# Learning Minimal Abstractions

Percy Liang
UC Berkeley
pliang@cs.berkeley.edu

Omer Tripp
Tel-Aviv University
omertrip@post.tau.ac.il

Mayur Naik
Intel Labs Berkeley
mayur.naik@intel.com

## Abstract

Static analyses are generally parametrized by an abstraction which is chosen from a family of abstractions. We are interested in flexible families of abstractions with many parameters, as these families can allow one to increase precision in ways tailored to the client without sacrificing scalability. For example, we consider $k$-limited points-to analyses where each call site and allocation site in a program can have a different $k$ value. We then ask a natural question in this paper: What is the minimal (coarsest) abstraction in a given family which is able to prove a set of client queries? In addressing this question, we make the following two contributions: (i) we introduce two machine learning algorithms for efficiently finding a minimal abstraction; and (ii) for a static race detector backed by a $k$-limited points-to analysis, we show empirically that minimal abstractions are actually quite coarse: it suffices to provide context/object sensitivity to a very small fraction (0.4–2.3%) of the sites to yield equally precise results as providing context/object sensitivity uniformly to all sites.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification

***General Terms*** Measurement, Experimentation, Verification

***Keywords*** heap abstractions, static analysis, concurrency, machine learning, randomization

## 1. Introduction

Static analyses typically have parameters that control the tradeoff between precision and scalability. For example, in a $k$-CFA-based or $k$-object-sensitivity-based points-to analysis [10–13, 20, 26], the parameter is the $k$ value, which determines the amount of context sensitivity and object sensitivity. Increasing $k$ yields more precise points-to information, but the complexity of the analysis also grows exponentially with $k$. Shape analysis [19] and model checkers based on predicate abstraction [3, 5] are parametrized by some number of predicates; these analyses also exhibit this tradeoff.

In many analyses, these tradeoffs are controlled by a small number of parameters, for instance, a single $k$ value. Past studies (e.g., client-driven [7] and demand-driven [9] approaches) have shown that it is often not necessary to provide context sensitivity to each call site or object sensitivity to each allocation site. This motivates working with a larger family of abstractions parametrized by a separate $k$ value for each site, akin to the parametric framework of Milanova et al. [12, 13]. More generally, we represent an abstraction as a binary vector (e.g., component $j$ of the vector specifies whether site $j$ should be treated context-sensitively). But how much context/object sensitivity is absolutely needed, and where is it needed?

In this paper, we formulate and tackle the following problem: Given a family of abstractions, find a minimal (coarsest) abstraction sufficient to prove all the queries provable by the finest abstraction in the family. Studying this problem is important for two reasons: (i) a minimal abstraction provides insight into which aspects of a program need to be modeled precisely for a given client; and (ii) reducing the complexity of the abstraction along some components could enable us to increase the complexity of the abstraction along other components more than before. For example, keeping the $k$ values of most sites at zero enables us to use higher $k$ values for a select subset of sites.

To find these minimal abstractions, we introduce two machine learning algorithms. Both treat the static analysis as a black box which takes an abstraction (and a set of client queries) as input and produces the set of proven queries as output. The first algorithm, StatRefine, starts with the coarsest abstraction, runs the static analysis on randomly chosen abstractions, and from these training examples detects statistical correlations between components of the abstraction and whether a query is proven; components highly correlated with proven queries are added (Section 4.1). The second algorithm, ActiveCoarsen, starts with the finest abstraction and samples coarser abstractions at random, incrementally reducing the abstraction to a minimal one (Section 4.2).


*POPL'11,* January 26–28, 2011, Austin, Texas, USA.

We also provide a theoretical analysis of our algorithms. Let $p$ be the number of components of the abstraction family and $s$ be the number of components in the largest minimal abstraction. We show that although the number of abstractions considered is exponential in $p$, we only need $O(s \log p)$ calls to the static analysis to find a minimal abstraction. The significance of this result is that when a very small abstraction suffices to prove the queries ($s \ll p$), our algorithms are much more efficient than a naïve approach, which would require $O(p)$ calls. Empirically, we found that minimal abstractions are indeed very small. This is an instance of *sparsity*, an important property in machine learning and statistics [4, 24], that very few components of an unknown vector are non-zero.

Our approach represents a significant departure from traditional program analysis, where iterative refinement techniques are the norm [2, 7, 22]. In particular, our methods exploit randomness to generate information in the form of statistical correlations. Also note that iterative refinement is in general not guaranteed to find a minimal abstraction, whereas our techniques do have this guarantee. We present one iterative refinement technique (DATALOGREFINE in Section 3.1) and show that it refines 18–85% of the components whereas the minimal abstractions found by our approach only refine 0.4–2.3% (Section 6).

All our empirical results are for a static race detection client [14] backed by a $k$-limited points-to analysis. Points-to information is used extensively by the race detector to determine which statements may be reachable, which statements may access the same memory locations, which statements may access thread-shared memory locations, and which statements may happen in parallel. Our points-to analysis is both context-sensitive and object-sensitive (see Section 5 for details).

## 2. Problem Formulation

Let $(\mathcal{A}, \preceq)$ be a poset corresponding to a family of abstractions, and let $\mathbb{Q}$ be a set of queries that we would like to prove. We assume we have access to a static analysis, $\mathbf{F} : \mathcal{A} \mapsto \{0,1\}^{\mathbb{Q}}$, which maps an abstraction $\mathbf{a} \in \mathcal{A}$ to a binary vector $\mathbf{F}(\mathbf{a})$ where component $q$ is 0 if query $q$ is proven and 1 if it is not.

We assume that $\mathbf{F}$ is monotone with respect to the abstraction family $(\mathcal{A}, \preceq)$, that is, if $\mathbf{a} \preceq \mathbf{a}'$, then $\mathbf{F}(\mathbf{a}) \succeq \mathbf{F}(\mathbf{a}')$. In other words, refining an abstraction (increasing $\mathbf{a}$) can only enable us to prove more queries (decrease $\mathbf{F}(\mathbf{a})$).

The focus of this paper is on the following problem:

**Definition 1** (Minimal Abstraction Problem)**.** *Given a family of abstractions $\mathcal{A}$ with a unique top element $\mathbf{1}$ (the most precise), output an abstraction $\mathbf{a} \preceq \mathbf{1}$ such that*

1. *$\mathbf{a}$ is correct—that is, $\mathbf{F}(\mathbf{a}) = \mathbf{F}(\mathbf{1})$; and*
2. *$\mathbf{a}$ is minimal—that is, such that $\{\mathbf{a}' \preceq \mathbf{a} : \mathbf{F}(\mathbf{a}') = \mathbf{F}(\mathbf{a})\} = \{\mathbf{a}\}$.*

In general, there could be multiple minimal abstractions, but we are content with choosing any one of them. Furthermore, we would like to find a minimal abstraction efficiently, i.e., minimizing the number of calls to $\mathbf{F}$.

### 2.1 Binary Abstractions

We now specialize to abstractions representable by binary vectors, that is, $\mathcal{A} = \{0,1\}^{\mathbb{J}}$ for some index set of *components* $\mathbb{J}$, where the partial order is component-wise inequality ($\mathbf{a} \preceq \mathbf{a}'$ iff $\mathbf{a}_j \leq \mathbf{a}'_j$ for all $j \in \mathbb{J}$). The idea is that for an abstraction $\mathbf{a} \in \mathcal{A}$, $\mathbf{a}_j$ denotes whether component $j \in \mathbb{J}$ has been refined or not. It will also be convenient to treat $\mathbf{a} \in \mathcal{A}$ directly as a set of components (namely, $\{j \in \mathbb{J} : \mathbf{a}_j = 1\}$), so that we can use set notation (e.g., $\mathbf{a} \cup \{j\}$). We use $\emptyset$ and $\mathbb{J}$ to denote the coarsest and finest abstractions in the family, and we denote the *size* of abstraction $\mathbf{a}$ by $|\mathbf{a}|$.

Many commonly-used abstraction families are binary. In predicate abstraction [3, 5], $\mathbb{J}$ is the set of candidate abstraction predicates, and an abstraction $\mathbf{a}$ would specify a subset of these predicates to include in the analysis. Shape analysis [19] uses predicates on the heap graph, e.g., reachability from a variable. Similar to predicate abstraction, $\mathbf{a}_j = 1$ if predicate $j$ is to be treated as an abstraction predicate.

### 2.2 $k$-limited Abstractions

In this paper, we focus on $k$-limited abstractions. Let $\mathbb{H}$ be the set of allocation sites and $\mathbb{I}$ be the set of call sites in a program. We use $\mathbb{S} = \mathbb{H} \cup \mathbb{I}$ to denote the set of *sites* of both types. Consider the family of abstractions defined by setting a non-negative integer $k$ for each site $s \in \mathbb{S}$. In the case of $k$-CFA, this integer specifies that the $k$ most recent elements of the call stack should be used to distinguish method calls and objects allocated at various sites. We denote this abstraction family as $\mathcal{A}^{\text{K}} = \{0, 1, \ldots, k_{\text{max}}\}^{\mathbb{S}}$, where $k_{\text{max}}$ is the largest allowed $k$ value.

At first glance, it may not be evident that the $k$-limited abstraction family $\mathcal{A}^{\text{K}}$ is a binary abstraction family. However, we can represent $\mathcal{A}^{\text{K}}$ using binary vectors as follows: Let $\mathbb{J} = \mathbb{S} \times \{1, \ldots k_{\text{max}}\}$ be the set of components. We map each element $\mathbf{a} \in \mathcal{A} = \{0,1\}^{\mathbb{J}}$ in the binary abstraction family to a unique element $\mathbf{a}^{\text{K}} \in \mathcal{A}^{\text{K}}$ in the original $k$-limited abstraction family by $\mathbf{a}^{\text{K}}_s = \sum_{k=1}^{k_{\text{max}}} \mathbf{a}_{(s,k)}$. Essentially $\mathbf{a} \in \mathcal{A}$ is a unary encoding of the $k$ values of $\mathbf{a}^{\text{K}} \in \mathcal{A}^{\text{K}}$. Note that multiple binary vectors $\mathbf{a}$ map onto the same $\mathbf{a}^{\text{K}}$ (i.e., $(1,1,0)$ and $(1,0,1)$ both represent $k = 2$), but this is not important. It is crucial, however, that the mapping from $\mathcal{A}$ to $\mathcal{A}^{\text{K}}$ respects the partial ordering within each family.

## 3. Deterministic Approaches

In this section, we discuss two deterministic approaches for finding small abstractions. Conceptually, there are two ways to proceed: start with the coarsest abstraction and refine, or start with the finest abstraction and coarsen. We present two algorithms: DATALOGREFINE (Section 3.1) and

SCANCOARSEN (Section 3.2), which operate in these two directions.

### 3.1 Refinement via Datalog Analysis

We first present the DATALOGREFINE algorithm, which assumes that static analysis $\mathbf{F}$ is expressed as a Datalog program $P$. The basic idea behind the algorithm is as follows: run the static analysis with the coarsest abstraction $\emptyset$ and look at queries which were not proven. DATALOGREFINE then inspects $P$ to find *all* the components of the abstraction which could affect the unproven queries and refines exactly these components.

A Datalog program $P$ consists of (i) a set of *relations* $\mathcal{R}$ (e.g., $\texttt{ptsV} \in \mathcal{R}$, where $\texttt{ptsV}(v, o)$ denotes whether variable $v$ may point to abstract object $o$); (ii) a set of *input tuples* $\mathcal{I}$ (for example, $\texttt{ptsV}(\texttt{v5}, \texttt{o7}) \in \mathcal{I}$); and (iii) a set of *rules* of the following form:

$$R_0(w_0) \Leftarrow R_1(w_1), \ldots, R_m(w_m), \tag{1}$$

where for each $i = 0, \ldots, m$, we have a relation $R_i \in \mathcal{R}$ and a tuple of variables $w_i$ of the appropriate arity (e.g., when $R_0 = \texttt{ptsV}$, $w_0 = (v, o)$).

Given a Datalog program, we derive new tuples from the input tuples $\mathcal{I}$ by using the rules. Formally, let a *derivation* be a sequence $t_1, \ldots, t_n$ of tuples satisfying the following two conditions: (i) for each $i = 1, \ldots, n$, either $t_i$ is an input tuple ($t_i \in \mathcal{I}$) or there exist indices $j_1, \ldots, j_m$ all smaller than $i$ such that $t_i \Leftarrow t_{j_1}, \ldots, t_{j_m}$ is an instantiation of a rule; and (ii) for each $j = 1, \ldots, n-1$, tuple $t_j$ appears on the right-hand side of an instantiation of a rule with some $t_i$ ($i > j$) on the left-hand side.

In this formalism, each query $q \in \mathbb{Q}$ is a tuple. In race detection, the set of queries is

$$\mathbb{Q} = \{\texttt{race}(p_1, p_2) : p_1, p_2 \in \mathbb{P}\}, \tag{2}$$

where $\mathbb{P}$ is the set of program points. For each query $q$, we define $\mathbf{F}(\mathbf{a})_q = 1$ if and only if there exists a derivation of $q$. In other words, $\mathbf{F}(\mathbf{a})_q = 0$ if and only if $q$ is proven.

The abstraction $\mathbf{a}$ determines the input tuples $\mathcal{I}$. More specifically, let $\mathbf{A}(t, j)$ denote whether the value $\mathbf{a}_j$ of component $j$ affects the input tuple $t \in \mathcal{I}$. For example, $\mathbf{A}(\texttt{ptsV}(v, o), j) = 1$ for any $j = (s, k)$ and abstract object $o$ where $o$ can represent a concrete object allocated at allocation site $s \in \mathbb{H}$. Given $\mathbf{A}$, let $J^{\mathbf{A}}$ denote the components which are involved in a derivation of some (unproven) query:

$$J^{\mathbf{A}} \triangleq \{j \in \mathbb{J} : \exists t \in \mathcal{I}, q \in \mathbb{Q}, \mathbf{A}(t, j) = 1 \wedge t \rightsquigarrow q\}, \tag{3}$$

where $t \rightsquigarrow q$ denotes that there exists some derivation $t_1, \ldots, t_n$ where $t_i = t$ for some $i$ and $t_n = q$. The key point is that a component not in $J^{\mathbf{A}}$ cannot eliminate existing derivations of $q$, which would be necessary to prove $q$. Such a component is therefore irrelevant and not refined.

***Computation via a Datalog Program Transformation*** We now define the DATALOGREFINE algorithm, which computes and returns $J^{\mathbf{A}}$ as the abstraction. DATALOGREFINE works by taking a Datalog program $P$ as input and transforming it into another Datalog program $P'$ whose output contains $J^{\mathbf{A}}$. Having this general transformation allows us to leverage existing Datalog solvers [25] to efficiently compute the set of relevant components $J^{\mathbf{A}}$ to refine.

The new program $P'$ contains all the rules of $P$ plus additional ones. For each Datalog rule of $P$ taking the form given in (1), $P'$ will contain $m$ rules, one for each $i = 1, \ldots, m$:

$$R'_i(t_i) \Leftarrow R'_0(t_0), R_1(t_1), \ldots, R_m(t_m). \tag{4}$$

$R'_i(t_i)$ is true iff $R_i(t_i) \rightsquigarrow q$ for any query $q \in \mathbb{Q}$. We also add the following rule for each $R \in \mathcal{R}$, which aggregates the relevant components:

$$J^{\mathbf{A}}(j) \Leftarrow R'(t), \mathbf{A}(t, j). \tag{5}$$

It can be verified that $J^{\mathbf{A}}(j)$ is true exactly when $j \in J^{\mathbf{A}}$, where $J^{\mathbf{A}}$ is defined in (3).

Note that DATALOGREFINE is correct in that it outputs an abstraction $\mathbf{a}$ which is guaranteed to prove all the queries that $\mathbf{1}$ can, but $\mathbf{a}$ will most likely not be minimal.

***$k$-limited Abstractions*** We now describe how we use DATALOGREFINE for $k$-limited abstractions. Recall that in our binary representation of $k$-limited abstractions (Section 2.2), the components are $(s, k)$, where $s \in \mathbb{S}$ is a site and $0 \leq k \leq k_{\text{max}}$. We start with the abstraction $\mathbf{a}_0 = \mathbf{0}$, corresponding to 0-CFA. We then iterate $k = 1, \ldots, k_{\text{max}}$, where at each iteration, we use $\mathbf{a}_{k-1}$ to construct the input tuples and call DATALOGREFINE to produce a set $J^{\mathbf{A}}$. We refine the sites specified by $J^{\mathbf{A}}$, setting $\mathbf{a}_k = \mathbf{a}_{k-1} \cup \{(s, k') : (s, k) \in J^{\mathbf{A}}, k = k'\}$. In this way, in each iteration we increase the $k$ value of each site by at most one. Because we always refine all relevant components, after $k$ iterations, $\mathbf{a}_k$ is guaranteed to prove the same subset of queries as $k$-CFA. The approach is the same for $k$-object-sensitivity.

### 3.2 Coarsening via Scanning

In the previous section, we started with the coarsest abstraction and refined it. We now introduce a simple algorithm, SCANCOARSEN, which does the opposite: it takes the most refined abstraction $\mathbf{a}$ and coarsens it, preserving correctness (Definition 1) along the way. To simplify presentation, suppose we have one query. We will revisit the issue of multiple queries in Section 4.4.

The idea behind the algorithm is quite simple: for each component, try removing it from the abstraction; if the resulting abstraction no longer proves the query, add the component back. The pseudocode of the algorithm is given in Figure 1. The algorithm maintains the invariant that $\mathbf{a}^{\text{L}}$ is a

Coarsening via Scanning

ScanCoarsen($\mathbf{a}^{\text{L}}, \mathbf{a}^{\text{U}}$):
  if $\mathbf{a}^{\text{L}} = \mathbf{a}^{\text{U}}$: return $\mathbf{a}^{\text{U}}$
  choose any component $j \in \mathbf{a}^{\text{U}} \backslash \mathbf{a}^{\text{L}}$
  if $\mathbf{F}(\mathbf{a}^{\text{U}} \backslash \{j\}) = 0$:  [try coarsening $j$]
    return ScanCoarsen($\mathbf{a}^{\text{L}}, \mathbf{a}^{\text{U}} \backslash \{j\}$)  [don't need $j$]
  else:
    return ScanCoarsen($\mathbf{a}^{\text{L}} \cup \{j\}, \mathbf{a}^{\text{U}}$)  [need $j$]

Figure 1: Algorithm that finds a minimal abstraction.

subset of some minimal abstraction and $\mathbf{a}^{\text{U}}$ is a superset sufficient to prove the query. The algorithm requires $|\mathbb{J}|$ calls to $\mathbf{F}$, and therefore is only practical when the number of components under consideration is small.

**Theorem 1** (Properties of ScanCoarsen)**.** *The algorithm* ScanCoarsen($\emptyset, \mathbb{J}$) *returns a minimal abstraction* $\mathbf{a}$ *with* $O(|\mathbb{J}|)$ *calls to* $\mathbf{F}$.

*Proof.* Let $\mathbf{a}$ be the returned abstraction. It suffices to show that $\mathbf{F}(\mathbf{a} \backslash \{j\}) = 1$ for all $j \in \mathbf{a}$ (that is, we fail to prove the query with removal of any $j$). Take any $j \in \mathbf{a}$. Since $j$ was kept, $\mathbf{F}(\mathbf{a}^{\text{U}} \backslash \{j\}) = 1$, where $\mathbf{a}^{\text{U}}$ corresponds to the value when $j$ was considered. However, we also have $\mathbf{a} \subset \mathbf{a}^{\text{U}}$, so $\mathbf{F}(\mathbf{a} \backslash \{j\}) = 1$ by monotonicity of $\mathbf{F}$. □

## 4. Machine Learning Approaches

We now present two machine learning algorithms for finding minimal abstractions, which is the main theoretical contribution of this paper. The two algorithms are StatRefine, which refines an abstraction by iteratively adding components (Section 4.1) and ActiveCoarsen, which coarsens an abstraction by removing components (Section 4.2).

At a high level, these two algorithms parallel their deterministic counterparts, DatalogRefine and ScanCoarsen, presented in the previous section. However, there are two important distinctions worth noting: (i) the machine learning algorithms find a minimal abstraction much more effectively by exploiting *sparsity*, the property that a minimal abstraction contains a small fraction of the full set of components; and (ii) randomization is used to exploit this sparsity.

For clarity of presentation, we again focus on the case where we have a single query; Section 4.4 addresses the multiple-query setting.

### 4.1 Refinement via Statistical Learning

We call a component $j \in \mathbb{J}$ *dependent* if $j$ appears in any minimal abstraction. Let $D \subset \mathbb{J}$ be the set of dependent components and let $d = |D|$. Note that $D$ is the union of all minimal abstractions. Define $s$ to be the size of the largest minimal abstraction, observing that $s \leq d$. StatRefine identifies dependent components by sampling $n$ independent random abstractions and running the static analysis $\mathbf{F}$ on them. The component $j$ associated with the most number of proven queries is then added to the abstraction, and we iterate. The pseudocode of the algorithm is given in Figure 2.

Refinement via Statistical Learning

Parameters:
  $\alpha$: refinement probability
  $s$: size of largest minimal abstraction
  $n$: number of training examples per iteration

Sample($\alpha, \mathbf{a}^{\text{L}}, \mathbf{a}^{\text{U}}$):
  $\mathbf{a} \leftarrow \mathbf{a}^{\text{L}}$
  for each component $j \in \mathbf{a}^{\text{U}} \backslash \mathbf{a}^{\text{L}}$:
    $\mathbf{a}_j \leftarrow 1$ with probability $\alpha$
  return $\mathbf{a}$

StatRefine($\mathbf{a}^{\text{L}}$):
  if $\mathbf{F}(\mathbf{a}^{\text{L}}) = 0$ or $|\mathbf{a}^{\text{L}}| = s$: return $\mathbf{a}^{\text{L}}$
  for $i = 1, \ldots, n$:  [create training examples]
    $\mathbf{a}^{(i)} \leftarrow$ Sample($\alpha, \mathbf{a}^{\text{L}}, \mathbb{J}$)
  for each $j \notin \mathbf{a}^{\text{L}}$:  [compute a score for each component]
    $n_j \leftarrow |\{i : \mathbf{a}^{(i)}_j = 1, \mathbf{F}(\mathbf{a}^{(i)}) = 0\}|$
  $j^* \leftarrow \text{argmax}_{j \notin \mathbf{a}^{\text{L}}} n_j$  [choose best component]
  return StatRefine($\mathbf{a}^{\text{L}} \cup \{j^*\}$)

Figure 2: Algorithm for finding a minimal abstraction by iteratively adding dependent components determined via statistical learning.

While DatalogRefine inspects the Datalog program backing $\mathbf{F}$ to compute the set of *relevant* components, StatRefine relies instead on correlations with the output of $\mathbf{F}$ to find *dependent* components.[1] As Theorem 2 will show, with high probability, a dependent component can be found with $n$ calls to $\mathbf{F}$, where $n$ is only logarithmic in the total number of components $|\mathbb{J}|$.

We must also ensure that $n$ depends only polynomially on $s$ and $d$. The main technical challenge is to set the refinement probability $\alpha$ properly to achieve this. To appreciate this problem, suppose that $s = d$, so that $\mathbf{F}(\mathbf{a})$ consists of a simple conjunction ($\mathbf{F}(\mathbf{a}) = 0$ iff $\mathbf{a}_j = 1$ for each $j$ in the minimal abstraction). If we set $\alpha$ to a constant, then it would take an exponential number of examples ($(\frac{1}{\alpha})^s$ in expectation) to even see an example where $\mathbf{F}(\mathbf{a}) = 0$. Fortunately, the following theorem shows that if $\alpha$ is set properly, then we obtain the desired polynomial dependence (see Appendix A for the proof):

**Theorem 2** (Properties of StatRefine)**.** *Let* $d$ *be the number of dependent components in* $\mathbb{J}$ *and* $s$ *be the size of the largest minimal abstraction. Suppose we set the refinement probability* $\alpha = (\frac{d}{d+1})^d$ *and obtain* $n = \Theta(d^2(\log|\mathbb{J}| +$

[1] Note that dependent components are a subset of relevant components.

Coarsening via Active Learning

Parameters:
  $\alpha$: refinement probability
  $s$: size of largest minimal abstraction

ACTIVECOARSEN($\mathbf{a}^{\mathrm{U}}$):
  if $|\mathbf{a}^{\mathrm{U}}| \leq s+1$: return SCANCOARSEN($\emptyset, \mathbf{a}^{\mathrm{U}}$)
  $\mathbf{a} \leftarrow$ SAMPLE($\alpha, \emptyset, \mathbf{a}^{\mathrm{U}}$)
  if $\mathbf{F}(\mathbf{a}) = 0$:  [run static analysis]
    return ACTIVECOARSEN($\mathbf{a}$)  [reduced]
  else:
    return ACTIVECOARSEN($\mathbf{a}^{\mathrm{U}}$)  [try again]

Figure 3: ACTIVECOARSEN returns a minimal abstraction $\mathbf{a}$ by iteratively removing a random $\alpha$-fraction of the components from an upper bound. SAMPLE is defined in Figure 2.

$\log(s/\delta)))$ *training examples from* $\mathbf{F}$ *each iteration. Then with probability* $1-\delta$, STATREFINE($\emptyset$) *outputs a minimal abstraction with* $O(sd^2(\log|\mathbb{J}| + \log(s/\delta)))$ *total calls to* $\mathbf{F}$.

### 4.2 Coarsening via Active Learning

We now present our second machine learning algorithm, ACTIVECOARSEN. Like SCANCOARSEN, it starts from the finest abstraction $\mathbb{J}$ and tries to remove components from $\mathbb{J}$. But instead of doing this one at a time, ACTIVECOARSEN tries to remove a *random constant fraction* of the components at once. As we shall see, this allows us to hone in on a minimal abstraction much more quickly.

The pseudocode of the algorithm is given in Figure 3. It maintains an upper bound $\mathbf{a}^{\mathrm{U}}$ which is guaranteed to prove the query. It repeatedly tries random abstraction $\mathbf{a} \preceq \mathbf{a}^{\mathrm{U}}$. until $\mathbf{a}$ can prove the query ($\mathbf{F}(\mathbf{a}) = 0$). Then we set $\mathbf{a}^{\mathrm{U}}$ to $\mathbf{a}$ and repeat.

Recall that SCANCOARSEN, which removes one component at a time, requires an exorbitant $O(|\mathbb{J}|)$ calls to the static analysis. The key idea behind ACTIVECOARSEN is to remove a constant fraction of components each iteration. Then we would hope to need only $O(\log_{1/\alpha}|\mathbb{J}|)$ iterations.

However, the only wrinkle is that it might take a lot of trials to sample an $\mathbf{a}$ that proves the query ($\mathbf{F}(\mathbf{a}) = 0$). To appreciate the severity of this problem, suppose $\mathbf{F}(\mathbf{a}) = \neg(\mathbf{a}^* \preceq \mathbf{a})$ for some unknown set $\mathbf{a}^*$ with $|\mathbf{a}^*| = s$; that is, we prove the query if all the components in $\mathbf{a}^*$ are refined by $\mathbf{a}$. Then there is only a $\alpha^s$ probability of sampling a random abstraction $\mathbf{a}$ that proves the query. The expected number of trials until we prove the query is thus $(\frac{1}{\alpha})^s$, which has an unfortunate exponential dependence on $s$. On the other hand, when we succeed, we reduce the number of components by a factor of $\alpha$. There is therefore a tradeoff here: setting $\alpha$ too small results in too many trials per iteration, but setting $\alpha$ too large results in too many iterations. Fortunately, the following theorem shows that we can balance the two to yield an efficient algorithm (see Appendix A for the proof):

**Theorem 3** (Properties of ACTIVECOARSEN)**.** *Let* $s$ *be the size of the largest minimal abstraction. If we set the refinement probability* $\alpha = e^{-1/s}$*, the expected number of calls to the analysis* $\mathbf{F}$ *made by* ACTIVECOARSEN($\mathbb{J}$) *is* $O(s\log|\mathbb{J}|)$.

### 4.3 Adapting the Refinement Probability

Until now, we have assumed that the size of the largest minimal abstraction $s$ is known, and indeed Theorems 2 and 3 depend crucially on setting $\alpha$ properly in terms of $s$. In practice, $s$ is unknown, so we seek a mechanism for setting $\alpha$ without this knowledge.

The intuition is that setting $\alpha$ properly ensures that queries are proven with a probability $p(\mathbf{F}(\mathbf{a}) = 0)$ bounded away from 0 by a constant. Indeed, in STATREFINE, following the prescribed setting of $\alpha$, we get $p(\mathbf{F}(\mathbf{a}) = 0) = (\frac{d}{d+1})^d$; in ACTIVECOARSEN, we have $p(\mathbf{F}(\mathbf{a}) = 0) = (e^{-1/s})^s = e^{-1}$. (Interestingly, $(\frac{d}{d+1})^d$ is lower bounded by $e^{-1}$ and tends exactly to $e^{-1}$ as $d \to \infty$.)

The preceding discussion motivates a method that keeps $p(\mathbf{F}(\mathbf{a}) = 0) \approxeq e^{-1} \triangleq t$, which we call the *target probability*. We can accomplish this by adapting $\alpha$ as we get new examples from $\mathbf{F}$. The adaptive strategy we will derive is simple: if $\mathbf{F}(\mathbf{a}) = 0$, we decrease $\alpha$; otherwise, we increase $\alpha$. But by how much?

To avoid boundary conditions, we parametrize $\alpha = \sigma(\theta) = (1+e^{-\theta})^{-1}$, which maps $-\infty < \theta < \infty$ to $0 < \alpha < 1$. For convenience, let us define $g(\theta) = p(\mathbf{F}(\mathbf{a}) = 0)$. Now consider minimizing the following function:

$$\mathcal{O}(\theta) = \frac{1}{2}(g(\theta) - t)^2. \tag{6}$$

Clearly, the optimum value (zero) is obtained by setting $\theta$ so that $g(\theta) = t$. We can optimize $\mathcal{O}(\theta)$ by updating its gradient:

$$\theta \leftarrow \theta - \eta\frac{d\mathcal{O}}{d\theta}, \quad \frac{d\mathcal{O}}{d\theta} = (g(\theta) - t)\frac{dg(\theta)}{d\theta}, \tag{7}$$

where $\eta$ is the step size. Of course we cannot evaluate $g(\theta)$, but the key is that we can obtain unbiased samples of $g(\theta)$ by evaluating $\mathbf{F}(\mathbf{a})$ (which we needed to do anyway); specifically, $\mathbb{E}[1 - \mathbf{F}(\mathbf{a})] = g(\theta)$. We can therefore replace the gradient with a stochastic gradient, a classic technique with a rich theory [18]. We note that $\frac{dg(\theta)}{d\theta} > 0$, so we absorb it into the step size $\eta$.[2] This leaves us with the following rule for updating $\theta$ given a random $\mathbf{a}$:

$$\theta \leftarrow \theta - \eta(1 - F(\mathbf{a}) - t). \tag{8}$$

[2] Note that we have not verified the step size conditions that guarantee convergence. Instead, we simply set $\eta = 0.1$ for our experiments, which worked well in practice.

### 4.4 Multiple Queries and Parallelization

So far, we have presented all our algorithms for one query. Given multiple queries, we could just solve each query independently, but this is quite wasteful, since the information obtained from answering one query is not used for other queries. We therefore adopt a *lazy splitting* strategy, where we initially place all the queries in one group and partition the groups over time as we run either STATREFINE or ACTIVECOARSEN. More specifically, we maintain a partition of $\mathbb{Q}$ into a collection of groups $G$, where each $g \in G$ is a subset of $\mathbb{Q}$. We run the algorithm independently for each $g \in G$. After each call to $\mathbf{F}(\mathbf{a})$, we create two new groups, $g_0 = \{q \in g : \mathbf{F}(\mathbf{a})_q = 0\}$ and $g_1 = \{q \in g : \mathbf{F}(\mathbf{a})_q = 1\}$, and set $G$ to $(G \backslash \{g\}) \cup \{g_0, g_1\}$, throwing away empty groups. In $g_0$, we take the $\mathbf{F}(\mathbf{a}) = 0$ branch of the algorithm and in $g_1$, we take the $\mathbf{F}(\mathbf{a}) = 1$ branch.

We thus maintain the invariant that for any two queries $q_1, q_2 \in g$, we have $\mathbf{F}(\mathbf{a})_{q_1} = \mathbf{F}(\mathbf{a})_{q_2}$ for any $\mathbf{a}$ that we have run $\mathbf{F}$ on for $g$ or any of $g$'s ancestral groups. Conceptually, from the point of view of a fixed $q \in \mathbb{Q}$, it is as if we had run the algorithm on $q$ alone, but all of the calls to $\mathbf{F}$ are shared by other queries. When the algorithm terminates, all the queries in one group share the same minimal abstraction. In Section 6, we will see that the number of groups is much smaller than the number of queries.

Our algorithms have been presented as sequential algorithms, but parallelization is possible. STATREFINE is trivial to parallelize because the $n$ training examples are generated independently. Parallelizing ACTIVECOARSEN is slightly more intricate because of the sequential dependence of calls to $\mathbf{F}$. With one processor, we set $\alpha$ so that the target probability is $e^{-1}$. When we have $m$ processors, we set the target probability to $e^{-1}/m$, so that the expected time until a reduction is approximately the same. The upshot of this is that $\alpha$ (monotonically related to $t$) is now smaller and thus we obtain larger reductions.

### 4.5 Discussion of Algorithms

Table 1 summarizes the properties of the four algorithms we have presented in this paper. One of the key advantages of the learning-based approaches (STATREFINE and ACTIVECOARSEN) is that they have a logarithmic dependence on $|\mathbb{J}|$ since they take advantage of *sparsity*, the property that a minimal abstraction has at most $s$ components.

Both algorithms sample random abstractions by including each component with probability $\alpha$, and to avoid an exponential dependence on $s$, it is important to set the probability $\alpha$ properly—for STATREFINE, so that the profile of an irrelevant component is sufficiently different from that of a relevant component; for ACTIVECOARSEN, so that the probability of obtaining a successful reduction of the abstraction is sufficiently large.

The algorithms are also complementary in several respects: STATREFINE is a Monte Carlo algorithm (the running time is fixed, but there is some probability that it does not find a minimal abstraction), whereas ACTIVECOARSEN is a Las Vegas algorithm (the running time is random, but we are guaranteed to find a minimal abstraction). Note that STATREFINE has an extra factor of $d^2$, because it implicitly tries to reason globally about all possible minimal abstractions which involve $d$ dependent components, whereas ACTIVECOARSEN tries to hone in on one minimal abstraction. In practice, we found ACTIVECOARSEN to be more effective, and thus used it to obtain our empirical results.

## 5. Site-varying $k$-limited Points-to Analysis

We now present the static analysis ($\mathbf{F}(\mathbf{a})$ in our general notation) for the abstraction family $\mathcal{A}^{\mathrm{K}}$ (defined in Section 2.2), which allows each allocation and call site to have a separate $k$ value. Figure 4 describes the basic analysis. Each node in the control-flow graph of each method $m \in \mathbb{M}$ is associated with a simple statement (e.g., $v_2 = v_1$). We omit statements that have no effect on our analysis (e.g., operations on data of primitive type). For simplicity, we assume each method has a single argument and no return value. Our actual implementation is a straightforward extension of this simplified analysis which handles multiple arguments, return values, class initializers, and objects allocated through reflection.

Our analysis uses sequences of call sites (in the case of $k$-CFA) or allocation sites (in the case of $k$-object-sensitivity) to represent method contexts. In either case, abstract objects are represented by an allocation site plus the context of the containing method in which the object was allocated. Our abstraction $\mathbf{a}$ maps each site $s \in \mathbb{S}$ to the maximum length $\mathbf{a}_s$ of the context or abstract object to maintain. For example, $k$-CFA (with heap specialization) is represented by $\mathbf{a}_h = k+1$ for each allocation site $h \in \mathbb{H}$ and $\mathbf{a}_i = k$ for each call site $i \in \mathbb{I}$; $k$-object-sensitivity is represented by $\mathbf{a}_h = k$ for each allocation site $h \in \mathbb{H}$. The abstraction determines the input tuples $\mathtt{ext}(s, c, c')$, where prepending $s$ to $c$ and truncating at length $\mathbf{a}_s$ yields $c'$. For example, if $\mathbf{a}_{\mathtt{h2}} = 2$ then we have $\mathtt{ext}(\mathtt{h2}, [\mathtt{i3}, \mathtt{i7}], [\mathtt{h2}, \mathtt{i3}])$.

Our analysis computes the reachable methods (`reachM`), reachable statements (`reachP`), and points-to sets of local variables (`ptsV`), each with the associated context; the context-insensitive points-to sets of static fields (`ptsG`) and heap graph (`heap`); and a context-sensitive call graph (`cg`).

We briefly describe the analysis rules in Datalog. Rule (1) states that the main method $m_{\text{main}}$ is reachable in a distinguished context []. Rule (2) states that a target method of a reachable call site is also reachable. Rule (3) states that every statement in a reachable method is also reachable. Rules (4) through (9) implement the transfer function associated with each kind of statement. Rules (10a) and (10b) populate the call graph while rules (11a) and (11b) propagate the points-to set from the argument of a call site to the formal argument of each target method. Rules (10a) and (11a) are used in the case of $k$-CFA whereas rules (10b) and (11b)

| Algorithm | Minimal | Correct | # calls to $\mathbf{F}$ |
|---|---|---|---|
| DATALOGREFINE | no | yes | $O(1)$ |
| SCANCOARSEN | yes | yes | $O(\|\mathbb{J}\|)$ |
| STATREFINE | prob. $1-\delta$ | prob. $1-\delta$ | $O(sd^2(\log\|\mathbb{J}\| + \log(s/\delta))$ |
| ACTIVECOARSEN | yes | yes | $O(s\log\|\mathbb{J}\|)$ [in expectation] |

Table 1: Summary showing the two properties of Definition 1 for the four algorithms we have presented in this paper. Note that the two machine learning algorithms have only a logarithmic dependence on $|\mathbb{J}|$, the total number of components, and a linear dependence on the size of the largest minimal abstraction $s$.

are used in the case of $k$-object-sensitivity. As dictated by $k$-object-sensitivity, rule (10b) analyzes the target method $m$ in a separate context $o$ for each abstract object $o$ to which the distinguished `this` argument of method $m$ points, and rule (11b) sets the points-to set of the `this` argument of method $m$ in context $o$ to the singleton $\{o\}$.

***Race Detection*** We use the points-to information computed above to answer datarace queries of the form presented in (2), where we include pairs of program points corresponding to heap-accessing statements of the same field in which at least one statement is a write. We implemented the static race detector of [14], which declares a $(p_1, p_2)$ pair as racing if both statements may be reachable, may access thread-escaping data, may point to the same object, and may happen in parallel. All four components rely heavily on the context- and object-sensitive points-to analysis.

## 6. Experiments

In this section, we apply our algorithms (Sections 3 and 4) to the $k$-limited analysis for race detection (Section 5) to answer the main question we started out with: how small are minimal abstractions empirically?

### 6.1 Setup

Our experiments were performed using IBM J9VM 1.6.0 on 32-bit Linux machines. All the analyses (the basic $k$-limited analysis, DATALOGREFINE, and the race detector) were implemented in Chord, an extensible program analysis framework for Java bytecode.[3] The machine learning algorithms simply use the race detector as a black box.

The experiments were applied to five multi-threaded Java benchmarks: an implementation of the Traveling Salesman Problem (`tsp`), a discrete event simulation program (`elevator`), a web crawler (`hedc`), a website downloading and mirroring tool (`weblech`), and a text search tool (`lusearch`).

Table 2 provides the number of classes, number of methods, number of bytecodes of methods, and number of allocation/call sites deemed reachable by 0-CFA in these benchmarks. Table 3 shows the number of races (unproven queries) reported by the coarsest and finest abstractions.

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

| | # classes | # methods | # bytecodes | $\|\mathbb{H}\|$ | $\|\mathbb{I}\|$ |
|---|---|---|---|---|---|
| `tsp` | 167 | 635 | 40K | 656 | 1,721 |
| `elevator` | 170 | 637 | 42K | 663 | 1,893 |
| `hedc` | 335 | 1,965 | 153K | 1,580 | 7,195 |
| `weblech` | 559 | 3,181 | 225K | 2,584 | 12,405 |
| `lusearch` | 627 | 3,798 | 266K | 2,873 | 13,928 |

Table 2: Benchmark characteristics. $|\mathbb{H}|$ is the number of allocation sites, and $|\mathbb{I}|$ is the number of call sites. Together, these determine the number of components in the abstraction family $|\mathbb{J}|$. For $k$-CFA, $|\mathbb{J}| = k(|\mathbb{H}| + |\mathbb{I}|)$; for $k$-object-sensitivity, $|\mathbb{J}| = (k-1)|\mathbb{H}|$.

| `a` | `tsp` | `elevator` | `hedc` | `weblech` | `lusearch` |
|---|---|---|---|---|---|
| $\emptyset$ (CFA) | 570 | 510 | 21,335 | 27,941 | 37,632 |
| $\mathbb{J}$ (CFA) | 494 | 441 | 17,837 | 8,208 | 31,866 |
| diff. ($\|\mathbb{Q}\|$) | 76 | 69 | 3,498 | 19,733 | 5,766 |
| $\emptyset$ (OBJ) | 536 | 475 | 17,137 | 8,063 | 31,428 |
| $\mathbb{J}$ (OBJ) | 489 | 437 | 16,124 | 5,523 | 20,929 |
| diff. ($\|\mathbb{Q}\|$) | 47 | 38 | 1,013 | 2,540 | 10,499 |

Table 3: Number of races (unproven queries) reported using the coarsest abstraction $\emptyset$ (0-CFA/1-object-sensitivity) and the finest abstraction $\mathbb{J}$ (2-CFA/3-object-sensitivity for `tsp`, `elevator` and 1-CFA/2-object-sensitivity for `hedc`, `weblech`, `lusearch`). The difference is the set of queries under consideration (those provable by $\mathbb{J}$ but not by $\emptyset$).

Their difference is the set of queries $\mathbb{Q}$ that we want to prove with a minimal abstraction.

### 6.2 Results

Table 4 summarizes the basic results for DATALOGREFINE and ACTIVECOARSEN. While both find abstractions which prove the same set of queries, ACTIVECOARSEN obtains this precision using an abstraction which is minimal and an order of magnitude smaller than the abstraction found by DATALOGREFINE, which is not guaranteed to be minimal (and is, in fact, far from minimal in our experiments).

Algorithms aside, it is noteworthy in itself that very small abstractions exist. For example, on `tsp`, we can get the same precision as 2-CFA by essentially using a "0.01-CFA" analysis. Indeed, our static analysis using this minimal abstraction was as fast as using 0-CFA, whereas 2-CFA took significantly longer.

**Domains**:

$$
\begin{array}{rrcl}
\text{(method)} & m & \in & \mathbb{M} = \{m_{\text{main}}, ...\} \\
\text{(local variable)} & v & \in & \mathbb{V} \\
\text{(global variable)} & g & \in & \mathbb{G} \\
\text{(object field)} & f & \in & \mathbb{F} \\
\text{(method call site)} & i & \in & \mathbb{I} \\
\text{(allocation site)} & h & \in & \mathbb{H} \\
\text{(allocation/call site)} & s & \in & \mathbb{S} = \mathbb{H} \cup \mathbb{I} \\
\text{(statement)} & p & \in & \mathbb{P} \\
\text{(method context)} & c & \in & \mathbb{C} = \bigcup_{k \geq 0} \mathbb{S}^k \\
\text{(abstract object)} & o & \in & \mathbb{O} = \mathbb{H} \times \mathbb{C} \\
\text{(abstraction)} & \mathbf{a} & \in & \mathcal{A}^{\text{K}} = \{0, 1, \dots\}^{\mathbb{S}}
\end{array}
$$

**Input relations**:

$$
\begin{array}{lcll}
\texttt{body} & \subset & \mathbb{M} \times \mathbb{P} & \text{(method contains statement)} \\
\texttt{trgt} & \subset & \mathbb{I} \times \mathbb{M} & \text{(call site resolves to method)} \\
\texttt{argI} & \subset & \mathbb{I} \times \mathbb{V} & \text{(call site's argument variable)} \\
\texttt{argM} & \subset & \mathbb{M} \times \mathbb{V} & \text{(method's formal argument variable)} \\
\texttt{ext} & \subset & \mathbb{S} \times \mathbb{C} \times \mathbb{C} & \text{(extend context with site)} \\
 & = & \{(s, c, (s, c)[1.. \min\{\mathbf{a}_s, 1+|c|\}]) : s \in \mathbb{S}, c \in \mathbb{C}\} &
\end{array}
$$

**Output relations**:

$$
\begin{array}{lcll}
\texttt{reachM} & \subset & \mathbb{C} \times \mathbb{M} & \text{(reachable methods)} \\
\texttt{reachP} & \subset & \mathbb{C} \times \mathbb{P} & \text{(reachable statements)} \\
\texttt{ptsV} & \subset & \mathbb{C} \times \mathbb{V} \times \mathbb{O} & \text{(points-to sets of local variables)} \\
\texttt{ptsG} & \subset & \mathbb{G} \times \mathbb{O} & \text{(points-to sets of static fields)} \\
\texttt{heap} & \subset & \mathbb{O} \times \mathbb{F} \times \mathbb{O} & \text{(heap graph)} \\
\texttt{cg} & \subset & \mathbb{C} \times \mathbb{I} \times \mathbb{C} \times \mathbb{M} & \text{(call graph)}
\end{array}
$$

$$p \ ::= \ v = \texttt{new}\, h \ \mid \ v_2 = v_1 \ \mid \ g = v \ \mid \ v = g \ \mid \ v_2.f = v_1 \ \mid \ v_2 = v_1.f \ \mid \ i(v)$$

**Rules**:

$$
\begin{array}{lcll}
\texttt{reachM}([], m_{\text{main}}). & & & (1) \\
\texttt{reachM}(c, m) & \Leftarrow & \texttt{cg}(*, *, c, m). & (2) \\
\texttt{reachP}(c, p) & \Leftarrow & \texttt{reachM}(c, m),\ \texttt{body}(m, p). & (3) \\
 & & & \\
\texttt{ptsV}(c, v, o) & \Leftarrow & \texttt{reachP}(c, v = \texttt{new}\, h),\ \texttt{ext}(h, c, o). & (4) \\
\texttt{ptsV}(c, v_2, o) & \Leftarrow & \texttt{reachP}(c, v_2 = v_1),\ \texttt{ptsV}(c, v_1, o). & (5) \\
\texttt{ptsG}(g, o) & \Leftarrow & \texttt{reachP}(c, g = v),\ \texttt{ptsV}(c, v, o). & (6) \\
\texttt{ptsV}(c, v, o) & \Leftarrow & \texttt{reachP}(c, v = g),\ \texttt{ptsG}(g, o). & (7) \\
\texttt{heap}(o_2, f, o_1) & \Leftarrow & \texttt{reachP}(c, v_2.f = v_1),\ \texttt{ptsV}(c, v_1, o_1),\ \texttt{ptsV}(c, v_2, o_2). & (8) \\
\texttt{ptsV}(c, v_2, o_2) & \Leftarrow & \texttt{reachP}(c, v_2 = v_1.f),\ \texttt{ptsV}(c, v_1, o_1),\ \texttt{heap}(o_1, f, o_2). & (9) \\
 & & & \\
\texttt{cg}(c_1, i, c_2, m) & \Leftarrow & \texttt{reachP}(c_1, i),\ \texttt{trgt}(i, m),\ \texttt{ext}(i, c_1, c_2). & (10a) \\
\texttt{cg}(c, i, o, m) & \Leftarrow & \texttt{reachP}(c, i),\ \texttt{trgt}(i, m),\ \texttt{argI}(i, v),\ \texttt{ptsV}(c, v, o). & (10b) \\
\texttt{ptsV}(c_2, v_2, o) & \Leftarrow & \texttt{cg}(c_1, i, c_2, m),\ \texttt{argI}(i, v_1),\ \texttt{argM}(m, v_2),\ \texttt{ptsV}(c_1, v_1, o). & (11a) \\
\texttt{ptsV}(c, v, c) & \Leftarrow & \texttt{reachM}(c, m),\ \texttt{argM}(m, v). & (11b)
\end{array}
$$

Figure 4: Datalog implementation of our $k$-limited points-to analysis with call-graph construction. Our abstraction $\mathbf{a}$ affects the analysis solely through `ext`, which specifies that when we prepend $s$ to $c$, we truncate the resulting sequence to length $\mathbf{a}_s$. If we use rules (10a) and (11a), we get $k$-CFA; if we use (10b) and (11b), we get $k$-object-sensitivity.

***Query Groups*** Recall from Section 4.4 that to deal with multiple queries, we partition the queries into groups and find one minimal abstraction for each group. The abstraction sizes reported so far are the union of the abstractions over all groups. We now take a closer look at the abstractions for individual queries in a group.

First, Table 5 shows that the number of groups is much smaller than the number of queries, which means that many queries share the same minimal abstraction. This is intuitive since many queries depend on the same data and control properties of a program.

Next, Figure 5 shows a histogram of the abstraction sizes across queries. Most queries required a tiny abstraction, only requiring a handful of sites to be refined. For example, for object-sensitivity on `hedc`, over 80% of the queries require just a single allocation site to be refined. Even the most demanding query requires only 9 of the 1,580 sites to be refined. Recall that refining 37 sites suffices to prove all the queries (Table 4). For comparison, DATALOGREFINE refines 906 sites.

## 7. Related Work

One of the key algorithmic tools that we used to find minimal abstractions is randomization. Randomization, a powerful idea, has been previously applied in program analysis, e.g., in random testing [8] and random interpretation [6].

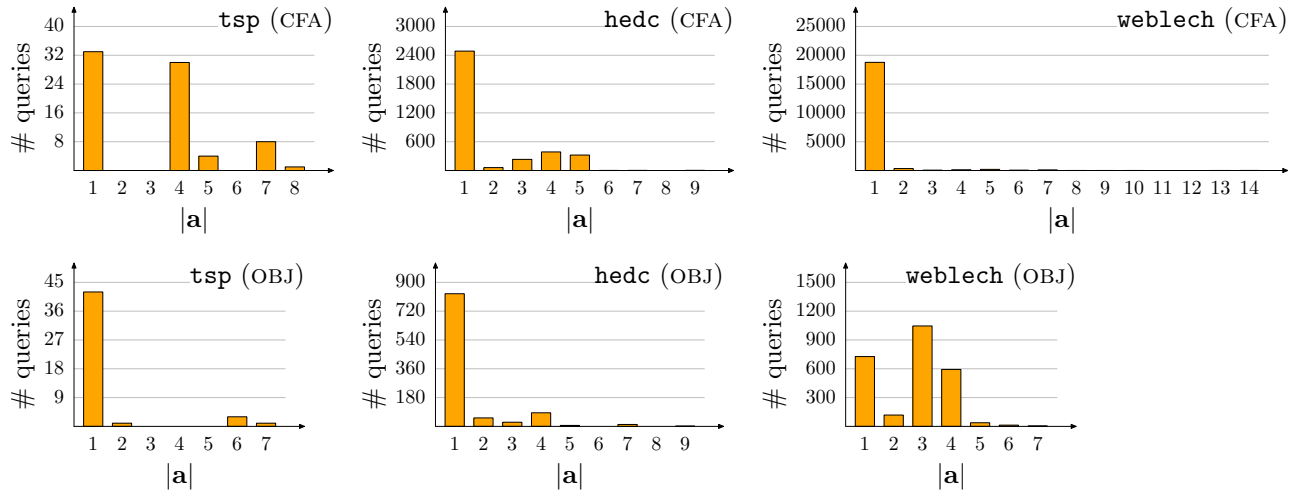


Figure 5: A histogram showing for each abstraction size $|\mathbf{a}|$, the number of queries that have a minimal abstraction of that size (for three of the five benchmarks). Note that most queries need very small abstractions (some need only one site to be refined).

| | $\|\mathbb{J}\|$ | DATALOGREFINE | Minimal |
|---|---|---|---|
| `tsp` (CFA) | 4,754 | 3,170 (67%) | 27 (0.6%) |
| `tsp` (OBJ) | 1,312 | 236 (18%) | 7 (0.5%) |
| `elevator` (CFA) | 5,112 | 3,541 (69%) | 18 (0.4%) |
| `elevator` (OBJ) | 1,326 | 291 (22%) | 6 (0.5%) |
| `hedc` (CFA) | 8,775 | 7,270 (83%) | 90 (1.0%) |
| `hedc` (OBJ) | 1,580 | 906 (57%) | 37 (2.3%) |
| `weblech` (CFA) | 14,989 | 12,737 (85%) | 157 (1.0%) |
| `weblech` (OBJ) | 2,584 | 1,768 (68%) | 48 (1.9%) |
| `lusearch` (CFA) | 16,801 | 14,864 (88%) | 250 (1.5%) |
| `lusearch` (OBJ) | 2,873 | 2,085 (73%) | 56 (1.9%) |

Table 4: This table shows our main results. $|\mathbb{J}|$ is the number of components (the size of the finest abstraction). The next two columns show abstraction sizes (absolute and fraction of $|\mathbb{J}|$) for the abstraction found by DATALOGREFINE and a minimal abstraction found by ACTIVECOARSEN. All abstractions prove the same set of queries. DATALOGREFINE refines anywhere between 18%–85% of the components, while the minimal abstraction is an order of magnitude smaller (0.4%–2.3% of the components).

| | # groups | min. | mean. | max. |
|---|---|---|---|---|
| `tsp` (CFA) | 10 | 1 | 8 | 26 |
| `tsp` (OBJ) | 5 | 1 | 9 | 22 |
| `elevator` (CFA) | 6 | 3 | 12 | 22 |
| `elevator` (OBJ) | 4 | 1 | 10 | 18 |
| `hedc` (CFA) | 63 | 1 | 56 | 546 |
| `hedc` (OBJ) | 43 | 1 | 24 | 300 |
| `weblech` (CFA) | 79 | 1 | 250 | 17,164 |
| `weblech` (OBJ) | 49 | 1 | 52 | 899 |
| `lusearch` (CFA) | 140 | 1 | 41 | 1,346 |
| `lusearch` (OBJ) | 72 | 1 | 146 | 5,104 |

Table 5: Recall that our learning algorithms group the queries (Section 4.4) so that all the queries in one group share the same minimal abstraction. The minimum, mean, and maximum size of a group is reported. In all cases, there is large spread of the number of queries in a group.

Our approach is perhaps more closely associated with machine learning, although there is an important difference in our goals. Machine learning, as exemplified by the PAC learning model [23], is largely concerned with prediction—that is, an algorithm is evaluated on how accurately it can learn a function that predicts well on future inputs. We are instead concerned with finding the smallest input on which a function evaluates to 0. As a result, many of the results, for example on learning monotone DNF formulae [1], are not directly applicable, though many of the bounding techniques used are similar.

One of the key properties that our approach exploited was sparsity—that only a small subset of the components of the abstraction actually matters for proving the desired query. This enabled us to use a logarithmic rather than linear number of examples. Sparsity is one of the main themes in machine learning, signal processing, and statistics. For example, in the area of compressed sensing [4], one also needs a logarithmic number of linear measurements to recover a sparse signal.

Past research in program analysis, and pointer analysis specifically, has proposed various ways to reduce the cost of the analysis while still providing accurate results. Parametrization frameworks provide a mechanism for the user to control the tradeoff between cost and precision of the analysis. Client-driven approaches are capable of computing an exhaustive solution of varying precision while demand-

driven approaches are capable of computing a partial solution of fixed precision. Below we expand upon each of these topics in relation to our work.

Milanova et al. [12, 13] present a parametrized framework for a $k$-object-sensitive points-to analysis, where each local variable can be separately treated context-sensitively or context-insensitively, and different $k$ values can be chosen for different allocation sites. Instantiations of the framework using $k$=1 and $k$=2 are evaluated on side-effect analysis, call-graph construction and virtual-call resolution, and are shown to be significantly more precise than 1-CFA while being comparable to 0-CFA in performance, if not better. Lhoták and Hendren [10, 11] present Paddle, a parametrized framework for BDD-based, $k$-limited alias analysis. They empirically evaluate various instantiations of Paddle, including conventional $k$-CFA, $k$-object-sensitivity and $k$-CFA with heap cloning, on the monomorphic-call-site and cast-safety clients, as well as using traditional metrics, and show that $k$-object-sensitivity is superior to other approaches both in performance and in precision.

Plevyak and Chien [15] use a refinement-based algorithm to determine the concrete types of objects in programs written in the Concurrent Aggregates object-oriented language. When imprecision in the analysis causes a type conflict, the algorithm can improve context sensitivity by performing *function splitting*, and object sensitivity through *container splitting*, which divides object creation sites and thus enables the creation of objects of different types at a single site. In a more recent study, Sridharan and Bodik [21] present a demand-driven, refinement-based alias analysis for Java, and apply it to a cast-safety client. Their algorithm computes an overapproximation of the points-to relation, which is successively refined in response to client demand. At each stage of refinement, the algorithm simultaneously refines handling of heap accesses and method calls along paths, establishing the points-to sets of variables that are relevant for evaluating the client's query.

Guyer and Lin [7] present a client-driven pointer analysis for C that adjusts its precision in response to inaccuracies in the client analysis. Their analysis is a two-pass algorithm: In the first pass, a low-precision pointer analysis is run to detect which statements lead to imprecision in the client analysis; based on this information, a fine-grained precision policy is built for the second pass, which treats these statements with greater context and flow sensitivity. This is similar to our DATALOGREFINE in spirit, but DATALOGREFINE is more general.

In contrast to client-driven analyses, which compute an exhaustive solution of varying precision, demand-driven approaches compute a partial solution of fixed precision. Heintze and Tardieu's demand-driven alias analysis for C [9] performs a provably optimal computation to determine the points-to sets of variables queried by the client. Their analysis is applied to call-graph construction in the presence of function pointers. A more recent alias analysis developed by Zheng and Rugina [27] uses a demand-driven algorithm that is capable of answering alias queries without constructing points-to sets.

Reps [16, 17] shows how to automatically obtain demand-driven versions of program analyses from their exhaustive counterparts by applying the "magic-sets transformation" developed in the logic-programming and deductive-database communities. The exhaustive analysis is expressed in Datalog, akin to that in our DATALOGREFINE approach, but unlike us, they are interested in answering specific queries of interest in the program being analyzed as opposed to all queries. For instance, while we are interested in finding all pairs of points $(p_1, p_2)$ in a given program that may be involved in a race, they may be interested in finding all points that may be involved in a race with the statement at point `p42` in the given program. The magic-set transformation takes as input a Datalog program which performs the exhaustive analysis along with a set of specified queries. It produces as output the demand-driven version of the analysis, also in Datalog, but which eliminates computation from the original analysis that is unnecessary for evaluating the specified queries. In contrast, our transformation takes as input a parametrized analysis expressed in Datalog and produces as output another analysis, also in Datalog, whose goal is to compute all possible parameters that may affect the output of the original analysis, for instance, all possible sites in a given program whose small $k$ values may be responsible for a set of races being reported by the original analysis.

## 8. Conclusion

We started this study with a basic question: what is the minimal abstraction needed to prove a set of queries of interest? To answer this question, we developed two machine learning algorithms and applied them to find minimal $k$ values of call/allocation sites for a static race detector. The key theme in this work is *sparsity*, the property that very few components of an abstraction are needed to prove a query. The ramifications are two-fold: Theoretically, we show that our algorithms are efficient under sparsity; empirically, we found that the minimal abstractions are quite small—only 0.4–2.3% of the sites are needed to prove all the queries of interest.

## A. Proofs

*Proof of Theorem 2.* Note that the algorithm will run for at most $s$ iterations, where $s$ is the size of the largest minimal abstraction. If we set $n$ so that STATREFINE chooses a dependent component each iteration with probability at least $1 - \delta/s$, then the algorithm will succeed with probability at least $1 - \delta$ (by a union bound).

Let us now focus on one iteration. The main idea is that a dependent component $j$ is more correlated with proving the query ($\mathbf{F}(\mathbf{a}) = 0$) than one that is independent. This enables

picking it out with high probability given sufficiently large $n$.

Recall that $D$ is the set of dependent components with $|D| = d$. Fix a dependent component $j^+ \in D$. Let $B_{j^-}$ be the event that $n_{j^-} > n_{j^+}$, and $B$ be the event that $B_{j^-}$ holds for any independent component $j^- \in \mathbb{J} \backslash D$. Note that if $B$ does not happen, then the algorithm will correctly pick a dependent component (possibly $j^+$). Thus, the main focus is on showing that $P(B) \leq \delta/s$. First, by a union bound, we have

$$P(B) \leq \sum_{j^-} P(B_{j^-}) \leq |\mathbb{J}| \max_{j^-} P(B_{j^-}). \tag{9}$$

For each training example $\mathbf{a}^{(i)}$, define $X_i = (1 - \mathbf{F}(\mathbf{a}^{(i)}))(\mathbf{a}^{(i)}_{j^-} - \mathbf{a}^{(i)}_{j^+})$. Observe that $B_{j^-}$ happens exactly when $\frac{1}{n}(n_{j^-} - n_{j^+}) = \frac{1}{n}\sum_{i=1}^n X_i > 0$. We now bound this quantity using Hoeffding's inequality,[4] where the mean is

$$\mathbb{E}[X_i] = p(\mathbf{F}(\mathbf{a}) = 0, \mathbf{a}_{j^-} = 1) - p(\mathbf{F}(\mathbf{a}) = 0, \mathbf{a}_{j^+} = 1),$$

and the bounds are $a = -1$ and $b = +1$. Setting $\epsilon = -\mathbb{E}[X_i]$, we get:

$$p(B_{j^-}) \leq e^{-n\epsilon^2/2}, \quad j^+ \in D, j^- \notin D. \tag{10}$$

Substituting (10) into (9) and rearranging terms, we can solve for $n$:

$$\delta/s \leq |\mathbb{J}| e^{-n\epsilon^2/2} \;\Rightarrow\; n \geq \frac{2(\log|\mathbb{J}| + \log(s/\delta))}{\epsilon^2}. \tag{11}$$

Now it remains to lower bound $\epsilon$, which intuitively represents the gap (in the amount of correlation with proving the query) between a dependent component and an independent one. Note that $p(\mathbf{a}_j = 1) = \alpha$ for any $j \in \mathbb{J}$. Also, $j^-$ is independent of $\mathbf{F}(\mathbf{a})$, so $p(\mathbf{F}(\mathbf{a}) = 0 \mid \mathbf{a}_{j^-} = 1) = p(\mathbf{F}(\mathbf{a}) = 0)$. Using these two facts, we have:

$$\epsilon = \alpha(p(\mathbf{F}(\mathbf{a}) = 0 \mid \mathbf{a}_{j^+} = 1) - p(\mathbf{F}(\mathbf{a}) = 0)). \tag{12}$$

Let $C$ be the set of minimal abstractions of $\mathbf{F}$. We can think of $C$ as a set of clauses in a DNF formula: $\mathbf{F}(\mathbf{a}; C) = \neg \bigvee_{c \in C} \bigwedge_{j \in c} \mathbf{a}_j$, where we explicitly mark the dependence of $\mathbf{F}$ on the clauses $C$. For example, $C = \{\{1,2\},\{3\}\}$ corresponds to $\mathbf{F}(\mathbf{a}) = \neg[(\mathbf{a}_1 \wedge \mathbf{a}_2) \vee \mathbf{a}_3]$. Next, let $C_j = \{c \in C : j \in c\}$ be the clauses containing $j$. Rewrite $p(\mathbf{F}(\mathbf{a}) = 0)$ as the sum of two parts, one that depends on $j^+$ and one that does not:

$$\begin{aligned} p(\mathbf{F}(\mathbf{a}) = 0) = {} & p(\mathbf{F}(\mathbf{a}; C_{j^+}) = 0, \mathbf{F}(\mathbf{a}; C \backslash C_{j^+}) = 1) + \\ & p(\mathbf{F}(\mathbf{a}; C \backslash C_{j^+}) = 0). \end{aligned} \tag{13}$$

Computing $p(\mathbf{F}(\mathbf{a}) = 0 \mid \mathbf{a}_{j^+} = 1)$ is similar; the only difference due to conditioning on $\mathbf{a}_{j^+} = 1$ is that the first term has an additional factor of $\frac{1}{\alpha}$ because conditioning divides by $p(\mathbf{a}_{j^+} = 1) = \alpha$. The second term is unchanged because no $c \notin C_{j^+}$ depends on $\mathbf{a}_{j^+}$. Plugging these two results back into (12) yields:

$$\epsilon = (1 - \alpha) p(\mathbf{F}(\mathbf{a}; C_{j^+}) = 0, \mathbf{F}(\mathbf{a}; C \backslash C_{j^+}) = 1). \tag{14}$$

Now we want to lower bound (14) over all possible $\mathbf{F}$ (equivalently, $C$), where $j^+$ is allowed to depend on $C$. It turns out that the worst possible $C$ is obtained by either having $d$ disjoint clauses ($C = \{\{j\} : j \in D\}$) or one clause ($C = \{D\}$ if $s = d$). The intuition is that if $C$ has $d$ clauses, there are many opportunities ($d - 1$ of them) for some $c \notin C_{j^+}$ to activate, making it hard to realize that $j^+$ is a dependent component; in this case, $\epsilon = (1 - \alpha)\alpha(1 - \alpha)^{d-1}$. If $C$ has one clause, then it is very hard (probability $\alpha^d$) to even activate this clause; in this case, $\epsilon = (1 - \alpha)\alpha^d$.

Let us focus on the case where $C$ has $d$ clauses. We can maximize $\epsilon$ with respect to $\alpha$ by setting the derivative $\frac{d\epsilon}{d\alpha} = 0$ and solving for $\alpha$. Doing this yields $\alpha = \frac{1}{d+1}$ as the optimal value. Plugging this value back into the expression for $\epsilon$, we get that $\epsilon = \frac{1}{d+1}(\frac{d}{d+1})^d$. The second factor can be lower bounded by $e^{-1}$, so $\epsilon^{-2} = O(d^2)$. Combining this with (11) completes the proof. □

*Proof of Theorem 3.* Let $T(\mathbf{a}^{\mathrm{U}})$ be the expected number of calls that ACTIVECOARSEN($\mathbf{a}^{\mathrm{U}}$) makes to $\mathbf{F}$. The recursive computation of this quantity parallels the pseudocode of Figure 3:

$$T(\mathbf{a}^{\mathrm{U}}) \tag{15}$$
$$= \begin{cases} |\mathbf{a}^{\mathrm{U}}| & \text{if } |\mathbf{a}^{\mathrm{U}}| \leq s+1 \\ 1 + \mathbb{E}[(1 - \mathbf{F}(\mathbf{a}))T(\mathbf{a}) + \mathbf{F}(\mathbf{a})T(\mathbf{a}^{\mathrm{U}})] & \text{otherwise,} \end{cases}$$

where $\mathbf{a} \leftarrow$ SAMPLE($\emptyset, \mathbf{a}^{\mathrm{U}}$) is a random binary vector. By assumption, there exists an abstraction $\mathbf{a}^* \preceq \mathbf{a}^{\mathrm{U}}$ of size $s$ that proves the query. Define $\mathbf{G}(\mathbf{a}) = \neg(\mathbf{a}^* \preceq \mathbf{a})$, which is 0 when all components in $\mathbf{a}^*$ are active under the random $\mathbf{a}$. We have $p(\mathbf{G}(\mathbf{a}) = 0) = p(\mathbf{a}^* \preceq \mathbf{a}) = \alpha^s$. Note that $\mathbf{G}(\mathbf{a}) \geq \mathbf{F}(\mathbf{a})$, as activating $\mathbf{a}^*$ suffices to prove the query. We assume $T(\mathbf{a}) \leq T(\mathbf{a}^{\mathrm{U}})$ ($T$ is monotonic), so we get an upper bound by replacing $\mathbf{F}$ with $\mathbf{G}$ and performing some algebra:

$$T(\mathbf{a}^{\mathrm{U}}) \leq 1 + \mathbb{E}[(1 - \mathbf{G}(\mathbf{a}))T(\mathbf{a}) + \mathbf{G}(\mathbf{a})T(\mathbf{a}^{\mathrm{U}})] \tag{16}$$
$$\leq 1 + \alpha^s \mathbb{E}[T(\mathbf{a}) \mid \mathbf{a}^* \preceq \mathbf{a}] + (1 - \alpha^s)T(\mathbf{a}^{\mathrm{U}}) \tag{17}$$
$$\leq \mathbb{E}[T(\mathbf{a}) \mid \mathbf{a}^* \preceq \mathbf{a}] + \alpha^{-s}. \tag{18}$$

Overloading notation, we write $T(n) = \max_{|\mathbf{a}|=n} T(\mathbf{a})$ to be the maximum over abstractions of size $n$. Note that $|\mathbf{a}|$ given $\mathbf{a}^* \preceq \mathbf{a}$ is $s$ plus a binomial random variable $N$ with expectation $\alpha(n - s)$.

Using the crude bound $T(n) \leq (1 - \alpha^n)T(n-1) + \alpha^n T(n)\alpha^{-s}$, we see that $T(n) \leq \frac{\alpha^{-s}}{1-\alpha^n} \cdot n$; in particular, $T(n)$ is sublinear. Moreover, $T(n)$ is concave for large

[4] Hoeffding's inequality: if $X_1, \ldots, X_n$ are i.i.d. random variables with $a \leq X_i \leq b$, then $p(\frac{1}{n}\sum_{i=1}^n X_i > \mathbb{E}[X_i] + \epsilon) \leq \exp\left\{-\frac{2n\epsilon^2}{(b-a)^2}\right\}$.

enough $n$, so we can use Jensen's inequality to swap $T$ and $\mathbb{E}$:

$$T(n) \leq T(\mathbb{E}[s+N]) + \alpha^{-s} = T(s+\alpha(n-s)) + \alpha^{-s}. \tag{19}$$

Solving the recurrence, we obtain:

$$T(n) \leq \frac{\alpha^{-s}\log n}{\log \alpha^{-1}} + s + 1. \tag{20}$$

From (20), we can see the tradeoff between reducing the number of iterations (by increasing $\log \alpha^{-1}$) versus reducing the number of trials (by decreasing $\alpha^{-s}$).

We now set $\alpha$ to minimize the upper bound. Differentiate with respect to $x = \alpha^{-1}$ and set the derivative to zero: $\frac{sx^{s-1}}{\log x} - \frac{x^{s-1}}{\log^2 x} = 0$. Solving this equation yields $\alpha = e^{-1/s}$. Plugging this value back into (20) yields $T(n) = es\log n + s + 1 = O(s\log n)$. □

## References


[1] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

[2] T. Ball and S. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of ACM Symp. on Principles of Programming Languages (POPL)*, pages 1–3, 2002.

[3] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.

[4] D. Donoho. Compressed sensing. *IEEE Trans. on Information Theory*, 52(4):1289–1306, 2006.

[5] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. pages 72–83, 1997.

[6] S. Gulwani. *Program Analysis using Random Interpretation*. PhD thesis, UC Berkeley, 2005.

[7] S. Guyer and C. Lin. Client-driven pointer analysis. In *Proceedings of Intl. Static Analysis Symposium*, pages 214–236, 2003.

[8] D. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978, 1994.

[9] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Proceedings of ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 24–34, 2001.

[10] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In *Proceedings of Intl. Conf. on Compiler Construction*, pages 47–64, 2006.

[11] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Transactions on Software Engineering and Methodology*, 18(1):1–53, 2008.

[12] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of ACM Intl. Symp. on Software Testing and Analysis*, pages 1–11, 2002.

[13] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2005.

[14] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 308–319.

[15] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 324–340.

[16] T. W. Reps. Demand interprocedural program analysis using logic databases. In *Workshop on Programming with Logic Databases*, pages 163–196, 1993.

[17] T. W. Reps. Solving demand versions of interprocedural analysis problems. In *Proceedings of Intl. Conf. on Compiler Construction*, pages 389–403, 1994.

[18] H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[19] M. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[20] O. Shivers. Control-flow analysis in Scheme. In *Proceedings of ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 164–174, 1988.

[21] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of ACM Conf. on Programming Language Design and Implementation*, pages 387–400, 2006.

[22] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *Proceedings of ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 59–76, 2005.

[23] L. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[24] M. J. Wainwright. Sharp thresholds for noisy and high-dimensional recovery of sparsity using $\ell_1$-constrained quadratic programming (lasso). *IEEE Transactions on Information Theory*, 55:2183–2202, 2009.

[25] J. Whaley. *Context-Sensitive Pointer Analysis using Binary Decision Diagrams*. PhD thesis, Stanford University, 2007.

[26] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 131–144, 2004.

[27] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *Proceedings of ACM Symp. on Principles of Programming Languages (POPL)*, pages 197–208, 1998.