Accelerating Program Analyses in Datalog by Merging Library Facts

Yifan Chen[1], Chenyang Yang[1], Xin Zhang[1] *, Yingfei Xiong[1], Hao Tang[2], Xiaoyin Wang[3], and Lu Zhang[1]

[1] Key Laboratory of High Confidence Software Technologies, MoE
Department of Computer Science and Technology, EECS, Peking University
Beijing, China
{yf_chen, chenyangy, xin, xiongyf, zhanglucs}@pku.edu.cn
[2] Alibaba Group
Hangzhou, China
albert.th@alibaba-inc.com
[3] Department of Computer Science, University of Texas at San Antonio
Texas, USA
xiaoyin.wang@utsa.edu

Abstract. Static program analysis uses sensitivity to balance between precision and scalability. However, finer sensitivity does not necessarily lead to more precise results but may reduce scalability. Recently, a number of approaches have been proposed to finely tune the sensitivity of different program parts. However, these approaches are usually designed for specific program analyses, and their abstraction adjustments are coarse-grained as they directly drop sensitivity elements.
In this paper, we propose a new technique, 4DM, to tune abstractions for program analyses in Datalog. 4DM merges values in a domain, allowing fine-grained sensitivity tuning. 4DM uses a data-driven algorithm for automatically learning a merging strategy for a library from a training set of programs. Unlike existing approaches that rely on the properties of a certain analysis, our learning algorithm works for a wide range of Datalog analyses. We have evaluated our approach on a points-to analysis and a liveness analysis, on the DaCapo benchmark suite. Our evaluation results suggest that our technique achieves a significant speedup and negligible precision loss, reaching a good balance.

Keywords: Static Analysis · Datalog · Data-Driven Analysis · Domain-Wise Merging

1 Introduction

One key problem in program analysis is to choose what information to keep in the program abstraction in order to balance its precision and scalability. It is often controlled through a domain of sensitivity elements. For example, call-site sensitivity [19, 17] distinguishes information under different calling contexts

* Corresponding Author

using the string of most recent call sites, and it is parameterized by the length of this string. In general, the finer the sensitivity is, the higher the precision is, yet the lower the scalability is. However, it is not always the case. Under some circumstances, coarse-grained sensitivity is enough and making it finer does not lead to higher precision. Using fine-grained sensitivity in such cases would lead to unnecessary time cost.

In order to reduce the time spent by such "inefficient" sensitivity, many existing approaches focus on using sensitivity selectively. A common practice is to drop sensitivity elements at specific program points. Take tuning call-site sensitivity as an example: Smaragdakis et al. [20] used a crafted rule to decide whether to keep the contexts for specific call sites; Jeong et al. [11] tried to cut the context length for each call site using a data-driven method; and Zhang et al. [27] proposed an approach which started by dropping all contexts and then selectively added them back by identifying call sites where adding contexts helps. However, a binary choice of whether to drop a sensitivity attribute can be too coarse-grained to achieve the best precision and efficiency. Imagine, in a 1-object-sensitive points-to analysis, a method is invoked by 10 different objects at a call site, 9 of them leading to the same analysis result while the other one leading to a different result. Dropping all of the contexts would lead to imprecise analysis, but keeping the contexts leads to 8 rounds of unproductive analysis.

Recently, Tan et al. [21] proposed a different idea of tuning abstraction which is more fine-grained. Their method, MAHJONG, merges heap objects into different equivalent classes according to type and field-points-to information. In the previous example, if certain conditions are met, MAHJONG could merge the 9 objects that lead to the same analysis result, therefore achieving the same precision as a 1-object-sensitive analysis but with better efficiency. However, this approach is limited to tuning heap abstraction for type-related queries in points-to analysis. Specifically, it only allows merging heap objects and relies on a pre-analysis to identify which objects to merge. If the query is not about types, e.g., querying aliases of a variable, it is not applicable.

We propose a new merging-based method, named **4DM** (Data-Driven Datalog Domain-wise Merging), to resolve the weakness of the above two approaches. For given domains, 4DM's **domain-wise merging** merges concrete values that contribute to similar results into the same abstract value. It makes finer adjustment on domains of sensitivity elements by merging into multiple abstract values than simply dropping as in context reduction. Furthermore, it generalizes context reduction. For example, reducing the contexts for a particular call site c from 2-objective-sensitive to 1-object-sensitive can be seen as transforming all triples (o_i^1, o_i^2, c) to $(*, o_i^2, c)$, where o_i^1 and o_i^2 are the calling contexts and $*$ is an abstract value. 4DM merges values in various domains as opposed to only heap objects, e.g. call sites, program points, variables, and can be applied to tune various sensitivities in a wide range of analyses.

In order to apply merging to various domains in different analyses, 4DM employs a general framework for **Datalog**-based analyses. It transforms Datalog rules to embed merging in them, and guarantees soundness.

```
1 package library;
2
3 public class Lib {
4   public static A id(A x){
5     return x;
6   }
7   public static A foo(A u){
8     v = id(u); // cs5
9     return v;
10  }
11  public static A bar(A s){
12    t = id(s); // cs6
13    return t;
14  }
15 }
```

```
package client1;                                    1
import library;                                     2
public class Clt1 {                                 3
  public static void main(String[] args){           4
    A p1 = new A(); // o1                           5
    A p2 = new A(); // o2                           6
    x1 = library.Lib.id(p1); // cs1                 7
    x2 = library.Lib.id(p1); // cs2                 8
    x3 = library.Lib.foo(p2); // cs3                9
    x4 = library.Lib.bar(p2); // cs4                10
    assert(x1==x2); // Q1: safe?                    11
    assert(x3==x4); // Q2: safe?                    12
    assert(x1==x3); // Q3: safe?                    13
  }                                                 14
}                                                   15
```

Listing 1.1: Example code that higher sensitivity leads unnecessary computation

Now the challenge is to find an effective merging strategy under 4DM's representation that accelerates analysis while keeping precision. Our insight to address this challenge is that library code occupies a large part in analysis and the merging strategy on shared library code would be similar for different programs. Under this insight, we propose a **data-driven** method to learn a good merging for programs that share a common library.

We have implemented 4DM, and evaluated it on two different Datalog-based analyses, a points-to analysis and a liveness analysis. The results suggest 4DM could achieve significant speedup on both analyses with minimal precision loss.

This paper makes the following contributions:

- A general framework for accelerating analyses in Datalog by merging sensitivity elements.
- A learning algorithm to discover an effective strategy for merging sensitivity elements in library code.
- Empirical evaluation that demonstrates the effectiveness our approach.

The rest of this paper is organized as follows. Section 2 uses a motivating example to give a comprehensive overview of 4DM. Section 3 prepares some knowledge of Datalog. Section 4 describes how to apply 4DM's merging to Datalog rules in detail and proves some of its important properties. Section 5 explains 4DM's algorithm to learns a merging strategy from input programs that share a common library. Section 6 describes the implementation and evaluation of 4DM.

2 Overview

In this section, we informally describe 4DM using a motivating example.

Listing-1.1 demonstrates an example where finer-grained sensitivity leads to unnecessary computation. The code snippet contains two packages, i.e. `library`

and `client1`. Package `library` declares method `foo`, `bar` and `id`. Package `client1` declares method `main`. In the comments, $cs_1 \dots cs_6$ represent the six call sites in the program, while o_1, o_2 represent abstract objects allocated by `new` statements in the corresponding lines. At the end of the `main` method, the developer queries whether three assertions may be violated, denoted as **Q1**, **Q2**, and **Q3** respectively. It is easy to see that while **Q1** and **Q2** hold, **Q3** does not.

We can derive the correct results by applying a 1-call-site-sensitive (*1cs*) points-to analysis. In particular, it is sufficient to distinguish calls to `id` in `main` and calls to `id` in `foo` and `bar`. When applying a 1cs analysis, there are 4 different contexts for variable x in line 5 of `library`, and x points to different objects in these contexts, as follows.

$$x \mapsto \{o_1\} \text{ in two contexts } [cs_1], [cs_2]$$
$$x \mapsto \{o_2\} \text{ in two contexts } [cs_5], [cs_6]$$

While 1cs analysis successfully resolves all three queries, there is redundancy in the computation. In particular, distinguishing the call sites cs_1 and cs_2 does not increase the precision. Similar for cs_5 and cs_6. Ideally we want to remove such redundancy.

2.1 Accelerating by Domain-Wise Merging

Before introducing our idea, let us first see whether we can remove this redundancy using existing methods. A dominating approach for tuning analysis abstractions is to select different sensitivities for different program points [11, 10, 14, 15, 26, 16]. In this case, it allows dropping the contexts for certain call sites to `id`. However, to preserve the precision, we can only drop the contexts either for both $\{cs_1, cs_2\}$ or for both $\{cs_5, cs_6\}$. However, there is still redundancy in the call sites where the context is not dropped. On the other hand, a previous merging-based approach, Mahjong [21], only allows merging heap objects.

4DM uses a novel method for abstraction-tuning, which we refer to as "domain-wise merging". In the running example, our approach would conclude that the call sites cs_1 and cs_2 have the same effect on the queries (**Q1**, **Q2** and **Q3**) and thus can be merged. Therefore, our approach would treat them as an equivalent class and use symbol $*_1$ to represent the equivalent class. Similarly, our approach would identify cs_5 and cs_6 as equivalent and use $*_2$ to represent the class. As a result, the original four contexts for variable x in line 5 become 2 contexts:

$$[*_1] = \{[cs_1], [cs_2]\} \text{ and } [*_2] = \{[cs_5], [cs_6]\}$$

This merged abstraction of calling contexts removes the redundancy in the original analysis while keeping the precision.

In order to apply the idea of domain-wise merging to different sensitivities in a wide range of analysis, we propose a general framework that allows rich ways to merge facts in Datalog-based analyses (Section 4). Further, we prove that under this framework, any merging strategy would yield a sound analysis if the original analysis is sound.

```
1 package client2;
2 import library;
3 public class Clt2 {
4   public static void main(String[] args) {
5     A q1 = new A(); // o3
6     y1 = library.Lib.foo(p1); // cs7
7     y2 = library.Lib.bar(p1); // cs8
8     assert(y1==y2); // Q4: safe?
9   }
10 }
```

Listing 1.2: Another example client that uses the `library` package

2.2 Learning a Merging Over library Code

While the above framework defines the space of sound merging strategies, the next question is how to find a merging that accelerates the analysis while keeping the precision as much as possible.

We propose a data-driven method focusing on library code. Many modern program analyses spend a large portion of its time in analyzing large library code (e.g., JDK for Java programs). Based on the assumption that different programs use libraries in similar ways, we can obtain a heuristic that merges facts in a library by observing analysis runs on a training set of programs that share this library. Then we can use this heuristic to accelerate analyzing a new program that also uses this library.

For example, suppose there is another client package `client2` in Listing-1.2 that also invokes `foo` and `bar` in the `library` package. Similar to the `client1`, it passes the same object to these two functions. At the end, there is an aliasing assertion **Q4**. In a 1cs analysis, it still takes a large portion of runtime on package `library`, while the merging strategy on `library` is the same as `client1`, as merging call sites cs_5 and cs_6 in function `foo` and `bar` can accelerate the analysis without losing precision for **Q4**. Inspired by this observation, we can discover this merging strategy by trying out various merging strategies on `client1`, and then use this strategy to accelerate analyzing `client2`.

We describe a general method for obtaining such merging strategies in Section 5. In particular, our approach allows training on multiple programs. When the training set is large enough, we are confident that the learnt merging can be applied to programs outside the training set. Furthermore, our framework is general to allow specifically-designed training algorithms for certain analyses, and our evaluation shall demonstrate such an algorithm for liveness analysis.

3 Preliminary

Before we describe 4DM, we first briefly introduce Datalog. Datalog is a declarative logic programming language, which started for querying deductive database. Recently, it has been widely used for specifying program analyses.

			(relations)	$r \in \mathbb{R} = \{R^0, R^1, \ldots\}$
(program)	$C ::= \bar{c};\ o$		(variables)	$v \in \mathbb{V} = \{X, Y, \ldots\}$
(rule)	$c ::= l \leftarrow \bar{l}$		(constants)	$d \in \mathbb{D} = \{0, 1, \ldots\}$
(literal)	$l ::= r(\bar{a})$		(domains)	$D \in \mathbb{M} = \{D_1, D_2, \ldots\} \subseteq \mathcal{P}(\mathbb{D})$
(argument)	$a ::= v \mid d$			$dom(r) = \overline{D}$
(output)	$o ::= \text{'}output\text{'}\ \bar{r}$		(tuples)	$t \in \mathbb{T} \subseteq \mathbb{R} \times \mathbb{D}^*$
			(substitutions)	$\sigma \in \Sigma \subseteq \mathbb{V} \to \mathbb{D}$

Fig. 1: Syntax of Datalog and auxiliary definitions

$$
\begin{array}{ll}
[\![C]\!] \in \mathcal{P}(\mathbb{T}) & F_C, f_c \in \mathcal{P}(\mathbb{T}) \to \mathcal{P}(\mathbb{T}) \\
[\![C]\!] = lfp(F_C) & F_C(T) = T \cup \bigcup\{f_c(T) \mid c \in C\} \\
\multicolumn{2}{r}{f_{l_0 \leftarrow l_1, \ldots, l_n}(T) = \{\sigma(l_0) \mid \sigma(l_k) \in T \ for\ 1 \le k \le n} \\
\multicolumn{2}{r}{\wedge \sigma(l_0)[i] \in dom(l_0)[i] \ for\ 1 \le i \le |l_0|\}}
\end{array}
$$

Fig. 2: Semantics of Datalog

The syntax of Datalog is listed in Figure 1. A Datalog program is constructed from a list of rules and an output instruction. (Here, overbar like $\bar{c}$ represents a list of zero, one or more elements.) Each rule has a head and a body. A head is one literal, and a body is a list of literals. Each literal consists of a relation name and several arguments. Each argument is either a variable or a constant. We call literals containing no variables as **ground literals** or **tuples**, and call constants in tuples as values. A rule should be well-formed, in the sense that all variables occurring in the head should also appear in the body. In addition, we use an output instruction to mark some relations as the analysis results.

All relations are assigned with domains for each of its dimensions. The domain constrains possible constants that a variable at this dimension can be substituted with, and constants should also conform the constraints. We define a relation **super-domain** among domains: – when a variable appears both in the head and body in rule, the corresponding domain D_H in the head is a (direct) super-domain of the corresponding domain D_B in the body, – and transitively, super-domains of D_H are also super-domains of D_B.

Each Datalog program C denotes a set of tuples derived using its rules, as detailed in Figure 2. Each rule $l_0 \leftarrow l_1, \ldots, l_n$ is interpreted as deriving a tuple from known tuples: if there exists a substitution σ such that $\sigma(l_1), \ldots, \sigma(l_n)$ are all known tuples, and every constant in $\sigma(l_0)$ satisfies the domain constraint, $\sigma(l_0)$ is derived. The program denotes the least fixed-point (lfp) of repeated applications of the rules in C. We use a subscript $[\![C]\!]_o$ to denote the derived tuples of output relations o.

Usually, we want to keep all derived tuples, so we require that if domain D_H in the head is a super-domain of D_B in the body, D_H's valueset should also be a superset of D_B's. Thus, the domain constraints are always satisfied during derivation.

4 Constructing Domain-wise Merging in Datalog

In this subsection we introduce our definition of domain-wise merging and how we transform the Datalog rules to implement domain-wise merging.

4.1 1-Domain-Wise Merging

We first give our definition of 1-domain-wise merging, which specifies what elements in a domain should be merged into an abstract element. Then we discuss how to transform the Datalog rules to support this kind of merging. Finally, we prove our transformation is sound: any facts that can be produced by the original rule set can still be produced by the transformed rule set.

Defining Merging. Usually, for a Datalog program, we only cares about derived tuples of output relations. Our goal is to keep derived tuples in output relations unchanged, so we hope not to merge any values in domains of output relations. We also need to avoid merging values in the domains where output tuples are derived from. Therefore, we first give the definition of sensitivity domains, in which the values can be merged.

Definition 1 (Sensitivity Domain). *For a Datalog program C and a set of output relations o, a domain $D \in \mathbb{M}$ is a sensitivity domain iff no domain of output relations is D's super-domain.*

Then, a 1-domain-wise merging is defined as a function mapping concrete values in a domain to abstract values. When multiple concrete values are mapped to the same abstract values, these values are merged. As a result, the domain itself is also changed.

Definition 2 (1-Domain-Wise Merging). *A (1-domain-wise) merging in the sensitivity domain D_α is a function that maps some of its values $\Delta = \{d_1, d_2, \ldots\}$ to abstract values $\widehat{\Delta} = \{\alpha_1, \alpha_2, \ldots\}$ while keeping other values unchanged.*

$$\pi : D_\alpha \to \widehat{D_\alpha},\ \textit{where}\ \widehat{D_\alpha} = (D_\alpha \setminus \Delta) \cup \widehat{\Delta}$$

In Section 5 we shall discuss how to obtain mergings through learning over a set of client programs.

Transforming Datalog Programs. When we have a merging π on domain D_α, we would like to perform the Datalog analysis over the merged abstract values rather than the original concrete values to accelerate the analysis. Therefore, we propose a transformation of Datalog rules to apply this merging.

We start from replacing all occurrences of the concrete values to merge in D_α with abstract values, changing D_α to $\widehat{D_\alpha}$:

- If there is a constant d in the corresponding dimension of D_α, it is changed to $\pi(d)$.
- If D_α is derived from other domains, or say, there is a variable X of D_α in a rule head,

$$R^0(\ldots, X : D_\alpha, \ldots) \leftarrow \ldots, R^k(\ldots, X, \ldots), \ldots.$$

we add a relation $Abstract(X, \widehat{X})$ ($dom(Abstract) = D_\alpha \times \widehat{D_\alpha}$) in the rule body and change X in the head to $\widehat{X}$,

$$R^0(\dots, \widehat{X} : \widehat{D_\alpha}, \dots) \leftarrow \dots, R^k(\dots, X, \dots), \dots, Abstract(X, \widehat{X}).$$

Thus, every time a constant in D_α is derived by this rule, it is merged into $\widehat{D_\alpha}$.

- If D_α derives to its super-domain, or say, there is a variable X of D_α in a rule body and the same variable of D_β in the rule head, the concrete values to merge in D_β should also be replaced with abstract values by derivation. It involves no syntactical change, but D_β should be changed to $(\widehat{D_\beta} \setminus \Delta) \cup \widehat{\Delta}$. Transitively, all super-domains of D_α are changed. The previously described transformations are applied to all these super-domains. In the following paragraphs, we refer to D_α and all its super-domains as *merging domains*.

However, the change in domains will break some original derivation, when two domains in the body of a rule joins with each other, i.e.,

$$R^0(\dots) \leftarrow \dots, R^i(\dots, X, \dots), \dots, R^j(\dots, X, \dots), \dots.$$

If D_i and D_j, domains of X in the two relations, are both non-merging domains or both merging domains, the equivalence of values between the two domains are unchanged, so original derivation still holds. However, if D_i is a merging domain but D_j is not (the order does not matter here), the derivation would break after replacing concrete values in D_i with abstract values, because the abstract values cannot matchf the unchanged concrete values in D_j. To restore the derivation, the rule should be transformed as:

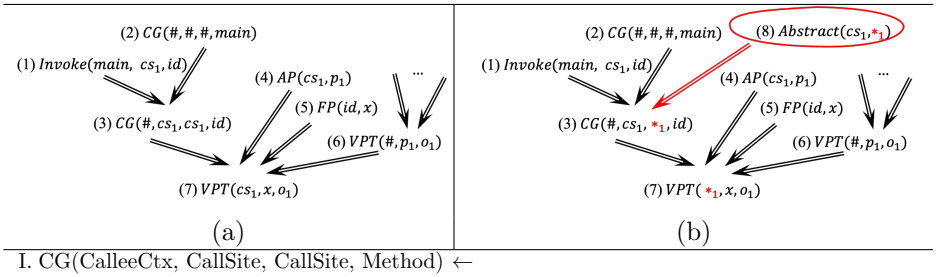
$$R^0(\dots) \leftarrow \dots, R^k(\dots, X, \dots), \dots, R^l(\dots, \widehat{X}, \dots), \dots, Abstract(X, \widehat{X}).$$

When X is substituted with the original concrete constant d and $\widehat{X}$ is substituted with abstract value $\pi(d)$, the derivations still holds.

We use an example to demonstrate the effect of the rule transformation. Figure 3(a) shows a proof tree that is used to derive a points-to instance of the example in Section 2. Each node is a tuple derived from other tuples by a rule. Node (3) is derived from (1)(2) by rule I and Node (7) is derived from (3)(4)(5)(6) by rule II. Note that the second domain of $CallGraph(CG)$ is the call-site of a function call, and the third domain of CG is context of called function, represented by its most recent call-site. In (3), it means call-site cs_1 under initial context # calls method id with context cs_1, and we would like to replace cs_1 in the context of called functions with abstract value $*_1$, while keep the cs_1 in call-site domain unchanged.

Figure 3(b) shows a proof tree after our transformation. When the call-site cs_1 in $Invoke$ derives into the context domain of CG in (3), the transformed rule for CG replace it with abstract value $*_1$; meanwhile, the call-site domain in CG is not super-set of the context domain, so the cs_1 in the second dimension of CG is not replaced. Furthermore, though rule II is not transformed, its first domain - context of variable - is a super-domain of context of called function in CG according to rule II, so cs_1 in (7) is also changed accordingly.

Soundness. Finally, we show that our transformation is sound: any relation that can be produced by the original rules is still produced by the transformed rules.



I. CG(CalleeCtx, CallSite, CallSite, Method) ←
Invoke(Callee, CallSite, Method), CG(CallerCtx, Caller, CalleeCtx, Callee).
II. VPT(CalleeCtx, FormalVar, Obj) ←
CallGraph(CallerCtx, CallSite, CalleeCtx, Method), ActualParam(CallSite, ActualVar), FormalParam(Method, FormalVar), VarPointsTo(CallerCtx, ActualVar, Obj).

Fig. 3: An example of embedding merging rules within proof trees

Theorem 1 (Soundness). *For a Datalog program C, given a domain-wise merging of sensitivity elements π on domain D_π, the derived tuples of output relations after applying π , denoted as $[\![C|\pi]\!]_o$, contains all of the original derived tuples $[\![C]\!]_o$, i.e. $[\![C]\!]_o \subseteq [\![C|\pi]\!]_o$.*

Proof. For any derived tuple of relations in the original Datalog analysis, C, $R(d_1, d_2, \ldots) \in [\![C]\!]$, there is a derivation for it. According to the process of transforming Datalog rules, the derivation still hold after transforming the rules and replacing merged concrete values with abstract values. So by induction, $\Pi[R(d_1, d_2, \ldots)] = R(\Pi(d_1), \Pi(d_2), \ldots) \in [\![C|\pi]\!]$ ($\Pi(d) = \pi(d)$ if d is in merging domains, otherwise $\Pi(d) = d$). Since an output relation r_o has no super-domain of D_π, its tuples are unchanged, so $R_o(d_1, d_2, \ldots) \in [\![C|\pi]\!]$. Thus, $[\![C]\!]_o \subseteq [\![C|\pi]\!]_o$ is proved.

The above theorem concerns only the standard Datalog. In practice, negation operator $\neg$ is frequently used, which supports the negation operation over relations. Unfortunately, the above soundness property does not hold if negations is made on the domain to be merged. In such a case, we may still merge the domains where no negation is applied, or merge the values that are in a negated domain but would never be involved in a negation calculation during an execution.

4.2 N-Domain-Wise Merging

Under some circumstances, there are several sensitivity domains in a Datalog program to which we can apply mergings. We can find mergings in these domains and apply these mergings one by one, but this approach cannot capture the correlation among these sensitivity domains. For example, in a context-sensitive points-to analysis, heap objects and their contexts are usually combined to derive some relations, such as field points-to relations. So we attempt to extend our merging function on the Cartesian product of multiple sensitivity domains:

$$\pi : D_1 \times \cdots \times D_N \to \widehat{D_{1\ldots N}}$$

Here $\widehat{D_{1\times\cdots\times N}} = (D_1 \times \cdots \times D_N \setminus \Delta_{1\ldots N}) \cup \widehat{\Delta_{1\ldots N}}$, $\Delta_{1\ldots N}$ is the set of concrete value tuples to merge and $\widehat{\Delta_{1\ldots N}}$ is the set of abstract values.

We can apply the merging by transforming Datalog rules similarly, changing $\mathit{Abstract}(X, \widehat{X})$ to $\mathit{Abstract}(X_1, \ldots, X_N, \widehat{X}, \ldots, \widehat{X})$ ($\widehat{X}$ is the abstract value copied N times to keep the N-D shape). It can be proved similarly that soundness of N-domain-wise merging still holds.

However, there is a challenge in defining such an N-dimension-wise merging function. Here we take the 2-dimension case of heap objects (D_H) and heap contexts (D_C), for illustration. Basically we can define arbitrary merging function in the form: $\pi_{H,C}(d_H, d_C) = (\widehat{d}, \widehat{d})$. However, in an object-sensitive setting, values of heap contexts are derived from values of heap objects, i.e. D_C is D_H's super-domain. Replacing values of D_H would change the values in D_C as well. How can we define a merging function on the values that depends on the results of this function? To resolve this recursive dependency, we propose **Incremental Merging**. Instead of defining π directly, we define a series of mergings, the first one merging a single domain and the others each merging a larger set of domains.

Here we introduce the 2-dimension case. Suppose we want to define a merging π on two domains D_1 and D_2, and D_2 is a super-domain of D_1.

We first merge in the D_1 independently by defining $\pi_1 : D_1 \to \widehat{D_1}$.

Since D_2 is a super-set of D_1, it is also changed according to transformation of rules. But we can know the changes in D_2 through the output of π_1. Then we make a second merging in changed domain $\widehat{D_2}$, but dependent on the merged values in $\widehat{D_1}$, by a function $\pi_2 : \widehat{D_1} \times \widehat{D_2} \to \widehat{D_{1,2}}$.

Thus, (d_1, d_2) in $D_1 \times D_2$ is merged to $(\widehat{d}, \widehat{d})$, where $\widehat{d} = \pi_2(\pi_1(d_1), \pi_1(d_2))$. The added argument $\widehat{D_1}$ in π_2 allows us to make different mergings on D_2 dependently on D_1, thus still capturing the correlation between the two domains.

Incremental Merging can be extended to general N-dimensional cases as well. We present it in Appendix A.

4.3 Properties of Mergings

We have introduced how to transform Datalog rules to apply merging, and proved its soundness. Then we need to choose a good merging.

The number of different mergings over a given domain is equivalent to the number of different partitions of its value set $\mathit{Bell}(n)$ (where n is the size of the set), which is prohibitively large[4]. In general, when more values are merged, the datalog program may run faster, but meanwhile it is more likely to lose the original precision. We can formalize that the precision of results is monotone to the mergings.

First we need to define a partial order over mergings.

Definition 3 (Partial Order of Mergings). *Given a set of N sensitivity domains $D_1 \ldots D_N$, a merging π_a of $D_1 \times \cdots \times D_N$ is finer than another merging*

[4] Bell number can be recursively calculated as $B(n+1) = \Sigma_{k=0}^{n} C_n^k B(n)$

π_b (and π_b is coarser than π_a) iff any tuple of elements merged into one tuple of abstract values in π_a are also merged into one tuple of abstract values in π_b.

$$\pi_a \succeq \pi_b \iff \forall x_1, y_1 \in D_1, \ldots, x_N, y_N \in D_N,$$
$$\pi_a(x_1, \ldots, x_N) = \pi_a(y_1, \ldots, y_N) \rightarrow \pi_b(x_1, \ldots, x_N) = \pi_b(y_1, \ldots, y_N).$$

We can also define the **Meet** and **Join** since merging values in a domain is equivalent to partitioning over its value set.

Thus, mergings of sensitive elements form a lattice. And the (transformed) Datalog program is a function on this lattice. Then we can prove the monotonicity of analysis results on this lattice (we present the proof in Appendix B).

Theorem 2 (Monotonicity). *Given a Datalog program C. If the merging π_b of the domains $D_1, \ldots, D_N$ is a finer merging than π_a, then applying π_b to C will deduce no fewer results than π_a. It means*

$$\pi_a \succeq \pi_b \rightarrow [\![C|\pi_a]\!]_o \subseteq [\![C|\pi_b]\!]_o$$

As is shown in the example in Section 2, the monotonicity is not strict and there are some mergings that generate just the same results as origin.

Definition 4 (Precision-preserving merging). *Given a Datalog program C. A merging π is a precision-preserving merging on iff $[\![C|\pi]\!]_o = [\![C]\!]_o$.*

All the precision preserving merging can keep the precision of the original result. Among these mergings, in order to improve efficiency, we want to find one that reduces the domain size as much as possible. So we define maximal merging as our target of finding mergings.

Definition 5 (Maximal Merging). *A precision-preserving merging π_a is a maximal merging iff there is no other precision preserving merging is coarser than π_a.*

5 Algorithm

We have proved soundness of domain-wise merging and defined maximal merging as a good merging. Our current goal is approximating the target merging.

5.1 Learn Merging Heuristics of Library Facts from Input Programs

The first challenge is that our defined maximal merging is specific to one given input program. It would be time-consuming if we use a pre-analysis to generate a maximal merging every time before analyzing a new program. So a general merging that can apply to different programs is preferable.

While it is impossible to find a universal merging that suits all kinds of input programs, the good news is that we can take advantage of the fact that large part of modern software is shared library code. Library code is usually large and analyzing it occupies a large portion of the analysis time. Among library functions, some functions are internal methods that are always called by other

library functions; some are called in a fixed pattern due to code paradigms: therefore, library codes share similar behaviour across different client codes.

Based on this observation, we can assume that if a merging can reduce running time with precision kept for analysis on a rather large number of input programs that share a library, it can also accelerate the analysis on other input programs using the same library. Thus, we can generate a merging heuristic for this library by learning from these input programs as a training set. Though there is no guarantee that precision is kept on new programs because they are not the same as the training programs, we can introduce fewer false positives by enlarging the training set.

When the user specifies a sensitivity domain to merge, 4DM first finds out all the values of the specified domain for each input program in the training set. It can be done either by collecting from input instances or running an original analysis to dump the sensitivity domain. Then it selects ones that are only related to the library from the union set of these values, and explores a merging on this set that reduces execution time and keeps precision for all input programs in the training set. We take the found merging as a merging heuristic for the library.

With a library heuristic, we can apply a merging to another program using this library, where only values specified by the heuristic are merged while other values stay unchanged.

5.2 Finding a Maximal Merging

How can we find a good merging on the set of library-related values? That is the remaining problem we need to solve in this part.

If we have some insight about the rules and domains we want to merge, we can use a specifically-designed rule to find a good merging. For example, we can use the heap equivalent automata from Tan et al. [21] as a guide to generate merging heuristics of $Heap$ domain in analysis.

And what if we do not have enough insight? We propose a highly general method `GenMax`. We use a greedy algorithm to partition these values iteratively. In each iteration, we find a maximal set of concrete values that can be replaced by one assigned abstract value while preserving the original precision. In this situation, no more values in this domain can be added into this set without losing precision, so they are excluded from the rest of values in following iterations. When all values are tested, the algorithm terminates.

This resulting merging is maximal, i.e. any two abstract values cannot be merged without losing precision. According to definition, there can be more than one maximal merging for the given set of concrete values, but evaluation in Section 6 shows the maximal merging found by our greedy algorithm is adequate.

To find a maximal merging set in every iteration, we explore two different approaches, an enumerative one and a randomized one.

Enumeration for Maximal Merging Set. In order to find the maximal number of mergeable concrete values, a direct method is to enumerate over every concrete value. As is shown in Algorithm-5.1, given the set of all concrete

Algorithm 5.1: Enumeration

```
Input  : Set of all concrete values 𝔼
Output: Current merging set N
Data: Set of concrete values outside merging set C
Data: Set of unchecked concrete values M
1  begin
2      N ← ∅;
3      C ← ∅;
4      M ← 𝔼;
5      for v ∈ 𝔼 do
6          M ← M − {v};
7          if merge N ∪ {v} preserves precision then
8              N ← N ∪ {v}
9          end
10         C ← C ∪ {v};
11     end
12 end
```

values $\mathbb{E}$, each time we randomly choose one unchecked value from $\mathbb{E}$, check whether it can be added into current merging set, until all values are checked. Note that the analysis is embedded with rules of merging through Section 4. In this method, number of calls to logic analysis is linear to the size of the set of abstract values. But the time complexity is worse than $O(|\mathbb{E}|)$ because runtime of each attempt of analysis also gets longer when input program grows larger. This method would be too time-consuming especially for large programs.

Active Learning for Maximal Merging Set. It would be better if we can try to add more than one concrete values into current merging set at once. Liang et al. [16] proposes an active learning algorithm, `ActiveCoarsen`, to find a minimal binary abstraction which is estimated to be more efficient. The algorithm is transformed in our setting and described in Algorithm-5.2.

Each time, `ActiveCoarsen` picks multiple concrete values by the ratio α, and tries to add them into current merging set. If failed, put the concrete values back and re-pick again. If we set $\alpha = e^{-\frac{1}{s}}$, where s is the size of the maximal merging set of concrete values, the expected number of calls to the analysis is $O(s \log n)$. However, we have no knowledge of the size s. Liang et al. [16] also introduce a mechanism for setting and updating α without knowledge of s, which is detailed in their article.

Optimization Though `ActiveCoarsen` tries more concrete values at a time, when the size of remaining concrete values in M is small, randomly selecting more than one value to merge at each trial would be less effective than plain enumeration. We approximately calculate that when the remaining size n is smaller than about $(1 + \sqrt{5})/2$ times the size s of minimal critical set, it would be better to switch to enumeration for minimal critical set.

Another optimization is that since each iteration we expand a merging set to max, the generated merging sets would get smaller since remaining concrete

Algorithm 5.2: ActiveCoarsen

```
Input  : Set of all concrete values 𝔼
Input  : Probability of random selection α
Output: Set of concrete values outside merging set N
Data: Set of undetermined concrete values M
Data: Set of random-selected concrete values T
1  begin
2      N ← ∅;
3      M ← 𝔼;
4      while M is not empty do
5          T ← select(M, α);
6          if merge N ∪ T preserves precision then
7              N ← N ∪ T;
8              M ← M − T;
9          end
10         update α;
11     end
12 end
```

values get fewer. The benefit-cost ratio would decrease a lot. So after finding several large merging sets, we cut the exploration of merging early.

6 Implementation and Evaluation

To evaluate the effectiveness and generality of our approach 4DM, we implemented 4DM in Python and compared it with existing approaches over two different Datalog-based analyses: points-to analysis and liveness analysis.

6.1 Points-to analysis

We first carried out an experiment on a context-sensitive points-to analysis over Java programs. The analysis is from the Doop framework [3]. We use 2-object-sensitive+heap (2o1h) as the sensitivity configuration in our evaluation because it is the most precise configuration on which most benchmark projects can be analysed with reasonable amount of time. We transformed the Soufflé implementation of Datalog rules as input of 4DM's merging.

In this experiment, we trained abstractions for all libraries in JDK. The abstraction is the Cartesian product of domains $Heap$ and $HeapContext$. Since in the setting of object-sensitivity, heap context is a super-domain of heap objects, we used incremental abstraction on them as described in Section 4.

We selected 15 Java projects as subjects from Dacapo benchmark [2] and pjbench [18]. We excluded some projects (*bloat*, *batik* and *jython*) from our subject set because it takes too long time to run 2o1h analysis on them.

To check whether 4DM is stable when using different training sets, we performed a 5-fold cross-validation over the selected projects. In particular, we

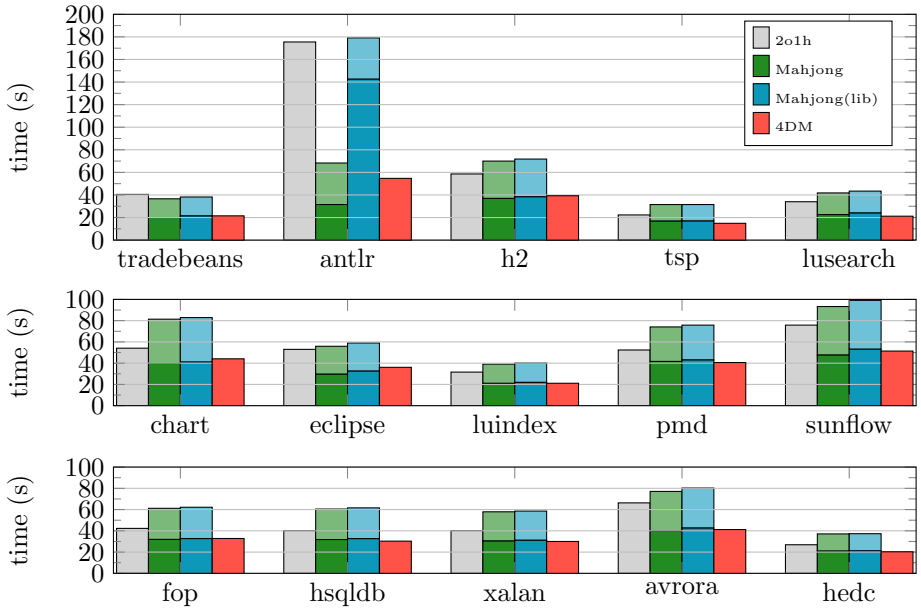


Fig. 4: Comparison of Execution Time (in sec)

randomly partitioned these projects into 5 groups, each containing 3 projects. In each fold of cross-validation, every 4 groups form a training set and the remaining one group forms a testing set. We apply the learned merging heuristics from the 12 programs to the rest 3 programs. Each column in Figure 4 and each segment in Table 1 show the results on one group by training on the other 4 groups (e.g., tradebeans, chart, and fop are a group). Though it takes 2 to 3 days to find a merging heuristic from a training set, the learnt heuristic can be reused across different programs in the testing set, and thus, the training process is offline. The execution time and precision measures in the following paragraphs are the results on the testing set.

In our evaluation, we compare the execution time and precision of 4DM with 3 existing approaches:

- Standard 2o1h analysis (denoted as *2o1h*). This is the baseline.
- The 2o1h analysis with Mahjong [21] (denoted as *Mahjong*). Its execution consists of two parts: the pre-analysis is a light-weight analysis to learn merging rules and the post-analysis is the 2o1h analysis based on the merging rules learned from the pre-analysis.
- The 2o1h analysis with a variant of Mahjong (denoted as *Mahjong(lib)*). The process is similar to *Mahjong*, but the difference is that all values outside JDK are excluded from the merging sets generated by pre-analysis.

In our evaluation, we try to answer the following three research questions:

RQ1: How effective our technique is on the acceleration of points-to analysis? To answer this question, we present the execution time of compared approaches in Figure 4. All the execution time is measured on a machine with 2

Intel Xeon Gold 6230 CPUs and 512GB RAM, equipped with Ubuntu 18.04 and JDK 1.7.0-80. All analyses are executed single-threaded. In the Figure, the execution time of each project is presented as a cluster of four columns. From left to right, the columns represent the execution time of *2o1h*, *Mahjong*, *Mahjong(lib)*, and *4DM* respectively. For *Mahjong* and *Mahjong(lib)*, the lighter-color part of the column represents the pre-analysis time, and the darker-color part of the column represents the post-analysis time.

From this figure, we can observe that our technique can significantly accelerate the points-to analysis compared with the standard implementation *2o1h* on all the experimented projects (with an average speedup of 1.6×).

Comparing with the post-analysis of *Mahjong*, *4DM* is faster in 6 of the 15 projects, and slower in 9. The difference in time is modest in most of the projects. But if we take *Mahjong*'s pre-analysis for each project into account, *4DM* is significantly faster than *Mahjong* on all projects. While merging learnt by *4DM* can be applied to different programs sharing the common library once it is obtained, the pre-analysis in *Mahjong* must be executed whenever analyzing a new project.

Since *4DM* only merges library elements, we also compare it with a variant of *Mahjong* which merges only heap objects in the library, and *4DM* is faster than *Mahjong(lib)*'s post-analysis in 10 of 15 projects. It implies that *4DM*'s data-driven method learnt a comparably good merging as the specifically-designed strategy in *Mahjong*.

RQ2: How much precision loss 4DM causes? To answer this question, we use two commonly used metrics for the precision of points-to analysis - polymorphic virtual call-sites (abbreviated as *poly*) and may-failing typecasts (abbreviated as *cast*), and compare experimented approaches on the two metrics. The more call-sites the analysis identified as polymorphic and the more typecasts the analysis identified as may-failing, the analysis is more imprecise.

The details are in Table 1. Columns *poly* and *cast* present the number of polymorphic virtual call-sites and may-failing typecasts detected by baseline *2o1h*, and Columns $\Delta poly$ and $\Delta cast$ refers to the number of additional false positives detected by other approaches compared with the baseline *2o1h*.

According to analysis results, the precision loss of our technique is minimal. Compared with standard *2o1h*, *4DM* causes no precision loss in 10 out of 15 projects. In the remaining projects, the highest precision loss happens in the *eclipse* project, with 3.0% extra polymorphic virtual call-sites reported. Comparing with *Mahjong*, though the maximal precision loss of *Mahjong* (2.0% in *cast* of *eclipse*) is smaller than *4DM*, it loses precision in *cast* in more projects than *4DM*. Thus, we can conclude that our approach is also comparable to *Mahjong* in precision loss.

RQ3: Is our approach stable when using different training sets? We need to check that if the training set changes, 4DM's learnt merging heuristic still reach a significant acceleration with minimal precision loss. From evaluation results in **RQ1** and **RQ2**, we can see that all the five sets of heuristics generated with five different training sets achieve significant acceleration and minimal pre-

Analysis	2o1h		Mahjong		Mahjong (lib)		4DM	
Program	poly	cast	Δpoly	Δcast	Δpoly	Δcast	Δpoly	Δcast
tradebeans	850	567	0	1	0	0	0	0
chart	1446	1279	0	1	0	0	3	3
fop	838	519	0	1	0	0	0	0
antlr	1643	640	0	1	0	0	0	0
eclipse	1318	1020	14	20	0	0	40	10
hsqldb	802	515	0	1	0	0	0	0
h2	942	559	0	2	0	0	0	0
luindex	929	549	0	1	0	0	0	0
xalan	808	514	0	1	0	0	0	0
tsp	784	441	0	1	0	0	0	0
pmd	886	911	0	1	0	0	0	0
avrora	936	715	0	1	0	0	2	0
lusearch	1136	596	0	1	0	0	0	0
sunflow	2000	1528	0	1	0	0	25	8
hedc	871	458	0	1	0	0	25	10

Table 1: Comparison of Precision Loss.

cision loss. Thus, we can conclude that our technique is stable across different training sets.

RQ4: What is the performance of our approach in large applications? We excluded 3 large projects (*bloat*, *batik* and *jython*) previously, as it takes a long time to run the training algorithm in Section 5 on them. But we can apply the learnt heuristics from smaller projects to accelerate the points-to analysis on large approaches. We apply the 5 heuristics learnt above to the 3 projects, and find that both *bloat* and *batik* are accelerated with a minimal precision loss (*bloat* speeds up by 1.0% with 0 precision loss, *batik* speeds up by 11.0% with precision loss less than 1%). However, *jython* still exceeds a 3-hour time limit. The details are in Appendix C.

Evaluation summary on points-to analysis. From the answers of the above research questions, we can see that (1) 4DM can significantly accelerate baseline *2o1h* with minimal precision loss; (2) compared with *Mahjong*, 4DM can achieve comparable efficiency and precision loss without performing pre-analysis for each new project; and (3) 4DM's efficiency and precision are stable when using different training sets.

6.2 Liveness analysis

In this subsection, we evaluate our method on an inter-procedure liveness analysis over Java programs to validate the generality of our approach.

To further challenge our approach on generality, we use a different domain, program points, as the sensitivity domain for liveness analysis. The observation is that for many program points, their live variable sets are exactly the same, i.e. there exists much redundant propagation in the analysis. If we view program

points as sensitivity elements, and apply domain-wise merging, we could speed up the analysis by reducing redundant computation from tuning flow-sensitivity.

In particular, in order to find appropriate merging for sensitivity elements, we use the following heuristic: if two program points are adjacent in the control flow graph, and the kill sets of them are empty, then the two program points are mergeable. By considering all adjacent pairs in the control flow graph, we could obtain our desired abstraction for the program.

Experiment setting. We evaluated our approach on 9 projects from Dacapo benchmark. They are divided into 3 groups and each group contains 3 projects. For each group, we learn a unique abstraction of the library code by applying 4DM, and test the library abstraction on the other two groups.

Results. Our approach accelerates the liveness analysis on all of the benchmarks. The average speed-up is 19.8%, and the average precision loss is 4.9%. The detailed performance is listed in Appendix D.

We stress that we only use a simple heuristic in a new domain, and the results show that our method still works remarkably well. It indicates that by carefully choosing domains, and applying various learning techniques, our method could speed up many other analyses in a way orthogonal to existing works.

7 Related Work

There have been many approaches proposed to accelerate program analysis. Among these approaches, three types are relevant to our paper.

Context reduction. Lhoták and Hendren [13] conducts an empirical study that demonstrates there are very few contexts related to the precision of the analysis results. Liang et al. [16] propose to finding the minimal set of call sites that preserve the precision. The result shows that most call sites does not affect the precision. Both Lhoták and Hendren [13] and Liang et al. [16] do not directly accelerate context-sensitive analysis, instead they see opportunities to reduce contexts for acceleration. Actually, the algorithm for finding minimal partitions in our paper are inspired by Liang et al. [16]

Based on the above observation, researchers [27, 11, 10, 14, 15, 26] propose to adjust context sensitivities at different program points to accelerate context-sensitive points-to analysis. There are two main directions. One is refinement, that is, iteratively increasing sensitivity on some program points on demand to improve precision. For example, Zhang et al. [27] starts from the most coarse abstraction and finds suitable program points to increase context length using an online SAT-solver-based algorithm, to reduce false positives. The other is coarsening, which analyzes the code beforehand, and performs the full analysis with coarser sensitivity at specific program points. For example, Li et al. [14] neglects context sensitivity in some methods which are calculated during pre-processing phase.

As is already mentioned in the overview, our domain-wise merging could merge redundant contexts in a fine-grained way that cannot be implemented by these approaches. On the other hand, our method considers more in a coarsening

direction as it merges concrete values. As a result, in the future potentially our approach could develop in another direction, combined with refinement-based methods. Furthermore, our work deals with not only context-sensitive points-to analysis but also more general cases.

Equivalence classes. The idea of equivalence classes has already been used to accelerate analysis. Cycle detection [8, 16, 9] is a common way to detect variable and object equivalence, which can reduce the number of variables or objects in points-to analysis. Tan et al. [21] uses an idea of equivalent automata to merging type-consistent heap objects. In addition, Xu and Rountev [25] and Xiao and Zhang [24] exploit context equivalence and use custom encoding to merge contexts, and are also considered as ways for context reduction. Our definition of domain-wise merging could be viewed as a generalization of the previous abstractions. Our approach can automatically learn the merging from a training set, in contrast to existing approaches relying on properties of certain analyses.

Library summarization. Library summarization techniques keep only necessary library facts in library summaries so as to accelerate client analysis. Client analysis can use summaries for reasoning, without inner reasoning for the library. Tang et al. [23, 22] propose conditional reachability to summarize all potential reachability between library boundaries. Polymer [12] learns library behavior from training clients to create conditional summaries for a library. However, a full summarization of the library is usually too large to store and load. For example, the summary produced by Tang et al. [22] needs tens of Gigabytes for some of the JDK classes over a simple data-dependence analysis. In contrast to these approaches that try to fully summarize the library, our work tries to learn a merging that is suitable for analyzing the library. The learnt merging heuristic of heap objects with heap contexts for the whole JDK library in our evaluation in Section 6.1 consist of only around 1000 lines of Datalog facts, respective for different training sets. Such a merging heuristic is easy to store and load.

Abstract Interpretation of Logic Programs. There is classic literature on extending abstract interpretation to logic programs [4–6, 1, 7]. Previous research on this topic mainly aims to analyze properties of logic programs themselves, such as variable binding and predicate type; while in 4DM, the analysis is expressed in a logic program but it analyzes properties of other programs. But the way 4DM transforms Datalog programs and merges analysis values can be viewed as abstracting the Datalog program expressing the analysis. It lifts the "concrete domain" of the original analysis values to the abstract domain of merged values. This is a special case of abstract interpretation of logic programs.

8 Conclusion

In this paper we have introduced 4DM, a new framework for tuning abstractions in a program analysis using domain-wise merging. In particular, it uses a data-driven method to automatically learn an effective merging heuristic for library code from a training set of programs. 4DM can be applied to merging different kinds of sensitivity elements in various analyses that are expressed in Datalog.

Our evaluation results show that our approach significantly accelerates a context-sensitive pointer analysis and a flow-sensitive liveness analysis with minimal precision loss.

Acknowledgements This work is supported in part by the National Key Research and Development Program of China No. 2019YFE0198100, National Natural Science Foundation of China under Grant Nos. 61922003, and a grant from ZTE-PKU Joint Laboratory for Foundation Software.

References

1. Barbuti, R., Giacobazzi, R., Levi, G.: A general framework for semantics-based bottom-up abstract interpretation of logic programs. ACM Transactions on Programming Languages and Systems (TOPLAS) **15**(1), 133–181 (1993)
2. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The dacapo benchmarks: Java benchmarking development and analysis. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications. pp. 169–190. OOPSLA '06, ACM, New York, NY, USA (2006)
3. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications. pp. 243–262 (2009)
4. Cortesi, A., Filé, G.: Abstract interpretation of logic programs: an abstract domain for groundness, sharing, freeness and compoundness analysis. In: ACM SIGPLAN Notices, Volume 26, Issue 9, PEPM '91. pp. 52–61. ACM (1991)
5. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. The Journal of Logic Programming **13**(2), 103–179 (1992)
6. Debray, S.K.: Global optimization of logic programs (analysis, transformation, compilation). (1987)
7. Delzanno, G., Giacobazzi, R., Ranzato, F.: Static analysis, abstract interpretation and verification in (constraint logic) programming. In: A 25-Year Perspective on Logic Programming, pp. 136–158. Springer (2010)
8. Fähndrich, M., Foster, J.S., Su, Z., Aiken, A.: Partial online cycle elimination in inclusion constraint graphs. In: Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998. pp. 85–96 (1998)
9. Hardekopf, B., Lin, C.: The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007. pp. 290–299 (2007)
10. Jeon, M., Jeong, S., Oh, H.: Precise and scalable points-to analysis via data-driven context tunneling. PACMPL **2**(OOPSLA), 140:1–140:29 (2018)
11. Jeong, S., Jeon, M., Cha, S.D., Oh, H.: Data-driven context-sensitivity for points-to analysis. PACMPL **1**(OOPSLA), 100:1–100:28 (2017)

12. Kulkarni, S., Mangal, R., Zhang, X., Naik, M.: Accelerating program analyses by cross-program training. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016. pp. 359–377 (2016)
13. Lhoták, O., Hendren, L.J.: Context-sensitive points-to analysis: Is it worth it? In: Compiler Construction, 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings. pp. 47–64 (2006)
14. Li, Y., Tan, T., Møller, A., Smaragdakis, Y.: Precision-guided context sensitivity for pointer analysis. PACMPL **2**(OOPSLA), 141:1–141:29 (2018)
15. Li, Y., Tan, T., Møller, A., Smaragdakis, Y.: Scalability-first pointer analysis with self-tuning context-sensitivity. In: Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018. pp. 129–140 (2018)
16. Liang, P., Tripp, O., Naik, M.: Learning minimal abstractions. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 31–42 (2011)
17. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (1999)
18. pjBench: Parallel java benchmarks (2014)
19. Shivers, O.: Control-flow analysis of higher-order languages (1991)
20. Smaragdakis, Y., Kastrinis, G., Balatsouras, G.: Introspective analysis: Context-sensitivity, across the board. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 485–495. PLDI '14, ACM, New York, NY, USA (2014)
21. Tan, T., Li, Y., Xue, J.: Efficient and precise points-to analysis: Modeling the heap by merging equivalent automata. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 278–291. PLDI 2017, ACM, New York, NY, USA (2017)
22. Tang, H., Wang, D., Xiong, Y., Zhang, L., Wang, X., Zhang, L.: Conditional dyck-cfl reachability analysis for complete and efficient library summarization. In: Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. pp. 880–908 (2017)
23. Tang, H., Wang, X., Zhang, L., Xie, B., Zhang, L., Mei, H.: Summary-based context-sensitive data-dependence analysis in presence of callbacks. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 83–95 (2015)
24. Xiao, X., Zhang, C.: Geometric encoding: forging the high performance context sensitive points-to analysis for java. In: Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011. pp. 188–198 (2011)
25. Xu, G., Rountev, A.: Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis. pp. 225–236. ISSTA '08, ACM, New York, NY, USA (2008)

26. Yan, H., Sui, Y., Chen, S., Xue, J.: Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. pp. 327–337. ACM (2018)
27. Zhang, X., Mangal, R., Grigore, R., Naik, M., Yang, H.: On abstraction refinement for program analyses in datalog. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. pp. 239–248. ACM (2014)

A Incremental Merging in N-domain-wise cases

Here we describe how to define an incremental merging function for general cases of N-domain-wise merging.

Suppose we want to define a merging function on N domains $D_1, D_2, \ldots, D_N$. Without loss of generality, we suppose D_i can only be a super-domain of D_j when $i > j$. We define π_i $(1 \leq i \leq N)$ by induction:

- When $i = 1$, we can define an arbitrary 1-domain-wise merging function on D_1.
- Suppose we have defined a function $\pi_i : D_1 \times \cdots \times D_i \to \widehat{D_{1\ldots i}}$, we can define $\pi_{i+1} : D_1 \times \cdots \times D_i \times D_{i+1} \to \widehat{D_{1\ldots i+1}}$ with a helper function π'_{i+1}:
$$\pi'_{i+1} : \widehat{D_{1\ldots i}} \times \widehat{D_{i+1}} \to \widehat{D_{1\ldots i+1}}$$
Here $\widehat{D_{i+1}}$ represents the changed domain of original D_{i+1} after applying the merging π_i, and $\pi_{i+1}(d_1, \ldots, d_i, d_{i+1}) = \pi'_{i+1}(\pi_i(d_1, \ldots, d_i), \pi_i(d_{i+1}))$. Note that $\pi_i(d_{i+1})$, which d_{i+1} would become under π_i, is determined after applying π_i to the Datalog program.

With $\pi = \pi_N$, we get an N-domain-wise merging function on $D_1 \times \cdots \times D_N$.

B Proof: the monotonicity of mergings

Theorem 3 (Monotonicity). *Given a Datalog program C. If the merging π_b of the domains $D_1, \ldots, D_N$ is a finer merging than π_a, then applying π_b to C will deduce no fewer results than π_a. It means*
$$\pi_a \succeq \pi_b \to [\![C|\pi_a]\!]_o \subseteq [\![C|\pi_b]\!]_o$$

Proof. Given a Datalog program C, any derivation on the proof tree in $[\![C|\pi_a]\!]$ has the form
$$R^0(\Pi_a(d_1^0), \Pi_a(d_2^0), \ldots) ::= \ldots, R^j(\Pi_a(d_1^j), \Pi_a(d_2^j), \ldots), \ldots.$$
(note that the concrete values $\{d_i^j\}$ may not originally match with each other, but get matched after applying Π_a).

As is defined, for any domain D, $\forall x, y \in D, \Pi_a(x) = \Pi_a(y) \to \Pi_b(x) = \Pi_b(y)$, so in $[\![C|\pi_b]\!]$'s proof tree,
$$R^0(\Pi_b(d_1^0), \Pi_b(d_2^0), \ldots) ::= \ldots, R^j(\Pi_b(d_1^j), \Pi_b(d_2^j), \ldots), \ldots.$$
also holds as a renaming of the previous instantiation.

Therefore, $[\![C|\pi_a]\!]_o \subseteq [\![C|\pi_b]\!]_o$.

C Detailed performance of 4DM in points-to analysis on large projects

Table 2 presents the performance of 4DM by applying learnt heuristics from small projects to large projects. Each column of 4DM shows the result of applying a heuristic learnt from a training set of 12 projects. The partition of training set is the same as in Section 6.1.

project	analysis	2o1h	4DM				
			1	2	3	4	5
bloat	time (s)	3122.32	3083.92	3093.46	3082.87	3097.3	3100.15
	poly	1577	1577	1577	1577	1577	1577
	cast	1526	1526	1526	1526	1526	1526
batik	time (s)	1001.57	908.73	894.49	882.18	886.73	879.52
	poly	4798	4836	4831	4846	4812	4828
	cast	2445	2473	2467	2476	2453	2459

Table 2: Performance of applying heuristics learnt by 4DM from small projects to large projects.

D Detailed performance of 4DM in liveness analysis

Table 3 shows the detailed results of applying 4DM to liveness analysis.

One practical concern in evaluation is that liveness analysis is usually very fast, hence random events in the processor could have a big influence on the measured run time of compiled executable. We tackle this problem by using Soufflé interpreter to run the analysis and obtain a more reliable record of analysis time.

We use a T-test to check whether the distribution of analysis time changes significantly after merging. The smaller the p-value, the stronger the evidence that two distributions are different. We perform 50 independent runs for each benchmark and analysis. The analysis time is the average of the 50 runs, and the p-value is calculated base on these data.

We measure the precision by the size of calculated live-variable set at all call sites. Note that since our approach is sound, this property is non-decreasing; and the less the size increases, the more precise the analysis is.

Table 3: Performance for liveness analysis. lib1 = {luindex, sunflow, hsqldb}, lib2 = {avrora, batik, bloat}, lib3 = {chart, lusearch, pmd}.

		avrora	batik	bloat	chart	lusearch	pmd	hsqldb	luindex	sunflow
original	analysis time (s)	5.09	13.37	5.14	7.81	3.36	4.95	6.71	3.33	6.33
	#call-sites live-var	43735	126230	73215	64332	43493	71099	68129	38380	50085
lib1	analysis time (s)	4.4	11.84	4.45	6.41	2.34	3.5	-	-	-
	speed up (%)	13.6	11.4	13.4	17.9	30.4	29.3	-	-	-
	p-value	2.58E-74	3.32E-69	1.81E-82	2.40E-91	7.31E-117	3.54E-114	-	-	-
	#call-sites live-var	45401	128879	75067	69117	45388	73587	-	-	-
	precision loss (%)	3.8	2.1	2.5	7.4	4.4	3.5	-	-	-
lib2	analysis time (s)	-	-	-	6.47	2.35	3.46	5.52	2.34	5.57
	speed up (%)	-	-	-	17.2	30.1	30.1	17.7	29.7	12.0
	p-value	-	-	-	4.16E-83	1.12E-114	4.48E-117	1.73E-82	7.08E-110	1.95E-72
	#call-sites live-var	-	-	-	66783	45790	73518	71003	40590	52675
	precision loss (%)	-	-	-	3.9	5.3	3.4	4.2	5.8	5.2
lib3	analysis time (s)	4.39	11.77	4.44	-	-	-	5.48	2.32	5.34
	speed up (%)	13.7	11.9	13.6	-	-	-	18.3	30.3	15.6
	p-value	1.70E-85	6.31E-72	6.78E-85	-	-	-	8.49E-93	5.12E-115	2.22E-84
	#call-sites live-var	45880	129707	75644	-	-	-	71334	40695	57910
	precision loss (%)	4.9	2.8	3.3	-	-	-	4.7	6.0	15.6