Accelerating Program Analyses by Cross-Program Training

Sulekha Kulkarni Ravi Mangal Xin Zhang Mayur Naik
Georgia Institute of Technology, USA
{sulekha, ravi.mangal, xin.zhang, naik}@gatech.edu

Abstract

Practical programs share large modules of code. However, many program analyses are ineffective at reusing analysis results for shared code across programs. We present POLYMER, an analysis optimizer to address this problem. POLYMER runs the analysis offline on a corpus of training programs and learns analysis facts over shared code. It *prunes* the learnt facts to eliminate intermediate computations and then reuses these pruned facts to accelerate the analysis of other programs that share code with the training corpus.

We have implemented POLYMER to accelerate analyses specified in Datalog, and apply it to optimize two analyses for Java programs: a call-graph analysis that is flow- and context-insensitive, and a points-to analysis that is flow- and context-sensitive. We evaluate the resulting analyses on ten programs from the DaCapo suite that share the JDK library. POLYMER achieves average speedups of 2.6× for the call-graph analysis and 5.2× for the points-to analysis.

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

Keywords Optimization, Program Analysis, Declarative Analysis, Modular Summaries

1. Introduction

Many static analyses face the challenge of analyzing large programs efficiently. Such programs often share large modules of code. For example, Java programs heavily use the Java standard library, and Android applications extensively use the Android framework. As a result, a compelling approach to improve the performance of an analysis is to reuse its results on shared code across programs.

A common technique to realize this approach is *modular* or *compositional* analysis [7, 8, 11, 14, 28]. The theory of such analyses is well-studied [10] but designing and implementing them for realistic languages is challenging. Complex data- and control-flow resulting from language features such as dynamically allocated memory and higher-order functions hinder key aspects of modular analysis, such as accounting for all potential calling contexts of a procedure in a sound and precise way using summaries, representing the summaries compactly, and instantiating them efficiently. As a result, many practical static analyses reanalyze entire programs from scratch. Popular frameworks for such analyses (e.g., [9, 20, 23]) have been integrated into analysis tools such as SLAM [4], Soot [5], WALA [12], and Chord [18]. While easier to design and implement, however, the performance of such analyses is hindered by their inability to reuse analysis results for shared code across programs. For instance, a flow- and context-insensitive points-to analysis of a simple "Hello World" Java program requires analyzing over 3,000 classes in the Java standard library.

There are many challenges to reusing results of such an analysis on shared code in a manner that achieves speedup while ensuring correctness. First, the vocabulary of base facts is different across programs. Second, not all analysis facts about shared code are amenable to unconditional reuse across programs which share that code; certain facts may be conditional on facts that differ across programs. Third and most importantly, to maximize speedup, we must prune *intermediate* analysis facts about shared code, but preserve all *externally visible* analysis facts. Pruning is thus a precarious balancing act: under-pruning curtails performance gains whereas over-pruning produces unsound results.

To enable different analyses to take advantage of pruning, we propose a pruning-based framework POLYMER for arbitrary analyses specified in Datalog, a logic programming language. Datalog is a popular choice for expressing many analyses [6, 15, 24, 26, 27]. In POLYMER, besides specifying the analysis as a set of inference rules, the analysis writer also specifies which classes of analysis facts to prune.

The POLYMER framework comprises an offline phase and an online phase. In the offline phase, it takes as input a corpus of training programs and a specification of what constitutes shared code. It computes analysis facts for the shared code, prunes them as directed by the pruning specification, and stores them in a persistent database. In the online phase, it uses the pruned analysis facts in a sound manner to accelerate the analysis of a new program. POLYMER addresses the

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4444-9/16/11...$15.00
http://dx.doi.org/10.1145/2983990.2984023

challenge of reusing analysis facts in a uniform and analysis-agnostic manner by generating a condition to ensure soundness for every analysis fact that is slated for reuse. A pruned analysis fact is reused only if its associated soundness condition is satisfied.

We instantiate POLYMER on two analyses for Java programs: a call-graph analysis that is flow and context-insensitive, and a points-to analysis that is flow and context-sensitive. We employ two popular variants of the points-to analysis: one that uses a pre-computed call-graph and another that constructs the call-graph on-the-fly. We evaluate the resulting analyses on ten programs from the DaCapo suite. These programs are of size 208-419 KLOC each and share the JDK library. By picking each of these programs in turn as the test program while the remaining programs form the training corpus, POLYMER achieves average speedups of 2.6× for the call-graph analysis and 5.2× for the points-to analysis. Our experiments also provide insights into the extent and limits on speedup by varying the training corpus.

We summarize the contributions of this work:

1. We introduce the concept of using *pruned* analysis facts, learnt over the analysis of a corpus of training programs, in order to accelerate the analysis of a new program. These learnt analysis facts are over code shared across the training corpus and the new program.
2. We develop a framework POLYMER that applies this approach to arbitrary analyses specified in Datalog. If POLYMER is provided with a sound pruning specification, the accelerated analysis is guaranteed to be sound and produces the same result as the original.
3. We demonstrate significant performance gains using POLYMER for two fundamental static analyses on a set of realistic Java benchmark programs.

2. Example

POLYMER aims to accelerate the analysis of programs that share large modules of code. For this purpose, POLYMER needs the analysis to be specified in Datalog. It also needs the specification of two analysis-specific functions called *PickGoodPre* and *PickGoodPost*. These functions guide POLYMER in terms of *what* to learn from the analysis of a program (or programs), and *how* to apply the learnt results to accelerate the analysis of another program (or programs). In this section, we explain our approach using a graph reachability analysis. In the example that follows, POLYMER trains on the analysis of a single graph and applies the learnt results to the analysis of another graph that has some subgraph in common with the training graph.

Figure 1 shows two example graphs A and B with a shared subgraph L. Figure 2 shows the Datalog specification required by POLYMER to learn from the analysis of graph A and apply the learnt results to accelerate the analysis of graph B. Each graph is a directed graph consisting of two kinds of nodes: application nodes (labeled A0, ..., A2 and B0, ..., B3) and

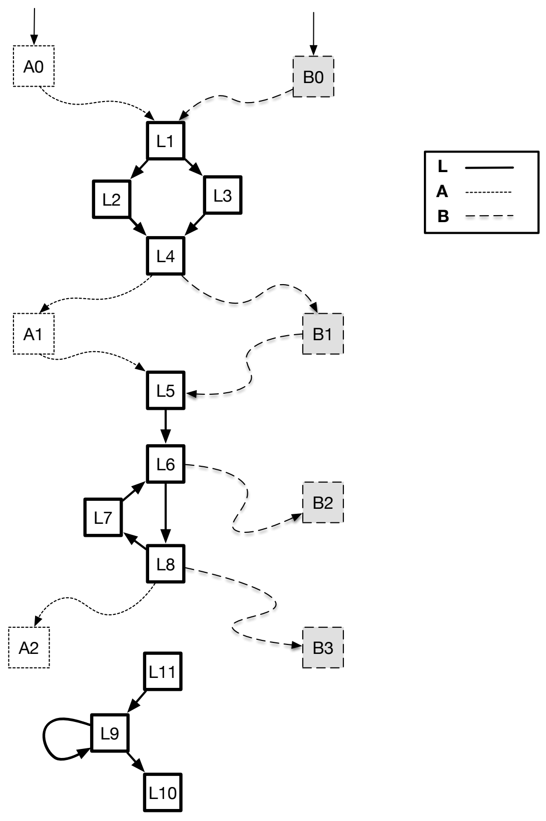

Figure 1: Example graphs A and B with shared subgraph L. POLYMER trains on the analysis of graph A and accelerates the analysis of graph B.

library nodes (labeled L1, ..., L11). The subgraph formed with only library nodes is common for both graphs. Graph A comprises the application nodes A0, ..., A2, the library nodes, and all the edges over those nodes. Likewise, graph B comprises the application nodes B0, ..., B3, the library nodes, and all the edges over those nodes. Intuitively, each graph represents a program: application nodes encode the application-specific facts, while the library nodes encode the facts of the shared library. The nodes A0 and B0 are distinguished nodes encoding the facts that hold at the entry of each program.

Problem Description. The goal of the graph reachability problem is to compute the set of *application nodes* that are reachable from node A0 for graph A, and from node B0 for graph B. We solve this problem using a Datalog program comprising two rules (1) and (2) as shown in Figure 2. Input relation edge contains the set of edges in the given graph while relation app contains the set of application nodes in the given graph. Output relation reachable contains the set of nodes that are reachable from the distinguished entry node m_0 of the graph being analyzed. Rule (1) is the base case, which states that m_0 is reachable. Rule (2) is the inductive

Graph Reachability Analysis:

Domains:

$\mathbb{N}$ is a set of nodes.

Input relations:

$\mathsf{edge}(m : \mathbb{N}, n : \mathbb{N})$	// Edge from node m to n.
$\mathsf{app}(m : \mathbb{N})$	// Node m is an application node.

Output relations:

$\mathsf{reachable}(m : \mathbb{N})$	// Node m is reachable from m_0.

Rules:

$$\mathsf{reachable}(m_0). \quad (1)$$

$$\mathsf{reachable}(m) \text{ :- } \mathsf{reachable}(n), \mathsf{edge}(n, m). \quad (2)$$

Specification for POLYMER:

$$\mathit{PickGoodPre} = \{ \{n\} \mid \neg\mathsf{app}(n) \wedge \exists m. (\mathsf{app}(m) \wedge \mathsf{reachable}(m) \wedge \mathsf{edge}(m, n)) \}$$

$$\mathit{PickGoodPost} = \lambda(N, M). (\mathit{post}, \mathit{check})$$

where M is the set of all library nodes reachable from nodes in N using only edges in the library, and

	naive	**pruned**
post	M	$\{ p \mid p \in M \wedge \exists q. (\mathsf{edge}(p, q) \wedge \mathsf{app}(q)) \}$
check	*true*	$\forall p \in M \setminus \mathit{post}. \nexists q. (\mathsf{edge}(p, q) \wedge \mathsf{app}(q))$

Figure 2: Inputs to POLYMER: (1) The analysis in Datalog, (2) a function *PickGoodPre* to compute suitable pre-conditions, and (3) a function *PickGoodPost* to compute suitable post-conditions, shown with two alternate definitions: (i) **naive**, without pruning and a trivial checking function, and (ii) **pruned**, with maximal pruning.

step, which states that if n is reachable, and there exists an edge from n to m, then m is also reachable. Henceforth, we will denote $\mathsf{reachable}(n)$ as $\mathsf{r}(n)$, and $\mathsf{edge}(n, m)$ as $\mathsf{e}(n, m)$ for brevity.

Executing the Datalog program produces the reachable relation as $\{ \mathsf{r}(\mathtt{A}i) \mid i \in [0, 2] \} \cup \{ \mathsf{r}(\mathtt{L}i) \mid i \in [1, 8] \}$ for graph A and $\{ \mathsf{r}(\mathtt{B}i) \mid i \in [0, 3] \} \cup \{ \mathsf{r}(\mathtt{L}i) \mid i \in [1, 8] \}$ for graph B. Although both the graphs share the same library subgraph, the Datalog program recomputes the entire reachability information for both the graphs. It is desirable to avoid repeatedly computing the library reachability information across Datalog runs. Moreover, to maximize speedup, we should avoid deriving library facts such as $\mathsf{r}(\mathtt{L2})$ as it is not necessary to our above-stated goal of deciding reachability of application nodes.

Existing Approaches. One solution to this problem is to compute library reachability information independent of the application, similar to modular program analyses. However, this is impractical as the library subgraph can be very large, making the computation an expensive process. Moreover, in most cases, large parts of the library are irrelevant for the application under consideration. For example, in Figure 1, the subgraph over the nodes L9, L10 and L11 is not involved in the reachability computation for either of the two graphs. This is in line with the observation that, while it is expensive to compute a precise summary for all calling contexts of a given library method, only a few calling contexts are often encountered in applications, whose summaries can be efficiently computed.

***Our Approach* POLYMER.** POLYMER trains on the analysis of graph A and learns summaries over those parts of the library subgraph reachable from the application nodes. It then accelerates the analysis of graph B by applying these learnt summaries in a sound manner. We refer to the training phase as the *offline phase* and the reuse phase as the *online phase*. We next explain these two phases in detail.

***Offline Phase of* POLYMER.** To train on the analysis of graph A, POLYMER first runs the original Datalog program on A and produces library facts $\mathsf{r}(\mathtt{L1})$ through $\mathsf{r}(\mathtt{L8})$ as well as application facts $\mathsf{r}(\mathtt{A0})$ through $\mathsf{r}(\mathtt{A2})$. Next, POLYMER analyzes the Datalog run to construct summaries. A *summary* is of the form $\mathit{tuples}_{\mathsf{pre}} \Rightarrow \mathit{tuples}_{\mathsf{post}}$, where each of $\mathit{tuples}_{\mathsf{pre}}$ and $\mathit{tuples}_{\mathsf{post}}$ is a set of tuples (i.e., analysis facts), and has the meaning: "if all tuples in $\mathit{tuples}_{\mathsf{pre}}$ can be derived, then all tuples in $\mathit{tuples}_{\mathsf{post}}$ can be derived". In this example, by analyzing the Datalog run, POLYMER concludes that (1) if $\mathsf{r}(\mathtt{L1})$ is derived, then $\mathsf{r}(\mathtt{L2})$ through $\mathsf{r}(\mathtt{L4})$ will also be derived and (2) if $\mathsf{r}(\mathtt{L5})$ is derived, then $\mathsf{r}(\mathtt{L6})$ through $\mathsf{r}(\mathtt{L8})$ will also be derived. As a result, POLYMER constructs the following two summaries:

$$\{\mathsf{r}(\mathtt{L1})\} \Rightarrow \{\mathsf{r}(\mathtt{L2}), \mathsf{r}(\mathtt{L3}), \mathsf{r}(\mathtt{L4})\},$$
$$\{\mathsf{r}(\mathtt{L5})\} \Rightarrow \{\mathsf{r}(\mathtt{L6}), \mathsf{r}(\mathtt{L7}), \mathsf{r}(\mathtt{L8})\}.$$

When $\mathsf{r}(\mathtt{L1})$ is derived during the analysis of an unseen graph, instead of deriving $\mathsf{r}(\mathtt{L2})$ through $\mathsf{r}(\mathtt{L4})$ by applying Rule (2) multiple times, POLYMER applies this summary to derive them at once. We say POLYMER *applies* a summary when it suppresses derivations initiated by the pre-condition $\{\mathsf{r}(\mathtt{L1})\}$ and inserts all the tuples in the post-condition, that is, $\{\mathsf{r}(\mathtt{L2}), \mathsf{r}(\mathtt{L3}), \mathsf{r}(\mathtt{L4})\}$.

Note that in this example, it is intuitive to construct library summaries with $\{\mathsf{r}(\mathtt{L1})\}$ and $\{\mathsf{r}(\mathtt{L5})\}$ as the pre-conditions since nodes L1 and L5 represent the boundary between the application and the library nodes. But POLYMER could also have constructed $\{\mathsf{r}(\mathtt{L2})\} \Rightarrow \{\mathsf{r}(\mathtt{L4})\}$ as a valid library summary. Such a summary, however, would lead to lower computational savings since deriving $\mathsf{r}(\mathtt{L2})$ before the summary can be applied would require an additional application of Rule (2). In general, there is a large space of possible pre-conditions to use for summary construction, and POLYMER requires a function *PickGoodPre* that specifies the form of pre-conditions to use. For our graph reachability analysis, the *PickGoodPre* function specified in Figure 2 exactly captures the intuition of using library nodes at application-library boundary in the pre-conditions.

To further reduce the computation cost, we observe that, in the first summary, only $\mathsf{r}(\mathtt{L4})$ is directly used in deriving application facts. POLYMER takes advantage of this to prune away the other tuples and produce a pruned summary as: $\{\mathsf{r}(\mathtt{L1})\} \Rightarrow \{\mathsf{r}(\mathtt{L4})\}$. When applying the pruned summary in the online phase, POLYMER will prevent deriving intermediate library facts such as r(L2) and r(L3). This example is just illustrative, and typically libraries often have a lot of internal facts that can be pruned away.

However, it is not always sound to apply the pruned summary on a given graph. Consider the second summary learnt by POLYMER: $\{\mathsf{r}(\mathtt{L5})\} \Rightarrow \{\mathsf{r}(\mathtt{L6}), \mathsf{r}(\mathtt{L7}), \mathsf{r}(\mathtt{L8})\}$. The corresponding pruned summary is: $\{\mathsf{r}(\mathtt{L5})\} \Rightarrow \{\mathsf{r}(\mathtt{L8})\}$ since r(L8) is the only fact directly used in deriving application facts for A. Applying the pruned summary in B will cause the analysis erroneously conclude that B2 is unreachable from B0. This is because the fact r(L6) has been pruned away and e(L6, B2) can no longer join with r(L6) to conclude that B2 is reachable from B0.

To solve this problem, while pruning away the intermediate facts from the summary, POLYMER generates a checking function that ensures the sound application of the summary in the online phase. The checking function takes the input relations of the online phase and returns whether it is sound to apply the pruned summary. We make the notion of sound pruning precise in Section 4.2. For the graph reachability example, we observe that a reachable tuple can only derive another tuple by joining with edge tuples via Rule (2). Since the library subgraph is the same for all graphs, applying the pruned summary will only become unsound if there exists an edge from any pruned library node to an application node. As a result, POLYMER generates a checking function for each of the pruned summaries as below:

$$\forall i \in \{2,3\}.\ \nexists n.\ \mathsf{edge}(\mathtt{L}i, n) \wedge \mathsf{app}(n) \quad (1)$$
$$\forall i \in \{6,7\}.\ \nexists n.\ \mathsf{edge}(\mathtt{L}i, n) \wedge \mathsf{app}(n) \quad (2)$$

In general, in the offline phase, it is impossible to predict what facts to prune away from a given summary. We need to ensure that an intermediate library fact that is pruned away is not exposed to the application facts in the online phase. Therefore, POLYMER requires the specification of a *PickGoodPost* function that specifies what facts to prune away from any given summary and how to generate the corresponding checking function required to ensure the soundness of pruning. Figure 2 includes two different specifications of *PickGoodPost* for the graph reachability example. In general, *PickGoodPost* takes as input the pre-condition N and corresponding post-condition M of an unpruned summary. The **naive** version of *PickGoodPost* in Figure 2 corresponds to the case when no pruning is performed and thus, it just returns M as the post-condition without any pruning. Further, in this case, the checking function *check* always returns true, implying that it is sound to apply such an unpruned summary on any given graph. The **pruned** version of *PickGoodPost* returns a post-condition containing only those library nodes that directly interact with application nodes. Moreover, the checking function captures our intuition that a pruned summary is unsound for a given graph if there exists an edge in this graph from any pruned library node to an application node.

$$\begin{array}{rlrl}
(\textit{analysis}) C ::= \{c_1, ..., c_n\} & (\textit{argument})\ \alpha ::= v \mid d \\
(\textit{constraint})\ c ::= l_0 \,\text{:-}\, l_1, ..., l_n & (\textit{variable})\ v \in \mathbb{V} = \{x, y, ..\} \\
(\textit{literal})\ l ::= r(\alpha_1, ..., \alpha_n) & (\textit{constant})\ d \in \mathbb{N} = \{0, 1, ..\} \\
(\textit{relation name})\ r \in \mathbb{R} = \{\mathsf{a}, \mathsf{b}, ..\} & (\textit{tuple})\ t \in \mathbb{T} = \mathbb{R} \times \mathbb{N}^* \\
(\textit{analysis input})\ i \in \mathbb{I} \subseteq \mathbb{T} & (\textit{analysis output})\ o \in \mathbb{O} \subseteq \mathbb{T}
\end{array}$$

Figure 3: Syntax of a Datalog analysis.

$$\begin{aligned}
[\![C]\!] &\in 2^{\mathbb{I}} \to 2^{\mathbb{T}} \\
[\![c]\!] &\in 2^{\mathbb{T}} \to 2^{\mathbb{T}} \\
[\![l]\!] &\in \mathbf{\Sigma} \to \mathbb{T}, \text{ where } \mathbf{\Sigma} = \mathbb{V} \to \mathbb{N} \\
[\![C]\!](I) &= \textbf{lfp}\ \lambda T.T \cup I \cup \textstyle\bigcup_{c \in C} [\![c]\!](T) \\
[\![l_0 \,\text{:-}\, l_1, ..., l_n]\!](T) &= \{[\![l_0]\!](\sigma) \mid \textstyle\bigwedge_{1 \le k \le n} [\![l_k]\!](\sigma) \in T\ \wedge \sigma \in \mathbf{\Sigma}\} \\
[\![r(\alpha_1, ..., \alpha_n)]\!](\sigma) &= r(sub(\alpha_1), ..., sub(\alpha_n)), \text{where} \\
sub(\alpha) &= \begin{cases} \sigma(\alpha), & \text{if } \alpha \in \mathbb{V} \\ \alpha, & \text{if } \alpha \in \mathbb{N} \end{cases} \\
Gr(C, T) &= \{[\![l_0]\!](\sigma) \,\text{:-}\, [\![l_1]\!](\sigma), ..., [\![l_n]\!](\sigma) \mid \\
&\quad l_0 \,\text{:-}\, l_1, ..., l_n \in C \\
&\quad \wedge \textstyle\bigwedge_{1 \le k \le n} \sigma(l_k) \in [\![C]\!](T) \wedge \sigma \in \mathbf{\Sigma}\}
\end{aligned}$$

Figure 4: Semantics of a Datalog analysis.

***Online Phase of* POLYMER.** We discuss how POLYMER uses the pruned summaries learned from the offline phase to accelerate the execution of the Datalog program on graph B. POLYMER first applies the checking function on the edge and app relations to see if it is sound to apply each available summary. For the first summary, the checking function returns *true* as there is no edge from any pruned library node (i.e., L2 or L3) to application B's nodes. For the second summary, the checking function returns *false* as e(L6, B2) exists. As a result, POLYMER concludes it is unsound to apply the second summary. Since it can reuse only the first summary, POLYMER executes the Datalog program on graph B with the modification that blocks the rule r(L2) :- r(L1), e(L1, L2). By running the Datalog program with this blocking in effect, POLYMER derives r(B0) and r(L1). At this point, as the pre-condition of the first pruned summary is satisfied, it adds r(L4) to the set of derived tuples and continues the Datalog execution. It does not use the second summary and instead re-derives the facts r(L6), r(L7), and r(L8). Finally, it concludes that application nodes B0 through B3 are reachable. Thus, POLYMER computes the same result as the original Datalog program without re-analyzing parts of the library.

3. Preliminaries

In this section, we introduce the syntax and semantics of program analyses specified in Datalog. We also extend the standard Datalog so that POLYMER is able to avoid re-computing certain analysis results for a given program.

Figure 3 shows the syntax of a Datalog analysis. A Datalog analysis C consists a set of constraints $\{c_1, ..., c_n\}$. Each such constraint $c \in C$ consists of a head literal l_0, and a body $\{l_1, \ldots, l_n\}$ which is a set of literals. A literal consists of a relation name r and a list $(\alpha_1, \ldots, \alpha_n)$ of variables or

$$
\begin{array}{rl}
(\textit{blocking set})\ B & \subseteq 2^{\mathbb{T}} \\
(\textit{instrumented analysis})\ C_B & \triangleq (C, B) \\
\\
[\![C_B]\!] \in & 2^{\mathbb{I}} \to 2^{\mathbb{T}} \\
[\![(c, B)]\!] \in & 2^{\mathbb{T}} \to 2^{\mathbb{T}} \\
[\![(C, B)]\!](I) = & \textbf{lfp}\ \lambda T.T \cup I \cup \bigcup_{c \in C} [\![(c, B)]\!](T) \\
[\![(l_0 \,\text{:-}\, l_1, ..., l_n, B)]\!](T) = & \{[\![l_0]\!](\sigma) \mid \bigwedge_{1 \le k \le n} [\![l_k]\!](\sigma) \in T \ \wedge \\
& \forall b \in B : (\exists k \in [1..n] : [\![l_k]\!](\sigma) \notin b) \\
& \wedge\ \sigma \in \mathbf{\Sigma}\}
\end{array}
$$

Figure 5: Semantics of an instrumented Datalog analysis.

constants. We call a literal containing only constants a *tuple* or *grounded literal*. The input to a Datalog analysis is a set of tuples $e \in \mathbb{I}$ which we refer to as *analysis input*. The output of a Datalog analysis is also a set of tuples. To enable avoiding the re-computation of certain analysis results, we designate a subset of output tuples $o \in \mathbb{O}$ that we are interested in, and we refer to this subset as *analysis output*.

Figure 4 shows the semantics of a Datalog analysis. It derives the output tuples as the least fixpoint (**lfp**) of the analysis constraints with respect to the input tuples. This derivation starts with the input tuples and repeatedly applies each constraint as follows until no further tuples can be derived: under a certain substitution σ which maps variables to constants, if the tuples in the body are present in the current set of derived tuples, then the head tuple is added to the set. The constraint resulting from such a substitution contains only constants and is called a *grounded constraint*. We use $Gr(C, T)$ to denote all grounded constraints used in applying Datalog analysis C to analysis input tuples T.

Figure 5 shows the syntax and semantics of our modified Datalog analysis. This instrumented Datalog analysis takes a set $B = \{T_1, ..., T_n\}$ along with a set of input tuples, and derives a set of output tuples. The difference between the semantics of the original and the instrumented Datalog analyses is that the latter avoids applying grounded constraints all of whose body tuples are in any $T_i \in B, 1 \le i \le n$. We refer to B as the *blocking set*.

4. The POLYMER Framework

POLYMER consists of an offline phase and an online phase. In the offline phase, it runs a given analysis on a training program corpus and computes facts about the shared library code; in the online phase, it runs the same analysis on a test program and reuses the analysis facts learnt in the offline phase to speed up the analysis.

4.1 The Offline Phase

Algorithm 1 describes the offline phase of POLYMER, which we refer to as OFFLINE. The input to OFFLINE is a Datalog analysis, and a set of input tuples. The output of OFFLINE is a set of summaries.

We store the learnt analysis facts in summaries. A summary is of the form of $(T_{pre}, T_{post}, \Gamma)$, which represents that if T_{pre} is derived by C on a given set of input tuples, T_{post} will also be derived. The checking function Γ maps a set of input tuples to either *true* or *false*, which represents whether the corresponding summary can be applied in the online phase with the given input tuples.

Algorithm 1 OFFLINE phase of POLYMER.

```
INPUT  $C$, a Datalog analysis; $I_{train}$, analysis input tuples.
OUTPUT  A set of summaries.
1: $preSet$ := $PickGoodPre(C, I_{train})$
2: $sumSet := \emptyset$
3: for each $T_{pre} \in preSet$ do
4:     $T_{post} := [\![C]\!](T_{pre})$
5:     $(T'_{post}, \Gamma) := PickGoodPost(C, T_{pre}, T_{post})$
6:     $sumSet := sumSet \cup \{(T_{pre}, T'_{post}, \Gamma)\}$
7: end for
8: return $sumSet$
```

The algorithm OFFLINE stats by generating the set of T_{pre}'s by invoking *PickGoodPre* on the analysis C and the input tuple set I_{train} (line 1). Then for each T_{pre}, it constructs the corresponding T_{post} and Γ as follows: OFFLINE computes the initial T_{post} by applying Datalog analysis C on T_{pre} (line 4); to further improve the computation savings enabled by the summary, it invokes *PickGoodPost* to prune away a subset of tuples in T_{post}, and generates the final T_{post} (denoted by T'_{post} in the algorithm) and the corresponding checking function Γ (line 5).

When instantiating POLYMER for a specific analysis, we require the analysis writer to provide the implementation for *PickGoodPre* and *PickGoodPost*. Intuitively, *PickGoodPre* returns the library facts that can be used to derive other facts in the library. On the other hand, *PickGoodPost* is crucial to the soundness and effectiveness of applying a summary, which we elaborate next.

4.2 Pruning Analysis Facts Soundly and Effectively

The main and interesting challenge of the offline phase is the implementation of *PickGoodPost* such that it is both sound (that is, applying the pruned summaries does not change the output of the analysis) and effective (that is, it prunes away as many tuples as possible).

For a given T_{pre}, firstly, note that we cannot remove any tuple t from $[\![C]\!](T_{pre})$ that is an output tuple ($t \in \mathbb{O}$) as this may directly change the analysis output.

For the rest of tuples, our key observation is: if tuple t does not directly participate in the derivation of any tuple outside $[\![C]\!](T_{pre})$ in the online phase, it can be safely ignored. We call such t an *intermediate tuple* of the summary for the input tuples used in the online phase. An intermediate tuple can only directly derive a tuple in the pre-condition or post-condition of the unpruned summary. In other words, an intermediate tuple is not necessary for deriving tuples outside the summary. In general, we cannot predict whether a tuple is intermediate as the offline phase lacks the knowledge about the online phase. Our solution to address this challenge is to

Algorithm 2 ONLINE phase of POLYMER.

```
INPUT  $C$, a Datalog analysis; $I_{test}$, analysis input tuples;
       $\{(T^1_{pre}, T^1_{post}, \Gamma_1), ..., (T^n_{pre}, T^n_{post}, \Gamma_n)\}$, summaries.
OUTPUT  $O$, a set of output tuples.
 1: $sumSet := \{(T^i_{pre}, T^i_{post}) \mid i \in [1, n] \wedge \Gamma_i(I_{test})\}$
 2: $blockSet := \{T_{pre} \mid \exists\, T_{post}.(T_{pre}, T_{post}) \in sumSet\}$
 3: $T_{out} := [\![(C, blockSet)]\!](I_{test})$
 4: while $sumSet \neq \emptyset$ do
 5:     $hasSummaryApplied := false$
 6:     for all $(T_{pre}, T_{post}) \in sumSet$ do
 7:         if $T_{pre} \subseteq T_{out}$ then
 8:             $T_{out} := T_{out} \cup T_{post}$
 9:             $sumSet := sumSet \setminus \{(T_{pre}, T_{post})\}$
10:             $hasSummaryApplied := true$
11:         end if
12:     end for
13:     if $\neg hasSummaryApplied$ then
14:         $(T_{pre_i}, T_{post}) :=$ Anyof$(sumSet)$
15:         $sumSet := sumSet \setminus \{(T_{pre}, T_{post})\}$
16:         $blockSet := blockSet \setminus \{T_{pre}\}$
17:     end if
18:     $T_{out} := [\![(C, blockSet)]\!](T_{out})$
19: end while
20: return $\{t \mid t \in T_{out} \wedge t \in \mathbb{O}\}$
```

remove the tuples that are likely intermediate from $[\![C]\!](T_{pre})$ and generate a checking function Γ in the offline phase, and then perform the soundness check using Γ in the online phase. If the check passes, the summary can be soundly applied in the online phase.

Definition 1 states the specification of *PickGoodPost* that captures the above observations. In Section 5, we show the instantiations of *PickGoodPost* for two different analyses that satisfy the specification.

Definition 1 (Sound Pruning)**.** For a given T_{pre}, let $(T_{post}, \Gamma) = PickGoodPost(C, T_{pre}, [\![C]\!](T_{pre}))$. We say *PickGoodPost* is a sound pruning function, if the following condition holds:

$$\forall\, t \in [\![C]\!](T_{pre}) \setminus T_{post} \,.\, t \notin \mathbb{O} \;\wedge (\forall I \subseteq \mathbb{I} \,.\, \Gamma(I) \Rightarrow (\forall\, t_0 \,\text{:-}\, t_1, ..., t_n \in Gr(C, I), \text{where } t \in \{t_1, ..., t_n\} \,.\, t_0 \in [\![C]\!](T_{pre}))).$$

4.3 The Online Phase

Algorithm 2 describes the online phase of POLYMER, which we refer to as ONLINE. The input to ONLINE is a Datalog analysis, a set of input tuples, a set of summaries. The output of ONLINE is a set of analysis output tuples.

The ONLINE algorithm starts by applying each checking function on the input tuples and constructs $sumSet$, which is the set of summaries that can be soundly applied (line 1). Then it constructs $blockSet$ which is the set of T_{pre}'s in $sumSet$ (line 2). Next, it computes the initial set of derived tuples T_{out} by running the instrumented analysis on I_{test} with C and $blockSet$ such that all grounded constraints covered by the summaries in $sumSet$ are blocked (line 3). Next, it iterates until every summary in $sumSet$ is either applied or discarded (line 4-19). In each iteration, the algorithm applies every summary whose T_{pre} is contained in T_{out} by adding its T_{post} to T_{out} (line 6-12). If no summary can be applied, to make the analysis proceed, it removes an arbitrary summary from the $sumSet$ and unblocks the grounded constraints covered by it (line 13-17). Here we only discard one summary instead of all, as with new tuples derived in the next iteration, some summaries might become applicable.

To understand why discarding a summary is sometimes necessary, consider the following example. Let S be a summary, where S is $(\{t_0, t_1, t_2\}, \{t_3, t_4\}, \Gamma)$. Here, $\{t_0, t_1, t_2\}$ is the pre-condition of summary S, and $\{t_3, t_4\}$, the post-condition. Then, $sumSet$ is $\{(\{t_0, t_1, t_2\}, \{t_3, t_4\})\}$ (line 1). Suppose currently only tuple t_0 is derived and there is an applicable grounded constraint $t_3 \,\text{:-}\, t_0$ (meaning tuple t_0 derives tuple t_3). The pre-condition of summary S belongs to $blockSet$ (line 2). Therefore the above grounded constraint is not applied and tuple t_3 is not derived. The check on line 13 is true because summary S is not applicable. At this point, no summary is applicable and tuple t_3 is not yet derived. The online phase is not able to make progress. The algorithm avoids this problem by shrinking the blocking set (lines 14-16). Now, the pre-condition of summary S, $\{t_0, t_1, t_2\}$, no longer belongs to $blockSet$. Therefore, the constraint $t_3 \,\text{:-}\, t_0$ will apply, deriving tuple t_3. In general, when no summary is applicable, one arbitrary summary is removed from $sumSet$ to unblock the grounded constraints covered by it.

At the end of the iteration, the algorithm updates T_{out} by rerunning the instrumented analysis with T_{out} as the input (line 18). Finally, it returns the set of analysis output tuples in T_{out} (line 20).

We now state the soundness of POLYMER, that is, by applying the summaries computed in the offline phase, the online phase produces the same result as the original analysis.

Theorem 2 (Soundness)**.** *Let $summaries$ = OFFLINE(C, I_{train}), where C is a Datalog analysis and I_{train} is a set of analysis input tuples. If PickGoodPost applied in* OFFLINE *is a sound pruning function, then*

$$\forall I_{test} \subseteq \mathbb{I} : \text{ONLINE}(C, I_{test}, summaries) = [\![C]\!](I_{test}) \cap \mathbb{O}.$$

Though we have only discussed the case of applying summaries learnt from one program in the online phase, it is easy to see that POLYMER can apply sets of summaries learnt from multiple programs by providing their union as an input to ONLINE.

5. Instances of POLYMER

We instantiate two different analyses using POLYMER: a context- and flow-insensitive Call-Graph Analysis similar to a class hierarchy analysis, and a context- and flow-sensitive

Domains:

$\mathbb{M}$ is a set of methods.
$\mathbb{I}$ is a set of method call sites.
$\mathbb{T}$ is a set of class types.

Input relations:

$\mathsf{dispatch}(m1:\mathbb{M}, t:\mathbb{T}, m2:\mathbb{M})$	// Method $m2$ in type t overrides method $m1$.
$\mathsf{body}(m:\mathbb{M}, i:\mathbb{I})$	// Method m contains call site i.
$\mathsf{binding}(i:\mathbb{I}, m:\mathbb{M})$	// Call site i resolves to method m.
$\mathsf{receiver}(i:\mathbb{I}, t:\mathbb{T})$	// The receiver of call site i has type t.
$\mathsf{subtype}(t1:\mathbb{T}, t2:\mathbb{T})$	// Type t_1 is a subtype of type t_2.

Output relations:

$\mathsf{rMethod}(m:\mathbb{M})$	// Method m is reachable.
$\mathsf{rInvoke}(i:\mathbb{I})$	// Call site i is reachable.
$\mathsf{target}(i:\mathbb{I}, m:\mathbb{M})$	// Method m is invoked at i.
$\mathsf{callGraph}(i:\mathbb{I}, m:\mathbb{M})$	// The call-graph, same as target.

Rules:

$\mathsf{rMethod}(m_{\text{main}}).$ (1)
$\mathsf{rInvoke}(i)$:- $\mathsf{rMethod}(m), \mathsf{body}(m, i)$ (2)
$\mathsf{target}(i, m_2)$:- $\mathsf{rInvoke}(i), \mathsf{receiver}(i, t_1), \mathsf{binding}(i, m_1), \mathsf{subtype}(t_1, t_2), \mathsf{dispatch}(m_1, t_2, m_2).$ (3)
$\mathsf{rMethod}(m)$:- $\mathsf{target}(\_, m).$ (4)
$\mathsf{callGraph}(i, m)$:- $\mathsf{target}(i, m).$ (5)

Figure 6: Call-Graph Analysis in Datalog.

// The constants $\mathbb{N}$ are divided into application specific constants $\mathcal{A}$ and library constants $\mathcal{L}$.
$\mathbb{N} = \mathcal{A} \uplus \mathcal{L}$

// $lib(t) = true$ when t is a library tuple.
$lib = \lambda r(d_1, .., d_n).\ \bigwedge_{1 \le i \le n} d_i \in \mathcal{L}$

// $con_T(t, t') = true$ when t and t' are involved in the same grounded rule applied when executing C on T.
$con_T = \lambda(t, t').\exists\ t_0 \text{ :- } t_1, ..., t_n \in Gr(C, T).$
$\{t, t'\} \subseteq \{t_0, t_1, .., t_n\}$

// $reach_T(t, t') = true$ when t and t' are reachable from each other on the derivation graph of executing C on T.
$reach_T = \lambda(t, t').(t, t') \in R^+$, where
$R = \{(t, t') \mid con_T(t, t')\}$

// $collectPre(T_1, T_2)$ returns tuples in T_1, and tuples in T_2 that reach them on the derivation graph of executing C on $T_1 \cup T_2$.
$collectPre = \lambda(T_1, T_2).T_1 \cup \{t \mid t \in T_2 \wedge$
$\exists\ t' \in T_1.reach_{(T_1 \cup T_2)}(t, t')\}$

Figure 7: Auxiliary definitions for defining *PickGoodPre* for an analysis C.

interprocedural Points-To Analysis. These analyses are specified in Datalog but they analyze programs written in Java. Instantiating POLYMER on an analysis requires defining the *PickGoodPre* and *PickGoodPost* functions. We next describe the two analyses and the instantiations of POLYMER for these.

PickGoodPre $= \lambda I.\{collectPre(\{t\}, I_{lib}) \mid t \in T_s\}$, where $T_s = \{t \mid t \in [\![C]\!](I) \wedge t.r = \mathsf{rMethod} \wedge lib(t)\}$ and $I_{lib} = \{t \mid t \in I \wedge lib(t)\}$.

PickGoodPost $= \lambda(T_1, T_2).(T_3, \Gamma)$, where $T_3 = \{t \mid t \in T_2 \wedge t.r = \mathsf{callGraph}\}$ and $\Gamma = \lambda I.\ \forall \mathsf{subtype}(t_1, t_2) \in T_2.(\forall\ \mathsf{subtype}(t_1, t_3) \in I.\mathsf{subtype}(t_1, t_3) \in T_2)$.

Figure 8: Specification of *PickGoodPre* and *PickGoodPost* for Call-Graph Analysis.

Call-Graph Analysis. Figure 6 specifies the Datalog rules of our Call-Graph Analysis. This analysis computes the call-graph for a given program in the form of the relation callGraph. This relation contains tuples (i, m) where m is any method reachable from the main method, that could possibly be a target of the call site i. Given that a method m, defined in an object of type t, is called at a call-site i, to compute all possible call targets for i, the analysis first gets all subtypes of t. From these subtypes, it chooses the methods that override the definition of m, as possible call targets of the call site i (Rule (3) of Figure 6).

We next discuss the instantiation of functions *PickGoodPre* and *PickGoodPost* for the Call-Graph Analysis. Intuitively, if a method at the boundary of shared code is reached, it is very likely that the call-graph rooted at this method is the same across programs using the shared code. Therefore, we direct POLYMER to generate summaries capturing call-graphs rooted at such methods by appropriately defining *PickGoodPre*.

The function *PickGoodPre* generates the pre-conditions to capture such summaries. For each method m deemed reachable in the shared code, the pre-condition includes the fact reachable(m). It also includes all the constant analysis facts that it might need to compute the call-graph rooted at this method. Given the derivation graph of the analysis on a training program, this computation is very straightforward.

Figure 7 defines function $collectPre$ declaratively, which is used to compute the set of constants required to compute the call-graph rooted at any given method. Figure 8 gives the actual definition of *PickGoodPre*. Here, the set T_s captures all the reachable library methods. Next, *PickGoodPre* applies $collectPre$ to compute the pre-condition for each such method.

We next describe the instantiation of *PickGoodPost* given in Figure 8. The main task of this function is to specify (1) what tuples to prune away from a given summary, and, (2) the checking function that will ensure that it is sound to use the summary pruned this way. For the Call-Graph Analysis, we observe that all derived tuples other than the tuples of the relation callGraph represent intermediate computations and therefore can be pruned away. We retain only the tuples of relation callGraph in the post-condition of the summary, as indicated by the definition of *PickGoodPost* in Figure 8.

Reusing a summary pruned as described above is sound only if, whenever the method m in the pre-condition of the summary is deemed reachable in the online phase ($\mathsf{reachable}(m)$ is derived), the call-graph rooted at this method is the same as the one in the post-condition of the summary. Since the call-graph is entirely over the shared code, the above condition will be only violated if an application method is included in this call-graph during the online phase. This will only happen if an application class overrides a library method in this call-graph. To enforce such a condition conservatively, *PickGoodPost* generates a checking function that examines whether any application class in the online phase overrides a library class whose method is in this call-graph.

Points-To Analysis. Figure 9 specifies the Datalog rules of our Points-To Analysis. This analysis is adapted from the points-to analysis specified in [16], which computes the points-to information at each program point. It is a summary-based context- and flow-sensitive analysis that applies the IFDS algorithm [20]. We instantiate two variants of this analysis: one that uses a less precise pre-computed call-graph, and another that uses a more precise call-graph constructed on-the-fly. In both these variants, the computation of the points-to information at all program points remains the same. It differs only in the way in which the call targets of a method call are determined. Figure 9 highlights the commonalities and the differences. The light-gray boxes contain the parts of the analysis specification present only when the call-graph is pre-computed. The dark-gray boxes contain the parts present only when the call-graph is constructed on-the-fly. The rest is common to both.

The rules for both variants of Points-To Analysis are mostly common. For simplicity, we elide the rules encoding the concrete transfer functions and only show the rules related to the IFDS algorithm in Figure 9: Rule (1) starts the analysis at the entry of the main method with the initial abstract state; Rule (2) applies a transfer function to compute the outgoing abstract state from the incoming abstract state at any given program point; Rule (3) computes the abstract state at a method entry by transferring the points-to information from the arguments of a method call to the formal parameters of the target method definition; Rule (4) generates a method summary which captures the relation between the abstract state at the method entry and that at the method exit; Rule (5) computes the abstract state after a call site by applying the summary of the invoked method.

Rule (6) is present only when the call-graph is pre-computed: it just copies the input relation call into relation cg. Rule (7) is present only when the call-graph is constructed on-the-fly: it computes the context-sensitive call-graph at program point p. We explain this rule in more detail. Suppose, the program point p contains a method call $x.foo()$. Then, the relation extract gives all possible allocation sites h, pointed-to by x, in the abstract state s_2. The corresponding

Domains:
$\mathbb{P}$ is a set of program points.
$\mathbb{M}$ is a set of methods.
$\mathbb{S}$ is a set of abstract states. For points-to analysis, they are points-to information at each program point.

$\mathbb{H}$ is a set of allocation sites.
$\mathbb{T}$ is a set of class types.

Input relations:

$\mathsf{head}(m : \mathbb{M}, p : \mathbb{P})$	// Program point p is the entry of method m.
$\mathsf{tail}(m : \mathbb{M}, p : \mathbb{P})$	// Program point p is the exit of method m.
$\mathsf{next}(p : \mathbb{P}, q : \mathbb{P})$	// The successor of program point p is q.
$\mathsf{itrans}(p : \mathbb{P}, m : \mathbb{M}, s_1 : \mathbb{S}, s_2 : \mathbb{S})$	// Transfer function to pass // parameters when m is called at p.
$\mathsf{rtrans}(p : \mathbb{P}, m : \mathbb{M}, s_1 : \mathbb{S}, s_2 : \mathbb{S})$	// Transfer function for // method return when m is called at p.
$\mathsf{trans}(p : \mathbb{P}, s_1 : \mathbb{S}, s_2 : \mathbb{S})$	// Transfer function for statement at p.

$\mathsf{call}(p : \mathbb{P}, m : \mathbb{M})$	// Program point p invokes method m.

$\mathsf{type}(h : \mathbb{H}, t : \mathbb{T})$	// An object allocated at site h // has type t.
$\mathsf{virtual}(p : \mathbb{P}, m : \mathbb{M})$	// Program point p invokes // virtual method m.
$\mathsf{dispatch}(n : \mathbb{M}, t : \mathbb{T}, m : \mathbb{M})$	// Method m in type t overrides // method n.
$\mathsf{extract}(p : \mathbb{P}, s : \mathbb{S}, h : \mathbb{H})$	// Allocation site h is pointed to // by the receiver of the virtual method call at program point p.

Intermediate relations:

$\mathsf{cg}(p : \mathbb{P}, s : \mathbb{S}, m : \mathbb{M})$	// Call graph at program // point p in abstract state s.

Output relations:

$\mathsf{ppointsTo}(p : \mathbb{P}, s_1 : \mathbb{S}, s_2 : \mathbb{S})$	// Path edges.
$\mathsf{mpointsTo}(m : \mathbb{M}, s_1 : \mathbb{S}, s_2 : \mathbb{S})$	// Summary edges.

Rules:

$$\mathsf{ppointsTo}(p, s_{\text{init}}, s_{\text{init}}) \;\text{:-}\; \mathsf{head}(m_{\text{main}}, p). \quad (1)$$

$$\mathsf{ppointsTo}(q, s_1, s_3) \;\text{:-}\; \mathsf{ppointsTo}(p, s_1, s_2), \mathsf{trans}(p, s_2, s_3), \mathsf{next}(p, q). \quad (2)$$

$$\mathsf{ppointsTo}(q, s_3, s_3) \;\text{:-}\; \mathsf{ppointsTo}(p, s_1, s_2),\ \mathsf{cg}(p, s_2, m), \mathsf{itrans}(p, m, s_2, s_3),\ \mathsf{head}(m, q). \quad (3)$$

$$\mathsf{mpointsTo}(m, s_1, s_2) \;\text{:-}\; \mathsf{ppointsTo}(p, s_1, s_2),\ \mathsf{tail}(m, p). \quad (4)$$

$$\mathsf{ppointsTo}(r, s_1, s_5) \;\text{:-}\; \mathsf{ppointsTo}(p, s_1, s_2),\ \mathsf{cg}(p, s_2, m), \mathsf{itrans}(p, m, s_2, s_3),\ \mathsf{next}(p, r), \mathsf{mpointsTo}(m, s_3, s_4), \mathsf{rtrans}(p, m, s_4, s_5). \quad (5)$$

$$\mathsf{cg}(p, s_2, m) \;\text{:-}\; \mathsf{ppointsTo}(p, s_1, s_2), \mathsf{call}(p, m). \quad (6)$$

$$\mathsf{cg}(p, s_2, m) \;\text{:-}\; \mathsf{ppointsTo}(p, s_1, s_2), \mathsf{extract}(p, s_2, h), \mathsf{type}(h, t), \mathsf{virtual}(p, n), \mathsf{dispatch}(n, t, m). \quad (7)$$

Figure 9: Points-To Analysis in Datalog. Domains, relations and rules in light gray boxes are present only when call-graph is pre-computed and the ones in dark gray boxes are present only when the call-graph is constructed on-the-fly.

PickGoodPre $= \lambda I.\{collectPre(\{t\}, I_{lib}) \mid t \in T_s\}$, where $T_s = \{t \mid t \in [\![C]\!](I) \wedge lib(t) \wedge t = \mathsf{ppointsTo}(p, s, s)$ where $\exists\ \mathsf{head}(m, p).\mathsf{head}(m, p) \in I_{lib}\}$ and $I_{lib} = \{t \mid t \in I \wedge lib(t)\}$.

PickGoodPost $= \lambda(T_1, T_2).(T_3, \Gamma)$, where $T_3 = \{t \mid t \in T_2 \wedge t.r = \mathsf{mpointsTo}\}$ and $\Gamma = \lambda I.\ \forall \mathsf{call}(p, m_1) \in T_2.(\forall\ \mathsf{call}(p, m_2) \in I.\mathsf{call}(p, m_2) \in T_2)$.

Figure 10: Specification of *PickGoodPre* and *PickGoodPost* for Points-To Analysis with pre-computed call-graph.

PickGoodPre $= \lambda I.\{collectPre(\{t\}, I_{lib}) \mid t \in T_s\}$, where $T_s = \{t \mid t \in [\![C]\!](I) \wedge lib(t) \wedge t = \mathsf{ppointsTo}(p, s, s)$ where $\exists\ \mathsf{head}(m, p).\mathsf{head}(m, p) \in I_{lib}\}$ and $I_{lib} = \{t \mid t \in I \wedge lib(t)\}$.

PickGoodPost =
$\lambda(T_1, T_2).(T_3, \Gamma)$, where $T_3 = \{t \mid t \in T_2 \wedge t.r = \mathsf{mpointsTo}\}$ and $\Gamma = \lambda I.\ \forall\ \mathsf{type}(h, t_1), \mathsf{dispatch}(m, t_1, m_1) \in T_2.$
$(\forall\ \mathsf{type}(h, t_2), \mathsf{dispatch}(m, t_2, m_2) \in I.$
$\mathsf{type}(h, t_2), \mathsf{dispatch}(m, t_2, m_2) \in T_2)$.

Figure 11: Specification of *PickGoodPre* and *PickGoodPost* for Points-To Analysis with on-the-fly call-graph construction.

types t for all these allocation sites, is given by relation type. The fact that program point p invokes the virtual method $foo()$ comes from the input relation virtual. Finally, all possible call targets are the methods m that override $foo()$ in the types t: this comes from the input relation dispatch.

The instantiations of *PickGoodPre* and *PickGoodPost* for the two variants of Points-To Analysis are given in Figures 10 and 11. The definition of *PickGoodPre* remains the same for both variants and we discuss it below. Similar to the Call-Graph Analysis, we observe that a given library method m analyzed in a given abstract context, will likely produce the same analysis facts during the analysis of an unseen program if the analysis reaches m in the same abstract context. Based on such observation, *PickGoodPre* constructs the set T_s by taking all the ppointsTo tuples encoding the entry states of all library methods. Similar to the Call-Graph Analysis, *PickGoodPre* completes the pre-condition for each tuple in T_s by applying $collectPre$ to collect all the constants needed to compute the analysis facts for the corresponding method. Therefore, *PickGoodPre* constructs a pre-condition for each abstract context in which a method is analyzed.

For each such pre-condition corresponding to a method, say m, POLYMER constructs a post-condition. This post-condition comprises all possible tuples that can be derived by applying the rules of Figure 9 to the tuples in the pre-condition. That is, such a post-condition will comprise ppointsTo tuples at all program points in the method m, and at all program points in all methods called transitively by m. Secondly, it will comprise mpointsTo tuples associated with method m, and all methods called transitively by m. Lastly, it will comprise cg tuples associated with all program points that are invoke statements in method m, and in all methods called transitively by m. POLYMER then applies the function *PickGoodPost* to prune such a post-condition.

We next discuss the definition of *PickGoodPost* in Figures 10 and 11. From rule (5) in Figure 9, it is clear that a post-condition needs to retain only mpointsTo facts. This is because only mpointsTo facts associated with a method m are used after the analysis of method m is complete. The definition of *PickGoodPost* captures this observation by pruning away tuples belonging to other relations, and retaining only the tuples of the mpointsTo relation. For both variants of Points-To Analysis, the pruned summaries need to retain only the mpointsTo tuples in the post-condition.

Again, reusing such a pruned summary is sound only if, for a given method whose entry state is captured by the summary pre-condition, the abstract state computed at the method exit in the online phase (mpointsTo fact) is the same as the abstract state in the post-condition of the pruned summary. This condition will be violated only if the call-graph rooted at the method under consideration is not the same in the online and the offline phases. In other words, this condition means that for any given method m, all other methods called transitively by m, must be the same in the online and offline phases. In addition, a check on this condition automatically excludes reusing summaries for methods that contain application callbacks. This is because the condition check will always fail while analyzing such methods, since these methods (transitively) contain calls to application methods that will never be the same in the offline and online phases.

For each summary, *PickGoodPost* generates a checking function that ensures this condition. Here, there is a difference in the checking function generated for the two variants of Points-To Analysis. When the call-graph is pre-computed, the checking function examines the call relation used in the online phase (Figure 10). When the call-graph is constructed on-the-fly, we observe by looking at rule (7), that the set of tuples of the type and dispatch relations used in that rule completely determine the call-graph at that program point. Therefore, the checking function examines the type and dispatch relations in the online phase (Figure 11).

Discussion. Though we observe from these instantiations of POLYMER that *PickGoodPre* and *PickGoodPost* are not very difficult to specify, they do require some insight into the analyses. POLYMER does not require the most optimal definitions of these functions. More naive definitions will only cause POLYMER to achieve more modest speedups.

Another point that these instantiations of POLYMER illustrate is that POLYMER records summaries in the same abstract domain as the original analysis. These summaries are deduced from grounded facts encountered during the actual

	brief description	# classes		# methods		bytecode (KB)		source (KLOC)	
		app	total	app	total	app	total	app	total
`antlr`	generates parsers and lexical analyzers	109	1,091	873	7,220	81	467	26	224
`avrora`	AVR microcontroller simulator	78	1,062	523	6,905	35	423	16	214
`bloat`	Java bytecode analysis/optimization tool	277	1,269	2,651	9,133	195	586	59	258
`chart`	plots graphs and render them as PDF	181	1,756	1,461	11,450	101	778	53	366
`hsqldb`	relational database engine	189	1,341	2,441	10,223	190	670	96	322
`luindex`	document indexing tool	193	1,175	1,316	7,741	99	487	38	237
`lusearch`	text searching tool	173	1,157	1,119	7,601	77	477	33	231
`pmd`	Java source code analyzer	348	1,357	2,590	9,105	186	578	46	247
`sunflow`	photo-realistic image rendering system	165	1,894	1,328	13,356	117	934	25	419
`xalan`	XML to HTML transforming tool	42	1,036	372	6,772	28	417	9	208

Table 1: Benchmark characteristics. The "total" and "app" columns report numbers with and without counting shared code, respectively. Shared code denotes the JDK library.

analysis of a training program. Since POLYMER processes grounded facts to deduce summaries, it is not hindered by higher-order functions or dynamically allocated memory.

6. Empirical Evaluation

We implemented POLYMER in a tool for accelerating analyses specified in Datalog for Java programs. It uses Chord [18] as the Java bytecode analysis front-end and bddbddb [29] as the Datalog solver. POLYMER can be configured to run in the `Offline` phase or the `Online` phase. In the `Offline` phase, it takes as input a corpus of training programs and a specification of what constitutes shared code. It learns analysis facts over the shared code and stores them in a persistent database. In the `Online` phase, it uses the learnt facts to accelerate the analysis of the input program.

6.1 Experimental Setup

We evaluate POLYMER on the two analyses described earlier, using ten programs from the DaCapo suite, shown in Table 1.

These ten programs are diverse, widely-used programs whose shared code is primarily the Java standard library (JDK). We therefore designate the JDK as shared code for these programs. These programs are 9-96 KLOC and 208-419 KLOC in size, excluding and including the size of JDK code reachable from them, respectively.

Our evaluation addresses two main questions:

1. How much can POLYMER speed up the analysis of a program when provided with training data? (Section 6.2).
2. How sensitive is POLYMER's acceleration to variations in training data? (Section 6.3).

All results were obtained using Oracle HotSpot JVM 1.6 on Linux machines with 3.0 GHz processors and 16 GB RAM.

6.2 Speedup Measurements

Methodology. To measure speedup, we compare POLYMER's performance in three settings of varying training data:

- `Baseline`: POLYMER analyzes the input program from scratch. No training data is available. This execution is the same as the standard formulation of the analysis (in Datalog). We do not modify the analysis specification in any way in order to compute `Baseline` metrics.
- `Ideal`: All possible learnt facts over shared code that are sound for cross-program use are available to POLYMER. In this setting, the perfect training data is available to POLYMER. Therefore, POLYMER is able to achieve maximum acceleration while analyzing the input program, giving an upper bound on the speedup achievable by POLYMER. POLYMER simulates this setting by two steps:
  - training on a program and recording all learnt facts for the shared code that can be reused soundly across programs; and
  - reusing these learnt facts on the same program.
- `Actual`: Only facts learnt from the training corpus are available to POLYMER. This setting captures the real-life scenario in which POLYMER is used. Therefore, our evaluation compares POLYMER's performance in this setting to that in both the previous settings. The comparison between `Actual` and `Ideal` settings gives an indication of how close the speedup provided by real-life training data is to the speed-up provided by ideal training data.

In our experiments, the training corpus used in the `Actual` setting consists of all benchmarks in the same suite with the exception of the benchmark being tested.

The results of our measurements are shown in Figure 12. For each analysis, the figure shows two graphs:

- **Speedup:** The speedups in `Ideal` and `Actual` settings over the running time in `Baseline` setting. The raw running time in `Baseline` setting is shown at the top of the bars, for each benchmark.
- **Reduction in facts computed:** The ratio of analysis tuples computed in `Ideal` and `Actual` settings to that computed in `Baseline` setting. The total number of analysis tuples computed in `Baseline` setting is shown at the top of the bars, for each benchmark.

Results. For Call-Graph Analysis, the `Actual` setting yields speedups ranging from 1.9× to 3.4× with an average of 2.6×. For Points-To Analysis with a pre-computed call-graph, these speedups range from 1.3× to 11× with an average

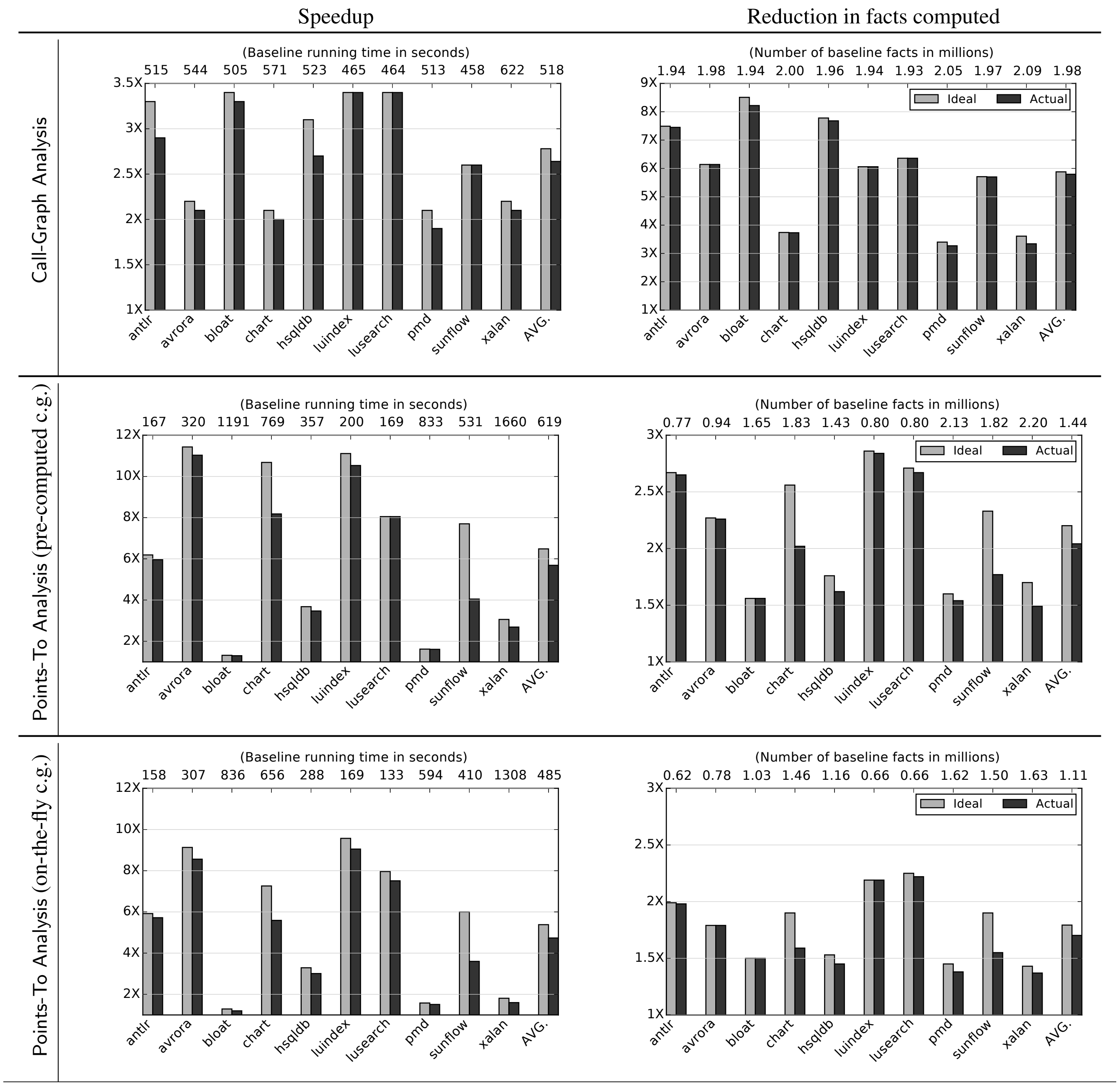

Figure 12: Speedup achieved by POLYMER over `Baseline` and the corresponding reduction in analysis facts computed.

of 5.7×. For Points-To Analysis with call-graph constructed on-the-fly, the speedups range from 1.2× to 9.1× with an average of 4.7×. These numbers are corroborated by the reduction in computed analysis facts plotted alongside the speedup graphs for each analysis. Intuitively, the reduction in computed analysis facts captures the reduction in work done. While there is indeed a correlation between the reduction in work done and speedup, however, the number of tuples computed is not an exact measure of the work done. The actual work done depends on the internals of the Datalog solver.

All measurements of running time are averaged over three runs and the variability across these measurements is minimal (under 5%). Since the number of computed analysis tuples is a count of tuples, it is exact and there is no variability in these measurements.

The performance of POLYMER in the `Actual` case compares favorably to that in the `Ideal` case in most cases. The difference between the speedups observed in these two cases is explained by the fact that not all learnt facts required by the input program may be provided by the training programs. On average, the drop in speedup from `Ideal` to `Actual` is very minimal. This is expected as the DaCapo benchmarks use the JDK for common features like containers and I/O. Therefore, the likelihood of the training set providing the necessary learnt facts is high. For `pmd` and `bloat`, absolute speedups are modest because they have among the highest ratios of application to library code, but the speedup for `Actual` relative to `Ideal` is similar to other benchmarks.

The speedups produced for Points-To Analysis when the call-graph is constructed on-the-fly are lesser than the speedups produced when the call-graph is pre-computed. Correspondingly, the decrease in the number of analysis facts computed also shows a similar trend. The reason is that fewer summaries get reused across programs. Since condition checking is more involved for the points-to analysis that constructs the call-graph on-the-fly, summaries are less likely to match across programs. Another observation is that the raw metrics in `Baseline` setting (running time and the number of analysis facts computed) are smaller when the call-graph is constructed on-the-fly. This is because lesser code is analyzed since on-the-fly call-graph is more precise.

We observe that the speedup achieved by POLYMER is better for Points-To Analysis than for Call-Graph Analysis. This is because the complexity of Points-To Analysis is super-linear in program size, and therefore it benefits more than Call-Graph Analysis from available training data.

In conclusion, we expect Polymer to scale for any analysis that scales when expressed in Datalog. However, the speedups achieved are contingent on appropriate definitions for *PickGoodPre* and *PickGoodPost* as well as the availability of suitable training data.

6.3 Robustness Measurements

This section evaluates the sensitivity of POLYMER's acceleration to variations in training data.

Methodology. For the analysis of a given program, the available training data could vary along two dimensions. In one dimension, the training data covers specific modules but not the entire breadth of the exercised shared code. In this case, we say there is variation in the *functionality* of the training data. To measure sensitivity to this kind of variation, we make POLYMER analyze an input program multiple times, each time using training data from a *single* other program in the same benchmark suite. Since different training programs can exercise different modules of shared libraries, this simulates the variation in functionality of the training data.

Libraries typically have many layers of abstraction, and training data at different layers of the shared code may be available to POLYMER, causing it to reanalyze the shared code to different depths. This is the other dimension in which available training data could vary; we refer to this as the variation in *abstraction layers*. We give an example below to illustrate the effect of summaries at different layers of abstraction.

Suppose there is a program A that calls a library method $foo()$ which in turn calls another library method $bar()$. Suppose further that the summaries for library methods $foo()$ and $bar()$ are available to the analysis of program A. At the program point where method $foo()$ is invoked, POLYMER recognizes that a summary for method $foo()$ is available and uses it (assume that the summary passes the condition checks). Thereby, POLYMER prevents the re-analysis of methods $foo()$ and $bar()$. Note that the summary for method $bar()$ is never used even though it is available. Now suppose, only the summary for method $bar()$ is available (in other words, a summary at a deeper level of abstraction is available). Then POLYMER will re-analyze method $foo()$ and use the summary for method $bar()$. In this case, POLYMER prevents the re-analysis of only method $bar()$.

To simulate the variation in abstraction layers, we make POLYMER analyze an input program multiple times. The first time, we provide POLYMER the original set of learnt facts corresponding to the `Ideal` case in Section 6.2, but each subsequent time, we remove all those analysis facts that were used by POLYMER in the previous run.

The results of the experiments to measure robustness along the two dimensions discussed above, are shown in Figure 13. For each analysis, graphs showing measurements along both dimensions are placed beside one another.

Variation in functionality. We show measurements for this dimension as box plots of the speedup achieved when POLYMER analyzes an input program with training data from different programs. We drill down to analyze the box plot in the column labeled `avrora` for Points-To Analysis with a pre-computed call-graph. It shows the speedup when `avrora` is

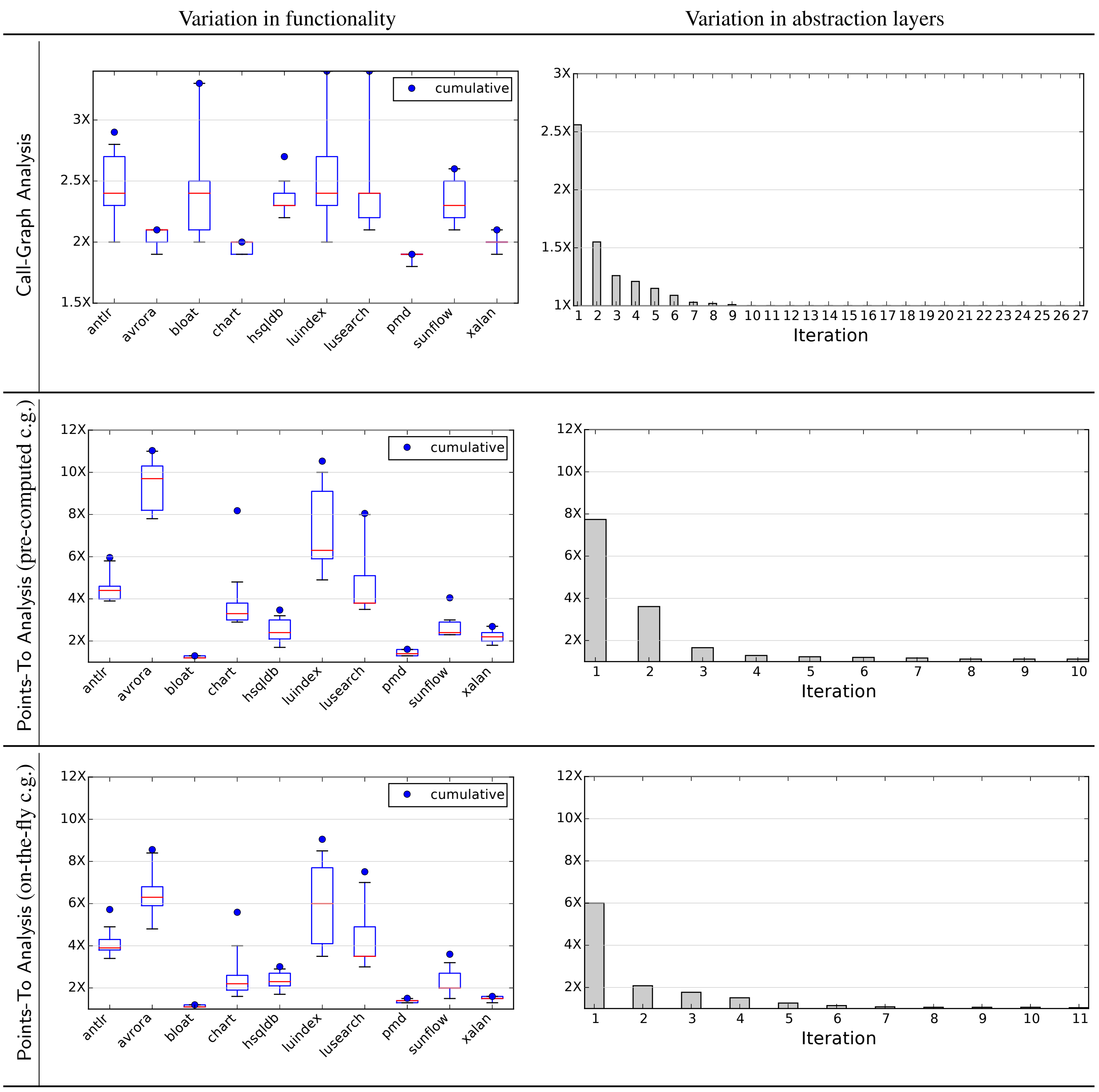

Figure 13: (1) Variation in functionality: Speedup of Points-To Analysis on each program by using different training programs. (2) Variation in abstraction layers: Reduction in speedup of Points-To Analysis with training data from successively deeper abstraction layers of shared code, for one benchmark (`sunflow`).

analyzed with training data from each of the other nine DaCapo benchmarks. The box plot says that most training programs give a speedup between 8.2× and 10.3×. These numbers, indicated by the bottom and top boundaries of the box, correspond to the first and third quartiles of all the speedups. The whiskers mark the extremes: at the lower end is a training program that provides 7.8× speedup and at the higher end is one that provides 11× speedup. The red line within the box indicates the median speedup. The blue dot on top, specified as cumulative, is the speedup when `avrora` is analyzed using the training data from all the nine other benchmarks combined. This speedup is the same as in the `Actual` setting described in Section 6.2. In the case of `avrora`, the blue dot coinciding with the top whisker indicates that the single best training program is providing training data that is almost as good as the cumulative training data.

These plots reveal that for a particular input program under analysis, not all training programs are equivalent in terms of the training data they provide and the speedup they enable. Typically, there is some training program that stands out due to similar library usage as the current program under analysis. However, we also see that even if the best training program were not available, there are other training programs that would provide comparable, if not equal, speedup. These measurements also indicate how far away the outlier training programs are, in terms of the usefulness of the training data they provide. They also give a quick estimate of the contribution to the training data by individual programs, compared to the cumulative contribution by all programs.

Variation in abstraction layers. The measurements for this dimension are bar graphs of the speedup achieved by POLYMER when provided with training data for successively deeper layers of abstraction of the shared code. The bar graphs in Figure 13 show the effect of summaries at different abstraction layers. All the bar graphs show this for program $sunflow$ (they are similar for other programs). We see that the first bar for iteration 1, corresponds to the speedup in `Ideal` setting. In the subsequent iterations, the summaries available to POLYMER are at deeper and deeper levels of abstraction. The drop in speedup over these iterations shows that denying POLYMER training data for the higher layers of abstraction in the shared code dramatically diminishes performance gains for all the analyses. Intuitively, retaining only analysis facts for deeper abstraction layers of the shared code does not yield significant speedup. This also highlights the value of pruning in POLYMER because pruning increases the reusability of summaries. Therefore, there is a greater chance that pruned summaries at higher levels of abstraction get used.

7. Related Work

Most work on reusing analysis results is based on interprocedural analysis techniques, since procedures provide a natural interface to summarize analysis results.

Complete summaries. Classical work on interprocedural analysis [20, 23, 28] computes a complete summary of each procedure in a program, and applies each summary to analyze the procedure at each call site. For many analysis domains, including relational domains, such summaries are difficult to represent compactly or infer efficiently [27, 28]. Yorsh et al. [32] generalize top-down summaries by replacing explicit summaries with symbolic representations, thus increasing reuse without losing precision. Ball et al. [4] generalize top-down summaries by encoding the transfer functions using BDDs. Other techniques generalize top-down summaries by pruning irrelevant parts of the calling context for points-to analysis [8] and shape analysis [21].

These approaches focus on high reusability but incur a high cost for instantiating the summaries at the time of reuse. In contrast, POLYMER achieves reusability of analysis facts by descending deeper into shared code in the offline phase, in return for incurring a negligible cost for instantiating the summaries and re-analyzing the shallower parts of the shared code in the online phase.

Partial summaries. Godefroid et al. [13] propose constructing partial summaries to summarize program behaviors relevant to a particular trace, that is a potential counterexample to a property, under an abstraction maintained by the analysis. The analysis facts that POLYMER learns are not computed in response to a failure by an analysis, but instead are computed to collect information from the completed run of a successful analysis.

Zhang et al. [33] propose a framework to combine top-down and bottom-up interprocedural analysis in order to gain the benefits of efficient computation and instantiation of top-down summaries with effective reuse of bottom-up summaries. Their approach requires the analysis writer to specify both the top-down and bottom-up analysis.

Partial transfer functions [17, 30] summarize the input/output behavior for only a subset of a procedure or region to speedup the analysis of the surrounding region or program. Their motivation is to circumvent the drawbacks of constructing complete summaries, especially in the presence of higher-order functions and complex transfer functions.

In all of the above approaches, the summaries are not reusable across programs, as they assume the same vocabulary of base facts when building and applying summaries.

Summarizing libraries. Rountev et al. [22, 31] propose an approach to summarize library code independent of any client code. They address the general class of interprocedural distributive environment (IDE) dataflow problems but use a graph representation of dataflow summary functions that need to be instantiated at every call site. Moreover, this approach is only able to handle call sites with single target methods as determined by a call-graph analysis.

Ali et al. [1, 2] propose a technique to over-approximate library code. Their approach creates an application-only call-graph in which the entire library is abstracted by a single method denoted *library*. All calls in the application to the library have a call-graph edge to this node. The call-graph edges from the *library* node to the application nodes, that represent callbacks, are determined precisely by analyzing the points-to information for the library.

Pre-processing libraries. Smaragdakis et al. [25] propose an optimization technique in which the source program is transformed so that it is optimized for a flow-insensitive points-to analysis. Analysis of the original and the transformed programs yields the same points-to facts. The source-level transformation is done by a pre-analysis that reasons about the flow of points-to facts. This approach makes it possible to transform large libraries once and for all, thereby optimizing subsequent whole-program points-to analyses.

Allen et al. [3] propose a demand- and query-driven approach for points-to analysis that also involves source program transformation. They employ static program slicing and compaction to reduce the input program to a smaller program that is semantically equivalent for the points-to queries under consideration. Whole-program flow-insensitive points-to analysis is then performed on this smaller, transformed program yielding the results for the points-to queries.

Oh et al. [19] propose using an existing codebase of programs to learn the the parameters to be used for a parametric static analysis. These parameters affect the analysis precision-cost tradeoff. They use Bayesian optimization to efficiently learn these parameters which are then used when analyzing unseen programs. In contrast, POLYMER uses the existing codebases to learn the analysis summaries.

8. Conclusion

Scaling program analyses to large programs is an ongoing challenge. In this paper, we proposed POLYMER, an analysis optimizer that addresses this problem in the common scenario where such programs share large modules of code. POLYMER operates by reusing analysis results for shared code across programs. POLYMER consists of two stages: an offline stage, in which it learns analysis facts over shared code from a corpus of training programs, followed by an online stage, in which it reuses the learnt facts to accelerate the analysis on new programs. Crucial to POLYMER's effectiveness is a pruning specification provided by the user that dictates how to discard intermediate analysis facts about shared code in a manner that yields performance gains without compromising soundness. We demonstrated that POLYMER achieves average speedups of 2.6× for a call-graph analysis and 5.2× for a points-to analysis, when applied to Java programs containing 208-419 KLOC.

Acknowledgments

We thank the referees for helpful feedback. This work was supported by DARPA under agreement #FA8750-15-2-0009, NSF awards #1253867 and #1526270, a Facebook Fellowship, and a Microsoft Fellowship. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright thereon.

A. Appendix

We provide the proof for Theorem 2 in this appendix. We start with the definitions of some utility functions followed by the proof of the theorem. The proof uses Lemmas 7, 8, 9, 10 and 11. The proofs for these lemmas follow the proof of the theorem.

Definition 3 (Summary Accessors)**.** If $(T_{pre}, T_{post}, \Gamma)$ is a summary, then we define the following accessor functions:

$$\begin{aligned} pre(T_{pre}, T_{post}, \Gamma) &= T_{pre} \\ post(T_{pre}, T_{post}, \Gamma) &= T_{post} \\ fpost(T_{pre}, T_{post}, \Gamma) &= [\![C]\!](T_{pre}) \\ chkfun(T_{pre}, T_{post}, \Gamma) &= \Gamma. \end{aligned}$$

Definition 4 (Constraint Accessors)**.** If c is a Datalog constraint of the form $l_0 \,{:}\text{-}\, l_1, ..., l_n$, then we define $cbody(c) = \{l_1, ..., l_n\}$ and $chead(c) = \{l_0\}$.

Definition 5 (Apply Functions)**.** If $aSet$ is a set of summaries where each summary of the form $(T_{pre}, T_{post}, \Gamma)$, then we define two functions:

$$\begin{aligned} blk(aSet) &= \{T_{pre} \mid (T_{pre}, T_{post}, \Gamma) \in aSet\} \\ load(aSet) &= \bigcup\{T_{post} \mid (T_{pre}, T_{post}, \Gamma) \in aSet\}. \end{aligned}$$

Definition 6 (Applicable Summary)**.** If sum is a summary, then we call it as an *applicable* summary if and only if $chkfun(sum)(I_{test}) \;\wedge\; pre(sum) \subseteq [\![C]\!](I_{test})$ where $I_{test} \subseteq \mathbb{I}$.

We assume the following propositions and omit their proof for brevity.

$$\begin{aligned}
&B_{blk1} \subseteq B_{blk2} \Rightarrow [\![C, B_{blk2}]\!](T_1) \subseteq [\![C, B_{blk1}]\!](T_1) && (P1)\\
&T_1 \subseteq T_2 \Rightarrow [\![C, B_{blk}]\!](T_1) \subseteq [\![C, B_{blk}]\!](T_2) && (P2)\\
&T_1 \subseteq T_2 \Rightarrow [\![C]\!](T_1) \subseteq [\![C]\!](T_2) && (P3)\\
&T_1 \subseteq [\![C, B_{blk}]\!](T_1) && (P4)\\
&[\![C, \emptyset]\!](T_1) = [\![C]\!](T_1) && (P5)\\
&[\![C, B_{blk}]\!]([\![C, B_{blk}]\!](T_1)) = [\![C, B_{blk}]\!](T_1) && (P6)\\
&T_1 \subseteq [\![C]\!](T_2) \Rightarrow [\![C]\!](T_1) \subseteq [\![C]\!](T_2) && (P7)\\
&T_1 \subseteq T_2 \Rightarrow [\![C]\!](T_2) = [\![C]\!](T_2 \cup T_1) && (P8)
\end{aligned}$$

Theorem 2. *Let $summaries$ = OFFLINE(C, I_{train}), where C is a Datalog analysis and I_{train} is a set of analysis input tuples. If PickGoodPost applied in* OFFLINE *is a sound pruning function, then*

$$\forall I_{test} \subseteq \mathbb{I} : \quad \text{ONLINE}(C, I_{test}, summaries) = [\![C]\!](I_{test}) \cap \mathbb{O}.$$

Proof. Let $summaries = \{sum_1, ..., sum_n\}$ where each element is of the form $(T_{pre}, T_{post}, \Gamma)$. Let $I_{test} \subseteq I$. Lines 1-2 of ONLINE define (1.1) $sumSet^0 = \{sum \mid sum \in summaries \;\wedge\; (chkfun(sum))(I_{test})\}$ and (1.2) $B^0_{blk} = blk(sumSet^0)$.

By line 3 of ONLINE, we have (2) $T^0_{out} = [C, B^0_{blk}](I_{test})$. Let S be the set of summaries chosen by ONLINE to apply. (3) $S^0 = \emptyset$.

The kth iteration, $k \geq 1$, of the while loop (lines 4-19 of ONLINE) computes (4),(5) and (6) as shown below.

(4) The kth iteration of the while loop chooses a subset of $sumSet^{k-1}$ to process. Let that subset be $pSet^k$. By the condition check on line 7 of ONLINE, either (4.1) or (4.2) below hold but both compute $pSet^k = \{sum_1, ..., sum_m\} \subseteq sumSet^{k-1}$.

(4.1) lines 7-11 of ONLINE compute: (4.1.1) $T^k_{load} = load(pSet^k)$, (4.1.2) $B^k_{blk} = B^{k-1}_{blk}$ and (4.1.3) $S^k = S^{k-1} \cup pSet^k$.

(4.2) lines 13-17 of ONLINE compute: (4.2.1) $T^k_{load} = \emptyset$, (4.2.2) $B^k_{blk} = B^{k-1}_{blk} \setminus \{\{T_{pre}\} \mid (T_{pre}, T_{post}, \Gamma) \in pSet^k\}$ and (4.2.3) $S^k = S^{k-1}$.

$sumSet$ is updated as (5) $sumSet^k = sumSet^{k-1} \setminus pSet^k$ and the tuples derived by the analysis as (6) $T^k_{out} = [C, B^k_{blk}](T^{k-1}_{out} \cup T^k_{load})$.

We observe that the computation of T_{out} has the following structure in the kth iteration of the while loop: (7) $[C, B^k_{blk}](...([C, B^1_{blk}]([C, B^0_{blk}](I_{test}) \cup T^1_{load})...) \cup T^k_{load})$.

Since the while loop executes lines 8-10 or 14-16 of ONLINE, we have either (8) or (9) where: (8) $B^k_{blk} = B^{k-1}_{blk} \wedge T^k_{load} \neq \emptyset \wedge pSet^k \neq \emptyset$ and (9) $B^k_{blk} \subset B^{k-1}_{blk} \wedge T^k_{load} = \emptyset \wedge pSet^k \neq \emptyset$.

(7) can be reduced to the expression (I) below by repeatedly applying Lemma 7 if (8) holds and Lemma 8 if (9) holds. (I) $T^k_{out} = [C, B^k_{blk}](I_{test} \cup \bigcup_{1 \leq i \leq k} T^i_{load})$.

From (1.2), (4.1.2) and (4.1.3), we have (10) if $pSet^k \subseteq S^k$ then $blk(pSet^k) \subseteq B^k_{blk}$. From (4.2.2) and (4.2.3), we have (11) if $pSet^k \not\subseteq S^k$ then $blk(pSet^k) \not\subseteq B^k_{blk}$. From (10) and (11), we have (12) $B^k_{blk} = blk(S^k)$. From (4.1.1), (4.1.3), (4.2.1) and (4.2.3), we have (13) $\bigcup_{1 \leq i \leq k} T^i_{load} = load(S^k)$.

Substituting (12) and (13) in (I), we have (II) $T^k_{out} = [C, blk(S^k)](I_{test} \cup load(S^k))$.

Since $pSet^k \neq \emptyset$, and $sumSet$ is finite, the while loop eventually terminates. If the while loop exits after n iterations, we need to prove (III) $T^n_{out} \cap O = [C](I_{test}) \cap O$.

From P1, we know that (14) $T^0_{out} \subseteq [C](I_{test})$. From (4.1.1) and (4.1.3), we know that (15) $load(S^k) \subseteq T^{k-1}_{out}$. From (14), (15) and repeatedly applying P1 and P2 in (7), we have (16) $T^k_{out} \subseteq [C](I_{test})$. From (16) and the condition check on line 7 of ONLINE, we have (17) $\forall sum \in S^k : pre(sum) \subseteq [C](I_{test})$. From (1) and (17), we have (18) S^k is a set of applicable summaries.

From (II), (18) and Lemma 9, we have (III), as required. □

Lemma 7. $[\![C, B_{blk}]\!]([\![C, B_{blk}]\!](T_1) \cup T_2) = [\![C, B_{blk}]\!](T_1 \cup T_2)$.

Proof. To prove Lemma 7, it suffices to prove that: $([\![C, B_{blk}]\!](T_1 \cup T_2) \subseteq [\![C, B_{blk}]\!]([\![C, B_{blk}]\!](T_1) \cup T_2) \wedge [\![C, B_{blk}]\!]([\![C, B_{blk}]\!](T_1) \cup T_2) \subseteq [\![C, B_{blk}]\!](T_1 \cup T_2))$.
To prove:
(I) $[\![C, B_{blk}]\!](T_1 \cup T_2) \subseteq [\![C, B_{blk}]\!]([\![C, B_{blk}]\!](T_1) \cup T_2)$. From P4, we have (1) $T_1 \subseteq [\![C, B_{blk}]\!](T_1)$. From (1), we have (2) $T_1 \cup T_2 \subseteq [\![C, B_{blk}]\!](T_1) \cup T_2$.

From (2) and P2, we have $[\![C, B_{blk}]\!](T_1 \cup T_2) \subseteq [\![C, B_{blk}]\!]([\![C, B_{blk}]\!](T_1) \cup T_2)$, as required.

To prove: (II) $[\![C, B_{blk}]\!]([\![C, B_{blk}]\!](T_1) \cup T_2) \subseteq [\![C, B_{blk}]\!](T_1 \cup T_2)$.

We know that (3) $T_1 \subseteq T_1 \cup T_2$. From (3) and P2, we have (4) $[\![C, B_{blk}]\!](T_1) \subseteq [\![C, B_{blk}]\!](T_1 \cup T_2)$. From (4), we have (5) $[\![C, B_{blk}]\!](T_1) \cup T_2 \subseteq [\![C, B_{blk}]\!](T_1 \cup T_2) \cup T_2$.

From (5) and P2, we have (6) $[\![C, B_{blk}]\!]([\![C, B_{blk}]\!](T_1) \cup T_2) \subseteq [\![C, B_{blk}]\!]([\![C, B_{blk}]\!](T_1 \cup T_2) \cup T_2)$.

From P4, we have (7) $T_2 \subseteq [\![C, B_{blk}]\!](T_1 \cup T_2)$.

From (7), we have (8) $[\![C, B_{blk}]\!](T_1 \cup T_2) \cup T_2 = [\![C, B_{blk}]\!](T_1 \cup T_2)$. Substituting (8) in (6), we have (9) $[\![C, B_{blk}]\!]([\![C, B_{blk}]\!](T_1) \cup T_2) \subseteq [\![C, B_{blk}]\!]([\![C, B_{blk}]\!](T_1 \cup T_2))$.

From (9) and P6, we have $[\![C, B_{blk}]\!]([\![C, B_{blk}]\!](T_1) \cup T_2) \subseteq [\![C, B_{blk}]\!](T_1 \cup T_2)$, as required. □

Lemma 8. *If* $B_{blk1} \subseteq B_{blk2}$ *then* $[\![C, B_{blk1}]\!]([\![C, B_{blk2}]\!](T_1)) = [\![C, B_{blk1}]\!](T_1)$.

Proof. To prove Lemma 8, it suffices to prove that: If $B_{blk1} \subseteq B_{blk2}$ then $([\![C, B_{blk1}]\!]([\![C, B_{blk2}]\!](T_1)) \subseteq [\![C, B_{blk1}]\!](T_1) \wedge [\![C, B_{blk1}]\!](T_1) \subseteq [\![C, B_{blk1}]\!]([\![C, B_{blk2}]\!](T_1)))$.

Assume (1) $B_{blk1} \subseteq B_{blk2}$.
To prove:
(I) $[\![C, B_{blk1}]\!]([\![C, B_{blk2}]\!](T_1)) \subseteq [\![C, B_{blk1}]\!](T_1)$. From P1 and (1), we have (2) $[\![C, B_{blk2}]\!](T_1) \subseteq [\![C, B_{blk1}]\!](T_1)$. From P2 and (2), we have (3) $[\![C, B_{blk1}]\!]([\![C, B_{blk2}]\!](T_1)) \subseteq [\![C, B_{blk1}]\!]([\![C, B_{blk1}]\!](T_1))$. From (3) and P6, we have $[\![C, B_{blk1}]\!]([\![C, B_{blk2}]\!](T_1)) \subseteq [\![C, B_{blk1}]\!](T_1)$, as required.
To prove:
(II) $[\![C, B_{blk1}]\!](T_1) \subseteq [\![C, B_{blk1}]\!]([\![C, B_{blk2}]\!](T_1))$. From P4, we have (4) $T_1 \subseteq [\![C, B_{blk2}]\!](T_1)$. From (4), and P2, we have $[\![C, B_{blk1}]\!](T_1) \subseteq [\![C, B_{blk1}]\!]([\![C, B_{blk2}]\!](T_1))$, as required. □

Lemma 9. *If* $S = \{sum_1, ..., sum_n\}$ *is a set of n applicable summaries, then* $[\![C, blk(S)]\!](I_{test} \cup load(S)) \cap O = [\![C]\!](I_{test}) \cap O$.

Proof. Let $I_{test} \subseteq I$. Let $S = \{sum_1, ..., sum_n\}$.

(1) Assume S is a set of applicable summaries. We prove Lemma 9 by induction on $|S|$. Here the **Inductive Hypothesis** is, $[\![C, blk(S)]\!](I_{test} \cup load(S)) \cap O = [\![C]\!](I_{test}) \cap O$.

Proving the base case for n = 1. Let $S = \{sum_1\}$ where $sum_1 = (T^1_{pre}, T^1_{post}, \Gamma_1)$. From (1) of Definition 5, we have, (2) $blk(S) = T^1_{pre}$. From (2) of Definition 5, we have, (3) $load(S) = T^1_{post}$. From (2), (3) and the Induction Hypothesis, we need to prove, (I) $[\![C, T^1_{pre}]\!](I_{test} \cup T^1_{post}) \cap O = [\![C]\!](I_{test}) \cap O$. (I) can be rewritten as, (II) $[\![C, \emptyset \cup T^1_{pre}]\!](I_{test} \cup \emptyset \cup T^1_{post}) \cap O = [\![C]\!](I_{test}) \cap O$. From (1) and applying Lemma 10, we have (II) as required.

Proving the inductive case, the Induction Hypothesis holds for $n = k$. To prove for $n = k + 1$. Let $S = \{sum_1, ..., sum_{k+1}\}$ where $\forall i \in [1..k + 1] : sum_i = (T^i_{pre}, T^i_{post}, \Gamma_i)$. We need to prove that, (III) $[\![C, blk(S)]\!](I_{test} \cup load(S)) \cap O = [\![C]\!](I_{test}) \cap O$.

(4) Let $S' = \{sum_1, ..., sum_k\}$ where $\forall i \in [1..k] : sum_i = (T^i_{pre}, T^i_{post}, \Gamma_i)$. From (1) of Definition 5, we have, (5) $blk(S) = blk(S') \cup \{T^{k+1}_{pre}\}$. From (2) of Definition 5, we have, (6) $load(S) = load(S') \cup T^{k+1}_{post}$. From (5) and (6), (III) can be rewritten as, (IV) $[\![C, blk(S') \cup \{T^{k+1}_{pre}\}]\!](I_{test} \cup load(S') \cup T^{k+1}_{post}) \cap O = [\![C]\!](I_{test}) \cap O$. From (4) and (1) of Definition 5, we have, (7) $blk(S') \cap \{T^{k+1}_{pre}\} = \emptyset$. From (1) , (7) and Lemma 10, we have, (8) $[\![C, blk(S') \cup \{T^{k+1}_{pre}\}]\!](I_{test} \cup load(S') \cup T^{k+1}_{post}) \cap O = [\![C, blk(S')]\!](I_{test} \cup load(S')) \cap O$. From (8) and Induction Hypothesis, we have (III), as required. □

Lemma 10. *If $S = \{sum_1, ..., sum_k\}$ is a set of k applicable summaries and $sum = (T_{pre}, T_{post}, \Gamma)$ is an applicable summary such that $sum \notin S$, then $[\![C, blk(S) \cup \{T_{pre}\}]\!](I_{test} \cup load(S) \cup T_{post}) \cap O = [\![C, blk(S)]\!](I_{test} \cup load(S)) \cap O$.*

Proof. Let $I_{test} \subseteq I$. Let $S = \{sum_1, ..., sum_k\}$. (1) Assume S is a set of applicable summaries. (2) Assume sum is applicable. (3) Assume $sum \notin S$. (4) Let $T_{fpost} = fpost(sum)$. (5) Let $T_{prune} = T_{fpost} \setminus T_{post}$.

From (1), (2), (3) and Lemma 11, we have, (6) $[\![C, blk(S)]\!](I_{test} \cup load(S)) = [\![C, blk(S) \cup \{T_{pre}\}]\!](I_{test} \cup load(S) \cup T_{fpost})$. From (5) and (6), we have, (7) $[\![C, blk(S)]\!](I_{test} \cup load(S)) = [\![C, blk(S) \cup \{T_{pre}\}]\!](I_{test} \cup load(S) \cup T_{post} \cup T_{prune})$.

By the definition of sound pruning in Definition 1, we have, (8) $\forall t \in T_{prune} : t \notin O \;\wedge\; (\forall g \in Gr(C, I_{test}) : t \in cbody(g) \Rightarrow chead(g) \subseteq [\![C]\!](T_{pre})))$. From (8) we have, (9) $\forall t \in T_{prune} : t \notin O \;\wedge\; \forall g \in Gr(C, I_{test}) : t \in cbody(g) \Rightarrow chead(g) \subseteq T_{fpost})$.

From (1), (2), P7, Definition 5 and Definition 6, we have, (10) $load(S) \cup T_{fpost} \subseteq [\![C]\!](I_{test})$. From (10) and P8, we have, (11) $[\![C]\!](I_{test}) = [\![C]\!](I_{test} \cup load(S) \cup T_{fpost})$. From P5, (11) and P1, we have, (12) $[\![C, \emptyset]\!](I_{test} \cup load(S) \cup T_{fpost}) \supseteq [\![C, blk(S) \cup \{T_{pre}\}]\!](I_{test} \cup load(S) \cup T_{fpost})$. Since $\emptyset \subseteq blk(S) \cup \{T_{pre}\}$ and from (9), P5, (12), we have, (13) $\forall t \in T_{prune} : t \notin O \;\wedge\; (\forall g \in Gr(C, blk(S) \cup \{T_{pre}\}, I_{test} \cup load(S) \cup T_{fpost}) : t \in cbody(g) \Rightarrow chead(g) \subseteq T_{fpost})$.

From (5) and (13), we have, (14) $\forall t \in T_{prune} : t \notin O \;\wedge\; (\forall g \in Gr(C, blk(S) \cup T_{pre}, I_{test} \cup load(S) \cup T_{post} \cup T_{prune}) : t \in cbody(g) \Rightarrow chead(g) \subseteq T_{fpost})$. From (14), we have, (15) $[\![C, blk(S) \cup \{T_{pre}\}]\!](I_{test} \cup load(S) \cup T_{fpost}) = [\![C, blk(S) \cup \{T_{pre}\}]\!](I_{test} \cup load(S) \cup T_{post}) \cup T_{prune}$. From (6) and (15), we have, (16) $[\![C, blk(S)]\!](I_{test} \cup load(S)) = [\![C, blk(S) \cup \{T_{pre}\}]\!](I_{test} \cup load(S) \cup T_{post}) \cup T_{prune}$.

By the definition of sound pruning in Definition 1, $T_{prune} \cap O = \emptyset$. Therefore, we have, $[\![C, blk(S)]\!](I_{test} \cup load(S)) \cap O = [\![C, blk(S) \cup \{T_{pre}\}]\!](I_{test} \cup load(S) \cup T_{post}) \cap O$, as required. □

Lemma 11. *If $S = \{sum_1, ..., sum_k\}$ is a set of k applicable summaries, $sum = (T_{pre}, T_{post}, \Gamma)$ is an applicable summary such that $sum \notin S$ and $T_{fpost} = fpost(sum)$, then $[\![C, blk(S) \cup \{T_{pre}\}]\!](I_{test} \cup load(S) \cup T_{fpost}) = [\![C, blk(S)]\!](I_{test} \cup load(S))$.*

Proof. Let $I_{test} \subseteq I$. Let $S = \{sum_1, ..., sum_k\}$. (1) Assume S is a set of applicable summaries. (2) Assume sum is applicable. (3) Assume $sum \notin S$. (4) Assume $T_{fpost} = fpost(sum)$. We extend the definition of the set of grounded constraints for instrumented Datalog analyses.

$$\begin{aligned} Gr(C, B_{blk}, I_1) = \{ & [\![l_0]\!](\sigma) \,\text{:-}\, [\![l_1]\!](\sigma), ..., [\![l_n]\!](\sigma) \mid \\ & l_0 \,\text{:-}\, l_1, ..., l_n \in C \;\wedge \\ & \textstyle\bigwedge_{1 \le k \le n}([\![l_k]\!](\sigma) \in [\![C, B_{blk}]\!](I_1) \wedge \sigma \in \Sigma)\} \end{aligned}$$

We next define a function $F_{C,B_{blk}}$ that defines the tuples that can be derived in one step from a given set of tuples. Let $F_{C,B_{blk}} \in 2^T \to 2^T$. $F_{C,B_{blk}}(T) = T \cup \bigcup_{c \in C}(\{[\![c, B_{blk}]\!](T)\})$. Let $F^i_{C,B_{blk}}$ be i applications of $F_{C,B_{blk}}$. That is, $F^i_{C,B_{blk}} = F_{C,B_{blk}}(...(F_{C,B_{blk}}(T))), i$ times.

(5) Let $T_1 = I_{test} \cup load(S)$. (6) Let $B_{blk} = blk(S)$. (7) Let $B'_{blk} = B_{blk} \cup \{T_{pre}\}$. (8) Let $t \in [\![C, B'_{blk}]\!](T_1 \cup T_{fpost})$.

We state below that tuple t must be derived in some ith application of $F_{C,B'_{blk}}$. For some $n \ge 0$, $[\![C, B'_{blk}]\!](T_1 \cup T_{fpost}) = F^n_{C,B'_{blk}}(T_1 \cup T_{fpost})$. From (8), $\exists i \in [1..n] : t \in F^i_{C,B'_{blk}}(T_1 \cup T_{fpost})$.

We first prove: (I) $[\![C, B_{blk} \cup \{T_{pre}\}]\!](T_1 \cup T_{fpost}) \subseteq [\![C, B_{blk}]\!](T_1)$ by induction on i. Here the **Inductive Hypothesis** is: if $(\exists i \in [1..n] : t \in F^i_{C,B'_{blk}}(T_1 \cup T_{fpost}))$, then $t \in [\![C, B_{blk}]\!](T_1)$.

Proving the base case for $i = 1$, (9) $t \in F^1_{C,B'_{blk}}(T_1 \cup T_{fpost})$. From (9), we have, (9.1) $\exists g \in Gr(C, B'_{blk}, T_1 \cup T_{fpost}) : t \in chead(g) \;\wedge\; cbody(g) \in T_1 \cup T_{fpost}$. From P4, we have, (10) $T_1 \subseteq [\![C, B_{blk}]\!](T_1)$. From (5), we have, (11) $I_{test} \subseteq T_1$. From (2) and Definition 4, we have, (12) $T_{pre} \subseteq [\![C]\!](I_{test})$. From (11) and (12), we have, (13) $T_{pre} \subseteq [\![C]\!](T_1)$. From (3) and (6), we have, (14) $T_{pre} \notin B_{blk}$. From (13), (14) and the definition in Figure 5, we have, (15) $T_{fpost} \subseteq [\![C, B_{blk}]\!](T_1)$. From (9.1), (10) and (15), we have, (16) $cbody(g) \subseteq [\![C, B_{blk}]\!](T_1)$. We know from (9) that t is not blocked by $B_{blk} \cup \{T_{pre}\}$. So, it is not blocked by B_{blk}. From this and (16), we have, $t \in [\![C, B_{blk}]\!](T_1)$, as required.

Proving the inductive case, the inductive hypothesis holds for $i = k$. To prove it for $i = k + 1$. (17) $t \in F^{k+1}_{C,B'_{blk}}(T_1 \cup T_{fpost})$. From (17), $\exists g \in Gr(C, B'_{blk}, T_1 \cup T_{fpost}) : t \in chead(g) \;\wedge\; cbody(g) \in F^k_{C,B'_{blk}}(T_1 \cup T_{fpost})$. By the inductive hypothesis, $cbody(g) \in [\![C, B_{blk}]\!](T_1)$. From (17), we know that, g is not blocked by $B_{blk} \cup \{T_{pre}\}$. Therefore, it will not be blocked by B_{blk}. Therefore, $t \in [\![C, B_{blk}]\!](T_1)$ and, $[\![C, B_{blk} \cup \{T_{pre}\}]\!](T_1 \cup T_{fpost}) \subseteq [\![C, B_{blk}]\!](T_1)$, as required.

Next, we prove, (II) $[\![C, B_{blk}]\!](T_1) \subseteq [\![C, B_{blk} \cup \{T_{pre}\}]\!]$ $(T_1 \cup T_{fpost})$ by induction on i. Here the **Inductive Hypothesis** is: if $\exists i \in [1..n] : t \in F^{i}_{C,B_{blk}}(T_1))$, then $t \in [\![C, B_{blk} \cup \{T_{pre}\}]\!](T_1 \cup T_{fpost})$.

Proving the base case for $i = 1$, (18) $t \in F^{1}_{C,B_{blk}}(T_1)$. From (18), we have, (19) $\exists g \in Gr(C, B_{blk}, T_1) : t \in chead(g) \wedge cbody(g) \subseteq T_1$.

We prove by contradiction that, $g \in Gr(C, B_{blk} \cup \{T_{pre}\}, T_1 \cup T_{fpost})$. Let $g \notin Gr(C, B_{blk} \cup \{T_{pre}\}, T_1 \cup T_{fpost})$. (20) Then g is blocked by $B_{blk} \cup \{T_{pre}\}$. There are two cases:

Case 1: g is blocked by B_{blk}. That is, $\exists b \in B_{blk} : cbody(g) \subseteq b$. From (19) and the definition of an instrumented Datalog analysis, we conclude that if g had been blocked by B_{blk}, then t could not have belonged to $F^{1}_{C,B_{blk}}(T_1)$. (21) Therefore, g is not blocked by B_{blk}. From (21), we have, (22) $t \in [\![C, B_{blk} \cup \{T_{pre}\}]\!](T_1 \cup T_{fpost})$. Therefore, $g \in Gr(C, B_{blk} \cup \{T_{pre}\}, T_1 \cup T_{fpost})$, as required.

Case 2: g is blocked by T_{pre}. That is, $cbody(g) \subseteq T_{pre}$. From this and definition of Datalog semantics, we have, (23) $t \in [\![C]\!](T_{pre}) = T_{fpost}$. From P4 and (23), we have, (24) $t \in [\![C, B_{blk} \cup \{T_{pre}\}]\!](T_1 \cup T_{fpost})$. Even if g is blocked by T_{pre}, (24) holds. Therefore, $g \in Gr(C, B_{blk} \cup \{T_{pre}\}, T_1 \cup T_{fpost})$, as required.

Proving the inductive case, inductive hypothesis holds for $i = k$. To prove it for $i = k + 1$, (25) $t \in F^{k+1}_{C,B_{blk}}(T_1)$. From (25), we have, (26) $\exists g \in Gr(C, B_{blk}, T_1) : t \in chead(g) \wedge cbody(g) \in F^{k}_{C,B_{blk}}(T_1)$. By the Induction Hypothesis, $cbody(g) \subseteq [\![C, B_{blk} \cup \{T_{pre}\}]\!](T_1 \cup T_{fpost})$. We prove by contradiction that, $g \in Gr(C, B_{blk} \cup \{T_{pre}\}, T_1 \cup T_{fpost})$. Let $g \notin Gr(C, B_{blk} \cup \{T_{pre}\}, T_1 \cup T_{fpost})$. (27) Then g is blocked by $B_{blk} \cup \{T_{pre}\}$. There are two cases:

Case 1: g is blocked by B_{blk}. That is, $\exists b \in B_{blk} : cbody(g) \subseteq b$. From (26) and the definition of an instrumented Datalog analysis, we conclude that if g had been blocked by B_{blk}, then t could not have belonged to $F^{k+1}_{C,B_{blk}}(T_1)$. (28) Therefore, g is not blocked by B_{blk}. From (28), we have, (29) $t \in [\![C, B_{blk} \cup \{T_{pre}\}]\!](T_1 \cup T_{fpost})$. Therefore, $g \in Gr(C, B_{blk} \cup \{T_{pre}\}, T_1 \cup T_{fpost})$, as required.

Case 2: g is blocked by T_{pre}. That is, (30) $cbody(g) \subseteq T_{pre}$. From (30) and definition of Datalog semantics, we have, (31) $t \in [\![C]\!](T_{pre}) = T_{fpost}$. From P4 and (31), we have, (32) $t \in [\![C, B_{blk} \cup \{T_{pre}\}]\!](T_1 \cup T_{fpost})$. Even if g is blocked by T_{pre}, (32) holds. Therefore, $g \in Gr(C, B_{blk} \cup \{T_{pre}\}, T_1 \cup T_{fpost})$, as required.

Therefore, as required by (II), $[\![C, B_{blk}]\!](T_1) \subseteq [\![C, B_{blk} \cup \{T_{pre}\}]\!](T_1 \cup T_{fpost})$. □

References

[1] K. Ali and O. Lhoták. Application-only call graph construction. In *ECOOP*, 2012.

[2] K. Ali and O. Lhoták. Averroes: Whole-program analysis without the whole program. In *ECOOP*, 2013.

[3] N. Allen, B. Scholz, and P. Krishnan. Staged points-to analysis for large code bases. In *CC*, 2015.

[4] T. Ball and S. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *PASTE*, 2001.

[5] E. Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *SOAP*, 2012.

[6] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, 2009.

[7] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.

[8] R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *POPL*, 1999.

[9] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E. Neuhold, editor, *IFIP Conference on Formal Description of Programming Concepts*. North-Holland, 1977.

[10] P. Cousot and R. Cousot. Modular static program analysis. In *International Conference on Compiler Construction (CC'02)*, 2002.

[11] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, 2011.

[12] J. Dolby, S. Fink, and M. Sridharan. T. J. Watson Libraries for Analysis. `http://wala.sourceforge.net/`, 2006.

[13] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *POPL*, 2010.

[14] B. S. Gulavani, S. Chakraborty, G. Ramalingam, and A. V. Nori. Bottom-up shape analysis using LISF. *ACM Trans. Program. Lang. Syst.*, 33(5), 2011.

[15] G. Kastrinis and Y. Smaragdakis. Hybrid context sensitivity for points-to analysis. In *PLDI*, 2013.

[16] R. Mangal, M. Naik, and H. Yang. A correspondence between two approaches to interprocedural analysis in the presence of join. In *ESOP*, 2014.

[17] B. R. Murphy and M. S. Lam. Program analysis with partial transfer functions. In *PEPM*, 2000.

[18] M. Naik. Chord: A versatile program analysis platform for Java bytecode. `http://jchord.googlecode.com`, 2006.

[19] H. Oh, H. Yang, and K. Yi. Learning a strategy for adapting a program analysis via Bayesian optimisation. In *OOPSLA*, 2015.

[20] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[21] N. Rinetzky, J. Bauer, T. W. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, 2005.

[22] A. Rountev, M. Sharp, and G. Xu. IDE dataflow analysis in the presence of large object-oriented libraries. In *Compiler Construction*, 2008.

[23] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[24] Y. Smaragdakis and M. Bravenboer. Using datalog for fast and easy program analysis. In *Datalog 2.0 Workshop*, 2010.

[25] Y. Smaragdakis, G. Balatsouras, and G. Kastrinis. Set-based pre-processing for points-to analysis. In *OOPSLA*, 2013.

[26] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *POPL*, 2013.

[27] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.

[28] J. Whaley and M. C. Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA*, 1999.

[29] J. Whaley, D. Avots, M. Carbin, and M. Lam. Using datalog with binary decision diagrams for program analysis. In *APLAS*, 2005.

[30] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI*, 1995.

[31] D. Yan, G. Xu, and A. Rounte. Rethinking soot for summary-based whole-program analysis. In *SOAP*, 2012.

[32] G. Yorsh, E. Yahav, and S. Chandra. Generating precise and concise procedure summaries. In *POPL*, 2008.

[33] X. Zhang, R. Mangal, M. Naik, and H. Yang. Hybrid top-down and bottom-up interprocedural analysis. In *PLDI*, 2014.