Learning Graph-Based Heuristics for Pointer Analysis without Handcrafting Application-Specific Features

MINSEOK JEON, MYUNGHO LEE, and HAKJOO OH*, Korea University, Republic of Korea

We present Graphick, a new technique for automatically learning graph-based heuristics for pointer analysis. Striking a balance between precision and scalability of pointer analysis requires designing good analysis heuristics. For example, because applying context sensitivity to all methods in a real-world program is impractical, pointer analysis typically uses a heuristic to employ context sensitivity only when it is necessary. Past research has shown that exploiting the program's graph structure is a promising way of developing cost-effective analysis heuristics, promoting the recent trend of "graph-based heuristics" that work on the graph representations of programs obtained from a pre-analysis. Although promising, manually developing such heuristics remains challenging, requiring a great deal of expertise and laborious effort. In this paper, we aim to reduce this burden by learning graph-based heuristics automatically, in particular without hand-crafted application-specific features. To do so, we present a feature language to describe graph structures and an algorithm for learning analysis heuristics within the language. We implemented Graphick on top of Doop and used it to learn graph-based heuristics for object sensitivity and heap abstraction. The evaluation results show that our approach is general and can generate high-quality heuristics. For both instances, the learned heuristics are as competitive as the existing state-of-the-art heuristics designed manually by analysis experts.

CCS Concepts: • **Software and its engineering → Automated static analysis**;

Additional Key Words and Phrases: Data-driven static analysis, Machine learning for program analysis, Pointer analysis, Context sensitivity, Heap abstraction

ACM Reference Format:
Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning Graph-Based Heuristics for Pointer Analysis without Handcrafting Application-Specific Features. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 179 (November 2020), 30 pages. https://doi.org/10.1145/3428247

1 INTRODUCTION

Pointer analysis is a fundamental program analysis technique that serves as a key component of various software engineering tools. The goal of pointer analysis is to statically and conservatively estimate heap objects that pointer variables may refer to at runtime. The pointer information is essential for virtually all kinds of program analysis tools, including bug detectors [Blackshear et al. 2015; Livshits and Lam 2003; Naik et al. 2006, 2009; Sui et al. 2014], security analyzers [Arzt et al. 2014; Avots et al. 2005; Grech and Smaragdakis 2017; Tripp et al. 2009; Yan et al. 2017], program verifiers [Fink et al. 2008], symbolic executors [Kapus and Cadar 2019], and program repair tools [Gao et al. 2015; Hong et al. 2020; Lee et al. 2018; Xu et al. 2019]. The success of

*Corresponding author

Authors' address: Minseok Jeon, minseok_jeon@korea.ac.kr; Myungho Lee, myungho_lee@korea.ac.kr; Hakjoo Oh, hakjoo_oh@korea.ac.kr, Department of Computer Science and Engineering, Korea University, 145, Anam-ro, Sungbuk-gu, Seoul, 02841, Republic of Korea.

2475-1421/2020/11-ART179
https://doi.org/10.1145/3428247

these tools depends eventually on the precision and scalability of the underlying pointer analysis algorithm.

Developing a fast and precise pointer analysis requires coming up with good analysis heuristics. For example, context sensitivity is critical for accurately analyzing object-oriented programs as it distinguishes method's local variables and objects in different calling contexts [Smaragdakis and Balatsouras 2015]. In reality, however, it is too expensive to apply deep context sensitivity (e.g. 2-object-sensitivity) to all methods in a nontrivial program. Therefore practical pointer analysis applies context sensitivity selectively using a context abstraction heuristic that determines the amount of context sensitivity that each method should receive [Jeong et al. 2017; Li et al. 2018a; Lu and Xue 2019; Smaragdakis et al. 2014]. Similarly, the performance of pointer analysis depends heavily on how heap objects are represented [Kanvar and Khedker 2016]. Pointer analysis usually employs allocation-site-based heap abstraction, which models heap objects with their allocation sites. However, because uniformly applying it to all heap objects is costly, a heap abstraction heuristic can be used to apply it selectively and otherwise use a less precise scheme such as type-based abstraction [Tan et al. 2017].

Trend: Graph-based Heuristics. A recent trend in state-of-the-art pointer analyses is use of graph-based analysis heuristics [Li et al. 2018a,b; Lu and Xue 2019; Tan et al. 2016, 2017]. These graph-based heuristics commonly work in the following two steps: (1) they first use a cheap pre-analysis to construct a graph representation of the input program and (2) they reason about the graph structure to produce a program-specific policy for the main analysis.

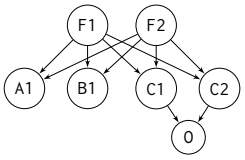
For example, Tan et al. [2016] presented Bean, which first runs a context-insensitive pre-analysis to generate the object allocation graph (OAG) and infers from it a policy for improving the precision of k-object-sensitive analysis. Li et al. [2018b] proposed Scaler, which also uses a context-insensitive pre-analysis to derive the object allocation graph and analyzes its structure to identify method calls that are likely to blow up the analysis cost during the 2-object-sensitive analysis. Li et al. [2018a] presented Zipper, another graph-based heuristic for context-sensitive analysis. Zipper uses a pre-analysis to generate a so-called precision flow graph (PFG) and identifies precision-critical method calls that may lose precision significantly if context insensitivity is used. Lu and Xue [2019] presented a graph-based heuristic, called Eagle, that uses a CFL-reachability-based pre-analysis to find out variables and objects that need context sensitivity in the main analysis. Tan et al. [2017] developed Mahjong, a graph-based heap abstraction heuristic that first runs a cheap pre-analysis to derive a field points-to graph (FPG) and decides when to merge and differentiate heap objects based on the structure of the points-to graph.

This Work. In this paper, we aim to advance this line of research by automating the process of creating graph-based analysis heuristics for pointer analysis. While all of the existing graph-based heuristics have been designed manually by analysis experts, our technique generates such heuristics automatically from a given graph without any human effort, significantly increasing applicability and accessibility of the emerging and promising approach in pointer analysis.

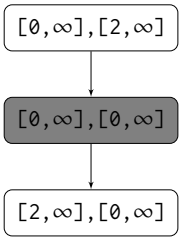
We achieve this goal by developing (1) a feature language for describing graph structures and (2) an algorithm for learning analysis heuristics in terms of the sentences of the language. We first present a language for describing structural features of nodes in a graph. This feature description language is simple and general, allowing it to be reused for various analysis instances (e.g. object sensitivity and heap abstraction). Second, we present a learning algorithm that takes training programs (and their graph representations) and produces graph-based heuristics by automatically discovering features appropriate for the given analysis task. Compared to prior data-driven static analysis techniques [He et al. 2020; Jeon et al. 2018; Jeong et al. 2017; Singh et al. 2018], a salient characteristic of our technique is that it does not require pre-designed, application-specific features;

```
1  class A{} class B{}
2  class C{
3    Object data;
4    void C(){
5      data = new Object;//O
6    }
7    void set(Object e){
8      data = e;
9    }
10   Object get(){
11     return data;
12   }
13 }
14 class F{
15   void foo(){
16     C c1 = new C();//C1
17     C c2 = new C();//C2
18
19     c1.set(new A());//A1
20     c2.set(new B());//B1
21     A a = (A)c1.get();
22     //query1
23     B b = (B)c2.get();
24     //query2
25     }
26   }
27 }
28
29 main(){
30   F f1 = new F(); //F1
31   f1.foo();
32   F f2 = new F(); //F2
33   f2.foo();
34 }
```

(a) Example code



(b) Object allocation graph



(c) Object-sensitivity heuristic

Fig. 1. Example to illustrate our graph-based object-sensitivity heuristic

instead, it uses a feature language to generate a proper set of features during the learning process. By contrast, existing learning-based techniques for static analysis need a different set of hand-tuned features for each analysis task.

The evaluation results show that our technique is effective and general; it can automatically produce competitive heuristics for two different analysis instances. We implemented our approach on top of the Doop pointer analysis framework for Java [Bravenboer and Smaragdakis 2009]. We used our approach to produce a object-sensitivity heuristic from the object allocation graph on which the state-of-the-art object-sensitivity heuristic Scaler [Li et al. 2018b] was developed. Additionally, we learned a heap abstraction heuristic from the field points-to graph, which is used in the state-of-the-art heap abstraction heuristic Mahjong [Tan et al. 2017]. For both instances, our approach successfully generated high-quality heuristics that are as competitive as Scaler and Mahjong in terms of the precision and scalability of the main analysis. In particular, the generated heuristic by our framework successfully analyzes large programs which the state-of-the-art heap abstraction heuristic, Mahjong, cannot handle within a time budget.

Contributions. In summary, this paper makes the following contributions:

- We present Graphick, a new technique for automatically learning graph-based heuristics for pointer analysis. Key technical contributions include a feature description language and a learning algorithm, which allow our approach to be generally used for different analysis instances without manually designing application-specific features.
- We demonstrate the effectiveness and generality of our technique in comparison with state-of-the-art heuristics for object sensitivity and heap abstraction.

2 OVERVIEW

We illustrate how our graph-based heuristic looks like and works with an example.

Example Program. Figure 1a is an example program with two queries checking the down-casting safety. This example has a main method that calls the method `foo` with two different receiver objects `F1` and `F2`. Class `C` provides getter and setter methods to manipulate its field `data`. Class `F` has a method `foo` which allocates two objects `C1` and `C2` to variables `c1` and `c2`, respectively. These variables call the `set` method with newly allocated objects `A1` and `B1`. There are two queries `query1` and `query2` asking the down-casting safety at lines 21 and 23. The safety holds because the `get` method returns objects `A1` and `B1` at lines 21 and 23, respectively.

Goal: Selective Object Sensitivity. Our goal is to analyze the program cost-effectively by applying context sensitivity only when it is necessary. To prove the queries, we need an object-sensitive analysis that differentiates the methods called under receiver objects `C1` and `C2`. Without object sensitivity, the analysis merges the methods `get` and `set` called on receiver objects `C1` and `C2`; eventually, the analysis misjudges that the `get` method can return both `A` and `B` at lines 21 and 23, and fails to prove the down-casting safety. The context-sensitive analysis, however, is not necessary for the method `foo` called from other objects `F1` and `F2` because `foo` is not related to the queries. If we apply context sensitivity to this method, it only increases the analysis cost without any precision gain. Thus, our heuristic aims to infer the following policy for the main analysis:

Apply object sensitivity only to method calls whose receiver objects are `C1` or `C2`.

Graph-based Heuristics. To generate such a policy, graph-based heuristics first run a cheap pre-analysis (e.g. context-insensitive analysis) to obtain a graph representation of the program. For object-sensitivity heuristics, the object allocation graph (OAG) has been considered as a good program representation [Li et al. 2018b; Tan et al. 2016]. Nodes in an OAG are heap objects (represented by allocation sites) and edges represent the connections between objects and their allocators. Figure 1b shows the OAG of the example program. In Figure 1b, for instance, two objects `F1` and `F2` have edges toward the objects `A1`, `B1`, `C1`, and `C2` because these four objects are allocated inside the method `foo` that is called on the receiver objects `F1` and `F2`. Given the OAG, the goal of graph-based heuristics is to choose a set of nodes in the graph. Ideally, a good heuristic would accurately identify the set {`C1`, `C2`} that needs object sensitivity during the main analysis.

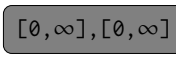
How Our Heuristic Works. Our heuristic consists of a set of *features*, where a feature describes a set of nodes in the given graph. A feature is of the form $(prev, ([a, b], [c, d]), succ)$, where $[a, b]$ and $[c, d]$ are intervals, and $prev$ and $succ$ are sequences of pairs of intervals. A node n in a graph is described by the feature iff the number of incoming edges of n is between a and b, the number of outgoing edges is between c and d, and the node has a sequence of predecessors satisfying *prev*, and the node has a sequence of successors satisfying *succ*.

For example, Figure 1c shows a heuristic comprising of a single feature $(prev, ([0, \infty], [0, \infty]), succ)$, where $prev$ and $succ$ are single pair of intervals (i.e. $prev = ([0, \infty], [2, \infty])$ and $succ = ([2, \infty], [0, \infty])$). It describes nodes that have at least 0 incoming and 0 outgoing edges, have a predecessor with at least 0 incoming and 2 outgoing edges, and have a successor with at least 2 incoming and 0 outgoing edges. In Figure 1b, `C1` and `C2` are the only nodes that satisfy these conditions because they have a successor (i.e. `O`) with two incoming edges and a predecessor (`F1` or `F2`) with four outgoing edges. From a set of training programs, our learning algorithm in Section 4.4 can generate such features automatically.

Given a graph and a set of features, our heuristic finds out all the nodes that satisfy one of the features. This information is used by the main analysis to perform a selective object-sensitive

analysis; the methods called under receiver objects C1 and C2 are analyzed with 1-object-sensitivity while the others are analyzed context insensitively.

Note that the performance of the main analysis heavily depends on the features in learned heuristics. For example, assume that a heuristic contains the following feature which takes off a predecessor of a target node from the feature in Figure 1c:



Unlike the feature in Figure 1c, which only C1 and C2 satisfy, four nodes C1, C2, F1, and F2 are implied by the above feature. Because the feature still includes precision-critical nodes, C1 and C2, the heuristic is able to prove the queries; however, it pays additional analysis costs as the set of nodes include F1 and F2 which are not related to the queries. As such, inappropriately learned heuristics can degrade the performance in costs and even the precision of the main analysis. Therefore, the goal of our learning algorithm is to find out qualified heuristics that are able to maintain as many precision-critical nodes as possible while excluding others that are not.

3 PRELIMINARIES

In this section, we define the baseline pointer analysis for Java-like languages (Section 3.1) and explain how to parameterize its context sensitivity (Section 3.2) and heap abstraction (Section 3.2).

3.1 Baseline Pointer Analysis

We consider the standard k-object-sensitive pointer analysis with allocation-site-based heap abstraction [Milanova et al. 2002; Smaragdakis et al. 2011].

Notation. Given a program, let $\mathbb{V}$ be the set of program variables, $\mathbb{H}$ the set of allocation sites, $\mathbb{M}$ the set of methods, $\mathbb{F}$ the set of field names, and $\mathbb{T}$ the set of class types in the program. We write $\mathbb{C}$ for the set of calling contexts and $\mathbb{HC}$ for the set of heap contexts. In object sensitivity, $\mathbb{C}$ and $\mathbb{HC}$ are defined to be sequences of allocation sites, i.e., $\mathbb{C} = \mathbb{HC} = \mathbb{H}^*$. Let $typeof : \mathbb{H} \to \mathbb{T}$ be a function that maps allocation sites to the types of the allocated objects. Given a method m, we write $this_m, param_m, return_m$ for the this variable, formal parameter, and return variable of the method m, respectively. Given a sequence $s = \langle a_1, a_2, \ldots, a_n \rangle$ and an element a', we write $s + a'$ for $\langle a_1, a_2, \ldots, a_n, a' \rangle$ and write $\lceil \langle a_1, a_2, \ldots, a_n \rangle \rceil_k$ for $\langle a_{n-k+1}, \ldots, a_n \rangle$.

Program. We consider five types of instructions: heap allocation, move, field load, field store, and method call. We assume that instructions are represented by the following relations:

$$\begin{aligned}
(var, heap, inMeth) \in \mathit{Alloc} &\subseteq \mathbb{V} \times \mathbb{H} \times \mathbb{M} \\
(to, from, inMeth) \in \mathit{Move} &\subseteq \mathbb{V} \times \mathbb{V} \times \mathbb{M} \\
(to, from, fld, inMeth) \in \mathit{FldLoad} &\subseteq \mathbb{V} \times \mathbb{V} \times \mathbb{F} \times \mathbb{M} \\
(to, fld, from, inMeth) \in \mathit{FldStore} &\subseteq \mathbb{V} \times \mathbb{F} \times \mathbb{V} \times \mathbb{M} \\
(return, base, callee, arg, caller) \in \mathit{Call} &\subseteq \mathbb{V} \times \mathbb{V} \times \mathbb{M} \times \mathbb{V} \times \mathbb{M}
\end{aligned}$$

The set *Alloc* represents all heap-allocating instructions in a given program. For example, when a heap cell is allocated and stored in a variable v at an allocation-site h (i.e. $v = new\, C$, where the instruction label is h and C denotes a class type), we represent the instruction by (v, h, m) where m is the method containing the instruction. Similarly, when m is the enclosing method, move $(x = y)$, field store $(x.f = y)$, field load $(x = y.f)$ instructions are represented by (x, y, m), (x, f, y, m), and (x, y, f, m), respectively. *Call* represents method calls in the program. When a method m_{caller} contains a call instruction $x = y.m_{callee}(arg)$, *Call* includes $(x, y, m_{callee}, arg, m_{caller})$. For presentation simplicity, we assume that methods take a single argument.

$$\frac{(var, heap, inMeth) \in \mathit{Alloc} \quad ctx \in \mathit{MethodCtx}(inMeth) \quad hctx = \lceil ctx \rceil_{maxH}}{(heap, hctx) \in \mathit{VarPtsTo}(var, ctx)}$$

$$\frac{(to, from, inMeth) \in \mathit{Move} \quad ctx \in \mathit{MethodCtx}(inMeth)}{\mathit{VarPtsTo}(from, ctx) \subseteq \mathit{VarPtsTo}(to, ctx)}$$

$$\frac{\begin{array}{c}(to, from, fld, inMeth) \in \mathit{FldLoad} \quad ctx \in \mathit{MethodCtx}(inMeth) \\ (heap, hctx) \in \mathit{VarPtsTo}(from, ctx)\end{array}}{\mathit{FldPtsTo}(heap, hctx, fld) \subseteq \mathit{VarPtsTo}(to, ctx)}$$

$$\frac{\begin{array}{c}(to, fld, from, inMeth) \in \mathit{FldStore} \quad ctx \in \mathit{MethodCtx}(inMeth) \\ (heap, hctx) \in \mathit{VarPtsTo}(to, ctx)\end{array}}{\mathit{VarPtsTo}(from, ctx) \subseteq \mathit{FldPtsTo}(heap, hctx, fld)}$$

$$\frac{\begin{array}{c}(return, base, callee, arg, caller) \in \mathit{Call} \quad ctx \in \mathit{MethodCtx}(caller) \\ (heap, hctx) \in \mathit{VarPtsTo}(base, ctx) \quad ctx' = \lceil hctx +\!\!+ heap \rceil_{maxK}\end{array}}{\begin{array}{c}ctx' \in \mathit{MethodCtx}(callee) \quad \mathit{VarPtsTo}(arg, ctx) \subseteq \mathit{VarPtsTo}(param_{callee}, ctx') \\ (heap, hctx) \in \mathit{VarPtsTo}(this_{callee}, ctx') \\ \mathit{VarPtsTo}(return_{callee}, ctx') \subseteq \mathit{VarPtsTo}(return, ctx)\end{array}}$$

Fig. 2. Pointer analysis rules with object sensitivity and allocation-site-based heap abstraction

Analysis Output. The goal of the analysis is to compute the following information:

- $\mathit{VarPtsTo} : \mathbb{V} \times \mathbb{C} \to \wp(\mathbb{H} \times \mathbb{HC})$
- $\mathit{FldPtsTo} : \mathbb{H} \times \mathbb{HC} \times \mathbb{F} \to \wp(\mathbb{H} \times \mathbb{HC})$
- $\mathit{MethodCtx} : \mathbb{M} \to \wp(\mathbb{C})$

The points-to information is classified into *VarPtsTo* and *FldPtsTo*. *VarPtsTo* maps each pointer variable qualified with a calling context to a set of abstract heaps, where an abstract heap consists of an allocation site and a heap context. *FldPtsTo* maps each object's field locations to abstract heaps. *MethodCtx* maps each method to the set of its reachable contexts.

In recent pointer analyses, graph representations of the analysis results have been widely used and our technique also leverages them. Notable examples include object allocation graph (OAG) [Tan et al. 2016] and field points-to graph (FPG) [Tan et al. 2017]. The object allocation graph is a directed graph, $G_{OAG} = (N_{OAG}, \hookrightarrow_{OAG})$, where nodes are allocation sites in the program (i.e. $N_{OAG} = \mathbb{H}$) and edges $(\hookrightarrow_{OAG}) \subseteq \mathbb{H} \times \mathbb{H}$ describe the object allocation relation defined as follows:

$$h \hookrightarrow_{OAG} h' \iff \exists m \in \mathbb{M}.\ (h, \_) \in \mathit{VarPtsTo}(this_m, \_) \text{ and } (\_, h', m) \in \mathit{Alloc}.$$

In words, we have $h \hookrightarrow_{OAG} h'$ if h is a receiver object of method m, i.e. $(h, \_) \in \mathit{VarPtsTo}(this_m, \_)$, and m allocates h', i.e. $(\_, h, m) \in \mathit{Alloc}$. Intuitively, object allocation graph is the "call graph" in object sensitivity, which provides information about how each context is constructed in k-object-sensitive analysis [Li et al. 2018b]. The field points-to graph $G_{FPG} = (N_{FPG}, \hookrightarrow_{FPG})$ is simply a context-insensitive representation of the *FldPtsTo* relation. We define $N_{FPG} = \mathbb{H}$ and $h \hookrightarrow_{FPG} h' \iff (h', \_) \in \mathit{FldPtsTo}(h, \_, \_)$.

Analysis Rules. Figure 2 shows the rules for computing the analysis results. Let $maxK$ and $maxH$ be the maximum lengths to maintain for call and heap contexts, respectively. Suppose that $(var, heap, inMeth)$ is in *Alloc*, ctx is a reachable context of $inMeth$ (i.e. $ctx \in \mathit{MethodCtx}(inMeth)$),

$$\frac{\begin{array}{c}(return, base, callee, \arg, caller) \in Call \\ ctx \in MethodCtx(caller) \quad (heap, hctx) \in VarPtsTo(base, ctx) \\ ctx' = \begin{cases} \lceil hctx +\!\!+ heap \rceil_{maxK} & \text{if } ContextAbstraction(heap) = maxK \\ \lceil hctx +\!\!+ heap \rceil_{maxK-1} & \text{if } ContextAbstraction(heap) = maxK - 1 \\ \ldots & \\ \lceil hctx +\!\!+ heap \rceil_{0} & \text{if } ContextAbstraction(heap) = 0 \end{cases}\end{array}}{\begin{array}{c} ctx' \in MethodCtx(callee) \quad VarPtsTo(arg, ctx) \subseteq VarPtsTo(param_{callee}, ctx') \\ (heap, hctx) \in VarPtsTo(this_{callee}, ctx') \quad VarPtsTo(return_{callee}, ctx') \subseteq VarPtsTo(return, ctx)\end{array}}$$

Fig. 3. Parametric object sensitivity

$$\frac{\begin{array}{c}(var, heap, inMeth) \in Alloc \qquad ctx \in MethodCtx(inMeth) \\ hctx = \lceil ctx \rceil_{maxH} \quad heap' = \begin{cases} heap & \text{if } HeapAbstraction(heap) = \text{`}alloc\text{'} \\ typeof(heap) & \text{if } HeapAbstraction(heap) = \text{`}type\text{'} \end{cases}\end{array}}{(heap', hctx) \in VarPtsTo(var, ctx)}$$

Fig. 4. Parametric heap abstraction

and $hctx$ is a heap context obtained by truncating the last $maxH$ elements of ctx (i.e. $hctx = \lceil ctx \rceil_{maxH}$). Then, *VarPtsTo*$(var, ctx)$ should include $(heap, ctx)$. Analysis rules for *Move*, *FldLoad*, and *FldStore* are defined similarly. The rule for *Call* describes the standard k-object-sensitive analysis [Milanova et al. 2005; Smaragdakis et al. 2011]. Suppose a method is called on a base variable $base$ with a context ctx, $(heap, hctx)$ is a receiver object, and ctx' is a new calling context. The context ctx' is obtained by appending $heap$ to the heap context $hctx$ of the receiver object (i.e. $hctx +\!\!+ heap$) and truncating the result (i.e. $\lceil hctx +\!\!+ heap \rceil_{maxK}$). Then, ctx' becomes a reachable context of the callee (i.e. $ctx' \in MethodCtx(callee)$), the points-to set of the formal parameter of the callee (denoted $param_{callee}$) is updated with that of the actual parameter, the $this$ variable of the callee points-to the receiver object, and the points-to set of the return variable of the callee (denoted $return_{callee}$) is transferred to the return variable of the caller.

3.2 Parameterization

Next, we parameterize the baseline pointer analysis.

Parametric Object Sensitivity. The analysis in Figure 2 uses the same $maxK$ value for every method call. The parametric object-sensitive analysis generalizes it to be able to assign different call depths for different method calls. To do so, the parameterized analysis uses the rule in Figure 3 instead of the last rule in Figure 2. In Figure 3, we use the function $ContextAbstraction : \mathbb{H} \to [0, maxK]$, which assigns a context depth between 0 and $maxK$ to each method call. When a method is called on a receiver object $heap$, $ContextAbstraction$ produces an appropriate context depth for it. In Section 4, we present a technique for automatically learning a heuristic that produces the $ContextAbstraction$ information for a given program.

Parametric Heap Abstraction. The analysis in Figure 2 uses allocation-based heap abstraction for every heap object. We can generalize it to support selective use of allocation-site- and type-based heap abstractions. We first need to generalize the analysis results as follows:

- *VarPtsTo* : $\mathbb{V} \times \mathbb{C} \to \wp((\mathbb{H} + \mathbb{T}) \times \mathbb{HC})$

- $\mathit{FldPtsTo} : (\mathbb{H} + \mathbb{T}) \times \mathbb{HC} \times \mathbb{F} \to \wp((\mathbb{H} + \mathbb{T}) \times \mathbb{HC})$
- $\mathit{MethodCtx} : \mathbb{M} \to \wp(\mathbb{C})$

VarPtsTo maps each variable with a context to abstract heaps, where an abstract heap is now either allocation site ($\mathbb{H}$) or class type ($\mathbb{T}$) with their heap context. *FldPtsTo* is also extended in a similar way to support type-based abstraction. We replace the rule for *Alloc* in Figure 2 by the parameterized rule in Figure 4. The new rule uses the function $\mathit{HeapAbstraction}$ that takes an allocation site (heap) and determines whether we use allocation-site-based abstraction ('*alloc*') or type-based abstraction ('*type*'). When $\mathit{HeapAbstraction}$ returns '*type*', we use the class type of the allocated object (i.e. $\mathit{typeof}\,(\mathit{heap})$) instead of the allocation site. Otherwise, the analysis performs the conventional allocation-site-based heap abstraction. Our technique in Section 4 can be also used for learning a heuristic that produces appropriate $\mathit{HeapAbstraction}$ for each input program.

4 GRAPHICK

In this section, we present our approach for automatically learning graph-based analysis heuristics. In Section 4.1, we define static analyses with k-limited abstractions. Section 4.2 presents a feature description language for directed graphs, which is important for the generality and effectiveness of our approach. In Section 4.3, we define a parameterized abstraction heuristic based on the feature language. Section 4.4 presents our algorithm for learning parameters of the heuristic.

4.1 Static Analyses with K-Limited Abstractions

Let us first model a static analysis with k-limited abstractions. Given a program P to analyze, let $\mathbb{C}_P$ be a finite set of program components in P. For example, $\mathbb{C}_P$ may denote the set of methods [Jeong et al. 2017] or the set of allocation sites [Tan et al. 2017] in P. We define $\mathcal{A}_P$ to be the set of abstractions over $\mathbb{C}_P$ as follows:

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

An abstraction $\mathbf{a} \in \mathcal{A}_P$ maps program components to natural numbers between 0 and k. For example, in a partially context-sensitive analysis, it assigns one of $0, 1, \ldots, k$ to each method call, indicating the amount of context sensitivity that each method is allowed to receive during the analysis. Abstractions are partially ordered as follows:

$$\mathbf{a}_1 \sqsubseteq \mathbf{a}_2 \iff \forall c \in \mathbb{C}_P.\ \mathbf{a}_1(c) \leq \mathbf{a}_2(c).$$

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

$$\mathbf{k} = \lambda c \in \mathbb{C}_P.\ k, \qquad \mathbf{0} = \lambda c \in \mathbb{C}_P.\ 0$$

We assume that a set $\mathbb{Q}_P$ of assertions is given together with the program P. For instance, $\mathbb{Q}_P$ may denote the set of all type casts in the program. The goal of static analysis is to prove that assertions in $\mathbb{Q}_P$ do not fail at runtime. We model a static analyzer as a function that takes as input an abstraction and produces a set of proved queries and, as a by-product, a directed graph over program components:

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

where $\mathbb{G}_P$ denotes the set of possible graphs. A graph $G = (N, \hookrightarrow) \in \mathbb{G}_P$ consists of nodes $N = \mathbb{C}_P$ and edges $(\hookrightarrow) \subseteq \mathbb{C}_P \times \mathbb{C}_P$. For example, G is the object allocation graph [Li et al. 2018b] or the field points-to graph [Tan et al. 2017] depending on the purpose of the analysis. We use two projection functions, `proved` and `graph`, which are used for obtaining the proven queries and the graph, respectively, from the analysis output.

In this paper, we generally assume the analysis F_P is monotone with respect to the abstractions in the sense that more refined abstractions imply higher analysis precision:

$$\mathbf{a} \sqsubseteq \mathbf{a}' \implies \texttt{proved}(F_P(\mathbf{a})) \subseteq \texttt{proved}(F_P(\mathbf{a}')). \tag{1}$$

Many static analysis problems are monotone [Jeong et al. 2017; Li et al. 2018a; Liang and Naik 2011; Liang et al. 2011; Tan et al. 2017; Zhang et al. 2014] and therefore our approach is directly applicable to them. For non-monotone analyses (e.g. interval analysis with widening [Cha et al. 2016]), our approach is still applicable in practice but it does not guarantee its theoretical property (Theorem 4.2).

4.2 Feature Description Language

Our approach uses a simple and general language for describing properties of nodes in a graph.

Observation. Our feature language has been inspired from existing graph-based heuristics. Existing works [Li et al. 2018a,b; Tan et al. 2016] have demonstrated that the number of incoming and outgoing edges of nodes in graphs play key roles in designing analysis heuristics. For example, Li et al. [2018a] identify precision-critical method calls by figuring out the nodes with multiple incoming edges in precision flow graph (PFG). Besides the PFG, the number of incoming edges in object allocation graph (OAG) also helps to design effective analysis heuristics, which is used by both Tan et al. [2016] and Li et al. [2018b]. The conventional 2-object-sensitive analysis produces only one heap context for objects when they have only one incoming edge in OAG. Tan et al. [2016], however, design an analysis heuristic which assigns alternative multiple heap contexts to these objects for improving precision. When an object has lots of incoming edges, multiple contexts are applied to the methods called from the object in 2-object-sensitive analysis. These methods are critical for scalability in pointer analysis, and Li et al. [2018b] identify these methods to apply alternatively coarse yet cheap contexts to improve the performance in scalability. Based on these observations, we designed a feature language that can express various combinations of the number of edges around nodes, successors, and predecessors.

Formal Definition. Let $G = (N, \hookrightarrow)$ be a directed graph over program components, i.e., $N = \mathbb{C}_P$ and $(\hookrightarrow) \subseteq \mathbb{C}_P \times \mathbb{C}_P$. Let $In_G(n)$ and $Out_G(n)$ be the numbers of incoming and outgoing edges, respectively, of node n in graph G:

$$In_G(n) = |\{p \in N \mid p \hookrightarrow n\}|, \qquad Out_G(n) = |\{s \in N \mid n \hookrightarrow s\}|.$$

A *feature* in our language denotes a set of nodes. We define a feature f to be a triple:

$$f \in Feature = \widehat{Node}^* \times \widehat{Node} \times \widehat{Node}^*$$

where $\widehat{Node}$ means abstract nodes:

$$\begin{array}{rcl} \hat{n}, \hat{p}, \hat{s} \in \widehat{Node} & = & Itv \times Itv \\ Itv & = & \{[a, b] \mid a \in \mathbb{N}, b \in \mathbb{N} \cup \{\infty\}\} \end{array}$$

An abstract node $([a, b], [c, d]) \in \widehat{Node}$ is a pair of intervals and denotes a set of nodes as follows:

$$\gamma_G(([a, b], [c, d])) = \{n \in N \mid a \leq In_G(n) \leq b, c \leq Out_G(n) \leq d\}.$$

We extend the definition to a sequence of abstract nodes ($\widehat{Node}^*$). The empty sequence ϵ denotes the empty set of nodes. A non-empty sequence $\langle (itv_0, itv'_0), (itv_1, itv'_1), \ldots, (itv_m, itv'_m)\rangle$ of pairs of intervals denotes sequences of nodes as follows:

$$\begin{array}{l} \gamma_G(\langle (itv_0, itv'_0), (itv_1, itv'_1), \cdots, (itv_m, itv'_m)\rangle) = \\ \qquad \{\langle n_0, n_1, \ldots, n_m\rangle \in N^* \mid n_0 \hookrightarrow n_1 \hookrightarrow \cdots \hookrightarrow n_m, \forall i \in [0, m].\ n_i \in \gamma_G((itv_i, itv'_i))\}. \end{array}$$

Finally, a feature $(\langle \hat{p}_0, \hat{p}_1, \ldots, \hat{p}_q \rangle, \hat{n}, \langle \hat{s}_0, \hat{s}_1, \ldots, \hat{s}_r \rangle) \in \mathit{Feature}$ denotes a set of nodes in $\gamma_G(\hat{n})$ whose predecessors and successors are described as $\langle \hat{p}_0, \hat{p}_1, \ldots, \hat{p}_q \rangle$ and $\langle \hat{s}_0, \hat{s}_1, \ldots, \hat{s}_r \rangle$, respectively:

$$\gamma_G(\langle \hat{p}_0, \hat{p}_1, \ldots, \hat{p}_q \rangle, \hat{n}, \langle \hat{s}_0, \hat{s}_1, \ldots, \hat{s}_r \rangle) = \{n \in \gamma_G(\hat{n}) \mid \exists p_0, p_1, \ldots, p_q, s_0, s_1, \ldots, s_r \in N.$$
$$\langle p_0, p_1, \ldots, p_q \rangle \in \gamma_G(\langle \hat{p}_1, \hat{p}_2, \ldots, \hat{p}_q \rangle), p_q \hookrightarrow n \hookrightarrow s_0, \langle s_0, s_1, \ldots, s_r \rangle \in \gamma_G(\langle \hat{s}_1, \hat{s}_2, \ldots, \hat{s}_r \rangle).\}$$

For example, feature $(\epsilon, ([0,3],[5,\infty]), \langle([0,2],[0,0])\rangle)$ describes the set of nodes that have 1) three or less incoming edges and five or more outgoing edges, and 2) a successor node with two or less incoming edges and no outgoing edges. For another example, the following feature

$$(\langle([0,0],[0,5]),([1,2],[3,\infty])\rangle, ([0,3],[100,\infty), \langle([1,1],[2,2])\rangle)$$

describes a node n iff 1) n has three or less incoming edges and 100 or more outgoing edges, 2) n has a predecessor p with one or two incoming edges and three or more outgoing edges, 3) p also has a predecessor with no incoming edge and five or less outgoing edges, and 4) n has a successor s with a single incoming edge and two outgoing edges.

4.3 Parameterized Abstraction Heuristic

In our approach, abstraction heuristics work on a graph over program components. That is, a heuristic $\mathcal{H}$ is a function that takes a graph G for program P and produces an abstraction of P:

$$\mathcal{H}(G) : \mathbb{C}_P \to \{0, 1, \ldots, k\}.$$

The graph G is typically obtained by running an imprecise but fast pre-analysis [Li et al. 2018b; Lu and Xue 2019; Tan et al. 2016, 2017]. For example, it can be obtained by running the analysis F_P with the least precise abstraction:

$$G = \texttt{graph}(F_P(\mathbf{0})).$$

We define a *template* of such heuristics whose behavior is controlled by k parameters $\Pi = \langle \mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k \rangle$, where each parameter $\mathcal{F}_i \subseteq \mathit{Feature}$ is a set of features in our language. We define the parameterized heuristic $\mathcal{H}_\Pi$ as follows:

$$\mathcal{H}_\Pi(G) = \lambda c \in \mathbb{C}_P. \begin{cases} k & \text{if } c \in \gamma_G(\mathcal{F}_k) \\ k-1 & \text{if } c \in \gamma_G(\mathcal{F}_{k-1}) \wedge c \notin \gamma_G(\mathcal{F}_k) \\ \quad\cdots & \\ k-i & \text{if } c \in \gamma_G(\mathcal{F}_{k-i}) \wedge c \notin \bigcup_{k \geq j > k-i} \gamma_G(\mathcal{F}_j) \\ \quad\cdots & \\ 1 & \text{if } c \in \gamma_G(\mathcal{F}_1) \wedge c \notin \bigcup_{k \geq j > 1} \gamma_G(\mathcal{F}_j) \\ 0 & \text{otherwise} \end{cases}$$

where $\gamma_G(\mathcal{F}_i) = \bigcup_{f \in \mathcal{F}_i} \gamma_G(f)$. Basically, the heuristic $\mathcal{H}_\Pi$ assigns an abstraction degree j to program component c if c is implied by the j^{th} parameter $\mathcal{F}_j$. If c is implied by multiple parameters, the heuristic assigns the highest abstraction degree among them. For example, when $c \in \mathcal{F}_1$ and $c \in \mathcal{F}_2$, we define $\mathcal{H}_\Pi(G)(c) = 2$.

4.4 Learning Algorithm

Now we present an algorithm for learning parameters $\Pi = \langle \mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k \rangle$ from a set $\mathrm{P} = \{P_1, P_2, \ldots, P_n\}$ of training programs.

Overall Process. Algorithm 1 describes the overall learning process. The algorithm takes training programs P, static analyzer F, and the maximum abstraction degree k. As output, it produces k parameters $\Pi = \langle \mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k \rangle$. Initially, parameters Π are set to empty sets $\langle \emptyset, \emptyset, \ldots, \emptyset \rangle$ (line 2). At line 3, the algorithm computes a map A_P from programs in P to their *ideal* abstractions. For each training program $P_i \in \mathrm{P}$, $A_\mathrm{P}(P_i)$ denotes the desired abstraction for P_i that we want our heuristic

Algorithm 1 Overall learning algorithm

Input: Training programs P, static analyzer F, maximum abstraction degree k
Output: Parameters $\langle \mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k \rangle$

1: **procedure** LEARN(P, F, k)
2: $\langle \mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k \rangle \leftarrow \langle \emptyset, \emptyset, \ldots, \emptyset \rangle$
3: $A_{\mathrm{P}} \leftarrow \lambda P \in \mathrm{P}.$LEARNMINIMALABSTRACTION($P, F, k$) ▷ minimal abstractions
4: $G_{\mathrm{P}} \leftarrow \lambda P \in \mathrm{P}.\texttt{graph}(F_P(\mathbf{0}))$ ▷ graphs from pre-analysis
5: **for** $i = 1$ to k **do**
6: $\mathcal{F}_i \leftarrow$ LEARNSETOFFEATURES($i, A_{\mathrm{P}}, G_{\mathrm{P}}$)
7: **end for**
8: **return** $\langle \mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k \rangle$
9: **end procedure**

Algorithm 2 Learning minimal abstraction

Input: Program P, static analyzer F, maximum abstraction degree k
Output: A minimal abstraction for P

1: **procedure** LEARNMINIMALABSTRACTION(P, F, k)
2: $C \leftarrow \mathbb{C}_P$
3: $\mathbf{a} \leftarrow \lambda c.k$
4: **for** $i = k$ to 1 **do**
5: $C' \leftarrow C$
6: **while** $C' \neq \emptyset$ **do**
7: $c' \leftarrow pick(C')$
8: $C' \leftarrow C' \setminus \{c'\}$
9: $\mathbf{a}' \leftarrow \lambda c.\mathit{if}\ c = c'\ \mathit{then}\ i - 1\ \mathit{else}\ \mathbf{a}(c)$
10: **if** $\texttt{proved}(F_P(\mathbf{k})) = \texttt{proved}(F_P(\mathbf{a}'))$ **then**
11: $\mathbf{a} \leftarrow \mathbf{a}'$
12: **end if**
13: **end while**
14: $C \leftarrow C \setminus \{c \mid \mathbf{a}(c) = i\}$
15: **end for**
16: **return a**
17: **end procedure**

to produce for P_i. The ideal abstraction is computed by procedure LEARNMINIMALABSTRACTION, which is explained shortly. At line 4, we run a pre-analysis (e.g. $F_P(\mathbf{0})$) to transform each training program P into its graph representation: G_{P} is a map from programs in P to their graph representations. At lines 5 and 6, the algorithm uses procedure LEARNSETOFFEATURES to learn each parameter $\mathcal{F}_i$ ($1 \leq i \leq k$), which denotes the set of nodes in the graphs in G_{P} that should receive the abstraction degree i. Although $\mathcal{F}_i$ is obtained iteratively, there is no dependency between loop iterations and therefore the k different tasks at lines 5 and 6 can be run in parallel to reduce the learning cost. At line 8, the learned parameters $\langle \mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k \rangle$ are returned as final output.

Learning Minimal Abstraction. The objective of learning is to find a set of parameters $\Pi = \langle \mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k \rangle$ with which the heuristic $\mathcal{H}_\Pi$ can produce ideal abstractions for training programs. We define ideal abstractions to be minimal abstractions [Liang et al. 2011] and therefore the learning

objective is as follows:

$$\text{Find } \Pi = \langle \mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k \rangle \text{ such that } \forall P_i \in \mathrm{P}. \mathcal{H}_\Pi(G_i) \text{ is a minimal abstraction for } P_i.$$

where G_i is a graph obtained by running a pre-analysis on P_i (e.g. $G_i = \texttt{graph}(F_{P_i}(\mathbf{0}))$). The definition of minimal abstractions is as follows:

Definition 4.1 (Minimal Abstraction [Liang et al. 2011]). An abstraction $\mathbf{a}$ is a minimal abstraction for program P if

(1) $\mathbf{a}$ is precise: $\texttt{proved}(F_P(\mathbf{a})) = \texttt{proved}(F_P(\mathbf{k}))$, and
(2) $\mathbf{a}$ is minimal: $(\mathbf{a}' \sqsubseteq \mathbf{a} \wedge \texttt{proved}(F_P(\mathbf{a}')) = \texttt{proved}(F_P(\mathbf{a}))) \implies \mathbf{a}' = \mathbf{a}$.

Algorithm 2 presents our algorithm for efficiently computing a minimal abstraction for program P. Our algorithm is similar to the ScanCoarsen algorithm by Liang et al. [2011], but ours is more efficient than the prior algorithm as we exploit the high-level structure of k-limited abstractions to reduce the search space. The algorithm by Liang et al. [2011] first transforms k-limited abstractions into binary abstractions (where k is 1), losing the opportunity to leverage the properties of the search space induced by monotone k-limited analyses. As a result, the size of search space is $(k+1)^{|\mathbb{C}_P|}$ for the existing algorithm [Liang et al. 2011]. We safely reduce the space to $k \cdot 2^{|\mathbb{C}_P|}$.

At line 2, we set C to all program components $\mathbb{C}_P$. The algorithm begins with the most precise abstraction (line 3). At lines 4–15, it considers each of the abstraction degrees $1, 2, \ldots, k$ in reverse. Iterating the abstraction degrees in reverse (from k to 1) is important to reduce the search space safely. At lines 6–13, it iteratively picks a program component (line 7) and assigns the lower abstraction degree $i-1$ to it (line 9). At line 10, the algorithm checks if the refined abstraction still preserves the precision; if so, the lower abstraction degree is sufficient for that program component. Otherwise, the program component needs the degree i to preserve the precision. At the end of the iteration (line 14), we exclude from C the program components that are determined to require the current degree i (i.e. $\{c \mid \mathbf{a}(c) = i\}$). In the worst case (when the minimal abstraction is $\lambda c.0$), our algorithm iterates $k \cdot |C|$ times where the search space for each degree i is 2^C and we have k different degrees. Although the algorithm considers a significantly smaller search space than the original one, it still guarantees to find a minimal abstraction:

Theorem 4.2. *Algorithm 2 returns a minimal abstraction for the input program P.*

Proof. See the proof in the link.[1] □

Learning a Set of Features. Algorithm 3 describes the algorithm for learning a set of features. It takes the abstraction level i, minimal abstractions A_{P}, and graphs G_{P} as input. It returns as output a set of features $\mathcal{F}$ that best describe the nodes assigned the abstraction level i according to minimal abstractions in A_{P}. At line 2, it collects all program components C (e.g. nodes) whose abstraction degrees are i according to minimal abstractions. At line 3, it initializes $\mathcal{F}$ to be the empty set. At lines 4–8, the algorithm iteratively calls LearnFeature to generate a feature. The algorithm adds the generated features to $\mathcal{F}$ until the features cover all program components, and returns $\mathcal{F}$ as learned features when $\mathcal{F}$ does so.

Learning a Feature. Algorithm 4 presents how each feature f in $\mathcal{F}$ is learned. LearnFeature takes as input components C and graphs G_{P}, and aims to generate a feature f that maximizes the following score function:

$$Score(f, C) = \frac{\sum_{P \in \mathrm{P}} |C \cap \gamma_{G_{\mathrm{P}}(P)}(f)|}{\sum_{P \in \mathrm{P}} |\gamma_{G_{\mathrm{P}}(P)}(f)|}$$

[1] http://doi.org/10.5281/zenodo.4216569

Algorithm 3 Learning a set of features

Input: Abstraction level i, minimal abstractions A_{P}, graphs G_{P}
Output: A set $\mathcal{F}$ of features

```
1:  procedure LearnSetOfFeatures(i, A_P, G_P)
2:      C ← {c | P ∈ P, c ∈ ℂ_P, A_P(P)(c) = i}
3:      F ← ∅
4:      while C ≠ ∅ do
5:          f ← LearnFeature(C, G_P)
6:          F ← F ∪ {f}
7:          C ← C \ {c | P ∈ P, c ∈ ℂ_P, c ∈ γ_{G_P(P)}(f)}
8:      end while
9:      return F
10: end procedure
```

Algorithm 4 Learning a feature

Input: Program components C, graphs G_{P}
Output: A feature f

```
1:  procedure LearnFeature(C, G_P)
2:      f ← (ϵ, ([0, ∞], [0, ∞]), ϵ)
3:      f' ← (ϵ, ([0, ∞], [0, ∞]), ϵ)
4:      do
5:          f ← f'
6:          f' ← Refine(f, C)
7:          if Score(f', C) ≥ θ then
8:              return f'
9:          end if
10:     while Score(f', C) > Score(f, C)
11:     return f
12: end procedure
```

where the score is a real number between 0 and 1. Intuitively, the score describes how accurately a feature describes the program components in C. For example, the score becomes the highest value 1 when $\forall P \in \mathrm{P}.\ \gamma_{G_{\mathrm{P}}(P)}(f) \subseteq C$. The score decreases as the feature selects components not in C.

The algorithm starts from the most general feature, i.e., $(\epsilon, ([0,\infty],[0,\infty]), \epsilon)$, and iteratively refines it until the feature becomes sufficiently informative, meaning that the score of the refined feature becomes higher than the hyper parameter θ. The value of θ has great impacts on the performance of learned heuristics, and we discuss how we determine the value of θ in Section 5.2. At lines 4–10, the algorithm iteratively calls *Refine* to make the current feature f more specific. When no more improvement is possible (i.e. $\mathit{Score}(f', C) \leq \mathit{Score}(f, C)$), the loop terminates and the algorithm returns the current feature f. We define the refinement function *Refine* as follows:

$$\mathit{Refine}(f, C) = \underset{f' \in \mathit{Append}(f) \cup \mathit{Replace}(f)}{\operatorname{argmax}} \mathit{Score}(f', C)$$

where $\mathit{Append}(f)$ and $\mathit{Replace}(f)$ produce new features that are more specific than f. From the set of new features, *Refine* chooses the one with the highest score. $\mathit{Append}(f)$ denotes the features that

are obtained by appending an abstract node to f:

$$\begin{aligned}&\mathit{Append}((\langle \hat{p}_0, \ldots, \hat{p}_q\rangle, \hat{n}, \langle \hat{s}_0, \ldots, \hat{s}_r\rangle)) = \\ &\{(\langle \hat{a}', \hat{p}_0, \ldots, \hat{p}_q\rangle, \hat{n}, \langle \hat{s}_0, \ldots, \hat{s}_r\rangle), (\langle \hat{p}_0, \ldots, \hat{p}_q\rangle, \hat{n}, \langle \hat{s}_0, \ldots, \hat{s}_r, \hat{a}'\rangle) \mid \hat{a}' \in \mathit{Specify}([0,\infty],[0,\infty]))\}\end{aligned}$$

where the function *Specify* denotes a strategy for making a feature more specific. In experiments, we used the following strategy:

$$\mathit{Specify}([a,b],[c,d])) = \{([\tfrac{a+b}{2}, b],[c,d]), ([a, \tfrac{a+b}{2}],[c,d]), ([a,b],[\tfrac{c+d}{2}, d]), ([a,b],[c, \tfrac{c+d}{2}])\}$$

where, if b (resp., d) equals to ∞, it is replaced by the maximum number of incoming (resp., outgoing) edges in the training graphs (G_{P}).

The definition of *Specify* is a design choice. For example, we can consider the following definition for *Specify*:

$$\begin{aligned}&\mathit{Specify}([a,b],[c,d])) = \\ &\{([a+\tfrac{b-a}{3}, b],[c,d]), ([a, b-\tfrac{b-a}{3}],[c,d]), ([a,b],[c+\tfrac{d-c}{3}, d]), ([a,b],[c, d-\tfrac{d-c}{3}])\}.\end{aligned}$$

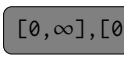
With the above definition, we can still find desirable features for the training set. It, however, takes more iterations to obtain such features because it specifies the interval value of features more carefully than the one we used.

On the other hand, $\mathit{Replace}(f)$ denotes the features that are obtained by replacing one of the abstract nodes in f by more specific ones:

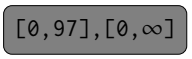
$$\begin{aligned}&\mathit{Replace}((\langle \hat{p}_0, \ldots, \hat{p}_q\rangle, \hat{n}, \langle \hat{s}_0, \ldots, \hat{s}_r\rangle)) = \\ &\quad \{(\langle \hat{p}_0, \ldots, \hat{p}_q\rangle, \hat{n}', \langle \hat{s}_0, \ldots, \hat{s}_r\rangle) \mid \hat{n}' \in \mathit{Specify}(\hat{n})\} \\ &\cup \{(\langle \hat{p}'_0, \ldots, \hat{p}'_q\rangle, \hat{n}, \langle \hat{s}_0, \ldots, \hat{s}_r\rangle) \mid j \in [0,q], \hat{p}'_j \in \mathit{Specify}(\hat{p}_j), \forall i \neq j.\ \hat{p}'_j = \hat{p}_j\} \\ &\cup \{(\langle \hat{p}_0, \ldots, \hat{p}_q\rangle, \hat{n}, \langle \hat{s}'_0, \ldots, \hat{s}'_r\rangle) \mid j \in [0,r], \hat{s}'_j \in \mathit{Specify}(\hat{s}_j), \forall i \neq j.\ \hat{s}'_j = \hat{s}_j\}\end{aligned}$$

Example. With an example, we explain how an actual feature used in our evaluation (Section 5.1.2) is generated where θ is 0.5.

(1) Our algorithm starts from the most general feature f:

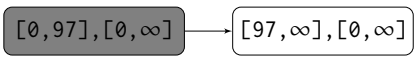


(2) It enumerates 12 cases of refined features from $f = (\epsilon, ([0,\infty],[0,\infty]), \epsilon)$ (e.g., 8 cases of $\mathit{Append}(f)$ and 4 cases of $\mathit{Replace}(f)$). It chooses the following feature produced from $\mathit{Replace}(f)$:



which has the highest score 0.06 among the 12 cases of features.

(3) Because the score is less than 0.5, it refines the feature again to the following specific one, which comes from $\mathit{Append}(\epsilon, ([0,97],[0,\infty]), \epsilon)$, with the same manner:

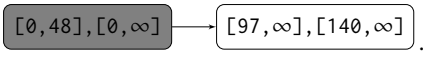


where the feature has score 0.23.

(4) To find a better one, it refines the feature further; it enumerates 16 cases of refined features (e.g., 8 cases for replacing and 8 cases for appending a node). The following feature is selected:

where the score is 0.37.

(5) In the next iteration, it finally finds an informative feature which has a score 0.55:



5 EVALUATION

In this section, we experimentally evaluate our technique for learning graph-based heuristics. We aim to answer the following research questions:

- **Effectiveness and Generality:** How effectively does the learned heuristic perform compared to the state-of-the-art heuristics? Is it generally applicable for different analysis tasks without manual effort for designing application-specific features?
- **Learning Algorithm:** How much does the learning cost? How does the hyper-parameter θ affect the performance of the learned heuristics?
- **Learned Insight:** Does our approach produce explainable heuristics? What are the insights learned from the generated heuristics?

Overall Setting. We implemented our approach, as a tool Graphick, on top of Doop [Bravenboer and Smaragdakis 2009], a pointer analysis framework for Java that has been widely used in prior works [Jeon et al. 2018; Jeong et al. 2017; Smaragdakis et al. 2014; Tan et al. 2016, 2017]. For the precision and scalability metrics, we follow existing works [Jeong et al. 2017; Li et al. 2018a,b; Tan et al. 2017] and use the number of may-fail casts alarms and the time spent on each analysis. We also use the number of polymorphic call sites (i.e. call sites whose targets are not uniquely determined by each pointer analysis) and call-graph edges as additional precision metrics. For all precision metrics, the lower is the better. We set the time budget as 3 hours (10,800 sec) for all analyses. For the hyper-parameter θ, we chose the one among various values (e.g., 0.1, 0.2, ..., 0.9) via cross validation (we explain how this is done in section 5.2). For each feature, we limit it to have at most three nodes due to scalability. All the experiments were done on a machine with i7 CPU and 64 GB RAM running Ubuntu 16.04 (64bit). We used the OpenJDK (1.6.0_24) library.

We used a total of 17 programs: 10 programs (luindex, lusearch, antlr, pmd_m, chart, eclipse, fop, bloat, xalan, and jython) from the DaCapo 2006-10-MR2 benchmark suite [Blackburn et al. 2006] and 7 programs (pmd_s, jedit, briss, soot, findbugs, JPC, and checkstyle) obtained from the artifacts provided by Tan et al. [2017] and Li et al. [2018b]. Here, we used two different versions of pmd where pmd_m is a small program used by Tan et al. [2017], and pmd_s is an open-source application used by Li et al. [2018b]. We split the benchmark programs into training, validation, and test sets. The training and validation sets are used for learning a heuristic, and the test set is used for evaluating the performance of the learned heuristic. For the training set, we used relatively small benchmarks, because our algorithm includes a process to obtain minimal abstractions and this task is too expensive to run for large programs. The validation set is used for choosing the hyper-parameter θ; we chose the one that leads the heuristic to the best performance on the validation set.

5.1 Effectiveness and Generality

We demonstrate the effectiveness and generality of our technique by comparing it with two state-of-the-art graph-based heuristics: Scaler [Li et al. 2018b] and Mahjong [Tan et al. 2017].

5.1.1 *Comparison with Scaler.*

Setting. Scaler is a context-sensitivity heuristic that works on the object allocation graph (OAG) [Li et al. 2018b]. From the OAG, it infers a policy to assign one of 2-object-sensitivity, 2-type-sensitivity, 1-type-sensitivity, and context-insensitivity to each method. We used the same pre-analysis of Scaler to obtain the OAG and let our technique produce a heuristic. We set $maxK$ in Section 3.2 to 3, where 0, 1, 2, and 3 correspond to context-insensitivity, 1-type-sensitivity, 2-type-sensitivity, and 2-object-sensitivity, respectively. Unlike Scaler, our heuristic assigns a

context for each heap allocation site. It poses 4^N possibilities where N denotes the number of allocation-sites in the program.

Although the primary objective in this evaluation is to compare with Scaler, we evaluated two more heuristics as well: Zipper [Li et al. 2018a] and Data [Jeong et al. 2017]. Zipper is another graph-based context-sensitivity heuristic that works on the precision flow graph (PFG). Data is not graph-based, but we include it because Data is currently the state-of-the-art data-driven pointer analysis algorithm (with hand-crafted features). In short, we compare the following pointer analysis algorithms:

- Scaler: A hand-crafted graph-based object-sensitivity heuristic for OAG [Li et al. 2018b]
- Graphick: Our learning-based graph-based object-sensitivity heuristic for OAG
- Zipper: A hand-crafted graph-based object-sensitivity heuristic for PFG [Li et al. 2018a]
- Data: A state-of-the-art learning-based object-sensitivity heuristic [Jeong et al. 2017]
- *2objH*: The 2-object-sensitivity with 1-context-sensitive heap (precision upper bound)
- *Insens*: The context-insensitive analysis (scalability upper bound)

We used three programs (luindex, lusearch, antlr) as the training set, one program (findbugs) as the validation set, and the remaining thirteen programs (pmd_s, chart, eclipse, jedit, briss, soot, jython, pmd_m, fop, bloat, JPC, checkstyle, xalan) as the test set. We chose findbugs as a validation program because it is a popular Java application and requires suitable heuristics to be analyzed cost-effectively. For example, *2objH* does not terminate on this program even after thousands of seconds or more.

Results. Table 1 and 2 present the performance of the context-sensitivity heuristics described above. The number in a parenthesis for graph-based heuristics (i.e. Graphick, Zipper, and Scaler) represents the sum of time spent on performing the pre-analysis (i.e. context-insensitive analysis) and running the heuristics on the graphs for extracting context abstractions.

The results show that our technique can automatically generate a cost-effective heuristic that performs as competitive as the state-of-the-art object-sensitivity heuristics. Compared to the baseline heuristic Scaler, which employs the same graph OAG, Graphick shows a better precision than Scaler with some losses in scalability for the test programs pmd_s, eclipse, and briss. For example, Graphick reports 101 less may-fail casts alarms than Scaler for the test program pmd_s while taking 216 more seconds. In addition, Graphick shows better performance in both precision and scalability than Scaler on the test programs (except pmd_m) in Table 2. For example, in jedit, Graphick produces 201 less alarms with 35% less analysis time. In comparison to Zipper, Graphick consistently outperforms in scalability. For example, Graphick successfully analyzed pmd_s, jedit, and briss with remarkably less costs when Zipper fails to analyze them within the time budget. In comparison to Data, the result shows that Graphick performs far better in precision. Although Data presents better scalability than Graphick, it produces more than 92 alarms for the test programs pmd_s, eclipse, jedit and briss. Compared to *2objH*, Graphick shows better performance in scalability for the majority of test programs which *2objH* fails to analyze within the given time budget (3 hours).

5.1.2 *Comparison with Mahjong.*

Setting. Mahjong is a graph-based heap abstraction heuristic that works on the field points-to graph (FPG) [Tan et al. 2017]. From the FPG, which is obtained by running a context-insensitive pre-analysis, Mahjong infers a policy that determines whether to merge objects allocated in different allocation sites. We used the same pre-analysis to obtain the FPG and let our technique produce a heap abstraction heuristic. Unlike Mahjong, our heap abstraction heuristic (i.e. Graphick) assigns '*type*' (type-based heap abstraction) or '*alloc*' (allocation-site-based heap abstraction) to each heap

Table 1. Performance of the context-sensitivity heuristics against benchmarks. For all metrics, the lower is the better. For precision metric, we use the number of may-fail casts(#may-fail casts) and polymorphic call sites (#poly-call sites) whose targets are not uniquely determined by each pointer analysis. For scalability metric, we use analysis time, and the number in a parenthesis presents the sum of time spent during pre-analysis process. #call-graph-edges for the training and validation programs are omitted due to the lack of space.

			Graphick	Scaler	Zipper	Data	*2objH*	*Insens*
Training programs	luindex	analysis time (s)	22(+22)	36(+17)	33(+17)	19	36	15
		#may-fail casts	297	297	310	341	297	734
		#poly-call sites	682	675	677	702	675	940
	lusearch	analysis time (s)	24(+21)	63(+15)	62(+17)	19	66	15
		#may-fail casts	299	299	305	347	299	844
		#poly-call sites	858	850	853	883	850	1,133
	antlr	analysis time (s)	50(+94)	51(+24)	84(+26)	33	109	24
		#may-fail casts	409	412	420	513	409	918
		#poly-call sites	1,495	1,488	1,488	1,517	1,487	1,729
Test programs	pmd_s	analysis time (s)	710(+92)	494(+49)	>10,800	117	>10,800	48
		#may-fail casts	2,075	2,176	-	2,145	-	2,948
		#poly-call sites	3,507	3,536	-	3,647	-	4,183
		#call-graph-edges	92,589	92,775	-	94,328	-	104,457
	chart	analysis time (s)	63(+73)	184(+48)	113(+56)	35	196	48
		#may-fail casts	998	976	888	974	883	1,810
		#poly-call sites	1,392	1,402	1,379	1,435	1,378	1,852
		#call-graph-edges	52,544	53,198	52,377	52,647	52,374	63,453
	eclipse	analysis time (s)	1,395(+103)	652(+92)	9,701(+114)	159	>10,800	91
		#may-fail casts	2,989	3,211	2,897	3,178	-	4,190
		#poly-call sites	8,418	8,486	8,390	8,627	-	9,197
		#call-graph-edges	144,873	145,953	143,727	146,512	-	161,222
	jedit	analysis time (s)	845(+90)	1,377(+79)	>10,800	137	>10,800	78
		#may-fail casts	2,196	2,397	-	2,298	-	3,398
		#poly-call sites	3,917	4,012	-	4,091	-	4,769
		#call-graph-edges	98,401	99,536	-	99,697	-	120,309
	briss	analysis time (s)	2,368(+169)	907(+151)	>10,800	499	>10,800	149
		#may-fail casts	3,065	3,428	-	3,162	-	4,904
		#poly-call sites	5,099	5,323	-	5,291	-	6,297
		#call-graph-edges	150,351	152,761	-	151,861	-	176,785
	soot	analysis time (s)	>10,800	883(+727)	>10,800	>10,800	>10,800	698
		#may-fail casts	-	10,549	-	-	-	16,570
		#poly-call sites	-	14,822	-	-	-	16,532
		#call-graph-edges	-	374,877	-	-	-	415,476
	jython	analysis time (s)	>10,800	314(+96)	>10,800	425	>10,800	73
		#may-fail casts	-	1,852	-	1,773	-	2,234
		#poly-call sites	-	2,500	-	2,481	-	2,778
		#call-graph-edges	-	107,410	-	106,837	-	114,856
Valid	findbugs	analysis time (s)	305(+58)	191(+36)	1,399(+43)	59	2,458	35
		#may-fail casts	1,436	1,452	1,412	1,663	1,409	2,508
		#poly-call sites	2,188	2,195	2,182	2,220	2,182	2,925

Table 2. Performance comparison among various context sensitivity heuristics against the left six benchmarks. All the notations are the same with Table 1.

			Graphick	Scaler	Zipper	Data	*2objH*	*Insens*
Test programs	pmd_m	analysis time (s)	43(+67)	55(+44)	57(+78)	30	67	23
		#may-fail casts	288	287	300	327	287	679
		#poly-call sites	643	636	638	667	636	885
		#call-graph-edges	27,074	27,052	27,056	27,147	27,052	30,328
	fop	analysis time (s)	212(+88)	341(+54)	533(+71)	74	949	53
		#may-fail casts	1,568	1,732	1,449	1,600	1,446	2,458
		#poly-call sites	2,876	2,945	2,848	3,009	2,844	3,585
		#call-graph-edges	71,612	72,556	71,418	72,113	71,408	84,330
	bloat	analysis time (s)	216(+30)	290(+21)	2402(+23)	44	2,422	20
		#may-fail casts	1,215	1,222	1,205	1,288	1,193	1,924
		#poly-call sites	1,458	1,465	1,429	1,496	1,427	2,014
		#call-graph-edges	53,641	53,867	53,147	54,059	53,143	61,150
	JPC	analysis time (s)	118(+69)	274(+38)	266(+55)	45	398	37
		#may-fail casts	1,427	1,552	1,343	1,472	1,345	2,261
		#poly-call sites	4,210	4,228	4,187	4,322	4,186	4,924
		#call-graph-edges	79,912	80,098	79,787	80,208	79,783	94,569
	checkstyle	analysis time (s)	133(+70)	264(+45)	396(+52)	69	1,693	44
		#may-fail casts	600	625	590	644	581	1,114
		#poly-call sites	1,052	1,038	1,040	1,089	1,035	1,444
		#call-graph-edges	9,516	9,514	48,830	48,996	48,809	57,490
	xalan	analysis time (s)	226(+64)	539(+38)	119(+45)	44	881	37
		#may-fail casts	567	579	556	604	533	1,182
		#poly-call sites	1,533	1,523	1,533	1,583	1,522	1,898
		#call-graph-edges	45,269	44,887	9,125	45,549	44,871	5,1302

allocation-site which poses 2^N possibilities where N denotes the number of allocation-sites in the program. We compare the following four analyses:

- Mahjong: The state-of-the-art graph-based heap abstraction heuristic [Tan et al. 2017]
- Graphick: Our learning-based graph-based heap abstraction heuristic
- *Alloc-Based*: The uniform allocation-site-based heap abstraction (precision upper bound)
- *Type-Based*: The uniform type-based heap abstraction (scalability upper bound)

Following Mahjong [Tan et al. 2017], all analyses above use 3-object-sensitivity with 2-context-sensitive heap.

For this evaluation, we used the same benchmark programs in the section 5.1.1. We used four programs (luindex, lusearch, antlr, pmd_m) as the training set and twelve programs (fop, chart, bloat, xalan, JPC, checkstype, eclipse, pmd_s, jecit, briss, soot, jython) as the test set. We also used findbugs as a validation program.

Results. Table 3 and 4 show that our technique can produce a competitive graph-based heap abstraction heuristic from the FPG. In comparison with Mahjong, Graphick shows a far better scalability while losing precision a bit. Mahjong produced the same number of may-fail-casts with the most precise one, *Alloc-Based*, but it was unable to analyze large programs like chart and bloat within the time budget (3 hours). Although Graphick produced more alarms (103 at most)

Table 3. Performance of the heap abstraction heuristics against benchmarks. The notions are the same with those in Table 1.

			Graphick	Mahjong	*Alloc-Based*	*Type-Based*
Training programs	luindex	analysis time(s)	23(+90)	42(+21)	5,475	19
		#may-fail casts	358	358	358	795
		#poly-call sites	928	918	915	1,128
		#call-graph-edges	33,450	33,365	33,356	37,898
	lusearch	analysis time(s)	21(+92)	43(+19)	>10,800	19
		#may-fail casts	372	372	-	884
		#poly-call sites	1,127	1,116	-	1,331
		#call-graph-edges	36,298	36,237	-	41,211
	antlr	analysis time(s)	31(+101)	48(+33)	5,241	26
		#may-fail casts	463	463	463	1,002
		#poly-call sites	1,630	1,626	1,623	1,836
		#call-graph-edges	51,058	51,043	51,035	55,745
	pmd_m	analysis time(s)	44(+137)	88(+34)	9,146	42
		#may-fail casts	871	871	871	1,418
		#poly-call sites	1,142	1,133	1,130	1,388
		#call graph edges	44,094	4,4016	44,004	50,365
Test programs	fop	analysis time(s)	30(+117)	50(+26)	5,475	33
		#may-fail casts	376	375	375	779
		#poly-call sites	830	817	814	1,034
		#call graph edges	34,259	34,192	34,184	38,629
	chart	analysis time(s)	436(+350)	>10,800	>10,800	199
		#may-fail casts	1,331	-	-	2,299
		#poly-call sites	2,078	-	-	2,363
		#call graph edges	72,746	-	-	82,952
	bloat	analysis time(s)	376(+121)	>10,800	>10,800	26
		#may-fail casts	1,247	-	-	1,926
		#poly-call sites	1,593	-	-	1,793
		#call graph edges	56,535	-	-	64,220
	xalan	analysis time(s)	489(+162)	795(+29)	>10,800	59
		#may-fail casts	539	535	-	1,093
		#poly-call sites	1,601	1,591	-	1,876
		#call graph edges	46,026	45,950	-	51,761
	JPC	analysis time(s)	1,730(+366)	3,309(+47)	>10,800	524
		#may-fail casts	1,300	1,226	-	2,007
		#poly-call sites	4,211	4,139	-	4,646
		#call graph edges	79,864	79,370	-	91,248
	checkstyle	analysis time(s)	1,333(+563)	2,346(+53)	>10,800	48
		#may-fail casts	1,085	1,022	-	1,749
		#poly-call sites	2,202	2,168	-	2,489
		#call-graph-edges	66,321	65,943	-	77,962
Valid	findbugs	analysis time(s)	96(+363)	273(+70)	>10,800	92
		#may-fail casts	1,774	1,671	-	3,089
		#poly call sites	3,576	3,534	-	4,281

Table 4. Performance comparison between the heap abstraction heuristics against the left benchmarks.

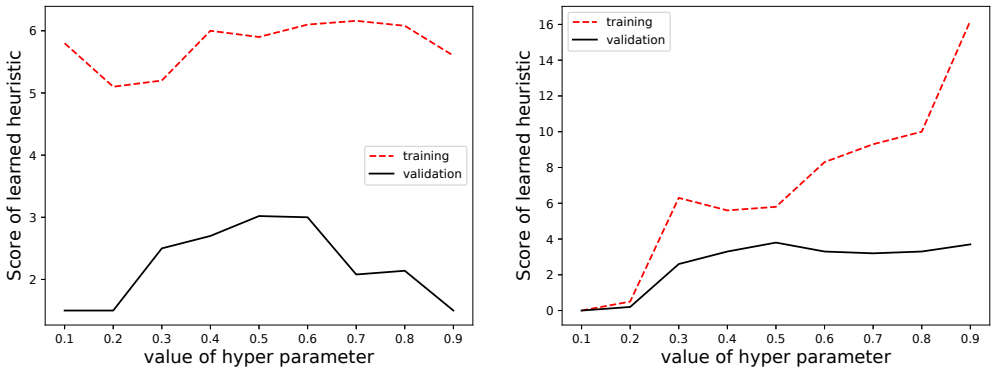
			Graphick	Mahjong	*Alloc-Based*	*Type-Based*
Test programs	eclipse	analysis time (s)	>10,800	>10,800	>10,800	222
		#may-fail casts	-	-	-	4,852
		#poly-call sites	-	-	-	10,177
		#call-graph-edges	-	-	-	182,000
	pmd_s	analysis time (s)	>10,800	>10,800	>10,800	4,317
		#may-fail casts	-	-	-	2,941
		#poly-call sites	-	-	-	4,124
		#call-graph-edges	-	-	-	106,490
	jedit	analysis time (s)	454(+242)	1,392(+40)	8,001	245
		#may-fail casts	1,143	1,094	1,094	1,786
		#poly-call sites	1,732	1,688	1,684	2,064
		#call-graph-edges	55,476	55,156	55,145	64,825
	briss	analysis time (s)	>10,800	>10,800	>10,800	>10,800
		#may-fail casts	-	-	-	-
		#poly-call sites	-	-	-	-
		#call-graph-edges	-	-	-	-
	soot	analysis time (s)	>10,800	>10,800	>10,800	7,741
		#may-fail casts	-	-	-	15,885
		#poly-call sites	-	-	-	14,617
		#call-graph-edges	-	-	-	359,358
	jython	analysis time (s)	>10,800	>10,800	>10,800	187
		#may-fail casts	-	-	-	1,211
		#poly-call sites	-	-	-	1,487
		#call-graph-edges	-	-	-	50,544

than Mahjong, it successfully analyzed programs (i.e. chart and bloat) which Mahjong failed to analyze. Currently, the overhead, the time taken by extracting an abstraction from the FPG, of our heuristic is bigger than Mahjong because Mahjong designed an efficient algorithm to produce an abstraction from FPG while ours is not optimized to minimize it. The results, however, still demonstrate that Graphick is competitive and has a strength in scalability compared to the state-of-the-art technique as it successfully analyzed the large programs, chart and bloat, which Mahjong cannot handle.

5.2 Learning Algorithm

Learning Cost . To learn a context-sensitivity heuristic, our learning algorithm took 169 hours in total, where 144 hours are for getting minimal abstractions over the training programs and 25 hours for generating features. To learn a heap abstraction heuristic, the algorithm took 107 hours, where 72 hours are for minimal abstraction generation and 35 hours for feature generation. We note that, although the learning algorithm is expensive, it saves more expensive human costs by automating the manual process of designing analysis heuristics that would take weeks or months.

Choosing Hyper-Parameter θ. Through our evaluation, we observed that the value of the hyper-parameter θ plays an important role in the performance of the learned heuristic. Figure 5 depicts how performance of learned heuristics changes over the values of θ. The X-axis presents



(a) Object-sensitivity heuristic (b) Heap abstraction heuristic

Fig. 5. How score of learned heuristic changes over the value of θ.

the value of θ set to learn each heuristic, and the Y-axis presents scores that we measured for performance of each heuristic $\mathcal{H}_\theta$ according to $\frac{\sum_{P \in \mathrm{P}} \texttt{proved}(F_P(\mathcal{H}_\theta(G)))}{\sum_{P \in \mathrm{P}} \texttt{cost}(F_P(\mathcal{H}_\theta(G)))}$ where `cost` denotes analysis time. This score function presents the number of queries proved per second; thereby, more precise and scalable the analysis, higher the score. The red dotted and black solid lines present how the scores change over the training programs P and the validation program, respectively. For the training programs, the score of the learned heuristic increases as the higher θ is given because the heuristic becomes more fitted to the training programs.[2] In our evaluation, both learned heuristics, however, perform the best on the validation program when θ is 0.5; thus, Graphick in Table 1 and Table 3 corresponds to $\mathcal{H}_{0.5}$.

5.3 Learned Insights

The learning algorithm generated 197 features in total for the object-sensitivity heuristic (68 features for 2-object-sensitivity, 29 for 2-type-sensitivity, and 100 features for 1-type-sensitivity). It generated 96 features for the heap abstraction heuristic.

Top-5 Features. Figure 6 describes the most informative features generated by our technique for each abstraction degree and their concretization in the given graphs. The second column Top 5 Features, in decreasing order of portion, presents top 5 features which have the greatest number of precision-critical nodes satisfying the features with scores above 0.5. For example, the first feature in 1type contains 47% nodes of the total precision-critical nodes which are to be applied 1-type-sensitivity, and has a score of 0.57. If a feature is too general (e.g., $(\epsilon, ([0, \infty], [0, \infty]), \epsilon)$), it is excluded even with a large portion (e.g., 100%) because its score is under 0.5. Similarly, if a feature is too specific, it is also excluded because it includes a small number of precision-critical nodes even with a good score. For the features, the gray colored abstract nodes correspond to the target one $\hat{n}$ in each feature (e.g., $(\langle \hat{p}_0, \hat{p}_1, \ldots, \hat{p}_q \rangle, \hat{n}, \langle \hat{s}_0, \hat{s}_1, \ldots, \hat{s}_r \rangle) \in \mathit{Feature}$). Other nodes are predecessors or successors of the target abstract nodes (e.g., $\hat{p}_0$, $\hat{s}_0$, and $\hat{s}_1$). For each feature, we show the number

[2]It learned a bit bad heuristic when θ is 0.9 in object-sensitivity heuristic because it is difficult to generate specific features that satisfy such high precision constraints; it eventually generates a general feature that include lots of nodes.

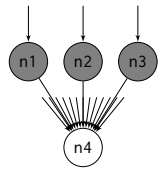
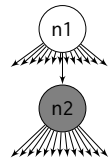
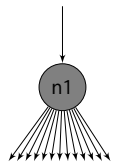
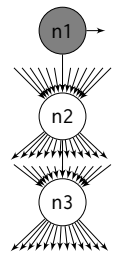
		Top 5 Features	portion	score	Concretization (Top 1)
Object-Sensitivity Heuristic	1type	[0,∞],[61,∞] → [46,∞],[0,∞]	47%	0.57	
		[0,∞],[0,∞] → [0,∞],[117,∞]	36%	0.63	
		[0,∞],[100,∞] → [0,∞],[29,∞]	35%	0.55	
		[0,∞],[100,∞] → [0,∞],[0,∞] → [0,∞],[36,43]	29%	0.71	
		[0,∞],[109,∞] → [0,∞],[0,∞] → [171,∞],[0,∞]	25%	0.57	
	2type	[0,∞],[36,39] → [0,∞],[73,75]	9%	0.66	
		[105,155],[0,∞]	9%	1	
		[0,∞],[0,61] → [60,76],[0,61] → [0,22],[0,∞]	9%	0.5	
		[0,∞],[29,61] → [171,228],[0,∞] → [0,46],[0,∞]	4%	1	
		[84,91],[0,∞]	4%	0.5	
	2obj	[0,∞],[53,61]	9%	0.53	
		[0,∞],[24,25]	6%	0.53	
		[0,∞],[0,7] → [9,11],[0,∞] → [76,∞],[0,∞]	2%	0.82	
		[0,∞],[43,∞] → [0,∞],[0,14] → [22,24],[0,∞]	1%	0.63	
		[0,∞],[145,147] → [0,∞],[0,∞] → [0,46],[0,∞]	1%	0.69	
Heap Abstraction Heuristic		[0,∞],[0,3] → [48,∞],[0,∞] → [0,∞],[140,∞]	35%	0.61	
		[0,12],[0,3] → [0,97],[0,∞] → [0,∞],[140,∞]	33%	0.53	
		[38,∞],[0,62] → [0,∞],[236,∞]	27%	0.59	
		[0,48],[0,62] → [72,84],[0,∞]	24%	0.53	
		[0,24],[0,∞] → [21,∞],[140,∞] → [0,97],[0,∞]	22%	0.53	

Fig. 6. Top-5 features learned by our technique, and concrete nodes implied by the top-1 feature. Gray colored abstract nodes in the features correspond to the target nodes and others are predecessors or successors. Gray colored nodes in the column Concretization are precision-critical nodes which are selected by the first features; other nodes are predecessors or successors that make the gray colored nodes satisfy the features.

Table 5. Performance of our manually-designed graph-based heap abstraction heuristic for FPG

benchmarks		alarms	time(s)	benchmarks		alarms	time(s)
luindex	*heuristic*	374	29(+29)	lusearch	*heuristic*	388	33(+31)
	Alloc-Based	358	5,475		*Alloc-Based*	-	>10,800
antlr	*heuristic*	478	44(+47)	pmd_m	*heuristic*	886	83(+65)
	Alloc-Based	463	5,241		*Alloc-Based*	871	9,146
fop	*heuristic*	391	41(+46)	xalan	*heuristic*	548	841(+85)
	Alloc-Based	-	>10,800		*Alloc-Based*	-	>10,800

of satisfying nodes over the total precision-critical nodes in the given graphs (portion) and the scores (score). The right most column, Concretization, illustrates the visualized concretization for each first feature in Top 5 Features column, where the gray colored nodes correspond to the target abstract nodes of the first feature. For space reasons, we draw each node to have at most 13 incoming and outgoing edges although it can have more than 13 edges.

Insights. The generated features during the learning process provide hints on designing analysis heuristics from the graphs. For example, we investigated the features of the heap abstraction heuristic in Figure 6 and found two commonalities in them. First, the features have the form of $(\epsilon, \hat{n}, \hat{s})$ where $\hat{s}$ is not ϵ, which implies that we should consider successors more than predecessors when designing heap abstraction heuristics from points-to graph. The second commonality is that $\hat{s}$ or $\hat{n}$ tends to include an abstract node $\widehat{Node}$ that presents nodes with lots of outgoing edges, i.e., $\widehat{Node} = (itv, [b, \infty])$ where the number b is about 3% of the total nodes in a graph of a training program. From these observations, we manually designed a graph-based heap abstraction heuristic which assigns allocation-site based heap abstraction to the target nodes if at least 3% of the total nodes in FPG belong to either the target node or its successor nodes (i.e. $\mathcal{H} = \langle\{(\epsilon, ([0,\infty], [b',\infty]), \epsilon), (\epsilon, \top, \langle([0,\infty], [b',\infty])\rangle), (\epsilon, \top, \langle\top, ([0,\infty], [b',\infty])\rangle), \dots\}\rangle$ where $\top$ equals to the most general one $([0,\infty], [0,\infty])$ and b' is 3% of the total nodes in the given graph). Otherwise, the heuristic assigns type-based heap abstraction to the others. Table 5 demonstrates the performance of the manually-crafted heuristic. In comparison to *Alloc-Based*, it reduces about 99% of analysis cost while producing only 2% more alarms.

Intuitively, the nodes with lots of successors in FPG should be analyzed precisely because merging the objects with others would produce lots of spurious analysis results. For example, if there exists an object with lots of field objects which we want to merge with another one with a few field objects, it eventually produces lots of spurious results stating that the both heaps can have lots of field objects. Such insight is related with that of Mahjong which merges the objects if their successors have the same type; statistically, if an object has lots of successors, there hardly exist the other objects with exactly the same types of successors. Surprisingly, it is easy to find such insight through the features generated by our technique. Note that this insight is general as it is not dependent to Java programs. For example, when analyzing a C program, it is a required task not to merge such heaps with others as it would produce lots of spurious results.

Interestingly, Figure 6 also shows the difference between the statistically-learned insight behind Graphick and the logical insight behind Scaler in deciding which nodes to analyze more precisely. Based on the logical insight, Scaler relies heavily on the number of incoming edges as that number in the object allocation graph indicates how many contexts will be constructed in object sensitivity. Graphick, however, treats the number of neighbor nodes' outgoing edges more

Table 6. Performance comparison between conventional 2-hybrid-context-sensitivity (*S2objH*) and 2-hybrid-context-sensitivity with our learned heuristic for 2-object-sensitivity (Graphick).

		pmd_m	chart	eclipse	xalan	fop	bloat
Graphick	#may-fail casts	220	867	2,880	479	1,408	1,147
	analysis time (s)	45+(78)	83(+76)	1,426(+105)	185(+57)	245(+85)	215(+30)
S2objH	#may-fail casts	220	757	-	447	1,295	1,125
	analysis time (s)	42	195	>10,800	428	818	2,238

importantly, as shown in Figure 6. Such differences result in the performance gap between the two object-sensitivity heuristics.

Generality of learned heuristic. We found the learned heuristic for object sensitivity is general to the hybrid-context sensitivity [Kastrinis and Smaragdakis 2013]. Table 6 presents the performance of the conventional 2-hybrid-context sensitivity (*S2objH*) and 2-hybrid-context sensitivity with the learned heuristic (Graphick) used in Section 5.1.1. The table shows that Graphick is also cost-effective compared to *S2objH*. For example, on a test program bloat, Graphick produces only 22 more alarms while reducing about 90% of analysis costs.

5.4 Performance Variations on Different Training Datasets

We constructed a benchmark suite with the programs from the DaCapo suite, and used 3~4 small programs (i.e. luindex, lusearch, antlr, and pmd_m) as our training set. In this subsection, we evaluate Graphick on different combinations of training data to see how its performance is affected by the number of training programs.

We found that the amount of training data is overall critical, and using four small programs as a training set can produce competitive heap abstraction heuristics cost-effectively. Table 7 presents the performance and scores (i.e. $\frac{\text{\#proven casts}}{\text{analysis time (s)}}$) of each heuristic learned with various combinations of training programs (i.e., {luindex}, {luindex, lusearch}, {luindex, lusearch, antlr}, and {luindex, lusearch, antlr, pmd}) and an ideal heuristic (*ideal*) against the validation program findbugs. For the ideal heuristic (*ideal*), we assume that it has the precision of Mahjong and the scalability of *Type-Based* since they are the most precise and the most scalable respectively in our space of heap abstraction heuristic. The second row, analysis time (s), in table 7 indicates the amount of time each heuristic took to successfully analyze the validation program, and the third row, #proven casts, presents the number of castings proved to be safe; thereby, the more precise the analysis, the greater the number of proven casts. As shown in Table 7, the score increases with respect to the size of training set. The score of {luindex, lusearch, antlr, pmd} (i.e. 13.6) is nearly the same with that of the ideal heuristic (i.e. 15.4) in our evaluation. It implies that using four programs as a training set is sufficient to produce cost-effective heap abstraction heuristics.

Using four programs as training programs could produce cost-effective heuristics because, even though our training programs are the smallest among the total benchmark programs, they still provide sufficient learning data for our approach. First, the DaCapo suite itself is a collection of realistic programs. DaCapo has been carefully designed to include various behaviors and complex codes [Blackburn et al. 2006]. For example, even the smallest program (i.e. lusearch) in Dacapo has more methods than the largest one in the SPEC benchmark [SPECjvm98 1999]. Secondly, when training heuristics in our approach, what matters is the number of allocation-sites, not the number of programs; the learning algorithm of Graphick treats individual allocation-sites as labelled data.

Table 7. Performance comparison among heuristics learned from various combinations of training sets (i.e. {luindex}, {luindex, lusearch}, {luindex, lusearch, antlr}, and {luindex, lusearch, antlr, pmd}) and an ideal heuristic (*ideal*) against the validation program findbugs. #proven casts presents the number of casts proved to be safe; a more precise analysis produces a larger number of #proven casts. The row score presents the quality of the heuristics computed by $\frac{\text{\#proven casts}}{\text{analysis time (s)}}$.

	{luindex}	{luindex, lusearch}	{luindex, lusearch antlr}	{luindex, lusearch antlr, pmd}	*ideal*
analysis time (s)	4,090(+185)	411(+107)	153(+226)	96(+363)	92
#proven casts	672	1,321	1,289	1,315	1418
score	0.16	3.2	8.4	13.6	15.4

Our training programs provide sufficient training data to learn cost-effective heuristics in this sense. More precisely, the smallest program (lusearch) has 4,752 allocation-sites, and the remaining three training programs (lusearch, antlr, pmd_m) provide 14,068 unique allocation-sites in total; we have a total of 18,820 allocation-sites for training data.

In practice, we recommend a user to choose programs with less than 400 classes as training programs, for which we found Grahpick typically works well. Although limited, our experience shows that a collection of such programs can provide useful training data.

5.5 Limitations and Discussion

Graphick has several limitations. One major limitation is that the heuristics produced by our approach may not be generalized if the setting in training steps is substantially different from that used in evaluation in terms of analyzer F, target client, and $maxK$. More specifically, the learned heuristics by Graphick are dependent on the training data, the analyzer F used, and the target client (e.g., may-fail casts). For example, the precision on the number of may-fail casts can be unsatisfactory if we train the heuristics with the number of poly-call sites as the target client. The context-length $maxK$ for the main analysis is also limited to the one used in the training phase. Besides those generality issues, the training process of Graphick is time-consuming (i.e. it took 200 hours in our evaluation).

Despite those limitations, we believe Graphick can be useful in practice. First of all, note that in this paper we showed the effectiveness of Graphick in a realistic (yet particular) setting, where we used a real-world pointer analysis framework and benchmarks. In particular, we do not believe the expensive training phase of Graphick is a serious limitation in practice, because it is not only fully automatic but also rather cheap compared to the much more expensive process of handcrafting analysis heuristics or features by human experts. The learned heuristic is dependent on the training data but, as we showed in this paper (Section 5.4), using a small number of real programs is likely to provide a sufficient amount of actual training data (e.g., allocation sites) in practice. Also, the selection of training programs did not require careful engineering efforts in our case. The context length $maxK$ is rather limited (2-object-sensitivity), but 2-object-sensitive pointer analysis is generally considered to be highly precise in practice [Li et al. 2018a].

For the issue on target clients, we showed that training heuristics using the may-fail-cast client generalizes well for the three clients (may-fail-casts, poly-call-sites, and call-graph-edges). When the downstream clients are substantially different, however, one solution is to choose the target clients as general as possible in the training process. For example, we can use the size of context-insensitive variable points-to sets instead of the number of may-fail-casts as the context-insensitive

variable points-to set is one of the most general clients that affect the others. The clients we used in our evaluation (i.e. may-fail-casts, poly-call-sites, and call-graph-edges) are computed based on the context-insensitive variable points-to set and therefore minimizing it would likely minimize other clients too.

6 RELATED WORK

In this section, we discuss the prior works related to ours.

Heuristics for Static Analysis. Designing heuristics for precise and scalable static analysis has been an active research area. For example, Smaragdakis et al. [2014] proposed a context-sensitivity heuristic that runs pre-analysis (e.g., context-insensitive analysis) to identify scalability-detrimental method calls if context-sensitive analysis is applied; it analyzes those methods context-insensitively to obtain tractable scalability while sacrificing precision a bit. Oh et al. [2014] presented the idea of impact pre-analysis, which first estimates the impact of applying context sensitivity with a fully context-sensitive yet coarse pre-analysis and then performs selective context sensitivity during the main analysis. Hassanshahi et al. [2017] aimed to find a parameter which determines context depths for each heap. They performed context-insensitive analysis as a pre-analysis, and from the analysis results, determined the heap context depths for each object to achieve reasonable scalability without losing too much precision. Kastrinis and Smaragdakis [2013] introduced a hybrid context-sensitivity heuristic that applies object-sensitivity for the virtual calls while applying call-site-sensitivity to the static calls. Xu and Rountev [2008] proposed a technique to identify the equivalence classes; it merges the contexts in the same class in order to improve the scalability without any precision loss. Recently, to design cost-effective analysis policies, graph-based heuristics have arisen as a trending technique [Li et al. 2018a,b; Lu and Xue 2019; Tan et al. 2016, 2017]; our work lies in this line of research and aims to generate such graph-based heuristics automatically.

Data-driven Static Analysis. Our work also belongs to the family of techniques known as data-driven static analysis [Cha et al. 2016, 2018; He et al. 2020; Heo et al. 2017; Jeong et al. 2017; Oh et al. 2015]. Data-driven static analysis leverages machine learning to produce favorable program analysis heuristics automatically. Oh et al. [2015] proposed a data-driven technique based on Bayesian optimization to learn flow- and context-sensitivity heuristics. They designed features for variables and functions in C programs to learn flow- and context-sensitivity heuristics which are presented as linear combinations of the features. Later, the linear-model approach was extended to capture disjunctive program properties [Jeon et al. 2019; Jeong et al. 2017] Jeon et al. [2018] introduced an approach, called data-driven context tunneling, which constructs contexts with the most important k context elements instead of using the most recent k context elements as the conventional k context abstraction does. To learn context tunneling heuristics, they designed features for methods of Java programs to present which method calls require context tunneling for better performance in both precision and scalability. Heo et al. [2016] proposed a supervised learning algorithm to learn variable clustering strategy in the Octagon domain where the learned heuristics determine whether to keep relation between variables during analysis. He et al. [2020] introduced a data-driven approach Lait that learns neural policies for removing substantially redundant constraints that need not be computed in numeric program analysis. Singh et al. [2018] leveraged reinforcement learning to speed up numeric analysis with the Polyhedra domain. The prior works above require manually designed features to learn suitable heuristics. By contrast, our technique proposes to use a feature language to reduce the burden of manual effort on designing features.

Closely related to our work, Chae et al. [2017] also automatically generated features for data-driven static analysis. Given programs, it runs a program reducer to convert the programs into small feature programs which only maintain the query-related program components, and generates

features for data-driven static analysis from data-flow graphs obtained from the feature programs. Not to mention that the technique is specialized for C programs, it is hardly applicable to learning context-sensitivity heuristics because reducing programs spanning multiple procedures into reasonably small feature programs while maintaining the query-related components is challenging [Chae et al. 2017] . In this paper, we present a new technique based on a feature language, which does not have such a limitation and is effectively applicable to context-sensitive analysis for Java.

Finding Optimal Abstractions. Various techniques have been proposed to find optimal abstractions efficiently that precisely analyze the query-related program parts only [Liang et al. 2011; Zhang et al. 2014, 2013]. Our work is different from them as we aim for good-enough abstractions with much smaller overheads. Zhang et al. [2013] suggested a counterexample-guided abstraction refinement technique that iteratively refines an abstraction toward a desirable one in dataflow analysis. This approach was improved further by Zhang et al. [2014], which can find desirable abstractions in parametric program analysis written in Datalog. Liang et al. [2011] proposed an efficient algorithm to find minimal abstractions that precisely analyze the components related to queries only. In our work, we improved the algorithm by Liang et al. [2011] in terms of the size of search space (Section 4.4).

7 CONCLUSION

In this paper, we presented a technique, Graphick, that automatically learns graph-based analysis heuristics. Recently, designing heuristics on the graph representations of programs has been arisen as a promising approach in pointer analysis. Graphick aims to automate the designing process using a feature language and a learning algorithm. To demonstrate the performance of our approach, we implemented it on top of the Doop pointer analysis framework for Java. The experimental results show that Graphick successfully produces high-quality analysis heuristics that are as competitive as the existing state-of-the-art heuristics designed manually by analysis experts. We hope our work facilitates the recent development of graph-based heuristics for pointer analysis.

ACKNOWLEDGMENTS

We thank Donghoon Jeon for helpful comments on Algorithm 2. This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-51. This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337, (SW STAR LAB) Research on Highly-Practical Automated Software Repair and No.2017-0-00184, Self-Learning Cyber Immune Technology Development).

REFERENCES

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. 2005. Improving Software Security with a C Pointer Analysis. In *Proceedings of the 27th International Conference on Software Engineering* (St. Louis, MO, USA) *(ICSE '05)*. ACM, New York, NY, USA, 332–341. https://doi.org/10.1145/1062455.1062520

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 (Portland, Oregon, USA) *(OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2015. Selective Control-flow Abstraction via Jumping. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. ACM, New York, NY, USA, 163–182. https://doi.org/10.1145/2814270.2814293

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. *SIGPLAN Not.* 44, 10 (Oct. 2009), 243–262. https://doi.org/10.1145/1639949.1640108

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

Sooyoung Cha, Sehun Jeong, and Hakjoo Oh. 2018. A scalable learning algorithm for data-driven program analysis. *Information and Software Technology* 104 (2018), 1 – 13. https://doi.org/10.1016/j.infsof.2018.07.002

Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. 2017. Automatically Generating Features for Learning Program Analysis Heuristics for C-like Languages. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 101 (Oct. 2017), 25 pages. https://doi.org/10.1145/3133925

Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective Typestate Verification in the Presence of Aliasing. *ACM Trans. Softw. Eng. Methodol.* 17, 2, Article 9 (May 2008), 34 pages. https://doi.org/10.1145/1348250.1348255

Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe Memory-leak Fixing for C Programs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 459–470. http://dl.acm.org/citation.cfm?id=2818754.2818812

Neville Grech and Yannis Smaragdakis. 2017. P/Taint: Unified Points-to and Taint Analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 102 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133926

Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 2017. An Efficient Tunable Selective Points-to Analysis for Large Codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis* (Barcelona, Spain) *(SOAP 2017)*. ACM, New York, NY, USA, 13–18. https://doi.org/10.1145/3088515.3088519

Jingxuan He, Gagandeep Singh, Markus Püschel, and Martin Vechev. 2020. Learning Fast and Precise Numerical Analysis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1112–1127. https://doi.org/10.1145/3385412.3386016

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* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 519–529. https://doi.org/10.1109/ICSE.2017.54

Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. 2020. SAVER: Scalable, Precise, and Safe Memory-Error Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 271–283. https://doi.org/10.1145/3377811.3380323

Minseok Jeon, Sehun Jeong, Sungdeok Cha, and Hakjoo Oh. 2019. A Machine-Learning Algorithm with Disjunctive Model for Data-Driven Program Analysis. *ACM Trans. Program. Lang. Syst.* 41, 2, Article 13 (June 2019), 41 pages. https://doi.org/10.1145/3293607

Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and Scalable Points-to Analysis via Data-driven Context Tunneling. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 140 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276510

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 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133924

Vini Kanvar and Uday P. Khedker. 2016. Heap Abstractions for Static Analysis. *ACM Comput. Surv.* 49, 2 (2016), 29:1–29:47. https://doi.org/10.1145/2931098

Timotej Kapus and Cristian Cadar. 2019. A Segmented Memory Model for Symbolic Execution. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 774–784. https://doi.org/10.1145/3338906.3338936

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

Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. MemFix: Static Analysis-Based Repair of Memory Deallocation Errors for C. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium*

on the Foundations of Software Engineering (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 95–106. https://doi.org/10.1145/3236024.3236079

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018a. Precision-guided Context Sensitivity for Pointer Analysis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 141 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276511

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018b. Scalability-first Pointer Analysis with Self-tuning Context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/3236024.3236041

Percy Liang and Mayur Naik. 2011. Scaling Abstraction Refinement via Pruning. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 590–601. https://doi.org/10.1145/1993498.1993567

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* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 31–42. https://doi.org/10.1145/1926385.1926391

V. Benjamin Livshits and Monica S. Lam. 2003. Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Helsinki, Finland) *(ESEC/FSE-11)*. ACM, New York, NY, USA, 317–326. https://doi.org/10.1145/940071.940114

Jingbo Lu and Jingling Xue. 2019. Precision-Preserving yet Fast Object-Sensitive Pointer Analysis with Partial Context Sensitivity. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 148 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360574

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2002. Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Roma, Italy) *(ISSTA '02)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/566172.566174

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

Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. ACM, New York, NY, USA, 308–319. https://doi.org/10.1145/1133981.1134018

Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. 2009. Effective Static Deadlock Detection. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 386–396. https://doi.org/10.1109/ICSE.2009.5070538

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* (Edinburgh, United Kingdom) *(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* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. ACM, New York, NY, USA, 572–588. https://doi.org/10.1145/2814270.2814309

Gagandeep Singh, Markus Püschel, and Martin Vechev. 2018. Fast Numerical Program Analysis with Reinforcement Learning. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 211–229.

Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Foundations and Trends in Programming Languages* 2, 1 (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* (Austin, Texas, USA) *(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* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 485–495. https://doi.org/10.1145/2594291.2594320

SPEC SPECjvm98. 1999. Release 1.03. *Standard Performance Evaluation Corporation* (1999).

Y. Sui, D. Ye, and J. Xue. 2014. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Transactions on Software Engineering* 40, 2 (Feb 2014), 107–122. https://doi.org/10.1109/TSE.2014.2302311

Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting. In *Static Analysis*, Xavier Rival (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 489–510.

Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*

(Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 278–291. https://doi.org/10.1145/3062341.3062360

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* (Dublin, Ireland) *(PLDI '09)*. ACM, New York, NY, USA, 87–97. https://doi.org/10.1145/1542476.1542486

Guoqing Xu and Atanas Rountev. 2008. Merging Equivalent Contexts for Scalable Heap-cloning-based Context-sensitive Points-to Analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08)*. ACM, New York, NY, USA, 225–236. https://doi.org/10.1145/1390630.1390658

Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: Value-Flow-Guided Precise Program Repair for Null Pointer Dereferences. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 512–523. https://doi.org/10.1109/ICSE.2019.00063

Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2017. Machine-Learning-Guided Typestate Analysis for Static Use-After-Free Detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference* (Orlando, FL, USA) *(ACSAC 2017)*. ACM, New York, NY, USA, 42–54. https://doi.org/10.1145/3134600.3134620

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* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 239–248. https://doi.org/10.1145/2594291.2594327

Xin Zhang, Mayur Naik, and Hongseok Yang. 2013. Finding Optimum Abstractions in Parametric Dataflow Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 365–376. https://doi.org/10.1145/2491956.2462185