Scaling Abstraction Refinement via Pruning

Percy Liang
UC Berkeley
pliang@cs.berkeley.edu

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

Abstract

Many static analyses do not scale as they are made more precise. For example, increasing the amount of context sensitivity in a k-limited pointer analysis causes the number of contexts to grow exponentially with k. Iterative refinement techniques can mitigate this growth by starting with a coarse abstraction and only refining parts of the abstraction that are deemed relevant with respect to a given client.

In this paper, we introduce a new technique called *pruning* that uses client feedback in a different way. The basic idea is to use coarse abstractions to prune away parts of the program analysis deemed irrelevant for proving a client query, and then using finer abstractions on the sliced program analysis. For a k-limited pointer analysis, this approach amounts to adaptively refining and pruning a set of prefix patterns representing the contexts relevant for the client. By pruning, we are able to scale up to much more expensive abstractions than before. We also prove that the pruned analysis is both sound and complete, that is, it yields the same results as an analysis that uses a more expensive abstraction directly without pruning.

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

General Terms Algorithms, Experimentation, Theory, Verification

Keywords heap abstraction, static analysis, concurrency, abstraction refinement, pruning, slicing

1. Introduction

Making a static analysis more precise requires increasing the complexity of the underlying abstraction—in pointer analysis, by increasing the amount of context/object sensitivity [7, 8, 12, 13, 16, 22]; or in model checking, by adding more abstraction predicates [1, 3]. However, the complexity of these analyses often grows exponentially as the abstraction is refined. Much work has been done on curbing this exponential growth (e.g., client-driven [4] and demand-driven [5] approaches in pointer analysis; lazy abstraction [6, 11] and other iterative refinement approaches in model checking). We refer to these techniques as *selected refinement*, where the main idea is to only refine an abstraction along components deemed relevant according to client feedback.

In this paper, we introduce *pruning*, a new and orthogonal approach which represents a significant departure from existing selected refinement techniques. Pruning is applicable to static analyses expressed as a set of inference rules where a program property of interest (a client query) is proven by the inability to derive a designated fact using the given rules. For concreteness, assume that the static analysis is expressed as a Datalog program. A Datalog program takes a set of input tuples and derives new tuples via a set of inference rules. These inference rules capture the abstract semantics of the static analysis and the evaluation of the client query using the analysis result; the input tuples encode the program we are analyzing and the abstraction we are using. The program property is proven if a designated query tuple cannot be derived.

The key idea behind pruning is to identify input tuples which are provably irrelevant for deriving the query tuple and remove these tuples completely from analysis. Consequently, when the abstraction is refined, only the relevant tuples are refined, potentially resulting in major computational savings. It is helpful to think of pruning in terms of generalized program slicing, where irrelevant parts of the program (irrelevant input tuples) are removed, resulting in a smaller program that is cheaper to analyze. Existing selected refinement techniques attempt to keep the set of input tuples small by simply not refining some of them; pruning keeps the set small by removing some of them entirely.

Pruning can be a dangerous affair though. With selected refinement, we are always performing a static analysis with respect to an abstraction and therefore inherit the soundness guarantees of abstract interpretation. However, once we start pruning input tuples, we are no longer running a valid static analysis on the original program. Soundness therefore is no longer automatic, though we do prove that our method is sound with respect to a given client.

While soundness is trivial for selected refinement but requires some argument for pruning, the situation is reversed for *completeness*. By completeness, we mean that the analysis is as precise as if we had refined all the components of an abstraction. Selected refinement only refines a subset of an abstraction, so it is unclear that the resulting abstraction is as precise as an abstraction obtained by refining all components. However, with pruning, we conceptually work with the fully-refined abstraction; by removing input tuples, we cannot prove fewer queries; thus, completeness is automatic.

To capitalize on the idea of pruning, we propose an algorithm, which we call the *Prune-Refine algorithm*. The idea is to start with a coarse abstraction and prune the irrelevant input tuples before refining the abstraction; the algorithm iterates until the query is proven or a specified computational budget is reached. We prove that the Prune-Refine algorithm computes the same answers to client queries as directly using a refined abstraction without pruning, which would be precise but possibly infeasible.

We apply pruning to the k-object-sensitivity abstraction [12], where objects in the heap are abstracted using chains of allocation sites; these chains are the input tuples we maintain. To facilitate pruning, we introduce two new heap abstractions: The first abstraction truncates chains to avoid repeating allocation sites; this allows us to increase k without getting bogged down by long chains created due to recursion. The second abstraction replaces allocation sites by types (a generalization of [17]). We show that these abstractions can be composed to further improve the effectiveness of pruning.

PLDI'11, June 4–8, 2011, San Jose, California, USA.

We ran our experiments on five Java benchmarks using three clients that depend heavily on having a precise pointer analysis: downcast safety checking, monomorphic call site inference, and race detection. We show that with pruning, our Prune-Refine algorithm enables us to perform a k-object-sensitive pointer analysis with a substantially much finer abstraction (larger k) compared to a full k-object-sensitive pointer analysis or even using the selected refinement strategy of [10]. In a few cases, the non-pruning approaches hit a wall around $k = 3$ but the Prune-Refine algorithm is able to go well beyond $k = 10$.

2. Preliminaries

Our pruning technique works on Datalog, a general language which can be used to express static analyses declaratively [2, 21]. Normally, these analyses and their underlying abstractions are encoded by one monolithic Datalog program which evaluates a query abstractly. For us, it will be convenient to consider the Datalog program, which evaluates a query concretely,[1] as distinct from the abstraction, which transforms the input to the Datalog program, resulting in an abstract evaluation of the query. This separation allows us to make theoretical statements comparing the behavior of the same Datalog program across different abstractions.

We first define Datalog and the computation of a concrete query (Section 2.1). Then, we focus on the abstraction (Section 2.2), which interacts with the Datalog program by transforming the input tuples. Throughout this section, we will use Figure 1 as a running example.

2.1 Datalog

A *Datalog program* consists of a set of *constants* $\mathcal{C}$ (e.g., $0, [03] \in \mathcal{C}$), a set of *variables* $\mathcal{V}$ (e.g., $i, j \in \mathcal{V}$), and a set of *relations* $\mathcal{R}$ (e.g., $\texttt{edge} \in \mathcal{R}$).

A *term* t consists of a relation $t.r \in \mathcal{R}$ and a list of arguments $t.\mathbf{a}$, where each argument $t.a_i$ is either a variable or a constant, (that is, $t.a_i \in \mathcal{V} \cup \mathcal{C}$) for $i = 1, \ldots, |t.\mathbf{a}|$. We will write a term in any of the following three equivalent ways:

$$t \quad \equiv \quad t.r(t.\mathbf{a}) \quad \equiv \quad t.r(t.a_1, \ldots, t.a_{|t.\mathbf{a}|}). \tag{1}$$

For example, $\texttt{ext}(j, c, c')$ is a term. We call a term whose arguments are all constants a *tuple* (e.g., $\texttt{ext}(0, [\,], [0])$). Note that the tuple includes the relation as well as the arguments. We let x_{Q} denote a designated *query tuple* (e.g., $\texttt{common}(\text{G}_1, \text{G}_2, 3)$), whose truth value we want to determine.

Let $\mathcal{Z}$ denote the set of rules, where each *rule* $z \in \mathcal{Z}$ consists of a target term $z.t$ and a set of source terms $z.\mathbf{s}$. We write a rule with $z.t = t$ and $z.\mathbf{s} = \{s_1, \ldots, s_k\}$ as

$$t \Leftarrow s_1, \ldots, s_k. \tag{2}$$

An *assignment* is a function $f : \mathcal{V} \mapsto \mathcal{C}$ which maps variables to constants. To simplify notation later, we extend an assignment f so that it can be applied (i) to constants ($f(c) = c$ for $c \in \mathcal{C}$), and (ii) to terms by replacing the variables in the term with constants ($f(t) = t.r(f(t.a_1), \ldots, f(t.a_{|t.\mathbf{a}|}))$).

Derivations A Datalog program takes a set of input tuples and derives new tuples. To formalize this computation, we define the notation of a derivation.

A *derivation* (of the query x_{Q}) with respect to a set of input tuples X is a sequence $\mathbf{x} = (x_1, \ldots, x_n)$ such that

[1] We refer to this computation as "concrete" to contrast with abstract computation we will consider later, but note that this "concrete" computation could already contain some level of abstraction. For example, the Datalog program might correspond to ∞-object-sensitivity without abstraction and k-object-sensitivity with abstraction.

Graph Example

Input relations:

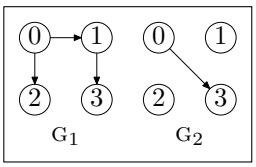
$\texttt{edge}(g, i, j)$	(edge from node i to node j in graph g)
$\texttt{head}(c, i)$	(first element of array c is i)
$\texttt{ext}(i, c, c')$	(i prepended to c yields c': $c' = [i] + c$)

Rules:

$\texttt{path}(g, [0])$.
$\texttt{path}(g, c') \Leftarrow \texttt{path}(g, c),\ \texttt{head}(c, i),\ \texttt{edge}(g, i, j),\ \texttt{ext}(j, c, c')$.
$\texttt{common}(g_1, g_2, i) \Leftarrow \texttt{path}(g_1, c),\ \texttt{path}(g_2, c),\ \texttt{head}(c, i)$.

Query tuple: $x_{\text{Q}} = \texttt{common}(\text{G}_1, \text{G}_2, 3)$.

Constants: $\mathcal{C} = \{\text{G}_1, \text{G}_2, 0, 1, 2, 3, [0], [01], \ldots\}$.



Input tuples:
$\texttt{edge}(\text{G}_1, 0, 1)$
$\texttt{edge}(\text{G}_1, 0, 2)$
$\texttt{edge}(\text{G}_1, 1, 3)$
$\texttt{edge}(\text{G}_2, 0, 3)$
$\texttt{head}([0], 0)$
...
$\texttt{ext}(1, [0], [10])$
...

Derived tuples:
$\texttt{path}(\text{G}_1, [0])$
$\texttt{path}(\text{G}_1, [10])$
$\texttt{path}(\text{G}_1, [20])$
$\texttt{path}(\text{G}_1, [310])$
$\texttt{path}(\text{G}_2, [0])$
$\texttt{path}(\text{G}_2, [30])$

Figure 1. A simple example illustrating Datalog: Suppose we have two graphs G_1 and G_2 defined on the same set of nodes $\{0, 1, 2, 3\}$, and we want to compute the query tuple $\texttt{common}(\text{G}_1, \text{G}_2, 3)$, asking whether the two graphs have a common path from node 0 to node 3. Given the input tuples encoding the graph, the Datalog program computes a set of derived tuples from the rules. In this case, the absence of $\texttt{common}(\text{G}_1, \text{G}_2, 3)$ from the derived tuples means the query is false (proven).

$\mathcal{C}$	(concrete values)
$\mathcal{P}(\mathcal{C})$	(abstract values)
$\alpha : \mathcal{C} \mapsto \mathcal{P}(\mathcal{C})$	(abstraction, maps to equivalence class)
x_{Q}	(designated query tuple)
$\mathbf{D}(X)$	(derivations of x_{Q} using input tuples X)
$\mathbf{E}(X)$	(tuples involved in deriving x_{Q})
$\mathbf{P}(X)$	(input tuples relevant to deriving x_{Q})
A_k	(abstract input tuples after k iterations)
$\tilde{A}_k$	(relevant abstract input tuples after pruning)

Figure 2. Notation.

(i) for each $i = 1, \ldots, n$, we have $x_i \in X$; or there exists a set of indices J such that (J, i) satisfies the following two conditions: $j < i$ for each $j \in J$, and there is a rule $z \in \mathcal{Z}$ and an assignment f such that $f(z.t) = x_i$ and $\{x_j : j \in J\} = \{f(s) : s \in z.\mathbf{s}\}$;

(ii) $x_n = x_{\text{Q}}$; and

(iii) for each $j = 1, \ldots, n-1$, there exists J such that $j \in J$ and an index i such that (J, i) satisfies the two conditions in (i).

Define $\mathbf{D}(X)$ to be the set of all derivations with respect to the input tuples X.

Condition (i) says that each tuple in a derivation should either be given as an input tuple ($x_i \in X$) or be the result of some rule $z \in \mathcal{Z}$. Condition (ii) says that the query tuple x_{Q} is derived at the end. Condition (iii) says that in the derivation of x_{Q}, every tuple is somehow "relevant" for deriving x_{Q}.

We say that the query x_Q is false (proven) if and only if $\mathbf{D}(X)$ is empty. Although this answer to the query is the ultimate quantity of interest, the Datalog program can be used to provide more information, which will be useful for pruning. Specifically, we define $\mathbf{E}(X)$ as the set of all tuples used in any derivation (of x_Q) and $\mathbf{P}(X)$ to be the subset of $\mathbf{E}(X)$ which are input tuples:

$$\mathbf{E}(X) \triangleq \bigcup_{\mathbf{x} \in \mathbf{D}(X)} \mathbf{x}, \tag{3}$$

$$\mathbf{P}(X) \triangleq X \cap \mathbf{E}(X). \tag{4}$$

We call $\mathbf{P}(X)$ the set of *relevant input tuples*. As we will see later, any tuple not in this set can be safely pruned. In fact, $\mathbf{P}(X)$ also tells us whether the query is true or false. In particular, $\mathbf{D}(X) = \emptyset$ if and only if $\mathbf{P}(X) = \emptyset$ (assuming x_Q cannot be derived trivially without inputs). This equivalence suggests that proving and pruning are intimately related; in some sense, proving the query is just pruning away the query tuple. In the remainder of the paper, we will make heavy use of $\mathbf{P}$ as the principal proving/pruning operator. In our graph example, $\mathbf{D}(X) = \mathbf{P}(X) = \emptyset$, but as we will see later, this is not true if we apply an abstraction to X.

Computation Given input tuples X, a Datalog solver returns the set of derived tuples Y; the query is proven if $x_Q \notin Y$. Note that Y is a superset of the relevant derived tuples $\mathbf{E}(X)$, which itself is a superset of the relevant input tuples $\mathbf{P}(X)$, which is needed for pruning.

We can compute $\mathbf{P}(X)$ by using the Datalog program transformation technique described in [10]: We augment our existing Datalog program with a set of new relations $\mathcal{R}' = \{r' : r \in \mathcal{R}\}$. For a term $t = t.r(t.\mathbf{a})$ we let $t' = t.r'(t.\mathbf{a})$ be the term that uses the corresponding new relation $t.r'$. We then add the following new Datalog rules:

$$x'_Q \Leftarrow x_Q, \tag{5}$$

$$s' \Leftarrow z.t', z.\mathbf{s} \quad \text{for each } z \in \mathcal{Z} \text{ and } s \in z.\mathbf{s}. \tag{6}$$

For example, the last rule of the original Datalog program in Figure 1 generates the following three new rules:

$$\begin{array}{rcl}
\texttt{path}'(g_1, c) & \Leftarrow & \texttt{common}'(g_1, g_2, i),\ \texttt{path}(g_1, c),\ \texttt{path}(g_2, c),\ \texttt{head}(c, i). \\
\texttt{path}'(g_2, c) & \Leftarrow & \texttt{common}'(g_1, g_2, i),\ \texttt{path}(g_1, c),\ \texttt{path}(g_2, c),\ \texttt{head}(c, i). \\
\texttt{head}'(c, i) & \Leftarrow & \texttt{common}'(g_1, g_2, i),\ \texttt{path}(g_1, c),\ \texttt{path}(g_2, c),\ \texttt{head}(c, i).
\end{array}$$

The key is that a tuple x' is derived by the new Datalog program if and only if $x \in \mathbf{E}(X)$. Rules generated by (5) and (6) construct $\mathbf{E}(X)$ recursively: The base case (5) states that the query tuple $x_Q \in \mathbf{E}(X)$. The recursive case (6) states that if $x \in \mathbf{E}(X)$ and a rule z (with some assignment f) was used to produce x, then for every source term $s \in z.\mathbf{s}$ of that rule, we also have $f(s) \in \mathbf{E}(X)$. Having obtained $\mathbf{E}(X)$, we get $\mathbf{P}(X)$ by keeping only tuples in X.

The advantage of this technique is that we can use any Datalog solver as a black-box to compute $\mathbf{P}(X)$. In practice, we will not actually run $\mathbf{P}$ on concrete input tuples X, but on abstract input tuples. From the point of view of the Datalog solver, there is no difference between the two. We consider constructing abstract tuples next.

2.2 Abstractions

Given a Datalog program, an *abstraction* is an equivalence relation over constants $\mathcal{C}$. In particular, we represent the abstraction as the function which maps each constant to its equivalence class.

Definition 1. *An abstraction is a function $\alpha : \mathcal{C} \to \mathcal{P}(\mathcal{C})$ such that for each set $s \in \text{range}(\alpha)$, we have $\alpha(c) = s$ for all $c \in s$.*

We will refer to constants $\mathcal{C}$ as the concrete values and range(α) as the abstract values. We assume the natural partial order on abstractions, where $\alpha_1 \preceq \alpha_2$ if and only if $\alpha_1(c) \supseteq \alpha_2(c)$ for all c—that is, α_2 is finer than α_1.

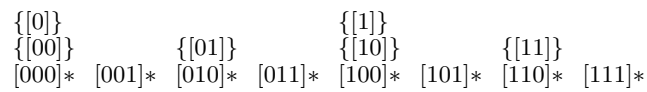


Figure 3. The 14 abstract values defined by the k-limited abstraction π_k with $\mathbb{H} = \{0, 1\}$ and $k = 3$. Specifically, π_k maps each chain $c \in \mathbb{H}^*$ to one of the values above.

Example: k-limited abstraction The main abstraction we will work with in this paper is the k-limited abstraction [12, 16]. Our general theory does not depend on this particular choice, but we present the abstraction here so we can use it as a running example.

First, we define some notation. Let $\mathbb{H}$ be an arbitrary set; for the graph example of Figure 1, let $\mathbb{H} = \{0, 1, 2, 3\}$ be the nodes of the graph; later, $\mathbb{H}$ will be the set of allocation sites in a program. Define a *chain* $c \in \mathbb{H}^*$ to be a finite sequence of elements from this set. Let $|c|$ denote the length of the chain. Let $c[i]$ be the i-th element of c (starting with index 1) and let $c[i..j]$ be the subchain $[c[i] \cdots c[j]]$ (boundary cases: $c[i..j] = [\,]$ if $i > j$ and $c[i..j] = c[i..|c|]$ if $j > |c|$). For two chains c_1 and c_2, let $c_1 + c_2$ denote their concatenation.

The *k-limited abstraction* partitions chains based on their length k prefix. First, for a chain c, let $c*$ denote the set of all chains with prefix c; formally:

$$c* \triangleq \{c' \in \mathbb{H}^* : c'[1..|c|] = c\}. \tag{7}$$

For an integer truncation level $k \geq 0$, define the k-limited abstraction π_k as follows:

$$\pi_k(c) \triangleq \begin{cases} \{c\} & \text{if } |c| < k \\ c[1..k]* & \text{if } |c| \geq k. \end{cases} \tag{8}$$

If the concrete chain c is shorter than length k, we map it to the singleton set $\{c\}$; otherwise, we map it to the set of chains that share the first k elements. It is easy to verify that π_k is a valid abstraction under Definition 1.

For example, if $c = [01]$, then we have that $\pi_1(c) = [0]* = \{[0], [00], [01], [000], \dots\}$ are the chains that start with $[0]$. As another example, Figure 3 shows the range of π_3.

It is important that we represent $\{c\}$ and $c*$ as distinct abstract values. In contrast, traditional k-limited analyses are parametrized by a set S of abstract values "c", where each abstract "c" represents the set of concrete chains whose longest matching prefix in S is c. With this setup, every concrete chain would map to some abstract value regardless of S (note that we must have "$[\,]$" $\in S$). Therefore, pruning would be impossible using this representation.

Extending the abstraction Given an abstraction α, it will be useful to extend the definition of α to not just concrete values, but also to abstract values, and (sets of) concrete/abstract tuples.

First, we extend α from concrete values c to abstract values s as follows:

$$\alpha(s) \triangleq \{\alpha(c) : c \in s\}, \quad s \in \mathcal{P}(\mathcal{C}). \tag{9}$$

Note that $\alpha(s)$ returns a set of abstract values. This allows us to naturally define the composition of two abstractions. In particular, given two abstractions, α and β, define their composition to be:

$$(\alpha \circ \beta)(c) \triangleq \cup_{s \in \alpha(\beta(c))} s. \tag{10}$$

Note that the composition $\alpha \circ \beta$ need not be an abstraction even if α and β are.[2] Therefore, when we compose abstractions in Section 3.2, it will be important to check that the resulting compositions are valid abstractions.

An important case in which compositions yield valid abstractions is when $\alpha \preceq \beta$ (β is finer than α). In this case, $\alpha \circ \beta = \alpha$, corresponding to the fact that applying a finer abstraction first has no impact.

Next, we extend α to concrete tuples x and sets of concrete tuples X in the natural way:

$$\alpha(x) \triangleq x.r(\alpha(x.a_1), \ldots, \alpha(x.a_{|x.\mathbf{a}|})), \tag{11}$$

$$\alpha(X) \triangleq \{\alpha(x) : x \in X\}. \tag{12}$$

Here, $\alpha(x)$ is an *abstract tuple* (one where the arguments are abstract values) and $\alpha(X)$ is a set of abstract tuples. For example:

$$\pi_1(\texttt{ext}(1, [0], [10])) = \texttt{ext}(1, [0]*, [1]*).$$

Finally, we extend α to abstract tuples b and sets of abstract tuples B:

$$\alpha(b) \triangleq \{b.r(s_1, \ldots, s_{|b.\mathbf{a}|}) : \forall i, s_i \in \alpha(b.a_i)\}, \tag{13}$$

$$\alpha(B) \triangleq \cup_{b \in B} \alpha(b). \tag{14}$$

(13) applies the abstraction function to each component of b and takes the cross product over the resulting abstract values; the result is a set of abstract tuples. (14) aggregates these sets of abstract tuples. For example:

$$\pi_1(\texttt{ext}(1, [00]*, [10]*)) = \{\texttt{ext}(1, [0]*, [1]*)\}.$$

Using the abstraction Given an abstraction α, we want to run the Datalog program to compute an abstract answer to the query. We do this by applying the abstraction to the concrete input tuples X, producing a set of abstract input tuples $\alpha(X)$. We then feed these tuples into the Datalog program to produce $\mathbf{P}(\alpha(X))$. (Note that the Datalog program is oblivious to whether the tuples are abstract or concrete.) Figure 4 shows an example of performing this computation on the graph example from Figure 1 with the k-limited abstraction π_1.

We say the query is proven by α if $\mathbf{P}(\alpha(X)) = \emptyset$. Because abstraction is sound, this happens only if the query is actually false ($\mathbf{P}(X) = \emptyset$). This fact is stated formally below (see Appendix A for the proof):

Proposition 1 (Abstraction is sound)**.** *Let α be an abstraction and let X be any set of input tuples. If $\mathbf{P}(\alpha(X)) = \emptyset$ (the query is false abstractly), then $\mathbf{P}(X) = \emptyset$ (the query is false concretely).*

If α is coarse, $\mathbf{P}(\alpha(X))$ will be imprecise; but if α is fine, $\mathbf{P}(\alpha(X))$ will be expensive to compute. The next section shows how pruning can allow us to use a fine abstraction α without incurring the full cost of computing $\mathbf{P}(\alpha(X))$.

3. General theory

We first describe the core idea behind pruning (Section 3.1) and then show how it can be used in the Prune-Refine algorithm (Section 3.2).

3.1 Pruning

Recall that the central operation of a static analysis is $\mathbf{P}$, which serves two roles: (i) determining if the query is proven (when $\mathbf{P}$ returns $\emptyset$); and (ii) returning the relevant input tuples. The following theorem provides the key equation that drives everything in this paper (see Appendix A for the proof):

Theorem 1 (Pruning is sound and complete)**.** *Let α and β be two abstractions such that $\beta \preceq \alpha$ (β is coarser than α). Then for any set of concrete input tuples X, we have:*

$$\boxed{\mathbf{P}(\alpha(X)) = \mathbf{P}(\alpha(X) \cap \alpha(\mathbf{P}(\beta(X))))}. \tag{15}$$

The left-hand side of (15) corresponds to running the analysis with respect to α. The right-hand side corresponds to first pruning the input tuples X with β and then running the analysis with α. The theorem states that the two procedures obtain identical results (the right-hand side is sound and complete with respect to the left-hand side). The significance of this is that the right-hand side is often much cheaper to compute than the left-hand side.

Let us decipher (15) a bit more. On the right-hand side, the abstract input tuples $\beta(X)$ are fed into the Datalog solver which computes $\mathbf{P}(\beta(X))$, which is the subset of input tuples, namely those that participate in any derivation of the abstract query tuple $\beta(x_{\text{Q}})$. These relevant tuples are then refined via α to yield a set of tuples which are used to prune $\alpha(X)$. The resulting subset is fed into the analysis $\mathbf{P}$. On the left-hand side, $\mathbf{P}(\alpha(X))$ is the result of directly running the analysis on the abstract tuples $\alpha(X)$ without pruning.

To obtain some intuition behind pruning, consider the following simpler idea: first run the analysis with β; if the query is proven, stop and declare *proven*; otherwise, run the analysis with α and output that answer. It is easy to see that this two-step procedure returns the same answer as just running α: Because $\beta \preceq \alpha$, if β proves the query, then so does α (Proposition 1). (15) can be thought of as an extension of this basic idea: instead of using β to just determine whether the query tuple is proven, we obtain more information, namely the whole set of input tuples that are relevant.

The complexity of an analysis is largely determined by the number of input tuples. Traditionally, the abstraction alone determines the set of input tuples and thus also the complexity of the analysis. In our case, however, the set of input tuples is pruned along the way, so the abstraction only partially determines the complexity. As we will see later, with sufficient pruning of the input tuples, we can use a very refined abstraction at a low cost.

3.2 The Prune-Refine algorithm

We now turn Theorem 1 into a full algorithm, which we call the Prune-Refine algorithm. Figure 5 shows the pseudocode of the algorithm and a diagram showing the computation of the various abstract input tuples computed.

This algorithm essentially applies (15) repeatedly. We first present a simplified version of the algorithm which ignores the pre-pruning step (we take $A_t = A'_t$). We are given a sequence of successively finer abstractions $\alpha_0, \alpha_1, \ldots$ (e.g., $\alpha_t = \pi_t$ for the k-limited abstractions) and a set of input abstract tuples A_0, which is computed under the initial abstraction α_0. Then the algorithm alternates between a pruning step and a refining step, maintaining only the abstract input tuples that could participate in a derivation of the query tuple x_{Q}. On iteration t, our current input tuples A_t are first pruned to $\tilde{A}_t$ using $\mathbf{P}$; this is subsequently refined to A_{t+1}. Figure 6 shows an example of running this algorithm on the graph example from Figure 1; the first pruning step is shown in Figure 4.

Now we discuss pre-pruning. Pre-pruning requires the user to provide another sequence of successively finer abstractions $\beta_0, \beta_1, \ldots$ which are coarser than $\alpha_0, \alpha_1, \ldots$, respectively. These abstractions will also be used to prune the input tuples. The idea is that before refining A_t to A_{t+1}, we perform two steps of pruning: (i) first we use β_t in a pre-pruning step; (ii) then we use α_t during the main pruning step.

[2] For example, suppose $\mathcal{C} = \{1, 2, 3\}$; $\alpha(1) = \alpha(2) = \{1, 2\}$, $\alpha(3) = \{3\}$; and $\beta(1) = \beta(3) = \{1, 3\}$, $\beta(2) = \{2\}$. Then $(\alpha \circ \beta)(1) = \{1, 2, 3\}$ but $(\alpha \circ \beta)(2) = \{1, 2\}$. Therefore, $\alpha \circ \beta$ is not a valid abstraction.

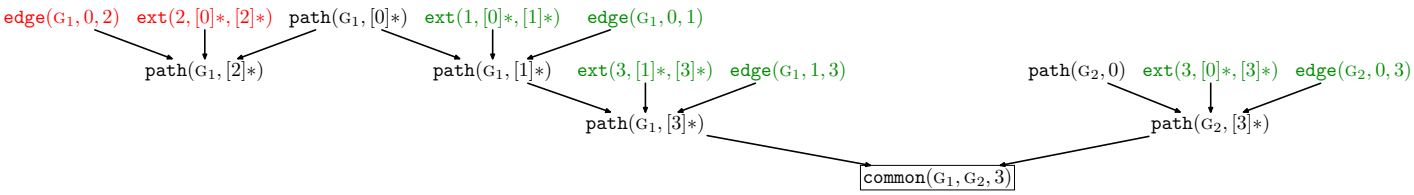


Figure 4. Computation of $\mathbf{P}(\pi_1(X))$ on the graph example from Figure 1, where X is the set of concrete input tuples, and π_1 is the 1-limited abstraction which maps each path onto the set of paths with the same first element. In the figure, each abstract tuple is derived by a rule whose source terms are connected via incoming edges. Relevant input tuples ($\mathbf{P}(\pi_1(X))$, shown in green) are the ones which are reachable by following the edges backwards; ones which are not backwards-reachable are pruned ($\pi_1(X) \backslash \mathbf{P}(\pi_1(X))$, shown in red).

Prune-Refine algorithm

Input:
Sequence of abstractions: $\alpha_0 \preceq \alpha_1 \preceq \alpha_2 \preceq \cdots$
[Auxiliary abstractions: $\beta_t \preceq \alpha_t, t = 0, 1, 2, \ldots$]
$A_0 = \alpha_0(X)$, set of tuples

For $t = 0, 1, 2, \ldots$:
[**Pre-prune**: $A'_t \leftarrow A_t \cap \alpha_t(\mathbf{P}(\beta_t(A_t)))$]
Prune: $\tilde{A}_t = \mathbf{P}(A'_t)$. If $\tilde{A}_t = \emptyset$: return *proven*.
Refine: $A_{t+1} = \alpha_{t+1}(\tilde{A}_t)$.

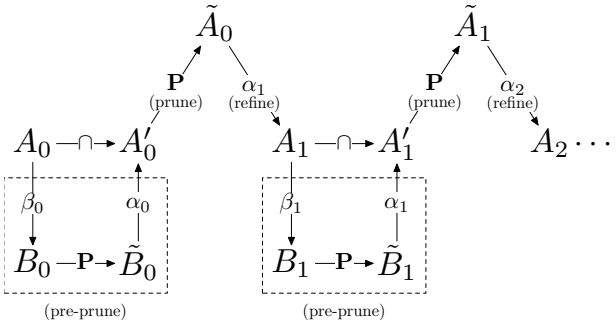


Figure 5. The pseudocode and the schema for the Prune-Refine algorithm. The algorithm maintains a set of (abstract) input tuples which could be involved in some derivation of the query x_Q and attempts to prune down this set. The basic version of the algorithm, which excludes the lines in square brackets and the dotted boxes, simply alternates between pruning and refining. The full version includes a pre-pruning step, which uses auxiliary abstractions to further reduce the number of tuples.

Pre-pruning requires a sequence of auxiliary abstractions (β_t) which are coarser than the main abstractions (α_t). A standard way to obtain auxiliary abstractions is by composing the main abstractions with another abstraction τ; formally, $\beta_t = \alpha_t \circ \tau$. We can use any τ for which $\alpha_t \circ \tau$ yields a valid abstraction, but the speedup we obtain from pre-pruning depends on the relationship between τ and α_t. If τ is the total abstraction ($\tau(c) = \mathcal{C}$), then pre-pruning will be fast but nothing will be pre-pruned, so we get no speedup. If τ is no abstraction ($\tau(c) = c$), then pre-pruning is equivalent to just running the pruning step, so we again get no speedup. A good rule of thumb is that τ should be "complementary" to α_t (we will see some examples in Section 5).

Theorem 2 states that the Prune-Refine algorithm is both sound and complete. In other words, pruning has no impact on the answer to a query. The proof is given in Appendix A and uses Theorem 1.

A_0	$\tilde{A}_0$	A_1	$\tilde{A}_1$
ext(1, [0]∗, [1]∗)	ext(1, [0]∗, [1]∗)	ext(1, {[0]}, [10]∗)	*(none)*
ext(2, [0]∗, [2]∗)	ext(3, [0]∗, [3]∗)	ext(3, {[0]}, [30]∗)	
ext(3, [0]∗, [3]∗)	ext(3, [1]∗, [3]∗)	ext(3, {[1]}, [31]∗)	
ext(3, [1]∗, [3]∗)		ext(3, [10]∗, [31]∗)	

Figure 6. The abstract input tuples computed by the Prune-Refine algorithm on the graph example from Figure 1 (without pre-pruning). We are using k-limited abstractions ($\alpha_t = \pi_t$). During the first pruning step, ext(2, [0]∗, [2]∗) is pruned from A_0, yielding $\tilde{A}_0$. In the refinement step, we expand [1]∗ to {[1]} and [10]∗. In the second pruning step, we prove the query (pruning everything).

Theorem 2 (Correctness of the Prune-Refine algorithm)**.** *At iteration t, using the incrementally pruned abstraction A_t is equivalent to using the full abstraction $\alpha_t(X)$ in that $\mathbf{P}(\alpha_t(X)) = \mathbf{P}(A_t)$. Consequently, if the algorithm returns "proven," then $\mathbf{P}(X) = \emptyset$ (the query is actually false).*

4. k-limited Pointer Analysis

We now introduce our k-object-sensitive pointer analysis [12], on which we will apply the Prune-Refine algorithm. 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.[3] Figure 7 describes the Datalog program corresponding to this analysis.

The analysis represents both contexts and abstract objects using chains of allocation sites ($\mathbb{C} = \mathbb{H}^*$). Contexts are extended into new contexts via the ext relation, which prepends an allocation site to a chain (e.g., ext(3, [12], [312])). Note that these chains are not truncated in the Datalog program, and therefore, running the Datalog program directly (ignoring the fact that it might not terminate) corresponds to performing an ∞-object-sensitivity analysis.

Although this Datalog program itself is an approximation to the concrete program semantics—it is flow-insensitive, does not handle primitive data, etc., we will informally say that a client query computed with respect to this analysis yields a concrete answer. In contrast, we obtain an abstract answer by computing the client query with respect to a k-limited abstraction, which we will discuss in Section 4.2.

We now briefly describe the Datalog rules in Figure 7. Rule (1) states that the main method m_{main} is reachable in a distinguished context [ ]. Rule (2) states that a target method of a reachable call

[3] Our actual implementation is a straightforward extension of this simplified analysis which handles multiple arguments, return values, class initializers, and objects allocated through reflection.

Domains:

$$
\begin{array}{rrcl}
\text{(method)} & m & \in & \mathbb{M} = \{m_{\text{main}}, ...\} \\
\text{(local variable)} & v & \in & \mathbb{V} \\
\text{(global variable)} & g & \in & \mathbb{G} \\
\text{(object field)} & f & \in & \mathbb{F} \\
\text{(method call site)} & i & \in & \mathbb{I} \\
\text{(allocation site)} & h & \in & \mathbb{H} \\
\text{(statement)} & p & \in & \mathbb{P} \\
\text{(method context)} & c & \in & \mathbb{C} = \mathbb{H}^* \\
\text{(abstract object)} & o & \in & \mathbb{O} = \mathbb{H}^*
\end{array}
$$

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

Input relations:

`body`	$\subset$	$\mathbb{M} \times \mathbb{P}$	(method contains statement)
`trgt`	$\subset$	$\mathbb{I} \times \mathbb{M}$	(call site resolves to method)
`argI`	$\subset$	$\mathbb{I} \times \mathbb{V}$	(call site's argument variable)
`argM`	$\subset$	$\mathbb{M} \times \mathbb{V}$	(method's formal argument variable)
`ext`	$\subset$	$\mathbb{H} \times \mathbb{C} \times \mathbb{C}$	(extend context with site)
	$=$	$\{(h, c, [h] + c) : h \in \mathbb{H}, c \in \mathbb{C}\}$	

Output relations:

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

Rules:

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

Figure 7. Datalog implementation of our k-object-sensitivity pointer 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 $\mathbf{a}_s$.

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. Rule (10) 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 (11) sets the points-to set of the `this` argument of method m in context o to the singleton $\{o\}$.

This pointer 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`).

4.1 Clients

The core pointer analysis just described is used by three clients, which each defines a set of queries.

Monomorphic call site detection Monomorphic call sites are dynamically dispatched call sites with at most one target method. These can be transformed into statically dispatched ones which are cheaper to execute. For each call site $i \in \mathbb{I}$ whose target is a virtual method, we create a query $\texttt{poly}(i)$ asking whether i is polymorphic. This query can be computed with the following rule:

$$
\texttt{poly}(i) \Leftarrow \texttt{cg}(*, i, *, m_1), \texttt{cg}(*, i, *, m_2), m_1 \neq m_2. \quad (16)
$$

Downcast safety checking A safe downcast is one that cannot fail because the object to which the downcast is applied is guaranteed to be a subtype of the target type. Therefore, safe downcasts obviate the need for run-time cast checking. We create a query for each downcast—statement of the form $v_1 = v_2$ where the declared type of v_2 is not a subtype of the declared type of v_1. The query can be computed with the following rule:

$$
\begin{aligned}
\texttt{unsafe}(v_1, v_2) \Leftarrow\ & \texttt{ptsV}(*, v_2, o), \texttt{typeO}(o, t_2), \texttt{typeV}(v_1, t_1), \\
& \neg\texttt{subtype}(t_1, t_2). \quad (17)
\end{aligned}
$$

Here, `typeV` is a relation on a variable and its declared type and `typeO` is a relation on an abstract object and its type (computed by inspecting the initial allocation site of o).

Race detection In race detection, each query consists of a pair of 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 pointer analysis.

4.2 Relationship to general notation

We now describe the k-object-sensitive pointer analysis (Figure 7) in terms of our general notation presented in Section 2. The set of concrete values $\mathcal{C}$ is the union of all the domains (e.g.., allocation sites $\mathbb{H} = \{1, 2, 3, \dots\}$, abstract objects $\mathbb{C} = \mathbb{H}^*$, etc.). The input tuples X are specified by the input relations (e.g., $X = \{\texttt{body}(m_{\text{main}}, x = \texttt{new}\, 3), \texttt{ext}(3, [12], [312])\}$). Each of the three clients defines a set of possible query tuples, for example, $x_{\text{Q}} = \texttt{unsafe}(\texttt{v4}, \texttt{v8})$ for downcast safety checking of an assignment $\texttt{v4} = \texttt{v8}$.

Recall that $\mathbf{P}(X)$ corresponds to obtaining an answer to a client query with respect to ∞-object-sensitivity. To obtain k-object-

```
class A {
  f() {
0:  v = new A
    if (*) return v
    else return v.f()
  }
}
```

```
1:  x1 = new A
2:  x2 = new A
    y1 = x1.f()
    y2 = x2.f()
```

Figure 8. An example illustrating the repetition of allocation sites. The points-to set of `y1` using ∞-object-sensitivity is $\{[01], [001], [0001], \dots\}$ (any positive number of zeros followed by a 1), and the points-to set of `y2` is $\{[02], [002], [0002], \dots\}$. While these two sets are disjoint, if we use a k-limited abstraction for any finite k, we would conclude erroneously that both variables might point to $\mathbf{0}_k*$, where $\mathbf{0}_k$ is a chain of k zeros. Incidentally, this demonstrates an intrinsic limitation of the k-object-sensitivity abstraction. Using the barely-repeating k-limited abstraction, we can increase k while avoiding chains longer than [00] since [00] is barely-repeating. This results in computational savings, and in this case, in no loss in precision.

sensitivity, we first apply the k-limited abstraction π_k to the input tuples and run the Datalog program on these abstract tuples ($\mathbf{P}(\pi_k(X))$). Note that only the `ext` tuples are affected by the abstraction.

5. Abstractions

We have already defined the k-limited abstraction, which corresponds to k-object-sensitivity. We now present two orthogonal variants of this basic abstraction: one that additionally limits the repetition of allocation sites (Section 5.1) and one that further abstracts allocation sites using type information (Section 5.2).

5.1 Barely-repeating k-limited abstraction

When we applied the k-limited abstraction in practice, we noticed empirically that a major reason why it did not scale was the seemingly unnecessary combinatorial explosion associated with chains formed by cycling endlessly through the same allocation sites. For k-CFA, this repetition corresponds to recursion. For k-object-sensitivity, this corresponds to recursive allocation, as illustrated in Figure 8.[4] We therefore wish to define an abstraction that not only truncates chains at length k but also truncates a chain when it starts repeating.

For a sequence c, we say c is *non-repeating* if all its elements are distinct. We say c is *barely-repeating* if (i) c excluding the last element ($c[1..|c|-1]$) is *non-repeating* and (ii) the last element of c is repeated earlier in c. Let $\delta(c)$ be the length of the longest prefix of c that is barely-repeating, if it exists, and ∞ otherwise:

$$\delta(c) \triangleq \begin{cases} \max_{m' : c[1..m'] \text{ is barely-repeating}} m' & \text{if } m' \text{ exists,} \\ \infty & \text{otherwise.} \end{cases} \quad (18)$$

For example, $\delta([10010]) = 3$ because [100] is barely-repeating, but [1001] is not.

We now define the *barely-repeating k-limited abstraction* $\hat{\pi}_k$ as follows:

$$\hat{\pi}_k(c) \triangleq \pi_{\min\{k, \delta(c)\}}(c), \quad (19)$$

Figure 9 shows an example of $\hat{\pi}_k$. We show that $\hat{\pi}_k$ is a valid abstraction:

[4] Incidentally, the example in the figure also gives an interesting example where k-object-sensitivity for any finite k (no matter how large) is less precise than ∞-object-sensitivity.

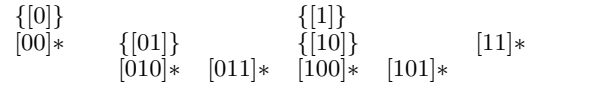


Figure 9. For the barely-repeating k-limited abstraction for $\mathbb{H} = \{0, 1\}$ and $k = 3$, we show the equivalence classes under $\hat{\pi}_k$. Compare this with the classes for the k-limited abstraction (Figure 3). Note that, for example, $[000]*$ and $[001]*$ are collapsed into $[00]*$ since [000] and [001] are not barely-repeating, but [00] is.

Proposition 2. *The function $\hat{\pi}_k$ defined in (19) is a valid abstraction (Definition 1).*

Proof. We consider two cases: (i) for $\{c\} \in \text{range}(\hat{\pi}_k)$, we have $\hat{\pi}_k(c) = \{c\}$; and (ii) for any $c* \in \text{range}(\hat{\pi}_k)$, either $|c| = k$ or c is barely-repeating; in either case, it is easy to see that any extension $c' \in c*$ will have $\hat{\pi}_k(c') = c*$. □

Remark: one might wonder why we defined the abstraction using the barely-repeating criterion as opposed to the simpler non-repeating criterion. It turns out that using the latter in (18) would not result in a valid abstraction. If $\hat{\pi}_k$ were defined using the non-repeating criterion, then $\hat{\pi}_3([00]) = [0]*$. But for $[01] \in [0]*$, we have $\hat{\pi}_3([01]) = \{[01]\} \neq [0]*$.

5.2 Type-based abstraction

We now introduce an abstraction that we will use in the pre-pruning step of the Prune-Refine algorithm. We start by defining an equivalence relation over allocation sites $\mathbb{H}$, represented by a function $\tau : \mathbb{H} \mapsto \mathcal{P}(\mathbb{H})$ mapping each allocation site $h \in \mathbb{H}$ to its equivalence class. In the graph example, we might have $\tau(0) = \tau(1) = \{0, 1\}$ and $\tau(2) = \tau(3) = \{2, 3\}$.

Given such a τ, we extend it to sequences by taking the cross product over elementwise applications:

$$\tau(c) = \tau(c[1]) \times \cdots \times \tau(c[|c|]), \quad c \in \mathbb{H}^*. \quad (20)$$

In the running example, $\tau([02]) = \{[02], [03], [12], [13]\}$.

To construct τ for k-limited pointer analysis, we consider using two sources of type information associated with an allocation site, motivated by [17]:

$$\text{I}(h) = \text{declaring type of allocation site } h \quad (21)$$

$$\text{C}(h) = \text{type of class containing allocation site } h \quad (22)$$

Using these two functions, we can construct three equivalence relations, τ_I, τ_C, and $\tau_{\text{I}\times\text{C}}$ as follows:

$$\tau_f(h) = \{h' : f(h) = f(h')\}, \quad (23)$$

for $f \in \{\text{I}, \text{C}, \text{I} \times \text{C}\}$.

Now we have three choices for τ: one that uses the declaring type (τ_I), one that uses the type of the containing class (τ_C), and one that uses both ($\tau_{\text{I}\times\text{C}}$). Recall that all three are complementary to the k-limited abstraction π_k: Specifically, τ abstracts a chain by abstracting each site uniformly, whereas π_k performs no abstraction on the first k sites, but performs a total abstraction on the rest of the chain. Recall that this complementarity is desirable for effective pre-pruning.

Since τ is not coarser than π_k, we cannot use it directly in the Prune-Refine algorithm. We must compose τ with π_k or $\hat{\pi}_k$ to yield another abstraction which is coarser than π_k or $\hat{\pi}_k$, respectively. But in what order should we compose? We must be careful because the composition of two abstractions is not necessarily an abstraction. Fortunately, the following proposition shows which compositions are valid:

Proposition 3. *The functions (i) $\pi_k \circ \tau$ and (ii) $\tau \circ \pi_k$ are valid abstractions (see Definition 1) and equivalent; (iii) $\hat{\pi}_k \circ \tau$ is also valid, but (iv) $\tau \circ \hat{\pi}_k$ is not.*

Proof. For each of these four composed functions, each set s in the range of the function must be either of the form $s = w_1 \times \cdots \times w_m$ for $m < k$ (case 1) or $s = w_1 \times \cdots \times w_m \times \mathbb{H}^*$ for some $m \leq k$ (case 2), where $w_i \in \text{range}(\tau)$ for each $i = 1, \ldots, m$.

For (i) and (ii), it is straightforward to check that $(\pi_k \circ \tau)(c) = (\tau \circ \pi_k)(c) = s$ for each $c \in s$. Intuitively, the truncation (π_k) and coarsening (τ) operate independently and can be interchanged.

For (iii) and (iv), the two dimensions do not act independently; the amount of truncation depends on the amount of coarsening: the coarser τ is, the more truncation one might need to limit repetitions. Showing that $\hat{\pi}_k \circ \tau$ is valid proceeds in a similar manner to Proposition 2. If s falls under case 1, note that no $c \in s$ is repeating because the w_i's must be disjoint; therefore $\hat{\pi}_k(\tau(c)) = s$. If s falls under case 2, note that for any $c[1..m] \in s$ must be barely-repeating but any longer prefix is not, and therefore, $\hat{\pi}_k(\tau(c)) = s$.

To show that (iv) is not an abstraction, consider the following counterexample: let $\mathbb{H} = \{0, 1, 2\}$, and define $\tau(h) = \mathbb{H}$ for all $h \in \mathbb{H}$ (there is one equivalence class). Consider applying $\tau \circ \hat{\pi}_3$ to two elements $[01]$ and $[00]$: For $[01]$, we have $\hat{\pi}_3([01]) = \{[01]\}$, so $\tau(\hat{\pi}_3([01])) = \mathbb{H}^2$; for $[00]$, we have $\hat{\pi}_3([00]) = [00]*$, so $\tau(\hat{\pi}_3([00])) = \mathbb{H}^2 \times \mathbb{H}^*$. But $\mathbb{H}^2 \subsetneq \mathbb{H}^2 \times \mathbb{H}^*$ (notably, the two sets are neither equal nor disjoint), so $\tau \circ \hat{\pi}_3$ does not define a valid abstraction. □

In light of this result, we will use the valid abstractions $\pi_k \circ \tau$ and $\hat{\pi}_k \circ \tau$, which work by first applying the type-based abstraction τ and then applying π_k or $\hat{\pi}_k$.

6. Experiments

In this section, we apply the Prune-Refine algorithm (Section 3.2) to k-object-sensitivity for our three clients (Section 4.1): downcast safety checking (DOWNCAST), monomorphic call site inference (MONOSITE), and race detection (RACE). Our main empirical result is that across different clients and benchmarks, pruning is effective at curbing the exponential growth, which allows us to run analyses using abstractions finer than what is possible without pruning.

6.1 Setup

Our experiments were performed using IBM J9VM 1.6.0 on 64-bit Linux machines. All analyses were implemented in Chord, an extensible program analysis framework for Java bytecode,[5] which uses the BDD Datalog solver `bddbddb` [21]. We evaluated our analyses on five Java benchmarks shown in Table 1. In each run, we allocated 8GB of memory and terminated the process when it ran out of memory.

We experimented with various combinations of abstractions and refinement algorithms (see Table 2). As a baseline, we consider running an analysis with a full abstraction α (denoted $\text{FULL}(\alpha)$). For α, we can either use k-limited abstractions ($\boldsymbol{\pi} = (\pi_k)_{k=0}^{\infty}$), in which case we recover ordinary k-object-sensitivity, or the barely-repeating variants ($\hat{\boldsymbol{\pi}} = (\hat{\pi}_k)_{k=0}^{\infty}$). We also consider the site-based refinement algorithm of [10], which considers a sequence of abstractions $\boldsymbol{\alpha} = (\alpha_0, \alpha_1, \ldots)$ but stops refining sites which have been deemed irrelevant. This algorithm is denoted $\text{SITE}(\boldsymbol{\alpha})$.

As for the new algorithms that we propose in this paper, we have $\text{PR}(\boldsymbol{\alpha})$, which corresponds to the Prune-Refine (PR) algorithm using a sequence of abstractions $\boldsymbol{\alpha}$ with no pre-pruning; and $\text{PR}(\boldsymbol{\alpha}, \tau)$, which performs pre-pruning using $\beta_t = \alpha_t \circ \tau$ for $t = 0, 1, 2, \ldots$. We consider three choices of τ which use different kinds of type information ($\tau_{\text{I}}, \tau_{\text{C}}, \tau_{\text{I}\times\text{C}}$).

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

Abstractions	
$\boldsymbol{\pi} = (\pi_k)_{k=0}^{\infty}$	(k-limited abstractions (8))
$\hat{\boldsymbol{\pi}} = (\hat{\pi}_k)_{k=0}^{\infty}$	(barely-repeating k-limited abstractions (19))
τ_{I}	(abstraction using type of allocation site)
τ_{C}	(abstraction using type of containing class)
$\tau_{\text{I}\times\text{C}}$	(abstraction using both types)
Algorithms	
$\text{FULL}(\alpha)$	(standard analysis using an abstraction α)
$\text{SITE}(\boldsymbol{\alpha})$	(site-based refinement [10] on abstractions $\boldsymbol{\alpha}$)
$\text{PR}(\boldsymbol{\alpha})$	(PR algorithm using $\boldsymbol{\alpha}$, no pre-pruning)
$\text{PR}(\boldsymbol{\alpha}, \tau)$	(PR algorithm using $\boldsymbol{\alpha}$, using $\boldsymbol{\alpha} \circ \tau$ to pre-prune)

Table 2. Shows the abstractions and algorithms that we evaluated empirically. For example, $\text{PR}(\hat{\boldsymbol{\pi}}, \tau_{\text{I}\times\text{C}})$ means running the Prune-Refine algorithm on the barely-repeating k-limited abstraction ($\alpha_k = \hat{\pi}_k$), using a composed abstraction based on the type of an allocation site (τ_{I}) and the type of the declaring class (τ_{C}) to do pre-pruning (specifically, $\beta_k = \hat{\pi}_k \circ \tau_{\text{I}\times\text{C}}$).

In our implementation of the Prune-Refine algorithm, we depart slightly from our presentation. Instead of maintaining the full set of relevant input tuples, we instead maintain only the set of allocation site chains which exist in some relevant input tuple. This choice results in more conservative pruning, but reduces the amount of information that we have to keep. We can modify the original Datalog program so that the original Prune-Refine algorithm computes this new variant: Specifically, first introduce new input tuples $\texttt{active}(c)$ for each $c \in \mathbb{H}^*$. Then encode existing `ext` input tuples as rules with no source terms; `ext` is no longer an input relation. Finally, add $\texttt{active}(c)$ to the right-hand side of each existing rule that uses a chain-valued variable. Computing the relevant input tuples in this modified Datalog program corresponds exactly to computing the set of relevant allocation site chains.

6.2 Results

We ran the four algorithms of Table 2 using k-limited abstractions, seeing how far we could increase k until the analyses ran out of memory. For each analysis, we also measured the number of input tuples given to the Datalog solver; this quantity is denoted as $|A'_t|$ (see Figure 5). In this section, the number of iterations t is the same as the k value.

Figure 10 plots the number of tuples $|A'_t|$ as a function of number of iterations t. We see that the non-pruning algorithms completely hit a wall after a few iterations, with the number of tuples exploding exponentially. On most benchmark-client pairs, the pruning algorithms are able to continue increasing k much further, though on several pairs, pruning only manages to increase k by one beyond the non-pruning algorithms. We also observed that pruning does yield speedups, although these are less pronounced than the differences in the number of tuples. Nonetheless, pruning overcomes the major bottleneck—that standard k-limited analyses run out of memory even for moderate k. By curbing the growth of the number of tuples, pruning makes it possible to run some analyses at all.

However, there are several caveats with pruning: First, we are using BDDs, which can actually handle large numbers of tuples so long as they are structured; pruning destroys some of this structure, yielding less predictable running times. Second, pruning requires solving the transformed Datalog program for computing $\mathbf{P}(X)$, which is more expensive than the original Datalog program. Finally, we must solve the Datalog program several times, not just

	description	# classes	# methods	# bytecodes	$\lvert\mathbb{H}\rvert$
`elevator`	discrete event simulation program	154	629	39K	637
`hedc`	web crawler	309	1,885	151K	1,494
`weblech`	website downloading and mirroring tool	532	3,130	230K	2,545
`lusearch`	text indexing and search tool	611	3,789	267K	2,822
`avrora`	simulation and analysis framework for AVR microcontrollers	1,498	5,892	312K	4,823

Table 1. Benchmark characteristics: the number of classes, number of methods, total number of bytecodes in these methods, and number of allocation sites ($|\mathbb{H}|$) deemed reachable by 0-CFA.

	$\frac{\lvert B_t\rvert}{\lvert A_t\rvert}$	$\frac{\lvert \tilde{B}_t\rvert}{\lvert B_t\rvert}$	$\frac{\lvert A'_t\rvert}{\lvert A_t\rvert}$	$\frac{\lvert \tilde{A}_t\rvert}{\lvert A_t\rvert}$	$\frac{\lvert A_{t+1}\rvert}{\lvert A_t\rvert}$
DOWNCAST/`hedc`	0.28	0.72	0.68	0.65	1.63
DOWNCAST/`weblech`	0.19	0.18	0.26	0.19	3.28
DOWNCAST/`lusearch`	0.17	0.04	0.03	0.02	1.89
DOWNCAST/`avrora`	0.21	0.03	0.05	0.03	1.57
MONOSITE/`elevator`	0.10	0.55	0.21	0.21	1.67
MONOSITE/`hedc`	0.22	0.30	0.36	0.29	3.78
MONOSITE/`weblech`	0.19	0.18	0.25	0.18	3.33
MONOSITE/`lusearch`	0.26	0.10	0.15	0.12	3.39
MONOSITE/`avrora`	0.30	0.05	0.04	0.03	1.85
RACE/`elevator`	0.10	0.57	0.22	0.21	1.58
RACE/`hedc`	0.28	0.28	0.34	0.25	4.01
RACE/`weblech`	0.19	0.18	0.27	0.18	3.43
RACE/`lusearch`	0.30	0.15	0.18	0.14	3.96
RACE/`avrora`	0.38	0.08	0.08	0.06	2.71
Average	0.23	0.24	0.22	0.18	2.72

Table 3. Shows the shrinking and growth of the number of tuples during the various pruning and refinement operations (see Figure 5) for our best algorithm $\text{PR}(\boldsymbol{\pi}, \tau_{\text{I}\times\text{C}})$ across all the clients and benchmarks, averaged across iterations. The columns are as follows: First, $\frac{|B_t|}{|A_t|}$ measures the number of tuples after projecting down to the auxiliary abstraction $\beta_t = \pi_t \circ \tau_{\text{I}\times\text{C}}$ for pre-pruning; note that running the analysis using types instead of allocation sites is much cheaper. Next, $\frac{|\tilde{B}_t|}{|B_t|}$ shows the fraction of abstract values kept during pre-pruning; When we return from types to allocation sites, we see that the effect of pre-pruning carries over ($\frac{|A'_t|}{|A_t|}$). Next, pruning kept $\frac{|\tilde{A}_t|}{|A_t|}$ of the chains. Finally, $\frac{|A_{t+1}|}{|A_t|}$ measures the ratio between iterations, which includes both pruning and refinement. Note that there is still almost a three-fold growth of the number of tuples (on average), but this growth would have been much more unmanageable without the pruning.

once. These three caveats also apply to the site-based refinement algorithm of [10] (SITE), so pruning is at least a strict improvement over that algorithm.

We found that the best instantiation of the Prune-Refine algorithm is $\text{PR}(\boldsymbol{\pi}, \tau_{\text{I}\times\text{C}})$, which involves pre-pruning with both kinds of type information ($\tau_{\text{I}\times\text{C}}$); this works better than both no pre-pruning and pre-pruning with only τ_{I} or τ_{C} alone.

Table 3 provides more details on the quantitative impact of pruning for $\text{PR}(\boldsymbol{\pi}, \tau_{\text{I}\times\text{C}})$. We see that pre-pruning has a significant impact: we can eliminate about three-quarters of the tuples by just operating on the coarser level of types rather than allocation sites (see the $\frac{|\tilde{B}_t|}{|B_t|}$ column). Importantly, the effect of this pruning carries over to the original k-limited abstraction (see the $\frac{|A'_t|}{|A_t|}$ column).

So far, we have been using the k-limited abstraction; we now compare this abstraction with the barely-repeating k-limited abstraction introduced in Section 5.1. As Figure 11 shows, for a few cases, the barely-repeating k-limited abstraction requires fewer tuples than the k-limited abstraction; but in most cases, it does not improve scalability. The reason is that the barely-repeating abstraction curbs refinement, but often, somewhat paradoxically, it is exactly the refinement which enables more pruning.

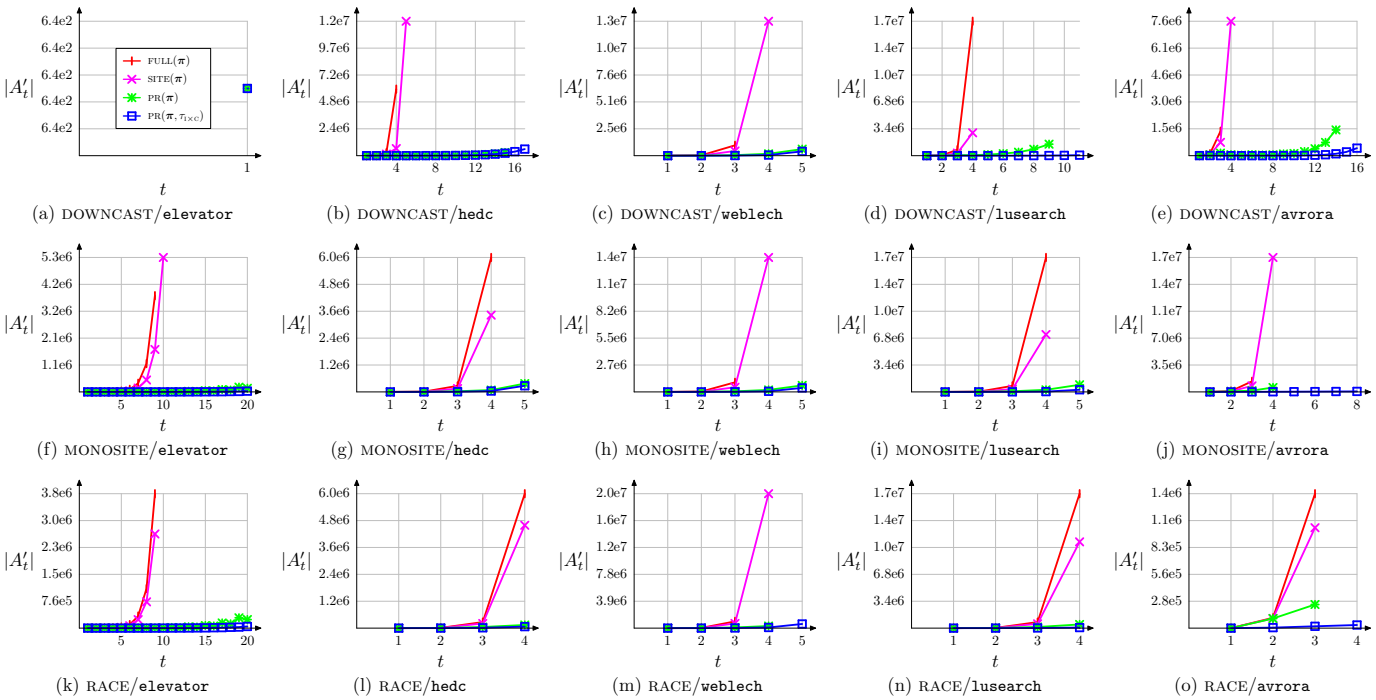
client/benchmark \ k	1	2	3	4	5
DOWNCAST/`elevator`	0	-	-	-	-
DOWNCAST/`hedc`	10	8	3	**2**	**2**
DOWNCAST/`weblech`	24	14	6	**6**	-
DOWNCAST/`lusearch`	36	14	6	**5**	**5**
DOWNCAST/`avrora`	12	10	6	**6**	**6**
MONOSITE/`elevator`	1	1	1	1	1
MONOSITE/`hedc`	164	149	149	**149**	-
MONOSITE/`weblech`	273	258	252	**252**	-
MONOSITE/`lusearch`	593	454	447	**447**	-
MONOSITE/`avrora`	288	278	272	-	-
RACE/`elevator`	475	440	437	437	437
RACE/`hedc`	23,033	22,043	21,966	-	-
RACE/`weblech`	7,286	4,742	4,669	-	-
RACE/`lusearch`	33,845	23,509	16,957	-	-
RACE/`avrora`	62,060	61,807	61,734	-	-

Table 4. The number of unproven queries (unsafe downcasts, polymorphic sites, races) for each of the clients and benchmarks over the first five iterations. All analyses obtain the exact results on iterations where they obtain an answer. Bolded numbers refer to k values reached by $\text{PR}(\boldsymbol{\pi}, \tau_{\text{I}\times\text{C}})$ but not by any non-pruning algorithm. While pruning enables to increase k more, we get strictly more precise results for only two of the client/benchmark pairs (DOWNCAST/`hedc` and DOWNCAST/`lusearch`). This points out inherent limitations of this family of k-limited abstractions.

Finally, Table 4 shows the effect on the number of queries proven. While pruning enables us to increase k much more than before, it turns out that our particular analyses for these clients saturate quite quickly, so over all the clients and benchmarks, we were only able to prove two queries more than using the non-pruning techniques. On the surface, these findings seem to contradict [9], which showed a sharp increase in precision around $k = 4$ for k-CFA. However, this discrepancy merely suggests that our flow-insensitive analyses are simply limited: since [9] offers upper bounds on precision, we know for sure that low k values are insufficient; the fact that we don't see an increase in precision for higher k suggests that the non-k-related aspects of our analyses are insufficient. Given that our pruning approach is general, it would be interesting to tackle other aspects of program analysis such as flow-sensitivity.

7. Related Work

There is a wealth of literature which attempts to scale static analyses without sacrificing precision. One general theme is to work with a flexible family of abstractions, which in principle allows us to conform to the needs of the client. Milanova et al. [12, 13] consider abstractions where each local variable can be independently treated context-sensitively or context-insensitively, and different k values can be chosen for different allocation sites. Lhoták and Hendren [7, 8] present Paddle, a parametrized framework for BDD-based, k-limited pointer analyses. [17] scale up k-object-

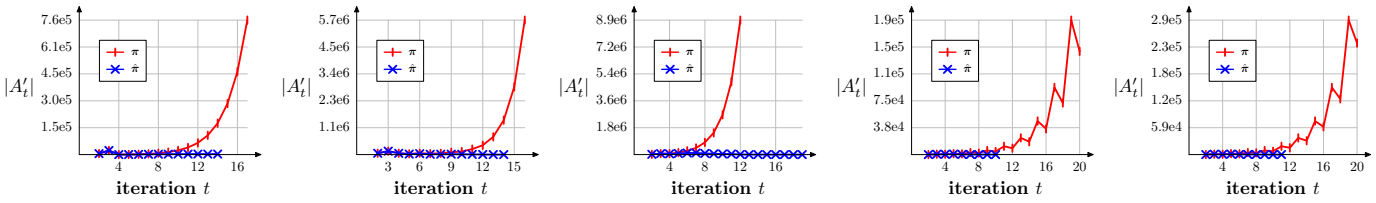


(a) DOWNCAST/elevator (b) DOWNCAST/hedc (c) DOWNCAST/weblech (d) DOWNCAST/lusearch (e) DOWNCAST/avrora

(f) MONOSITE/elevator (g) MONOSITE/hedc (h) MONOSITE/weblech (i) MONOSITE/lusearch (j) MONOSITE/avrora

(k) RACE/elevator (l) RACE/hedc (m) RACE/weblech (n) RACE/lusearch (o) RACE/avrora

Figure 10. For each client/benchmark pair, we show the growth of the number of input tuples $|A'_t|$ across iterations (recall that A'_t are the tuples fed into the Datalog program). Table 2 describes the four algorithms. We see that the pruning algorithms (PR($\boldsymbol{\pi}$) and PR($\boldsymbol{\pi}, \tau_{1\times C}$)) drastically cut down the number of input tuples by many orders of magnitude.



(a) DOWNCAST/hedc (b) DOWNCAST/avrora (c) DOWNCAST/lusearch (d) MONOSITE/elevator (e) RACE/elevator

Figure 11. Shows the 5 (out of the 15) client/benchmark pairs for which using the barely-repeating k-limited abstraction ($\hat{\boldsymbol{\pi}}$) allows one to increase k much more than the plain k-limited abstraction ($\boldsymbol{\pi}$). On the other 10 client/benchmarks where k cannot get very large, limiting repetitions is actually slightly worse in terms of scalability. Also note that three of the plots for $\hat{\boldsymbol{\pi}}$ stop early, not because the algorithm runs out of memory, but because the algorithm has actually converged and increasing k would have no effect.

sensitivity, increasing k by one using types rather than allocation sites. However, in all of this work, which parts of the abstraction should be more refined is largely left up to the user.

Client-driven approaches use feedback from a client query to determine what parts of an abstraction to refine. Plevyak and Chien [15] use a refinement-based algorithm for type inference, where context-sensitivity is driven by detecting type conflicts. Guyer and Lin [4] present a pointer analysis for C which detects loss of precision (e.g., at merge points) and introduce context-sensitivity. Our method for determining relevant input tuples is similar in spirit but more general.

Section 3.1 of Liang et al. [10] (not the focus of that work) also computes the set of relevant tuples by running a transformed Datalog program. However, what is done with this information is quite different there. [10] merely stops refining the irrelevant sites whereas we actually prune all irrelevant tuples, thereby exploiting this information more fully. As we saw in Section 6, this difference had major ramifications.

Demand-driven analyses [5, 23] do not refine the abstraction but rather try to compute an analysis on an existing abstraction more efficiently. Sridharan et al. [19] presents an algorithm which casts pointer analysis as a CFL-reachability problem and relaxes the problem by introducing additional "match" edges.

Our Prune-Refine algorithm has a client-driven flavor in that we refine our abstraction, but also a demand-driven flavor in that we do not perform a full computation (in particular, ignoring tuples which were pruned). However, there are two important differences between the present work and the work described earlier: First, while most of that work is specific to pointer analysis, our Prune-Refine algorithm is applicable to any Datalog program. Second, past work is based on selected refinement, which is orthogonal to pruning. Selected refinement merely governs the abstractions $(\alpha_t)_{t=0}^{\infty}$ that we use, whereas pruning focuses on removing input

tuples. Given that the input tuples encode the program analysis, pruning is analogous to program slicing.

Other forms of pruning have been implemented in various settings. [20] uses dynamic analysis to prune down the set of paths and then focuses a static analysis on these paths. [18] uses pruning for type inference in functional languages, where pruning is simply a heuristic which shortcuts a search algorithm. As a result, pruning can hurt precision. One advantage of our pruning approach is that it comes with strong soundness and completeness guarantees.

8. Conclusion

We have introduced pruning as a general technique for scaling up static analyses written in Datalog. The basic idea is to run an analysis using a coarse abstraction, only keeping input tuples deemed relevant, and then using a finer abstraction on the remaining tuples. Theoretically, we showed that pruning is both sound and complete (our analysis is valid and we lose no precision). Empirically, we showed that pruning enables us to scale up analyses based on k-object-sensitivity much more than previous approaches.

Acknowledgments

We thank Mooly Sagiv and Hongseok Yang for discussion and useful feedback. We also thank the anonymous reviewers for their insightful comments.

References

[1] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, pages 203–213, 2001.

[2] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, pages 243–262, 2009.

[3] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. *Computer Aided Verification*, 1254:72–83, 1997.

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

[5] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *PLDI*, pages 24–34, 2001.

[6] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

[7] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In *CC*, pages 47–64, 2006.

[8] 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.

[9] P. Liang, O. Tripp, M. Naik, and M. Sagiv. A dynamic evaluation of static heap abstractions. In *OOPSLA*, pages 411–427, 2010.

[10] P. Liang, O. Tripp, and M. Naik. Learning minimal abstractions. In *POPL*, 2011.

[11] K. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.

[12] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA*, 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 *PLDI*, pages 308–319, 2006.

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

[16] O. Shivers. Control-flow analysis in Scheme. In *PLDI*, pages 164–174, 1988.

[17] Y. Smaragdakis, M. Bravenboer, and O. Lhotak. Pick your contexts well: Understanding object-sensitivity. In *POPL*, 2011.

[18] S. A. Spoon and O. Shivers. Demand-driven type inference with subgoal pruning: Trading precision for scalability. In *ECOOP*, 2004.

[19] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.

[20] V. Vipindeep and P. Jalote. Efficient static analysis with path pruning using coverage data. In *International Workshop on Dynamic Analysis (WODA)*, 2005.

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

[22] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, 2004.

[23] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *POPL*, pages 197–208, 1998.

A. Proofs

Instead of directly proving Proposition 1, we state a more general theorem which will be useful later:

Theorem 3 (Soundness)**.** *Let α and β be two abstractions with $\beta \preceq \alpha$ (β is coarser), and let X be any set of input tuples. For any derivation $\mathbf{a} \in \mathbf{D}(\alpha(X))$, define $\mathbf{b} = (b_1, \ldots, b_{|\mathbf{a}|})$ where each b_i is the unique element in $\beta(a_i)$. Then $\mathbf{b} \in \mathbf{D}(\beta(X))$.*

Proof of Theorem 3. Define $A = \alpha(X)$ and $B = \beta(X)$. Consider $\mathbf{a} \in \mathbf{D}(A)$ and let $\mathbf{b}$ be as defined in the theorem. For each position i, we have two cases. First, if $a_i \in A$, then $b_i \in (\beta \circ \alpha)(X) = \beta(X) = B$. Otherwise, let $z \in \mathcal{Z}$ be the rule and J be the indices of the tuples used to derive a_i. The same rule z and the corresponding tuples $\{b_j : j \in J\}$ can also be used to derive b_i. Therefore, $\mathbf{b} \in \mathbf{D}(B)$. □

Proof of Proposition 1 (abstraction is sound). Apply Theorem 3 with $\beta = \alpha$ and α as the identity function (no abstraction). □

Before we prove Theorem 1, we state a useful lemma.

Lemma 1 (Pruning is idempotent)**.** *For any set of tuples (concrete or abstract) X, $\mathbf{P}(X) = \mathbf{P}(\mathbf{P}(X))$.*

Proof. Since $\mathbf{P}(X) \subset X$ by definition and $\mathbf{P}$ is monotonic, we have $\mathbf{P}(\mathbf{P}(X)) \subset \mathbf{P}(X)$. For the other direction, let $x \in \mathbf{P}(X)$. Then x is part of some derivation ($x \in \mathbf{x} \in \mathbf{D}(X)$). All the input tuples of $\mathbf{x}$ (those in $\mathbf{x} \cap X$) are also in $\mathbf{P}(X)$, so $\mathbf{x} \in \mathbf{D}(\mathbf{P}(X))$. Therefore $x \in \mathbf{P}(\mathbf{P}(X))$. □

Proof of Theorem 1 (pruning is sound and complete). We define variables for the intermediate quantities in (15): $A = \alpha(X)$ and $B = \beta(X)$, $\tilde{B} = \mathbf{P}(B)$, $\tilde{A} = \alpha(\tilde{B})$, and $A' = A \cap \tilde{A}$. We want to show that pruning is sound ($\mathbf{P}(A) \subset \mathbf{P}(A')$) and complete ($\mathbf{P}(A) \supset \mathbf{P}(A')$). Completeness follows directly because $A \supset A'$ and $\mathbf{P}$ is monotonic (increasing the number of input tuples can only increase the number of derived tuples).

Now we show soundness. Let $a \in \mathbf{P}(A)$. By definition of $\mathbf{P}$ ((3)), there is a derivation $\mathbf{a} \in \mathbf{D}(A)$ containing a. For each $a_i \in \mathbf{a}$, let b_i be the unique element in $\beta(a_i)$ (a singleton set because $\beta \preceq \alpha$), and let $\mathbf{b}$ be the corresponding sequence constructed from the b_is. Since $\beta \preceq \alpha$, we have $\mathbf{b} \in \mathbf{D}(B)$ by Theorem 3, and so each input tuple in $\mathbf{b}$ is also in $\mathbf{P}(B) = \tilde{B}$; in particular, $b \in \tilde{B}$ for $\beta(a) = \{b\}$. Since $\beta \preceq \alpha$, $a \in \alpha(b)$, and so $a \in \tilde{A}$. We have thus shown that $\mathbf{P}(A) \subset \tilde{A}$. Finishing up, $\mathbf{P}(A') = \mathbf{P}(A \cap \tilde{A}) \supset \mathbf{P}(A \cap \mathbf{P}(A)) = \mathbf{P}(\mathbf{P}(A)) = \mathbf{P}(A)$, where the last equality follows from idempotence (Lemma 1). □

We now show that the Prune-Refine algorithm is correct, which follows from a straightforward application of Theorem 1.

Proof of Theorem 2 (correctness of the Prune-Refine algorithm). First, we argue that pre-pruning is correct. For each iteration t, we invoke Theorem 1 with $\alpha = \alpha_t, \beta = \beta_t$ and X be such that $\alpha(X) = A_t$. The result is that $\mathbf{P}(A_t) = \mathbf{P}(A'_t)$, so without loss of generality, we will assume $A_t = A'_t$ for the rest of the proof.

Now fix an iteration t. We will show that $\mathbf{P}(\alpha_t(X)) = \mathbf{P}(A_t)$ by induction, where the inductive hypothesis is $\mathbf{P}(\alpha_t(X)) = \mathbf{P}(\alpha_t(\tilde{A}_s))$. For the base case ($s = -1$), we define $\tilde{A}_{-1} = \{X\}$, we get a tautology. For the inductive case, apply the theorem with applied to $\beta = \alpha_s, \alpha = \alpha_t$ and X such that $\alpha_s(X) = \alpha_s(\tilde{A}_{s-1})$, we get that

$$\mathbf{P}(\alpha_t(\tilde{A}_{s-1})) = \mathbf{P}(\alpha_t(\mathbf{P}(\alpha_s(\tilde{A}_{s-1})))) = \mathbf{P}(\alpha_t(\tilde{A}_s)).$$

When $s = t - 1$, we have $\mathbf{P}(\alpha_t(X)) = \mathbf{P}(\alpha_t(\tilde{A}_{t-1})) = \mathbf{P}(A_t)$, completing the claim. Finally, if the algorithm returns *proven*, we have

$$\emptyset = \mathbf{P}(A_t) = \mathbf{P}(\alpha_t(X)) \supset \mathbf{P}(X).$$

□