Abstractions from Tests

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

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

Ghila Castelnuovo
Tel-Aviv University, Israel
ghila.castelnuovo@gmail.com

Mooly Sagiv
Tel-Aviv University, Israel
mooly.sagiv@gmail.com

Abstract

We present a framework for leveraging dynamic analysis to find good abstractions for static analysis. A static analysis in our framework is parametrised. Our main insight is to directly and efficiently compute from a concrete trace, a necessary condition on the parameter configurations to prove a given query, and thereby prune the space of parameter configurations that the static analysis must consider. We provide constructive algorithms for two instance analyses in our framework: a flow- and context-sensitive thread-escape analysis and a flow- and context-insensitive points-to analysis. We show the efficacy of these analyses, and our approach, on six Java programs comprising two million bytecodes: the thread-escape analysis resolves 80% of queries on average, disproving 28% and proving 52%; the points-to analysis resolves 99% of queries on average, disproving 29% and proving 70%.

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 Parametrised Static Analysis, Testing, Thread-Escape Analysis, Points-to Analysis, Necessary-Condition Problem

1. Introduction

Static analyses based on the Abstract Interpretation technique [6] are guaranteed to be sound: if such an analysis reports that a given query holds for a given program, it must indeed hold. In practice, however, the analysis may not be cheap enough to apply to the program, or it may not be precise enough to prove the query. This problem is inherent with the undecidability of static analysis.

One of the most interesting questions in static analysis concerns how to *specialise* a given static analysis to prove a given query. The idea is to make the analysis cheap yet precise by tailoring it to prove a specific query. Counterexample-guided abstraction refinement (CEGAR) [1, 5, 13, 19] aims to solve this problem. The idea is to gradually refine the abstraction until either a concrete counterexample is found or the query is proven. Despite many advances, the CEGAR approach includes many limitations which hinder its practicality: it is costly and may fail due to limitations of the theorem prover, the inability to appropriately refine the abstraction, the huge size of the counterexample, and the cost of the static analysis to compute the intermediate abstractions.

This paper takes a radically different approach: it uses dynamic analysis to compute good abstractions for static analysis. The static analysis is parametrised, and the space of parameter configurations is typically large, e.g., exponential in program size or even infinite. Given a query, a set of parameter configurations, and a concrete trace, our dynamic analysis computes a *necessary condition* on the parameter configurations for proving the query. Our limited experience shows that this condition can be used to choose, from among the remaining parameter configurations, one that yields an abstraction that is cheap enough to drastically cut the cost of the static analysis yet precise enough to prove the query.

One advantage of our approach is that the necessary condition is inferred efficiently and directly from the concrete trace without the need to run the static analysis on the whole program, which may be infeasible. Additionally, our method can be seen as an auditing procedure for static analysis: when an abstraction fails to satisfy our condition, it can neither prove nor disprove the query, independently of the existence of the counterexample. This means that some abstractions are pruned even before they are computed, which is in contrast to the CEGAR approach.

We do not provide a general algorithm to compute the necessary condition from a given trace. Instead, we provide a formal, non-constructive definition of the *necessary-condition problem*, and algorithms that compute the condition directly from the trace for two instance analyses: a flow- and context-sensitive thread-escape analysis, and a flow- and context-insensitive points-to analysis.

We have evaluated our approach on six real-world Java programs comprising two million bytecodes in total, the largest being 500K bytecodes. For thread-escape analysis, our approach resolves 80% of queries on average, disproving 28% and proving 52%. For points-to analysis, it resolves 99% of queries on average, disproving 29% and proving 70%. The fact that the vast majority of queries are resolved—and, more significantly, proven—shows that our approach is precise in practice. It is also scalable: despite being top-down, inter-procedural, and fully flow- and context-sensitive, our thread-escape analysis takes only 7 seconds on average over 1,750 invocations, with a maximum of 20 seconds.

To summarise, the key contributions of this paper are as follows:

POPL'12, January 25–27,2012,Philadelphia,PA,USA.

1. We present a novel approach for using concrete traces in order to obtain good abstractions in certain cases for static analysis.
2. We present a formulation of a necessary condition for an abstraction to be precise enough for a given concrete trace.
3. We provide constructive algorithms to compute the necessary condition from a concrete trace for two static analyses (thread-escape analysis and points-to analysis).
4. We provide empirical evidence that our approach can be efficiently implemented and that the resulting static analysis is both precise and scalable.

2. Example

In this section, we provide the reader a flavour of our approach using thread-escape analysis.

2.1 The thread-escape analysis problem

We introduce the thread-escape analysis problem using the example Java program in Figure 1. Object allocation sites have unique labels `h1`, `h2`, and so on, and we elide all types. Variables `u`, `v`, and `w` are locals and `g` is a global (i.e., static field). The program is multi-threaded: in each iteration of the loop, the main thread executes the statement `u.start()`, which calls the `start()` method of class `java.lang.Thread`, which asynchronously starts a new thread corresponding to the object pointed to by `u`. Thus, after the call, the main thread proceeds to the next loop iteration while the freshly started child thread runs code not shown in the figure.

The graph at the top of Figures 1(a)–(c) (ignoring the dotted boxes) shows the concrete data structure created just before program position `pc` in any iteration `i` ≥ 1 of the loop. For convenience, each object is labeled with the site at which it was created (`h1`, `h2`, etc.). The clear nodes denote *thread-local* objects and the shaded nodes denote *thread-escaping* objects. An object is thread-local if it is reachable from at most one thread, and thread-escaping otherwise. An object becomes reachable from multiple threads if it is assigned to a global or if the `start()` method of class `java.lang.Thread` is invoked on it: in the former case, the object becomes reachable from all threads in the program, while in the latter case, the object becomes reachable from at least the parent thread and the freshly started child thread. Moreover, any object reachable in the heap from a thread-escaping object is also thread-escaping. Finally, once an object thread-escapes, it remains thread-escaping for the rest of the execution. In our example, the statement `g = new h3` which writes to global `g` causes the freshly created object to thread-escape (depicted by the shaded object labeled `h3`). Likewise, the call `u.start()` causes the object pointed to by `u` to thread-escape (depicted by the shaded object labeled `h1` from an earlier iteration). Moreover, the object pointed to by `w` is reachable from this object in the heap via field `f2`; hence, that object also thread-escapes (depicted by the shaded object labeled `h4` from an earlier iteration).

We formulate the thread-escape analysis problem in the form of program queries. The query we focus on in our example is $q_{\mathsf{local}}(\mathtt{pc}, \mathtt{w})$, which is true if, whenever any thread reaches program position `pc`, the object pointed to by local variable `w` in that thread's environment is thread-local. It is easy to see that this query is true in our example: the only thread that reaches `pc` is the main thread, and whenever it does, `w` points to the object most recently created at site `h4`, which is thread-local (note that this object thread-escapes only after the following statement `u.start()` is executed). Many clients in verification, testing, optimisation, and program understanding for multi-threaded programs can benefit from answering such queries. For instance, proving our example query enables a static race detector to prove that field `id` is race-free.

2.2 Parametrised thread-escape analysis

A key challenge in proving thread-escape analysis queries lies in choosing a heap abstraction that separates thread-local objects from thread-escaping objects. One class of heap abstractions involves partitioning all objects based on some static program property. A natural way is to use a separate partition for all objects created at the same allocation site. But even this simple heap abstraction can be too costly when applied to a large program with procedures. Our goal is to derive cheaper heap abstractions that in most cases are still precise enough to prove the given query.

The first step in our approach is to parameterise the static analysis in a manner that admits a large (possibly infinite) family of abstractions, and cast the problem as a search for a good parameter configuration to use for proving a given query. Examples of such analyses abound: a safety model checker can be viewed as parametrised by a set of program predicates that dictates the *predicate abstraction* it computes, *shape analysis* is parametric in which predicates to use as *abstraction predicates*, and cloning-based points-to analyses (k-CFA, etc.) are parametrised by a vector of integers that dictate the degree of context- and object-sensitivity the analysis must use for each call site and each allocation site.

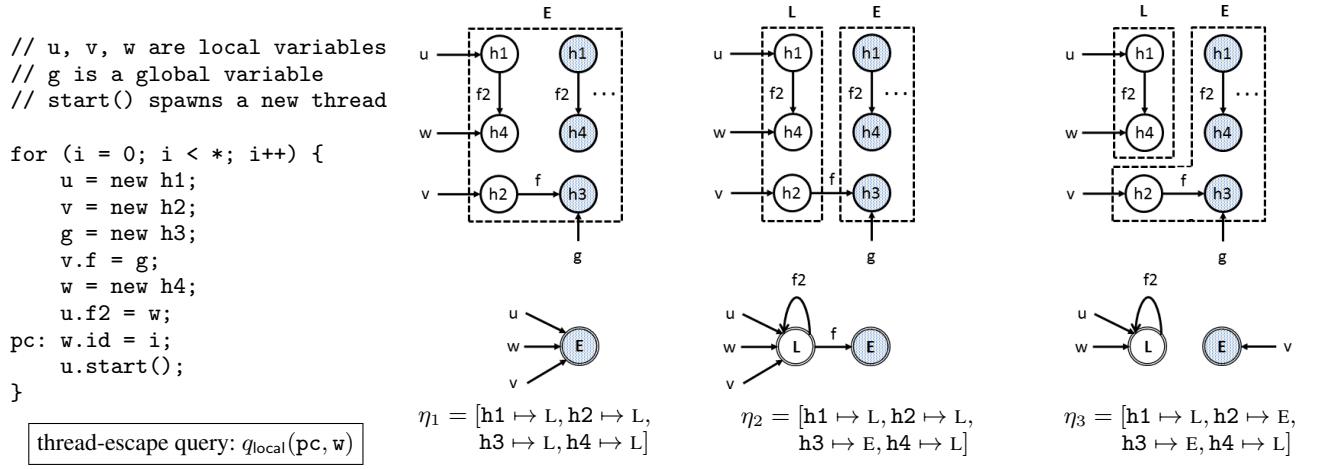
Our thread-escape analysis uses heap abstractions with two partitions, denoted L and E, that summarise "definitely thread-local" objects and "possibly thread-escaping" objects, respectively. The analysis is parametrised by the partition to be used for summarising objects created at each allocation site. A parameter configuration η thus maps each allocation site in the program to either L or E. Each object created at a particular site starts in the partition dictated by η for that site, and may subsequently migrate from L to E but not vice versa. The analysis succeeds in proving a query $q_{\mathsf{local}}(pc, x)$ if variable x does not point to an object summarised by the E partition in any abstract state at program position pc.

Our goal in the parametrised analysis setting is to efficiently find a parameter configuration that is cheap yet precise enough to prove a given query. In practice, most configurations yield abstractions that are not cheap enough to apply to the given program, or not precise enough to prove the given query. Moreover, proving different queries can require different configurations. We next illustrate the difficulty in efficiently finding good configurations. We begin by noting that enumerating and testing configurations is infeasible due to the large space from which they can be chosen.

Consider the trivial η_1 in Figure 1(a), which maps each site to L. The analysis using η_1 computes at position `pc` the abstract state shown at the bottom of the figure, with the corresponding concrete state shown above it. This analysis fails to prove our query because variable `w` points to partition E in the abstract state at position `pc`, despite *all* objects starting in partition L. This is because statement `g = new h3` "pollutes" the L partition since η_1 maps `h3` to L (recall that objects can migrate from partition L to partition E).

This leads us to η_2 in Figure 1(b), which is similar to η_1 but maps `h3` to E. The analysis using η_2 proves our query, but it is not the cheapest configuration. Our analysis reasons about reads from objects that are summarised by the L partition and so it tracks outgoing fields from such objects. Being fully flow- and context-sensitive, the analysis is exponential in the number of such fields. Hence, it is cheapest to map as few sites as possible in η to L, and thereby limit the number of such fields. In particular, it is not necessary to map `h2` to L, which causes field `f` to be tracked.

This leads us to η_3 in Figure 1(c) which is the cheapest configuration that proves our query. Note that further coarsening it fails to prove the query: mapping `h4` to E clearly fails because query variable `w` is allocated at site `h4`, but mapping `h1` to E also fails because variable `u` is allocated at site `h1` and statement `u.f2 = w` will "pollute" the L partition.



(a) fails to prove query (b) proves query but not cheapest (c) proves query and cheapest

Figure 1. Example Java program and abstract states computed by our thread-escape analysis at pc using different parameter configurations.

2.3 Parameter configurations from tests

We have seen that choosing a good parameter configuration for a given query requires striking a delicate balance between precision and scalability. The key insight in our work is to directly and efficiently infer from a concrete program trace a *necessary condition* on the parameter configurations for proving the query. Our work does not provide a general algorithm for computing the necessary condition but it serves as a principle for designing good abstractions. We provide constructive algorithms using the principle for two instance analyses: a thread-escape analysis and a points-to analysis. We also prove that these algorithms indeed compute a necessary condition: any parameter configuration not satisfying the condition will fail to prove the query.

In the case of our thread-escape analysis, the necessary condition is computed using an algorithm we call **backward pointer reachability**: Given a query $q_{\text{local}}(pc, x)$ and a concrete program state (ρ, π, σ) at location pc, where ρ and π are the environments providing the values of local and global variables respectively and σ is the heap, this algorithm dictates that any parameter configuration that can prove this query must map the allocation site of each object from which the object $\rho(x)$ is reachable in σ to L. For our example query, this algorithm outputs $[\texttt{h1} \mapsto \text{L}, \texttt{h4} \mapsto \text{L}]$. This is because, whenever program position pc is reached, variable w points to an object allocated at site h4, and the only other object from which that object is reachable in the heap is the one pointed to by variable u, which is allocated at site h1.

Note that the configuration output by the algorithm does not constrain the values of h2 and h3. Moreover, many allocation sites in the program may not even be visited in the trace. Since it is cheaper to summarise objects using the E partition instead of the L partition (recall that the analysis tracks outgoing fields of only objects in the L partition), we simply map all unconstrained sites to E. Thus, our approach yields configuration η_3 shown in Figure 1(c), which is the cheapest configuration that proves the query.

Our backward pointer reachability algorithm is not the only algorithm that satisfies our necessary condition principle. In the extreme, one could envision an algorithm that also infers a condition that is *sufficient* for the given trace. Such an algorithm, for instance, would map h3 to E (recall from Figure 1(a) that any configuration that maps h3 to L fails to prove the query). However, there is a trade-off between the amount of computation that the dynamic analysis does and the quality of the configuration it infers, and the motivation underlying our backward pointer reachability algorithm is to strike a good balance.

3. Necessary-condition problem

In this section, we explain the formal setting of our result. In particular, we define the necessary-condition problem, whose solution plays a crucial role in our approach.

DEFINITION 1. *A* **transition system** *is a triple* $(\mathcal{S}, T, I)$ *where* $T \subseteq \mathcal{S} \times \mathcal{S}$ *and* $I \subseteq \mathcal{S}$. *A* **query** q *on a transition system* $(\mathcal{S}, T, I)$ *is a function from* $\mathcal{S}$ *to* $\{\mathsf{true}, \mathsf{false}\}$.

Intuitively, $\mathcal{S}$ defines the state space of the transition system, T all the possible state changes, and I the set of initial states. A query q specifies a particular safety property of the transition system.

We assume that a transition system $(\mathcal{S}, T, I)$ and a query q are given as input to a static analysis. The goal is to prove that the query is true in all reachable states: $\forall s \in T^*(I).\ q(s) = \mathsf{true}$.

We solve this problem by using a dynamic analysis together with a parametrised static analysis. A dynamic analysis here is simply a run of the given transition system from some initial state. In our combination, this dynamic analysis is run first, and it either disproves the query or discovers a condition on the parameters of the static analysis, which should hold in order for the analysis to prove the query at all. In the latter case, this condition is converted to a particular parameter setting, and the static analysis is run with this setting to attempt to prove the query.

The transition system used by the dynamic analysis is often instrumented to track extra information about program execution in its states. One example is to record in each object the timestamp of its creation. Then, just looking at a single state provides the order of creation of objects in the state. Note that finding such an order in a standard semantics requires looking at a concrete trace. The instrumentation transforms such trace properties to state properties, which can be better exploited by our dynamic analyses.

In the following subsections, we describe our parametrised static analysis and how we combine dynamic and static analyses in detail. We fix a transition system $(\mathcal{S}_c, T_c, I_c)$ and a query q_c, and call this transition system the **concrete program**.

3.1 Parameterised static analysis

We remind the reader of a standard definition of a sound static analysis and a well-known consequence of the soundness condition:

DEFINITION 2. *A* **static analysis** *is a tuple* $(\mathcal{D}, T^\sharp, I^\sharp, \gamma)$ *of a complete lattice* $\mathcal{D}$*, a monotone function* $T^\sharp : \mathcal{D} \to \mathcal{D}$*, an element* $I^\sharp \in \mathcal{D}$ *representing the set of initial states, and a monotone function* $\gamma : \mathcal{D} \to \mathcal{P}(\mathcal{S}_c)$*. We call the function* γ **concretisation map***, and it gives the meaning of elements in* $\mathcal{D}$ *as sets of states in the concrete program. A static analysis is* **sound** *iff*

$$I_c \subseteq \gamma(I^\sharp) \ \wedge \ \forall d \in \mathcal{D}.\ T_c(\gamma(d)) \subseteq \gamma(T^\sharp(d)).$$

LEMMA 3. *If a static analysis* $(\mathcal{D}, T^\sharp, I^\sharp, \gamma)$ *is sound, every prefixpoint of* $\lambda d.\, I^\sharp \sqcup T^\sharp(d)$ *overapproximates the set* $T_c^*(I_c)$ *of reachable states in the concrete program:*

$$\forall d \in \mathcal{D}.\ \ (I^\sharp \sqcup T^\sharp(d)) \sqsubseteq d \implies T_c^*(I_c) \subseteq \gamma(d).$$

We consider static analyses with multiple parameters. We formalise the domain of parameter settings as follows:

DEFINITION 4. *A domain* $\mathsf{PConfig}$ *of* **parameter configurations** *is a set of functions from a set* Param *of parameters to a set* PVal *of parameter values. The domain does not necessarily contain all such functions, and it is ranged over using symbol* η*.*

Intuitively, parameters p in Param decide parts of the analysis to be controlled, such as the abstract semantics of a particular memory allocation in the given program. A parameter configuration η maps such p's to elements in PVal, which determine an abstraction strategy to employ for the p part of the analysis. We allow the possibility that multiple parameter configurations essentially express the same abstraction strategy, and we make this duplication explicit assuming an equivalence relation $\sim$ on $\mathsf{PConfig}$.

DEFINITION 5. *A* **parametrised static analysis** *is a family of static analyses* $\{(\mathcal{D}_\eta, T_\eta^\sharp, I_\eta^\sharp, \gamma_\eta)\}_{\eta \in \mathsf{PConfig}}$ *indexed by parameter configurations in some* $(\mathsf{PConfig}, \sim)$*. We require that equivalent components of the analysis satisfy the condition:*

$$\eta \sim \eta' \implies \{\gamma_\eta(d) \mid d \in \mathcal{D}_\eta\} = \{\gamma_{\eta'}(d) \mid d \in \mathcal{D}_{\eta'}\}.$$

A parametrised static analysis is **sound** *if all of its component analyses are sound.*

3.2 Problem description

Assume that we are given a sound parametrised static analysis $\{(\mathcal{D}_\eta, T_\eta^\sharp, I_\eta^\sharp, \gamma_\eta)\}_{\eta \in \mathsf{PConfig}}$. We now formulate the necessary-condition problem for this parametrised static analysis. A solution extracts useful information for a parametrised static analysis from results of a dynamic analysis, and it forms the main component of our combination of dynamic and static analyses.

We say that a finite subset S_d of $\mathcal{S}_c$ **validates the query** q_c when $q_c(s) = \mathsf{true}$ for every $s \in S_d$. In our setting, such a validating set S_d of states is obtained from multiple runs performed during dynamic analysis, and it carries information potentially useful for the following static analysis. One example of such information is formalised by our predicate $\mathsf{cannotProve}(\eta, S_d)$ on parameter configurations $\eta \in \mathsf{PConfig}$:[1]

$$\mathsf{cannotProve}(\eta, S_d) \iff (\forall d \in \mathcal{D}_\eta.\, S_d \subseteq \gamma_\eta(d) \implies \exists s' \in \gamma_\eta(d).\, q_c(s') = \mathsf{false}).$$

Intuitively, the predicate holds for η, when the η-component analysis cannot separate S_d from states violating the query q_c. As a result, $\mathsf{cannotProve}(\eta, S_d)$ implies that the η component cannot prove that the query holds for all the states in S_d. Using this predicate, we describe our necessary-condition problem:

> **Necessary-condition problem (in short, NC problem).** Find an algorithm that takes a finite set S_d validating the query q_c and returns a set $N \subseteq \mathsf{Param} \times \mathsf{PVal}$ satisfying the conditions below: for all $\eta \in \mathsf{PConfig}$,
>
> $$\neg(\exists \eta' \in \mathsf{PConfig}.\, \eta \sim \eta' \wedge \forall (p, v) \in N.\, \eta'(p) = v) \iff \mathsf{cannotProve}(\eta, S_d) \quad (1)$$

The left-to-right implication implies that if η is not equivalent to any parameter configuration that realises all the parameter bindings (p, v) in N, we can skip the option of setting parameters to η, because the resulting η analysis cannot prove the query. Hence, to have any hope for proving the query with the static analysis, we should ensure that some parameter configuration equivalent to our setting η respects all the bindings (p, v) in N. The phrase "necessary condition" mirrors this property of N. The other right-to-left direction is a completeness requirement, and it asks the algorithm to discover all the binding pairs that can be used to detect the satisfaction of $\mathsf{cannotProve}(-, S_d)$.

We make two further remarks on the NC problem. First, if every parameter configuration $\eta \in \mathsf{PConfig}$ satisfies $\mathsf{cannotProve}(\eta, S_d)$ so that no parameter settings can make the static analysis prove the query, the problem requires that the algorithm should return an unsatisfiable set N up to the equivalence $\sim$: for every η, no η' equivalent to η follows all the bindings in N. This usually happens when N contains two different bindings (p, v) and (p, v') for the same parameter p. Second, even if some parameter configuration equivalent to η respects all the bindings in N, the analysis with η can fail to prove the query. According to the equivalence, such η satisfies $\neg\mathsf{cannotProve}(\eta, S_d)$, but this just means that the η-component analysis has an abstract element in $\mathcal{D}_\eta$ that can overapproximate S_d without including bad states (i.e., those violating the given query). In practice, however, we found that $\neg\mathsf{cannotProve}(\eta, S_d)$ is a good indicator of the success of the analysis with η, especially when states S_d are instrumented and carry additional information about concrete traces.

A solution to the NC problem enables an interesting combination of a dynamic analysis and a parametrised static analysis. In this combination, a dynamic analysis is first run, and it gives a set of states $s_1, \ldots, s_n$ that are reachable from some initial states in I_c. If the given query q_c does not hold for some s_i, we have found a counterexample, and the combined analysis terminates with this counterexample s_i. Otherwise, the solution to the NC problem is run for $s_1, \ldots, s_n$, and then it computes a set N. The combined analysis then checks the unsatisfiability of N as follows:

$$\exists p \in \mathsf{Param}.\, \exists v, v' \in \mathsf{PVal}.\ (p, v) \in N \wedge (p, v') \in N \wedge v \neq v'.$$

If the check goes through, the analysis stops and returns "impossible to prove". Otherwise, it picks one element v_0 from PVal (which normally makes the static analysis run fast), and constructs a parameter configuration η_N as follows:

$$\eta_N(p) = \text{if } ((p, v) \in N \text{ for some } v) \text{ then } v \text{ else } v_0. \quad (2)$$

The element v_0 is chosen carefully so that η_N belongs to $\mathsf{PConfig}$. Finally, the analysis with the parameter setting η_N is run on the given program. In what follows, we explain how to solve the NC problem for instance analyses.

4. Generic solution

We have developed solutions to the NC problem for two instance analyses. This section is a prelude of the description of these solutions, where we explain their commonalities. In particular, we clar-

[1] The cannotProve predicate is related to so called supervaluation of abstract elements. The formula $\neg\mathsf{cannotProve}(\eta, S_d)$ holds iff there is an abstract element d in $\mathcal{D}_\eta$ overapproximating S_d and containing only those concrete states where q_c evaluates to true. The latter property of d is the definition of q_c having the super-truth at d.

ify an assumption made by both solutions on parametrised static analyses and queries, and describe a recipe for developing an algorithm for the NC problem, called **generic solution**, from which the solutions can be derived. This recipe can also be used for other instance analyses. As in the previous section, we assume a fixed concrete program $(\mathcal{S}_c, T_c, I_c)$ and a fixed query q_c on $\mathcal{S}_c$.

Our generic solution requires that a parametrised static analysis $\{(\mathcal{D}_\eta, T^\sharp_\eta, I^\sharp_\eta, \gamma_\eta)\}_{\eta \in \mathsf{PConfig}}$ should **have coupled components**, which means that the following three conditions hold:

1. The component static analyses use the same abstract domain. That is, $\mathcal{D}_{\eta_0} = \mathcal{D}_{\eta_1}$ for all $\eta_0, \eta_1 \in \mathsf{PConfig}$. We let $\mathcal{D}$ be this common abstract domain. Note that we do not impose a similar requirement on γ_η and component analyses can, therefore, use different concretisation maps. This means that although the components share the same set $\mathcal{D}$ of abstract representations, they can still have different abstract domains, because they might interpret these representations differently.
2. For all $\eta_0, \eta_1 \in \mathsf{PConfig}$ and all $d \in \mathcal{D}$,

   $$(\forall s \in \gamma_{\eta_0}(d).\, q_c(s) = \mathsf{true}) \iff (\forall s \in \gamma_{\eta_1}(d).\, q_c(s) = \mathsf{true}).$$

   Note that both sides of the equivalence are the same except the subscripts η_0 and η_1. The common part

   $$\forall s \in \gamma_{-}(d).\, q_c(s) = \mathsf{true} \tag{3}$$

   means that query q_c holds for all states abstracted by d. Hence, if the static analysis with a certain parameter configuration returns such d, it can prove that query q_c holds for all the reachable states of the given concrete program. For this reason, we call d satisfying formula (3) a **good abstract element**. Because of the equivalence above, the identification of such good abstract elements does not depend on the choice of a parameter configuration η, and we use some $\eta \in \mathsf{PConfig}$ and define the set of good elements D_g as follows:

   $$D_g = \{d \mid \forall s \in \gamma_\eta(d).\, q_c(s) = \mathsf{true}\}$$

3. There are a finite lattice Aux_s, and monotone functions $F_s : \mathsf{Aux}_s \to \mathsf{Aux}_s$ and $G_s : \mathsf{Aux}_s \to \mathcal{P}(\mathsf{Param} \times \mathsf{PVal})$ for each $s \in \mathcal{S}_c$, such that the equivalence below holds for all finite subsets $S_d \subseteq \mathcal{S}_c$:

   $$\begin{aligned}(\exists d \in D_g.\, S_d \subseteq \gamma_\eta(d)) \iff \\ \exists \eta'.\, \eta \sim \eta' \wedge (\forall s \in S_d.\, \exists a \in \mathsf{Aux}_s.\, (F_s(a) \sqsubseteq a) \\ \wedge\, \forall (p, v) \in G_s(a).\, \eta'(p) = v).\end{aligned}$$

   The left side of the equivalence means that the static analysis with η can use a good abstract element $d \in D_g$ to abstract a given set of states S_d. Hence, it can at least separate S_d from bad states where query q_c gets evaluated to false. According to the equivalence, this property on η can be checked using F and G. That is, we iterate over all parameter configurations η' equivalent to η, and do the following. For every state $s \in S_d$, we compute some pre-fixpoint of F_s over Aux_s, map this pre-fixpoint to a subset N_0 of $\mathsf{Param} \times \mathsf{PVal}$ using G_s, and check whether η' respects all the bindings in N_0. If the check succeeds, we stop the iteration, and return "yes". If the iteration finishes without any successful check, we return "no".

Assume that we are given a sound parametrised analysis that has coupled components. Our generic algorithm for solving the NC problem is given in Figure 2. Given a finite set S_d validating query q_c, the algorithm iterates over every element $s \in S_d$, and computes the least fixpoint $\mathsf{leastFix}\ F_s$, which is mapped to a subset N_s of $\mathsf{Param} \times \mathsf{PVal}$ by G_s. The resulting subsets N_s from iterations are combined, and become the result $N = \bigcup_{s \in S_d} N_s$.

Input: a finite subset $S_d \subseteq \mathcal{S}_c$ such that $\forall s \in S_d.\, q_c(s) = \mathsf{true}$
Output: the finite subset $N \subseteq \mathsf{Param} \times \mathsf{PVal}$ computed by

$$N = \bigcup \{G_s(a) \mid s \in S_d \wedge a = \mathsf{leastFix}\ F_s\}$$

Figure 2. Generic algorithm for solving the NC problem.

Note that the least fixpoint of F_s exists and can be computed by the standard method (which generates $\bot, F_s(\bot), F_s(F_s(\bot)), \ldots$ until the fixpoint is reached), because Aux_s is a finite lattice (hence complete) and F_s is monotone.

THEOREM 6. *Our generic algorithm solves the NC problem.*

Proof: Let N be the result of our algorithm when it is given a set S_d of states as the input. We need to prove that for every parameter configuration $\eta \in \mathsf{PConfig}$, $\mathsf{cannotProve}(\eta, S_d)$ holds if and only if we cannot find η' equivalent to η such that $\eta'(p) = v$ for all $(p, v) \in N$. We will discharge this proof obligation by showing:

$$\neg\mathsf{cannotProve}(\eta, S_d) \iff \exists \eta'.\, \eta \sim \eta' \wedge \forall (p, v) \in N.\, \eta'(p) = v.$$

We first transform the left side of the equivalence by unrolling the definition of $\mathsf{cannotProve}$ and using the first condition of having coupled components, which says that $\mathcal{D}_\eta = \mathcal{D}$ for all η:

$$\begin{aligned}&\neg\mathsf{cannotProve}(\eta, S_d) \\ &\iff \neg(\forall d \in \mathcal{D}.\, S_d \subseteq \gamma_\eta(d) \implies \exists s' \in \gamma_\eta(d).\, q_c(s') = \mathsf{false}) \\ &\iff \exists d \in \mathcal{D}.\, S_d \subseteq \gamma_\eta(d) \wedge \forall s' \in \gamma_\eta(d).\, q_c(s') = \mathsf{true}.\end{aligned}$$

In the first equivalence, we unroll the definition of $\mathsf{cannotProve}$ and replace $\mathcal{D}_\eta$ by $\mathcal{D}$ in the result of unrolling. The second equivalence is the standard one from classical logic.

Next, we use the second condition of having coupled components, which allows us to define the set D_g of good abstract elements independently of a parameter configuration η:

$$\begin{aligned}&\exists d \in \mathcal{D}.\, S_d \subseteq \gamma_\eta(d) \wedge \forall s' \in \gamma_\eta(d).\, q_c(s') = \mathsf{true} \\ &\iff \exists d \in \mathcal{D}.\, S_d \subseteq \gamma_\eta(d) \wedge d \in D_g \\ &\iff \exists d \in D_g.\, S_d \subseteq \gamma_\eta(d) \\ &\iff \exists \eta'.\, \eta \sim \eta' \wedge \exists d \in D_g.\, S_d \subseteq \gamma_{\eta'}(d)\end{aligned} \tag{4}$$

The first equivalence uses the definition of D_g, and the second equivalence and the left-to-right implication of the third follow from standard reasoning in classical logic. The remaining right-to-left implication in the last equivalence relies on the following condition on $\sim$:

$$\eta \sim \eta' \implies \{\gamma_\eta(d) \mid d \in \mathcal{D}\} = \{\gamma_{\eta'}(d) \mid d \in \mathcal{D}\}$$

as well as the fact that if $\gamma_\eta(d) = \gamma_{\eta'}(d')$, then both d and d' belong to D_g, or neither does so.

Finally, we use the last condition of having coupled components to reach the property on the result N of our generic algorithm. By the third condition of having coupled components, the property in (4) is equivalent to

$$\begin{aligned}\exists \eta', \eta''.\, \eta \sim \eta' \wedge \eta' \sim \eta'' \wedge (\forall s \in S_d. \exists a \in \mathsf{Aux}_s.\, F_s(a) \sqsubseteq a \\ \wedge\, \forall (p, v) \in G_s(a).\, \eta''(p) = v).\end{aligned}$$

Since $\sim$ is an equivalence relation, the property above means the same as the following condition:

$$\exists \eta'.\, \eta \sim \eta' \wedge \forall s \in S_d. \exists a.\, F_s(a) \sqsubseteq a \wedge \forall (p, v) \in G_s(a).\, \eta'(p) = v.$$

In what follows, we transform the second conjunct of the condition until we reach our target property for N:

$$\begin{aligned}
&(\forall s \in S_d.\, \exists a.\ F_s(a) \sqsubseteq a \wedge \forall (p,v) \in G_s(a).\ \eta'(p) = v) \\
&\iff \forall s \in S_d.\, \forall (p,v) \in G_s(\mathsf{leastFix}\ F_s).\ \eta'(p) = v \\
&\iff \forall (p,v) \in \{G_s(\mathsf{leastFix}\ F_s) \mid s \in S_d\}.\ \eta'(p) = v \\
&\iff \forall (p,v) \in N.\ \eta'(p) = v.
\end{aligned}$$

The last two equivalences are standard equivalence-preserving steps from logic. The right-to-left direction of the first equivalence holds, because $F_s(\mathsf{leastFix}\, F_s) \sqsubseteq \mathsf{leastFix}\, F_s$. Now, it remains to show the left-to-right direction of the same equivalence. To do so, we pick $s \in S_d$, and assume that for some $a_0 \in \mathsf{Aux}_s$,

$$F_s(a_0) \sqsubseteq a_0 \wedge \forall (p,v) \in G_s(a_0).\, \eta(p) = v. \tag{5}$$

Since F_s is monotone on a complete lattice, $\mathsf{leastFix}\ F_s$ is the least element in Aux_s satisfying $F_s(a) \sqsubseteq a$. Hence,

$$\mathsf{leastFix}\ F_s \sqsubseteq a. \tag{6}$$

Furthermore, since G_s is monotone, $G_s(\mathsf{leastFix}\ F_s)$ is a subset of $G_s(a)$. This subset relationship and the second conjunct of the property (5) for a_0 imply that

$$\forall (p,v) \in G_s(\mathsf{leastFix}\ F_s).\, \eta(p) = v \tag{7}$$

We get the right side of the second equivalence from (6) and (7). □

5. Instance analyses

Following our general recipe for solving the NC problem, we have developed algorithms for solving the problem for two instance analyses. In this section, we describe these algorithms. We start with a model of concrete program states. This storage model is used by both instance analyses, as it is or in a slightly adjusted form.

Our storage model defines a set of concrete states of a given heap-manipulating program. It assumes the nonempty set PC of program positions in the given program. Also, the model assumes five nonempty disjoint sets: a finite set LVar for local variables, another finite set GVar for global variables, yet another finite set Fld for fields, and two countable sets, Loc for objects and $\mathsf{AllocSite}$ for allocation sites[2]. The formal definition of our model is given by these equations:

$$\begin{aligned}
\mathsf{ILoc} &= \mathsf{AllocSite} \times \mathsf{Loc} \qquad \mathsf{Val} = \mathsf{ILoc} \cup \{\mathsf{nil}\} \\
\mathsf{Local} &= \mathsf{LVar} \to \mathsf{Val} \qquad \mathsf{Global} = \mathsf{GVar} \to \mathsf{Val} \\
\mathsf{Heap} &= \mathsf{ILoc} \times \mathsf{Fld} \rightharpoonup_{fin} \mathsf{Val} \\
\mathcal{S}_{\mathsf{base}} &= \{(pc, \rho, \pi, \sigma) \in \mathsf{PC} \times \mathsf{Local} \times \mathsf{Global} \times \mathsf{Heap} \mid \\
&\qquad \text{there are no dangling pointers in } \rho,\, \pi \text{ and } \sigma\}
\end{aligned}$$

Intuitively, $(h, o) \in \mathsf{ILoc}$ means an object o instrumented with its allocation site h. Such instrumented objects and nil form the set of values. The storage model defines states as tuples of four components. [3] The first is the current program position pc, and the other two, denoted ρ and π, hold the values of local variables and global variables, respectively. The last component is the heap σ with finitely many allocated objects. Note that we treat local and global variables separately. This separation helps formulate one of our instance analyses below.

We consider two analyses over the storage model:

1. **Thread-escape analysis:** It attempts to prove that at a given program position pc, a particular *local* variable x never stores an object that is reachable from any *global* variable. Note the use of our separation of local variables from global variables.
2. **Points-to analysis:** This analysis tries to show that program variables x and y always point to different heap objects at all program positions.

The objectives of both analyses have the same format, and demand the proof that a certain query should hold for all reachable program states. We denote these queries by $q_{\mathsf{local}}(pc_q, x_q)$ and $q_{\mathsf{noalias}}(x_q, y_q)$.

5.1 Thread-escape analysis

Our first instance analysis is a fully flow- and context-sensitive thread-escape analysis. It answers the query $q_{\mathsf{local}}(pc_q, x_q)$, which asks whether, at program position pc_q, local variable x_q never points to an object that is reachable from global variables.

The thread-escape analysis summarises objects in a program state using two abstract locations L and E, such that L abstracts nil and a set of objects that are not reachable from global variables, and E abstracts the set of all remaining objects in the state. Thus, E includes all objects reachable from any global variable, and possibly more. The abstract domain tracks outgoing fields from objects summarised by L, and it is defined as follows:

$$\begin{aligned}
\mathsf{Val}^\sharp &= \{\mathrm{L}, \mathrm{E}\} \qquad \mathsf{Local}^\sharp = \mathsf{LVar} \to \mathcal{P}(\mathsf{Val}^\sharp) \\
\mathsf{Heap}^\sharp &= \mathsf{Fld} \to \mathcal{P}(\mathsf{Val}^\sharp) \qquad \mathcal{D} = \mathsf{PC} \to \mathsf{Local}^\sharp \times \mathsf{Heap}^\sharp
\end{aligned}$$

An abstract heap is a map from fields to sets of abstract locations. This map concerns only the objects summarised by L, and overapproximates the values stored in the fields of these objects. Note that we do not track values stored in global variables or objects summarised by E. This is our intentional choice based on the following observation: if an object is reachable from a global variable, it usually remains so, and as a result, tracking what are stored in such escaping objects does not normally help improve precision of the analysis.

The thread-escape analysis is an instance of our parametrised static analysis that has coupled components. Its parameters are defined as follows:

$$\begin{aligned}
\mathsf{Param} &= \mathsf{AllocSite} \qquad \mathsf{PVal} = \{\mathrm{L}, \mathrm{E}\} \\
\mathsf{PConfig} &= \mathsf{Param} \to \mathsf{PVal} \qquad \eta \sim \eta' \iff \eta = \eta'
\end{aligned}$$

Parameters are allocation sites, and parameter configurations η map them to one of the abstract locations L and E. Setting an allocation site h to $v \in \mathsf{PVal}$ entails that objects allocated at h are summarised initially by v. This initial membership of a newly created object can change, but only in a limited manner: an object can move from L to E but not vice versa. In this way, a parameter configuration controls how objects are abstracted using L and E, and it affects the precision and scalability of the analysis, because the analysis generally tracks information about L more precisely but at a higher cost. Finally, the thread-escape analysis does not have symmetry among component analyses, so that the equivalence relation on parameter configurations is simply the equality.

The intuition described so far on the analysis is formalised by our concretisation map γ_η, which we will now explain. For a set $L \in \mathcal{P}(\mathsf{ILoc})$ of objects, let $\mathsf{abs}_L : \mathsf{Val} \to \mathsf{Val}^\sharp$ be the following function, which abstracts concrete values in Val:

$$\mathsf{abs}_L(v) = \text{if } (v \in L \cup \{\mathsf{nil}\}) \text{ then L else E}.$$

The subscript L here provides the meaning of the abstract location L, which our abstraction function abs exploits in a standard way to abstract concrete values. Using this value abstraction, we define

[2] Although there are infinitely many values for denoting allocation sites, only a finite subset of them are used in a given program, because the program includes only finitely many instructions for memory allocations.

[3] The formal statement of the absence of dangling pointers is: $\mathsf{range}(\rho) \cup \mathsf{range}(\pi) \cup \mathsf{range}(\sigma) \subseteq \{l \mid \exists f.\, (l, f) \in \mathsf{dom}(\sigma)\} \cup \{\mathsf{nil}\}$.

concretisation maps $\gamma_\eta : \mathcal{D} \to \mathcal{P}(\mathcal{S}_{\mathsf{base}})$ as follows:

$$
\begin{aligned}
(pc, \rho, \pi, \sigma) \in \gamma_\eta(d) \iff\ & \exists \rho^\sharp, \sigma^\sharp.\ d(pc) = (\rho^\sharp, \sigma^\sharp) \wedge \\
& (\exists L \in \mathcal{P}(\mathsf{ILoc}).\ (\forall x \in \mathsf{LVar}.\ \mathsf{abs}_L(\rho(x)) \in \rho^\sharp(x)) \\
& \wedge (\forall (l, f) \in (L \times \mathsf{Fld}) \cap \mathsf{dom}(\sigma).\ \mathsf{abs}_L(\sigma(l, f)) \in \sigma^\sharp(f)) \\
& \wedge (\forall (l, f) \in (\mathsf{ILoc} \times \mathsf{Fld}) \cap \mathsf{dom}(\sigma).\ \sigma(l, f) \in L \implies l \in L) \\
& \wedge (\forall g \in \mathsf{GVar}.\ \pi(g) \notin L) \wedge (\forall (h, a) \in L.\ \eta(h) = \mathrm{L})).
\end{aligned}
$$

This definition requires that the abstract location L should have an appropriate interpretation as a set L of objects, with respect to the abstract stack $\rho^\sharp$ and heap $\sigma^\sharp$ at pc. By the word "appropriate", we mean that L should satisfy the five conjuncts given above. The first two of these conjuncts express that $\rho^\sharp$ and $\sigma^\sharp$ overapproximate values that are stored in local variables and in fields of objects in L. The next two are concerned with L containing only objects that are unreachable from global variables. They say that L is closed under backward pointer reachability and it does not contain any object stored in any global variable. Hence, when these conjuncts hold, no object in L can be reached from global variables. Finally, the last conjunct says that all objects in L are allocated at sites mapped by η to L. Equivalently, it says that L never contains objects from sites mapped by η to E. This is the place where the concretisation depends on the parameter configuration η, and the conjunct describes a unique property of the η-component analysis, which holds because objects from sites mapped to E are abstracted using E and this membership in E never changes during the analysis.

We order $\mathcal{P}(\mathsf{Val}^\sharp)$ using the subset relation, and $\mathsf{Local}^\sharp$ and $\mathsf{Heap}^\sharp$ by the pointwise extension of this subset order. Then, from these order relations of $\mathsf{Local}^\sharp$ and $\mathsf{Heap}^\sharp$, we construct the order on our abstract domain $\mathcal{D}$, again using a standard pointwise extension for the product and function spaces.

LEMMA 7. *The abstract domain $\mathcal{D}$ is a complete lattice. Furthermore, γ_η is monotone for every $\eta \in \mathsf{PConfig}$.*

5.1.1 NC algorithm

Assume that we are given a query $q_{\mathsf{local}}(pc_q, x_q)$ for some program position pc_q and a local variable x_q. Our NC algorithm for this query takes a finite set $S_d \subseteq \mathcal{S}_{\mathsf{base}}$ such that every state $s \in S_d$ satisfies the query. Then, the algorithm computes a subset N of $\mathsf{Param} \times \mathsf{PVal}$, which describes a necessary condition for proving the query, as formulated by the equivalence (1) in Section 3.

Our algorithm works as follows. Given an input S_d, it iterates over every state $s = (pc, \rho, \pi, \sigma) \in S_d$ with $pc = pc_q$, and calculates **backward pointer reachability**, starting from the queried object $\rho(x_q)$. Concretely, the backward reachability first looks up the object stored in variable x_q in the state s, then it computes all the objects that reach object $\rho(x_q)$ via fields in the state, and finally it takes the allocation sites A_s of the resulting objects and builds the set of parameter binding $N_s = \{(h, \mathrm{L}) \mid h \in A_s\}$. Once all the iterations are completed, the algorithm gathers the N_s's and returns their union $N = \bigcup_{s \in S_d} N_s$ as a result.

Formally, the NC algorithm is an instantiation of the generic solution in Figure 2, with the following data specific to the thread-escape analysis:

1. The first datum is the sub-domain $D_g \subseteq \mathcal{D}$ of good abstract elements, whose concretisations do not contain bad states violating $q_{\mathsf{local}}(pc_q, x_q)$. This property of abstract elements should hold regardless of what parameter configuration η is chosen to do the concretisation. Such a sub-domain D_g exists for the thread-escape analysis, and it has the definition: $D_g = \{d \mid \forall \rho^\sharp, \sigma^\sharp.\ (d(pc_q) = (\rho^\sharp, \sigma^\sharp) \wedge \forall x.\ \rho^\sharp(x) \neq \emptyset) \implies \rho^\sharp(x_q) = \{\mathrm{L}\}\}$.
2. The second datum is a finite lattice Aux_s for each state $s = (pc, \rho, \pi, \sigma) \in \mathcal{S}_{\mathsf{base}}$. In the case of the thread-escape analysis, $\mathsf{Aux}_s = \mathcal{P}(\{l \mid \exists f \in \mathsf{Fld}.\ (l, f) \in \mathsf{dom}(\sigma)\} \cup \{\mathsf{nil}\})$.
3. The remaining data are monotone functions $F_s : \mathsf{Aux}_s \to \mathsf{Aux}_s$ and $G_s : \mathsf{Aux}_s \to \mathcal{P}(\mathsf{Param} \times \mathsf{PVal})$ for all $s \in \mathcal{S}_{\mathsf{base}}$:

$$
\begin{aligned}
F_{(pc,\rho,\pi,\sigma)}(L) &= \{\rho(x_q) \mid pc = pc_q\} \cup L \\
&\quad \cup \{l \mid \exists f \in \mathsf{Fld}.\ (l, f) \in \mathsf{dom}(\sigma) \wedge \sigma(l, f) \in L \cup \{\mathsf{nil}\}\} \\
G_{(pc,\rho,\pi,\sigma)}(L) &= \{(h, \mathrm{L}) \mid \exists o.\ (h, o) \in L\} \\
&\quad \cup \{(h, \mathrm{E}) \mid \exists o, g.\ (h, o) \in L \wedge (h, o) = \pi(g)\}
\end{aligned}
$$

   The first function F_s comes from the query $q_{\mathsf{local}}(x_q, pc_q)$ and a condition on L in the concretisation γ_η, which says that L should be closed under backward pointer reachability. The function computes one-step backward closure of the set L, and extends the result with the object $\rho(x_q)$ stored in x_q. The second function $G_s(L)$ collects all the allocation sites appearing in L, and turns them to conditions that those sites should be mapped to L. When L contains an object (h, o) stored in some global variable (so the object $\rho(x_q)$ is escaping), $G_s(L)$ adds both (h, L) and (h, E), so that no parameter configurations can satisfy all the bindings in $G_s(L)$. The computation of the fixpoint of F_s and its conversion via G_s are the formal implementation of the backward reachability calculation alluded to in our informal explanation of the algorithm above.

LEMMA 8. *The data $D_g, \mathsf{Aux}_s, F_s, G_s$ satisfy the conditions in Section 4. Hence, our parametrised thread-escape analysis has coupled components. Also, the induced NC algorithm solves the NC problem for the thread-escape analysis.*

5.1.2 Construction of a parameter configuration

The result N of our NC algorithm needs to be converted to a specific parameter configuration, so that our parametrised thread-escape analysis can be instantiated with that configuration. We use a simple conversion described at the end of Section 3. If N contains two different bindings for a single allocation site, we return "impossible to prove." Otherwise, we construct a parameter configuration $\eta(h) = (\text{if } (h, v) \in N \text{ then } v \text{ else } \mathrm{E})$. Note that in the construction, we chose E as a default value. Usually, setting an allocation site to L makes the analysis more precise, but slower as well. Hence, our choice of E corresponds to using the most abstract and also cheapest component of the analysis, which can still separate good states S_d obtained by the dynamic analysis from bad states violating the query $q_{\mathsf{local}}(pc_q, x_q)$.[4]

5.2 Points-to analysis

Our second instance analysis is a flow- and context-insensitive points-to analysis. It answers the query $q_{\mathsf{noalias}}(x_q, y_q)$, which asks whether program variables x_q and y_q point to different objects at all program positions.

Our version of the points-to analysis uses parameter configurations to optimise an existing flow- and context-insensitive points-to analysis. The purpose of looking for such optimisation is not to improve the existing analysis. It is well-known that the existing analysis scales. Rather, our purpose is to test whether our approach of combining dynamic and static analyses can produce a cheap abstraction that is still good enough for proving a given query.

[4] Our thread-escape analysis finds a minimal abstraction for proving a given query in the following sense. Consider a partial order L, E defined by $\mathrm{L} \sqsubseteq \mathrm{E}$, and extend this order to parameter configurations pointwise. A successful run of our analysis computes a minimal parameter configuration according to this extended order. If one accepts that this order correctly compares the degree of abstractions of parameter configurations, she or he can see that the computed parameter configuration is also a minimal abstraction.

Our points-to analysis abstracts objects using three abstract locations P1, P2 and P3. These abstract locations form a partition of all objects, and they are used to describe aliasing relationships among program variables and fields that arise during program execution. Based on this intuition on abstract locations, we define the abstract domain of our points-to analysis:

$$\mathsf{Loc}^\sharp = \{\text{P1}, \text{P2}, \text{P3}\} \qquad \mathsf{Var} = \mathsf{LVar} \cup \mathsf{GVar}$$
$$\mathsf{Stack}^\sharp = \mathsf{Var} \to \mathcal{P}(\mathsf{Loc}^\sharp) \quad \mathsf{Heap}^\sharp = \mathsf{Loc}^\sharp \times \mathsf{Fld} \to \mathcal{P}(\mathsf{Loc}^\sharp)$$
$$\mathcal{D} = \mathsf{Stack}^\sharp \times \mathsf{Heap}^\sharp$$

An abstract state $(\pi^\sharp, \sigma^\sharp)$ conservatively describes all objects stored in program variables and fields. Note that a program position is not a part of an abstract state. This omission implies that our abstract state $(\pi^\sharp, \sigma^\sharp)$ specifies a flow-insensitive property of a given program, as expected for any flow-insensitive static analyses.

Our points-to analysis is parametrised by maps from allocation sites to abstract locations:

$$\mathsf{Param} = \mathsf{AllocSite} \qquad \mathsf{PVal} = \mathsf{Loc}^\sharp$$
$$\mathsf{PConfig} = \{\eta : \mathsf{Param} \to \mathsf{PVal} \mid \forall l^\sharp \in \mathsf{Loc}^\sharp.\, \exists h.\, \eta(h) = l^\sharp\}$$
$$\eta \sim \eta' \iff (k \circ \eta = \eta' \text{ for some bijection } k \text{ on } \mathsf{PVal})$$

A parameter configuration $\eta \in \mathsf{PConfig}$ decides, at each allocation site, which abstract location to use to summarise objects created at the site. Since there are only three abstract locations, all sites are partitioned into three groups, each of which is summarised using one abstract location. Unlike the thread-escape analysis, once an object is summarised by an abstract location, say P1, this summary relationship never changes during the analysis, so the object never becomes summarised by P2 or P3 later. For each parameter configuration $\eta \in \mathsf{PConfig}$, we care only about how η partition allocation sites into three groups, not about the names of these groups. Whether groups are named $(\text{P1}, \text{P2}, \text{P3})$ or $(\text{P2}, \text{P3}, \text{P1})$ does not matter for the behavior of the analysis. This independence on names is made explicit by our equivalence relation $\sim$ on parameter configurations above.

The way that parameter configurations control the analysis here can be seen in our concretisation map γ_η, which we present next. For sets of objects $L_1, L_2 \subseteq \mathsf{ILoc}$ with $L_1 \cap L_2 = \emptyset$, let $\mathsf{abs}_{L_1,L_2} : \mathsf{Val} \to \mathcal{P}(\mathsf{Loc}^\sharp)$ be the following function that abstracts concrete values:

$$\mathsf{abs}_{L_1,L_2}(v) = \text{if } (v = \mathsf{nil}) \text{ then } \{\} \text{ else}$$
$$(\text{if } v \in L_1 \text{ then } \{\text{P1}\} \text{ else (if } v \in L_2 \text{ then } \{\text{P2}\} \text{ else } \{\text{P3}\}))$$

Note the role of subscripts L_1 and L_2. They give the meaning of P1 and P2, and guide the function to abstract concrete objects according to this meaning. These subscripts are usually constructed by taking the inverse image from a parameter configuration:

$$(L_1^\eta,\, L_2^\eta) = (\{(h, a) \mid \eta(h) = \text{P1}\},\, \{(h, a) \mid \eta(h) = \text{P2}\}).$$

Another thing to notice is that the concrete value nil gets abstracted to the empty set. Hence, every abstract value $v \in \mathcal{P}(\mathsf{Loc}^\sharp)$ represents a non-empty set of concrete values, which contains nil. Using both the value abstraction and the subscript generation explained so far, we define the concretisation map γ_η as follows:

$$(pc, \rho, \pi, \sigma) \in \gamma_\eta(\pi^\sharp, \sigma^\sharp) \iff$$
$$(\forall x \in \mathsf{LVar}.\, \mathsf{abs}(\rho(x)) \subseteq \pi^\sharp(x)) \wedge (\forall g \in \mathsf{GVar}.\, \mathsf{abs}(\pi(g)) \subseteq \pi^\sharp(g))$$
$$\wedge\, (\forall (l, f) \in (L_1^\eta \times \mathsf{Fld}) \cap \mathsf{dom}(\sigma).\, \mathsf{abs}(\sigma(l, f)) \subseteq \sigma^\sharp(\text{P1}, f))$$
$$\wedge\, (\forall (l, f) \in (L_2^\eta \times \mathsf{Fld}) \cap \mathsf{dom}(\sigma).\, \mathsf{abs}(\sigma(l, f)) \subseteq \sigma^\sharp(\text{P2}, f))$$
$$\wedge\, (\forall (l, f) \in ((\mathsf{ILoc} - L_1^\eta - L_2^\eta) \times \mathsf{Fld}) \cap \mathsf{dom}(\sigma).$$
$$\mathsf{abs}(\sigma(l, f)) \subseteq \sigma^\sharp(\text{P3}, f))$$

We omit the subscripts L_1^η, L_2^η from $\mathsf{abs}_{L_1^\eta, L_2^\eta}$ to avoid clutter. The first two conjuncts ensure the sound abstraction of objects stored in local and global variables. The remaining ones guarantee that $\sigma^\sharp$ overapproximates the concrete heap σ, according to the partitioning scheme dictated by the parameter configuration η.

We order elements in $\mathcal{D}$ in a standard way, by extending the subset order for $\mathcal{P}(\mathsf{Loc}^\sharp)$ pointwise over the function space first and the product space next.

LEMMA 9. *The abstract domain $\mathcal{D}$ is a complete lattice. Furthermore, for all $\eta \in \mathsf{PConfig}$, their concretisation maps γ_η are monotone, and satisfy the following condition:*

$$\eta \sim \eta' \implies \{\gamma_\eta(d) \mid d \in \mathcal{D}_\eta\} = \{\gamma_{\eta'}(d) \mid d \in \mathcal{D}_{\eta'}\}.$$

5.2.1 NC algorithm

Given a set S_d of states from the dynamic analysis, our NC algorithm for the points-to analysis first computes the following sets H_x and H_y:

$$H_x = \{h \mid \exists (pc, \rho, \pi, \sigma) \in S_d.\, \exists o.\, (\rho \uplus \pi)(x_q) = (h, o)\},$$
$$H_y = \{h \mid \exists (pc, \rho, \pi, \sigma) \in S_d.\, \exists o.\, (\rho \uplus \pi)(y_q) = (h, o)\}.$$

The first set H_x consists of allocation sites of x_q-pointed objects that appear in some states of S_d. Similarly, the second H_y is made from allocation sites of y_q-pointed objects appearing in some $s \in S_d$. Next, our algorithm converts H_x, H_y to a set N of parameter bindings: $N = \{(h, \text{P1}) \mid h \in H_x\} \cup \{(h, \text{P2}) \mid h \in H_y\}$, which is returned as a result of the algorithm.

Our algorithm is a solution to the NC problem for the points-to analysis. This is because it is an instance of the generic solution in Section 4 with the following data:

1. The η-independent set D_g of good elements exists, and it is: $D_g = \{(\pi^\sharp, \sigma^\sharp) \mid \pi^\sharp(x_q) \cap \pi^\sharp(y_q) = \emptyset\}$.
2. For each state s, we let $X_s = \mathcal{P}(\{l \mid \exists f.\, (l, f) \in \mathsf{dom}(\sigma)\} \cup \{\mathsf{nil}\})$, and define $\mathsf{Aux}_s = X_s \times X_s$. Hence, Aux_s consists of pairs (L_1, L_2), where L_i is a set of allocated locations in s or nil.
3. The remaining data are the following functions F_s and G_s for every $s = (pc, \rho, \pi, \sigma) \in \mathcal{S}_{\mathsf{base}}$:

$$F_{(pc,\rho,\pi,\sigma)}(L_1, L_2) = (L_1 \cup \{(\rho \uplus \pi)(x_q)\}, L_2 \cup \{(\rho \uplus \pi)(y_q)\})$$
$$G_{(pc,\rho,\pi,\sigma)}(L_1, L_2) =$$
$$\{(h, \text{P1}) \mid \exists o.\, (h, o) \in L_1\} \cup \{(h, \text{P2}) \mid \exists o.\, (h, o) \in L_2\}$$

The function F_s simply adds the value of x_q to the first set, and that of y_q to the second. Hence, the fixpoint of F_s is just $(\{(\rho \uplus \pi)(x_q)\},\ \{(\rho \uplus \pi)(y_q)\})$, which will be computed by one fixpoint iteration. From this fixpoint, the function G_s gets the bindings of allocation sites to P1 or P2.

LEMMA 10. *The data $D_g, \mathsf{Aux}_s, F_s, G_s$ satisfy the conditions in Section 4. Hence, our parametrised points-to analysis has coupled components. Also, the induced NC algorithm solves the NC problem for the points-to analysis.*

5.2.2 Construction of a parameter configuration

From the result N of the NC algorithm, we construct a parameter configuration $\eta \in \mathsf{PConfig}$ to be used by our static points-to analysis. Our construction follows the method described at the end of Section 3 with only a minor adjustment. As before, if the same allocation site is bound to P1 and P2 at the same time by N, our combined dynamic and static analysis stops and returns "impossible to prove." Otherwise, it chooses mutually distinct $h_1, h_2, h_3 \in \mathsf{AllocSite}$ that do not appear in a given program nor

N, and defines $N' = N \cup \{(h_1, \text{P1}), (h_2, \text{P2}), (h_3, \text{P3})\}$. Choosing such h_i's is possible since AllocSite is an infinite set and the given program uses only finitely many allocation sites in AllocSite. Then, our analysis uses P3 as a default parameter value, and constructs a configuration $\eta_{N'}(h) = (\text{if } (h, v) \in N' \text{ then } v \text{ else P3})$. Using P3 as a default value is our decision choice based on the observation: whatever parameter configuration η is used, the resulting η-component points-to analysis is very cheap, hence it is wise to go for the option that maximises precision, which is precisely to use P3 as a default value. We point out that $\eta_{N'}$ belongs to PConfig since N' includes the bindings $(h_1, \text{P1})$, $(h_2, \text{P2})$ and $(h_3, \text{P3})$.

6. Experimental evaluation

In this section, we evaluate the effectiveness of the two instance analyses of our framework: the thread-escape analysis and the points-to analysis. We implemented these analyses and applied them to the six multi-threaded Java programs described in Table 1, including four from the DaCapo benchmark suite [4].[5] All experiments were done using IBM J9 VM 1.6.0 on a Linux machine with two Intel Xeon 2.9 GHz six-core processors and 32GB RAM (though the experiments were run in a single thread and the JVM was limited to use up to 4GB RAM). We next evaluate the precision of these analyses (Section 6.1), their scalability (Section 6.2), and the quality of the computed abstractions (Section 6.3).

6.1 Precision

In this section, we evaluate the precision of our thread-escape and points-to analyses. Figure 3 shows the precision of our thread-escape analysis. Each query to this analysis is a pair (pc, x) where pc is the program position of a statement that accesses an instance field or an array element of an object denoted by local variable x:

$\texttt{pc} : \texttt{y} = \texttt{x.f};$ $\quad \texttt{pc} : \texttt{y} = \texttt{x[i]};$ $\quad \texttt{pc} : \texttt{x.f} = \texttt{y};$ $\quad \texttt{pc} : \texttt{x[i]} = \texttt{y};$

Such queries may arise from any analysis of multi-threaded programs that desires to reason only about instructions that possibly access thread-shared memory, such as a static race detection tool or a software transactional memory runtime.

The top of each column shows how many queries were considered for each benchmark, that is, queries where program position pc was reached at least once in a concrete trace of the benchmark on a single supplied input. It shows both the absolute number of considered queries and what fraction they constitute of the queries reachable in a static 0-CFA call graph. The latter provides a measure of the coverage achieved by each trace (29–60%). The considered queries are classified into three categories: those disproven by our dynamic analysis of the trace ("Escaping"), those proven by our static analysis using the parameter configuration inferred by the dynamic analysis ("Local"), and those neither disproven nor proven ("Unknown"). On average, 80% of the queries in each benchmark are either disproven (28%) or proven (52%), highlighting the effectiveness of our approach using only a single trace. Also, note that our approach does not preclude the use of multiple traces, which would only further improve both coverage and precision.

Figure 4 shows the precision of our points-to analysis. Each query to this analysis is a tuple (pc_1, x, pc_2, y) where (pc_1, x) and (pc_2, y) are identical to the queries described above for our thread-escape analysis, with the additional constraint that they both

[5] Among all benchmarks in dacapo-2006-10-MR2, dacapo-9.10-beta0, and dacapo-9.12, we excluded single-threaded benchmarks (`bloat`, `chart`, `antlr`, `fop`, etc.), and multi-threaded benchmarks with little concurrency (`batik`, `pmd`, etc.), because one of our instance analyses is thread-escape analysis. We also excluded `luindex` because it is too similar to `lusearch`, which we include (both are built atop Apache Lucene). We tried the remaining four benchmarks in our experiments.

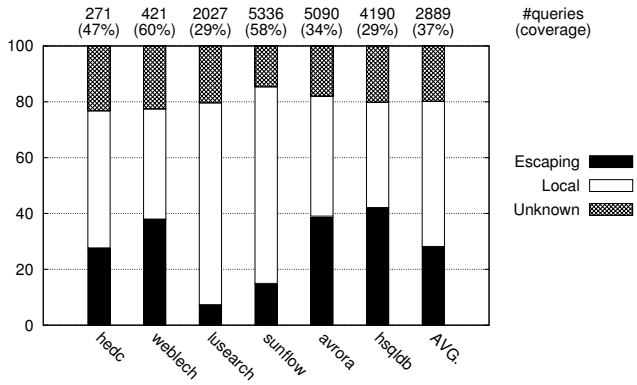


Figure 3. Precision results for our thread-escape analysis.

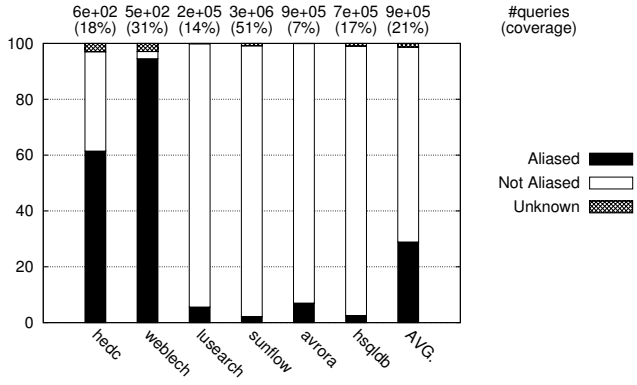


Figure 4. Precision results for our points-to analysis.

access array elements or they both access the same instance field, and at least one of them is a write. Such queries may be posed by, for instance, a static race detection client to determine whether the statements at pc_1 and pc_2 can be involved in a race.

The top of each column shows how many queries were considered for each benchmark, that is, queries where both program positions pc_1 and pc_2 were reached at least once in the single trace. The traces cover 7–51% of all statically reachable queries. The average coverage is lower for queries of this analysis compared to that of our thread-escape analysis (21% vs. 37%). This is because the points-to analysis requires *both* pc_1 and pc_2 to be reached for a query to be considered whereas the thread-escape analysis requires a single program position pc to be reached.

The considered queries are classified into three categories: (1) those disproven by our dynamic analysis of the trace ("Aliased"), namely, those where x and y pointed to objects created at the same allocation site at least once; (2) those proven by our static analysis using the parameter configuration inferred by the dynamic analysis ("Not Aliased"); and (3) those neither disproven nor proven ("Unknown"). Note that category (1) includes not only queries that are false concretely but also queries that might be true concretely but are impossible to prove using an object allocation site abstraction. Almost all queries (99% on average) are either disproven or proven. This result suggests that, in practice, a flow- and context-insensitive points-to analysis based on object allocation site abstraction for Java does not require representing objects allocated at each site using a separate abstract location; merely three abstract locations (albeit specialised to the query) suffice.

	description	# classes		# methods		# bytecodes (KB)		# alloc.
		app	total	app	total	app	total	sites
hedc	web crawler from ETH	44	355	234	2,062	16	161	1,587
weblech	website download/mirror tool (version 0.0.3)	57	579	311	3,295	20	237	2,636
lusearch	text indexing and search tool (dacapo-9.12)	229	648	1,510	3,893	100	273	2,879
sunflow	photo-realistic rendering system (dacapo-9.12)	164	1,018	1,327	6,652	117	480	5,170
avrora	microcontroller simulation/analysis tool (dacapo-9.12)	1,159	1,525	4,245	5,980	223	316	4,860
hsqldb	relational database engine (dacapo-2006-10-MR2)	199	837	2,815	6,752	221	491	4,564

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 static 0-CFA call-graph analysis. The "# bytecodes" column is the number of bytecodes of reachable methods. 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.

	pre-process	dynamic analysis		static analysis
	time	time	# events	time (serial)
hedc	18s	6s	0.6M	38s
weblech	33s	8s	1.5M	74s
lusearch	27s	31s	11M	8m
sunflow	46s	8m	375M	74m
avrora	36s	32s	11M	41m
hsqldb	44s	35s	25M	86m

Table 2. Running time of our thread-escape analysis.

6.2 Scalability

In this section, we evaluate the scalability of our thread-escape analysis. Table 2 provides the running time of the analysis. The "pre-process time" column reports the time to prepare the benchmark for analysis (resolving reflection, computing a call graph, etc.). The "dynamic analysis" column reports the running time of our dynamic analysis, which includes the time to instrument the benchmark and run it on a single supplied input. It also reports the length of the trace that was analyzed. The trace includes a separate event for each execution of each object allocation instruction, each instance field or array element access, and each thread-escaping instruction (i.e., a write to a static field or a call to the `start()` method of class `java.lang.Thread`). We tried multiple different inputs for each benchmark but found only marginal improvements in coverage and precision. This suggests that the truthhood of most queries, and the abstractions for proving them, are not sensitive to program inputs, which in turn plays into our approach's favour.[6]

For each reachable query, the dynamic analysis either disproves the query or provides a parameter configuration that is used by the subsequent static analysis. The "static analysis" column reports the serial running time of all invocations of the static analysis, one per set of queries for which the same parameter configuration is inferred by the dynamic analysis. Note that these invocations do not share anything and could be run in an embarrassingly parallel manner on a multi-core machine or a cluster; hence, we next study the running time of each invocation.

Figure 5 provides the cumulative distribution function (CDF) of the running times of individual invocations of the static analysis for each of our four large benchmarks (all from the DaCapo suite). The blue curve ($-\circ-$) shows the CDF of an optimised version of the static analysis while the red curve ($-\times-$) shows the CDF of the naive version (we explain the difference between the two versions shortly). The CDFs have a separate point for each different running time; the x-intercept of the point denotes that time while the y-intercept of the point denotes the number of invocations that took at most that time. The optimised version of the analysis takes almost constant time across different invocations for each benchmark, suggesting that our approach is effective at tailoring the abstraction to each query (Section 6.3 provides more statistics about the computed abstractions). Specifically, it takes an average of 7 seconds per invocation and a maximum of 20 seconds over all 1,743 invocations for all benchmarks. The naive version, on the other hand, takes an average of 30 seconds per invocation and runs out of memory for the 18 invocations (14 for `hsqldb` and 4 for `sunflow`) that are denoted by the points taking 300 seconds—the timeout we used for those invocations.

The optimised and naive versions of our static thread-escape analysis differ only in how they compute method summaries, which we explain next. Let $(pc^\sharp, \rho^\sharp, \sigma^\sharp)$ denote the incoming abstract state for a method. Recall that $\rho^\sharp$ provides the incoming abstract value of each formal argument of the method, and $\sigma^\sharp$ is the incoming abstract heap. If the incoming abstract value of none of the formal arguments contains abstract location L (i.e., $\forall x.\, \mathrm{L} \notin \rho^\sharp(x)$), then the analysis of that method will *never* read the incoming abstract heap $\sigma^\sharp$. The optimised version of our analysis exploits this observation and analyzes the method in context $(\rho^\sharp, \lambda f.\emptyset)$ (i.e., using an empty incoming abstract heap), whereas the naive version still analyzes it in context $(\rho^\sharp, \sigma^\sharp)$. In practice, we observed that our top-down inter-procedural analysis repeatedly visits methods with the same incoming abstract environment satisfying the above condition, but with different incoming abstract heaps. Thus, the naive version results in significantly lesser reuse of summary edges than the optimised version, evident from the CDFs plotted in Figure 6. These CDFs have a separate point for each different number of summary edges that was computed for any method in the benchmark in any invocation of the analysis; the x-intercept of the point provides that number while the y-intercept of the point provides the fraction of methods for which at most that many summary edges were computed in any invocation. Note that while in both versions, only under a handful of summary edges is computed for the vast majority of methods, there are outliers in the naive version for which vastly more summary edges are computed than in the optimised version, for all four benchmarks. These outliers cause the 18 invocations described above to run out of memory.

It is worth emphasizing that sophisticated optimizations have been proposed to efficiently compute and concisely represent method summaries over abstract heaps (e.g., [20]). While those optimizations are hard to understand and implement, ours is relatively simple yet highly effective in practice. Notice that the only reason why our optimization is enabled is due to the unique capability of our dynamic analysis to map only a few necessary allocation sites to L and the vast majority of allocation sites to E in the parameter configurations with which the static analysis is invoked (see Section 6.3); without this capability, many methods would be analyzed with incoming abstract environments containing L in the abstract

[6] We conjecture that the input insensitivity happens, because thread-escape analysis and pointer analysis concern heap structure and pointer connectivity, which is less sensitive to the specific data values in different program inputs (e.g., different keywords to be searched by `lusearch`, different SQL queries to be answered by `hsqldb`, etc.).

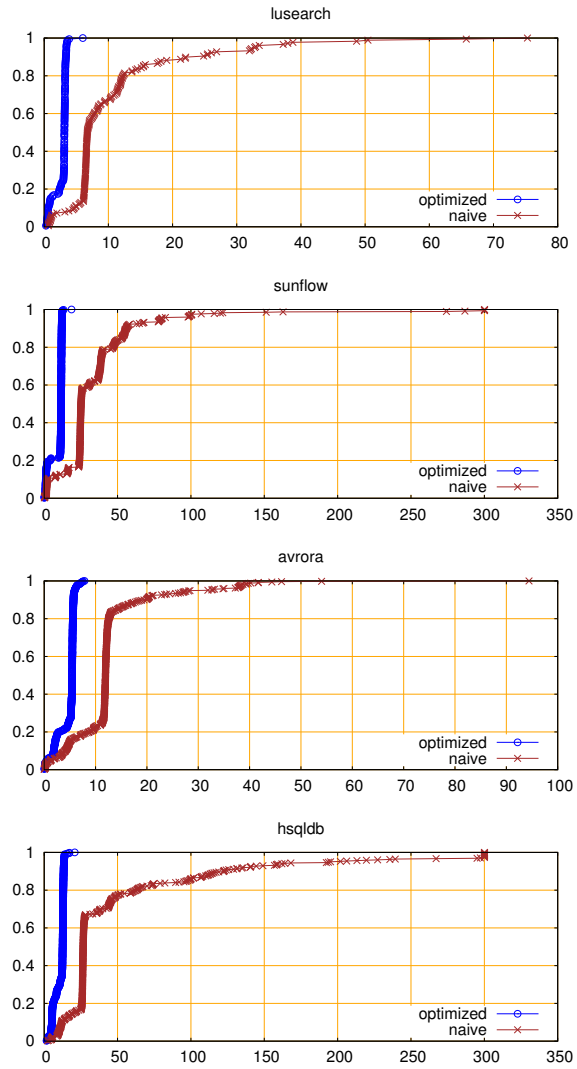


Figure 5. CDF of the running time (secs.) of invocations of the optimised and naive versions of our static thread-escape analysis.

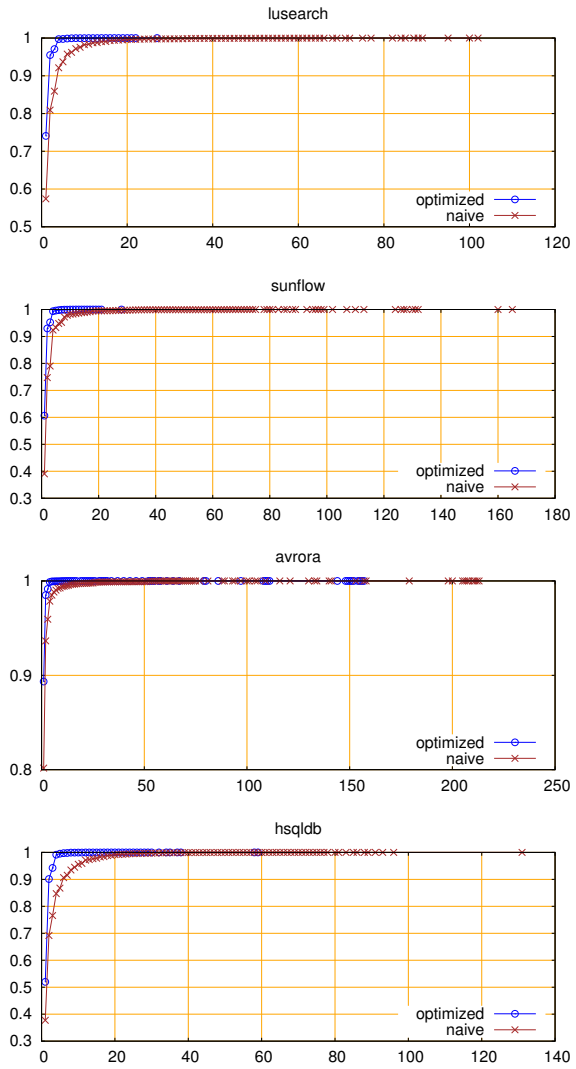


Figure 6. CDF of summary sizes of methods in invocations of the optimised and naive versions of our static thread-escape analysis.

values of formal arguments, which in turn would prevent the optimised version of our analysis from ignoring the incoming abstract heap and drastically hurt summary reuse.

6.3 Abstraction quality

This section evaluates the quality of the abstractions computed by our thread-escape analysis. The key question we want to answer is: how hard is the thread-escape analysis problem to justify using query specialisation and dynamic analysis?

We experimented with various purely static strawman approaches and found that they were not precise or scalable enough. Below, we describe two such approaches.

Strawman 1. The goal of this strawman was to determine how much flow and/or context sensitivity matters for precision. We implemented a flow- and context-insensitive analysis based on object allocation site abstraction, and found that it proved only 13-34% of the queries for our benchmarks, with an average of 27.5%. In contrast, the combined dynamic-static approach presented in this paper, which is flow- and context-sensitive, proves 38-72% of the queries, with an average of 52%. This shows that flow and/or context sensitivity is likely important for precision.

Strawman 2. The goal of this strawman was to determine whether a trivial parameter configuration is precise and scalable enough. We used the static thread-escape analysis presented in this paper, but invoked it with a parameter configuration that simply sets each allocation site in the program to L, instead of using our dynamic analysis to obtain parameter configurations tailored to individual queries. We found the resulting analysis to be highly unscalable—it ran out of memory on all our six benchmarks—but highly precise on much smaller benchmarks (not shown) that it was able to successfully analyze. This shows that: (1) even though the above trivial parameter configuration is not the most precise in principle, it is likely very precise in practice; and (2) the ability of our dynamic analysis to avoid unnecessarily mapping sites to L is critical in practice for scalability.

Lastly, we would like to answer the question: if a few allocation sites must be mapped to L for scalability, how small is that number to justify an approach as sophisticated as backward pointer reachability, and how different are the allocation sites for different queries to justify query specialisation? We next answer these questions.

Figure 7 shows the CDF of the number of allocation sites that were mapped to L for each query by the dynamic analysis. Recall

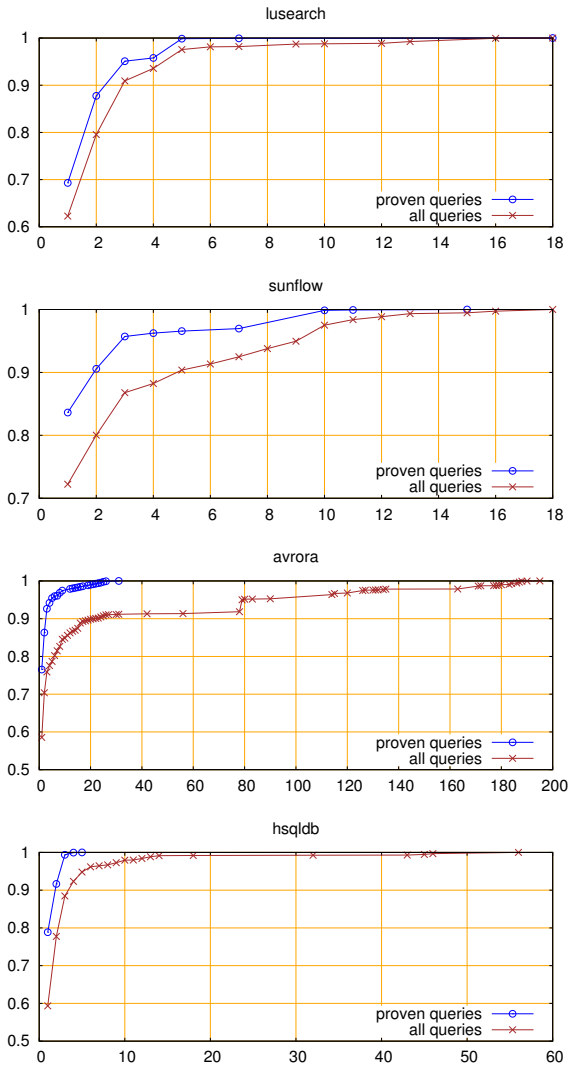


Figure 7. CDFs of the number of allocation sites mapped to L in each parameter configuration, for all queries considered by our static thread-escape analysis and for just those that were proven.

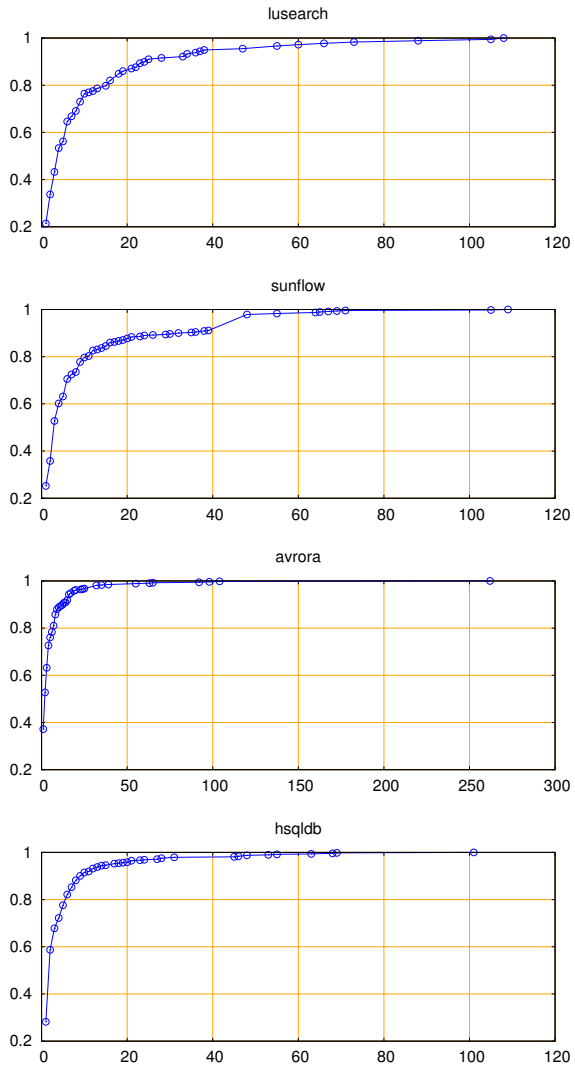


Figure 8. CDF of the number of queries in each invocation of our static thread-escape analysis.

that these are the sites the dynamic analysis has determined must be mapped to L: mapping any of them to E is guaranteed to make the static analysis fail to prove the query. The red curve (-x-) shows the CDF for all queries that were considered by the static analysis whereas the blue curve (-o-) shows the CDF for only queries that it ended up proving. For each point shown, the x-intercept denotes the number of sites that were mapped to L for some query, and the y-intercept denotes the fraction of queries that needed at most those many sites. The relative shapes of the two curves are expected: as the number of sites needed to be mapped to L by a query grows, the chance that our dynamic analysis will miss some needed site due to lack of coverage and thereby cause our static analysis to fail to prove the query increases. The CDFs show that, while just 1-2 sites are needed to prove around 50% of the queries for each benchmark, at least a handful of sites are needed to prove another 40%, and tens of sites are needed to prove the remaining 10%. On average, 4.8 sites are needed for all queries that the static analysis attempts to prove, with the highest being 195 sites for `avrora`.

Figure 8 shows the CDF of the number of queries in each invocation of our static thread-escape analysis for which the same parameter configuration is inferred by the dynamic analysis. For each point shown, the x-intercept denotes the size of at least one such query set, and the y-intercept denotes the fraction of query sets of at most that size. On average, an invocation of the static analysis considers 14 queries, with some outliers considering over hundred queries each. This shows that the abstraction needed for proving each query is neither too unique nor too generic.

7. Related work

Our approach is related to techniques for client-driven abstraction specialisation and to techniques for combining dynamic and static analyses. We next survey each of these kinds of techniques.

Client-driven abstraction specialisation. In the client-driven specialisation problem, a program and a query (assertion) are given and the goal is to find either a counterexample program trace showing a violation of the query, or an abstraction which is cheap yet precise enough for the analysis to efficiently prove the query on the pro-

gram. There are two natural solutions to this problem: *abstraction refinement*, which starts with a coarse abstraction and refines it, and *abstraction coarsening*, which starts with a precise abstraction and coarsens it. The CEGAR approach (e.g., [1, 3, 5, 13, 19]) falls in the first category, and refines the abstraction guided by false counterexample traces. Besides the CEGAR approach, there are other approaches to refine abstractions that are based on a dependence analysis that relates unproven queries to sources of imprecision such as flow-, context-, or object insensitivity in the abstraction [12, 14, 18]. In more recent work, Liang et al. [15] present randomised algorithms to find a *minimal abstraction* that are based on both abstraction refinement and coarsening.

A fundamental difference between our approach and all the above techniques is that we utilise the concrete trace in order to avoid performing the static analysis at all. This can be handy in cases where the static analysis is exploring too many paths and considering too many configurations. Interestingly, our method could be combined with abstraction refinement, e.g. by computing an initial abstraction which respects the necessary condition and then applying refinement if necessary.

Combining static and dynamic analyses. Recent work combines static and dynamic analyses in interesting ways. The Yogi project [2, 9, 10, 17] exploits a dynamic analysis to improve the performance of the abstraction-refinement step of a static analysis. A form of concolic testing is used to reduce the number of theorem prover calls and case splits due to pointer aliasing, which a static analysis considers during refinement.

Gupta et al. [11] uses a dynamic analysis to simplify non-linear constraints generated during inference of program invariants. Yorsh et al. [22] uses a static analysis to construct a concrete trace that is then used by an automated theorem prover to check if it covers all executions. It proves safety properties if the check passes on the concrete trace and produces a counterexample that can be used to generate new traces if the check fails.

One of the interesting questions for a program analysis is how to infer the right invariants to prove a query. One potential idea which is similar to ours is to start from a given concrete trace and then generalise it. McMillan [16] uses interpolants to perform generalisation. This approach is interesting but it requires computing interpolants which is not yet feasible for many programming language features. Also, like CEGAR this approach fails to generalise from multiple paths. In contrast we simplify the problem by assuming that the abstract domains already provide generalisation, and use the concrete trace to trim inadequate abstractions.

Techniques combining random test generation and concrete execution with symbolic execution and model generation have been explored [7, 8, 21]. They use symbolic methods to direct tests towards unexplored paths to find errors faster. But they do not use abstraction and in general cannot find proofs in the presence of loops.

To our knowledge, our technique is unique in that it uses a dynamic analysis for computing necessary conditions on abstractions, and also for directly providing a static analysis with an abstraction that is specialised to a given query and which in most cases succeeds in proving the query.

8. Conclusion

Efficiently finding good abstractions is a long-standing problem in static analysis. Parametrised static analyses offer the flexibility to specialise the abstraction to a given query but also pose a hard search for a suitable parameter configuration. We have presented a novel solution to this problem: using a dynamic analysis to compute a necessary condition on the parameter configurations for proving a given query. We have given constructive algorithms for two instance analyses: thread-escape analysis and points-to analysis. We have proven that these algorithms indeed compute necessary conditions and shown that, in practice, these algorithms are efficient and the resulting static analyses are both precise and scalable.

Acknowledgments We thank Percy Liang for technical discussions on thread-escape analysis and the necessary-condition problem, and Peter O'Hearn and the anonymous referees for helpful comments on the paper. Yang acknowledges support from EPSRC.

References

[1] T. Ball and S. Rajamani. The slam project: Debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[2] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA*, pages 3–14, 2008.

[3] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, pages 300–309, 2007.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.

[5] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *JACM*, 50(5), 2003.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL*, pages 238–252, 1977.

[7] C. Csallner and Y. Smaragdakis. Check 'n' Crash: combining static checking and testing. In *ICSE*, pages 422–431, 2005.

[8] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[9] P. Godefroid, A. Nori, S. Rajamani, and S. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *POPL*, pages 43–56, 2010.

[10] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In *SIGSOFT FSE*, pages 117–127, 2006.

[11] A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. In *TACAS*, pages 262–276, 2009.

[12] S. Guyer and C. Lin. Client-driven pointer analysis. In *SAS*, pages 214–236, 2003.

[13] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.

[14] P. Liang and M. Naik. Scaling abstraction refinement via pruning. In *PLDI*, pages 590–601, 2011.

[15] P. Liang, O. Tripp, and M. Naik. Learning minimal abstractions. In *POPL*, pages 31–42, 2011.

[16] K. McMillan. Relevance heuristics for program analysis. In *POPL*, pages 145–146, 2008.

[17] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The Yogi project: Software property checking via static analysis and testing. In *TACAS*, pages 178–181, 2009.

[18] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA*, pages 324–340, 1994.

[19] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th International Symposium on Programming*, pages 337–350, 1982.

[20] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, pages 296–309, 2005.

[21] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *FSE*, pages 263–272, 2005.

[22] G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: Better together! In *ISSTA*, pages 145–156, 2006.