Scaling Abstraction Refinement via Pruning

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 analysis causes the number of contexts to grow exponentially. Iterative refinement techniques can be employed to mitigate this growth by starting with a coarse abstraction and only refining parts of the abstraction deemed relevant with respect to a client.

In this paper, we introduce a new way to use this client feedback to greatly reduce the complexity of an analysis: *pruning*. The pruned analysis is no longer a valid abstraction, but we prove that we do not forfeit correctness and that our analysis is still sound with respect to a given client. In the context of k-limited analysis, our approach amounts to keeping track of a set of context prefix patterns. By iteratively refining and pruning these patterns, we show that our analysis increases scalability greatly.

1. Introduction

Making a static analysis more precise requires increasing the complexity of the supporting 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 increases. 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). The general theme of all these approaches is that the complexity of the abstraction is increased only in parts deemed useful using some feedback from a client query.

In this paper, we use *pruning*, a different approach from the above selected refinement techniques for making static analyses scale. Our approach works on any analysis that can be expressed as a Datalog program. A Datalog program takes a set of input tuples (determined by the abstraction) and derives new tuples via a set of inference rules; the answer to a client query corresponds to whether a designated tuple can be derived. The key idea is to identify input tuples which are provably irrelevant for deriving the query tuple and prune these away. Selected refinement techniques attempt to keep the set of input tuples small by not refining some of them; we keep the set small by removing some of them entirely.

Pruning can be a dangerous affair though. With selected refinement, at any point we are still performing static analysis with respect to an abstraction and therefore have soundness guarantees. However, once we start pruning input tuples, we are no longer running a valid static analysis and thus do not automatically inherit soundness guarantees. Nonetheless, we prove that our method is *sound with respect to the client*.

While soundness is trivial for selected refinement but requires some argument for pruning, the situation is reversed for *completeness*, that is, the guarantee that an analysis is as precise as some target. For selected refinement, it is difficult to argue that the chosen coarser abstraction is as precise as if we had refined everything. However, with pruning, we are working directly with the more complex abstraction, and thus automatically inherit its completeness guarantees. By removing input tuples, we can only prove more queries.

We apply our pruning technique to the k-object-sensitivity abstraction [12], where objects on the heap are abstracted into a chain of allocation sites. Pruning in this setting corresponds to maintaining a subset of these chains. Although the main contribution of the paper is the general pruning technique, we also make the following contributions specific to k-limited pointer analysis which arise because we are pruning: First, we show that we need a more careful treatment of the k-object-sensitivity abstraction than before due to pruning. Second, we introduce an abstraction which limits chains to length k and also truncates to avoid having too many repeating sites; this is an effective way to increase k further (possible now due to pruning) without getting bogged down by recursion. Third, we show that an abstraction that replaces allocation sites by types (a generalization of [17]) is effective for pruning.

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

2. Preliminaries

The pruning approach that we present in this paper works on Datalog, a general language that has been used recently to declaratively express static analyses [2, 21]. Normally, one thinks of the Datalog program as encapsulating the abstraction, but it will be useful for us to decouple the two.[1] We first define Datalog (Section 2.1), which describes the computation of a query (think more in the direction of concrete semantics). Then, we focus on the abstraction (Section 2.2), which only interacts with the Datalog program via 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 $i = 1, \dots, |t.\mathbf{a}|$. We will write a term in any of the following three equivalent ways:

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

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_0 denote a designated *query tuple* (e.g., $\texttt{common}()$), 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 if

[1] The reason is that we want to make theoretical statements comparing the same Datalog program across different abstractions.

Graph Example

Input relations:

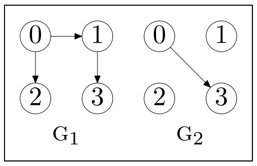
$\texttt{edge}(g,i,j)$	(edge from node i to node j in graph g)
$\texttt{head}(c,i)$	(first element of array c is i)
$\texttt{ext}(i,c,c')$	(i prepended to c yields c': $c' = [i] + c$)

Rules:

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

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

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



Input tuples:	**Derived tuples**:
$\texttt{edge}(\text{G}_1, 0, 1)$	$\texttt{path}(\text{G}_1, [0])$
$\texttt{edge}(\text{G}_1, 0, 2)$	$\texttt{path}(\text{G}_1, [10])$
$\texttt{edge}(\text{G}_1, 1, 3)$	$\texttt{path}(\text{G}_1, [20])$
$\texttt{edge}(\text{G}_2, 0, 3)$	$\texttt{path}(\text{G}_1, [310])$
$\texttt{head}([0], 0)$	$\texttt{path}(\text{G}_2, [30])$
$\texttt{ext}(1, [0], [10])$	
$\dots$	

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 $\texttt{common}(\text{G}_1, \text{G}_2, 3)$ means the query is false.

$z.t = t$ and $z.\mathbf{s} = \{s_1, \dots, s_k\}$ as

$$t \Leftarrow s_1, \dots, s_n. \tag{2}$$

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 ($f(t) = t.r(f(t.a_1), \dots, f(t.a_{|t.\mathbf{a}|}))$).

x_o	(designated query tuple)
$\mathcal{C}$	(set of concrete values)
$\mathbf{D}(X)$	(derivations using input tuples X)
$\mathbf{P}(X) \subset X$	(set of relevant input tuples)
$\alpha : \mathcal{C} \mapsto \mathcal{P}(\mathcal{C})$	(abstraction, maps to equivalence class)
$A_k \subset \mathcal{P}(\mathcal{C})$	(abstract input tuples after k iterations)
$\tilde{A}_k = \mathbf{P}(A_k)$	(relevant abstract input tuples)

Figure 2: Notation.

Derivations Intuitively, 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_o) with respect to a set of input tuples X is a sequence $\mathbf{x} = (x_1, \dots, x_n)$ such that

(i) for each $i = 1, \dots, n$, we have $x_i \in X$ or there exists a rule $z \in \mathcal{Z}$, an assignment f, such that $f(z.t) = x_i$ and for each $s \in z.\mathbf{s}$, there exists a $j < i$ such that $f(s) = x_j$;

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

(iii) for each $j = 1, \dots, n-1$, if x_j is removed, the resulting sequence no longer satisfies (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 x_o was finally produced via some sequence of rule applications. Condition (iii) says that the derivation of x_n is minimal; each tuple was used in some way; as we will see later, this condition is important because it allow us to define the set of relevant tuples for pruning.

A Datalog program computes $\mathbf{D}(X)$. We say that the query x_o is false (proven) if and only if $\mathbf{D}(X) = \emptyset$. Although the query is the ultimate quantity of interest, the Datalog program can be used to provide more information that is useful for pruning. Specifically, we define $\mathbf{P}(X)$ to be the subset of the input tuples X which were used in some derivation:

$$\mathbf{P}(X) \triangleq \{x \in X : x \in \mathbf{x} \in \mathbf{D}(X)\}. \tag{3}$$

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 $\mathbf{P}(X) = \emptyset$. 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 operation.

Computation We can compute $\mathbf{P}(X)$ in Datalog by using the Datalog program transformation technique described in [10]; this allows us to use off-the-shelf Datalog solvers which have already been optimized. Specifically, we define 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 using the corresponding new relation. We then add the following new Datalog rules:

$$x'_\text{o} \Leftarrow x_\text{o}. \tag{4}$$

$$s' \Leftarrow z.t', z.\mathbf{s}. \tag{5}$$

The key is that a tuple x' can be derived by the new Datalog program if and only if $x \in \mathbf{P}(X)$. These two rules construct $\mathbf{P}(X)$ recursively: The base case (4) states that the query tuple $x_\text{o} \in \mathbf{P}(X)$. The recursive case (5) states that if $x \in \mathbf{P}(X)$ 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 $f(s) \in \mathbf{P}(X)$.

2.2 Abstractions

Given a Datalog program, we define an *abstraction* as a equivalence relation over the constants $\mathcal{C}$ (concrete values). In particular, we represent it as the projection function which maps each element to its equivalence class (its abstract value):

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

We use the natural partial order on abstractions, where $\alpha_1 \preceq \alpha_2$ iff $\alpha_1(c) \supset \alpha_2(c)$ for all c, that is, α_2 is finer than α_1.

In the context of our example (Figure 1), we could define an abstraction that maps a sequence c to the set of sequences that have the same first element as c (e.g., $\alpha([10]) = \{[1], [10], [11], \dots\}$). Applied to a tuple, we get

$$\alpha(\texttt{ext}(1, [0], [10])) = \tag{6}$$
$$\texttt{ext}(1, \{[0], [00], [01], \dots\}, \{[1], [10], [11], \dots\}).$$

This abstraction is a simplified version of the abstractions we use in our k-limited analyses (see Section 2.2).

We extend α to sets of constants (which we also call abstract values):

$$\alpha(s) = \{\alpha(c) : c \in s\}, \quad s \in \mathcal{P}(\mathcal{C}), \tag{7}$$

which returns a set of abstract values. This allows us to naturally define compositions of abstractions by flattening sets of abstract values:

$$(\alpha_1 \circ \alpha_2)(c) \triangleq \cup_{s \in \alpha_1(\alpha_2(c))} s. \tag{8}$$

An important property of equivalence relations is that if $\alpha_1 \preceq \alpha_2$, then $\alpha_1 \circ \alpha_2 = \alpha_1$ (applying a finer abstraction first has no impact). Note that in general for arbitrary α_1, α_2, the composition $\alpha_1 \circ \alpha_2$ is not an abstraction; it is something that we must check when we consider compositions in Section 2.2.

We also extend α to concrete tuples x and sets of concrete tuples X:

$$\alpha(x) = x.r(\alpha(x.a_1), \dots, \alpha(x.a_{|x.\mathbf{a}|})), \tag{9}$$

$$\alpha(X) = \{\alpha(x) : x \in X\}. \tag{10}$$

Here, $\alpha(x)$ is an abstract tuple (one where the arguments are abstract values) and $\alpha(X)$ is a set of abstract tuples. Finally, we extend α to abstract tuples b and sets of abstract tuples B:

$$\alpha(b) = \{b.r(s_1, \dots, s_{|b.\mathbf{a}|}) : \forall i, s_i \in \alpha(b.a_i)\}, \tag{11}$$

$$\alpha(B) = \cup_{b \in B} \alpha(b). \tag{12}$$

(11) applies the abstraction function to each component and takes the cross product over the possible abstract values, yielding a set of abstract tuples; (12) aggregates these sets of abstract tuples.

Given an abstraction α, we can run an abstract version of the Datalog program to compute an abstract answer to the query tuple. We do this by applying the abstraction to the concrete input tuples X, to produce a set of abstract input tuples $\alpha(X)$. We then can feed these tuples into the Datalog program, which is oblivious to whether the tuples are abstract or concrete; conceptually, the Datalog program now operates on the constants $\mathcal{P}(\mathcal{C})$ rather than $\mathcal{C}$. Figure 3 shows an example of performing this computation on the graph example from Figure 1.

We say the query is proven by α if $\mathbf{P}(\alpha(X)) = \emptyset$, and it is proven only if the query is actually false ($\mathbf{P}(X) = \emptyset$). This is the standard soundness property, summarized 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).*

3. General theory

In this section, we show how we can use pruning to obtain sound analyses which are complete with respect to a given abstraction. We first describe the core idea behind pruning (Section 3.1) and then show how it can be used in our full PR algorithm (Section 3.2).

3.1 Pruning

Recall that the central operation is $\mathbf{P}$, which serves two functions: (i) determining queries are proven (when $\mathbf{P}$ returns $\emptyset$); and (ii) performing pruning by returning the subset of input tuples to keep. The following theorem is 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 input tuples X, we have:*

$$\boxed{\mathbf{P}(\alpha(X)) = \mathbf{P}(\alpha(X) \cap \alpha(\mathbf{P}(\beta(X))))}. \tag{13}$$

The left-hand side of (13) 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 significance of this is that the right-hand side is often a much cheaper way to compute the left-hand side.

Let us decipher (13) 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 that participate in a derivation of the abstract query tuple $\beta(x_o)$. These 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 to why pruning works, consider the following simpler idea: first run the analysis with β; if the query is proven, stop and declare *proven*; otherwise, we run the analysis with α and output. It is easier to see that this two-step procedure returns the same answer as just running α, because $\beta \preceq \alpha$ implies that if β proves the query, then so does α (Proposition 1). (13) can be thought of as an extension of this basic idea: instead of using β to just determine where the query tuple is proven, we obtain more information, namely all the input tuples that are relevant.

The complexity of an analysis is largely determined by the number of input tuples. Traditionally, the abstraction fully determines the set of input tuples and thus 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 very low cost.

3.2 The Pruning-Refinement (PR) algorithm

We now turn Theorem 1 into an algorithm, which we call the Pruning-Refinement (PR) algorithm. Figure 4 shows the pseudocode for the algorithm and a diagram showing the computation of the various abstract input tuples computed.

This algorithm essentially applies (13) 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 $\alpha_0, \alpha_1, \dots$, and an input set of abstract tuples A_0, computed under the initial abstraction α_0. Then the algorithm alternates between a pruning step and a refining step, always maintaining a set of abstract tuples whose elements are the only ones that could participate in a derivation of the query tuple x_o. 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 5 shows an example of running this algorithm on the graph example from Figure 1; the first pruning step is shown in Figure 3.

Now we discuss pre-pruning. Pre-pruning requires the user to provide another sequence of successively finer abstractions $\beta_0, \beta_1, \dots$ which are coarser than $\alpha_0, \alpha_1, \dots$, 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 using β_t during pre-pruning step, and then (ii) using α_t during the main pruning step. Pre-pruning does not affect the output of the pruning step ($\tilde{A}_t$); the purpose is solely to make the pruning step faster. One way to obtain the auxiliary abstractions is via compositions with another abstraction τ: $\beta_t = \alpha_t \circ \tau$. Intuitively, τ should neither be too coarse nor too fine. If τ is no abstraction, then pre-pruning is equivalent to running the pruning step; if τ is the trivial abstraction, pre-pruning will be fast but nothing will be pre-pruned. A rule of thumb is that τ should be complementary to α_t (we will see examples in Section 2.2).

Theorem 2 states that the PR algorithm is both sound and complete. In other words, pruning has no impact on our answer to a

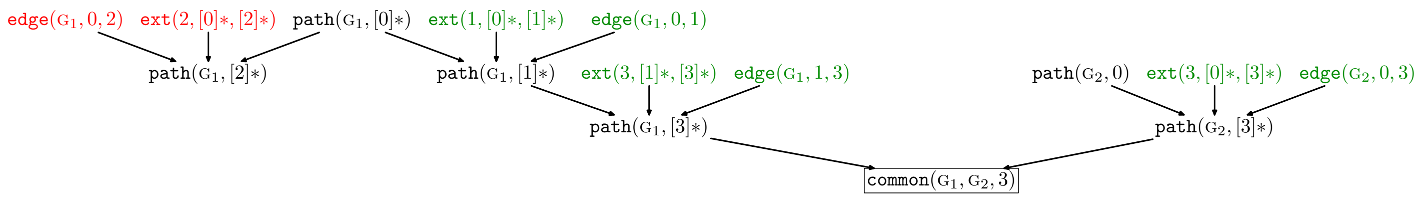


Figure 3: Computation of $\mathbf{P}(X)$ on the graph example from Figure 1 under an abstraction which maps a path onto the set of paths with the same first element (e.g., $\alpha([10]) = \{[1], [10], [11], \dots\} \triangleq [1]*$). Each abstract tuple is derived by a rule whose source terms corresponding to the incoming edges. Relevant input tuples ($\mathbf{P}(X)$, shown in green) are the ones which are reachable by following the edges backwards; ones which are not are pruned ($X \setminus \mathbf{P}(X)$, shown in red).

```
Pruning-Refinement (PR) Algorithm

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

For $t = 0, 1, 2, \dots$:
  [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: $A_{t+1} = \alpha_{t+1}(\tilde{A}_t)$.
```

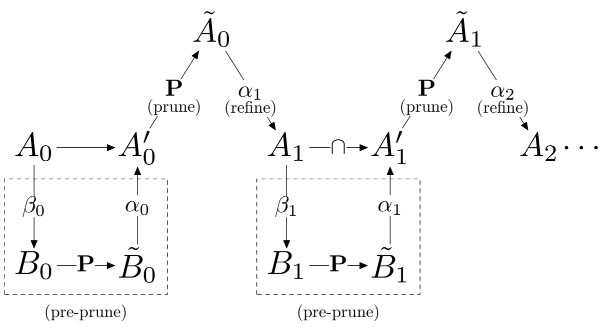


Figure 4: The pseudocode and the schema for the PR algorithm. The algorithm maintains a set of (abstract) input tuples which could be involved in some derivation of the query x_0 and attempts to prune down this set. The basic version, which excludes parts in square brackets, simply alternates between pruning and refining. The full version includes a pre-pruning step using auxiliary abstractions in order to further reduce the number of tuples.

query—we just compute it more efficiently. The proof is given in Appendix A and uses Theorem 1.

Theorem 2 (Correctness of the PR algorithm)**.** *At iteration t, the incrementally pruned abstraction A_t is equivalent to the full abstraction $\alpha_t(X)$ (formally, $\mathbf{P}(\alpha_t(X)) = \mathbf{P}(A_t)$). If the algorithm returns "proven," then $\mathbf{P}(X) = \emptyset$ (the query is actually false).*

4. k-limited Pointer Analysis

We now apply our general pruning approach to k-limited pointer 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

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 5: The abstract input tuples computed by the PR algorithm on the graph example from Figure 1 (without pre-pruning), where the abstraction α_t maps c to all paths that match c on the first $t+1$ nodes. Notation: $c*$ denotes the set of all paths with prefix c. During the first pruning step, ext$(2, [0]*, [2]*)$ is pruned from A_0, yielding $\tilde{A}_0$. In the refine step, we expand $[1]*$ to $[1]$ and $[10]*$, resulting in an extra tuple. In the second pruning step, we prove the query (pruning everything).

has a single argument and no return value.[2] Figure 6 describes the Datalog program corresponding to this analysis.

Our analysis represents both contexts and abstract objects using a list of allocation sites, which we will call *chains*. However, note that these chains are never truncated in the Datalog program. Therefore, running the Datalog program (ignoring the fact that it might not terminate) is performing an ∞-object-sensitivity analysis. Recall that we separate the computation of the query in Datalog with the abstraction which is applied to the input tuples (of course, our Datalog program is itself an approximation to the concrete semantics—it is flow-insensitive, does not handle primitive data, etc.). Section 2.2 will describe further abstractions of this analysis.

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_{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. 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 $\{o\}$.

[2] 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:

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

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

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`	$\subset$	$\mathbb{S} \times \mathbb{C} \times \mathbb{C}$	(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 6: Datalog implementation of our k-object-sensitivity 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 $\mathbf{a}_s$.

4.1 Clients

The core pointer analysis is used by three clients, which each define 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:

$$\texttt{poly}(i) \Leftarrow \texttt{cg}(*, i, *, m_1), \texttt{cg}(*, i, *, m_2), m_1 \neq m_2. \quad (14)$$

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:

$$\texttt{unsafe}(v_1, v_2) \Leftarrow \texttt{ptsV}(*, v_2, o), \texttt{typeO}(o, t_2), \texttt{typeV}(v_1, t_1), \neg\texttt{subtype}(t_1, t_2). \quad (15)$$

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 used 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.

5. Abstractions

Recall from Section 2 that we separate the Datalog program that computes the query from the abstraction. In Section 4, we described the Datalog programs that computed the queries for our three clients based on ∞-object-sensitivity. We now describe the abstractions that use to further abstract this analysis.

The basic abstraction (Section 5.1) corresponds exactly to k-object-sensitivity. We then present two orthogonal variants: one that in addition limits the repetition of allocation sites (Section 5.2) and one that further abstracts allocation sites using type information (Section 5.3).

5.1 k-limited abstraction

The *k-limited abstraction* essentially partitions chains based on their length k prefix. We first setup some notation. For a chain $c \in \mathbb{H}^*$, let $|c|$ denote the length of the chain. Let $c[i]$ be the i-th element of c (starting with index 1) and let 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, c_2, let $c_1 + c_2$ denote concatenation. For a chain c, let $c*$ denote the set of all chains with prefix c; formally:

$$c* \triangleq \{c' \times \mathbb{H}^* : c'[1..|c|] = c\}. \quad (16)$$

$\{[0]\}$ $\{[1]\}$

$\{[00]\}$ $\{[01]\}$ $\{[10]\}$ $\{[11]\}$

$[000]*$ $[001]*$ $[010]*$ $[011]*$ $[100]*$ $[101]*$ $[110]*$ $[111]*$

Figure 7: For the k-limited abstraction with $\mathbb{H} = \{0, 1\}$ and $k = 3$, the equivalence classes defined by π_k.

```
class A {
  f() {
0:  v = new A                     1:  x1 = new A
    if (*) return v               2:  x2 = new A
    else return v.f()                 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 $\{[02], [002], [0002], \dots\}$. These two sets are disjoint, but for any finite k, no matter how large, k-object-sensitivity would conclude that both point to $\mathbf{0}_k*$, where $\mathbf{0}_k$ is a chain of k zeros.

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

$$\pi_k(c) \triangleq \begin{cases} \{c\} & \text{if } |c| < k \\ c[1..k]* & \text{if } |c| \geq k. \end{cases} \tag{17}$$

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 as defined is a valid abstraction under Definition 1. Figure 7 shows an example.

Traditionally, k-limited analyses simply use a list of sites "c" to represent an abstract value, not making a distinction between $\{c\}$ and $c*$; however when we do pruning, this distinction is crucial. Suppose that $k = 3$. Naïvely, "[01]" would just represent $\{[01]\}$. The problem comes when we prune. If we prune "[010]", then "[01]" really represents $\{[01]\} \cup [010]*$ because any chain starting with [010] must now fall back on "[01]". This is unacceptable because pruning tuples should not change the abstraction. Actually, the situation is worse, because (i) in fact no pruning (in the sense that some chains are ignored) would happen and (ii) the resulting analysis would be much more imprecise because "[ ]" is always be present for representing the context of the main method, but this is a prefix of any chain.

5.2 Barely-repeating k-limited abstraction

When we applied the k-limited abstraction empirically, we noticed that a major reason why it did not scale was the seemingly unnecessary combinatorial explosion associated 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.[3] We therefore wish to define an abstraction that not only truncates at 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 c excluding the last element ($c[1..|c|-1]$) is *non-repeating*. Note that a barely-repeating

[3] 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.

$\{[0]\}$ $\{[1]\}$

$[00]*$ $\{[01]\}$ $\{[10]\}$ $[11]*$

$[010]*$ $[011]*$ $[100]*$ $[101]*$

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 7). Note that, for example, $[000]*$ and $[001]*$ are collapsed into $[00]*$ since [000] and [001] are not barely-repeating, but [00] is.

includes the non-repeating case. Let $\delta(c)$ be the length of the longest prefix that is barely-repeating:

$$\delta(c) \triangleq \max_{m' : c[1..m'] \text{ is barely-repeating}} m'. \tag{18}$$

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

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

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

Figure 9 shows an example of $\hat{\pi}_k$. It is not automatic that π_k is a valid abstraction, but we can check that it 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, we can check that any extension $c' \in c*$ will have $\pi_k(c') = c*$. □

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 $\hat{\pi}_3([01]) = \{[01]\} \neq [0]*$.

5.3 Class-based abstraction

We now introduce an abstraction that we will use in the pre-pruning step of the PR 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 its equivalence class. Given such such an τ, we could then extend it sequences by elementwise application:

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

To construct τ, we consider two sources of type information associated with an allocation site, motivated by [17]:

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

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

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

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

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

Now we have three options for τ which are complementary to the k-limited abstraction: τ abstracts a chain by abstracting each site uniformly; π_k keeps all sites up to a length k prefix precisely but abstracts away the rest completely. This complementarity is desirable for pre-pruning. However, τ is not coarser than π_k, and therefore cannot be used directly in the PR algorithm.

However, we can compose τ and π_k to yield another abstraction which will be coarser than π_k. But in what order should we compose? We must take care because the composition of two equivalence relations is not necessarily an equivalence relation. The following proposition shows which orderings 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 $m \le k$ (case 2), where $w_i \in \text{range}(\tau)$ for each $i = 1, \dots, m$.

For (i) and (ii), it is straightforward to check $(\pi_k \circ \tau)(c) = (\tau \circ \pi_k)(c) = s$ 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 $\tau(v) = \mathbb{H}$ for all $v \in \mathbb{H}$ (there is one equivalence class). Consider two elements [01] and [00] under the composed abstraction: 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 an equivalence relation. □

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 τ and then applying π_k or $\hat{\pi}_k$.

6. Experiments

In this section, we apply our PR algorithm (Section 3.2) to k-object-sensitivity for three clients: downcast safety checking (DOWNCAST), monomorphic call site inference (MONOSITE), and race detection (RACE). These analyses are described in Section 4. Our main empirical result is that across different clients and benchmarks, pruning is very effective at curbing the exponential growth, which allows us to run the analysis on very refined abstractions.

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,[4] which uses the BDD solver `bddbddb` [21]. We evaluated 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 FULL(α)). For α, we can either use k-limited abstractions ($(\pi_k)_{k=1}^{\infty}$), in which case we recover ordinary k-object-sensitivity, or the barely-repeating variants ($(\hat{\pi}_k)_{k=1}^{\infty}$). We also consider the site-based refinement algorithm of [10], which considers a sequence of abstractions $\boldsymbol{\alpha} = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_1, \dots)$ but stops refining sites which have been deemed irrelevant (this algorithm is denoted SITE($\boldsymbol{\alpha}$)).

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

Abstractions	
$\boldsymbol{\pi} = (\pi_k)_{k=1}^{\infty}$	(k-limited abstractions (17))
$\hat{\boldsymbol{\pi}} = (\hat{\pi}_k)_{k=1}^{\infty}$	(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	
FULL(α)	(standard analysis using abstraction α)
SITE($\boldsymbol{\alpha}$)	(site-based refinement [10] on abstractions $\boldsymbol{\alpha}$)
PR($\boldsymbol{\alpha}$)	(PR algorithm using $\boldsymbol{\alpha}$, no pre-pruning)
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, PR($\hat{\boldsymbol{\pi}}, \tau_{\text{I}\times\text{C}}$) means running the PR-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, $\beta_k = \hat{\pi}_k \circ \tau_{\text{I}\times\text{C}}$.

As for the new methods that we propose in this paper, we have PR($\boldsymbol{\alpha}$), which uses a sequence of abstractions $\boldsymbol{\alpha}$, which does no pre-pruning, or PR($\boldsymbol{\alpha}, \tau$), which performs pre-pruning using $\beta_t = \alpha_t \circ \tau$ for $t = 0, 1, 2, \dots$. We consider three choices of τ ($\tau_{\text{I}}, \tau_{\text{C}}, \tau_{\text{I}\times\text{C}}$) which use different kinds of type information.

In our implementation of the PR algorithm, we depart slightly from our presentation. Instead of maintaining the set of relevant input tuples as determined by our k-limited Datalog program across iterations, we instead maintain the allocation site chains which were involved in any relevant input tuple. We can modify the original Datalog program so that the PR algorithm is consistent with this implementation by introducing a new input relation `active`(c) for $c \in \mathbb{H}^*$, encoding existing input relations as rules consisting of zero source terms, and adding `active`(c) for each existing rule that uses a variable c which is an allocation site. Now computing the relevant input tuples corresponds exactly to computing the set of relevant allocation site chains.

6.2 Results

Figure 10 compares the number of input tuples needed by various algorithms; we see that the non-pruning methods completely hit a wall, whereas the pruning methods are able to continue increasing k farther. We also observed that the running time of these analyses tracked the number of tuples somewhat, so we do obtain speedups, but the differences are less pronounced. One reason is that we are using BDDs, which can handle large numbers of tuples so long as they are structured; pruning destroys some of this structure, yielding less predictable running times.

Table 3 provides more on the quantitative impact of pruning in our best instantiation of the PR algorithm. We see that pre-pruning has a significant positive impact: we are able to perform pruning at a much cheaper level (using types instead of allocation sites). Many of the tuples are pruned at this level, and the results of this pruning carry over to the original k-limited abstraction.

So far, we have been using the k-limited abstraction. In Figure 11, we show that when we can increase the k value, the barely-repeating k-limited abstraction is helpful, but in most cases, it does not improve scalability. The reason is that the barely-repeating abstraction curbs refinement, but often, and quite paradoxically, it is exactly refinement which enables us to prune more.

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 sat-

	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 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 ($\lvert\mathbb{H}\rvert$) deemed reachable by 0-CFA.

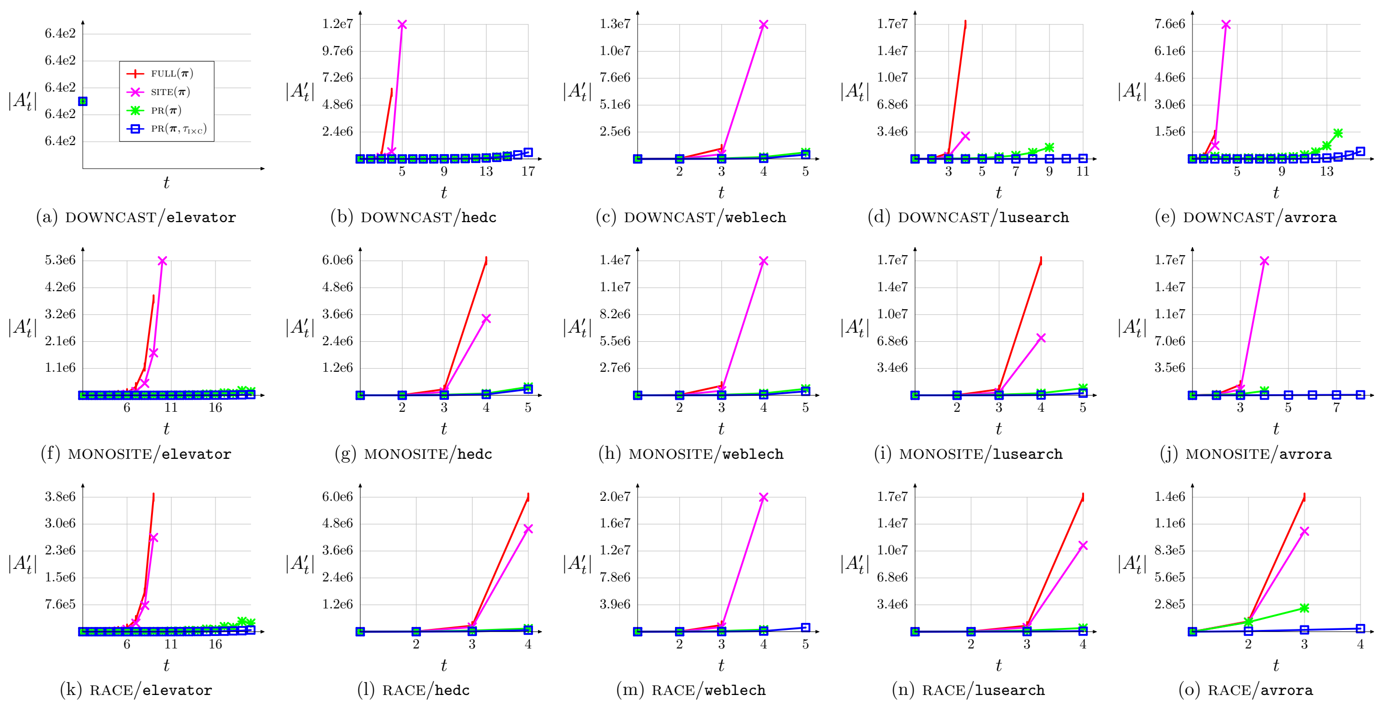


Figure 10: For each client/benchmark pair, we show the growth of the number of input tuples $|A'_t|$ across iterations. Recall that $|A'_t|$ is the number of tuples fed into the Datalog program. Table 2 describes the four methods. We see that the methods that prune drastically cut down the number of input tuples by many orders of magnitude.

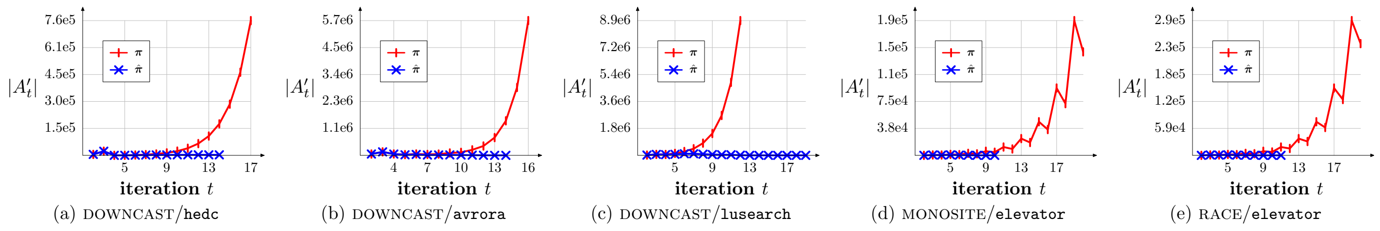


Figure 11: Shows the 5 (out of 15) client/benchmark pairs for which using the barely-repeating k-limited abstraction ($\hat{\boldsymbol{\pi}}$) allows one to increase k much more than the plain k-limited abstraction ($\boldsymbol{\pi}$). On the other 10 client/benchmarks where k cannot get very large, limiting repetitions is actually slightly worse in terms of scalability.

urate quite quickly, so we are only able to prove two more queries 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] obtains a lower bound, we know for sure that low k values are insufficient; the fact that we don't see an increase in precision suggests that the non-k-related aspects of our analyses are insufficient. However, given that our pruning approach is general, it would be interesting tackling other aspects such as flow-sensitivity.

	$\frac{\lvert B_t \rvert}{\lvert A_t \rvert}$	$\frac{\lvert \tilde{B}_t \rvert}{\lvert B_t \rvert}$	$\frac{\lvert A'_t \rvert}{\lvert A_t \rvert}$	$\frac{\lvert \tilde{A}_t \rvert}{\lvert A_t \rvert}$	$\frac{\lvert A_{t+1} \rvert}{\lvert A_t \rvert}$
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 4) for our best method $\text{PR}(\boldsymbol{\pi}, \tau_{1\times\text{C}})$ 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 β_t 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, see that the effect of pre-pruning essentially carries over ($\frac{|A'_t|}{|A_t|}$). Recall that we do the pruning step after pre-pruning; $\frac{|\tilde{A}_t|}{|A_t|}$ measures the total fraction of chains kept during pruning. 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.

	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 are ones obtained by our best method $\text{PR}(\boldsymbol{\pi}, \tau_{1\times\text{C}})$ but not by the full algorithm. We see that although pruning enables to increase k by much more than before, only for two the client/benchmark pairs do we get strictly more precise results. This points out inherent limitations of this family of k-limited abstractions.

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] also consider abstractions, where each local variable can be independently treated context sensitively or 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 alias analysis. [17] scale up k-object-sensitivity one more level by using types rather than allocation sites. However, 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 conflict. 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.

Perhaps the closest to this work is the part of Liang et al. [10], which uses the same technique to compute the set of relevant tuples. However, what is done with this information is quite different. [10] merely stops refining the irrelevant sites whereas we actually prune all irrelevant tuples, thereby exploiting this information fuller. As we saw in Section 6, this difference had major ramifications.

Demand-driven analyses [5, 23] do not refine the abstraction but rather compute an analysis on an existing abstraction faster. Some approaches such as [19] do both. Our PR algorithm has a flavor of the client-driven approach, in that we do refine our abstraction, but also of the demand-driven approach, in that we do not perform a full computation (in particular, ignoring tuples which were pruned).

Pruning has been implemented in various settings. [20] use dynamic analysis to prune down set of paths and then focus a static analysis on these paths. [18] use pruning for type inference in functional languages, where pruning is simply a heuristic which short-cuts 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 way of scaling up static analyses written as Datalog programs. The key idea is to run an analysis using a coarse abstraction, only keeping input tuples deemed relevant for a finer abstraction. Theoretically, we showed that pruning is both sound and complete (our analysis is valid and we lose no precision). Empirically, we showed that our pruning approach enables us to scale up analyses based on k-object-sensitivity much more than previous approaches.

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 X be any set of input tuples. For any derivation $\mathbf{a} \in \mathbf{D}(\alpha(X))$, define $\mathbf{b}$ by $b_i \in \beta(a_i)$ (in fact, b_i is unique) for each position 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 define $\mathbf{b}$ as 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 corresponding to 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), $\mathbf{P}(X) = \mathbf{P}(\mathbf{P}(X))$.*

Proof. Since $\mathbf{P}(X) \subset X$ by defnition 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 $\mathbf{x} \in \mathbf{D}(\mathbf{P}(X))$ (we can still derive $\mathbf{x}$ using those tuples). 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 (13): $A = \alpha(X)$ 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 drived 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, $b = \tilde{B}$ 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 PR algorithm is correct, which follows from a straightforward application of Theorem 1.

Proof of Theorem 2 (correctness of the PR *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

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

☐