L}\} \\
d[f \mapsto \mathtt{E}] & \text{if } d(v) = \mathtt{L} \wedge \{d(f), d(v')\} = \{\mathtt{N}, \mathtt{E}\} \\
d & \text{if } d(v) = \mathtt{L} \wedge d(f) = d(v')
\end{cases}$$

Figure 5. Transfer function for the thread-escape analysis.

of the other atomic commands in the figure can be understood similarly by referring to the concrete semantics and approximating it over abstract states.

Running Example. Figure 6 revisits the example from Figure 1 with `v.f = u` chosen for S. It shows the abstract state (denoted by $\Downarrow$) computed by our thread-escape analysis at each point of an abstract counterexample trace of this program (for now, ignore the abstract states computed by the backward meta-analysis, denoted by $\Uparrow$). The results of the analysis in parts (a) and (b1) of the figure are obtained using the same parameter value $[\mathtt{h1} \mapsto \mathtt{E}, \mathtt{h2} \mapsto \mathtt{E}]$ and are thus identical; the results in part (b2) are obtained using a different parameter value $[\mathtt{h1} \mapsto \mathtt{L}, \mathtt{h2} \mapsto \mathtt{E}]$.

Example: Simplified Typestate Analysis. The second example is a simplified version of the typestate analysis from [5]. This analysis tracks the so-called typestate of a heap object, which is initially set to *init* but changes depending on the method called on the object. In order to do this tracking correctly and precisely, the original analysis keeps various information about pointer aliasing. Our simplified version only keeps information about must aliases.

We assume we are given a set $\mathbb{TS}$ of typestates containing *init*, which represents the initial typestate of heap objects, and a function $[\![m]\!] : \mathbb{TS} \to \mathbb{TS} \cup \{\top\}$ for every method m, which describes how a call $x.m()$ changes the typestate of object x and when it leads to an error, denoted $\top$. Using these assumed data, we specify the domains and transfer functions of the analysis in Figure 7.

The parameter π of the analysis is a set of local or global variables that determines what can appear in the must-alias set of an abstract state. The abstract state d of the analysis has form (Σ, X) or $\top$, where X should be a subset of the parameter π, and it tracks information about a single heap object. In the former case, Σ represents all the possible typestates of the tracked object and X, the must-alias set of this object. The latter means that the heap object can be in any typestate including the error state $\top$. For brevity, we present transfer functions only for simple assignments and method calls. Those for assignments $x = y$ and $x = \mathtt{null}$ update the must-alias set according to their concrete semantics and the parameter value π. The transfer function for a method call $x.m()$ updates the Σ component of the input abstract state using $[\![m]\!]$. In the case where the must-alias set is imprecise, this update includes both the old and new typestates.

According to our order $\preceq$, a parameter value π is smaller than π' when its cardinality is smaller than π'. Hence, running this analysis with a smaller π implies that the analysis would be less

```
    ↓ [u ↦ N, v ↦ N]
    ↑ h1→E ∨ (h2→E ∧ h1→L)
u = new h1;
    ↓ [u ↦ E, v ↦ N]
    ↑ u→E ∨ (h2→E ∧ u→L) ∨
      (h2→L ∧ (f→E ∧ u→L)
v = new h2;
    ↓ [u ↦ E, v ↦ E]
    ↑ u→E ∨ (v→E ∧ u→L) ∨
      (v→L ∧ f→E ∧ u→L)
v.f = u;
    ↓ [u ↦ E, v ↦ E]
    ↑ u→E
pc: local(u)?
```

(a) Iteration 1 without under-approximation.

```
    ↓ [u ↦ N, v ↦ N]
    ↑ h1→E
u = new h1;
    ↓ [u ↦ E, v ↦ N]
    ↑ u→E
v = new h2;
    ↓ [u ↦ E, v ↦ E]
    ↑ u→E
v.f = u;
    ↓ [u ↦ E, v ↦ E]
    ↑ u→E
pc: local(u)?
```

Iteration 1: $\pi = [\texttt{h1} \mapsto \texttt{E}, \texttt{h2} \mapsto \texttt{E}]$

```
    ↓ [u ↦ N, v ↦ N]
    ↑ h1→L ∧ h2→E
u = new h1;
    ↓ [u ↦ L, v ↦ N]
    ↑ u→L ∧ h2→E
v = new h2;
    ↓ [u ↦ L, v ↦ E]
    ↑ u→L ∧ v→E
v.f = u;
    ↓ [u ↦ E, v ↦ E]
    ↑ u→E
pc: local(u)?
```

Iteration 2: $\pi = [\texttt{h1} \mapsto \texttt{L}, \texttt{h2} \mapsto \texttt{E}]$

(b) With under-approximation.

Figure 6. Example illustrating our approach for finding a minimum-cost proof.

$$
\begin{array}{rl}
\text{(typestates)} & \sigma \in \mathbb{TS} \;\; \text{(we assume } \mathit{init} \in \mathbb{TS}) \\
\text{(local/global variables)} & x, y \in \mathbb{V} \\
\text{(analysis parameter)} & \pi \in \mathbb{P} = 2^{\mathbb{V}} \\
\text{(abstract state)} & d \in \mathbb{D} = (2^{\mathbb{TS}} \times 2^{\mathbb{V}}) \cup \{\top\} \\
\text{(order on parameters)} & \pi \preceq \pi' \Leftrightarrow |\pi| \leq |\pi'|
\end{array}
$$

$$
\text{(transfer function) } [\![a]\!]_\pi \;:\; \mathbb{D} \to \mathbb{D}
$$

$$
\begin{aligned}
[\![a]\!]_\pi(\top) &= \top \\
[\![x = y]\!]_\pi(\Sigma, X) &= \begin{cases} (\Sigma, X \cup \{x\}) & \text{if } y \in X \wedge x \in \pi \\ (\Sigma, X \setminus \{x\}) & \text{otherwise} \end{cases} \\
[\![x = \texttt{null}]\!]_\pi(\Sigma, X) &= (\Sigma, X \setminus \{x\}) \\
[\![x.m()]\!]_\pi(\Sigma, X) &= \begin{cases} \top & \text{if } \exists \sigma \in \Sigma.\ [\![m]\!](\sigma) = \top \\ (\{[\![m]\!](\sigma) \mid \sigma \in \Sigma\}, X) & \text{else if } x \in X \\ (\{\sigma\} \cup \{[\![m]\!](\sigma) \mid \sigma \in \Sigma\}, X) & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 7. Data for the typestate analysis.

precise about must-alias sets, which normally correlates the faster convergence of the fixpoint iteration of the analysis.

3.3 Parametric Static Analysis Problem

Parametric analyses are used in the context of query-driven verification, where we are given not just a program to analyze but also a query to prove. In this usage scenario, the most important matter to resolve before running the analysis is to choose a right parameter value π. Ideally, we would like to pick π that forces the analysis to keep enough information to prove a given query for a given program, but to discard information unnecessary for this proof, so that the analysis achieves high efficiency.

The parametric static analysis problem provides a guideline on resolving the issue of parameter value selection. It sets a specific target on parameter values that we should aim at. Assume that we are interested in queries expressed as subsets of $\mathbb{D}$. The problem is defined as follows:

DEFINITION 2 (Parametric Static Analysis Problem). *Given a program s, an initial abstract state d_I, and a query $q \subseteq \mathbb{D}$, find a minimum parameter value π such that $F_\pi[s]\{d_I\} \subseteq q$,*[1] *or show that $F_\pi[s]\{d_I\} \subseteq q$ does not hold for any π.*

Note that the problem asks for not a minimal π, but a minimum hence cheapest one. This requirement encourages any reasonable solution of the problem to exploit cheap parameter values (determined by $\preceq$) as much as possible.

[1] The condition that π be minimum means: if $F_{\pi'}[s]d_I \subseteq q$ then $\pi \preceq \pi'$.

Our approach to solve the problem is based on using a form of a backward meta-analysis and reasoning about the behavior of a parametric static analysis under different parameter values simultaneously. We explain this backward meta-analysis next.

4. Backward Meta-Analysis

In this section, we fix a parametric static analysis

$$(\mathbb{P}, \preceq, \mathbb{D}, [\![-]\!])$$

and describe corresponding backward meta-analyses. To avoid the confusion between these two analyses, we often call the parametric static analysis as ***forward analysis***.

A backward meta-analysis is a core component of our algorithm for the parametric static analysis problem. It is invoked when the forward analysis fails to prove a query. The meta-analysis attempts to determine why a run of the forward analysis with a specific parameter value π fails to prove a query, and to generalize this reason. Concretely, the inputs to the meta-analysis are a trace τ, a parameter value π, and an initial abstract state d_I, such that the π instance of the forward analysis fails to prove that a given query holds at the end of τ. Given such inputs, the meta-analysis analyzes τ backward, and collects parameter values that lead to a similar verification failure of the forward analysis. The collected parameter values are used subsequently when our top-level algorithm computes a necessary condition on parameters for proving the given query and chooses a next parameter value to try based on this condition.

Formally, the meta-analysis is specified by the following data:

- A set $\mathbb{M}$ and a function
$$\gamma : \mathbb{M} \to 2^{\mathbb{P} \times \mathbb{D}}.$$
Elements in $\mathbb{M}$ are the main data structures used by the meta-analysis, and γ determines their meanings. We suggest to read elements in $\mathbb{M}$ as predicates over $\mathbb{P} \times \mathbb{D}$. The meta-analysis uses such a predicate $\phi \in \mathbb{M}$ to express a sufficient condition for verification failure: for every $(\pi, d) \in \gamma(\phi)$, if we instantiate the forward analysis with π and run this instance from the abstract state d (over the part of a trace analyzed so far), we will fail to prove a given query.
- A function
$$[\![a]\!]^b : \mathbb{M} \to \mathbb{M}$$
for each atomic command a. The input $\phi_1 \in \mathbb{M}$ represents a postcondition on $\mathbb{P} \times \mathbb{D}$. Given such ϕ_1, the function computes the weakest precondition ϕ such that running $[\![a]\!]$ from any abstract state in $\gamma(\phi)$ has an outcome in $\gamma(\phi_1)$. This intuition

$$\begin{aligned} B[\tau] &: \mathbb{P} \times \mathbb{D} \times \mathbb{M} \to \mathbb{M} \\ B[\epsilon](\pi, d, \phi) &= \phi \\ B[a](\pi, d, \phi) &= \mathsf{approx}(\pi, d, [\![a]\!]^b(\phi)) \\ B[\tau\,;\,\tau'](\pi, d, \phi) &= B[\tau](\pi, d, B[\tau'](\pi, F_\pi[\tau](d), \phi)) \end{aligned}$$

Figure 8. Backward meta-analysis.

$$\begin{aligned} &\mathsf{negate}(p) = \mathsf{not}(p) \\ &\mathsf{negate}(\mathsf{true}) = \mathsf{false} \quad \mathsf{negate}(\phi \wedge \phi') = \mathsf{negate}(\phi) \vee \mathsf{negate}(\phi') \\ &\mathsf{negate}(\mathsf{false}) = \mathsf{true} \quad \mathsf{negate}(\phi \vee \phi') = \mathsf{negate}(\phi) \wedge \mathsf{negate}(\phi') \end{aligned}$$

$\mathsf{toDNF}(\phi)$ transforms ϕ to the DNF form and sorts disjuncts by size

$$\begin{aligned} &\mathsf{simplify}(\bigvee\{\phi_i \mid i \in \{1, \ldots, n\}\}) = \\ &\qquad \bigvee\{\phi_i \mid i \in \{1, \ldots, n\} \wedge \neg(\exists j < i. \phi_j \sqsubseteq \phi_i)\} \end{aligned}$$

$$\begin{aligned} &\mathsf{drop}_k(\pi, d, \bigvee\{\phi_i \mid i \in \{1, \ldots, n\}\}) = \\ &\qquad (\bigvee\{\phi_i \mid i \in \{1, \ldots, \min(k-1, n)\}\}) \vee \phi_j \\ &\qquad \text{(where } (\pi, d) \in \gamma(\phi_j) \text{ and } j \text{ is the smallest such index)} \end{aligned}$$

Figure 9. Functions for manipulating ϕ's.

is formalized by the following requirement on $[\![a]\!]^b$:

$$\forall \phi_1 \in \mathbb{M}.\ \gamma([\![a]\!]^b(\phi_1)) = \{(\pi, d) \mid (\pi, [\![a]\!]_\pi(d)) \in \gamma(\phi_1)\}. \quad (1)$$

- A function
$$\mathsf{approx} : \mathbb{P} \times \mathbb{D} \times \mathbb{M} \to \mathbb{M}.$$
The function is required to meet the following two conditions:
  1. $\forall \pi, d, \phi.\ \gamma(\mathsf{approx}(\pi, d, \phi)) \subseteq \gamma(\phi)$; and
  2. $\forall \pi, d, \phi.\ (\pi, d) \in \gamma(\phi) \Rightarrow (\pi, d) \in \gamma(\mathsf{approx}(\pi, d, \phi))$.

  The first condition ensures that $\mathsf{approx}(\pi, d, \phi)$ underapproximates the input ϕ, and the second says that this underapproximation should keep at least (π, d), if it is already in $\gamma(\phi)$. The main purpose of approx is to simplify the input ϕ. For instance, when ϕ is a logical formula, approx turns ϕ to a syntactically simpler one. The operator is invoked frequently by our meta-analysis, and helps the analysis to keep the complexity of the $\mathbb{M}$ value (the analysis's main data structure) under control.

Using the given data, our backward meta-analysis analyzes a trace τ backward as described in Figure 8. For each atomic command a in τ, it transforms an input ϕ using $[\![a]\!]^b$ first. Then, it simplifies the resulting ϕ' using the approx operator.

Our meta-analysis correctly tracks a sufficient condition that the forward analysis fails to prove a query. This condition is not trivial (i.e., it is a satisfiable formula), and includes enough information about the current failed verification attempt on τ by the forward analysis. Our theorem below formalizes these guarantees.

THEOREM 3 (Soundness). *For all τ, π, d and $\phi \in \mathbb{M}$,*

1. $(\pi, F_\pi[\tau](d)) \in \gamma(\phi) \Rightarrow (\pi, d) \in \gamma(B[\tau](\pi, d, \phi))$*; and*
2. $\forall (\pi_0, d_0) \in \gamma(B[\tau](\pi, d, \phi)).\ (\pi_0, F_{\pi_0}[\tau](d_0)) \in \gamma(\phi)$.

Proof See Appendix A.2 of our technical report [19].

4.1 Disjunctive Meta-Analysis and Underapproximation

Designing a good underapproximation operator approx is important for the performance of a backward meta-analysis, and it often requires new insights. In this subsection, we identify a special subclass of meta-analyses, called *disjunctive meta-analyses*, and define a generic underapproximation operator for these meta-analyses. All the example analyses in the paper have disjunctive meta-analyses, and use the generic underapproximation operator.

A meta-analysis is ***disjunctive*** if the following conditions hold:

- The set $\mathbb{M}$ consists of formulas ϕ:
$$\phi ::= p \mid \mathsf{true} \mid \phi \wedge \phi' \mid \mathsf{false} \mid \phi \vee \phi' \qquad (p \in \mathsf{PForm})$$
where PForm is a set of primitive formulas, and the conjunction and disjunction operators have the standard meanings:
$$\begin{aligned} \gamma(\mathsf{true}) &= \mathbb{P} \times \mathbb{D} & \gamma(\phi \wedge \phi') &= \gamma(\phi) \cap \gamma(\phi') \\ \gamma(\mathsf{false}) &= \emptyset & \gamma(\phi \vee \phi') &= \gamma(\phi) \cup \gamma(\phi') \end{aligned}$$
- The negation of every $p \in \mathsf{PForm}$ can be expressed by the disjunction of other primitive formulas $\phi = p_1 \vee \ldots \vee p_n$:
$$(\mathbb{P} \times \mathbb{D}) \setminus \gamma(p) = \gamma(p_1 \vee \ldots \vee p_n).$$
We pick one such ϕ for each $p \in \mathsf{PForm}$, and denote it $\mathsf{not}(p)$.
- The set $\mathbb{M}$ comes with a binary relation $\sqsubseteq$ such that
$$\forall \phi, \phi' \in \mathbb{M}.\ \phi \sqsubseteq \phi' \Rightarrow \gamma(\phi) \subseteq \gamma(\phi').$$

The second condition above has a consequence that the negation of every formula in $\mathbb{M}$ can be expressed in $\mathbb{M}$, as shown by the negate operator in Figure 9. Hence, the domain $\mathbb{M}$ of a disjunctive meta-analysis in a sense contains all the formulas from boolean logic, which are constructed from primitive ones in PForm.

We define a generic underapproximation operator for disjunctive meta-analyses as follows:

$$\begin{aligned} \mathsf{approx} &: \mathbb{P} \times \mathbb{D} \times \mathbb{M} \to \mathbb{M} \\ \mathsf{approx}(\pi, d, \phi) &= \text{let } \phi' = (\mathsf{simplify} \circ \mathsf{toDNF})(\phi) \text{ in} \\ &\quad\ \text{if (the number of disjuncts in } \phi' \leq k) \text{ then } \phi' \\ &\quad\ \text{else let } F = (\mathsf{simplify} \circ \mathsf{toDNF} \circ \mathsf{negate}) \text{ in} \\ &\qquad\quad \text{let } \phi'' = (F \circ F)(\phi') \text{ in } \mathsf{drop}_k(\pi, d, \phi'') \end{aligned}$$

The approx operator works in four steps. First, it transforms ϕ to the disjunctive normal form, and removes redundant disjuncts in the DNF formula, which are subsumed by other shorter disjuncts in the same formula. If the resulting formula ϕ' is simple enough in the sense that it has no more than k disjuncts (where k is predetermined by an analysis designer), the operator stops and returns ϕ' as a result. Otherwise, it moves to the next step. Second, the approx operator negates ϕ', converts the result $\mathsf{negate}(\phi')$ to the disjunctive normal form, and removes redundant disjuncts in the DNF formula. The function F above implements this second step. Third, the operator applies F again. The result after these two steps is bound to ϕ'' above, and it is in the disjunctive normal form. Finally, some disjuncts of ϕ'' are pruned if the number of disjuncts in ϕ'' goes above the threshold value k. During the pruning, the first $k-1$ disjuncts according to their syntactic size survive, together with the shortest disjunct ϕ_j that includes the input (π, d) (i.e., $(\pi, d) \in \gamma(\phi_j)$). Our pruning is an instance of beam search in Artificial Intelligence, which always keeps only the most promising k options during exploration of a search space. Subroutines negate, toDNF, simplify, and drop are defined in Figure 9.

A common confusion is to regard the application of $F \circ F$ inside the approx operator as redundant. Semantically, F just computes the negation of ϕ', as $\gamma(F(\phi')) = (\mathbb{P} \times \mathbb{D}) \setminus \gamma(\phi')$. Hence, $(F \circ F)(\phi')$ computes the double negation of a formula ϕ', which means the same as the original ϕ'. How can such an identity function do anything useful? A good way to clear this confusion is to remember that the main goal of $(F \circ F)(\phi')$ is to change the syntax of ϕ', and that $F(\phi')$ expresses the negated version of ϕ' without using any explicit negation operator. When $F(\phi')$ negates its argument, it expresses the negation of each primitive formula p in ϕ' in terms of $p_1 \vee \ldots \vee p_n$. Hence, the negated version usually has more disjuncts than ϕ', and has a higher chance of including redundant disjuncts, which will be pruned by the call to simplify in $F(\phi')$. Thus, applying F twice is not no-op, and usually simplifies ϕ' dramatically, which we observed in our experiments.

$$\llbracket a\rrbracket^b(\mathsf{true}) \;=\; \mathsf{true} \qquad \llbracket a\rrbracket^b(\mathsf{false}) \;=\; \mathsf{false} \qquad \llbracket a\rrbracket^b(\phi\wedge\phi') \;=\; \llbracket a\rrbracket^b(\phi)\wedge\llbracket a\rrbracket^b(\phi') \qquad \llbracket a\rrbracket^b(\phi\vee\phi') \;=\; \llbracket a\rrbracket^b(\phi)\vee\llbracket a\rrbracket^b(\phi')$$

$$\llbracket g = v\rrbracket^b(\delta\rightarrow o) \;=\; \begin{cases} \mathsf{false} & \text{if } \delta\equiv v\wedge o=\mathtt{L} \\ v\rightarrow\mathtt{L}\vee v\rightarrow\mathtt{E} & \text{if } \delta\equiv v\wedge o=\mathtt{E} \\ v\rightarrow\mathtt{N} & \text{if } \delta\equiv v\wedge o=\mathtt{N} \\ (v\rightarrow\mathtt{E}\vee v\rightarrow\mathtt{N})\wedge\delta\rightarrow\mathtt{L} & \text{if } \delta\in(\mathbb{L}\setminus\{v\})\wedge o=\mathtt{L} \\ (v\rightarrow\mathtt{L}\wedge\delta\rightarrow\mathtt{L})\vee\delta\rightarrow\mathtt{E} & \text{if } \delta\in(\mathbb{L}\setminus\{v\})\wedge o=\mathtt{E} \\ \delta\rightarrow\mathtt{N} & \text{if } \delta\in(\mathbb{L}\setminus\{v\})\wedge o=\mathtt{N} \\ (v\rightarrow\mathtt{E}\vee v\rightarrow\mathtt{N})\wedge\delta\rightarrow o & \text{if } \delta\in\mathbb{F}\wedge o\in\{\mathtt{L},\mathtt{E}\} \\ v\rightarrow\mathtt{L}\vee((v\rightarrow\mathtt{E}\vee v\rightarrow\mathtt{N})\wedge\delta\rightarrow\mathtt{N}) & \text{if } \delta\in\mathbb{F}\wedge o=\mathtt{N} \\ \delta\rightarrow o & \text{otherwise} \end{cases}$$

$$\llbracket v = g\rrbracket^b(\delta\rightarrow o) \;=\; \text{if } (\delta\equiv v\wedge o=\mathtt{E}) \text{ then } (\mathsf{true}) \text{ else } (\text{if } (\delta\equiv v\wedge o\neq\mathtt{E}) \text{ then } (\mathsf{false}) \text{ else } (\delta\rightarrow o))$$

$$\llbracket v = \mathsf{new}\, h\rrbracket^b(\delta\rightarrow o) \;=\; \text{if } (\delta\equiv v) \text{ then } (h\rightarrow o) \text{ else } (\delta\rightarrow o)$$

$$\llbracket v = \mathtt{null}\rrbracket^b(\delta\rightarrow o) \;=\; \text{if } (\delta\equiv v\wedge o=\mathtt{N}) \text{ then } (\mathsf{true}) \text{ else } (\text{if } (\delta\equiv v\wedge o\neq\mathtt{N}) \text{ then } (\mathsf{false}) \text{ else } (\delta\rightarrow o))$$

$$\llbracket v = v'\rrbracket^b(\delta\rightarrow o) \;=\; \text{if } (\delta\equiv v) \text{ then } (v'\rightarrow o) \text{ else } (\delta\rightarrow o)$$

$$\llbracket v = v'.f\rrbracket^b(\delta\rightarrow o) \;=\; \begin{cases} (v'\rightarrow\mathtt{L}\wedge f\rightarrow\mathtt{E})\vee v'\rightarrow\mathtt{E}\vee v'\rightarrow\mathtt{N} & \text{if } \delta\equiv v\wedge o=\mathtt{E} \\ v'\rightarrow\mathtt{L}\wedge f\rightarrow o & \text{if } \delta\equiv v\wedge o\neq\mathtt{E} \\ \delta\rightarrow o & \text{if } \delta\not\equiv v \end{cases}$$

$$\llbracket v.f = v'\rrbracket^b(\delta\rightarrow o) \;=\; \begin{cases} \delta\rightarrow o & \text{if } \delta\notin(\mathbb{L}\cup\mathbb{F}) \\ \delta\rightarrow\mathtt{E}\vee(\delta\rightarrow\mathtt{L}\wedge v\rightarrow\mathtt{E}\wedge v'\rightarrow\mathtt{L}) & \text{if } \delta\in\mathbb{L}\wedge o=\mathtt{E} \\ \qquad\vee(\delta\rightarrow\mathtt{L}\wedge v\rightarrow\mathtt{L}\wedge f\rightarrow\mathtt{L}\wedge v'\rightarrow\mathtt{E}) & \\ \qquad\vee(\delta\rightarrow\mathtt{L}\wedge v\rightarrow\mathtt{L}\wedge f\rightarrow\mathtt{E}\wedge v'\rightarrow\mathtt{L}) & \\ \delta\rightarrow o & \text{if } \delta\in\mathbb{L}\wedge o=\mathtt{N} \\ \delta\rightarrow o\wedge(v\rightarrow\mathtt{N}\vee(v\rightarrow\mathtt{E}\wedge(v'\rightarrow\mathtt{E}\vee v'\rightarrow\mathtt{N})) & \text{if } (\delta\in\mathbb{L}\wedge o=\mathtt{L}) \\ \qquad\qquad\vee(v\rightarrow\mathtt{L}\wedge(v'\rightarrow\mathtt{N}\vee f\rightarrow\mathtt{N} & \quad\vee(\delta\in\mathbb{F}\wedge o=\mathtt{E}\wedge\delta\not\equiv f) \\ \qquad\qquad\qquad\qquad\vee(v'\rightarrow\mathtt{L}\wedge f\rightarrow\mathtt{L}) & \quad\vee(\delta\in\mathbb{F}\wedge o=\mathtt{L}\wedge\delta\not\equiv f) \\ \qquad\qquad\qquad\qquad\vee(v'\rightarrow\mathtt{E}\wedge f\rightarrow\mathtt{E})))) & \\ \delta\rightarrow\mathtt{N}\vee(v\rightarrow\mathtt{E}\wedge v'\rightarrow\mathtt{L}) & \text{if } \delta\in\mathbb{F}\wedge o=\mathtt{N}\wedge\delta\not\equiv f \\ \qquad\vee(v\rightarrow\mathtt{L}\wedge f\rightarrow\mathtt{L}\wedge v'\rightarrow\mathtt{E}) & \\ \qquad\vee(v\rightarrow\mathtt{L}\wedge f\rightarrow\mathtt{E}\wedge v'\rightarrow\mathtt{L}) & \\ (\delta\rightarrow\mathtt{N}\wedge(v\rightarrow\mathtt{E}\vee v\rightarrow\mathtt{N}\vee(v\rightarrow\mathtt{L}\wedge v'\rightarrow\mathtt{N}))) & \text{if } \delta\in\mathbb{F}\wedge o=\mathtt{N}\wedge\delta\equiv f \\ \quad\vee(v\rightarrow\mathtt{E}\wedge v'\rightarrow\mathtt{L}) & \\ \quad\vee(\delta\rightarrow\mathtt{L}\wedge v\rightarrow\mathtt{L}\wedge v'\rightarrow\mathtt{E}) & \\ \quad\vee(\delta\rightarrow\mathtt{E}\wedge v\rightarrow\mathtt{L}\wedge v'\rightarrow\mathtt{L}) & \\ (\delta\rightarrow\mathtt{N}\wedge v\rightarrow\mathtt{L}\wedge v'\rightarrow\mathtt{L}) & \text{if } \delta\in\mathbb{F}\wedge o=\mathtt{L}\wedge\delta\equiv f \\ \quad\vee(\delta\rightarrow\mathtt{L}\wedge v\rightarrow\mathtt{N}) & \\ \quad\vee(\delta\rightarrow\mathtt{L}\wedge v\rightarrow\mathtt{E}\wedge(v'\rightarrow\mathtt{E}\vee v'\rightarrow\mathtt{N})) & \\ \quad\vee(\delta\rightarrow\mathtt{L}\wedge v\rightarrow\mathtt{L}\wedge(v'\rightarrow\mathtt{N}\vee v'\rightarrow\mathtt{L})) & \\ (\delta\rightarrow\mathtt{N}\wedge v\rightarrow\mathtt{L}\wedge v'\rightarrow\mathtt{E}) & \text{if } \delta\in\mathbb{F}\wedge o=\mathtt{E}\wedge\delta\equiv f \\ \quad\vee(\delta\rightarrow\mathtt{E}\wedge v\rightarrow\mathtt{N}) & \\ \quad\vee(\delta\rightarrow\mathtt{E}\wedge(v\rightarrow\mathtt{E}\vee v\rightarrow\mathtt{L})\wedge(v'\rightarrow\mathtt{E}\vee v'\rightarrow\mathtt{N})) & \end{cases}$$

Figure 10. Backward transfer function $\llbracket a\rrbracket^b$ for the thread-escape analysis. Here δ ranges over h, v, f, and we use o to denote $\mathtt{L}, \mathtt{E}, \mathtt{N}$.

Example: Meta-analysis for Thread-Escape. We define a disjunctive backward meta-analysis for the thread-escape analysis following the above recipe. Doing so means specifying four entities: the set of primitive formulas, their negation, the order on formulas, and a function $\llbracket - \rrbracket^b$. We specify all these entities one by one.

The domain $\mathbb{M}$ of the meta-analysis is constructed from the following primitive formulas p:

$$p ::= h\rightarrow o \;\mid\; v\rightarrow o \;\mid\; f\rightarrow o$$

where o is an abstract value in $\{\mathtt{L}, \mathtt{E}, \mathtt{N}\}$, and h, v, f are an allocation site, a local variable, and a field, respectively. These formulas describe properties about pairs (π, d) of parameter value and abstract state. Formula $h\rightarrow o$ says that a parameter value π should map h to o. Formula $v\rightarrow o$ means that an abstract state d should bind v to o; formula $f\rightarrow o$ expresses a similar fact on the field f. We formalize these meanings via function $\gamma : \mathbb{M} \rightarrow 2^{\mathbb{P}\times\mathbb{D}}$ below:

$$\gamma(h\rightarrow o) = \{(\pi, d) \mid \pi(h) = o\} \quad \gamma(v\rightarrow o) = \{(\pi, d) \mid d(v) = o\}$$
$$\gamma(f\rightarrow o) = \{(\pi, d) \mid d(f) = o\}$$

All of our primitive formulas express a form of binding properties. The negations of these formulas simply enumerate all the other possible bindings: when η is a local variable v or a field f,

$$\begin{array}{ll} \mathsf{not}(h\rightarrow\mathtt{L}) = h\rightarrow\mathtt{E} & \mathsf{not}(h\rightarrow\mathtt{E}) = h\rightarrow\mathtt{L} \\ \mathsf{not}(h\rightarrow\mathtt{N}) = \mathsf{true} & \mathsf{not}(\eta\rightarrow\mathtt{L}) = (\eta\rightarrow\mathtt{E}\vee\eta\rightarrow\mathtt{N}) \\ \mathsf{not}(\eta\rightarrow\mathtt{E}) = (\eta\rightarrow\mathtt{L}\vee\eta\rightarrow\mathtt{N}) & \mathsf{not}(\eta\rightarrow\mathtt{N}) = (\eta\rightarrow\mathtt{L}\vee\eta\rightarrow\mathtt{E}) \end{array}$$

The allocation sites are treated differently because no parameter value π maps a site h to $\mathtt{N}$.

We order formulas $\phi \sqsubseteq \phi'$ in $\mathbb{M}$ when our simple entailment checker concludes that ϕ' subsumes ϕ. This conclusion is reached when ϕ and ϕ' are the same, or both ϕ and ϕ' are conjunction of primitive formulas and all the primitive formulas in ϕ' appear in ϕ. This proof strategy is fast yet highly incomplete. However, we find it sufficient for our application, where the order is used for detecting redundant disjuncts in formulas in the DNF form.

For each atomic command a, our meta-analysis uses the function $\llbracket a\rrbracket^b$ that satisfies the requirement (4) of our framework. This

$$\text{(primitive formula)}\ p \in \mathsf{PForm}$$

$$p ::= \mathsf{err} \mid \mathsf{errNot} \mid \mathsf{param}(x) \mid \mathsf{paramNot}(x) \mid \mathsf{var}(x) \mid \mathsf{varNot}(x) \mid \mathsf{type}(\sigma) \mid \mathsf{typeNot}(\sigma)$$

$$\begin{aligned}
\gamma(\mathsf{err}) &= \{(\pi, \top)\} \\
\gamma(\mathsf{errNot}) &= \{(\pi, d) \mid d \neq \top\} \\
\gamma(\mathsf{param}(x)) &= \{(\pi, d) \mid x \in \pi\} \\
\gamma(\mathsf{paramNot}(x)) &= \{(\pi, d) \mid x \notin \pi\} \\
\gamma(\mathsf{var}(x)) &= \{(\pi, (\Sigma, X)) \mid x \in X\} \\
\gamma(\mathsf{varNot}(x)) &= \{(\pi, (\Sigma, X)) \mid x \notin X\} \\
\gamma(\mathsf{type}(\sigma)) &= \{(\pi, (\Sigma, X)) \mid \sigma \in \Sigma\} \\
\gamma(\mathsf{typeNot}(\sigma)) &= \{(\pi, (\Sigma, X)) \mid \sigma \notin \Sigma\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{not}(\mathsf{err}) &= \mathsf{errNot} \\
\mathsf{not}(\mathsf{errNot}) &= \mathsf{err} \\
\mathsf{not}(\mathsf{param}(x)) &= \mathsf{paramNot}(x) \\
\mathsf{not}(\mathsf{paramNot}(x)) &= \mathsf{param}(x) \\
\mathsf{not}(\mathsf{var}(x)) &= \mathsf{err} \vee \mathsf{varNot}(x) \\
\mathsf{not}(\mathsf{varNot}(x)) &= \mathsf{err} \vee \mathsf{var}(x) \\
\mathsf{not}(\mathsf{type}(\sigma)) &= \mathsf{err} \vee \mathsf{typeNot}(\sigma) \\
\mathsf{not}(\mathsf{typeNot}(\sigma)) &= \mathsf{err} \vee \mathsf{type}(\sigma)
\end{aligned}$$

$p \sqsubseteq p' \Leftrightarrow p = p'$, or p' is errNot and p is different from err
$\phi \sqsubseteq \phi' \Leftrightarrow \phi = \phi'$, or both ϕ and ϕ' are conjunction of primitive formulas and for every conjunct p' of ϕ', there exists a conjunct p of ϕ such that $p \sqsubseteq p'$

Figure 11. Data for backward meta-analysis for typestate analysis.

requirement means that $[\![a]\!]^b$ collects all the parameter values π and abstract pre-states d such that the run of the π-instantiated analysis with d generates a result satisfying ϕ. That is, $[\![a]\!]^b(\phi)$ computes the weakest precondition of $[\![a]\!]_\pi$ with respect to the postcondition ϕ. The definition of $[\![a]\!]^b$ is given in Figure 10.

LEMMA 4. *For every atomic command a, backward transfer function $[\![a]\!]^b$ in Figure 10 satisfies requirement* (4) *of our framework.*

Proof See Appendix A.3 of our technical report [19].

Running Example. Figure 6 shows the backward meta-analysis for thread-escape analysis without and with underapproximation, on an abstract counterexample trace of the program in Figure 1 with `v.f = u` chosen for S. The trace in part (a) is generated by the forward analysis using initial parameter value [`h1` $\mapsto$ `E`, `h2` $\mapsto$ `E`]. The abstract states computed by the meta-analysis at each point of this trace without doing underapproximation are denoted by $\uparrow$. It correctly computes the sufficient condition for failure at the beginning of the trace as `h1`$\to$`E` $\vee$ (`h2`$\to$`E` $\wedge$ `h1`$\to$`L`), thus yielding the cheapest parameter value that proves the query as [`h1` $\mapsto$ `L`, `h2` $\mapsto$ `L`]. Despite taking a single iteration, however, the lack of underapproximation causes an evident blow-up in the size of the formula tracked by the meta-analysis with the length of the trace.

Part (b) shows the result with underapproximation, using $k = 1$ in the beam search via function drop_k. This time, the first iteration, shown in part (b1), yields a stronger sufficient condition for failure, `h1`$\to$`E`, causing our approach to next run the forward analysis using parameter value [`h1` $\mapsto$ `L`, `h2` $\mapsto$ `E`]. However, the analysis again fails to prove the query, and the second iteration, shown in part (b2), computes the sufficient condition for failure as `h1`$\to$`L` $\wedge$ `h2`$\to$`E`. By combining these conditions from the two iterations, our approach finds the same cheapest parameter value as that without underapproximation in part (a). Despite taking an extra iteration, the formulas our meta-analysis tracks in part (b) are much more compact than those in part (a).

$$\begin{aligned}
[\![a]\!]^b(\mathsf{true}) &= \mathsf{true} \\
[\![a]\!]^b(\phi_1 \wedge \phi_2) &= [\![a]\!]^b(\phi_1) \wedge [\![a]\!]^b(\phi_2) \\
[\![a]\!]^b(\mathsf{false}) &= \mathsf{false} \\
[\![a]\!]^b(\phi_1 \vee \phi_2) &= [\![a]\!]^b(\phi_1) \vee [\![a]\!]^b(\phi_2) \\
[\![x = y]\!]^b(\mathsf{err}) &= \mathsf{err} \\
[\![x = \mathtt{null}]\!]^b(\mathsf{err}) &= \mathsf{err} \\
[\![x.m()]\!]^b(\mathsf{err}) &= \mathsf{err} \vee \bigvee\{\mathsf{type}(\sigma) \mid [\![m]\!](\sigma) = \top\} \\
[\![a]\!]^b(\mathsf{errNot}) &= \mathsf{negate}([\![a]\!]^b(\mathsf{err})) \\
[\![x = y]\!]^b(\mathsf{param}(z)) &= \mathsf{param}(z) \\
[\![x = \mathtt{null}]\!]^b(\mathsf{param}(z)) &= \mathsf{param}(z) \\
[\![x.m()]\!]^b(\mathsf{param}(z)) &= \mathsf{param}(z) \\
[\![a]\!]^b(\mathsf{paramNot}(x)) &= \mathsf{negate}([\![a]\!]^b(\mathsf{param}(x))) \\
[\![x = y]\!]^b(\mathsf{var}(z)) &= \begin{cases} \mathsf{param}(x) \wedge \mathsf{var}(y) & \text{if } x \equiv z \\ \mathsf{var}(z) & \text{otherwise} \end{cases} \\
[\![x = \mathtt{null}]\!]^b(\mathsf{var}(z)) &= \begin{cases} \mathsf{false} & \text{if } x \equiv z \\ \mathsf{var}(z) & \text{otherwise} \end{cases} \\
[\![x.m()]\!]^b(\mathsf{var}(z)) &= \mathsf{var}(z) \wedge \bigwedge\{\mathsf{typeNot}(\sigma) \mid [\![m]\!](\sigma) = \top\} \\
[\![a]\!]^b(\mathsf{varNot}(z)) &= \mathsf{negate}([\![a]\!]^b(\mathsf{var}(z))) \wedge \mathsf{errNot} \\
[\![x = y]\!]^b(\mathsf{type}(\sigma)) &= \mathsf{type}(\sigma) \\
[\![x = \mathtt{null}]\!]^b(\mathsf{type}(\sigma)) &= \mathsf{type}(\sigma) \\
[\![x.m()]\!]^b(\mathsf{type}(\sigma)) &= \mathsf{errNot} \\
&\quad \wedge (\bigwedge\{\mathsf{typeNot}(\sigma') \mid [\![m]\!](\sigma') = \top\}) \\
&\quad \wedge (\mathsf{varNot}(x) \wedge \mathsf{type}(\sigma) \\
&\qquad \vee \bigvee\{\mathsf{type}(\sigma') \mid [\![m]\!](\sigma') = \sigma\}) \\
[\![a]\!]^b(\mathsf{typeNot}(x)) &= \mathsf{negate}([\![a]\!]^b(\mathsf{type}(x))) \wedge \mathsf{errNot}
\end{aligned}$$

Figure 12. Backward transfer function for the typestate analysis.

Example: Meta-Analysis for Typestate. The backward meta-analysis for the typestate analysis is also disjunctive. All the data for this meta-analysis are given in Figures 11 and 12.

The meta-analysis uses six primitive formulas. The first two are err and errNot, and they say that the d component of a pair (π, d) is $\top$ (in the case of err) or non-$\top$ (in the case of errNot). These formulas are useful for capturing the fact that the forward transfer functions often behave differently depending on whether the d component is $\top$ or not. The remaining four formulas describe elements that should be included or excluded in some component of (π, d). For instance, $\mathsf{var}(x)$ says that the d component is a non-$\top$ value (Σ, X) such that the X part contains x. The formula $\mathsf{varNot}(x)$ makes a similar statement on the form of d but it says that the X part of d does not include x. The negation of these primitive formulas is defined following their meanings.

We order formulas $\phi \sqsubseteq \phi'$ in $\mathbb{M}$ when ϕ and ϕ' are the same, or both ϕ and ϕ' are conjunction of primitive formulas and every primitive formula p' in ϕ' corresponds to some primitive formula p in ϕ that implies p'. Finally, the backward transfer function for each atomic command a is given from the requirement (4) of our framework, which determines the semantics of the function in terms of the weakest precondition.

LEMMA 5. *For every atomic command a, backward transfer function $[\![a]\!]^b$ in Figure 12 satisfies requirement* (4) *of our framework.*

Proof See Appendix A.4 of our technical report [19].

5. Iterative Forward-Backward Analysis

This section presents our top-level algorithm, called TRACER, which brings a parametric analysis and a corresponding backward meta-analysis together, and solves the parametric static analysis problem. Throughout the section, we fix a parametric analysis and

	description	# classes		# methods		bytecode (KB)		KLOC		# alloc.
		app	total	app	total	app	total	app	total	sites
tsp	Traveling Salesman implementation from ETH	4	997	21	6,423	2.6	391	0.7	269	6,175
elevator	discrete event simulator	5	998	24	6,424	2.3	390	0.6	269	6,180
hedc	web crawler from ETH	44	1,066	234	6,881	16	442	6	283	7,326
weblech	website download/mirror tool	57	1,263	312	8,201	20	504	13	326	7,663
antlr	A parser/translator generator	118	1,134	1,180	7,786	131	532	29	303	7,748
avrora	microcontroller simulation/analysis tool	1,160	2,192	4,253	10,985	224	634	64	340	10,151
hsqldb	relational database engine	199	1,420	2,818	11,301	222	756	103	415	9,789
lusearch	text indexing and search tool	229	1,236	1,508	8,171	101	511	42	314	7,395

Table 1. Benchmark characteristics. The "# classes" column is the number of classes containing reachable methods. The "# methods" column is the number of reachable methods computed by a 0-CFA call-graph analysis. The "bytecode" column is the size of bytecode of reachable methods. The "KLOC" column is the number of thousands of line of Java source code of reachable classes. The "total" columns report numbers for *all* reachable code whereas the "app" columns report numbers for only application code (excluding JDK library code). The "# alloc. sites" column is the number of object allocation sites in reachable methods.

a backward meta-analysis, and denote them by $(\mathbb{P}, \preceq, \mathbb{D}, [\![-]\!])$ and $(\mathbb{M}, \gamma, [\![-]\!]^b, \mathsf{approx})$, respectively.

Our TRACER algorithm assumes that queries are expressed by elements ϕ in $\mathbb{M}$ satisfying the following condition:

$$\exists D_0 \subseteq \mathbb{D}.\, \gamma(\phi) = \mathbb{P} \times D_0 \wedge (\exists \phi' \in \mathbb{M}.\, \gamma(\phi') = \mathbb{P} \times (\mathbb{D} \setminus D_0)).$$

The first conjunct means that ϕ is independent of parameter values, and the second, that the negation of ϕ is expressible inside $\mathbb{M}$. We call ϕ satisfying these two conditions ***query***, and use a different symbol q to denote it. The negation of a query q is also a query, and we write $\mathsf{not}(q)$ to mean this negation.

TRACER takes as inputs initial abstract state d_I, a program s, and a query $q \in \mathbb{M}$. Given such inputs, TRACER repeatedly invokes the forward analysis with different parameter values, until it proves the query or finds that the forward analysis cannot prove the query no matter what parameter value is used. The most tricky part of TRACER is to choose a new parameter value π' to try after the forward analysis fails to prove the query using some π. TRACER does this parameter selection using the backward meta-analysis, which goes over an abstract counterexample trace of the forward analysis and computes a condition on parameter values necessary for proving the query. Among parameter values satisfying this necessary condition, TRACER chooses a minimum-cost π'.

The TRACER algorithm is shown in Algorithm 1. It uses the variable Π_{viable} to track parameter values that potentially lead to the proof of the query. Whenever TRACER calls the forward analysis, it picks a minimum π from Π_{viable}, and instantiates the forward analysis with π before running the analysis (lines 8-9). Also, whenever TRACER learns a necessary condition from the backward meta-analysis for proving a query (i.e., $\mathbb{P} \setminus \Pi$ in line 14), it conjoins the condition with Π_{viable} (line 15). In the description of the algorithm, we do not specify a particular way to choose an abstract counterexample trace τ from a failed run of the forward analysis. Such traces can be chosen by well-known techniques from software model checking [1, 16].

THEOREM 6. TRACER(d_I, s, q) *computes correct results, and it is guaranteed to terminate when* $\mathbb{P}$ *is finite.*

Proof See Appendix A.5 of our technical report [19].

Algorithm 1 TRACER(d_I, s, q): *iterative forward-backward analysis*

```
1: INPUTS: Initial abstract state $d_I$, program $s$, and query $q$
2: OUTPUTS: Minimum $\pi$ according to $\preceq$ such that
   $F_\pi[s](\{d_I\}) \subseteq \{d \mid (\pi, d) \in \gamma(q)\}$. Or impossibility
   meaning that $\nexists \pi : F_\pi[s](\{d_I\}) \subseteq \{d \mid (\pi, d) \in \gamma(q)\}$.
3: var $\Pi_{viable}$ := $\mathbb{P}$
4: while true do
5:   if $\Pi_{viable} = \emptyset$ then
6:     return impossible
7:   end if
8:   choose a minimum $\pi \in \Pi_{viable}$ according to $\preceq$
9:   let $D = (F_\pi[s](\{d_I\}) \cap \{d \mid (\pi, d) \in \gamma(\mathsf{not}(q))\})$ in
10:    if $D = \emptyset$ then
11:      return $\pi$
12:    end if
13:    choose any $\tau \in \mathsf{trace}(s) : F_\pi[\tau](d_I) \in D$
14:    let $\Pi = \{\pi' \mid (\pi', d_I) \in \gamma(B[\tau](\pi, d_I, \mathsf{not}(q)))\}$ in
15:      $\Pi_{viable}$ := $\Pi_{viable} \cap (\mathbb{P} \setminus \Pi)$
16:    end let
17:  end let
18: end while
```

6. Experiments

We implemented our approach for parametric static analyses of Java programs and evaluated it using our thread-escape analysis as a candidate analysis on a benchmark suite. We next describe our experimental setup and summarize our results. The source code of our implementation and detailed results are available at `http://pag-www.gtisc.gatech.edu/psa/`.

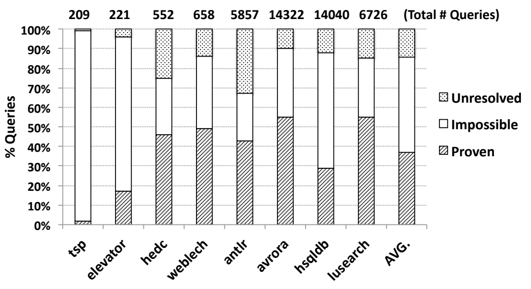
6.1 Experimental Setup

This section describes our generic implementation, a candidate analysis that instantiates it, and our benchmark suite.

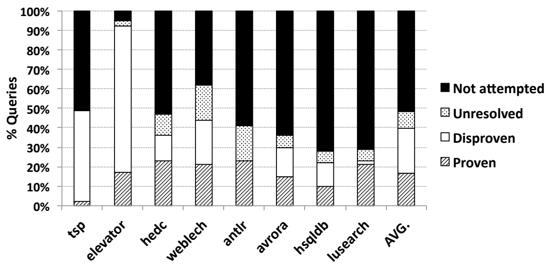
Implementation. We implemented our approach as a generic framework for Java bytecode. The forward analysis is expressed as an instance of the RHS tabulation framework [15] while the backward meta-analysis is expressed as an instance of a trace analysis framework that implements our proposed optimizations.

We presented our approach for a single query but in practice a client may have multiple queries in the same program. Our framework has the same effect as running our approach separately for each query but it uses a more efficient implementation: at any instant, it maintains a set of groups $\{G_1, ..., G_n\}$ of unresolved queries (i.e., queries that are neither proven nor shown impossible to prove). Two queries belong to the same group iff the sets of unviable parameter values computed so far for those queries are the same. All queries start in the same group with an empty set of unviable parameter values but split into different groups whenever different sets of unviable parameter values are computed for them. Finally, our implementation is capable of distributing this computation over a cluster, allowing different processes to handle different groups of queries in parallel.

Candidate Analysis. We implemented our thread-escape analysis in our framework. The set of possible parameter values to this



(a) Precision of current approach.



(b) Precision of previous approach [13].

Figure 13. Precision results of our current and previous approach.

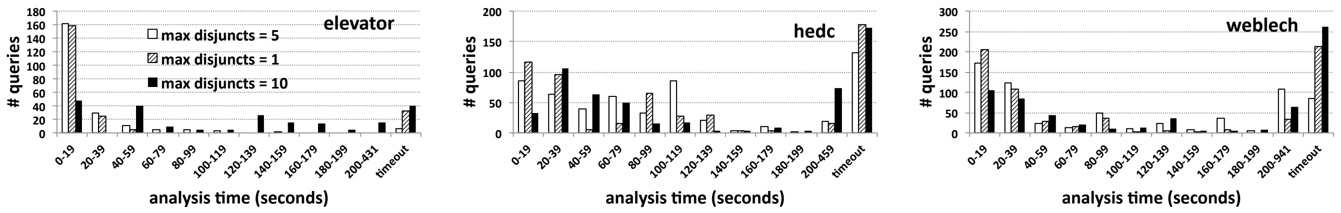


Figure 14. The running time of our approach for each query in three smaller benchmarks under three variants of underapproximation.

analysis is 2^N, where N is number of object allocation sites in the input program, since the parameter must set each site to L or E. A query is each pair (pc, v) such that the statement at program point pc in application code accesses (reads or writes) an instance field or an array element of the object denoted by variable v. We chose such queries for two reasons. First, they occur pervasively and uniformly in Java programs, and thereby enable a comprehensive evaluation of our approach. Second, these queries are posed by clients such as static race detection. Other clients may pose queries more selectively but our approach only stands to benefit in such cases by virtue of being query-guided.

We chose thread-escape analysis as it is challenging to scale and proving memory accesses thread-local is beneficial to a variety of concurrency analyses. Our earlier work [13] that also uses this client analysis shows that strawman solutions are ineffective: a flow-insensitive thread-escape analysis is highly imprecise while a trivial choice of the parameter value that sets all sites to L causes our analysis to run out of memory even on small programs.

Benchmark Suite. We evaluated our approach on a suite of eight real-world concurrent Java benchmarks comprising 2.5 MLOC. Table 1 shows characteristics of the benchmarks. They include four smaller ones to illustrate the performance impact of varying the extent of underapproximation in our meta-analysis; we were unable to run these variants on the four larger ones. All experiments were done using JDK 1.6 on Linux machines with 3.3 GHz processors and a maximum of 8GB memory per JVM process.

6.2 Evaluation Results

We now summarize our evaluation results, including precision, scalability, and useful statistics of proven queries.

Precision. Figure 13 (a) shows the precision of our approach. The absolute number of queries for each benchmark appears at the top. The queries are classified into three categories: those proven using a cheapest parameter value, those shown impossible to prove using any parameter value, and those that could not be resolved by our approach in 1000 minutes (we elaborate on these queries below).

Figure 13 (b) shows the precision of our old approach [13]. It involves performing a dynamic analysis on user-supplied inputs. A query might not be reached dynamically (denoted "Not attempted") in which case the query is unresolved; otherwise, the approach either observes the query to be false (denoted "Disproven"), or guesses a cheapest parameter value from dynamic information and uses it to attempt to prove the query. The attempt may succeed (denoted "Proven"), or fail (denoted "Unresolved") due to poor code coverage of the user-supplied inputs. From these two graphs, we find that on average per benchmark, the new approach proves 37.5% queries (versus 17.1% proven by the old approach) and it shows 49.1% queries impossible to prove (versus 23.9% observed to be false by the old approach), for a total of 86.6% resolved queries (versus 40.9% by the old approach). Thus, our new approach greatly outperforms the old approach. We contrast the two approaches further in Section 7.

We manually inspected several queries that were unresolved by our new approach, and found that all of them were true but impossible to prove using our thread-escape analysis, due to its limit of two abstract locations. There are two possible ways to address such queries depending on the desired goal: alter the backward meta-analysis to show impossibility more efficiently or alter the forward analysis to make the queries provable.

Scalability. It is challenging to scale backward meta-analyses. All our optimizations (underapproximation, double-negation, simplification by clause subsumption, and beam search) were motivated by experimentation: disabling *any* of them even on the smaller benchmarks made our approach timeout for *all* queries.

Our implementation allows to tune the amount of underapproximation by specifying the maximum number of disjuncts tracked in any DNF formula. Neither too aggressive nor too passive underapproximation is scalable: the former speeds up each iteration but increases the number of iterations whereas the latter has the opposite effect. Figure 14 affirms this by showing the total running time of our approach for each query in three representative smaller benchmarks (elevator, hedc, weblech) under three different choices of the maximum number of DNF disjuncts: 1, 5, and 10. The graphs show that 5 outperforms the extreme choices 1 and 10. More significantly, our approach timed out for *all* queries in the four larger benchmarks using the extreme choices; hence, we used a maximum of 5 disjuncts in all experiments.

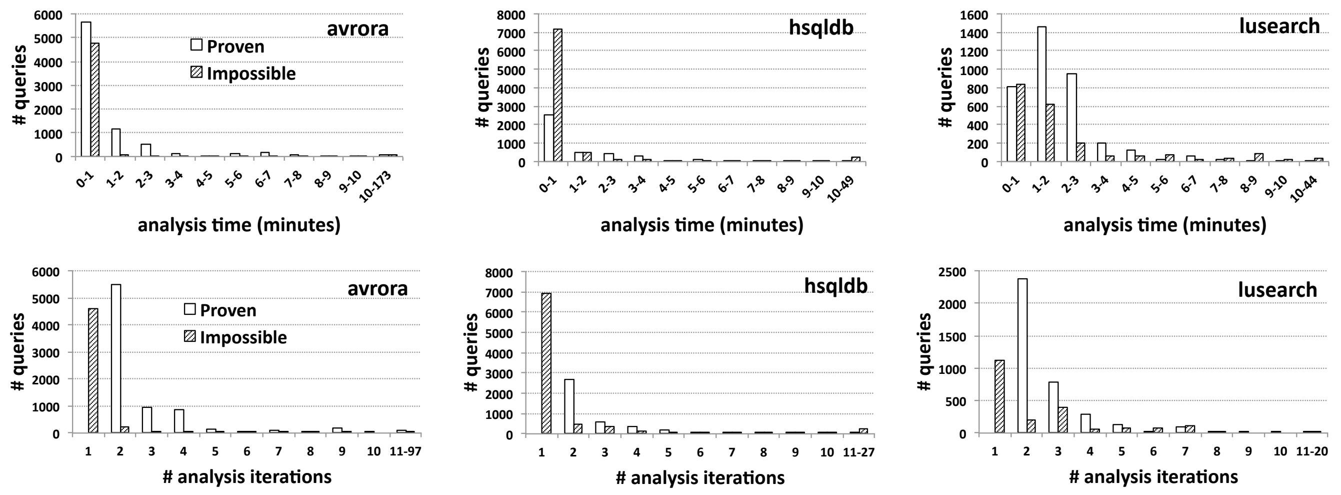


Figure 15. The running time of our approach for resolving each query in our three largest benchmarks. A query is resolved if it is proven or if it is shown impossible to prove. Graphs in the upper row show numbers of resolved queries grouped by the total running time in minutes of our approach. Graphs in the lower row show numbers of resolved queries grouped by the number of iterations of our approach.

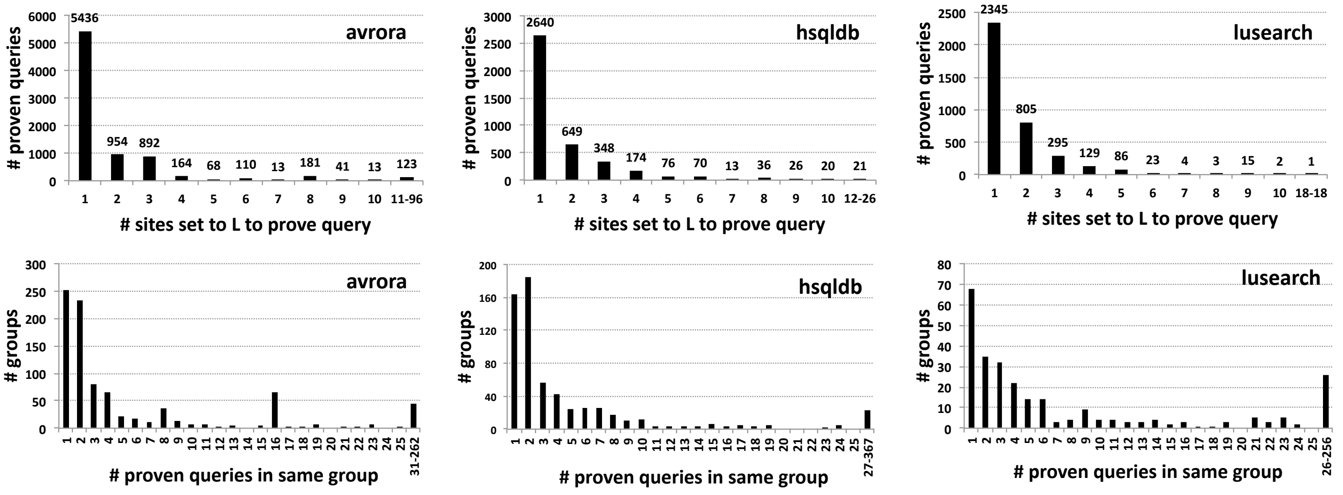


Figure 16. Statistics of the cheapest parameter values computed by our approach for proven queries. Graphs in the upper row show the size of the parameter value for each proven query. Graphs in the lower row show the number of queries proven using the same parameter value.

Figure 15 shows the total running time and the number of iterations our approach takes for each resolved query in our three largest benchmarks (unresolved queries are not shown since they are queries for which our approach timed out in 1000 minutes). We show different bars for queries that were proven and queries that were shown impossible to prove. The graphs highlight the scalability of our approach as the vast majority of queries are resolved in under two minutes each. They also reveal that our approach is very effective at finding queries that are impossible to prove as the vast majority of them are found in the first iteration.

Statistics of Proven Queries. We now present useful statistics about proven queries. The upper row of Figure 16 shows, for our three largest benchmarks, the size of the cheapest parameter value that our approach computed for each proven query. The size, plotted on the X axis, is the number of allocation sites set to L (recall that the fewer such sites the cheaper the analysis). The graphs show that the vast majority of queries require 1-2 such sites though we even prove queries that need up to 96 such sites.

Finally, it is natural to ask how different the cheapest parameter values computed by our approach are for these proven queries. The lower row of Figure 16 answers this question: each graph in the figure plots on the X axis the sizes of groups of proven queries for which the same cheapest parameter value was computed, and the number of groups of that size on the Y axis. These graphs show that most groups contain 1-2 queries, indicating that the cheapest parameter value is different for different queries, though there are also groups containing up to 367 queries.

These statistics underscore both the promise and the challenge of parametric static analysis: on one hand, most queries can be proven by instantiating the analysis using very inexpensive parameter values, but on the other hand, these parameter values tend to be very different for queries from different parts of the program.

7. Related Work

Our work is related to iterative refinement analyses but differs in the goal and the technique. They aim to find a cheap enough parameter value to prove a query while we aim to find a cheapest parameter value or show that none exists. We next contrast the techniques.

Many CEGAR-based model checkers compute a predicate abstraction and can be viewed as parametric in which program predicates to use for computing the abstraction. When the abstraction fails to prove a query, they analyze an abstract counterexample trace to compute an interpolant, which can be viewed as a minimal sufficient condition for the model checker to *succeed* in proving the query on the trace. In contrast, our meta-analysis computes a sufficient condition for the *failure* of the analysis to prove the query on the trace. One advantage of model checkers over our approach is that they can produce *concrete* counterexamples for false queries, whereas our approach can at best declare such queries impossible to prove using the given analysis. Conversely, our approach can declare when true queries are impossible to prove using the given analysis, whereas model checkers can diverge for such queries.

Refinement-based pointer analyses compute cause-effect dependencies for finding aspects of the abstraction that might be responsible for the failure to prove a query and then refine these aspects in the hope of proving it. These aspects include field reads and writes to be matched [18], methods or object allocation sites to be cloned [11, 14], or memory locations to be treated flow-sensitively [10]. A drawback of these analyses is that they can refine much more than necessary and thereby sacrifice scalability.

Combining forward and backward analysis has been proposed (e.g., [2]) but our approach differs in three key aspects. First, existing backward analyses are proven sound with respect to the program's concrete semantics, whereas ours is a *meta-analysis* that is proven sound with respect to the abstract semantics of the forward analysis. Second, existing backward analyses only track abstract states (to prune the over-approximation computed by the forward analysis), whereas ours also tracks parameter values. Finally, existing backward analyses may not scale due to tracking of program states that are unreachable from the initial state, whereas ours is guided by the abstract counterexample trace provided by the forward analysis, which also enables underapproximation.

Parametric static analysis is a search problem that may be tackled using various algorithms with different pros and cons. Liang et al. [12] propose iterative coarsening-based algorithms that start with the most precise parameter value (instead of the least precise one in the case of iterative refinement-based algorithms). Besides being impractical, these algorithms solve a different problem and cannot be adapted to ours: they find a *minimal* parameter value in terms of precision as opposed to a *minimum* or *cheapest* parameter value. Naik et al. [13] use dynamic analysis to infer a necessary condition on the parameter value to prove a query. They instantiate the parametric static analysis using a cheapest parameter value that satisfies this condition. However, there is no guarantee that it will prove the query, and the approach does not do refinement in case the analysis fails. In fact, the limitations of this approach motivated our current approach, particularly, its need for user-supplied program inputs, and its desire for those inputs to not induce long-running executions (for performance of the dynamic analysis) and to yield high code coverage (for strong necessary conditions). As our experiments showed, our current approach, which is fully static, proves 45.7% more queries on average per benchmark.

Finally, constraint-based and automated theorem proving techniques have been proposed that use search procedures similar in spirit to our approach: they too combine over-approximations and underapproximations, and compute strongest necessary and weakest sufficient conditions for proving queries (e.g, [3, 4, 8, 9]). A key difference is that none of these approaches address finding minimum-cost abstractions or proving impossibility results.

8. Conclusion

We presented a new approach to parametric static analysis with the goal of finding a cheapest parameter value that proves a given query or showing that no such parameter value exists. Our approach is CEGAR-based and applies a novel meta-analysis to abstract counterexample traces to efficiently eliminate unsuitable parameter values. We showed the generality of our approach by applying it to two parametric analyses in the literature. We also showed its effectiveness in practice by evaluating it on parametric thread-escape analysis for large real-world Java benchmarks.

Our approach exposed intriguing new problems. First, defining the transfer functions of the meta-analysis is tricky. One plausible solution is to devise a general recipe for synthesizing these functions automatically. Second, our approach requires the abstract domain of the parametric analysis to be disjunctive in order to be able to provide a counterexample trace to the meta-analysis. One possibility is to generalize our meta-analysis to operate on DAG counterexamples that have been proposed for non-disjunctive analyses [7]. Finally, the meta-analysis is a static analysis, and designing its abstract domain is an art. We proposed a DNF representation along with optimizations that were very effective in compacting the formulas tracked by the meta-analysis for our thread-escape analysis. It would be useful to devise a generic semantics-preserving simplification process to assist in compacting such formulas.

References

[1] T. Ball and S. K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *PASTE*, pages 97–103, 2001.

[2] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Autom. Softw. Eng.*, 6(1):69–95, 1999.

[3] I. Dillig, T. Dillig, and A. Aiken. Fluid updates: beyond strong vs. weak updates. In *ESOP'10*, .

[4] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI'08*, .

[5] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ISSTA'06*.

[6] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *CAV'97*.

[7] B. Gulavani, S. Chakraborty, A. Nori, and S. Rajamani. Automatically refining abstract interpretations. In *TACAS'08*.

[8] S. Gulwani and A. Tiwari. Assertion checking unified. In *VMCAI'07*.

[9] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL'08*.

[10] S. Guyer and C. Lin. Client-driven pointer analysis. In *SAS'03*.

[11] P. Liang and M. Naik. Scaling abstraction refinement via pruning. In *PLDI'11*.

[12] P. Liang, O. Tripp, and M. Naik. Learning minimal abstractions. In *POPL'11*.

[13] M. Naik, H. Yang, G. Castelnuovo, and M. Sagiv. Abstractions from tests. In *POPL'12*.

[14] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA'94*.

[15] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL'95*.

[16] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.

[17] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[18] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI'06*.

[19] X. Zhang, M. Naik, and H. Yang. Scalable parametric static analysis. `http://pag.gatech.edu/pubs/psa.pdf`, July 2012.

A. Proofs of Lemmas and Theorems

In this part of the appendix, we provide proofs of Lemmas and Theorems that are stated in the main text of the paper without proofs.

A.1 Proof of Lemma 1

LEMMA 1. *For all programs s, parameter values π, and abstract states d, we have that*

$$F_\pi[s](\{d\}) = \{F_\pi[\tau](d) \mid \tau \in \mathsf{trace}(s)\},$$

where $F_\pi[\tau]$ is the result of analyzing the trace τ as shown in Figure 4.

Proof We prove the following generalization of the lemma:

$$\begin{array}{l}\forall s.\, \forall \pi \in \mathbb{P}.\, \forall D \in 2^{\mathbb{D}}.\\ \qquad F_\pi[s](D) = \{F_\pi[\tau](d) \mid d \in D \wedge \tau \in \mathsf{trace}(s)\}.\end{array}$$

The proof is by induction on the structure of s. When s is an atomic command a,

$$F_\pi[s](D) = F_\pi[a](D) = \{[\![a]\!]_\pi(d) \mid d \in D\}.$$

Since $\mathsf{trace}(s) = \{a\}$, the equation above implies the lemma.

The next case is that $s = s_1 \,;\, s_2$. Let $D_1 = F_\pi[s_1](D)$ and $D_2 = F_\pi[s_2](D_1)$. By the induction hypothesis on s_1 and s_2, we have that

$$\begin{array}{l}D_1 = \{F_\pi[\tau_1](d) \mid \tau_1 \in \mathsf{trace}(s_1) \wedge d \in D\}\\ \wedge\, D_2 = \{F_\pi[\tau_2](d_1) \mid \tau_2 \in \mathsf{trace}(s_2) \wedge d_1 \in D_1\}.\end{array}$$

Hence, D_2 is

$$\begin{array}{l}\{F_\pi[\tau_2](F_\pi[\tau_1](d)) \mid \tau_2 \in \mathsf{trace}(s_2) \wedge \tau_1 \in \mathsf{trace}(s_1) \wedge d \in D\}\\ = \{F_\pi[\tau](d) \mid \tau \in \mathsf{trace}(s_1 \,;\, s_2) \wedge d \in D\}.\end{array}$$

We move on to the case that $s = s_1 + s_2$. In this case,

$$F_\pi[s](D) = F_\pi[s_1 + s_2](D) = F_\pi[s_1](D) \cup F_\pi[s_2](D).$$

By the induction hypothesis, the RHS of the above equation is the same as

$$\begin{array}{l}\{F_\pi[\tau](d) \mid \tau \in \mathsf{trace}(s_1) \wedge d \in D\}\\ \cup\, \{F_\pi[\tau](d) \mid \tau \in \mathsf{trace}(s_2) \wedge d \in D\},\end{array}$$

which equals

$$\begin{array}{l}\{F_\pi[\tau](d) \mid \tau \in \mathsf{trace}(s_1) \cup \mathsf{trace}(s_2) \wedge d \in D\}\\ = \{F_\pi[\tau](d) \mid \tau \in \mathsf{trace}(s_1 + s_2) \wedge d \in D\}.\end{array}$$

The remaining case is $s = s_1^*$. Let K and G be functions defined as follows:

$$\begin{array}{l}K = \lambda T.\, (\{\epsilon\} \cup \{\tau \,;\, \tau' \mid \tau \in T \wedge \tau' \in \mathsf{trace}(s_1)\}),\\ G = \lambda D'.\, (D \cup F_\pi[s_1](D')).\end{array}$$

The trace set $\mathsf{trace}(s_1^*)$ is the same as the least fixpoint of K, and $F_\pi[s_1^*](D)$ is defined by the least fixpoint of G. Let $\mathcal{R}$ be a relation on trace sets and sets of abstract states defined by

$$(T_1, D_1) \in \mathcal{R} \Leftrightarrow D_1 = \{F_\pi[\tau_1](d) \mid \tau_1 \in T_1 \wedge d \in D\}.$$

Meanwhile, we have that

$$\begin{array}{l}\{F_\pi[\tau](d) \mid \tau \in \mathsf{trace}(s_1^*) \wedge d \in D\}\\ = \{F_\pi[\tau](d) \mid \tau \in (\mathsf{leastFix}\, K) \wedge d \in D\}.\end{array}$$

Thus, it is sufficient to prove that $(\mathsf{leastFix}\, K, \mathsf{leastFix}\, G) \in \mathcal{R}$. This proof obligation can be further simplified because $\mathcal{R}$ is closed under arbitrary union: for every $\{(T_i, D_i)\}_{i \in I}$,

$$(\forall i \in I : (T_i, D_i) \in \mathcal{R}) \Rightarrow (\bigcup_{i \in I} T_i, \bigcup_{i \in I} D_i) \in \mathcal{R}.$$

This means that the least upper bounds of any two $\mathcal{R}$-related increasing sequences will be related by $\mathcal{R}$ as well. Hence, we can complete the proof of this loop case if we show that

$$\forall (T_1, D_1) \in \mathcal{R} : (K(T_1), G(D_1)) \in \mathcal{R}.$$

We will prove the simplified requirement. Pick $(T_1, D_1) \in \mathcal{R}$. We should show that

$$\{F_\pi[\tau_1](d) \mid \tau_1 \in K(T_1) \wedge d \in D\} = G(D_1).$$

We derive this desired equality as follows:

$$\begin{array}{l}\{F_\pi[\tau_1](d) \mid \tau_1 \in K(T_1) \wedge d \in D\}\\ = \{F_\pi[\tau_1](d) \mid d \in D \wedge (\tau_1 = \epsilon \vee (\tau_1 = \tau \,;\, \tau' \wedge \tau \in T_1\\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge\, \tau' \in \mathsf{trace}(s_1)))\}\\ = D \cup \{(F_\pi[\tau'] \circ F_\pi[\tau])(d) \mid\\ \qquad\qquad \tau \in T_1 \wedge \tau' \in \mathsf{trace}(s_1) \wedge d \in D\}\\ = D \cup \{F_\pi[\tau'](d') \mid \tau' \in \mathsf{trace}(s_1)\\ \qquad\qquad \wedge\, d' \in \{F_\pi[\tau](d) \mid \tau \in T_1 \wedge d \in D\}\}\\ = D \cup \{F_\pi[\tau'](d') \mid \tau' \in \mathsf{trace}(s_1) \wedge d' \in D_1\}\\ = D \cup F_\pi[s_1](D_1).\\ = G(D_1).\end{array}$$

The fourth equality comes from the assumption that $(T_1, D_1) \in \mathcal{R}$. The fifth equality holds because of the induction hypothesis. □

A.2 Proof of Theorem 3

THEOREM 3 (Soundness). *For all τ, π, d and $\phi \in \mathbb{M}$,*

1. $(\pi, F_\pi[\tau](d)) \in \gamma(\phi) \Rightarrow (\pi, d) \in \gamma(B[\tau](\pi, d, \phi))$*; and*
2. $\forall (\pi_0, d_0) \in \gamma(B[\tau](\pi, d, \phi)).\ (\pi_0, F_{\pi_0}[\tau](d_0)) \in \gamma(\phi)$.

Proof We prove the theorem by structural induction on τ. Pick arbitrary π, d, ϕ. We will prove the two items of this theorem for each case of τ.

When τ is ϵ, ϕ and $B[\tau](\pi, d, \phi)$ are the same, and $F_{\pi'}[\tau] = F_{\pi'}[\epsilon]$ is the identity for every π'. The claim of the theorem follows from these facts.

The next case is that τ is an atomic command a. In this case, $B[\tau](\pi', d', \phi) = \mathsf{approx}(\pi', d', [\![a]\!]^b(\phi))$ and $F_{\pi'}[\tau](d') = [\![a]\!]_{\pi'}(d')$ for all π', d'. Hence, the first item of the theorem can be proved as follows:

$$\begin{array}{rl}F_\pi[\tau](d) \in \gamma(\phi) & \Rightarrow [\![a]\!]_\pi(d) \in \gamma(\phi)\\ & \Rightarrow (\pi, d) \in \gamma([\![a]\!]^b(\phi))\\ & \Rightarrow (\pi, d) \in \gamma(\mathsf{approx}(\pi, d, [\![a]\!]^b(\phi)))\end{array}$$

The second and third implications come from the requirement on $[\![a]\!]^b$ and approx in our framework, respectively. For the second item of the theorem, pick $(\pi_0, d_0) \in \gamma(B[\tau](\pi, d, \phi))$. Then,

$$\begin{array}{rl}(\pi_0, d_0) \in \gamma(B[a](\pi, d, \phi)) & \Rightarrow (\pi_0, d_0) \in \gamma(\mathsf{approx}(\pi, d, [\![a]\!]^b(\phi))\\ & \Rightarrow (\pi_0, d_0) \in \gamma([\![a]\!]^b(\phi))\\ & \Rightarrow (\pi_0, [\![a]\!]_{\pi_0}(d_0)) \in \gamma(\phi).\end{array}$$

The first implication is just the unrolling of the definition of $B[a]$, the second comes from our requirement that approx does underapproximation, and the last holds because of the requirement on $[\![a]\!]^b$ in our framework. The conclusion of the above derivation implies the second item of this theorem.

The remaining case is that τ is $\tau_1 \,;\, \tau_2$. Let

$$d' = F_\pi[\tau_1](d) \ \text{ and } \ \phi' = B[\tau_2](\pi, d', \phi).$$

To prove the first item of the theorem, assume that $(\pi, F_\pi[\tau](d))$ belongs to $\gamma(\phi)$. Then,

$$(\pi, F_\pi[\tau_2](d')) = (\pi, F_\pi[\tau_2](F_\pi[\tau_1](d))) \in \gamma(\phi).$$

We apply the induction hypothesis on τ_2 here, and obtain

$$(\pi, d') \in \gamma(B[\tau_2](\pi, d', \phi)).$$

The LHS in this membership is the same as $(\pi, F_\pi[\tau_1](d))$, and the RHS equals $\gamma(\phi')$. This means that we can apply the induction hypothesis again, this time on τ_1, and get

$$(\pi, d) \in \gamma(B[\tau_1](\pi, d, \phi'))$$

But the RHS here is the same as

$$\gamma(B[\tau_1](\pi, d, B[\tau_2](\pi, d', \phi)) = \gamma(B[\tau](\pi, d, \phi)).$$

Hence, the first item of this theorem holds. Let's move on to the proof of the second item of this theorem. Consider $(\pi_0, d_0) \in \gamma(B[\tau](\pi, d, \phi))$. We derive the conclusion of the second item of the theorem as follows:

$$\begin{array}{l}(\pi_0, d_0) \in \gamma(B[\tau](\pi, d, \phi)) \\ \Rightarrow (\pi_0, d_0) \in \gamma(B[\tau_1](\pi, d, \phi')) \\ \Rightarrow (\pi_0, F_{\pi_0}[\tau_1](d_0)) \in \gamma(\phi') \\ \Rightarrow (\pi_0, F_{\pi_0}[\tau_1](d_0)) \in \gamma(B[\tau_2](\pi, d', \phi)) \\ \Rightarrow (\pi_0, F_{\pi_0}[\tau_2](F_{\pi_0}[\tau_1](d_0))) \in \gamma(\phi) \\ \Rightarrow (\pi_0, F_{\pi_0}[\tau](d_0)) \in \gamma(\phi)\end{array}$$

The first implication holds because of the definition of $B[\tau_1 ; \tau_2]$, the second and fourth implications use induction hypothesis, the third is the simple unrolling of the definition of ϕ', and the last implication follows from the definition of $F_{\pi_0}[\tau_1 ; \tau_2]$. □

A.3 Proof of Lemma 4

LEMMA 4. *For every atomic command a, the backward transfer function $[\![a]\!]^b$ in Figure 10 satisfies requirement* (4) *in Section 4.*

Proof For every atomic command a, define a function $W[a]$ on $\mathbb{M}$ by

$$W[a](\phi) = \{(\pi, d) \mid (\pi, [\![a]\!]_\pi(d)) \in \gamma(\phi)\}.$$

We need to prove that for all ϕ,

$$\forall a.\ \gamma([\![a]\!]^b(\phi)) = W[a](\phi). \qquad (1)$$

Our proof is done by induction on the structure of ϕ.

We start by considering the cases that ϕ is true, conjunction, false or disjunction. The reason that the equality in (1) holds for these cases is that $W[a]$ preserve all of true, conjunction, false and disjunction:

$$\begin{array}{ll} W[a](\mathsf{true}) = \mathbb{P} \times \mathbb{S}, & W[a](\phi \wedge \phi') = W[a](\phi) \cap W[a](\phi'), \\ W[a](\mathsf{false}) = \emptyset, & W[a](\phi \vee \phi') = W[a](\phi) \cup W[a](\phi'). \end{array}$$

The desired equality follows from this preservation and the induction hypothesis.

The remaining case is that $\phi = \delta{\to}o$ for some $\delta \in \mathbb{H}\cup\mathbb{L}\cup\mathbb{F}$ and $o \in \{\mathtt{L}, \mathtt{E}, \mathtt{N}\}$. This is handled by the case analysis on the atomic command a.

The first case is that a is $g = v$. We further subdivide this case such that (1) $\delta \equiv h$, (2) $\delta \in (\mathbb{L} \cup \mathbb{F} \setminus \{v\})$ and (3) $\delta \equiv v$. The first subcase is handled below:

$$(\pi, [\![g = v]\!]_\pi(d)) \in \gamma(h{\to}o) \;\Leftrightarrow\; \pi(h) = o \;\Leftrightarrow\; (\pi, d) \in \gamma(h{\to}o).$$

The second subcase can be proven as follows:

$$\begin{array}{l}(\pi, [\![g = v]\!]_\pi(d)) \in \gamma(\delta{\to}o) \\ \Leftrightarrow \\ (d(v) = \mathtt{L} \wedge \mathsf{esc}(d)(\delta) = o) \vee (d(v) \in \{\mathtt{E}, \mathtt{N}\} \wedge d(\delta) = o) \\ \Leftrightarrow \\ (\delta \in \mathbb{L} \wedge d(v) = \mathtt{L} \wedge d(\delta) = \mathtt{L} \wedge o = \mathtt{E}) \\ \;\vee (\delta \in \mathbb{L} \wedge d(v) = \mathtt{L} \wedge d(\delta) \in \{\mathtt{E}, \mathtt{N}\} \wedge o = d(\delta)) \\ \;\vee (\delta \in \mathbb{F} \wedge d(v) = \mathtt{L} \wedge o = \mathtt{N}) \\ \;\vee (d(v) \in \{\mathtt{E}, \mathtt{N}\} \wedge d(\delta) = o) \\ \Leftrightarrow \\ (\delta \in \mathbb{L} \wedge o = \mathtt{N} \wedge d(\delta) = \mathtt{N}) \\ \;\vee (\delta \in \mathbb{L} \wedge o = \mathtt{E} \wedge ((d(v) = \mathtt{L} \wedge d(\delta) = \mathtt{L}) \vee d(\delta) = \mathtt{E})) \\ \;\vee (\delta \in \mathbb{L} \wedge o = \mathtt{L} \wedge d(v) \in \{\mathtt{E}, \mathtt{N}\} \wedge d(\delta) = \mathtt{L}) \\ \;\vee (\delta \in \mathbb{F} \wedge o = \mathtt{N} \wedge (d(v) = \mathtt{L} \vee (d(v) \in \{\mathtt{E}, \mathtt{N}\} \wedge d(\delta) = \mathtt{N}))) \\ \;\vee (\delta \in \mathbb{F} \wedge o \in \{\mathtt{E}, \mathtt{L}\} \wedge d(v) \in \{\mathtt{E}, \mathtt{N}\} \wedge d(\delta) = o) \\ \Leftrightarrow \\ (\delta \in \mathbb{L} \wedge o = \mathtt{N} \wedge (\pi, d) \in \gamma(\delta{\to}\mathtt{N})) \\ \;\vee (\delta \in \mathbb{L} \wedge o = \mathtt{E} \wedge (\pi, d) \in \gamma((v{\to}\mathtt{L} \wedge \delta{\to}\mathtt{L}) \vee \delta{\to}\mathtt{E})) \\ \;\vee (\delta \in \mathbb{L} \wedge o = \mathtt{L} \wedge (\pi, d) \in \gamma((v{\to}\mathtt{E} \vee v{\to}\mathtt{N}) \wedge \delta{\to}\mathtt{L})) \\ \;\vee (\delta \in \mathbb{F} \wedge o = \mathtt{N} \wedge (\pi, d) \in \gamma(v{\to}\mathtt{L} \vee (v{\to}\mathtt{E} \vee v{\to}\mathtt{N}) \wedge \delta{\to}\mathtt{N})) \\ \;\vee (\delta \in \mathbb{F} \wedge o \in \{\mathtt{E}, \mathtt{L}\} \wedge (\pi, d) \in \gamma((v{\to}\mathtt{E} \vee v{\to}\mathtt{N}) \wedge \delta{\to}o)).\end{array}$$

The third subcase can be proven as follows:

$$\begin{array}{l}(\pi, [\![g = v]\!]_\pi(d)) \in \gamma(v{\to}o) \\ \Leftrightarrow \\ (d(v) = \mathtt{L} \wedge \mathsf{esc}(d)(v) = o) \vee (d(v) = o \wedge o \in \{\mathtt{E}, \mathtt{N}\}) \\ \Leftrightarrow \\ (d(v) = \mathtt{L} \wedge o = \mathtt{E}) \vee (d(v) = o \wedge o \in \{\mathtt{E}, \mathtt{N}\}) \\ \Leftrightarrow \\ (o = \mathtt{E} \wedge (\pi, d) \in \gamma(v{\to}\mathtt{L} \vee v{\to}\mathtt{E})) \vee (o = \mathtt{N} \wedge (\pi, d) \in \gamma(v{\to}\mathtt{N})).\end{array}$$

The calculations for the three subcases above imply that the claimed equality holds for $g = v$.

The second case is $v = g$. In this case, we can show the lemma as follows:

$$\begin{array}{l}(\pi, [\![v = g]\!]_\pi(d)) \in \gamma(\delta{\to}o) \\ \Leftrightarrow \\ (\pi, d[v : \mathtt{E}]) \in \gamma(\delta{\to}o) \\ \Leftrightarrow \\ (\delta \equiv v \wedge o = \mathtt{E}) \vee (\delta \in (\mathbb{L} \cup \mathbb{F}) \setminus \{v\} \wedge d(\delta) = o) \\ \quad \vee (\delta \in \mathbb{H} \wedge \pi(\delta) = o) \\ \Leftrightarrow \\ (\delta \equiv v \wedge o = \mathtt{E} \wedge (\pi, d) \in \gamma(\mathsf{true})) \vee (\delta \not\equiv v \wedge (\pi, d) \in \gamma(\delta{\to}o)).\end{array}$$

The third case is that a is $v = \mathsf{new}\, h$. The proof of this case is given below:

$$\begin{array}{l}(\pi, [\![v = \mathsf{new}\, h]\!]_\pi(d)) \in \gamma(\delta{\to}o) \\ \Leftrightarrow \\ (\pi, d[v : \pi(h)]) \in \gamma(\delta{\to}o) \\ \Leftrightarrow \\ (\delta \equiv v \wedge \pi(h) = o) \vee (\delta \in (\mathbb{L} \cup \mathbb{F}) \setminus \{v\} \wedge d(\delta) = o) \\ \quad \vee (\delta \in \mathbb{H} \wedge \pi(\delta) = o) \\ \Leftrightarrow \\ (\delta \equiv v \wedge (\pi, d) \in \gamma(h{\to}o)) \vee (\delta \not\equiv v \wedge (\pi, d) \in \gamma(\delta{\to}o)).\end{array}$$

The fourth case is that a is $v = \mathtt{null}$. We prove the lemma in the case as follows:

$$\begin{array}{l}(\pi, [\![v = \mathtt{null}]\!]_\pi(d)) \in \gamma(\delta{\to}o) \\ \Leftrightarrow \\ (\pi, d[v : \mathtt{N}]) \in \gamma(\delta{\to}o) \\ \Leftrightarrow \\ (\delta \equiv v \wedge \mathtt{N} = o) \vee (\delta \in (\mathbb{L} \cup \mathbb{F}) \setminus \{v\} \wedge d(\delta) = o) \\ \quad \vee (\delta \in \mathbb{H} \wedge \pi(\delta) = o) \\ \Leftrightarrow \\ (\delta \equiv v \wedge o = \mathtt{N} \wedge (\pi, d) \in \gamma(\mathsf{true})) \vee (\delta \not\equiv v \wedge (\pi, d) \in \gamma(\delta{\to}o)).\end{array}$$

The fifth case is that a is $v = v'$. We prove the lemma in the case as follows:

$$\begin{array}{l}(\pi, [\![v = v']\!]_\pi(d)) \in \gamma(\delta{\to}o) \\ \Leftrightarrow \\ (\pi, d[v : d(v')]) \in \gamma(\delta{\to}o) \\ \Leftrightarrow \\ (\delta \equiv v \wedge d(v') = o) \vee (\delta \in (\mathbb{L} \cup \mathbb{F}) \setminus \{v\} \wedge d(\delta) = o) \\ \quad \vee (\delta \in \mathbb{H} \wedge \pi(\delta) = o) \\ \Leftrightarrow \\ (\delta \equiv v \wedge (\pi, d) \in \gamma(v'{\to}o)) \vee (\delta \not\equiv v \wedge (\pi, d) \in \gamma(\delta{\to}o))\end{array}$$

The sixth case is that a is $v = v'.f$. We prove the case below:

$$
\begin{array}{l}
(\pi, \llbracket v = v'.f \rrbracket_\pi(d)) \in \gamma(\delta{\to}o) \\
\Leftrightarrow \\
(d(v') = \mathtt{L} \wedge (\pi, d[v : d(f)]) \in \gamma(\delta{\to}o)) \\
\quad \vee (d(v') \neq \mathtt{L} \wedge (\pi, d[v : \mathtt{E}]) \in \gamma(\delta{\to}o)) \\
\Leftrightarrow \\
(\delta \equiv v \wedge ((d(v') = \mathtt{L} \wedge d(f) = o) \vee (d(v') \neq \mathtt{L} \wedge o = \mathtt{E}))) \\
\quad \vee (\delta \not\equiv v \wedge d(v') = \mathtt{L} \wedge (\pi, d) \in \gamma(\delta{\to}o)) \\
\quad \vee (\delta \not\equiv v \wedge d(v') \neq \mathtt{L} \wedge (\pi, d) \in \gamma(\delta{\to}o)) \\
\Leftrightarrow \\
(\delta \equiv v \wedge o = \mathtt{E} \wedge ((d(v') = \mathtt{L} \wedge d(f) = \mathtt{E}) \vee d(v') \neq \mathtt{L})) \\
\quad \vee (\delta \equiv v \wedge o \neq \mathtt{E} \wedge d(v') = \mathtt{L} \wedge d(f) = o) \\
\quad \vee (\delta \not\equiv v \wedge (\pi, d) \in \gamma(\delta{\to}o)) \\
\Leftrightarrow \\
(\delta \equiv v \wedge o = \mathtt{E} \wedge (\pi, d) \in \gamma((v'{\to}\mathtt{L} \wedge f{\to}\mathtt{E}) \vee v'{\to}\mathtt{E} \vee v'{\to}\mathtt{N})) \\
\quad \vee (\delta \equiv v \wedge o \neq \mathtt{E} \wedge (\pi, d) \in \gamma(v'{\to}\mathtt{L} \wedge f{\to}o)) \\
\quad \vee (\delta \not\equiv v \wedge (\pi, d) \in \gamma(\delta{\to}o)).
\end{array}
$$

The last case is that a is $v.f = v'$. The following derivation shows the lemma for this case:

$$
\begin{array}{l}
(\pi, \llbracket v.f = v' \rrbracket_\pi(d)) \in \gamma(\delta{\to}o) \\
\Leftrightarrow \\
(d(v) = \mathtt{E} \wedge d(v') = \mathtt{L} \wedge (\pi, \mathsf{esc}(d)) \in \gamma(\delta{\to}o)) \\
\quad \vee (((d(v) = \mathtt{E} \wedge d(v') \neq \mathtt{L}) \vee d(v) = \mathtt{N}) \wedge (\pi, d) \in \gamma(\delta{\to}o)) \\
\quad \vee (d(v) = \mathtt{L} \wedge \{d(f), d(v')\} = \{\mathtt{L}, \mathtt{E}\} \wedge (\pi, \mathsf{esc}(d)) \in \gamma(\delta{\to}o)) \\
\quad \vee (d(v) = \mathtt{L} \wedge \{d(f), d(v')\} = \{\mathtt{L}, \mathtt{N}\} \wedge (\pi, d[f : \mathtt{L}]) \in \gamma(\delta{\to}o)) \\
\quad \vee (d(v) = \mathtt{L} \wedge \{d(f), d(v')\} = \{\mathtt{E}, \mathtt{N}\} \wedge (\pi, d[f : \mathtt{E}]) \in \gamma(\delta{\to}o)) \\
\quad \vee (d(v) = \mathtt{L} \wedge d(f) = d(v') \wedge (\pi, d) \in \gamma(\delta{\to}o)) \\
\Leftrightarrow \\
(\delta \not\in (\mathbb{L} \cup \mathbb{F}) \wedge (\pi, d) \in \gamma(v{\to}\mathtt{E} \wedge v'{\to}\mathtt{L} \wedge \delta{\to}o)) \\
\quad \vee (\delta \in \mathbb{L} \wedge o = \mathtt{E} \wedge (\pi, d) \in \gamma(v{\to}\mathtt{E} \wedge v'{\to}\mathtt{L} \wedge (\delta{\to}\mathtt{L} \vee \delta{\to}\mathtt{E}))) \\
\quad \vee (\delta \in \mathbb{L} \wedge o = \mathtt{N} \wedge (\pi, d) \in \gamma(v{\to}\mathtt{E} \wedge v'{\to}\mathtt{L} \wedge \delta{\to}o)) \\
\quad \vee (\delta \in \mathbb{F} \wedge o = \mathtt{N} \wedge (\pi, d) \in \gamma(v{\to}\mathtt{E} \wedge v'{\to}\mathtt{L})) \\
\quad \vee ((\pi, d) \in \gamma(((v{\to}\mathtt{E} \wedge (v'{\to}\mathtt{E} \vee v'{\to}\mathtt{N})) \vee v{\to}\mathtt{N}) \wedge \delta{\to}o)) \\
\quad \vee (\delta \not\in (\mathbb{L} \cup \mathbb{F}) \wedge (\pi, d) \in \gamma((f{\to}\mathtt{L} \wedge v'{\to}\mathtt{E} \vee f{\to}\mathtt{E} \wedge v'{\to}\mathtt{L}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge v{\to}\mathtt{L} \wedge \delta{\to}o)) \\
\quad \vee (\delta \in \mathbb{L} \wedge o = \mathtt{E} \wedge (\pi, d) \in \gamma((f{\to}\mathtt{L} \wedge v'{\to}\mathtt{E} \vee f{\to}\mathtt{E} \wedge v'{\to}\mathtt{L}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge v{\to}\mathtt{L} \wedge (\delta{\to}\mathtt{L} \vee \delta{\to}\mathtt{E}))) \\
\quad \vee (\delta \in \mathbb{L} \wedge o = \mathtt{N} \wedge (\pi, d) \in \gamma((f{\to}\mathtt{L} \wedge v'{\to}\mathtt{E} \vee f{\to}\mathtt{E} \wedge v'{\to}\mathtt{L}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge v{\to}\mathtt{L} \wedge \delta{\to}o)) \\
\quad \vee (\delta \in \mathbb{F} \wedge o = \mathtt{N} \wedge (\pi, d) \in \gamma((f{\to}\mathtt{L} \wedge v'{\to}\mathtt{E} \vee f{\to}\mathtt{E} \wedge v'{\to}\mathtt{L}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge v{\to}\mathtt{L})) \\
\quad \vee (\delta \not\equiv f \wedge (\pi, d) \in \gamma((f{\to}\mathtt{L} \wedge v'{\to}\mathtt{N} \vee f{\to}\mathtt{N} \wedge v'{\to}\mathtt{L}) \\
\qquad\qquad\qquad\qquad \wedge v{\to}\mathtt{L} \wedge \delta{\to}o)) \\
\quad \vee (\delta \equiv f \wedge o = \mathtt{L} \wedge (\pi, d) \in \gamma((f{\to}\mathtt{L} \wedge v'{\to}\mathtt{N} \vee f{\to}\mathtt{N} \wedge v'{\to}\mathtt{L}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge v{\to}\mathtt{L})) \\
\quad \vee (\delta \not\equiv f \wedge (\pi, d) \in \gamma((f{\to}\mathtt{E} \wedge v'{\to}\mathtt{N} \vee f{\to}\mathtt{N} \wedge v'{\to}\mathtt{E}) \\
\qquad\qquad\qquad\qquad \wedge v{\to}\mathtt{L} \wedge \delta{\to}o)) \\
\quad \vee (\delta \equiv f \wedge o = \mathtt{E} \wedge (\pi, d) \in \gamma((f{\to}\mathtt{E} \wedge v'{\to}\mathtt{N} \vee f{\to}\mathtt{N} \wedge v'{\to}\mathtt{E}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge v{\to}\mathtt{L})) \\
\quad \vee (\pi, d) \in \gamma((f{\to}\mathtt{L} \wedge v'{\to}\mathtt{L} \vee f{\to}\mathtt{E} \wedge v'{\to}\mathtt{E} \vee f{\to}\mathtt{N} \wedge v'{\to}\mathtt{N}) \\
\qquad\qquad\qquad \wedge v{\to}\mathtt{L} \wedge \delta{\to}o) \\
\Leftrightarrow \\
(\delta \not\in (\mathbb{L} \cup \mathbb{F}) \wedge (\pi, d) \in \gamma(\delta{\to}o \wedge (v{\to}\mathtt{E} \wedge v'{\to}\mathtt{L} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{L} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{E} \wedge v'{\to}\mathtt{L}))) \\
\quad \vee (\delta \in \mathbb{L} \wedge o = \mathtt{E} \wedge (\pi, d) \in \gamma((\delta{\to}\mathtt{L} \vee \delta{\to}o) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge (v{\to}\mathtt{E} \wedge v'{\to}\mathtt{L} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{L} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{E} \wedge v'{\to}\mathtt{L})))
\end{array}
$$

$$
\begin{array}{l}
\vee (\delta \in \mathbb{L} \wedge o = \mathtt{N} \wedge (\pi, d) \in \gamma(\delta{\to}o \wedge (v{\to}\mathtt{E} \wedge v'{\to}\mathtt{L} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{L} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{E} \wedge v'{\to}\mathtt{L}))) \\
\vee (\delta \in \mathbb{F} \wedge o = \mathtt{N} \wedge (\pi, d) \in \gamma(v{\to}\mathtt{E} \wedge v'{\to}\mathtt{L} \\
\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{L} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{E} \wedge v'{\to}\mathtt{L})) \\
\vee (\delta \not\equiv f \wedge (\pi, d) \in \gamma(v{\to}\mathtt{L} \wedge \delta{\to}o \wedge (f{\to}\mathtt{L} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee f{\to}\mathtt{N} \wedge v'{\to}\mathtt{L} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee f{\to}\mathtt{E} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee f{\to}\mathtt{N} \wedge v'{\to}\mathtt{E}))) \\
\vee (\delta \equiv f \wedge o = \mathtt{L} \wedge (\pi, d) \in \gamma(v{\to}\mathtt{L} \wedge (f{\to}\mathtt{L} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee f{\to}\mathtt{N} \wedge v'{\to}\mathtt{L}))) \\
\vee (\delta \equiv f \wedge o = \mathtt{E} \wedge (\pi, d) \in \gamma(v{\to}\mathtt{L} \wedge (f{\to}\mathtt{E} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee f{\to}\mathtt{N} \wedge v'{\to}\mathtt{E}))) \\
\vee (\pi, d) \in \gamma(\delta{\to}o \wedge (v{\to}\mathtt{L} \wedge f{\to}\mathtt{L} \wedge v'{\to}\mathtt{L} \\
\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{E} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{N} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{E} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{E} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{N})) \\
\Leftrightarrow \\
(\delta \not\in (\mathbb{L} \cup \mathbb{F}) \wedge (\pi, d) \in \gamma(\delta{\to}o)) \\
\vee (\delta \in \mathbb{L} \wedge o = \mathtt{E} \wedge (\pi, d) \in \gamma(\delta{\to}o \\
\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{L} \wedge v{\to}\mathtt{E} \wedge v'{\to}\mathtt{L} \\
\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{L} \wedge v{\to}\mathtt{L} \wedge f{\to}\mathtt{L} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{L} \wedge v{\to}\mathtt{L} \wedge f{\to}\mathtt{E} \wedge v'{\to}\mathtt{L})) \\
\vee (\delta \in \mathbb{L} \wedge o = \mathtt{N} \wedge (\pi, d) \in \gamma(\delta{\to}o)) \\
\vee (\delta \in \mathbb{L} \wedge o = \mathtt{L} \wedge (\pi, d) \in \gamma(\delta{\to}o \wedge v{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}o \wedge v{\to}\mathtt{E} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}o \wedge v{\to}\mathtt{E} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}o \wedge v{\to}\mathtt{L} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}o \wedge v{\to}\mathtt{L} \wedge f{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}o \wedge v{\to}\mathtt{L} \wedge v'{\to}\mathtt{L} \wedge f{\to}\mathtt{L} \\
\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}o \wedge v{\to}\mathtt{L} \wedge v'{\to}\mathtt{E} \wedge f{\to}\mathtt{E})) \\
\vee (\delta \in \mathbb{F} \wedge o = \mathtt{N} \wedge \delta \not\equiv f \wedge (\pi, d) \in \gamma(\delta{\to}o \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{E} \wedge v'{\to}\mathtt{L} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{L} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{E} \wedge v'{\to}\mathtt{L})) \\
\vee (\delta \in \mathbb{F} \wedge (a = \mathtt{E} \vee o = \mathtt{L}) \wedge \delta \not\equiv f \\
\qquad \wedge (\pi, d) \in \gamma(\delta{\to}o \wedge v{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad \vee \delta{\to}o \wedge v{\to}\mathtt{E} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad \vee \delta{\to}o \wedge v{\to}\mathtt{E} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad \vee \delta{\to}o \wedge v{\to}\mathtt{L} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad \vee \delta{\to}o \wedge v{\to}\mathtt{L} \wedge f{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad \vee \delta{\to}o \wedge v{\to}\mathtt{L} \wedge f{\to}\mathtt{L} \wedge v'{\to}\mathtt{L} \\
\qquad\qquad\qquad\qquad \vee \delta{\to}o \wedge v{\to}\mathtt{L} \wedge f{\to}\mathtt{E} \wedge v'{\to}\mathtt{E})) \\
\vee (\delta \in \mathbb{F} \wedge o = \mathtt{N} \wedge \delta \equiv f \wedge (\pi, d) \in \gamma(f{\to}\mathtt{N} \wedge v{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee f{\to}\mathtt{N} \wedge v{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee f{\to}\mathtt{N} \wedge v{\to}\mathtt{L} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{E} \wedge v'{\to}\mathtt{L} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{L} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{E} \wedge v'{\to}\mathtt{L})) \\
\vee (\delta \in \mathbb{F} \wedge o = \mathtt{L} \wedge \delta \equiv f \wedge (\pi, d) \in \gamma(f{\to}\mathtt{L} \wedge v{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee f{\to}\mathtt{L} \wedge v{\to}\mathtt{E} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee f{\to}\mathtt{L} \wedge v{\to}\mathtt{E} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{L} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{N} \wedge v'{\to}\mathtt{L} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{L} \wedge v'{\to}\mathtt{L})) \\
\vee (\delta \in \mathbb{F} \wedge o = \mathtt{E} \wedge \delta \equiv f \wedge (\pi, d) \in \gamma(f{\to}\mathtt{E} \wedge v{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee f{\to}\mathtt{E} \wedge v{\to}\mathtt{E} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee f{\to}\mathtt{E} \wedge v{\to}\mathtt{E} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{E} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{N} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{E} \wedge v'{\to}\mathtt{E}))
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \\
(\delta \notin (\mathbb{L} \cup \mathbb{F}) \wedge (\pi, d) \in \gamma(\delta{\to}o)) \\
\vee (\delta \in \mathbb{L} \wedge o = \mathtt{E} \wedge (\pi, d) \in \gamma(\delta{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{L} \wedge v{\to}\mathtt{E} \wedge v'{\to}\mathtt{L} \\
\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{L} \wedge v{\to}\mathtt{L} \wedge f{\to}\mathtt{L} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{L} \wedge v{\to}\mathtt{L} \wedge f{\to}\mathtt{E} \wedge v'{\to}\mathtt{L})) \\
\vee (\delta \in \mathbb{L} \wedge o = \mathtt{N} \wedge (\pi, d) \in \gamma(\delta{\to}o)) \\
\vee ((\delta \in \mathbb{L} \wedge o = \mathtt{L} \vee \delta \in \mathbb{F} \wedge o = \mathtt{E} \wedge \delta \not\equiv f \vee \delta \in \mathbb{F} \wedge o = \mathtt{L} \wedge \delta \not\equiv f) \\
\quad \wedge (\pi, d) \in \gamma(\delta{\to}o \wedge v{\to}\mathtt{N} \\
\qquad\qquad\qquad\quad \vee \delta{\to}o \wedge v{\to}\mathtt{E} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\quad \vee \delta{\to}o \wedge v{\to}\mathtt{E} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\quad \vee \delta{\to}o \wedge v{\to}\mathtt{L} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\quad \vee \delta{\to}o \wedge v{\to}\mathtt{L} \wedge f{\to}\mathtt{N} \\
\qquad\qquad\qquad\quad \vee \delta{\to}o \wedge v{\to}\mathtt{L} \wedge v'{\to}\mathtt{L} \wedge f{\to}\mathtt{L} \\
\qquad\qquad\qquad\quad \vee \delta{\to}o \wedge v{\to}\mathtt{L} \wedge v'{\to}\mathtt{E} \wedge f{\to}\mathtt{E})) \\
\vee (\delta \in \mathbb{F} \wedge o = \mathtt{N} \wedge \delta \not\equiv f \wedge (\pi, d) \in \gamma(\delta{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{E} \wedge v'{\to}\mathtt{L} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{L} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{L} \wedge f{\to}\mathtt{E} \wedge v'{\to}\mathtt{L})) \\
\vee (\delta \in \mathbb{F} \wedge o = \mathtt{N} \wedge \delta \equiv f \wedge (\pi, d) \in \gamma(\delta{\to}\mathtt{N} \wedge v{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{N} \wedge v{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{N} \wedge v{\to}\mathtt{L} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee v{\to}\mathtt{E} \wedge v'{\to}\mathtt{L} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{L} \wedge v{\to}\mathtt{L} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{E} \wedge v{\to}\mathtt{L} \wedge v'{\to}\mathtt{L})) \\
\vee (\delta \in \mathbb{F} \wedge o = \mathtt{L} \wedge \delta \equiv f \wedge (\pi, d) \in \gamma(\delta{\to}\mathtt{N} \wedge v{\to}\mathtt{L} \wedge v'{\to}\mathtt{L} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{L} \wedge v{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{L} \wedge v{\to}\mathtt{E} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{L} \wedge v{\to}\mathtt{E} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{L} \wedge v{\to}\mathtt{L} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{L} \wedge v{\to}\mathtt{L} \wedge v'{\to}\mathtt{L})) \\
\vee (\delta \in \mathbb{F} \wedge o = \mathtt{E} \wedge \delta \equiv f \wedge (\pi, d) \in \gamma(\delta{\to}\mathtt{N} \wedge v{\to}\mathtt{L} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{E} \wedge v{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{E} \wedge v{\to}\mathtt{E} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{E} \wedge v{\to}\mathtt{E} \wedge v'{\to}\mathtt{N} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{E} \wedge v{\to}\mathtt{L} \wedge v'{\to}\mathtt{E} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \delta{\to}\mathtt{E} \wedge v{\to}\mathtt{L} \wedge v'{\to}\mathtt{N}))
\end{array}
$$

□

A.4 Proof of Lemma 5

LEMMA 5. *For every atomic command a, the backward transfer function $[\![a]\!]^b$ in Figure 12 satisfies requirement* (4) *of our framework in Section 4.*

Proof For every atomic command a, define a function $W[a]$ on $\mathbb{M}$ by

$$W[a](\phi) = \{(\pi, d) \mid (\pi, [\![a]\!]_\pi(d)) \in \gamma(\phi)\}.$$

We need to prove that for all ϕ,

$$\forall a.\ \gamma([\![a]\!]^b(\phi)) = W[a](\phi). \tag{2}$$

Our proof is done by induction on the structure of ϕ.

We start by considering the cases that ϕ is $\mathtt{true}$, conjunction, false or disjunction. The reason that the equality in (2) holds for these cases is that $W[a]$ preserve all of $\mathtt{true}$, conjunction, false and disjunction:

$$
\begin{array}{ll}
W[a](\mathsf{true}) = \mathbb{P} \times \mathbb{S}, & W[a](\phi \wedge \phi') = W[a](\phi) \cap W[a](\phi'), \\
W[a](\mathsf{false}) = \emptyset, & W[a](\phi \vee \phi') = W[a](\phi) \cup W[a](\phi').
\end{array}
$$

The desired equality follows from this preservation and the induction hypothesis.

Now we prove the remaining cases one-by-one. The first case is that $\phi = \mathsf{err}$. When a is $x = y$ or $x = \mathtt{null}$, we prove the desired (2) as follows:

$$
\begin{array}{rcl}
(\pi, d) \in W[a](\mathsf{err}) & \Leftrightarrow & (\pi, [\![a]\!]_\pi(d)) \in \gamma(\mathsf{err}) \\
& \Leftrightarrow & [\![a]\!]_\pi(d) = \top \\
& \Leftrightarrow & d = \top \\
& \Leftrightarrow & (\pi, d) \in \gamma(\mathsf{err}) = \gamma([\![a]\!]^b(\mathsf{err})).
\end{array}
$$

When a is a method call $x.m()$, our proof of the desired property (2) is slightly different, and it is given below:

$$
\begin{array}{l}
(\pi, d) \in W[x.m()](\mathsf{err}) \\
\Leftrightarrow \\
(\pi, [\![x.m()]\!]_\pi(d)) \in \gamma(\mathsf{err}) \\
\Leftrightarrow \\
[\![x.m()]\!]_\pi(d) = \top \\
\Leftrightarrow \\
(d = \top) \vee (\exists \sigma, \Sigma, X.\, d = (\Sigma, X) \wedge \sigma \in \Sigma \wedge [\![m]\!](\sigma) = \top) \\
\Leftrightarrow \\
(\pi, d) \in \gamma(\mathsf{err} \vee \bigvee\{\mathsf{type}(\sigma) \mid [\![m]\!](\sigma) = \top\}) \\
\Leftrightarrow \\
(\pi, d) \in \gamma([\![x.m()]\!]^b(\mathsf{err})).
\end{array}
$$

The second case is errNot. In this case, we notice that $W[a]$ preserves the negation. Hence, we prove the desired equality (2) as follows: for all atomic commands a,

$$
\begin{array}{rcl}
W[a](\mathsf{errNot}) & = & W[a](\mathsf{not}(\mathsf{err})) \\
& = & \mathbb{P} \times \mathbb{D} \setminus W[a](\mathsf{err}) \\
& = & \mathbb{P} \times \mathbb{D} \setminus \gamma([\![a]\!]^b(\mathsf{err})) \\
& = & \gamma(\mathsf{negate}([\![a]\!]^b(\mathsf{err}))) \\
& = & \gamma([\![a]\!]^b(\mathsf{errNot})).
\end{array}
$$

The third equality uses what we just proved in the err case.

The third case is $\mathsf{param}(z)$. For all atomic commands a,

$$
\begin{array}{rcl}
(\pi, d) \in W[a](\mathsf{param}(z)) & \Leftrightarrow & (\pi, [\![a]\!]_\pi(d)) \in \gamma(\mathsf{param}(z)) \\
& \Leftrightarrow & z \in \pi \\
& \Leftrightarrow & (\pi, d) \in \gamma(\mathsf{param}(z)) \\
& \Leftrightarrow & (\pi, d) \in \gamma([\![a]\!]^b(\mathsf{param}(z))).
\end{array}
$$

The fourth case is $\mathsf{paramNot}(z)$. The proof of this case is essentially identical to that of the second case, except that we are now using the fact that (2) holds for the $\mathsf{param}(z)$ case, instead of the err case.

The fifth case is $\mathsf{var}(z)$. We handle three atomic commands separately. First, we prove that (2) holds for $x = y$:

$$
\begin{array}{l}
(\pi, d) \in \gamma(W[x = y](\mathsf{var}(z))) \\
\Leftrightarrow \\
(\pi, [\![x = y]\!]_\pi(d)) \in \gamma(\mathsf{var}(z)) \\
\Leftrightarrow \\
\exists \Sigma, X.\, d = (\Sigma, X) \\
\qquad\qquad \wedge (y \in X \wedge x \in \pi \wedge z \in X \cup \{x\} \vee z \in X \setminus \{x\}) \\
\Leftrightarrow \\
(z \equiv x \wedge (\pi, d) \in \gamma(\mathsf{var}(y) \wedge \mathsf{param}(x))) \\
\quad \vee (z \not\equiv x \wedge (\pi, d) \in \gamma(\mathsf{var}(z))) \\
\Leftrightarrow \\
(\pi, d) \in \gamma([\![x = y]\!]^b(\mathsf{var}(z))).
\end{array}
$$

Second, we prove the required equality for $x = \mathtt{null}$:

$$
\begin{array}{l}
(\pi, d) \in \gamma(W[x = \mathtt{null}](\mathsf{var}(z))) \\
\Leftrightarrow \\
(\pi, [\![x = \mathtt{null}]\!]_\pi(d)) \in \gamma(\mathsf{var}(z)) \\
\Leftrightarrow \\
\exists \Sigma, X.\, d = (\Sigma, X) \wedge (z \in X \setminus \{x\}) \\
\Leftrightarrow \\
z \not\equiv x \wedge (\pi, d) \in \gamma(\mathsf{var}(z)) \\
\Leftrightarrow \\
(\pi, d) \in \gamma([\![x = \mathtt{null}]\!]^b(\mathsf{var}(z))).
\end{array}
$$

Third, we show the desired equality for $x.m()$:

$$\begin{array}{l}(\pi, d) \in W[x.m()](\mathsf{var}(z)) \\ \Leftrightarrow \\ (\pi, [\![x.m()]\!]_\pi(d)) \in \gamma(\mathsf{var}(z)) \\ \Leftrightarrow \\ \exists \Sigma, X.\, d = (\Sigma, X) \wedge z \in X \wedge (\forall \sigma.\, [\![m]\!](\sigma) = \top \Rightarrow \sigma \notin \Sigma) \\ \Leftrightarrow \\ (\pi, d) \in \gamma(\mathsf{var}(z) \wedge \bigwedge\{\mathsf{typeNot}(\sigma) \mid [\![m]\!](\sigma) = \top\}) \\ \Leftrightarrow \\ (\pi, d) \in \gamma([\![x.m()]\!]^b(\mathsf{var}(z))).\end{array}$$

The sixth case is $\mathsf{varNot}(z)$. We use the fact that $W[a]$ preserves negation and conjunction, and derive the desired equality as follows:

$$\begin{aligned}W[a](\mathsf{varNot}(z)) &= W[a](\mathsf{not}(\mathsf{var}(z)) \wedge \mathsf{errNot}) \\ &= (\mathbb{P} \times \mathbb{D} \setminus W[a](\mathsf{var}(z)) \cap W[a](\mathsf{errNot}) \\ &= (\mathbb{P} \times \mathbb{D} \setminus \gamma([\![a]\!]^b(\mathsf{var}(z)))) \cap \gamma([\![a]\!]^b(\mathsf{errNot})) \\ &= \gamma(\mathsf{negate}([\![a]\!]^b(\mathsf{var}(z))) \wedge \mathsf{errNot}).\end{aligned}$$

The first equality just uses the fact that $\mathsf{not}(\mathsf{var}(z)) = \mathsf{varNot}(z) \vee \mathsf{err}$. The second uses the preservation of negation and conjunction by $W[a]$, and the third equality uses what we have proved in the err and errNot cases. The last holds because γ preserves the conjunction and negate implements the semantic negation.

The seventh case is $\mathsf{type}(\sigma)$. When a is $x = y$ or $x = \mathtt{null}$,

$$\begin{aligned}(\pi, d) \in W[a](\mathsf{type}(\sigma)) &\Leftrightarrow (\pi, [\![a]\!]_\pi(d)) \in \gamma(\mathsf{type}(\sigma)) \\ &\Leftrightarrow \exists \Sigma, X.\, d = (\Sigma, X) \wedge \sigma \in \Sigma \\ &\Leftrightarrow (\pi, d) \in \gamma(\mathsf{type}(\sigma)) \\ &\Leftrightarrow (\pi, d) \in \gamma([\![a]\!]^b(\mathsf{type}(\sigma))).\end{aligned}$$

This shows that the desired equality holds for these atomic commands a. When a is a method call $x.m()$, we prove the equality as follows:

$$\begin{array}{l}(\pi, [\![x.m()]\!]_\pi(d)) \in \gamma(\mathsf{type}(\sigma)) \\ \Leftrightarrow \\ \exists \Sigma, X.\, d = (\Sigma, X) \\ \qquad \wedge (\forall \sigma'.\, [\![m]\!](\sigma') = \top \Rightarrow \sigma' \notin \Sigma) \\ \qquad \wedge ((\exists \sigma'.\, \sigma' \in \Sigma \wedge [\![m]\!](\sigma') = \sigma) \vee x \notin X \wedge \sigma \in \Sigma) \\ \Leftrightarrow \\ (\pi, d) \in \gamma(\mathsf{errNot} \\ \qquad \wedge (\bigwedge\{\mathsf{typeNot}(\sigma') \mid [\![m]\!](\sigma') = \top\}) \\ \qquad \wedge ((\bigvee\{\mathsf{type}(\sigma') \mid [\![m]\!](\sigma') = \sigma\}) \\ \qquad\quad \vee (\mathsf{varNot}(x) \wedge \mathsf{type}(\sigma))) \\ \Leftrightarrow \\ (\pi, d) \in \gamma([\![x.m()]\!]^b(\mathsf{type}(\sigma))).\end{array}$$

The last case is $\mathsf{typeNot}(\sigma)$. The proof of this case is similar to that of the $\mathsf{varNot}(z)$ case. □

A.5 Proof of Theorem 6

We will first prove lemmas that describe the properties of TRACER. Then, we will use the lemmas and prove the theorem.

LEMMA 7. *If the domain $\mathbb{P}$ of parameters is finite,* TRACER(d_I, s, q) *terminates.*

Proof We will prove that each iteration removes at least one element from Π_{viable}. Since no new elements are added to Π_{viable} in each iteration, this combined with the fact that $\mathbb{P}$ (the initial value of Π_{viable}) is finite will conclude the proof. Suppose some $\pi \in \Pi_{viable}$ is chosen on line 8 and some trace $\tau \in \mathsf{trace}(s)$ is chosen on line 13 such that

$$F_\pi[\tau](d_I) \in (F_\pi[s](\{d_I\}) \cap \{d \mid (\pi, d) \in \gamma(\mathsf{not}(q))\}).$$

Such a trace τ exists because of Lemma 1. Hence, we have

$$(\pi, F_\pi[\tau](d_I)) \in \gamma(\mathsf{not}(q)). \tag{3}$$

Then, because of (3) above and the item 1 of Theorem 3, we have

$$(\pi, d_I) \in \gamma(B[\tau](\pi, d_I, \mathsf{not}(q))).$$

The equation above implies that the Π computed on line 14 contains π. Thus, at least π is removed from Π_{viable} on line 15. □

LEMMA 8. *If* TRACER(d_I, s, q) *removes π from Π_{viable} in some iteration then $F_\pi[s](\{d_I\})$ is not a subset of $\{d \mid (\pi, d) \in \gamma(q)\}$.*

Proof Suppose TRACER(d_I, s, q) removes π' from Π_{viable} in some iteration. We need to prove that

$$F_{\pi'}[s](\{d_I\}) \not\subseteq \{d \mid (\pi', d) \in \gamma(q)\}.$$

But this is equivalent to

$$F_{\pi'}[s](\{d_I\}) \cap \{d \mid (\pi', d) \in \gamma(\mathsf{not}(q))\} \neq \emptyset,$$

which we will show in this proof. Since π' is removed from Π_{viable} in some iteration, TRACER must have done so using π and $\tau \in \mathsf{trace}(s)$ such that

$$(\pi, F_\pi[\tau](d_I)) \in \gamma(\mathsf{not}(q)) \wedge (\pi', d_I) \in \gamma(B[\tau](\pi, d_I, \mathsf{not}(q))).$$

By the item 2 of Theorem 3, the second conjunct implies that

$$(\pi', F_{\pi'}[\tau](d_I)) \in \gamma(\mathsf{not}(q)). \tag{4}$$

Furthermore, because of $\tau \in \mathsf{trace}(s)$, by Lemma 1, we also have that

$$F_{\pi'}[\tau](d_I) \in F_{\pi'}[s](\{d_I\}). \tag{5}$$

From (4) and (5) follows

$$F_{\pi'}[s](\{d_I\}) \cap \{d \mid (\pi', d) \in \gamma(\mathsf{not}(q))\} \neq \emptyset,$$

as desired. □

LEMMA 9. *If* TRACER(d_I, s, q) *returns π, we have that*

$$F_\pi[s](\{d_I\}) \subseteq \{d \mid (\pi, d) \in \gamma(q)\}.$$

Furthermore, π is a minimum-cost parameter value (according to $\preceq$) satisfying the above subset relationship.

Proof TRACER(d_I, s, q) returns π only when

$$F_\pi[s](\{d_I\}) \cap \{d \mid (\pi, d) \in \gamma(\mathsf{not}(q))\} = \emptyset. \tag{6}$$

But there is some $D_0 \subseteq \mathbb{D}$ such that

$$\gamma(\mathsf{not}(q)) = (\mathbb{P} \times (\mathbb{D} \setminus D_0)) \wedge \gamma(q) = (\mathbb{P} \times D_0).$$

Hence, (6) implies the desired subset relationship on π. Let's move on to the proof that π is minimum. For the sake of contradiction, suppose that there exists π' such that

$$\pi \not\preceq \pi' \wedge F_{\pi'}[s](\{d_I\}) \subseteq \{d \mid (\pi', d) \in \gamma(q)\}. \tag{7}$$

Then, π' should not be in Π_{viable} in the iteration of TRACER that chose π from Π_{viable}. Since Π_{viable} is becoming a smaller set in each iteration of the algorithm, π' must have been removed from Π_{viable} in some previous iteration. According to Lemma 8, this can happen only if

$$F_{\pi'}[s](\{d_I\}) \not\subseteq \{d \mid (\pi', d) \in \gamma(q)\},$$

which contradicts the second conjunct of (7). □

LEMMA 10. *If* TRACER(d_I, s, q) *returns impossible, there is no π such that*

$$F_\pi[s](\{d_I\}) \subseteq \{d \mid (\pi, d) \in \gamma(q)\}.$$

Proof Suppose TRACER(d_I, s, q) returns impossible. This means that $\Pi_{viable} = \emptyset$ right before the algorithm terminates. Hence, every parameter value π was removed from Π_{viable} in some iteration of TRACER(d_I, s, q). This combined with Lemma 8 implies that $F_\pi[s](\{d_I\}) \not\subseteq \{d \mid (\pi, d) \in \gamma(q)\}$ for every π, as claimed by this lemma. □

THEOREM 6. TRACER(d_I, s, q) *computes correct results, and is guaranteed to terminate when* $\mathbb{P}$ *is finite.*

Proof The theorem follows from Lemmas 7, 9, and 10. □