Data-Driven Context-Sensitivity for Points-to Analysis

SEHUN JEONG, Korea University, Republic of Korea
MINSEOK JEON*, Korea University, Republic of Korea
SUNGDEOK CHA, Korea University, Republic of Korea
HAKJOO OH†, Korea University, Republic of Korea

We present a new data-driven approach to achieve highly cost-effective context-sensitive points-to analysis for Java. While context-sensitivity has greater impact on the analysis precision and performance than any other precision-improving techniques, it is difficult to accurately identify the methods that would benefit the most from context-sensitivity and decide how much context-sensitivity should be used for them. Manually designing such rules is a nontrivial and laborious task that often delivers suboptimal results in practice. To overcome these challenges, we propose an automated and data-driven approach that learns to effectively apply context-sensitivity from codebases. In our approach, points-to analysis is equipped with a parameterized and heuristic rules, in disjunctive form of properties on program elements, that decide when and how much to apply context-sensitivity. We present a greedy algorithm that efficiently learns the parameter of the heuristic rules. We implemented our approach in the Doop framework and evaluated using three types of context-sensitive analyses: conventional object-sensitivity, selective hybrid object-sensitivity, and type-sensitivity. In all cases, experimental results show that our approach significantly outperforms existing techniques.

CCS Concepts: • **Theory of computation → Program analysis**; • **Computing methodologies → Machine learning approaches**;

Additional Key Words and Phrases: Data-driven program analysis, Points-to analysis, Context-sensitivity

ACM Reference Format:
Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-Driven Context-Sensitivity for Points-to Analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 100 (October 2017), 27 pages.
https://doi.org/10.1145/3133924

1 INTRODUCTION

Points-to analysis is one of the most important static program analyses. It approximates various memory locations that a pointer variable may point to at runtime. While useful as a stand-alone tool for many program verification tasks (e.g., detecting null-pointer dereferences), it is a key ingredient of subsequent higher-level program analyses such as static bug-finders, security auditing tools, and program understanding tools.

For object-oriented languages, context-sensitive points-to analysis is important as it must distinguish method's local variables and objects in different calling-contexts. For languages like Java,

*The first and second authors contributed equally to this work
†Corresponding author

Authors' email addresses: S. Jeong, gifaranga@korea.ac.kr; M. Jeon, minseok_jeon@korea.ac.kr; S. Cha, scha@korea.ac.kr; H. Oh, hakjoo_oh@korea.ac.kr.

2475-1421/2017/10-ART100
https://doi.org/10.1145/3133924

context-sensitivity has greater impact on the analysis precision than any other precision-improving techniques such as flow-sensitivity, and diverse forms of context-sensitivity have been proposed. Examples include call-site-sensitivity [Sharir and Pnueli 1981], object-sensitivity [Milanova et al. 2005], type-sensitivity [Smaragdakis et al. 2011], and selective hybrid object-sensitivity [Kastrinis and Smaragdakis 2013b].

However, application of context-sensitivity posts significant challenge. It is well-known that deep object-sensitive analyses, such as 2-object-sensitivity with a context-sensitive heap (*2objH*) [Milanova et al. 2005], usually achieve high precision in practice, but they generally do not scale well to large programs. A recent hybrid approach (*S2objH*) [Kastrinis and Smaragdakis 2013b], which combines object-sensitivity and call-site-sensitivity, shows improved performance than the *2objH* analysis, but scalability remains an issue. To address the scalability problem of deep context-sensitivity, Smaragdakis et al. [2014] proposed two analyses, namely *2objH+IntroA* and *2objH+IntroB*[1], which selectively apply context-sensitivity to a subset of method invocations using manually-tuned heuristic rules. However, these analyses are also far from optimal. The former achieves much improvements in scalability than *2objH* at the cost of precision. Likewise, the latter improves precision at the cost of scalability.

In this paper, we present an automated and data-driven approach to context-sensitive analysis. Our approach is similar to that of Smaragdakis et al. [2014] in that we selectively apply deep contexts only to a subset of methods. Difference, though, is that heuristic rules on context-selection are automatically generated from codebases through a learning algorithm. The model is a parameterized heuristic that is expressive enough to capture sophisticated properties of methods. We use a set of k boolean formulas: $\{f_1, f_2, \ldots, f_k\}$ (k is the maximum context depth to maintain) where f_i is a boolean combination of the atomic features that captures complex and high-level properties of a method. Each atomic feature describes a low-level property such as whether a method has an allocation statement or not. Context-sensitive analysis of depth i is applied only to the methods whose properties are described by f_i. Key technical challenge is to efficiently determine a good set of boolean formulas as brute-force search would simply be impractical. In this paper, we demonstrate that it is possible to reduce the problem of simultaneously learning k boolean formulas into a set of k sub-problems of finding each formula, drastically reducing the search space. In addition, we developed a greedy algorithm to solve each sub-problem, which produces accurate yet general formulas by iteratively refining the formulas while keeping them in disjunctive normal form.

The experimental results show that our data-driven approach produces highly cost-effective context-sensitive points-to analysis. We implemented our approach in the Doop framework [Bravenboer and Smaragdakis 2009] and applied it to three context-sensitive analyses: selective object-sensitivity [Kastrinis and Smaragdakis 2013b], object-sensitivity [Milanova et al. 2005], and type-sensitivity [Smaragdakis et al. 2011]. In all analyses, the results show that our approach strikes an unprecedented balance between precision and scalability trade-offs. For instance, when we applied our technique to selective 2-object-sensitivity (*S2objH*), the resulting analysis has virtually the same scalability of the context-insensitive analysis while enjoying most of the precision benefits. In particular, our data-driven points-to analysis far excels the performance of the existing state-of-the-art heuristics, introspective analyses [Smaragdakis et al. 2014], in terms of precision and speed.

In summary, our key contributions are as follows:

- We present a new approach to data-driven program analysis. Although the idea of data-driven program analysis itself is not new [Cha et al. 2016; Heo et al. 2016, 2017; Oh et al. 2015], we make two novel contributions: use of nonlinear model for context-selection heuristics (Section 3.3) and efficient learning algorithm (Section 3.5).

[1]In fact, the idea of the introspective analysis by [Smaragdakis et al. 2014] is applicable to any context-sensitive analysis.

Input Relations	
Alloc (*var*: V, *heap*: H, *inMeth*: M)	FormalArg($meth : M, i : \mathbb{N}, arg : V$)
Move($to : V, from : V$)	ActualArg($invo : I, i : \mathbb{N}, arg : V$)
Load($to : V, base : V, fld : F$)	FormalReturn($meth : M, ret : V$)
Store($base : V, fld : F, from : V$)	ActualReturn($invo : I, var : V$)
VCall($base : V, sig : S, invo : I, inMeth : M$)	ThisVar($meth : M, this : V$)
SCall($meth : M, invo : I, inMeth : M$)	HeapType($heap : H, type : T$)
	LookUp($type : T, sig : S, meth : M$)

Output Relations
VarPointsTo($var : V, ctx : C, heap : H, hctx : HC$)
CallGraph($invo : I, callerCtx : C, meth : M, calleeCtx : C$)
FldPointsTo($baseH : H, baseHCtx : HC, fld : F, heap : H, hctx : HC$)
InterProcAssign($to : V, toCtx : C, from : V, fromCtx : C$)
Reachable($meth : M, ctx : C$)

Fig. 1. Input and output relations of points-to analysis from [Kastrinis and Smaragdakis 2013c]

- We demonstrate the effectiveness of our approach with applications to three flavors of context-sensitive points-to analysis (Section 4.1): selective hybrid object-sensitivity, object-sensitivity, and type-sensitivity. We also demonstrate that use of nonlinear model is a key to success; without it, the analysis becomes significantly less precise and costly (Section 4.2).

2 PARAMETRIC POINTS-TO ANALYSIS IN DATALOG

In this section, we define a parametric context-sensitive points-to analysis for Java. We build on the previous work [Kastrinis and Smaragdakis 2013c] that defines a generic context-sensitive points-to analysis in Datalog. We incrementally extend the analysis to allow different context depths for each method. This section will use the same notations introduced by Kastrinis and Smaragdakis [2013c].

2.1 Points-to Analysis by Kastrinis and Smaragdakis [2013c]

We summarize the parametric points-to analysis designed by Kastrinis and Smaragdakis [2013c]. For more details, we refer the readers to prior work [Kastrinis and Smaragdakis 2013c; Smaragdakis and Balatsouras 2015].

In [Kastrinis and Smaragdakis 2013c], a Java program is represented as Datalog relations shown in Fig. 1. Input relations are grouped into instructions and auxiliary information. The meaning of the instructions is straightforward. For instance, Alloc relation models a heap allocation, where V, H, and M denote the sets of program variables, heap abstractions (i.e., allocation-sites), and method identifiers, respectively. F, S, and I denote the sets of fields,method signatures, and instructions, respectively. The auxiliary relations encode the name and type information. For instance, FormalArg encodes that *arg* is the i-th formal argument of *meth* (resp., the method at *invo*).

Given the input relations, the analysis derives the output relations listed in the bottom of Fig. 1. The VarPointsTo and CallGraph relations represent results of the context-sensitive points-to analysis. The former describes that the variable *var* in the call context *ctx* may points to the heap location *heap* whose heap context is *hctx*. Likewise, CallGraph(*invo*, *callerCtx*, *meth*, *calleeCtx*) encodes the context-sensitive call graph: the method *meth* can be invoked at the instruction *invo* with respect to the caller and callee contexts: *callerCtx* and *calleeCtx*.

```
InterProcAssign(to, calleeCtx, from, callerCtx) ←
  CallGraph(invo, callerCtx, meth, calleeCtx), FormalArg(meth, i, to), ActualArg(invo, i, from).

InterProcAssign(to, callerCtx, from, calleeCtx) ←
  CallGraph(invo, callerCtx, meth, calleeCtx), FormalReturn(meth, from), ActualReturn(invo, to).

Record(heap, ctx)=hctx,
VarPointsTo(var, ctx, heap, hctx) ← Reachable(meth, ctx), Alloc(var, heap, meth).

VarPointsTo(to, ctx, heap, hctx) ← Move(to, from), VarPointsTo(from, ctx, heap, hctx).

VarPointsTo(to, toCtx, heap, hctx) ←
  InterProcAssign(to, toCtx, from, fromCtx), VarPointsTo(from, fromCtx, heap, hctx).

VarPointsTo(to, ctx, heap, hctx) ←
  Load(to, base, fld), VarPointsTo(base, ctx, baseH, baseHCtx),
  FldPointsTo(baseH, baseHCtx, fld, heap, hctx).

FldPointsTo (baseH, baseHCtx, fld, heap, hctx) ←
  Store(base, fld, from), VarPointsTo(from, ctx, heap, hctx), VarPointsTo(base, ctx, baseH, baseHCtx).

Merge (heap, hctx, invo, callerCtx)=calleeCtx,
Reachable (toMeth, calleeCtx),
VarPointsTo (this, calleeCtx, heap, hctx),
CallGraph (invo, callerCtx, toMeth, calleeCtx) ← VCall (base, sig, invo, inMeth),
  Reachable (inMeth, callerCtx), VarPointsTo (base, callerCtx, heap, hctx),
  HeapType (heap, heapT), LookUp (heapT, sig, toMeth), ThisVar (toMeth, this).

MergeStatic (invo, callerCtx) = calleeCtx,
Reachable (toMeth, calleeCtx),
CallGraph (invo, callerCtx, toMeth, calleeCtx) ←
  SCall (toMeth, invo, inMeth), Reachable (inMeth, callerCtx).
```

(a) Points-to analysis rules taken from [Kastrinis and Smaragdakis 2013c]

```
Merge (depth, heap, hctx, invo, callerCtx)=calleeCtx,
Reachable (toMeth, calleeCtx),
VarPointsTo (this, calleeCtx, heap, hctx),
CallGraph (invo, callerCtx, toMeth, calleeCtx) ← VCall (base, sig, invo, inMeth),
  Reachable (inMeth, callerCtx), VarPointsTo (base, callerCtx, heap, hctx), HeapType (heap, heapT),
  LookUp (heapT, sig, toMeth), ThisVar (toMeth, this), ApplyDepth(toMeth, depth).

MergeStatic (depth, invo, callerCtx) = calleeCtx,
Reachable (toMeth, calleeCtx),
CallGraph (invo, callerCtx, toMeth, calleeCtx) ←
  SCall (toMeth, invo, inMeth), Reachable (inMeth, callerCtx), ApplyDepth(toMeth, depth).
```

(b) Modified rules for our parametric points-to analysis

Fig. 2. Datalog rules for context-sensitive points-to analysis

Fig. 2(a) shows the points-to analysis rules used by Kastrinis and Smaragdakis [2013c], which performs a flow-insensitive and context-sensitive points-to analysis with on-the-fly call-graph construction. The rules specify, for each instruction type, how to derive the output relations from the input relations. For instance, the fourth rule corresponds to the copy instruction.

The most important feature of the analysis is that context-sensitivity is encapsulated by the following three constructor functions:

- **Record**($heap : H, ctx : C$) produces new heap contexts. It is used when allocating heap objects (i.e., Alloc) and creates new heap contexts for them. Given an allocation-site and a calling-context, **Record** returns a new heap context for the heap object.
- **Merge**($heap : H, hctx : HC, invo : I, ctx : C$) creates calling contexts for virtual calls. Given heap object, heap context, call-site, and calling context, it creates a new context for called functions.
- **MergeStatic**($invo : I, ctx : C$) is similar to **Merge** but it is used for static method calls. Given a method call with a calling context, it creates a new calling context.

Kastrinis and Smaragdakis [2013c] showed that a large class of context-sensitive analyses (including k-call-site sensitivity, k-object-sensitivity, k-type-sensitivity, and their variants) can be obtained by appropriately defining the constructor functions and the domains (HC and C). For instance, we get the standard 2-object-sensitive analysis with 1-context-sensitive heap (*2objH*) by using allocation-sites as heap contexts (i.e., $HC = H$) and two allocation-sites as calling contexts (i.e., $C = H \times H$). The definitions of the constructor functions are as follows:

$$\begin{aligned} \textbf{Record}(heap, ctx) &= first(ctx) \\ \textbf{Merge}(heap, hctx, invo, ctx) &= pair(heap, hctx) \\ \textbf{MergeStatic}(invo, ctx) &= ctx \end{aligned}$$

At virtual method calls (**Merge**), the context is created by appending the receiver object (*heap*) and its heap context (*hctx*). Note that **Record** uses the first element of *ctx*; the new heap context of an object is the receiver object of the allocating method. At static calls (**MergeStatic**), the calling context of the caller method is used without changes.

2.2 Extension to Our Parametric Analysis

We extend the analysis rules to assign different context depths to different methods (in a similar way to the parametric framework by Milanova et al. [2005]). For this purpose, we extend the prior analysis in two ways. First, our analysis requires the extra input relation:

$$\textbf{ApplyDepth}(meth : M, depth : \mathbb{N}).$$

The **ApplyDepth** relation maps methods to their context depths; the method (*meth*) is analyzed with the given context-sensitivity depth (*depth*). In this section, we assume that the mapping (i.e., a set of **ApplyDepth** relations) is given for the target program. The heuristic that we define in Section 3 will be used to generate the relations.

Second, we need to modify the context constructors **Merge** and **MergeStatic** so that they produce new contexts by considering the given context depths as well:

$$\begin{aligned} &\textbf{Merge}(depth : \mathbb{N}, heap : H, hctx : HC, invo : I, ctx : C) = newCtx : C \\ &\textbf{MergeStatic}(depth : \mathbb{N}, invo : I, ctx : C) = newCtx : C \end{aligned}$$

With these new constructors, we replace the last two rules in Fig. 2(a) by the rules in Fig. 2(b). For instance, a virtual method call VCall($base, sig, invo, inMeth$) is handled as follows:

(1) VarPointsTo figures out a set of *heaps* that the *base* can point to.
(2) From each *heap*, a type identifier *heapT* is revealed.

(3) Using the identifier and the invocation's signature, the target method *toMeth* is found.
(4) ApplyDepth returns *depth* according to the *toMeth*, and it is provided to the Merge constructor.

MergeStatic is defined in a similar way but, in this case, SCall itself has the *toMeth* information. The other seven rules in Fig. 2(a) are used without changes.

All existing context-sensitive analyses expressible by the previous framework [Kastrinis and Smaragdakis 2013c] can be easily extended to our framework. For instance, consider the *2objH* analysis. We use the same definition for HC while C is modified to allow shallower depths; $C = (H \cup \{\star\}) \times (H \cup \{\star\})$ is the new context type. With these domains, the constructor functions are defined as follows:

$$\begin{aligned}
\textbf{Record}(heap, ctx) &= first(ctx), \\
\textbf{Merge}(depth, heap, hctx, invo, ctx) &= \begin{cases} pair(heap, hctx) & \text{if } depth = 2 \\ pair(heap, \star) & \text{if } depth = 1 \\ pair(\star, \star) & \text{if } depth = 0 \end{cases} \\
\textbf{MergeStatic}(depth, invo, ctx) &= \begin{cases} pair(first(ctx), second(ctx)) & \text{if } depth = 2 \\ pair(first(ctx), \star) & \text{if } depth = 1 \\ pair(\star, \star) & \text{if } depth = 0 \end{cases}
\end{aligned}$$

When $depth = 2$, note that the analysis is identical to *2objH*. When $depth = 1$, the **Merge** and **MergeStatic** truncate the contexts and maintain only the last context element (i.e., *1objH*). When *depth* is 0, the method is analyzed with context-insensitivity. We can use the same principle to transform any analysis in [Kastrinis and Smaragdakis 2013c] to our parametric setting.

3 OUR DATA-DRIVEN APPROACH

In this section, we present the core contributions of this paper: a new data-driven program analysis with a nonlinear model and an efficient learning algorithm. Specifically, our goal is to generate the input relations **ApplyDepth** appropriately for the given program, which assigns the context depths in $[0, k]$ to each method in the program, where k is a pre-defined, maximum context depth. Our approach is data-driven; from a given codebase, we learn a heuristic to populate the **ApplyDepth** relations. The learned heuristic is then used for analyzing new programs. Though we have points-to analysis in mind, our approach in this section is general and applicable to other program analyses as well.

3.1 Modeling of Context-Sensitive Points-to Analysis

To formalize our approach, we abstractly model the parametric context-sensitive points-to analysis in Section 2. Let $P \in \mathbb{P}$ be a program to analyze. Let $\mathbb{M}_P$ be the set of methods in P. Let k be the maximum depth for context-sensitivity (e.g., $k = 2$ in our experiments). Then, we define the set $\mathcal{A}_P$ of abstractions for P as follows:

$$\mathbf{a} \in \mathcal{A}_P = \{0, 1, \ldots, k\}^{\mathbb{M}_P}.$$

Abstractions are vectors of natural numbers in $\{0, 1, \ldots, k\}$ with indices in $\mathbb{M}_P$, and are ordered pointwise:

$$\mathbf{a} \sqsubseteq \mathbf{a}' \iff \forall m \in \mathbb{M}_P.\ \mathbf{a}_m \leq \mathbf{a}'_m.$$

Intuitively, $\mathbf{a}_m = i$ means that the method $m \in \mathbb{M}_P$ is analyzed with i-context-sensitivity (i.e., the analysis distinguishing the last i context elements of the method). Our method can be used with any kind of context abstractions, e.g., call-site-sensitivity [Sharir and Pnueli 1981], object-sensitivity [Milanova et al. 2005], type-sensitivity [Smaragdakis et al. 2011], etc, and this section

does not concern about what kind of context-sensitivity is used. We can regard an abstraction $\mathbf{a} \in \mathcal{A}_P$ as a function from $\mathbb{M}_P$ to $\{0, 1, \ldots, k\}$:

$$\mathbf{a} \in \mathcal{A}_P = \mathbb{M}_P \to \{0, 1, \ldots, k\}.$$

We write $\mathbf{k}$ and $\mathbf{0}$ for the most precise and least precise abstractions, respectively:

$$\mathbf{k} = \lambda m \in \mathbb{M}_P.\ k, \qquad \mathbf{0} = \lambda m \in \mathbb{M}_P.\ 0$$

For instance, when we use object-sensitivity for context abstraction, the analysis with $\mathbf{k}$ represents the standard k-object-sensitive analysis while $\mathbf{0}$ means the context-insensitive analysis.

We assume that a set $\mathbb{Q}_P$ of assertions is given together with P. For instance, in our experiments, $\mathbb{Q}_P$ is the set of all type casts in P and the analysis attempts to prove that they do not fail at runtime. We model points-to analysis for P by the function:

$$F_P : \mathcal{A}_P \to \wp(\mathbb{Q}_P) \times \mathbb{N}.$$

Given a program P, the analysis takes an abstraction $\mathbf{a} \in \mathcal{A}_P$ of the program and returns a pair (Q, n) of the set $Q \subseteq \mathbb{Q}_P$ of assertions proved by the analysis and the natural number $n \in \mathbb{N}$ that represents the cost (e.g., time) of the analysis with the abstraction $\mathbf{a}$. For instance, Q denotes the set of type casts proved to be safe by the analysis. We define two projection functions: $\mathsf{proved}(F_P(\mathbf{a}))$ and $\mathsf{cost}(F_P(\mathbf{a}))$ denote the set of proved assertions (Q) and the cost (n) of the analysis $F_P(\mathbf{a})$, respectively.

In this section, we assume that the analysis is monotone in the following sense:

Definition 3.1 (Monotonicity of Analysis). Let $P \in \mathbb{P}$ be a program and $\mathbf{a}, \mathbf{a}' \in \mathcal{A}_P$ be abstractions of P. We say the analysis F_P is monotone if the following condition holds:

$$\mathbf{a} \sqsubseteq \mathbf{a}' \implies \mathsf{proved}(F_P(\mathbf{a})) \subseteq \mathsf{proved}(F_P(\mathbf{a}')).$$

That is, we assume that more precise abstractions lead to proving more assertions. Note that our notion of "proved assertions" means those in the original program, not datalog facts derived by the analysis. Therefore, the number of proved assertions goes up as the analysis precision increases, while the number of derived datalog facts goes down. The parametric analysis in Section 2 satisfies this property.

3.2 Goal

Suppose we have a codebase $\mathbf{P} = \{P_1, P_2, \ldots, P_m\}$, which is a collection of programs. Our goal is to automatically learn from $\mathbf{P}$ a context-selection heuristic $\mathcal{H}$:

$$\mathcal{H}(P) : \mathbb{M}_P \to \{0, 1, \ldots, k\}$$

which takes a program P and returns an abstraction (i.e., an assignment of context depths to each method) of the program. Once $\mathcal{H}$ is learned from the codebase, it is used to analyze previously unseen program P as follows:

$$F_P(\mathcal{H}(P)).$$

Our aim is to learn from $\mathbf{P}$ a good heuristic $\mathcal{H}$ such that the precision of the analysis $F_P(\mathcal{H}(P))$ is close to that of the most precise analysis $F_P(\mathbf{k})$ while its cost is comparable to that of the least precise analysis $F_P(\mathbf{0})$.

3.3 Modeling of Context-Selection Heuristics

To enable learning, we first need to define a hypothesis space of the selection heuristics, which is called model or inductive bias in the machine learning community. That is, we need to choose and represent a model which is a restricted subset of the entire selection heuristics. We use a nonlinear, disjunctive model that combines atomic features with boolean formulas. Use of the nonlinear model is a key to success in our approach; the linear model used in prior work [Oh et al. 2015] is not expressive enough to capture useful context-selection heuristics required for points-to analysis for Java (Section 4).

We assume that a set of *atomic features* is given: $\mathbb{A} = \{a_1, a_2, \ldots, a_n\}$. An atomic feature a_i describes a property of methods; it is a function from programs to predicates on methods:

$$a_i(P) : \mathbb{M}_P \to \{true, false\}.$$

The atomic features we used in experiments are described in Section 3.6. We define the following set of boolean formulas over the atomic features:

$$f \to true \mid false \mid a_i \in \mathbb{A} \mid \neg f \mid f_1 \wedge f_2 \mid f_1 \vee f_2$$

Given a program P, a boolean formula f means a set of methods:

$$\begin{array}{ll} [\![true]\!]_P = \mathbb{M}_P & [\![\neg f]\!]_P = \mathbb{M}_P \setminus [\![f]\!]_P \\ [\![false]\!]_P = \emptyset & [\![f_1 \wedge f_2]\!]_P = [\![f_1]\!]_P \cap [\![f_2]\!]_P \\ [\![a_i]\!]_P = \{m \in \mathbb{M}_P \mid a_i(P)(m) = true\} & [\![f_1 \vee f_2]\!]_P = [\![f_1]\!]_P \cup [\![f_2]\!]_P \end{array}$$

Suppose we are given a vector Π of k boolean formulas:

$$\Pi = \langle f_1, \ldots, f_k \rangle.$$

This vector will become the parameter of our model. Given a parameter $\Pi = \langle f_1, \ldots, f_k \rangle$, we define the model (i.e., parameterized heuristic), denoted $\mathcal{H}_\Pi$, as follows:

$$\mathcal{H}_\Pi(P) = \lambda m \in \mathbb{M}_P. \begin{cases} k & \text{if } m \in [\![f_k]\!]_P \\ k-1 & \text{if } m \in [\![f_{k-1}]\!]_P \wedge m \notin [\![f_k]\!]_P \\ \cdots & \\ k-i & \text{if } m \in [\![f_{k-i}]\!]_P \wedge m \notin \bigcup_{k \geq j > k-i} [\![f_j]\!]_P \\ \cdots & \\ 1 & \text{if } m \in [\![f_1]\!]_P \wedge m \notin \bigcup_{k \geq j > 1} [\![f_j]\!]_P \\ 0 & \text{otherwise} \end{cases}$$

Given P, the parameterized heuristic assigns a context depth j to each method, where the depth j is determined according to the model parameter Π. A method m is assigned the depth j if the j-th boolean formula f_j of Π includes the method m, i.e., $m \in [\![f_j]\!]_P$, and m is not implied by any other formulas $f_{j+1}, f_{j+2}, \ldots, f_k$ at higher levels. That is, when m belongs to both f_i and f_j ($i > j$), we favor assigning the greater context-depth i to m.

3.4 The Learning Problem

Once we define a model $\mathcal{H}_\Pi$, learning a good context-selection heuristic corresponds to finding a good model parameter Π. Given a codebase $\mathbf{P} = \{P_1, \ldots, P_m\}$ and the model $\mathcal{H}_\Pi$, we define the learning problem as the following optimization problem:

$$\text{Find } \Pi \text{ that minimizes } \sum_{P \in \mathbf{P}} \text{cost}(F_P(\mathcal{H}_\Pi(P))) \text{ while satisfying } \frac{\sum_{P \in \mathbf{P}} |\text{proved}(F_P(\mathcal{H}_\Pi(P)))|}{\sum_{P \in \mathbf{P}} |\text{proved}(F_P(\mathbf{k}))|} \geq \gamma. \quad (1)$$

That is, we aim to find a parameter Π that minimizes the cost of the analysis over the codebase while satisfying the precision constraint, $\frac{\sum_{P \in \mathbf{P}} |\text{proved}(F_P(\mathcal{H}_\Pi(P)))|}{\sum_{P \in \mathbf{P}} |\text{proved}(F_P(\mathbf{k}))|} \geq \gamma$, which asserts that the ratio of

the number of assertions proved by the analysis with Π to the number of assertions proved by the most precise analysis must be higher than a predefined threshold $\gamma \in [0, 1]$. For instance, setting γ to 0.9 means that we would like to ensure 90% of the full precision.

Although we assume a single client (e.g. safety of type casts) for presentation brevity, the optimization problem can be defined for multiple clients. Suppose we have n clients, each of which is accompanied with the corresponding projection function $\text{proved}_i (1 \leq i \leq n)$. Then, we can redefine the precision constraint by, for example, $\frac{1}{n}\sum_{j=1}^{n} \frac{\sum_{P\in\mathbf{P}} |\text{proved}_j(F_P(\mathcal{H}_\Pi(P)))|}{\sum_{P\in\mathbf{P}} |\text{proved}_j(F_P(\mathbf{k}))|} \geq \gamma$, where we evaluate the overall performance by averaging the results.

3.5 The Learning Algorithm

Note that solving the optimization problem in Equation (1) is extremely challenging. This is mainly because the space of parameters is intractably large. A model parameter Π consists of k boolean formulas. Assuming that $\mathbb{S}$ is the space of possible boolean formulas over which we learn, searching for k formulas simultaneously poses the huge search space of size $|\mathbb{S}|^k$. This space is typically too large to enable effective learning even for small k.

Overall Algorithm. We present a learning algorithm that drastically reduces the size of the search space from $|\mathbb{S}|^k$ to $k \cdot |\mathbb{S}|$. To do so, we first decompose the optimization problem in Equation (1) into k sub-problems: $\Psi_k, \Psi_{k-1}, \ldots, \Psi_1$. Note that the solution of the original problem is a vector of k boolean formulas: $\Pi = \langle f_1, \ldots, f_k \rangle$. In our approach, solving the sub-problem Ψ_i $(1 \leq i \leq k)$ produces the i-th boolean formula f_i of Π. Therefore, we solve the problems Ψ_i $(1 \leq i \leq k)$ separately and combine their solutions f_i $(1 \leq i \leq k)$ to form $\Pi = \langle f_1, \ldots, f_k \rangle$.

The solution f_i for the problem Ψ_i is defined in terms of $f_{i+1}, f_{i+2}, \ldots, f_k$, i.e., the solutions of the problems $\Psi_{i+1}, \Psi_{i+2}, \ldots, \Psi_k$ at higher levels. Suppose we already solved the problems $\Psi_{i+1}, \Psi_{i+2}, \ldots, \Psi_k$ and have their solutions $f_{i+1}, f_{i+2}, \ldots, f_k$. Then, the problem Ψ_i is defined as follows:

$$\Psi_i \equiv \text{ Find } f \text{ that minimizes } \sum_{P\in\mathbf{P}} \text{cost}(F_P(\mathcal{H}_{\Pi_i}(P))) \text{ while satisfying } \frac{\sum_{P\in\mathbf{P}} |\text{proved}(F_P(\mathcal{H}_{\Pi_i}(P)))|}{\sum_{P\in\mathbf{P}} |\text{proved}(F_P(\mathbf{k}))|} \geq \gamma. \tag{2}$$

where $\Pi_i = \langle \mathit{true}, \ldots, \mathit{true}, f, f_{i+1}, f_{i+2}, \ldots, f_k \rangle$. That is, when we solve the problem Ψ_i, we fix the currently available solutions $f_{i+1}, f_{i+2}, \ldots, f_k$ and attempts to find a formula f that achieves the best performance with respect to $f_{i+1}, f_{i+2}, \ldots, f_k$. Note that the first $i-1$ formulas of Π_i is *true*; according to the definition of $\mathcal{H}_\Pi$, this means that we apply the context depth $i-1$ to all remaining methods that are not selected by $f, f_{i+1}, f_{i+2}, \ldots, f_k$.

Since solving the problem Ψ_i requires to solve the higher-level problems Ψ_j $(j > i)$, we proceed in decreasing order from k to 1: We first solve the problem Ψ_k and use the result when we solve the problem Ψ_{k-1}, and so on. Let f_i be the solution of the problem Ψ_i $(1 \leq i \leq k)$. Then, the solution Π of the original problem in Equation (1) is simply obtained by combining the sub-solutions f_i's: $\Pi = \langle f_1, f_2, \ldots, f_k \rangle$.

Algorithm 1 presents the learning algorithm. It takes as input static analyzer F, codebase $\mathbf{P}$, context-depth k, and atomic features $\mathbb{A}$. A vector $\langle f_1, f_2, \ldots, f_k \rangle$ of boolean formulas is returned. At line 2, the formulas are initialized with *true*. At lines 3–5, it iterates the context depths $k, k-1, \ldots, 1$ in decreasing order and updates the boolean formula f_i of the current depth i. The update is done by invoking the function LearnBooleanFormula, which we describe shortly.

Property of Our Algorithm. Before explaining how LearnBooleanFormula works, we point out that while our learning approach reduces the search space significantly, it does not lose a

chance of finding good solutions. Specifically, our algorithm guarantees to preserve a *minimal* solution of the original problem (1). Let us first define the notion of minimal solutions.

Definition 3.2. Let $\mathbf{P}$ be a codebase and $\Pi = \langle f_1, f_2, \ldots, f_k \rangle$ be a parameter. We say Π is a minimal solution of the problem (1) if

(1) Π meets the precision constraint: $\frac{\sum_{P \in \mathbf{P}} |\text{proved}(F_P(\mathcal{H}_\Pi(P)))|}{\sum_{P \in \mathbf{P}} |\text{proved}(F_P(\mathbf{k}))|} \geq \gamma$, and

(2) there exists no solution smaller than Π: if Π' is a parameter that meets the precision constraint, i.e., $\frac{\sum_{P \in \mathbf{P}} |\text{proved}(F_P(\mathcal{H}_{\Pi'}(P)))|}{\sum_{P \in \mathbf{P}} |\text{proved}(F_P(\mathbf{k}))|} \geq \gamma$, and Π' is smaller than Π, i.e., $\forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'}(P) \sqsubseteq \mathcal{H}_\Pi(P)$, then Π' and Π are equivalent:

$$\forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'}(P) = \mathcal{H}_\Pi(P).$$

In a similar way, we can define the notion of minimal solutions for the sub-problems as follows:

Definition 3.3. Let $\mathbf{P}$ be a codebase and f_i be the solution of the problem Ψ_i. Let Π_i be the vector

$$\langle \mathit{true}, \ldots, \mathit{true}, f_i, f_{i+1}, \ldots, f_k \rangle$$

where $f_{i+1}, \ldots, f_k$ are solutions of problems $\Psi_{i+1}, \ldots, \Psi_k$, respectively. We say f_i is minimal if

(1) Π_i meets the precision constraint: $\frac{\sum_{P \in \mathbf{P}} |\text{proved}(F_P(\mathcal{H}_{\Pi_i}(P)))|}{\sum_{P \in \mathbf{P}} |\text{proved}(F_P(\mathbf{k}))|} \geq \gamma$, and

(2) Π_i is minimal: if $\Pi_i' = \langle \mathit{true}, \ldots, \mathit{true}, f_i', f_{i+1}, \ldots, f_k \rangle$ is a parameter that meets the precision constraint, i.e., $\frac{\sum_{P \in \mathbf{P}} |\text{proved}(F_P(\mathcal{H}_{\Pi_i'}(P)))|}{\sum_{P \in \mathbf{P}} |\text{proved}(F_P(\mathbf{k}))|} \geq \gamma$, and Π_i' is smaller than Π_i, i.e., $\forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi_i'}(P) \sqsubseteq \mathcal{H}_{\Pi_i}(P)$, then Π_i' and Π_i are equivalent:

$$\forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi_i'}(P) = \mathcal{H}_{\Pi_i}(P).$$

Theorem 3.4 below states that our stepwise learning algorithm is able to produce a minimal solution of the original problem if each formula f_i is a minimal solution of the problem Ψ_i.

Theorem 3.4. *Let $f_1, \ldots, f_k$ be minimal solutions of the problems $\Psi_1, \ldots, \Psi_k$. Then, $\langle f_1, \ldots, f_k \rangle$ is a minimal solution of the original problem (1).*

Proof. See Appendix A. □

Learning Boolean Formulas. Now we explain LearnBooleanFormula, which is used to solve each sub-problem Ψ_i. Note that the search space of the sub-problem Ψ_i is still huge; there are 2^{2^n} semantically different boolean functions over n boolean variables (i.e., the number of atomic features $\mathbb{A} = \{a_1, \ldots, a_n\}$). Therefore, it is intractable to exhaustively search for a good solution. To address this challenge, we developed a greedy search algorithm that produces good-enough solutions in practice.

Algorithm 2 presents our algorithm for learning a boolean formula f_i for each problem Ψ_i. The algorithm takes as input the current context-depth i, current formulas $\langle f_1, \ldots, f_k \rangle$, static analyzer F, codebase $\mathbf{P}$, and atomic features $\mathbb{A} = \{a_1, \ldots, a_n\}$. When the algorithm is used for solving the i-th problem (i.e., Ψ_i), we assume that the solutions $f_{i+1}, f_{i+2}, \ldots, f_k$ of the problems $\Psi_{i+1}, \Psi_{i+2}, \ldots, \Psi_k$ are already computed (this is ensured by Algorithm 1).

Given these inputs, the algorithm produces as output a boolean formula f in disjunctive normal form (DNF); f is a disjunction of conjunctions of literals:

$$f = \bigvee_x \bigwedge_y l_{x,y}$$

where a literal $l_{x,y}$ includes boolean constants, atomic features $a_j \in \mathbb{A}$, and their negations $\neg a_j$. In the algorithm, we represent a conjunctive clause (i.e., a conjunction of literals) by a set of literals, and a disjunction by a set of clauses.

Algorithm 1 Our Learning Algorithm

Input: Static analyzer F, codebase $\mathbf{P}$, context-depth k, atomic features $\mathbb{A}$
Output: A vector $\langle f_1, f_2, \ldots, f_k \rangle$ of k boolean formulas

1: **procedure** LEARN($F, \mathbf{P}, k$)
2: $\langle f_1, f_2, \ldots, f_k \rangle \leftarrow \langle \mathit{true}, \mathit{true}, \ldots, \mathit{true} \rangle$ ▷ initialize $f_1, f_2, \ldots, f_k$ with *true*
3: **for** $i = k$ to 1 **do**
4: $f_i \leftarrow$ LEARNBOOLEANFORMULA($i, \langle f_1, f_2, \ldots, f_k \rangle, F, \mathbf{P}, \mathbb{A}$) ▷ update f_i
5: **end for**
6: **return** $\langle f_1, f_2, \ldots, f_k \rangle$
7: **end procedure**

Algorithm 2 Algorithm for Learning a Boolean Formula

Input: Context-depth i, current formulas $\langle f_1, f_2, \ldots, f_k \rangle$, static analyzer F, codebase $\mathbf{P}$, atomic features $\mathbb{A}$
Output: Boolean formula f_i in disjunctive normal form

1: **procedure** LEARNBOOLEANFORMULA($i, \langle f_1, f_2, \ldots, f_k \rangle, F, \mathbf{P}, \mathbb{A}$)
2: $f \leftarrow \{\{a_j\} \mid a_j \in \mathbb{A}\} \cup \{\{\neg a_j\} \mid a_j \in \mathbb{A}\}$ ▷ initial formula
3: $W \leftarrow f$ ▷ initial workset (the set of all clauses in f)
4: $\mathit{bestCost} \leftarrow \infty$ ▷ initial best cost
5: **while** $W \neq \emptyset$ **do**
6: $c \leftarrow \mathsf{ChooseClause}(W, F, \mathbf{P})$ ▷ choose the most expensive clause from W
7: $W \leftarrow W \setminus \{c\}$
8: $a \leftarrow \mathsf{ChooseAtom}(\mathbb{A}, c, F, \mathbf{P})$ ▷ choose an atom from $\mathbb{A}$
9: $c' \leftarrow c \cup \{a\}$ ▷ refined clause
10: $f' \leftarrow (f \setminus \{c\}) \cup \{c'\}$ ▷ refined formula
11: $\Pi \leftarrow \langle f_1, \ldots, f_{i-1}, f', f_{i+1}, f_{i+2}, \ldots, f_k \rangle$ ▷ current parameter setting
12: $(\mathit{proved}, \mathit{cost}) \leftarrow \mathsf{Analyze}(\Pi, F, \mathbf{P})$
13: **if** $\mathit{cost} \leq \mathit{bestCost} \wedge \frac{|\mathit{proved}|}{\sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma$ **then** ▷ cheaper parameter found
14: $\mathit{bestCost} \leftarrow \mathit{cost}$ ▷ update the best cost
15: **if** $(f' \iff f \setminus \{c\})$ **then** ▷ check if f' is semantically refined
16: $f \leftarrow f' \setminus \{c'\}$ ▷ remove chosen clause from f
17: **continue**
18: **else**
19: $W \leftarrow W \cup \{c'\}$ ▷ c' can be refined further
20: $f \leftarrow f'$ ▷ update the formula
21: **end if**
22: **end if**
23: **end while**
24: **return** f
25: **end procedure**

At line 2, the algorithm initializes the formula f with a disjunction of all atomic features and their negations:

$$f = a_1 \vee a_2 \vee \cdots \vee a_n \vee \neg a_1 \vee \neg a_2 \vee \cdots \vee \neg a_n$$

Note that this formula denotes the set of all methods in the program, and therefore the initial formula leads to the most precise analysis that assigns the context depth i to every method (except for the methods already selected by $f_{i+1}, \ldots, f_k$). Beginning with this formula f, the goal of our algorithm is to refine each clause of f and obtain a boolean formula that minimizes the analysis cost while preserving the precision constraint (e.g., achieving 90% of the full precision).

To do so, the algorithm maintains a workset W which is a set of clauses to refine further. The workset initially contains all atomic clauses (line 3). The algorithm iterates while the workset is non-empty. At lines 6 and 7, a clause is selected and removed from the workset. Our algorithm is greedy in a sense that the ChooseClause function chooses the most expensive clause c from W:

$$\mathsf{ChooseClause}(W, F, \mathbf{P}) = \underset{c \in W}{\mathrm{argmax}} \sum_{P \in \mathbf{P}} \mathrm{cost}(F_P(\mathcal{H}_{\Pi_c}(P)))$$

where $\Pi_c = \langle f_1, \ldots, f_{i-1}, c, f_{i+1}, f_{i+2}, \ldots, f_k \rangle$. The heuristic, $\mathcal{H}_{\Pi_c}$, with Π_c assigns the context depth i to the methods for which c is true (except for methods for which some of $f_{i+1}, \ldots, f_k$ are true). All the other methods are assigned the depth $i-1$, because LearnBooleanFormula is invoked with $f_1, \ldots, f_{i-1}$ being *true*.

The next step is to refine the clause c by conjoining an atom $a \in \mathbb{A}$ to c (lines 8 and 9): i.e., $c' = c \wedge a$. The refined clause c' represents a smaller set of methods than c, which decreases the precision of the analysis. When refining the clause, our algorithm is conservative and chooses the atom $a \in \mathbb{A}$ with which refining c decreases the analysis precision as little as possible. More precisely, the ChooseAtom function is defined as follows:

$$\mathsf{ChooseAtom}(\mathbb{A}, c, F, \mathbf{P}) = \begin{cases} \underset{a \in (\mathbb{A} \cup \neg\mathbb{A}) \setminus c}{\mathrm{argmax}} \sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_{\Pi_{a \wedge c}}(P)))| & \text{if } (\mathbb{A} \cup \neg\mathbb{A}) \setminus c \neq \emptyset \\ \mathit{false} & \text{otherwise} \end{cases}$$

where $\Pi_{a \wedge c} = \langle f_1, \ldots, f_{i-1}, a \wedge c, f_{i+1}, f_{i+2}, \ldots, f_k \rangle$ and $\neg\mathbb{A} = \{\neg a \mid a \in \mathbb{A}\}$. When there exists an atom to choose (i.e., $\mathbb{A} \setminus c \neq \emptyset$), we conservatively choose the atom a with the greatest precision. Otherwise, there is no atom to refine with and *false* is returned so that the clause c does not get refined further. In the latter case, the algorithm eventually goes to line 17(because $f' \iff f \setminus \{c\}$ is valid) and attempts to choose another clause to refine. When an atom a is successfully chosen, we refine the clause (line 9) and the formula (line 10).

At lines 11–12, the refined formula is evaluated. We first construct the parameter setting Π with the current formula f' (line 11):

$$\Pi = \langle f_1, \ldots, f_{i-1}, f', f_{i+1}, f_{i+2}, \ldots, f_k \rangle.$$

Next, we analyze the programs in the codebase with Π. The Analyze function returns the set of queries proved and the cost spent with the parameter Π:

$$\mathsf{Analyze}(\Pi, F, \mathbf{P}) = (\sum_{P \in \mathbf{P}} \mathsf{proved}(F_P(\mathcal{H}_\Pi(P))), \sum_{P \in \mathbf{P}} \mathrm{cost}(F_P(\mathcal{H}_\Pi(P))).$$

At line 13, we check whether the cost is actually reduced while ensuring the precision constraint. If so, *bestCost* is updated with the current cost. At line 15, we check if the rest clauses of the old formula ($f \setminus \{c\}$) cover the refined clause c'. If so, we remove the clause c' from the formula (line 16) and try to refine another clause. For instance, suppose f is $a_1 \vee a_2 \vee a_3$ and it is refined to $f' = a_1 \vee (a_1 \wedge a_2) \vee a_3$. We remove the refined clause $a_1 \wedge a_2$ because $a_1 \wedge a_2 \implies a_1$. If the condition at line 15 is false, we update the workset with the refined clause c' (i.e., c' can be refined further) and f gets replaced by f'. If the performance is not improved or the precision constraint is violated, we do not add the refined clause c' to the workset and f does not get updated.

Note that the algorithm is guaranteed to terminate. First of all, the workset W never grows in each iteration of the loop. After a clause is removed from the workset at line 7, the algorithm either

goes into the next iteration (line 17) or refines the clause and pushes it back to the workset (line 19). Furthermore, a clause never gets endlessly refined during the algorithm. Once a clause becomes a conjunction of all atoms, the ChooseAtom function returns *false* which makes the condition at line 15 true and that clause is permanently removed from the workset. Therefore, the workset eventually becomes empty in finite steps.

3.6 Atomic Features

Table 1 shows the atomic features used in our model. In any application of machine learning, the success depends heavily on the quality of the features. For instance, in the existing approach by Oh et al. [2015], authors manually crafted 45 high-level features for program variables, which are then used for learning to apply flow-sensitivity in interval analysis. However, coming up with such high-quality features manually is a nontrivial task requiring a large amount of engineering effort and domain expertise.

In this work, to reduce the feature-engineering burden, we focus on generating only low-level and easy-to-obtain atomic features and utilize a learning algorithm to synthesize high-level features. We used features found in signature and body as source for readily available information from any Java frontend such as Soot [Vallée-Rai et al. 1999].

Using Soot, we generated two types of atomic features: features for method signatures and features for statements. A signature feature describes whether the method's signature contains a particular string. For instance, the first feature in Table 1 indicates whether the method contains string "java" in its signature. From a training program, we generated all words contained in method signatures and collected the top 10 words that most frequently appear. Features #1–10 show the signature features generated this way. A statement feature indicates whether the method has a particular type of statements. We used 15 statement types available in Soot (#11–25 in Table 1). For instance, the feature #11 indicates whether the method has at least one assignment statement in its body. Combining the types of features, we generated 25 atomic features.

Regarding signature features, we chose top-10 features because they provide enough frequency spectrum, both general and specific. For instance, the feature #1 ("java") appeared 142,097 times over the training programs, whereas the feature #10 ("init") appeared 31,984 times. First five features are general method properties, and the others are specific ones. Both of general and specific features are needed to generate accurate yet generalizable context-selection heuristics. For example, application of features #1 through #5, without specific features #6 through #10, our algorithm fails to find a cost-effective heuristic. Inclusion of specific features allow our analysis to become more efficient without significant trade-off on precision on analysis result. Without features #6 through #10 included, timeout would occur on large programs. In Section 4.1, we provide more detailed discussion with experimental results.

Our learning approach works well without high-level features, mainly because the learning model (i.e., parameterized heuristic) is powerful and able to automatically generate those features by combining the atomic features via boolean formulas. On the other hand, the learning model used by Oh et al. [2015] has limited expressiveness; the model combines the features by simple linear combination, which cannot express, for instance, disjunctions of atomic features.

4 EXPERIMENTS

In this section, we experimentally evaluate our data-driven approach with application to context-sensitive points-to analysis. The main objective of the evaluation is to answer the following research questions:

Table 1. Atomic features

Signature features									
#1	"java"	#3	"sun"	#5	"void"	#7	"int"	#9	"String"
#2	"lang"	#4	"()"	#6	"security"	#8	"util"	#10	"init"

Statement features					
#11	AssignStmt	#16	BreakpointStmt	#21	LookupStmt
#12	IdentityStmt	#17	EnterMonitorStmt	#22	NopStmt
#13	InvokeStmt	#18	ExitMonitorStmt	#23	RetStmt
#14	ReturnStmt	#19	GotoStmt	#24	ReturnVoidStmt
#15	ThrowStmt	#20	IfStmt	#25	TableSwitchStmt

- **Effectiveness and Generalization**: How well does our data-driven approach performs compared to the existing approaches? Does our learning approach generalize well on unseen data?
- **Adequacy of Our Learning Algorithm**: Is the disjunctive model essential for learning cost-effective context-sensitivity? How much is it better than the simpler non-disjunctive model?
- **Learned Features**: What are the interesting findings on learned boolean formulas?

We implemented our approach on top of the Doop framework used by Smaragdakis et al. [2014]. We used the DaCapo benchmark suite [Blackburn et al. 2006] to evaluate our approach. All experiments were done on a machine with Intel i5 CPU and 16 GB RAM running on Ubuntu 14.04 64bit operating system and JDK 1.6.0_24.

4.1 Effectiveness and Generalization

Setting. We applied our data-driven approach to three existing context-sensitive points-to analyses: selective 2-object-sensitive (*S2objH*), 2-object-sensitive (*2objH*), and 2-type-sensitive (*2typeH*) analyses, all with 1-context-sensitive heap. All of these analyses are readily available in Doop. *S2objH* and *2objH* are known to be the state-of-the-art points-to analyses for Java with good precision/cost trade-offs [Kastrinis and Smaragdakis 2013b; Milanova et al. 2005]. *2typeH* is another good alternative for precise yet scalable points-to analysis [Smaragdakis et al. 2011]. Following our approach in Sections 2 and 3, we made data-driven versions of these analyses: *S2objH+Data*, *2objH+Data*, and *2typeH+Data*. In addition, we also made the introspective versions [Smaragdakis et al. 2014] of the three analyses: *S2objH+IntroA*, *S2objH+IntroB*, *2objH+IntroA*, *2objH+IntroB*, *2typeH+IntroA*, and *2typeH+IntroB*. The introspective versions are available in Doop, except for *S2objH+IntroA* and *S2objH+IntroB*. We implemented these two analyses by reusing the code of introspective analysis in Doop.

In summary, we compared the performance of the following context-sensitive analyses:

- Selective object-sensitivity:
  - *S2objH*: selective 2-object-sensitivity with 1 context-sensitive heap hybrid [Kastrinis and Smaragdakis 2013b]
  - *S2objH+Data*: our data-driven version of *S2objH*.
  - *S2objH+IntroA*: introspective version of *S2objH* with the Heuristic A [Smaragdakis et al. 2014]

  - *S2objH+IntroB*: introspective version of *S2objH* with the Heuristic B [Smaragdakis et al. 2014]
- Object-sensitivity:
  - *2objH*: 2-object-sensitivity with 1 context-sensitive heap [Kastrinis and Smaragdakis 2013b; Milanova et al. 2005]
  - *2objH+Data*: our data-driven version of *2objH*.
  - *2objH+IntroA*: introspective version of *2objH* with the Heuristic A [Smaragdakis et al. 2014]
  - *2objH+IntroB*: introspective version of *2objH* with the Heuristic B [Smaragdakis et al. 2014]
- Type-sensitivity:
  - *2typeH*: 2-type-sensitivity with 1 context-sensitive heap [Smaragdakis et al. 2011]
  - *2typeH+Data*: our data-driven version of *2typeH*.
  - *2typeH+IntroA*: introspective version of *2typeH* with the Heuristic A [Smaragdakis et al. 2014]
  - *2typeH+IntroB*: introspective version of *2typeH* with the Heuristic B [Smaragdakis et al. 2014]

As it is done by Smaragdakis et al. [2014], we partitioned the ten programs from the DaCapo suite into four small (`antlr`, `lusearch`, `luindex`, and `pmd`) and six large (`eclipse`, `xalan`, `chart`, `bloat`, `hsqldb`, and `jython`) programs. We used the four small programs as a training set where we learned context-selection heuristics. We used `hsqldb` for choosing the value of γ, i.e., the precision threshold of the optimization problem in (1). To choose γ, for each of γ between 0.85 and 0.95 with interval 0.01, we learned from the training set a context-selection heuristic, evaluated its performance on `hsqldb`, and chose γ that shows best performance according to $\frac{\text{\#proved assertions}}{\text{analysis time(s)}}$. The final heuristic with the chosen γ was evaluated on the remaining five test programs (`eclipse`, `xalan`, `chart`, `bloat`, and `jython`). The best γ were 0.93, 0.92, and 0.88 for selective object-sensitivity, object-sensitivity, and type-sensitivity, respectively. We used `hsqldb` for choosing γ since it is one of the two most challenging programs (`jython` and `hsqldb`) in the DaCapo benchmark suite. Our learning algorithm took 30 hours for learning the depth-2 formula (f_2), and 24 hours for the depth-1 formula (f_1) on the four training programs. Lastly, while introspective analysis selects heap allocations as well, we analyzed all heap allocations context-sensitively.

Results. Fig. 3 compares the performance of our approach for selective object-sensitivity. We discuss the case of selective object-sensitivity in detail, as it is arguably the best context abstraction available to Java points-to analysis [Kastrinis and Smaragdakis 2013b]. In summary, the results show that our data-driven version (*S2objH+Data*) performs remarkably well compared to the other analyses. Detailed numbers are presented in Table 2.

Crucially, our analysis strikes an unprecedented balance between precision and cost. Notice that the running time of our analysis is less than 2 minutes for all programs; indeed, it achieves virtually the same speed of the context-insensitive analysis. In particular, the analysis is able to analyze `jython`, the most demanding benchmark, in 105 sec, for which *S2objH* does not terminate in a reasonable amount of time. Yet, the precision of our analysis is comparable to that of the most precise analysis (*S2objH*); our analysis increases the number of may-fail casts only by 18% on average while *S2objH+IntroA*, another analysis who completes all benchmarks within time budget, increases the number by 85% on average.

Our data-driven points-to analysis far excels the performance of the state-of-the-art hand-tuned points-to analyses. The introspective analyses [Smaragdakis et al. 2014], which also selectively assign varying context-depths to different methods based on pre-determined heuristics, do not show satisfactory performance. *S2objH+IntroA* scales well across all programs but it does so by sacrificing

Table 2. Comprehensive performance numbers for all analyses against testing and validation (denoted by *) benchmarks: context-insensitivity (Insens), selective object sensitivity, object-sensitivity, and type-sensitivity. For each analysis, except for Insens, we made four variants: the most precise analyses (*S2objH*, *2objH*, *2typeH*), introspective analyses (IntroA and IntroB), and our data-driven analysis (Ours). In all metrics, lower is better. Entries with dash (-) means the analysis did not finish within time constraint (5400 sec.). For precision metrics, we have the number of virtual calls that points-to analysis cannot uniquely resolve the target, the number of reachable methods, and the number of may-fail casts. For cost metrics, we have the number of call-graph edges and analysis time.

		Insens	Selective object sensitivity				Object-sensitivity				Type-sensitivity			
			S2objH	IntroA	IntroB	Ours	*2objH*	IntroA	IntroB	Ours	*2typeH*	IntroA	IntroB	Ours
eclipse	poly v-calls	1,334	979	1,118	1,045	1,066	980	1,118	1,046	1,060	1,031	1,161	1,100	1,119
	reachable mthds	8,465	7,910	8,216	8,000	7,971	7,911	8,319	8,001	7,959	7,933	8,336	8,026	7,995
	may-fail casts	1,139	456	892	676	596	546	977	764	661	665	1,004	850	807
	call-graph-edges	45,474	2.9M	0.8M	1.2M	0.1M	3.4M	0.9M	1.2M	0.1M	0.6M	0.2M	0.4M	59,389
	analysis time(s)	18	79	41	53	23	91	42	52	23	42	39	44	33
chart	poly v-calls	1,852	1,378	1,612	1,512	1,441	1,378	1,613	1,497	1,435	1,446	1,658	1,541	1,516
	reachable mthds	12,064	11,328	11,791	11,589	11,400	11,330	11,952	11,518	11,362	11,439	11,976	11,579	11,474
	may-fail casts	1,810	757	1,458	1,191	922	883	1,580	1,236	974	1,147	1,656	1,376	1,245
	call-graph-edges	63,453	8M	2.4M	3.9M	0.1M	9.3M	1.6M	2.9M	0.1M	0.6M	0.3M	0.5M	86,383
	analysis time(s)	34	196	188	213	34	178	91	133	34	60	76	85	62
bloat	poly v-calls	2,014	1,426	1,684	1,521	1,504	1,427	1,690	1,522	1,496	1,626	1,812	1,684	1,680
	reachable mthds	8,939	8,469	8,728	8,625	8,526	8,470	8,869	8,626	8,513	8,523	8,885	8,647	8,564
	may-fail casts	1,924	1,125	1,747	1,555	1,232	1,193	1,809	1,621	1,288	1,485	1,832	1,713	1,564
	call-graph-edges	61,150	35M	0.5M	2M	0.2M	35.1M	0.5M	2.0M	0.3M	0.7M	0.1M	0.3M	86,291
	analysis time(s)	22	2,184	39	96	30	2,187	43	96	43	53	44	51	42
xalan	poly v-calls	1,898	1,518	1,743	1,575	1,581	1,522	1,765	1,579	1,583	1,565	1,793	1,640	1,658
	reachable mthds	9,705	9,043	9,365	9,115	9,155	9,047	9,637	9,119	9,142	9,151	9,655	9,232	9,193
	may-fail casts	1,182	447	1,055	638	538	533	1,129	723	604	728	1,136	888	812
	call-graph-edges	51,302	9M	1.4M	5.6M	0.1M	11.6M	1.6M	6.8M	0.1M	0.9M	0.3M	0.7M	66,206
	analysis time(s)	29	414	75	329	35	672	78	484	38	71	75	92	63
jython	poly v-calls	2,778	-	2,616	-	2,500	-	2,632	-	2,481	-	2,665	2,479	2,556
	reachable mthds	12,718	-	12,596	-	12,024	-	12,663	-	12,008	-	12,679	12,143	12,048
	may-fail casts	2,234	-	2,109	-	1,722	-	2,202	-	1,773	-	2,209	1,984	1,913
	call-graph-edges	0.1M	-	6.6M	-	0.3M	-	6.2M	-	1.1M	-	0.5M	9M	0.1M
	analysis time(s)	73	-	348	-	105	-	353	-	438	-	171	1,443	132
hsqldb*	poly v-calls	1,592	-	1,390	1,257	1,204	-	1,482	1,260	1,195	1,187	1,495	1,288	1,247
	reachable mthds	11,486	-	10,852	10,371	10,395	-	11,367	10,378	10,387	10,333	11,373	10,397	10,438
	may-fail casts	1,662	-	1,385	953	1,064	-	1,558	1,034	1,053	1,028	1,578	1,180	1,257
	call-graph-edges	63,790	-	1.4M	10.7M	0.3M	-	1.0M	7.6M	0.7M	1.7M	0.2M	0.5M	0.1M
	analysis time(s)	42	-	77	247	43	-	74	203	261	127	79	78	68

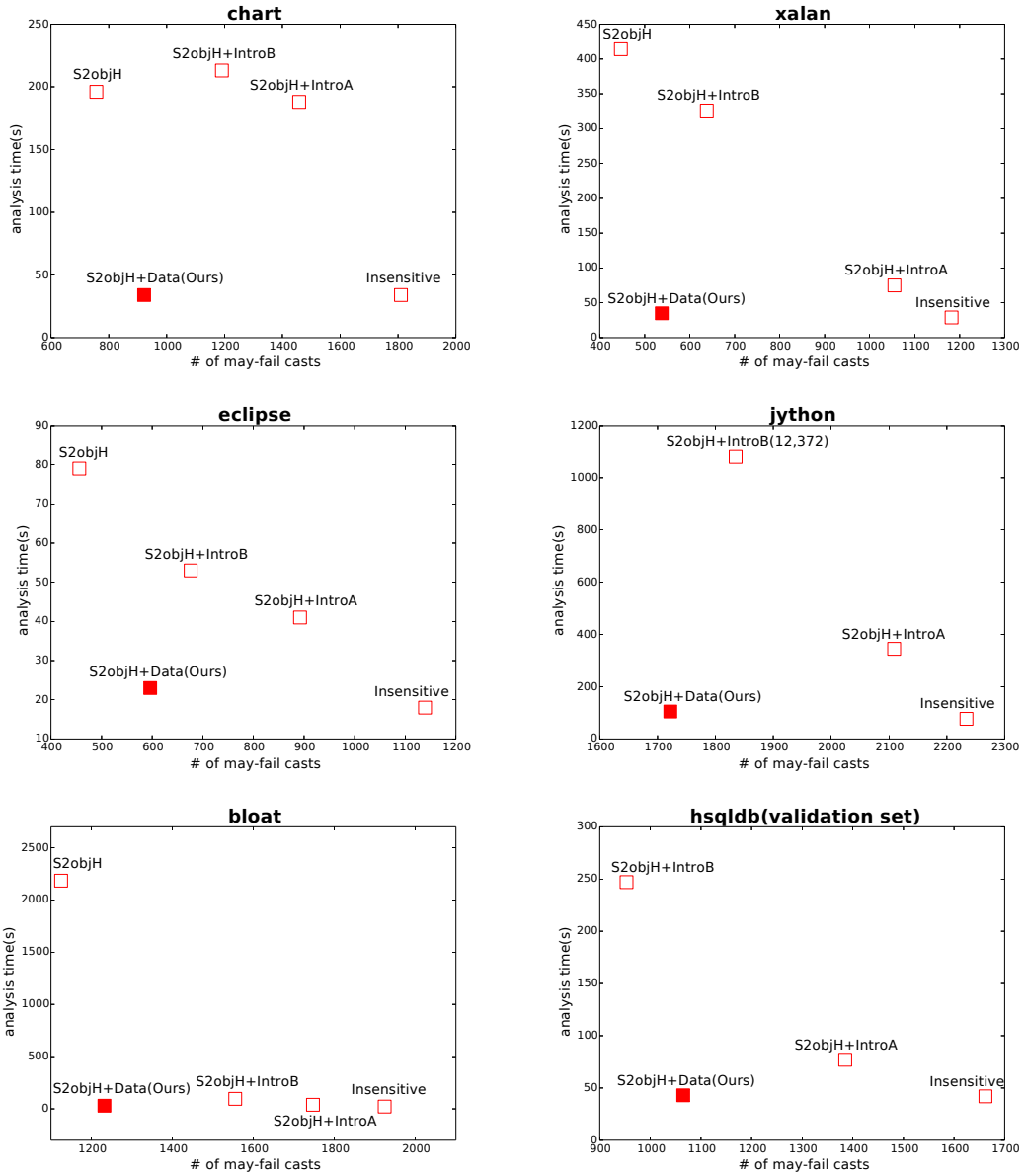

Fig. 3. Precision and cost comparisons of among selective object-sensitive class. We purposely made an exception in the case of *S2objH+IntroB* on `jython` benchmark, which is marked as timeout in Table 2, to provide readers broader performance spectrum.

the precision significantly. On the other hand, *S2objH+IntroB* improves the precision but it is at the expense of the scalability. For `chart`, *S2objH+IntroB* even requires more time than *S2objH* while sacrificing the precision. Indeed, our analysis (*S2objH+Data*) significantly outperforms *S2objH+IntroA* and *S2objH+IntroB* in *both* precision and cost on the five test programs (`eclipse`, `xalan`, `chart`, `bloat`,

Table 3. Statistics on the number of method invocations selected for context-sensitivity. Although our approach selects method definitions, not method invocations, we present the numbers for the final, selected invocations, in order to compare with the introspective analyses.

Benchmarks	Total Invos.	*S2objH+IntroA*		*S2objH+IntroB*		*S2objH+Data*(Ours)			
		Depth-2	%	Depth-2	%	Depth-1	%	Depth-2	%
eclipse	105,045	100,046	95.2	105,045	100.0	11,002	10.5	13,851	13.2
chart	232,794	226,101	97.1	231,129	99.3	32,831	14.1	26,319	11.3
bloat	112,450	100,730	89.6	112,146	99.7	11,030	9.8	16,092	14.3
xalan	211,997	205,430	96.9	211,993	100.0	27,937	13.2	22,695	10.7
jython	232,420	215,078	92.5	230,907	99.3	28,572	12.3	23,975	10.3
Avg.	178,941	169,477	94.3	178,244	99.7	22,274	12.0	20,586	12.0

and `jython`). Our approach shows similar performance improvements for object-sensitivity and type-sensitivity as well (Table 2).

Table 3 shows that our approach is very accurate in identifying methods that would benefit from context-sensitivity. Table 3 compares the number of method invocations selected by our approach and introspective analyses. Our approach chooses 12% of total method invocations on average for both context depths. On the other hand, introspective analyses A and B choose 94.3% and 99.7% of invocations, respectively.[2] Note that our analysis is more precise than introspective analyses, even though we choose much smaller sets of method invocations for context-sensitivity.

Generalization. The learned heuristics were generalized well to unseen programs, even from small programs to large programs. Table 4 and 5 show the performance of the learned heuristic for selective object-sensitivity on the training and test programs. The tables compare three analyses, context-insensitive, *S2objH*, and *S2objH+Data*, using two prime metrics, a number of may-fail casts and analysis time. We define two quality metrics, $quality_{precision}$ and $quality_{cost}$, to illustrate how our approach achieves desirable performance. For both definitions, higher values are better:

$$quality_{precision} = \frac{|unproven_{\text{INSENS}}| - |unproven_{S2objH+Data}|}{|unproven_{\text{INSENS}}| - |unproven_{S2objH}|} \times 100$$

$$quality_{cost} = \frac{cost_{S2objH} - cost_{S2objH+Data}}{cost_{S2objH} - cost_{\text{INSENS}}} \times 100.$$

The results show that our approach achieves similar precision gains on both cases. Our approach shows much better scalability gains on the (large) test programs.

Sensitivity to Atomic Features. As described in Section 3.6, we performed experiments without specific signature features #6 through #10. In total, we used 20 features (5 signature features and 15 statement features). The results are presented in Table 6.

Without specific features, our algorithm failed to find a cost-effective heuristic. Exclusion of specific features increased the analysis precision slightly, because the resulting heuristic selects more methods for context-sensitivity. For instance, 50.6% of methods were chosen for 2-object-sensitivity by the heuristic learned without specific features, while 10.6% of methods were chosen with those features. However, the analysis cost increased substantially and timeout occurred for

[2] Table 3 shows statistics only for selected method invocations. Introspective analyses also choose the set of heap allocations that will receive context-sensitivity.

Table 4. Learning performance on training and validation sets

Benchmarks	*Context-insensitive*		*S2objH*		*S2objH+Data*(Ours)			
	may-fail casts	time(s)	may-fail casts	time(s)	may-fail casts	quality	time(s)	quality
antlr	992	35	360	94	505	77%	48	78%
luindex	734	27	229	48	286	89%	31	81%
lusearch	844	21	231	73	294	90%	24	94%
pmd	1,263	44	585	73	655	90%	50	79%
hsqldb	1,662	42	timeout	timeout	1,064	N/A	43	N/A
TOTAL	5,495	169	1,405+	288+	2,804	86%	196	83%

Table 5. Learning performance on testing set

Benchmarks	*Context-insensitive*		*S2objH*		*S2objH+Data*(Ours)			
	may-fail casts	time(s)	may-fail casts	time(s)	may-fail casts	quality	time(s)	quality
chart	1,810	34	757	196	922	84%	34	100%
bloat	1,924	22	1,125	2,184	1,232	87%	30	100%
eclipse	1,139	18	456	79	596	80%	23	92%
xalan	1,182	29	447	414	538	88%	35	98%
jython	2,234	73	timeout	timeout	1,722	N/A	105	N/A
TOTAL	8,289	176	2,785+	2,873+	5,010	85%	227	97%

Table 6. Performance of our approach (*S2objH* + *Data*) without signature features #6 through #10.

	eclipse	chart	bloat	xalan	jython
poly v-calls	1,043	1,408	1,487	1,554	-
reachable mthds	7,948	11,365	8,502	9,124	-
may fail casts	543	856	1,195	491	-
call-graph-edges	38,555	52,582	53,983	45,412	-
analysis time(s)	59	105	66	273	-

`jython`. The results show that inclusion of specific features makes the analysis much more efficient without significant trade-off on precision.

4.2 Adequacy of Our Learning Approach

In this subsection, we motivate our choice of the disjunctive model by comparing the performance of the non-disjunctive model used in prior work [Oh et al. 2015]. The comparison is done for selective object-sensitivity (*S2objH*).

The idea of the previous method [Oh et al. 2015] is to compute the score of each program element by a linear combination of the feature vector and a real-valued parameter vector, and to choose a certain number of top scorers. Learning the vector of real numbers is formulated as an optimization problem and is solved using Bayesian optimization. To use this learning algorithm in our setting, we applied the algorithm [Oh et al. 2015] twice, one for selecting the set of methods that require the depth-2 context-sensitivity and the other for the depth-1 context-sensitivity. All the other methods are analyzed context-insensitively. We used 24-hour time budget for Bayesian optimization, giving the same amount of time required by our learning algorithm. We chose the same number of methods

Table 7. Performance comparison between disjunctive and non-disjunctive models.

Benchmarks	Non-disjunctive		Disjunctive(Ours)	
	may-fail casts	time(s)	may-fail casts	time(s)
eclipse	946	25	596	21
chart	1,569	48	937	33
bloat	1,771	46	1,232	27
xalan	996	42	539	33
jython	2069	346	1,738	104
Total	7,352	346	5,042	218

as our approach; we gave depth-2 to 10.6% of the methods and depth-1 to 10.9%. Also, we used the same set of atomic features and benchmark programs.

Table 7 compares the performance. The performance of the analysis learned by the linear learning algorithm is inferior to ours in both precision and cost. The non-disjunctive approach produces 1.5x more may-fail casts and takes 1.5x more time than ours.

The main reason is the non-disjunctive model fails to capture complex context-selection heuristics due to its limited expressiveness. A delicate selection of the methods to apply context-sensitivity is a key to both precision and cost in points-to analysis for Java. For example, consider the following boolean formula that our learning algorithm has inferred to describe the methods that require selective 1-object-sensitivity:

$$(\underline{1} \wedge \underline{2} \wedge \neg 3 \wedge \neg 6 \wedge \neg 7 \wedge \underline{\neg 8} \wedge \cdots \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25) \vee$$
$$(\underline{\neg 1} \wedge \underline{\neg 2} \wedge \underline{8} \wedge 5 \wedge \neg 9 \wedge 11 \wedge 12 \wedge \cdots \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25) \vee$$
$$(\neg 3 \wedge \neg 4 \wedge \neg 7 \wedge \underline{\neg 8} \wedge \neg 9 \wedge 10 \wedge 11 \wedge \cdots \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25)$$

The linear model cannot express such a feature. For example, the above formula shows that the underlined atomic features 1, 2, and 8 are used as in both positive and negative forms in different clauses. Non-disjunctive model cannot capture such mixed signals in different contexts due to its inherent limitations.

4.3 Learned Features

The features learned for each analysis are presented in Appendix B. We discuss some interesting findings from the learned features.

First, we observed that our approach produces similar features for similar context-abstractions. For instance, the learned boolean formulas for depth-2 are the same for all object-based context-sensitivities:

f_2 for *S2objH+Data*	:	$1 \wedge \neg 3 \wedge \neg 6 \wedge 8 \wedge \neg 9 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \cdots \wedge \neg 25$
f_2 for *2objH+Data*	:	$1 \wedge \neg 3 \wedge \neg 6 \wedge 8 \wedge \neg 9 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \cdots \wedge \neg 25$
f_2 for *2typeH+Data*	:	$1 \wedge \neg 3 \wedge \neg 6 \wedge 8 \wedge \neg 9 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \cdots \wedge \neg 25$
f_2 for call-site-sensitivity	:	$1 \wedge \neg 6 \wedge \neg 7 \wedge 11 \wedge 12 \wedge 13 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \cdots \wedge \neg 25$

Note that the object-based context-sensitive analyses (selective object-sensitivity, object-sensitivity, and type-sensitivity) share the same formula (f_2) for the depth-2 context-sensitivity. We conjecture that these analyses construct the calling-contexts using a heap context when their context-depth goes beyond two. Since the three abstractions use similar definitions of the heap contexts, precision gains from the heap context information are also similar. On the other hand, we obtained a

completely different formula for call-site-sensitivity, which uses different heap abstraction from other object-based sensitivities.[3]

Another unexpected observation was that the learned formulas have orders according to the theoretical orders of the analysis precision. For example, our learning algorithm produced depth-1 formulas (f_1) for object-sensitivity and type-sensitivity as follows:

f_1 for *2objH+Data*	:	$\mathbf{(1 \wedge 2 \wedge \neg 3 \wedge \neg 6 \wedge \neg 7 \wedge \neg 8 \wedge \neg 9 \wedge \neg 16 \wedge \cdots \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25)} \vee$ $(\neg 1 \wedge \neg 2 \wedge 8 \wedge 5 \wedge \neg 9 \wedge 11 \wedge 12 \wedge \cdots \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25) \vee$ $(\neg 3 \wedge \neg 4 \wedge \neg 7 \wedge \neg 8 \wedge \neg 9 \wedge 10 \wedge 11 \wedge \cdots \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25)$
f_1 for *2typeH+Data*	:	$\mathbf{1 \wedge 2 \wedge \neg 3 \wedge \neg 6 \wedge \neg 7 \wedge \neg 8 \wedge \neg 9 \wedge \neg 15 \wedge \neg 16 \wedge \cdots \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25}$

Note that the formula f_1 for object-sensitivity is logically more general than that for type-sensitivity, as boldfaced clause in f_1 for *2typeH+Data* is subsumed by the boldfaced clause in f_1 for *2objH+Data*. Therefore, f_1 for *2objH+Data* describes a superset of the methods described by f_1 for *2typeH+Data*. Theoretically, since object-sensitivity is more precise than type-sensitivity, the set of methods that benefit from object-sensitivity must be a superset of the methods that benefit from type-sensitivity. Interestingly, our learning algorithm automatically discovered this fact from data.

Lastly, we spotted that some atomic features are frequently used as negative forms. Breakpoint(16), EnterMonitor(17), ExitMonitor(18), Lookup(21), Nop(22), and Ret(23) statements rarely appear in the programs. Therefore, conjoining a formula with the negation of these features would make little difference. Methods that return the void type deserve shallower context depths because they are less likely to jeopardize points-to analysis than ones who return objects. We also found that some control-flow features also frequently appear in negated forms.

4.4 Threats to Validity

- Benchmarks: Our experimental evaluation were conducted on the DaCapo benchmark, but the DaCapo benchmark may not be a reputable material for machine learning purposes although it does for points-to analysis community.
- Generality: The DaCapo benchmark may not represent general Java programs as it is a collection of specific types of programs, comprising mostly compilers and interpreters. In experiments, we also assumed that a heuristic learned from smaller programs is likely to work well for larger programs, which may not be true in other circumstances.
- Features: We evaluated our approach with a fixed set of atomic features: signature and statement features. Different set of atomic features are likely to produce different results.

5 RELATED WORK

Context-sensitive points-to analysis has a vast amount of past literature, e.g., [Agesen 1994; Chatterjee et al. 1999; Grove et al. 1997; Hind 2001; Lhoták and Hendren 2006, 2008; Liang and Harrold 1999; Liang et al. 2005; Milanova et al. 2005; Ruf 1995, 2000; Wilson and Lam 1995]. In this section, we discuss prior works that are closely related to ours.

Tuning Context-Sensitivity in Points-to Analysis. Most of the existing techniques for tuning context-sensitivity in points-to analysis are traditional rule-based techniques [Kastrinis and Smaragdakis 2013a; Oh et al. 2014; Smaragdakis et al. 2014; Tripp et al. 2009]. They selectively apply context-sensitivity based on some manually-designed syntactic or semantic features of the program. For instance, in the approach by Smaragdakis et al. [2014], a cheap pre-analysis is used to identify when and where context-sensitivity would fail, and then the main analysis applies

[3]Although we do not discuss the performance of our approach for call-site-sensitivity since call-site-sensitivity is less important than others in points-to analysis for Java, we also evaluated the analysis and obtained similar performance improvements as in others.

context-sensitivity selectively based on the pre-analysis results and heuristic rules. Although this work provides useful insights about context-sensitivity and provides good heuristics, the resulting analyses are still not completely satisfactory. We believe the main reason is that those rules are manually-designed by analysis designers, which is likely to be suboptimal and unstable. The goal of this paper is to overcome the existing limitations by automating the process of generating such heuristic rules.

The recent technique by Tan et al. [2016] is orthogonal to our approach. Recently, Tan et al. [2016] proposed a technique to improve the precision of k-context-sensitive points-to analysis. The idea is to use k context slots with more informative elements even if they are located beyond the most recent k contexts. The authors identify such good elements by running a cheap pre-analysis using dependency graph among object allocations. As a result, for a given context-depth k, the resulting analysis is at least as precise as the conventional k-context-sensitive analysis. Our approach differs from this work as we focus on balancing precision and cost of an existing points-to analysis, so both approaches can be combined.

Data-Driven Program Analysis. Our data-driven points-to analysis improves the state-of-the-art data-driven program analysis in several aspects. Recently, a number of techniques for data-driven program analysis were proposed [Cha et al. 2016; Heo et al. 2016, 2017; Oh et al. 2015]. In this approach, program analysis is designed with parameterized heuristic rules, and their parameter values are found automatically from data through learning algorithms. Compared to prior works on data-driven program analysis, our work provides two novel contributions. First, we propose a new machine-learning model that is able to describe disjunctive properties of programs with boolean formulas. On the other hand, existing works [Cha et al. 2016; Oh et al. 2015] rely on simple linear models that cannot express disjunctive properties, or use off-the-shelf nonlinear models (e.g., decision trees) that require labeled data [Heo et al. 2016, 2017]. Second, we present a new algorithm that efficiently learns good parameters of our boolean-formula model. The use of more powerful model and learning algorithm enables us not only to solve the problem of describing complex context-selection heuristic rules precisely (Section 4.2) but also to make our approach less susceptible to the qualities of atomic features (Section 3.6).

The data-driven approach by Chae et al. [2017] deals with different problems in different contexts. While this paper aims to develop new learning model and algorithm suitable for context-sensitive points-to analysis for Java, Chae et al. [2017] aim to automatically generate features that are used for learning flow-sensitivity and variable clustering in relational analysis for C. The techniques of both papers could be combined but doing so requires to solve nontrivial problems. For example, it would be possible to replace the manually-designed features used in this paper with automatically generated features by using the technique by Chae et al. [2017]. This combination, however, is nontrivial to achieve because both approaches target substantially different languages (C vs. Java), use different settings for parametric analysis (program-part-based vs. query-based), and Chae et al. [2017] focus on program analyses whose parameters can be effectively found within a single procedure (e.g. flow-sensitivity and variable clustering) and its application to interprocedural setting (e.g. context-sensitivity) remains to be seen.

Parametric Program Analysis. The techniques in this paper differ from prior parametric program analyses [Liang et al. 2011; Oh et al. 2014; Zhang et al. 2014]. For instance, Zhang et al. [2014] proposed a CEGAR-based technique for context-sensitive points-to analysis for Java. They use CEGAR to find abstractions that only contain relevant program elements for proving all points-to queries in target programs. Although this approach guarantees that all queries provable by applying context-sensitivity are eventually resolved, the technique requires to iteratively analyze the program multiple times, which might be impractical for large programs (e.g., `jython`) in practice.

Liang et al. [2011] suggested an approach that finds minimal context-sensitivity of points-to analysis. However, they do not provide how to find the minimal abstractions before running the analysis. Oh et al. [2014] proposed a method that runs a pre-analysis to estimate the impact of context-sensitivity on the main analysis. The idea has been presented mainly for numeric analysis (e.g., using the interval and octagon domains), and the method requires the analysis designers to come up with a right abstraction for pre-analysis. For instance, a sign analysis that distinguishes non-negative integers is shown to be effective for interval analysis [Oh et al. 2014]. However, it is not trivial to design an appropriate pre-analysis for points-to analysis.

Demand-driven Points-to Analysis. Demand-driven points-to analyses [Guyer and Lin 2003; Heintze and Tardieu 2001; Sridharan and Bodík 2006; Sridharan et al. 2005] solve a scalability issue of points-to analysis by concentrating on a fixed set of queries. For a given program and a query in it, this technique selectively applies costly but precise analysis only to those who contribute to proving the query. Our technique differs from this approach as we do not target a specific query but try to capture general features of methods that contribute to maximizing the number of provable queries in programs.

6 CONCLUSION

In this paper, we showed that data-driven approach is a promising way of developing cost-effective context-sensitive points-to analysis. Our approach uses a heuristic rule parameterized by boolean formulas that are able to express complex, in particular disjunctive, properties of methods. The parameters (i.e., boolean formulas) of the heuristic are learned from codebases through a carefully designed learning algorithm. We have implemented this approach in Doop, and applied it to three representative context-sensitive analyses for Java: object-sensitivity, selective hybrid object-sensitivity, and type-sensitivity. Experimental results confirm that points-to analysis with the learned heuristic significantly outperforms the existing state-of-the-art analyses with manually designed heuristic rules. We believe this work provides a starting point for a shift from traditional rule-based points-to analysis to data-driven approaches.

A PROOF OF THEOREM 3.4

Let $\Pi = \langle f_1, f_2, \ldots, f_k \rangle$ be the output of our learning algorithm. Obviously, Π meets the precision constraint

$$\frac{\sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_\Pi(P)))|}{\sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma$$

because f_1 becomes a solution of Ψ_1 only if the condition above is true.

Next, we show that there exists no solution smaller than Π. Suppose $\Pi' = \langle f'_1, f'_2, \ldots, f'_k \rangle$ is a parameter that meets the precision constraint and Π' is smaller than Π:

$$\forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'}(P) \sqsubseteq \mathcal{H}_\Pi(P). \tag{3}$$

Our goal is to show that the following claim holds:

$$\forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'}(P) = \mathcal{H}_\Pi(P). \tag{4}$$

We show the claim by proving the more general statement:

$$\forall i \in [1, k].\ \forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'_i}(P) = \mathcal{H}_{\Pi_i}(P) \tag{5}$$

where

$$\Pi_i = \langle \mathit{true}, \ldots, \mathit{true}, f_i, f_{i+1}, \ldots, f_k \rangle$$
$$\Pi'_i = \langle \mathit{true}, \ldots, \mathit{true}, f'_i, f'_{i+1}, \ldots, f'_k \rangle$$

The claim (4) is a special case of (5) when $i = 1$. We prove (5) by induction on i in decreasing order. The proof uses the following fact

$$\forall i \in [1, k].\ \forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'_i}(P) \sqsubseteq \mathcal{H}_{\Pi_i}(P) \tag{6}$$

which is derived from (3) and the definition of $\mathcal{H}$.

- (Base case) When $i = k$, we need to prove that

  $$\forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'_k}(P) = \mathcal{H}_{\Pi_k}(P).$$

  From $\mathcal{H}_{\Pi'}(P) \sqsubseteq \mathcal{H}_{\Pi'_k}(P)$ for all P and the monotonicity of the analysis (Definition 3.1), we have

  $$\forall P \in \mathbf{P}.\mathsf{proved}(F_P(\mathcal{H}_{\Pi'}(P))) \subseteq \mathsf{proved}(F_P(\mathcal{H}_{\Pi'_k}(P))). \tag{7}$$

  From the assumption $\frac{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_{\Pi'}(P)))|}{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma$ and (7), we have

  $$\frac{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_{\Pi'_k}(P)))|}{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma. \tag{8}$$

  From (6), (8), Definition 3.3, and the assumption that f_k is a minimal solution of the problem Ψ_k, we have

  $$\forall P \in \mathbf{P}.\mathcal{H}_{\Pi'_k}(P) = \mathcal{H}_{\Pi_k}(P).$$

- (Inductive case) When $i = j$. The induction hypothesis is as follows:

  $$\forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'_j}(P) = \mathcal{H}_{\Pi_j}(P).$$

  Using the hypothesis, we would like to prove that

  $$\forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'_{j-1}}(P) = \mathcal{H}_{\Pi_{j-1}}(P).$$

  Let $\Pi''_{j-1} = \langle \mathit{true}, \ldots, \mathit{true}, f'_{j-1}, f_j, \ldots, f_k \rangle$. Since we assume $\forall P \in \mathbf{P}.\mathcal{H}_{\Pi_j}(P) = \mathcal{H}_{\Pi'_j}(P)$ (I.H.), we have

  $$\forall P \in \mathbf{P}.\mathcal{H}_{\Pi''_{j-1}}(P) = \mathcal{H}_{\Pi'_{j-1}}(P). \tag{9}$$

  From $\mathcal{H}_{\Pi'}(P) \sqsubseteq \mathcal{H}_{\Pi'_{j-1}}(P)$ for all P and the monotonicity of the analysis(Definition 3.1), we have

  $$\forall P \in \mathbf{P}.\mathsf{proved}(F_P(\mathcal{H}_{\Pi'}(P))) \subseteq \mathsf{proved}(F_P(\mathcal{H}_{\Pi'_{j-1}}(P))). \tag{10}$$

  From (10) and the assumption $\frac{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_{\Pi'}(P)))|}{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma$, we have

  $$\frac{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_{\Pi'_{j-1}}(P)))|}{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma. \tag{11}$$

  From (9) and (11), we have

  $$\frac{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_{\Pi''_{j-1}}(P)))|}{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma. \tag{12}$$

  From (6) and (9), we have

  $$\forall P \in \mathbf{P}.\mathcal{H}_{\Pi''_{j-1}}(P) \sqsubseteq \mathcal{H}_{\Pi_{j-1}}(P). \tag{13}$$

  From (12), (13), Definition 3.3, and the assumption that f_{j-1} is a minimal solution of the problem Ψ_k, we have

  $$\forall P \in \mathbf{P}.\mathcal{H}_{\Pi''_{j-1}}(P) = \mathcal{H}_{\Pi_{j-1}}(P). \tag{14}$$

  From (14), (9), we conclude

  $$\forall P \in \mathbf{P}.\mathcal{H}_{\Pi_{j-1}}(P) = \mathcal{H}_{\Pi'_{j-1}}(P).$$

B LEARNED BOOLEAN FORMULAS

We list the boolean formulas learned by our approach. The numbers in the formulas represent the atomic feature in Tables 1. The formulas for each analysis and context depth are as follows. Table 8 presents them by and-or tables.

- Selective object-sensitivity:
  - Depth-2 formula (f_2):

    $1 \wedge \neg 3 \wedge \neg 6 \wedge 8 \wedge \neg 9 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25$

  - Depth-1 formula (f_1):

$(1 \wedge \neg 3 \wedge \neg 4 \wedge \neg 7 \wedge \neg 8 \wedge 6 \wedge \neg 9 \wedge \neg 15 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25) \vee$
$(\neg 3 \wedge \neg 4 \wedge \neg 7 \wedge \neg 8 \wedge \neg 9 \wedge 10 \wedge 11 \wedge 12 \wedge 13 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge$
$\neg 25) \vee (\neg 3 \wedge \neg 9 \wedge 13 \wedge 14 \wedge 15 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25) \vee$
$(1 \wedge 2 \wedge \neg 3 \wedge 4 \wedge \neg 5 \wedge \neg 6 \wedge \neg 7 \wedge \neg 8 \wedge \neg 9 \wedge \neg 10 \wedge \neg 13 \wedge \neg 15 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22$
$\wedge \neg 23 \wedge \neg 24 \wedge \neg 25)$

- Object-sensitivity:
  - Depth-2 formula (f_2):

    $1 \wedge \neg 3 \wedge \neg 6 \wedge 8 \wedge \neg 9 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25$

  - Depth-1 formula (f_1):

$(1 \wedge 2 \wedge \neg 3 \wedge \neg 6 \wedge \neg 7 \wedge \neg 8 \wedge \neg 9 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25) \vee$
$(\neg 1 \wedge \neg 2 \wedge 5 \wedge 8 \wedge \neg 9 \wedge 11 \wedge 12 \wedge \neg 14 \wedge \neg 15 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge$
$\neg 25) \vee (\neg 3 \wedge \neg 4 \wedge \neg 7 \wedge \neg 8 \wedge \neg 9 \wedge 10 \wedge 11 \wedge 12 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge$
$\neg 24 \wedge \neg 25)$

- Type-sensitivity:
  - Depth-2 formula (f_2):

    $1 \wedge \neg 3 \wedge \neg 6 \wedge 8 \wedge \neg 9 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25$

  - Depth-1 formula (f_1):

$1 \wedge 2 \wedge \neg 3 \wedge \neg 6 \wedge \neg 7 \wedge \neg 8 \wedge \neg 9 \wedge \neg 15 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25$

- Call-site-sensitivity:
  - Depth-2 formula (f_2):

    $1 \wedge \neg 6 \wedge \neg 7 \wedge 11 \wedge 12 \wedge 13 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25$

  - Depth-1 formula (f_1):

    $(1 \wedge 2 \wedge \neg 7 \wedge \neg 16 \wedge \neg 17 \wedge \neg 18 \wedge \neg 19 \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25)$

Table 8. AND-OR table of learned boolean formulas

			OR												
	1	"java"	T			T	T	T	F		T	T	T	T	T
	2	"lang"				T		T	F			T		T	
	3	"sun"	F	F	F	F	F	F		F	F	F	F		
	4	"()"	F	F		T				F					
	5	"void"				F			T						
	6	"security"	T			F	F	F			F	F	F		F
	7	"int"	F	F		F		F		F		F		F	F
	8	"util"	F	F		F	T	F	T	F	T	F	T		
	9	"String"	F	F	F	F	F	F	F	F	F	F	F		
	10	"init"		T		F				T					
	11	AssignStmt		T					T	T					T
A	12	IdentityStmt		T					T	T					T
N	13	InvokeStmt		T	T	F									T
D	14	ReturnStmt			T				F						
	15	ThrowStmt	F		T	F			F			F			
	16	BreakpointStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
	17	EnterMonitorStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
	18	ExitMonitorStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
	19	GotoStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
	20	IfStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
	21	LookupStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
	22	NopStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
	23	RetStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
	24	ReturnVoidStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
	25	TableSwitchStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
			f_1				f_2	f_1			f_2	f_1	f_2	f_1	f_2
			S2objH+Data					*2objH+Data*				*2typeH +Data*		*2callH +Data*	

ACKNOWLEDGMENTS

This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-09. This research was also supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2016R1C1B2014062).

REFERENCES

Ole Agesen. 1994. *Constraint-based type inference and parametric polymorphism.* Springer Berlin Heidelberg, Berlin, Heidelberg, 78–100. https://doi.org/10.1007/3-540-58485-4_34

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. 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 (OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

Sooyoung Cha, Sehun Jeong, and Hakjoo Oh. 2016. *Learning a Strategy for Choosing Widening Thresholds from a Large Codebase.* Springer International Publishing, Cham, 25–41. https://doi.org/10.1007/978-3-319-47958-3_2

Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. 2017. Automatically Generating Features for Learning Program Analysis Heuristics. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017).

Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. 1999. Relevant Context Inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/292540.292554

David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. 1997. Call Graph Construction in Object-oriented Languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '97)*. ACM, New York, NY, USA, 108–124. https://doi.org/10.1145/263698.264352

Samuel Z. Guyer and Calvin Lin. 2003. Client-driven Pointer Analysis. In *Proceedings of the 10th International Conference on Static Analysis (SAS'03)*. Springer-Verlag, Berlin, Heidelberg, 214–236. http://dl.acm.org/citation.cfm?id=1760267.1760284

Nevin Heintze and Olivier Tardieu. 2001. Demand-driven Pointer Analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. ACM, New York, NY, USA, 24–34. https://doi.org/10.1145/378795.378802

Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2016. *Learning a Variable-Clustering Strategy for Octagon from Labeled Data Generated by a Static Analysis.* Springer Berlin Heidelberg, Berlin, Heidelberg, 237–256. https://doi.org/10.1007/978-3-662-53413-7_12

Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Machine-Learning-Guided Selectively Unsound Static Analysis. In *Proceedings of the 39th International Conference on Software Engineering*. ACM.

Michael Hind. 2001. Pointer Analysis: Haven't We Solved This Problem Yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*. ACM, New York, NY, USA, 54–61. https://doi.org/10.1145/379605.379665

George Kastrinis and Yannis Smaragdakis. 2013a. Efficient and Effective Handling of Exceptions in Java Points-to Analysis. In *Proceedings of the 22Nd International Conference on Compiler Construction (CC'13)*. Springer-Verlag, Berlin, Heidelberg, 41–60. https://doi.org/10.1007/978-3-642-37051-9_3

George Kastrinis and Yannis Smaragdakis. 2013b. Hybrid Context-sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 423–434. https://doi.org/10.1145/2491956.2462191

George Kastrinis and Yannis Smaragdakis. 2013c. Hybrid Context-sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 423–434. https://doi.org/10.1145/2491956.2462191

Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *Proceedings of the 15th International Conference on Compiler Construction (CC'06)*. Springer-Verlag, Berlin, Heidelberg, 47–64. https://doi.org/10.1007/11688839_5

Ondřej Lhoták and Laurie Hendren. 2008. Evaluating the Benefits of Context-sensitive Points-to Analysis Using a BDD-based Implementation. *ACM Trans. Softw. Eng. Methodol.* 18, 1, Article 3 (Oct. 2008), 53 pages. https://doi.org/10.1145/1391984.

1391987

Donglin Liang and Mary Jean Harrold. 1999. Efficient Points-to Analysis for Whole-program Analysis. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7)*. Springer-Verlag, London, UK, UK, 199–215. http://dl.acm.org/citation.cfm?id=318773.318943

Donglin Liang, Maikel Pennings, and Mary Jean Harrold. 2005. Evaluating the Impact of Context-sensitivity on Andersen's Algorithm for Java Programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '05)*. ACM, New York, NY, USA, 6–12. https://doi.org/10.1145/1108792.1108797

Percy Liang, Omer Tripp, and Mayur Naik. 2011. Learning Minimal Abstractions. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 31–42. https://doi.org/10.1145/1926385.1926391

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (Jan. 2005), 1–41. https://doi.org/10.1145/1044834.1044835

Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective Context-sensitivity Guided by Impact Pre-analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 475–484. https://doi.org/10.1145/2594291.2594318

Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. 2015. Learning a Strategy for Adapting a Program Analysis via Bayesian Optimisation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 572–588. https://doi.org/10.1145/2814270.2814309

Erik Ruf. 1995. Context-insensitive Alias Analysis Reconsidered. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*. ACM, New York, NY, USA, 13–22. https://doi.org/10.1145/207110.207112

Erik Ruf. 2000. Effective Synchronization Removal for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 208–218. https://doi.org/10.1145/349299.349327

Micha Sharir and Amir Pnueli. 1981. *Two approaches to interprocedural data flow analysis.* Prentice-Hall, Englewood Cliffs, NJ, Chapter 7, 189–234.

Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (April 2015), 1–69. https://doi.org/10.1561/2500000014

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 17–30. https://doi.org/10.1145/1926385.1926390

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective Analysis: Context-sensitivity, Across the Board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 485–495. https://doi.org/10.1145/2594291.2594320

Manu Sridharan and Rastislav Bodík. 2006. Refinement-based Context-sensitive Points-to Analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/1133981.1134027

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven Points-to Analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 59–76. https://doi.org/10.1145/1094811.1094817

Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*. 489–510. https://doi.org/10.1007/978-3-662-53413-7_24

Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 87–97. https://doi.org/10.1145/1542476.1542486

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13–. http://dl.acm.org/citation.cfm?id=781995.782008

Robert P. Wilson and Monica S. Lam. 1995. Efficient Context-sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/207110.207111

Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On Abstraction Refinement for Program Analyses in Datalog. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 239–248. https://doi.org/10.1145/2594291.2594327