Finding Optimum Abstractions in Parametric Dataflow Analysis

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

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

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

Abstract

We propose a technique to efficiently search a large family of abstractions in order to prove a query using a parametric dataflow analysis. Our technique either finds the cheapest such abstraction or shows that none exists. It is based on counterexample-guided abstraction refinement but applies a novel meta-analysis on abstract counterexample traces to efficiently find abstractions that are incapable of proving the query. We formalize the technique in a generic framework and apply it to two analyses: a type-state analysis and a thread-escape analysis. We demonstrate the effectiveness of the technique on a suite of Java benchmark programs.

Categories and Subject Descriptors D.2.4 [*SOFTWARE ENGINEERING*]: Software/Program Verification; F.3.2 [*LOGICS AND MEANINGS OF PROGRAMS*]: Semantics of Programming Languages—Program analysis

General Terms Languages, Verification

Keywords Dataflow analysis, CEGAR, abstraction refinement, optimum abstraction, impossibility, under-approximation

1. Introduction

A central problem in static analysis concerns how to balance its precision and cost. A *query-driven analysis* seeks to address this problem by searching for an abstraction that discards program details that are unnecessary for proving an individual query.

We consider query-driven dataflow analyses that are parametric in the abstraction. The abstraction is chosen from a large family that allow abstracting different parts of a program with varying precision. A large number of fine-grained abstractions enables an analysis to specialize to a query but poses a hard search problem in practice. First, the number of abstractions is infinite or intractably large to search naïvely, with most abstractions in the family being too imprecise or too costly to prove a particular query. Second, proving queries in different parts of the same program requires different abstractions. Finally, a query might be unprovable by all abstractions in the family, either because the query is not true or due to limitations of the analysis at hand.

We propose an efficient technique to solve the above search problem. It either finds the cheapest abstraction in the family that proves a query or shows that no abstraction in the family can prove the query. We call this the *optimum abstraction problem*. Our technique is based on counterexample-guided abstraction refinement (CEGAR) 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 abstractions such that the analysis instantiated using any abstraction in this set is guaranteed to fail to prove the query. Our technique finds such unviable abstractions by doing a backward analysis on the trace. This backward analysis 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 analysis avoids this problem as it is guided by the trace from the forward analysis. This trace also enables our backward analysis to do under-approximation while guaranteeing to find a non-empty set of unviable abstractions.

Like the forward analysis, the backward meta-analysis is a static analysis, and the performance of our technique depends on how this meta-analysis balances its own precision and cost. If it does under-approximation aggressively, it analyzes the trace efficiently but finds only the current abstraction unviable and needs more iterations to converge. On the other hand, if it does under-approximation passively, it analyzes the trace inefficiently but finds many abstractions unviable and needs fewer iterations. We present a generic framework to develop an efficient meta-analysis, which involves choosing an abstract domain, devising (backward) transfer functions, and proving these functions sound with respect to the forward analysis. Our framework uses a generic DNF representation of formulas in the domain and provides generic optimizations to scale the meta-analysis. We show the applicability of the framework to two analyses: a type-state analysis and a thread-escape analysis.

We evaluate our technique using these two client analyses on seven Java benchmark programs of 400K–600K bytecodes each. Both analyses are fully flow- and context-sensitive with 2^N possible abstractions, where N is the number of pointer variables for type-state analysis, and the number of object allocation sites for thread-escape analysis. The technique finds the cheapest abstraction or shows that none exists for 92.5% of queries posed on average per client analysis per benchmark program.

We summarize the main contributions of this paper:

- We formulate the optimum abstraction problem for parametric dataflow analysis. The formulation seeks a cheapest abstraction that proves the query or an impossibility result that none exists.
- We present a new refinement-based technique to solve the problem. The key idea is a meta-analysis that analyzes counterexamples to find abstractions that are incapable of proving the query.
- We present a generic framework to design the meta-analysis along with an efficient representation and optimizations to scale it. We apply the framework to two analyses in the literature.
- We demonstrate the efficacy of our technique in practice on the two analyses, a type-state analysis and a thread-escape analysis, for several real-world Java benchmark programs.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

The rest of the paper is organized as follows. Section 2 illustrates our technique on an example. Section 3 formalizes parametric dataflow analysis and the optimum abstraction problem. We first define a generic framework and then apply it to our two example analyses. Section 4 presents our meta-analysis, first in a generic setting and then for a domain that suffices for our two analyses. Section 5 gives our overall algorithm and Section 6 presents empirical results. Section 7 discusses related work and Section 8 concludes.

2. Example

We illustrate our technique using the program in Figure 1(a). The program operates on a File object which can be in either state *opened* or *closed* at any instant. It begins in state *closed* upon creation, transitions to state *opened* after the call to open(), and back to state *closed* after the call to close(). Suppose that it is an error to call open() in the *opened* state, or to call close() in the *closed* state. Statements $check_1$ and $check_2$ at the end of the program are queries which ask whether the state of the File object is *closed* or *opened*, respectively, at the end of every program run.

A static type-state analysis can be used to conservatively answer such queries. Such an analysis must track aliasing relationships between pointer variables in order to track the state of objects correctly and precisely. For instance, in our example program, the analysis must infer that variables x and y point to the same File object in order to prove query $check_1$. Analyses that track more program facts are typically more precise but also more costly. A query-based analysis enables striking a balance between precision and cost by not tracking program facts that are unnecessary for proving an individual query.

Making our type-state analysis query-based enables it to track only variables that matter for answering a particular query (such as $check_1$). We therefore parameterize this analysis by an *abstraction* π which specifies the set of variables that the analysis must track. An abstraction π_1 is cheaper than an abstraction π_2 if π_1 tracks fewer variables than π_2 (i.e., $|\pi_1| < |\pi_2|$). For a program with N variables there are 2^N possible abstractions. Figure 1(b) shows which abstractions in this family are suitable for each of our two queries. Any abstraction containing variables x and y is precise enough to let our analysis prove query $check_1$. The cheapest of these abstractions is $\{x, y\}$. On the other hand, our analysis cannot prove query $check_2$ using any abstraction in the family. This is because query $check_2$ is not true concretely, but the analysis may fail to prove even true queries due to its conservative nature.

We propose an efficient technique for finding the cheapest abstraction in the family that proves a query or showing that no abstraction in the family can prove the query. We illustrate our technique on our parametric type-state analysis. This analysis is fully flow- and context-sensitive, and tracks for each allocation site in the program a pair $\langle ts, vs\rangle$ or $\top$, where ts over-approximates the set of possible type-states of an object created at that site, and vs is a must-alias set—a set of variables that definitely point to that object. Only variables in the abstraction π used to instantiate the analysis are allowed to appear in any must-alias set. $\top$ denotes that the analysis has detected a type-state error.

Our technique starts by running the analysis with the cheapest abstraction. For our type-state analysis this corresponds to not tracking any variable at all. The resulting analysis fails to prove query $check_1$. Our technique is CEGAR-based and requires the analysis to produce an abstract counterexample trace as a failure witness. Such a trace showing the failure to prove query $check_1$ under the cheapest abstraction is shown in Figure 1(c). The trace is annotated with abstract states computed by the analysis, denoted $\downarrow$, that track information about the lone allocation site in the program. Note that state $\langle\{closed\}, \{\}\rangle$ incoming into call x.open() results in outgoing state $\langle\{opened, closed\}, \{\}\rangle$ even though the File object cannot be in the *closed* state after the call. This is because the analysis does a weak update as x is not in the must-alias set of the incoming abstract state.

At this point our technique has eliminated abstraction $\pi = \{\}$ as incapable of proving query $check_1$ and must determine which abstraction to try next. Since the number of abstractions is large, however, it first analyzes the trace to eliminate abstractions that are guaranteed to suffer a failure similar to the current one. It does so by performing a backward meta-analysis that inspects the trace backwards and analyzes the result of the (forward) type-state analysis. This meta-analysis is itself a static analysis and the abstract states it computes are denoted by $\uparrow$ in Figure 1(c). Each abstract state of the meta-analysis represents a set of pairs (d, π) consisting of an abstract state d of the forward analysis and an abstraction π. An abstract state of the meta-analysis is a boolean formula over primitives of the form: (1) *err* representing the set of pairs where the d component is $\top$; (2) $closed \in ts$ representing the set of pairs where the d component is $\langle ts, vs\rangle$ and ts contains *closed*; (3) $x \in vs$ which is analogous to (2) except that vs contains x; and (4) $x \in \pi$ meaning the π component contains x.

The meta-analysis starts with the weakest formula under which $check_1$ fails, which is $err \vee (opened \in ts)$, and propagates its weakest precondition at each step of the trace to obtain formula $(closed \in ts) \wedge (opened \notin ts) \wedge (x \notin \pi)$ at the start of the trace. This formula implies that any abstraction not containing variable x cannot prove query $check_1$. Our meta-analysis has two notable aspects. First, it does not require the family of abstractions to be finite: it can show that all abstractions in even an infinite family cannot prove a query. Second, it avoids the blowup inherent in backward analyses by performing under-approximation. It achieves this for the above domain of boolean formulae by dropping disjuncts in the DNF representation of the formulae. A crucial condition ensured by our meta-analysis during under-approximation is that the abstract state computed by the forward analysis is contained in the resulting simpler formula. Otherwise, it is not possible to guarantee that the eliminated abstractions cannot prove the query. Section 4 explains how our technique picks the best k disjuncts for a $k \geq 1$ specified by the analysis designer. Smaller k enables the meta-analysis to run efficiently on each trace but can require more iterations by eliminating only the current abstraction in each iteration in the extreme case. We use $k = 1$ for this example and highlight in bold the chosen (retained) disjunct in each formula with multiple disjuncts in Figure 1. For instance, we drop the second disjunct in formula $err \vee (opened \in ts)$ at the end point of the trace in Figure 1(c), since abstract state $\top$ computed by the forward analysis at that point is not in the set of states represented by $(opened \in ts)$. The weakest precondition of the chosen disjunct *err* with respect to the call y.close() is $err \vee (closed \in ts)$, but this time we drop disjunct *err* as the abstract state $\langle\{closed, opened\}, \{\}\rangle$ computed by the forward analysis is not in the set of states represented by *err*.

Having eliminated all abstractions that do not contain variable x, our technique next runs the type-state analysis with abstraction $\pi = \{x\}$, but again fails to prove query $check_1$, and produces the trace showed in Figure 1(d). Our meta-analysis performed on this trace infers that any abstraction containing variable x but not containing variable y cannot prove the query. Hence, our technique next runs the type-state analysis with abstraction $\pi = \{x, y\}$, and this time succeeds in proving the query. Our technique is effective at slicing away program details that are irrelevant to proving a query. For instance, even if either of the traces in Figure 1 (c) or (d) contained the statement "if (*) z = x", variable z would not be included in any abstraction that our technique tries; indeed, tracking variable z is not necessary for proving query $check_1$.

Finally, consider the query $check_2$. Our technique starts with the cheapest abstraction $\pi = \{\}$, and obtains a trace identical

x = new File;
y = x;
if (*) z = x;
x.open();
y.close();
if (*)
    $check_1$(x, *closed*);
else
    $check_2$(x, *opened*);

(a) Example program.

query	abstraction
$check_1$	any $\pi \supseteq \{x, y\}$
$check_2$	none

(b) Feasible solutions.

$\uparrow closed \in ts \wedge opened \notin ts \wedge\ x \notin \pi$
x = new File;
$\downarrow \langle\{closed\}, \{\}\rangle$
$\uparrow closed \in ts \wedge opened \notin ts \wedge\ x \notin vs$
y = x;
$\downarrow \langle\{closed\}, \{\}\rangle$
$\uparrow closed \in ts \wedge opened \notin ts \wedge\ x \notin vs$
x.open();
$\downarrow \langle\{closed, opened\}, \{\}\rangle$
$\uparrow err \vee \boldsymbol{closed \in ts}$
y.close();
$\downarrow \top$
$\uparrow \boldsymbol{err} \vee opened \in ts$
$check_1$(x, *closed*);

(c) Iteration 1 for query $check_1$ using $\pi = \{\}$.

$\uparrow closed \in ts \wedge opened \notin ts \wedge y \notin \pi \wedge x \in \pi$
x = new File;
$\downarrow \langle\{closed\}, \{x\}\rangle$
$\uparrow closed \in ts \wedge opened \notin ts \wedge y \notin \pi \wedge x \in vs$
y = x;
$\downarrow \langle\{closed\}, \{x\}\rangle$
$\uparrow closed \in ts \wedge opened \notin ts \wedge y \notin vs \wedge x \in vs$
x.open();
$\downarrow \langle\{opened\}, \{x\}\rangle$
$\uparrow opened \in ts \wedge closed \notin ts \wedge y \notin vs$
y.close();
$\downarrow \langle\{closed, opened\}, \{x\}\rangle$
$\uparrow err \vee \boldsymbol{opened \in ts}$
$check_1$(x, *closed*);

(d) Iteration 2 for query $check_1$ using $\pi = \{x\}$.

$\uparrow closed \in ts \wedge opened \notin ts \wedge x \in \pi$
x = new File;
$\downarrow \langle\{closed\}, \{x\}\rangle$
$\uparrow closed \in ts \wedge opened \notin ts \wedge x \in vs$
y = x;
$\downarrow \langle\{closed\}, \{x\}\rangle$
$\uparrow closed \in ts \wedge opened \notin ts \wedge x \in vs$
x.open();
$\downarrow \langle\{opened\}, \{x\}\rangle$
$\uparrow opened \in ts \wedge closed \notin ts$
y.close();
$\downarrow \langle\{closed, opened\}, \{x\}\rangle$
$\uparrow err \vee \boldsymbol{closed \in ts}$
$check_2$(x, *opened*);

(e) Iteration 2 for query $check_2$ using $\pi = \{x\}$.

Figure 1. Example illustrating our technique for a parametric type-state analysis.

to that in Figure 1(b) for query $check_1$, except that the meta-analysis starts by propagating formula *err* $\vee$ (*closed* $\in ts$) instead of *err* $\vee$ (*opened* $\in ts$). The same disjunct *err* in this formula is retained and the rest of the meta-analysis result is identical. Thus, in the next iteration, our technique runs the type-state analysis with abstraction $\pi = \{x\}$, and obtains the trace shown in Figure 1(e). This time, the meta-analysis retains disjunct (*closed* $\in ts$), and concludes any abstraction containing variable x cannot prove the query. Since in the first iteration all abstractions without variable x were eliminated, our technique concludes that the analysis cannot prove query $check_2$ using any abstraction.

3. Formalism

This section describes a formal setting used throughout the paper.

3.1 Programming Language

We present our results using a simple imperative language:

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

The language includes a (unspecified) set of atomic commands. Examples are assignments $v = w.f$ and statements $\mathsf{assume}(e)$ which filter out executions where e evaluates to false. The language also has 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 2 shows a standard way to generate all traces of a program s.

3.2 Parametric Dataflow Analysis

We consider dataflow analyses whose transfer functions for atomic commands are parametric in the abstraction. We specify such a parametric analysis by the following data:

1. A set $(\mathbb{P}, \preceq)$ with a preorder $\preceq$ (i.e., $\preceq$ is reflexive and transitive). Elements $\pi \in \mathbb{P}$ are parameter values. We call them *abstractions* as they determine the degree of approximation performed by the analysis. The preorder $\preceq$ on π's dictates the cost of the analysis: using a smaller π yields a cheaper 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 $\pi' \in P$).

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

Figure 2. Traces of a program s. Symbol ϵ denotes an empty trace.

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 an abstraction to be provided before the analysis starts. The abstract semantics in Figure 3 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 abstraction π is used when atomic commands are interpreted. Hence, π controls the analysis by changing the transfer functions for atomic commands. Besides this parameterization all the defining clauses are standard.

Our parametric analyses satisfy a fact of disjunctive analyses:

LEMMA 1. *For all programs s, abstractions π, and abstract states d, we have $F_\pi[s](\{d\}) = \{F_\pi[\tau](d) \mid \tau \in \mathsf{trace}(s)\}$, where $F_\pi[\tau]$ is the result of analyzing trace τ as shown in Figure 3.*

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 loops and is much simpler than the original program s. We show later how our technique exploits this simplicity (Section 4). We now formalize two example parametric analyses.

Type-State Analysis. Our first example is a variant of the type-state analysis from [8]. The original analysis maintains various kinds of aliasing facts in order to track the type-state of objects correctly and precisely. Our variant only tracks must-alias facts.

We assume we are given a set $\mathbb{T}$ of type-states containing *init*, which represents the initial type-state of objects, and a function $[\![m]\!] : \mathbb{T} \to \mathbb{T} \cup \{\top\}$ for every method m, which describes how a call $x.m()$ changes the type-state of object x and when it leads to

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

$$
\begin{array}{rcl}
F_\pi[\tau] & : & \mathbb{D} \to \mathbb{D} \\
F_\pi[\epsilon](d) & = & d \\
F_\pi[a](d) & = & [\![a]\!]_\pi(d) \\
F_\pi[\tau\,;\,\tau'](d) & = & F_\pi[\tau'](F_\pi[\tau](d))
\end{array}
$$

Figure 3. Abstract semantics. In the case of loop, we take the least fixpoint with respect to the subset order in the powerset domain $2^{\mathbb{D}}$.

$$
\begin{array}{rl}
\text{(type-states)} & \sigma \in \mathbb{T} \ \ \text{(we assume } \mathit{init} \in \mathbb{T}) \\
\text{(variables)} & x, y \in \mathbb{V} \\
\text{(analysis parameter)} & \pi \in \mathbb{P} = 2^{\mathbb{V}} \\
\text{(abstract state)} & d \in \mathbb{D} = (2^{\mathbb{T}} \times 2^{\mathbb{V}}) \cup \{\top\} \\
\text{(order on parameters)} & \pi \preceq \pi' \Leftrightarrow |\pi| \le |\pi'|
\end{array}
$$

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

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

Figure 4. Data for the type-state analysis.

an error, denoted $\top$. Using these data, we specify the domains and transfer functions of the analysis in Figure 4.

The abstraction π for the analysis is a set of variables that determines what can appear in the must-alias set of an abstract state during the analysis. An abstract state d has form (ts, vs) or $\top$, where vs should be a subset of π, and it tracks information about a single object. In the former case, ts represents all the possible type-states of the tracked object and vs, the must-alias set of this object. The latter means that the object can be in any type-state including the error state $\top$. For brevity, we show transfer functions only for simple assignments and method calls. Those for assignments $x = y$ and $x = \texttt{null}$ update the must-alias set according to their concrete semantics and abstraction π. The transfer function for a method call $x.m()$ updates the ts 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 type-states.

According to our order $\preceq$, an abstraction 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 precise about must-alias sets, which normally correlates the faster convergence of the fixpoint iteration of the analysis.

Thread-Escape Analysis. Our second example is a variant of the thread-escape analysis from [17]. A heap object in a multithreaded shared-memory program is thread-local when it is reachable only from at most a single thread. A thread-escape analysis conservatively answers queries about thread locality.

Let $\mathbb{H}$ be the set of allocation sites, $\mathbb{L}$ that of local variables, and $\mathbb{F}$ that of object fields in a program. Using these data, we specify the domains and transfer functions of the analysis in Figure 5. There `N` means the null value, and `L` and `E` are abstract locations, representing disjoint sets of heap objects, except that both `L` and `E` include the null value. Abstract location `E` summaries null, all the thread-escaping objects, and possibly some thread-local ones. On the other hand, `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 `E`-summarized objects are closed under pointer reachability: if a heap object is summarized by `E`, following any of its fields always gives null or `E`-summarized objects, but not `L`-summarized ones.

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

We order abstractions $\pi \preceq \pi'$ based on how many sites are mapped to `L` by π and π'. This is because the number of `L`-mapped sites usually correlates the performance of the analysis.

The thread-escape analysis includes transfer functions of the following heap-manipulating commands:

$$
\begin{array}{rl}
a \ ::= & v = \mathsf{new}\, h \ \mid\ g = v \ \mid\ v = g \ \mid\ v = \texttt{null} \ \mid\ v = v' \ \mid \\
& v = v'.f \ \mid\ v.f = v'
\end{array}
$$

Here g and v are global and local variables, respectively, and $h \in \mathbb{H}$ is an allocation site. Transfer functions of these commands simulate their usual meanings but on abstract instead of concrete states.

Function $[\![v = \mathsf{new}\, h]\!]_\pi(d)$ makes variable v point to abstract location $\pi(h)$, simulating the allocation of a $\pi(h)$-summarized 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 escape, because it exposes the object to other threads via the global variable g. When v points to `L`, the transfer function calls $\mathsf{esc}(d)$, which sets all local variables to `E` (unless they have value `N`) and resets all fields to `N`. This means that after calling $\mathsf{esc}(d)$, the analysis loses most of the information about thread locality, and concludes that all variables point to potentially escaping objects. This dramatic information loss is inevitable because if v points to `L` beforehand, the assignment $g = v$ can cause any object summarized by `L` to escape. The transfer functions of the remaining commands in the figure can be understood similarly by referring to the concrete semantics and approximating it over abstract states.

Figure 6 shows the abstract state (denoted by $\downarrow$) computed by our thread-escape analysis at each point of an example program. The results of the analysis in parts (a) and (b1) of the figure are obtained using the same abstraction $[\texttt{h1} \mapsto \texttt{E}, \texttt{h2} \mapsto \texttt{E}]$ and are thus identical; the results in part (b2) are obtained using a different abstraction $[\texttt{h1} \mapsto \texttt{L}, \texttt{h2} \mapsto \texttt{E}]$.

3.3 Optimum Abstraction Problem

Parametric dataflow 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 abstraction π. Ideally, we would like to pick π that causes 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 optimum abstraction problem provides a guideline on resolving the issue of abstraction selection. It sets a specific target on abstractions to aim at. Assume that we are interested in queries expressed as subsets of $\mathbb{D}$. The problem is defined as follows:

$$\begin{array}{l}
\quad\downarrow [\mathtt{u}\mapsto\mathtt{N},\mathtt{v}\mapsto\mathtt{N}]\\
\quad\uparrow \mathtt{h1}{\to}\mathtt{E} \vee (\mathtt{h2}{\to}\mathtt{E}\wedge\mathtt{h1}{\to}\mathtt{L})\\
\texttt{u = new h1;}\\
\quad\downarrow [\mathtt{u}\mapsto\mathtt{E},\mathtt{v}\mapsto\mathtt{N}]\\
\quad\uparrow \mathtt{u}{\to}\mathtt{E} \vee (\mathtt{h2}{\to}\mathtt{E}\wedge\mathtt{u}{\to}\mathtt{L})\ \vee\ (\mathtt{h2}{\to}\mathtt{L}\wedge\mathtt{f}{\to}\mathtt{E}\wedge\mathtt{u}{\to}\mathtt{L})\\
\texttt{v = new h2;}\\
\quad\downarrow [\mathtt{u}\mapsto\mathtt{E},\mathtt{v}\mapsto\mathtt{E}]\\
\quad\uparrow \mathtt{u}{\to}\mathtt{E} \vee (\mathtt{v}{\to}\mathtt{E}\wedge\mathtt{u}{\to}\mathtt{L})\ \vee\ (\mathtt{v}{\to}\mathtt{L}\wedge\mathtt{f}{\to}\mathtt{E}\wedge\mathtt{u}{\to}\mathtt{L})\\
\texttt{v.f = u;}\\
\quad\downarrow [\mathtt{u}\mapsto\mathtt{E},\mathtt{v}\mapsto\mathtt{E}]\\
\quad\uparrow \mathtt{u}{\to}\mathtt{E}\\
\texttt{pc: local(u)?}
\end{array}$$

(a) $\pi = [\mathtt{h1}\mapsto\mathtt{E},\mathtt{h2}\mapsto\mathtt{E}]$

$$\begin{array}{l}
\quad\downarrow [\mathtt{u}\mapsto\mathtt{N},\mathtt{v}\mapsto\mathtt{N}]\\
\quad\uparrow \mathtt{h1}{\to}\mathtt{E}\\
\texttt{u = new h1;}\\
\quad\downarrow [\mathtt{u}\mapsto\mathtt{E},\mathtt{v}\mapsto\mathtt{N}]\\
\quad\uparrow \mathtt{u}{\to}\mathtt{E}\\
\texttt{v = new h2;}\\
\quad\downarrow [\mathtt{u}\mapsto\mathtt{E},\mathtt{v}\mapsto\mathtt{E}]\\
\quad\uparrow \mathtt{u}{\to}\mathtt{E}\\
\texttt{v.f = u;}\\
\quad\downarrow [\mathtt{u}\mapsto\mathtt{E},\mathtt{v}\mapsto\mathtt{E}]\\
\quad\uparrow \mathtt{u}{\to}\mathtt{E}\\
\texttt{pc: local(u)?}
\end{array}$$

(b1) $\pi = [\mathtt{h1}\mapsto\mathtt{E},\mathtt{h2}\mapsto\mathtt{E}]$

$$\begin{array}{l}
\quad\downarrow [\mathtt{u}\mapsto\mathtt{N},\mathtt{v}\mapsto\mathtt{N}]\\
\quad\uparrow \mathtt{h1}{\to}\mathtt{L}\wedge\mathtt{h2}{\to}\mathtt{E}\\
\texttt{u = new h1;}\\
\quad\downarrow [\mathtt{u}\mapsto\mathtt{L},\mathtt{v}\mapsto\mathtt{N}]\\
\quad\uparrow \mathtt{u}{\to}\mathtt{L}\wedge\mathtt{h2}{\to}\mathtt{E}\\
\texttt{v = new h2;}\\
\quad\downarrow [\mathtt{u}\mapsto\mathtt{L},\mathtt{v}\mapsto\mathtt{E}]\\
\quad\uparrow \mathtt{u}{\to}\mathtt{L}\wedge\mathtt{v}{\to}\mathtt{E}\\
\texttt{v.f = u;}\\
\quad\downarrow [\mathtt{u}\mapsto\mathtt{E},\mathtt{v}\mapsto\mathtt{E}]\\
\quad\uparrow \mathtt{u}{\to}\mathtt{E}\\
\texttt{pc: local(u)?}
\end{array}$$

(b2) $\pi = [\mathtt{h1}\mapsto\mathtt{L},\mathtt{h2}\mapsto\mathtt{E}]$

Figure 6. Example showing our technique for thread-escape analysis with under-approximation (parts (b1) and (b2)) and without (part (a)).

$$\begin{array}{rl}
\text{(allocation sites)} & h \in \mathbb{H}\\
\text{(local variables)} & v \in \mathbb{V}\\
\text{(object fields)} & f \in \mathbb{F}\\
\text{(analysis parameter)} & \pi \in \mathbb{P} = \mathbb{H} \to \{\mathtt{L},\mathtt{E}\}\\
\text{(abstract state)} & d \in \mathbb{D} = (\mathbb{L}\cup\mathbb{F}) \to \{\mathtt{L},\mathtt{E},\mathtt{N}\}\\
\text{(order on parameters)} & \pi \preceq \pi' \Leftrightarrow |\{h\in\mathbb{H} \mid \pi(h)=\mathtt{L}\}| \le\\
 & \qquad |\{h\in\mathbb{H} \mid \pi'(h)=\mathtt{L}\}|
\end{array}$$

$$\begin{array}{rcl}
\mathsf{esc} & : & \mathbb{D}\to\mathbb{D}\\
\mathsf{esc}(d) & = & \lambda u.\ \text{if } (d(u)=\mathtt{N} \vee u\in\mathbb{F}) \text{ then } \mathtt{N} \text{ else } \mathtt{E}
\end{array}$$

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

$$\begin{array}{rcl}
[\![v = \mathsf{new}\ h]\!]_\pi(d) & = & d[v\mapsto\pi(h)]\\
[\![g = v]\!]_\pi(d) & = & \text{if } (d(v)=\mathtt{L}) \text{ then } \mathsf{esc}(d) \text{ else } d\\
[\![v = g]\!]_\pi(d) & = & d[v\mapsto\mathtt{E}]\\
[\![v = \mathtt{null}]\!]_\pi(d) & = & d[v\mapsto\mathtt{N}]\\
[\![v = v']\!]_\pi(d) & = & d[v\mapsto d(v')]\\
[\![v = v'.f]\!]_\pi(d) & = & \begin{cases} d[v\mapsto d(f)] & \text{if } (d(v')=\mathtt{L})\\ d[v\mapsto\mathtt{E}] & \text{otherwise}\end{cases}\\
[\![v.f = v']\!]_\pi(d) & = &
\end{array}$$

$$\begin{cases}
\mathsf{esc}(d) & \text{if } d(v)=\mathtt{E}\wedge d(v')=\mathtt{L}\\
d & \text{if } (d(v)=\mathtt{E}\wedge d(v')\neq\mathtt{L})\vee d(v)=\mathtt{N}\\
\mathsf{esc}(d) & \text{if } d(v)=\mathtt{L}\wedge\{d(f),d(v')\}=\{\mathtt{L},\mathtt{E}\}\\
d[f\mapsto\mathtt{L}] & \text{if } d(v)=\mathtt{L}\wedge\{d(f),d(v')\}=\{\mathtt{N},\mathtt{L}\}\\
d[f\mapsto\mathtt{E}] & \text{if } d(v)=\mathtt{L}\wedge\{d(f),d(v')\}=\{\mathtt{N},\mathtt{E}\}\\
d & \text{if } d(v)=\mathtt{L}\wedge d(f)=d(v')
\end{cases}$$

Figure 5. Data for the thread-escape analysis.

DEFINITION 2 (Optimal Abstraction Problem). *Given a program s, an initial abstract state d_I, and a query $q \subseteq \mathbb{D}$, find a minimum abstraction*[1] *π such that*

$$F_\pi[s]\{d_I\} \subseteq q, \qquad (1)$$

or show that $F_\pi[s]\{d_I\} \subseteq q$ does not hold for any π.

The problem asks for not a minimal π but a minimum hence cheapest one. This requirement encourages any solution to exploit cheap abstractions (determined by $\preceq$) as much as possible. Note that the requirement is aligned with the intended meaning of $\preceq$, which compares parameters in terms of the analysis cost. A different way of comparing parameters in terms of precision, not cost, is taken in Liang et al. [16]. In their case, a minimum parameter for proving a query often does not exist or it is expensive to compute, so they ask for a minimal parameter instead.

Our approach to solve the problem is based on using a form of a backward meta-analysis that reasons about the behavior of a parametric dataflow analysis under different abstractions simultaneously. We explain this backward meta-analysis next.

[1] The condition that π be minimum means: if $F_{\pi'}[s]d_I \subseteq q$, then $\pi \preceq \pi'$. In contrast, the minimality of π means the absence of π' that satisfies (1) and is strictly smaller than π according to $\preceq$.

4. Backward Meta-Analysis

In this section, we fix a parametric dataflow 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 analysis as ***forward analysis***.

A backward meta-analysis is a core component of our algorithm for the optimum abstraction 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 abstraction π fails to prove a query, and to generalize this reason. Concretely, the inputs to the meta-analysis are a trace τ, an abstraction 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 abstractions that lead to a similar verification failure of the forward analysis. The collected abstractions are used subsequently when our top-level algorithm computes a necessary condition on abstractions for proving the given query and chooses a next abstraction 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

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

Figure 7. Backward meta-analysis.

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

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

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

Figure 8. Functions for manipulating ϕ's.

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

$$\forall\phi_1\in\mathbb{M}.\ \gamma([\![a]\!]^b(\phi_1)) = \{(\pi,d) \mid (\pi,[\![a]\!]_\pi(d))\in\gamma(\phi_1)\}. \tag{2}$$

- 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 ensures that $\mathsf{approx}(\pi,d,\phi)$ under-approximates the input ϕ, and the second says that this under-approximation should keep at least (π,d), if it is already in $\gamma(\phi)$. The main purpose of approx is to simplify ϕ. For instance, when ϕ is a logical formula, approx converts ϕ to a syntactically simpler one. The operator is invoked frequently by our meta-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 7. 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. Proofs of all theorems are provided in the supplementary material.

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. $\forall(\pi_0,d_0)\in\gamma(B[\tau](\pi,d,\phi)).\ (\pi_0,F_{\pi_0}[\tau](d_0))\in\gamma(\phi)$.

4.1 Disjunctive Meta-Analysis and Underapproximation

Designing a good under-approximation 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 under-approximation operator for these meta-analyses. Both our example analyses have disjunctive meta-analyses and use the generic under-approximation operator.

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

- The set $\mathbb{M}$ consists of formulas ϕ:
$$\phi ::= p \mid \mathsf{true} \mid \mathsf{false} \mid \neg\phi \mid \phi\wedge\phi' \mid \phi\vee\phi' \quad (p\in\mathsf{PForm})$$

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

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

$$\gamma(\mathsf{err}) = \{(\pi,\top)\}$$
$$\gamma(\mathsf{param}(x)) = \{(\pi,d) \mid x\in\pi\}$$
$$\gamma(\mathsf{var}(x)) = \{(\pi,(ts,vs)) \mid x\in vs\}$$
$$\gamma(\mathsf{type}(\sigma)) = \{(\pi,(ts,vs)) \mid \sigma\in ts\}$$

$p\sqsubseteq p' \Leftrightarrow p = p'$

$\phi\sqsubseteq\phi' \Leftrightarrow \phi=\phi'$, 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 9. Data for backward meta-analysis for type-state analysis.

where PForm is a set of primitive formulas, and the boolean operators have the standard meanings:

$$\gamma(\mathsf{true}) = \mathbb{P}\times\mathbb{D} \quad \gamma(\mathsf{false}) = \emptyset \quad \gamma(\neg\phi) = (\mathbb{P}\times\mathbb{D})\setminus\gamma(\phi)$$
$$\gamma(\phi\wedge\phi') = \gamma(\phi)\cap\gamma(\phi') \qquad \gamma(\phi\vee\phi') = \gamma(\phi)\cup\gamma(\phi')$$

- 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 domain $\mathbb{M}$ of a disjunctive meta-analysis in a sense contains all boolean formulas which are constructed from primitive ones in PForm. We define a generic under-approximation operator for disjunctive meta-analyses as follows:

$$\mathsf{approx} \;:\; \mathbb{P}\times\mathbb{D}\times\mathbb{M}\to\mathbb{M}$$
$$\mathsf{approx}(\pi,d,\phi) = \text{let } \phi' = (\mathsf{simplify}\circ\mathsf{toDNF})(\phi) \text{ in if (number of disjuncts in } \phi'\le k) \text{ then } \phi' \text{ else } \mathsf{drop}_k(\pi,d,\phi')$$

Subroutines toDNF, simplify, and drop are defined in Figure 8. The approx operator first transforms ϕ to disjunctive normal form and removes redundant disjuncts in the DNF formula that are subsumed by other shorter disjuncts in the same formula. If the resulting formula ϕ' is simple enough in that it has no more than k disjuncts (where k is pre-determined by an analysis designer), then ϕ' is returned as the result. Otherwise, some disjuncts of ϕ' are pruned: 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 keeps only the most promising k options during exploration of a search space.

Meta-Analysis for Type-State. We define a disjunctive meta-analysis for our type-state analysis using the above recipe. Doing so means defining three entities shown in Figures 9 and 10: the set of primitive formulas, the order $\sqsubseteq$ on formulas, and a function $[\![-]\!]^b$.

The meta-analysis uses three primitive formulas. The first is err which says the d component of a pair (π,d) is $\top$. The remaining three formulas describe elements that should be included in some component of (π,d). For instance, $\mathsf{var}(x)$ says that the d component is a non-$\top$ value (ts,vs) such that the vs part contains x.

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, for each atomic command a, the meta-analysis uses transfer function $[\![a]\!]^b$, which we show in [23] satisfies requirement (2) of our framework. This requirement means that $[\![a]\!]^b$ collects all the abstractions 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.

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

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

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

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

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

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

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

Figure 11. Backward transfer function for thread-escape analysis. We use δ to range over h, v, f, and we use o to range over $\mathtt{L}, \mathtt{E}, \mathtt{N}$.

Meta-Analysis for Thread-Escape. The backward meta-analysis for the thread-escape analysis is also disjunctive. Its domain $\mathbb{M}$ is constructed from the following primitive formulas p:

$$p ::= h \to o \mid v \to o \mid f \to o$$

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 abstraction and abstract state. Formula $h \to o$ says that an abstraction π should map h to o. Formula $v \to o$ means that an abstract state d should bind v to o; formula $f \to o$ expresses a similar fact on the field f. We formalize these meanings via function $\gamma : \mathbb{M} \to 2^{\mathbb{P} \times \mathbb{D}}$ below:

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

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 found it sufficient in practice for our application, where the order is used to detect redundant disjuncts in formulas in the DNF form.

Figure 11 shows transfer function $\llbracket a \rrbracket^b$ of the meta-analysis for each atomic command a. We show in [23] that it satisfies requirement (2) of our framework which determines the semantics of the function using weakest preconditions.

Figure 6 shows the backward meta-analysis for thread-escape analysis without and with under-approximation on an example program. The trace in part (a) is generated by the forward analysis using initial abstraction $[\mathtt{h1} \mapsto \mathtt{E}, \mathtt{h2} \mapsto \mathtt{E}]$. The abstract states computed by the meta-analysis at each point of this trace without under-approximation are denoted by $\uparrow$. It correctly computes the sufficient condition for failure at the start of the trace as $\mathtt{h1} \to \mathtt{E} \vee (\mathtt{h2} \to \mathtt{E} \wedge \mathtt{h1} \to \mathtt{L})$, thus yielding the cheapest abstraction that proves the query as $[\mathtt{h1} \mapsto \mathtt{L}, \mathtt{h2} \mapsto \mathtt{L}]$. Despite taking a single iteration, however, the lack of under-approximation causes a blow-up in the size of the formula tracked by the meta-analysis.

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

Figure 10. Backward transfer function for type-state analysis.

Part (b) shows the result with under-approximation, 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, $\mathtt{h1}{\rightarrow}\mathtt{E}$, causing our technique to next run the forward analysis using abstraction $[\mathtt{h1} \mapsto \mathtt{L}, \mathtt{h2} \mapsto \mathtt{E}]$. But the analysis again fails to prove the query, and the second iteration, shown in part (b2), computes the sufficient condition for failure as $(\mathtt{h1}{\rightarrow}\mathtt{L} \wedge \mathtt{h2}{\rightarrow}\mathtt{E})$. By combining these conditions from the two iterations, our technique finds the same cheapest abstraction as that without under-approximation in part (a). Despite needing an extra iteration, the formulas tracked in part (b) are much more compact than those in part (a).

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 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 abstractions, and the second, that the negation of ϕ is expressible in $\mathbb{M}$. A ϕ satisfying these two conditions is called a ***query*** and is denoted by symbol q. The negation of a query q is also a query and is denoted $\mathsf{not}(q)$.

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 abstractions, until it proves the query or finds that the forward analysis cannot prove the query no matter what abstraction is used. The key part of TRACER is to choose a new abstraction π' to try after the forward analysis fails to prove the query using some π. TRACER does this abstraction selection using the backward meta-analysis which goes over an abstract counterexample trace of the forward analysis and computes a condition on abstractions that are necessary for proving the query. TRACER chooses a minimum-cost abstraction π' among such abstractions.

The TRACER algorithm is shown in Algorithm 1. It uses variable Π_{viable} to track abstractions that can potentially prove 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 how 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 [2, 20]. We show the correctness of our algorithm in [23].

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 π according to ⪯ such that
    F_π[s]({d_I}) ⊆ {d | (π, d) ∈ γ(q)}. Or impossibility
    meaning that ∄π : F_π[s]({d_I}) ⊆ {d | (π, d) ∈ γ(q)}.
3:  var Π_viable := ℙ
4:  while true do
5:     if Π_viable = ∅ then
6:        return impossible
7:     end if
8:     choose a minimum π ∈ Π_viable according to ⪯
9:     let D = (F_π[s]({d_I}) ∩ {d | (π, d) ∈ γ(not(q))}) in
10:       if D = ∅ then
11:          return π
12:       end if
13:       choose any τ ∈ trace(s) : F_π[τ](d_I) ∈ D
14:       let Π = {π' | (π', d_I) ∈ γ(B[τ](π, d_I, not(q)))} in
15:          Π_viable := Π_viable ∩ (ℙ \ Π)
16:       end let
17:    end let
18: end while
```

6. Experiments

We implemented our parametric dataflow analysis technique in Chord [1], an extensible program analysis framework for Java bytecode. The forward analysis is expressed as an instance of the RHS tabulation framework [19] while the backward meta-analysis is expressed as an instance of a trace analysis framework that implements our proposed optimizations.

We implemented our type-state and thread-escape analyses in our framework, and we evaluated our technique on both of them using a suite of seven real-world concurrent Java benchmark programs. Table 1 shows characteristics of these programs. All experiments were done using JDK 1.6 on Linux machines with 3.0 GHz processors and a maximum of 8GB memory per JVM process.

The last two columns of Table 1 show the size of the family of abstractions searched by each analysis for each benchmark. For the type-state analysis, it is 2^N where N is the number of pointer-typed variables in reachable methods, since an abstraction determines which variables the analysis can track in must-alias sets. For thread-escape analysis, it is 2^N where N is the number of object allocation sites in reachable methods, since the abstraction determines whether to map each such site to L or E.

We presented our technique for a single query but in practice a client may pose multiple queries in the same program. Our framework has the same effect as running our technique 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 if the sets of unviable abstractions computed so far for those queries are the same. All queries start in the same group with an empty set of unviable abstractions but split into separate groups when different sets of unviable abstractions are computed for them.

	description	# classes		# methods		bytecode (KB)		KLOC		$\log_2$(# abstractions)	
		app	total	app	total	app	total	app	total	type-state	thread-esc.
tsp	Traveling Salesman implementation	4	997	21	6,423	2.6	391	0.7	269	569	6,175
elevator	discrete event simulator	5	998	24	6,424	2.3	390	0.6	269	352	6,180
hedc	web crawler from ETH	44	1,066	234	6,881	16	442	6	283	1,400	7,326
weblech	website download/mirror tool	57	1,263	312	8,201	20	504	13	326	2,993	7,663
antlr	A parser/translator generator	118	1,134	1,180	7,786	131	532	29	303	16,563	7,748
avrora	microcontroller simulator/analyzer	1,160	2,192	4,253	10,985	224	634	64	340	37,797	10,151
lusearch	text indexing and search tool	229	1,236	1,508	8,171	101	511	42	314	14,508	7,395

Table 1. Benchmark statistics computed using a 0-CFA call graph analysis. The "total" and "app" columns report numbers with and without counting JDK library code, respectively. The last two columns determine the (log of the) number of abstractions for our two client analyses.

To avoid skewing our results by using a real type-state property to evaluate our type-state analysis, we used a fictitious one that tracks the state of every object allocated in the application code of the program (as opposed to, say, only file objects). The type-state automaton for this property has two states, $init$ and $error$. The type-state analysis tracks a separate abstract object for each allocation site h in application code that starts in the $init$ state, and transitions to $error$ upon any method call $v.m()$ in application code if the following two conditions hold: (i) v may point to an object created at site h according to a 0-CFA may-alias analysis that is used by the type-state analysis, and (ii) v is not in the current must-alias set tracked by the type-state analysis. If neither of these conditions holds, then the abstract object remains in the $init$ state, which corresponds to precise type-state tracking by the analysis.

To enable a comprehensive evaluation of our technique, we generated queries pervasively and uniformly from the application code of each benchmark. For the type-state analysis, we generated a query at each method call site, and for the thread-escape analysis, we generated a query at each instance field access and each array element access. Specific clients of these analyses may pose queries more selectively but our technique only stands to benefit in such cases by virtue of being query-driven. To avoid reporting duplicate results across different programs, we did not generate any queries in the JDK standard library, but our analyses analyze all reachable bytecode including that in the JDK.

Our type-state analysis answers each query (pc, h) such that the statement at program point pc is a method call $v.m()$ in application code and variable v may point to an object allocated at a site h that also occurs in application code. The query is proven if every object allocated at site h that variable v may refer to at the point of this call is in state $init$ (i.e., not $error$). These queries stress-test our type-state analysis since they fail to be proven if the underlying may-alias or must-alias analysis loses precision along any program path from the point at which any object is created in application code to the point at which any application method is called on it.

Our thread-escape analysis answers each query (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. Such queries may be posed by a client such as static datarace detection.

We chose type-state and thread-escape analyses as they are challenging to scale: both are fully flow- and context-sensitive analyses that use radically different heap abstractions. The thread-escape analysis is especially hard to scale: simply mapping all allocation sites to `L` in the abstraction causes the analysis to run out of memory even on our smallest benchmark. Moreover, these analyses are useful in their own right: type-state analysis is an important general analysis for object state verification while thread-escape analysis is beneficial to a variety of concurrency analyses.

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

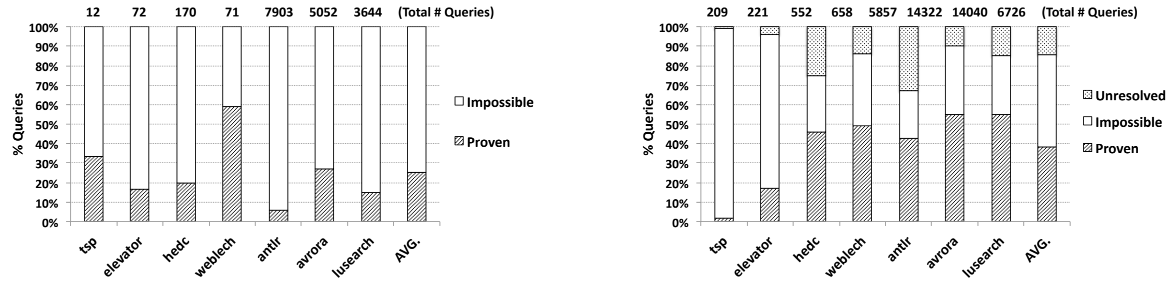
Precision. Figure 12 shows the precision of our technique. The absolute number of queries for each benchmark appears at the top. The queries are classified into three categories: those proven using a cheapest abstraction, those shown impossible to prove using any abstraction, and those that could not be resolved by our technique in 1,000 minutes (we elaborate on these queries below).

All queries are resolved in the type-state analysis. Of these queries, 25% are proven on average per benchmark. Queries impossible to prove are notably more than proven queries for the type-state analysis primarily due to the stress-test nature of the type-state property that the analysis checks. In contrast, for the thread-escape analysis, our technique proves 38% queries and it shows 47% queries impossible to prove, for a total of 85% resolved queries on average per benchmark. We manually inspected several queries that were unresolved, 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 (`L` and `E`). 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.

In summary, we found our technique useful at quantifying the limitations of a parametric dataflow analysis, and inspecting the queries deemed impossible to prove suggests what aspects of the analysis to change to overcome those limitations.

Scalability. It is challenging to scale backward static analyses. We found that underapproximation is crucial to the scalability of our backward meta-analysis: disabling it caused our technique to timeout for *all* queries even on our smallest benchmark. Recall that for disjunctive meta-analysis (Section 4.1) the degree of underapproximation can be controlled by specifying the maximum number of disjuncts k retained in the boolean formulae that are propagated backward by the meta-analysis. We found it optimal to set $k = 5$ for our two client analyses on all our benchmarks. We arrived at this setting by experimenting with different values of k. Figure 13 illustrates the effect of setting k to 1, 5, and 10 on the running time of our thread-escape analysis. We show these results only for our smallest four benchmarks as the analysis ran out of memory on the larger three benchmarks for $k = 1$ and $k = 10$. Intuitively, the reason is that doing underapproximation aggressively ($k = 1$) reduces the running time of the backward analysis in each iteration but it increases the number of iterations to resolve a query, whereas doing underapproximation passively ($k = 10$) reduces the number of iterations but increases the running time of the backward analysis in each iteration. Compared to these two extremes, setting $k = 5$ results in much fewer timeouts and better scalability overall.

Table 2 shows statistics about the number of iterations of our technique for resolved queries using $k = 5$. Minimum, maximum, and average number of iterations are shown separately for proven queries and for queries found impossible to prove. The table highlights the scalability of our technique as queries for most benchmarks are resolved in under ten iterations on average. The only exception is the type-state analysis on avrora which takes 48 iterations on average for proven queries. The reason is that, compared to the remaining benchmarks, for avrora our type-state analysis requires many more variables in the cheapest abstraction for most



(a) Type-state analysis. (b) Thread-escape analysis.

Figure 12. Precision measurements.

	number of iterations												running time of thread-escape analysis (s = seconds, m = minutes, h = hours)					
	type-state analysis						thread-escape analysis											
	proven			impossible			proven			impossible			proven			impossible		
	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg
tsp	2	2	2	1	2	1.5	2	2	2	1	1	1	14s	29s	21s	6s	8s	7s
elevator	2	3	2.1	1	9	3.8	2	3	2	1	5	1.3	12s	107s	34s	6s	144s	15s
hedc	2	3	2.3	1	2	1	2	6	2.4	1	4	1.8	17s	6m	89s	10s	5m	51s
weblech	2	3	2.1	1	3	1.4	2	17	6.9	1	3	1.2	20s	16m	5m	11s	150s	28s
antlr	2	18	8.9	1	47	7.8	2	88	3	1	18	1.4	18s	77m	98s	6s	21m	64s
avrora	2	82	47.7	1	30	3.5	2	97	3	1	38	1.4	16s	28m	67s	5s	3h	41s
lusearch	2	32	2.1	1	23	2	2	19	2.7	1	20	2.3	14s	13m	112s	6s	45m	131s

Table 2. Scalability measurements.

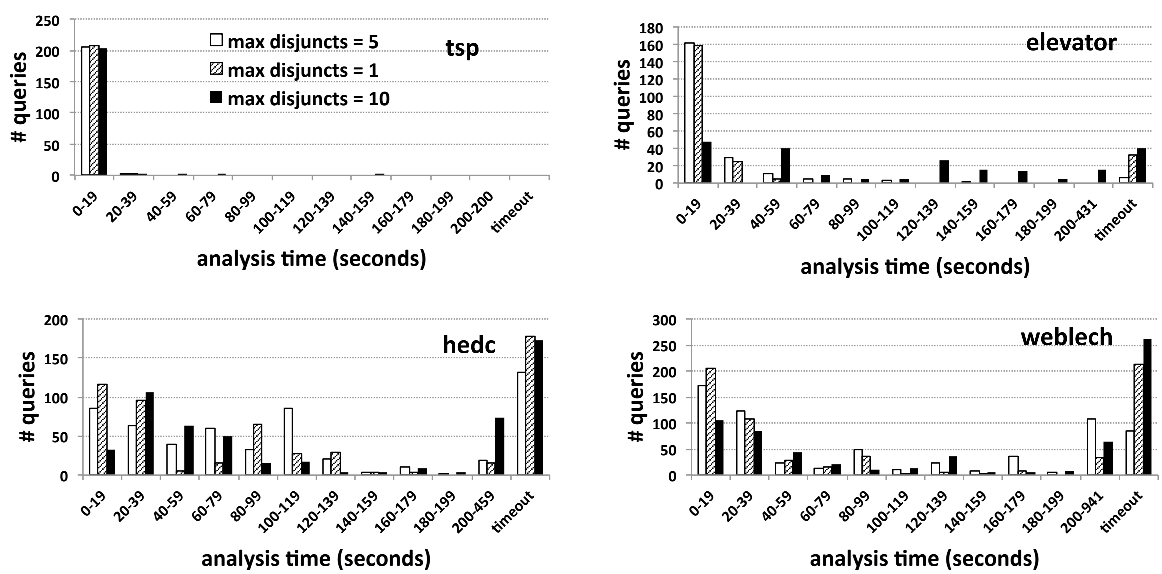


Figure 13. Running time of thread-escape analysis per query for different degrees of underapproximation $k = 1, 5, 10$ in its meta-analysis on our smallest four benchmarks. The timeout columns denote queries that could not be resolved in 1,000 minutes. Setting k to 1 or 10 resolves fewer queries than $k = 5$ for the shown benchmarks and also caused the analysis to run out of memory for the largest three benchmarks.

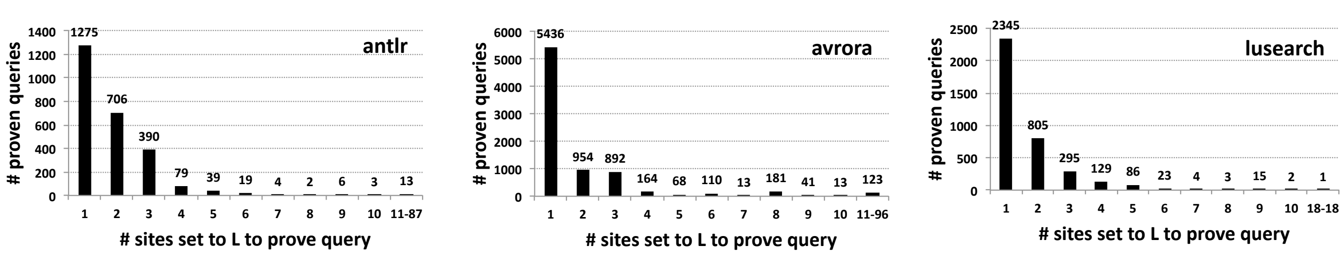


Figure 14. Sizes of the cheapest abstractions computed for proven queries of the thread-escape analysis on our largest three benchmarks.

	type-state analysis			thread-escape analysis		
	min	max	avg	min	max	avg
tsp	2	3	2.3	1	1	1
elevator	2	4	2.8	1	2	1.1
hedc	2	5	2.9	1	5	1.4
weblech	2	4	3.3	1	16	6.1
antlr	2	20	10.9	1	87	1.9
avrora	2	84	50.2	1	96	2.1
lusearch	2	34	16.5	1	18	1.7

Table 3. Statistics of cheapest abstraction size for proven queries.

	type-state analysis				thread-escape analysis			
	# groups	min	max	avg	# groups	min	max	avg
tsp	4	1	1	1	3	1	3	2
elevator	8	1	5	1.5	10	1	13	3.8
hedc	26	1	4	1.3	34	1	50	7.6
weblech	18	1	17	2.3	51	1	88	6.4
antlr	101	1	106	4.6	237	1	374	10.7
avrora	158	1	368	8.6	890	1	262	9
lusearch	151	1	139	3.6	270	1	256	13.7

Table 4. Statistics of cheapest abstraction reuse for proven queries.

of the proven queries (this is confirmed below by statistics on the sizes of the cheapest abstractions). The numbers of iterations also show that our technique is effective at finding queries that are impossible to prove since for most benchmarks it finds such queries in under four iterations on average. We also show the running time for thread-escape analysis in the table as it is relatively harder to scale than type-state analysis. For each benchmark, our technique takes one to two minutes on average for resolved queries.

Statistics of Proven Queries. We now present some useful statistics about proven queries. Table 3 shows the minimum, maximum, and average sizes of the cheapest abstraction computed by our technique for these queries. For type-state analysis, the size of the cheapest abstraction correlates the benchmark size and is greatly affected by the depth of method calls. The average number of variables that must be tracked in must-alias sets ranges from 2 to 50 from our smallest benchmark to our largest. On the other hand, thread-escape analysis only needs 1 to 2 sites mapped to `L` on average for most benchmarks, though there are queries that need upto 96 such sites (this means that no abstraction with fewer than those many sites can prove those queries). The graphs in Figure 14 show the distribution of the cheapest abstraction sizes for thread-escape analysis on our largest three benchmarks. We see that most queries are indeed proven using 1 or 2 allocation sites mapped to `L`.

Finally, it is worth finding how different the cheapest abstractions computed by our technique are for these proven queries. Table 4 answers this question by showing the numbers of queries grouped together sharing the same cheapest abstraction. The table shows that around ten or less queries on average share the same cheapest abstraction for both analyses, indicating that the cheapest abstraction tends to be different for different queries, though there are also a few large groups containing up to 368 queries for type-state analysis and 374 queries for thread-escape analysis.

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 abstractions, but on the other hand, these abstractions tend to be very different for queries from different parts of the same 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 abstraction to prove a query while we aim to find a cheapest abstraction or show that none exists. We next contrast the techniques.

CEGAR-based model checkers such as SLAM [3] and BLAST [14] compute a predicate abstraction of the given program to prove an assertion (query) in the program. Yogi [4, 9] combines CEGAR-based model checking and directed input generation to simultaneously search for proofs and violations of assertions. All these approaches can be viewed as parametric in which program predicates to use in the predicate abstraction. They differ from our approach primarily in the manner in which they analyze an abstract counterexample trace that is produced as a witness to the failure to prove a query using the currently chosen abstraction. In particular, these approaches 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, whereas our meta-analysis computes a sufficient condition for the *failure* of the given analysis to prove the query on the trace. Intuitively, our meta-analysis attempts to find as many other abstractions destined to a similar proof failure as the currently chosen abstraction; the next abstraction our approach attempts is simply a cheapest one not discarded by the meta-analysis. One advantage of the above approaches 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 the above approaches 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 [21, 22], methods or object allocation sites to be cloned [15, 18], or memory locations to be treated flow-sensitively [13]. 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., [5]) 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 analysis is a search problem that may be tackled using various algorithms with different pros and cons. Liang et al. [16] propose iterative coarsening-based algorithms that start with the most precise abstraction (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* abstraction in terms of precision as opposed to a *minimum* or *cheapest* abstraction. Naik et al. [17] use dynamic analysis to infer a necessary condition on the abstraction to prove a query. They instantiate the parametric analysis using a cheapest abstraction 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.

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- and under-approximations, and compute strongest necessary and weakest sufficient conditions for proving queries (e.g, [6, 7, 11, 12]). 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 dataflow analysis with the goal of finding a cheapest abstraction that proves a given query or showing that no such abstraction exists. Our approach is CEGAR-based and applies a novel meta-analysis on abstract counterexample traces to efficiently eliminate unsuitable abstractions. We showed the generality of our approach by applying it to two example analyses in the literature. We also showed its practicality by applying it to several real-world Java benchmark programs.

Our approach opens intriguing new problems. First, 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. Our meta-analysis relies on the existence of such a trace for scalability: the trace guides the meta-analysis in deciding which parts of the formulae it tracks represent infeasible states that can be pruned. One possibility is to generalize our meta-analysis to operate on DAG counterexamples that have been proposed for non-disjunctive analyses [10]. Second, 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 type-state analysis and our thread-escape analysis. It would be useful to devise a generic semantics-preserving simplification process to assist in compacting such formulas. Finally, manually defining the transfer functions of the meta-analysis can be tedious and error-prone. One plausible solution is to devise a general recipe for synthesizing these functions automatically from a given abstract domain and parametric analysis.

Acknowledgments

We thank Ravi Mangal for discussions and help with the implementation. We also thank the anonymous reviewers for many insightful comments. This research was supported in part by DARPA contract #FA8750-12-2-0020, awards from Google and Microsoft, and EPSRC.

References

[1] Chord:. `http://code.google.com/p/jchord/`.

[2] T. Ball and S. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *Proceedings of the ACM Workshop on Program Analysis For Software Tools and Engineering (PASTE'01)*, 2001.

[3] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, 2002.

[4] N. Beckman, A. Nori, S. Rajamani, R. Simmons, S. Tetali, and A. Thakur. Proofs from tests. *IEEE Trans. Software Eng.*, 36(4):495–508, 2010.

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

[6] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 29th ACM Conference on Programming Language Design and Implementation (PLDI'08)*, 2008.

[7] I. Dillig, T. Dillig, and A. Aiken. Fluid updates: beyond strong vs. weak updates. In *Proceedings of the 19th European Symposium on Programming (ESOP'10)*, 2010.

[8] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

[9] B. Gulavani, T. Henzinger, Y. Kannan, A. Nori, and S. Rajamani. Synergy: a new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'06)*, 2006.

[10] B. Gulavani, S. Chakraborty, A. Nori, and S. Rajamani. Automatically refining abstract interpretations. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, 2008.

[11] S. Gulwani and A. Tiwari. Assertion checking unified. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'07)*, 2007.

[12] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *Proceedings of the 35th ACM Symposium on Principles of Programming Language (POPL'08)*, 2008.

[13] S. Guyer and C. Lin. Client-driven pointer analysis. In *Proceedings of the 10th International Symposium on Static Analysis (SAS'03)*, 2003.

[14] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL'04)*, 2004.

[15] P. Liang and M. Naik. Scaling abstraction refinement via pruning. In *Proceedings of the 32nd ACM Conference on Programming Language Design and Implementation (PLDI'11)*, 2011.

[16] P. Liang, O. Tripp, and M. Naik. Learning minimal abstractions. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)*, 2011.

[17] M. Naik, H. Yang, G. Castelnuovo, and M. Sagiv. Abstractions from tests. In *Proceedings of the 39th ACM Symposium on Principles of Programming Languages (POPL'12)*, 2012.

[18] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the 9th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)*, 1994.

[19] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, 1995.

[20] 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.

[21] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 27th ACM Conference on Programming Language Design and Implementation (PLDI'06)*, 2006.

[22] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, 2005.

[23] X. Zhang, M. Naik, and H. Yang. Finding optimum abstractions in parametric dataflow analysis. Technical report, Georgia Institute of Technology, 2013. Available at `http://pag.gatech.edu/pubs/pldi13.pdf`.