# Scalable Pointer Analysis of Data Structures using Semantic Models

Pratik Fegade
Oracle Labs and Carnegie Mellon University, USA
ppf@cs.cmu.edu

Christian Wimmer
Oracle Labs, USA
christian.wimmer@oracle.com

## Abstract

Pointer analysis is widely used as a base for different kinds of static analyses and compiler optimizations. Designing a scalable pointer analysis with acceptable precision for use in production compilers is still an open question. Modern object oriented languages like Java and Scala promote abstractions and code reuse, both of which make it difficult to achieve precision. Collection data structures are an example of a pervasively used component in such languages. But analyzing collection implementations with full context sensitivity leads to prohibitively long analysis times.

We use *semantic models* to reduce the complex internal implementation of, e.g., a collection to a small and concise model. Analyzing the model with context sensitivity leads to precise results with only a modest increase in analysis time. The models must be written manually, which is feasible because a model method usually consists of only a few statements. Our implementation in GraalVM Native Image shows a rise in useful precision (1.35X rise in the number of checkcast statements that can be elided over the default analysis configuration) with a manageable performance cost (19% rise in analysis time).

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

***Keywords*** Pointer analysis, semantic models, data structure awareness, GraalVM Native Image

**ACM Reference Format:**
Pratik Fegade and Christian Wimmer. 2020. Scalable Pointer Analysis of Data Structures using Semantic Models. In *Proceedings of the 29th International Conference on Compiler Construction (CC '20), February 22–23, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377555.3377885

ACM ISBN 978-1-4503-7120-9/20/02…$15.00
https://doi.org/10.1145/3377555.3377885

## 1 Introduction

Whole program pointer analysis [48] has applications in a variety of different compiler analyses and optimizations. It has been used for autoparallelization [15, 44], security analysis of applications [29], bugfinding [20], high level synthesis [46] among other applications. Significant amount of work has been done in improving the precision and/or scalability of pointer analysis [18, 28, 51, 58]. Despite this, precise pointer analysis remains expensive and often not scalable.

Repeated analysis of methods under different calling contexts dominates execution time for top-down pointer analysis [59]. Commonly used components, as well as a high degree of abstractions in the form of pointer indirections thus lead to either high analysis costs or low analysis precision.

Collection data structures are an example of frequently used software components. Due to their pervasive use in many contexts in the same application, they have a significant impact on analysis precision. Data structure implementations, however, are often quite complex and involved. For example, the commonly used implementations of hash tables in Java, `HashMap` and `ConcurrentHashMap` contain over 2000 and 6000 lines of code [36, 37], respectively.

Past work has proposed *semantic models* [15] for handling such commonly used data structures. This involves baking in an understanding of common data structures in the compiler by extensions to its intermediate representation (IR). Our insight in this work is that such extensions to the compiler IR are not necessary for the particular case of pointer analysis. Instead, our semantic models can be obtained by manually simplifying data structure implementations by relying on analysis abstractions. This leads to semantic models that do not need modifications to the compiler IR for analysis purposes. We thus simplify the idea of semantic models without much loss in precision for the case of pointer analysis.

In the presence of semantic models, we analyze the actual implementation using imprecise contexts, to retain the analysis effects of the actual implementation, while analyzing semantic models with more precise contexts, to obtain higher precision. This makes the use of semantic models more robust by placing less of a burden on the model with respect to soundness, making it easier for library and framework developers to write semantic models for additional program modules.

```
1   abstract class S { public abstract void doWork(); }
2   class C1 extends S { public void doWork() { ... } }
3   class C2 extends S { public void doWork() { ... } }
4   void f1() {
5     Map<Object, S> m1 = new HashMap<>();
6     Object k1 = new Object();
7     C1 v1 = new C1();
8     m1.put(k1, v1);
9     S o1 = m1.get(k1);
10    o1.doWork();                 // virtual call
11  }
12  void f2() {
13    Map<Object, S> m2 = new HashMap<>();
14    Object k2 = new Object();
15    C2 v2 = new C2();
16    m2.put(k2, v2);
17    S o2 = m2.get(k2);
18    o2.doWork();                 // virtual call
19  }
```

**Listing 1.** Example to demonstrate the cost of analyzing data structure implementations.

In summary, this paper contributes the following:

- We simplify and adapt the idea of semantic models for use in the context of a scalable pointer analysis. Our handling of semantic models in the analysis reduces the burden of faithful modeling of data structures in the semantic model.
- We provide semantic models for common data structures and modify an existing pointer analysis implementation to support them.
- We evaluate our implementation on a set of diverse applications, and show improvements in analysis precision, with little loss in analysis scalability.

## 2 Motivation

Consider the program in Listing 1. It contains two functions, `f1` and `f2`. Each creates a local key-value map (lines 5 and 13) of the type `HashMap`, inserts a pair of objects (lines 8 and 16) and obtains the value back from the map (lines 9 and 17). Let us consider when a pointer analysis can infer that variables `o1` and `o2` refer only to objects of a single type (`o1` refers to instances of class `C1`, while `o2` refers to instances of class `C2`). Such inferences are useful because, in this case, they can be used to devirtualize the calls on lines 10 and 18.

Variables `o1` and `o2` get their values from the calls to `HashMap.get()`. Let us examine the internals of `HashMap` so that the data flow through the map is clear. An abstract representation of the heap structure of the map as implemented in the standard JDK library is shown in Figure 1. The map is a chained hash table containing an array (`elementData`) holding references to nodes (`Node` and `TreeNode`), which form either a linked list or a binary tree, and in turn hold references to key-value pairs. In order to precisely infer the

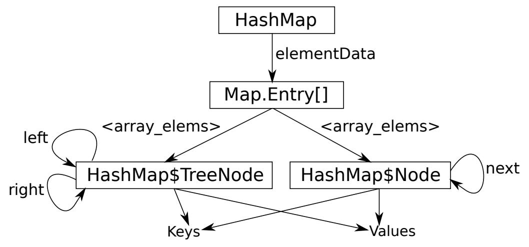


**Figure 1.** Internal heap organization of data in `HashMap`.

types of objects variables `o1` and `o2` refer to, the analysis needs to distinguish between the two maps, as well as their internal structures (the array and the nodes). This requires at least 3-deep heap contexts when using object sensitivity [31]. Previous work has shown that such deep contexts are not scalable [28]. Even when contexts are selectively used only for data structures, we show in §6 that the analysis performance is unsuitable for use in production compilers.

We note the following two problems that lead to imprecision and a lack of scalability.

- A lot of implementation artifacts do not affect the results of a static analysis, but lead to higher analysis costs. For example, the implementation of `HashMap` switches between organizing the nodes in a linked list and a binary tree for better lookup performance. Such performance optimizations lead to an increase in the analysis cost, while not affecting analysis correctness.
- To achieve context sensitivity, the actual implementation of data structures must be analyzed every time the data structure is used in a different context.

## 3 Background

In this section, we give a brief introduction to semantic models (§3.1) and GraalVM Native Image (§3.2), the compiler infrastructure we use.

### 3.1 Semantic Models

The idea of semantic models, as proposed in [15], is to enrich the compiler IR with operations on commonly used data structures like maps and lists. Then, purely for the purposes of static analysis, one can write alternate implementations of such data structures in terms of these added IR operations. A static analysis can then analyze the semantic model instead of the actual, possibly complex, implementation. Note that semantic models are never compiled or executed at runtime. They represent a mapping between concrete implementations of data structures (such as `HashMap` or `ArrayList`) and the data structure IR operations that the compiler understands. The developer of the data structure library or a user of the analysis can write such semantic models. The high level goal of semantic models therefore is to provide an alternate, simpler implementation that can be analyzed without the prohibitive costs of analyzing the actual implementation.

Semantic models can improve the analysis in terms of both precision and performance. Directly encoding the semantics of complex data structure operations into the compiler IR can lead to higher precision. For example, the extension to pointer analysis to handle the built-in map operations in [15] allows the analysis to keep track of mappings between abstract key and value objects. Semantic models are significantly more concise and simpler than the corresponding actual implementations. The analysis can thus get away with shallower contexts, without loss in precision. The concise models reduce the analysis effort as well.

### 3.2 GraalVM Native Image

GraalVM Native Image [34, 35, 55] is an ahead-of-time compiler for Java applications built on top of the GraalVM compiler [11, 12, 14, 27]. It optimizes for start up time and memory footprint of the generated binaries. In order to reduce the generated binary sizes as well as compilation times, it constructs a call graph of the application, which is used to determine the set of reachable methods to compile. The call graph construction is carried out in conjunction with a classic flow insensitive, path sensitive, SSA-based Andersen style pointer analysis [1]. The results of the analysis are also used for optimizations like devirtualization and checkcast elision.

GraalVM Native Image eagerly runs initialization code at build time. The user can also annotate certain portions of the application to execute at this time. A snapshot of the objects allocated by this initialization code is then taken and included in the generated binary. This so-called *image heap* is memory mapped in the heap of the application on startup, thus reducing startup cost.

In order to ensure that the pointer analysis has visibility of the image heap, the analysis is performed iteratively. Each iteration of the analysis scans the image heap objects and updates the points-to sets accordingly.

GraalVM Native Image makes a closed world assumption to enable ahead of time compilation, thus requiring the user to list, at build time, all uses of language features like reflection, native code access, and unsafe memory accesses (via `sun.misc.Unsafe` in Java). The interested reader is referred to [55] for further details regarding GraalVM Native Image.

## 4 Mechanism

Building on the previous sections, we now describe how a user can go about writing semantic models (§4.1), and how the pointer analysis uses them to gain precision (§4.2).

### 4.1 Writing Semantic Models

In the original proposal, semantic models involve extending the compiler IR with higher level operations. The authors used this additional information available to the compiler not just for pointer analysis but for autoparallelization as well. For the more common case of pointer analysis, however, we can significantly simplify the idea of semantic models. Instead of adding new IR operations, we aim to represent data structures with already available language level operations. We do so by taking advantage of the following observations:

**O.1** Performance optimizations often do not affect the API contract of data structures.

**O.2** Static analysis abstracts concrete program semantics for purposes of tractability and scalability [8]. For example, pointer analysis implementations, including the one in GraalVM Native Image, are often flow-insensitive, and thus disregard the order of statements in programs.

**O.3** Internal implementation details of a data structure, or an API in general, do not matter as long as the specification of the API is met.

Consider again the implementation of `HashMap`. Recall from Figure 1 that the implementation uses an array of `Map.Entry` objects, which are organized as linked lists or trees, and each of which holds a key-value pair.

We now simplify the data representation of the key-value map, based on the observations listed above. The internal map nodes can be organized as a binary tree, or a linked list. This is a performance optimization to avoid linear worst-case lookup times. Because we are concerned only with the correctness of the implementation, we can elide this performance optimization (observation **O.1**).

Most pointer analysis implementations collapse elements of an array into a single abstract location. The `elementData` array can be replaced with a single field that represents all locations of the array, with no effect on analysis precision.

Furthermore, observe that the `Node` class is completely internal to the map implementation, and hence can be inlined (observation **O.3**). Reifying these changes in the implementation, while keeping in mind the flow-insensitive nature of the analysis (observation **O.2**) leads to the semantic model shown in Listing 2. We only show a few methods for the purposes of illustration. The `allKeys` field is needed for operations such as iteration over keys in the map that are not shown in the excerpt.

Listing 2 is an intuitive summary of the data structure implementation: Read-methods return those, and only those, objects that are put in by write-methods. The fields `allKeys` and `allValues` thus represent the stores of all data that is put into the map.

### 4.2 Using Semantic Models: Allocation

A pointer analysis uses a single abstract object to represent multiple concrete objects. To use semantic models for objects of a certain type, we modify the creation of abstract objects in the analysis. Every time an abstract object is created for a type for which we have a semantic model available, we also create an abstract object of the corresponding semantic model type (except for the case of iterators and collection views, which we discuss in §4.4). For example, in

```
@SemanticModel(originalClass = java.util.HashMap.class)
class HashMap_Model<K, V> {
  K allKeys;
  V allValues;
  public HashMap_Model() {}
  public V get(Object key) { return this.allValues; }
  public V put(K key, V value) {
    this.allKeys = key;
    this.allValues = value;
    return this.allValues;
  }
  public boolean contains(Object key) {
    return unknownBoolean();
  }
  public V remove(Object key) {
    // do nothing because analysis
    // does not model strong updates.
    return this.allValues;
  }
}
```

**Listing 2.** Excerpts of the semantic model for HashMap.

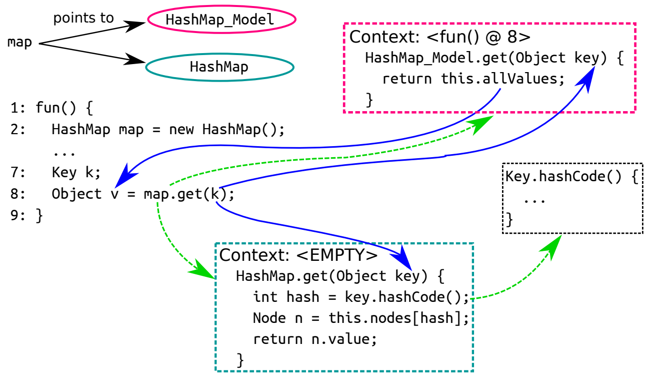
Figure 2, the variable map points to two abstract objects: one corresponding to the original type HashMap, and the other corresponding to the semantic model type HashMap_Model.

### 4.3 Using Semantic Models: Call Resolution

We now consider the resolution of calls to semantically modeled methods, similar to the call to HashMap.get() in Figure 2. As mentioned in the previous section, the variable map points to both an abstract object corresponding to the type HashMap and one corresponding to the semantic model type HashMap_Model. When the call to HashMap.get() on line 8 is resolved during analysis, we ensure that both the actual implementation as well as the semantic model are inferred to be targets of the call. On a first glance, this seems to negate the benefits of semantic models as we also analyze the actual implementation. We now explain why this is needed, and how we avoid the precision loss.

Analyzing the complex actual implementation has high analysis costs. Therefore, to obtain the most scalability benefits, the call should be resolved only to the semantic model method HashMap_Model.get(). This is however unsound as there can be methods reachable from the actual implementation that are not reachable from the semantic model implementation. Such methods could be part of the internal implementation of HashMap, such as constructors of the internal Node and TreeNode classes (Figure 1). If the analysis only ever analyzed the semantic model (shown in Listing 2), such methods would not be discovered as reachable, leading to unsoundness. Analyzing both the actual as well as the semantic model implementations ensures soundness.

In order to avoid the precision loss, we exploit the following observations:



```
1: fun() {
2:   HashMap map = new HashMap();
     ...
7:   Key k;
8:   Object v = map.get(k);
9: }
```

```
HashMap_Model.get(Object key) {
  return this.allValues;
}
```

```
HashMap.get(Object key) {
  int hash = key.hashCode();
  Node n = this.nodes[hash];
  return n.value;
}
```

```
Key.hashCode() {
  ...
}
```

**Figure 2.** Call resolution for a semantically modeled method. The blue arrows represent flow of analysis information, while the green dashed arrows are edges of the call graph.

1. Analysis soundness is unaffected by the precision of analysis contexts.
2. Most data structure implementations interact with the program that uses them only via function calls, and not via global variables or static fields.

Based on the first observation, it is clear that a context-insensitive analysis of the actual implementation is sufficient to ensure soundness. So while the call to HashMap.get() on line 8 of function fun() in Figure 2 gets resolved to both HashMap.get() as well as HashMap_Model.get(), the actual method gets analyzed under a coarse context <EMPTY>, while the model method is analyzed under the more precise context <fun()@8>.

In order to take advantage of the more precise analysis of semantic models, we now note the second observation. This suggests that the analysis would obtain more precise results if the flow of analysis information is blocked along return path from the actual implementation. Since we do not hinder the flow of information into the method, reachablility is not affected (we discuss the soundness of this approach in more detail in §4.6). Accordingly, note how analysis information flows into variable v only from the semantic model method and not the actual implementation in Figure 2.

Analyzing the actual implementation also helps us simplify semantic models further. The writer of the model does not have to worry about calling methods, both internal to the data structure as well as the API-specified callbacks into user code (like equals() and hashCode()), when writing the semantic models. This both reduces the effort of writing semantic models and simplifies the structure of the models.

In summary, analyzing the actual implementation is necessary to ensure the soundness of the analysis, while the lower analysis cost of semantic model enables us to use more precise contexts for them thus giving us precision. Our approach to handling call resolution can be seens as a compromise between the two analysis goals of soundness and precision.

### 4.4 Iterators and Collection Views

The Java Collections Framework offers different kinds of iterators and views for its collections. For example, the "keyset" view of a map is a set of all keys contained in the map. Mutations to the underlying collection (the map in our example) are reflected in the set view and vice versa. Listing 3 shows how such a keyset view is modeled for `HashMap`. Other views and iterators can be modeled similarly.

```java
class HashMap_Model<K, V> {
  public Set<K> keySet() {
    return new KeySet_Model(this);
  }
}
class KeySet_Model<K, V> {
  HashMap_Model<K, V> map;
  KeySet_Model(HashMap_Model<K, V> map) {
    this.map = map;
  }
  public boolean remove(K key) {
    return this.map.remove(key) != null;
  }
}
```

**Listing 3.** Modeling keyset views on `HashMap`.

We discussed in §4.2 how we create an abstract object corresponding to the appropriate semantic model type, if one is available. Iterators and views on collections are however handled differently. The semantic models explicitly allocate an instance of the semantic model type for iterators and views. This is illustrated in Listing 3. This is needed because the model object of an iterator or a view needs access to the model object of the underlying collection so that mutations to one are reflected in the other.

### 4.5 Modeling Other Collections

We have focused our discussion on semantically modeling a map. Modeling other container data types like lists is similar. Operations on a list, for example, can be abstracted to reads and writes to a single field. This field then represents all the elements in the list. Looked at this way, our model for a map approximates the map as two containers: one for all the keys in the map, and another for all the values.

The fact that semantic models model the public API behavior of an implementation means that a single semantic model works for multiple implementations of the API, barring a few implementation details. For example, the semantic models for `HashMap` and `ConcurrentHashMap` share a significant amount of code by the means of inheritance in our implementation.

### 4.6 Soundness

As discussed, semantic models are manually written. We thus require the model authors to ensure the correctness of the provided models. Given that the semantic models model API visible effects of the modeled implementation soundly, we now informally discuss the soundness of the approach in terms of the points-to information computed for variables and object fields, and the generated call graph. These are the primary results of the pointer analysis the compilation process in GraalVM Native Image uses.

Let $C$ be the call graph generated when the analysis does not use semantic models, and $C_s$ be the call graph generated when the analysis uses semantic models. Let M be the set of methods that are modeled using semantic models when generating $C_s$. We now see how all methods reachable in $C$, are also reachable in $C_s$. Let $R_m$ be the set of methods in $C$ that are reachable via call paths that include a method in $M$, and let $R_{nm}$ be the set of methods that are reachable via call paths that do not include any method in $M$. When generating $C_s$, since we analyze the actual implementation of the data structure as well (§4.3), methods in $R_m$ are also reachable in $C_s$. Since we also assume that the semantic models soundly model the API effects of the actual implementation, the points-to information for all variables and fields in the program is sound. This implies that methods in $R_{nm}$ are also reachable in $C_s$.

There is one case, that of a semantically modeled method calling another modeled method, which warrants some more attention. In the context of our running `HashMap` example, consider the code in Listing 4. Suppose the `HashMap` class had a method `containsPair` as shown in the listing. This method checks if the map contains a key-value pair. To do this, it calls the `HashMap.get()` method and compares the result with the parameter value object. So the actual implementation calls an API method that has a semantic model. Recall from §4.3 that we do not allow the return values of actual implementations to flow into the caller. As a result the return value of the call to the implementation method `HashMap.get()` does not flow out. However, the call to `HashMap.get()` also gets resolved to the method `HashMap_Model.get()` even though the call site is within the implementation class. Therefore, the correct return types flow out of the call of `get()`, and the `equals()` method is seen as reachable for all necessary types.

```java
class HashMap<K, V> {
  public boolean containsPair(Object k, Object v) {
    return this.get(k).equals(v);
  }
}
```

**Listing 4.** Modeled method call from another modeled method.

The soundness of using semantic models relies on the analysis being able to update the heap state of the abstract model objects in tandem with the actual abstract objects. This is

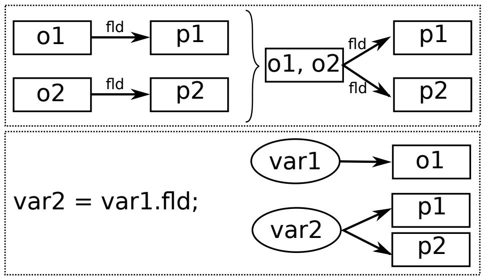


**Figure 3.** Precision loss after merging objects `o1` and `o2`.

achieved via the method implementations in the semantic models. When semanitically modeled data structures are mutated using Java features such as reflection, the Java Native Interface, or unsafe memory access via `sun.misc.Unsafe`, such mutations are not reflected in the corresponding model objects, resulting in unsoundness. As we model library data structures, we argue that mutating the internals of the data structures via any of the said mechanisms is rare as libraries do not provide guarantees about such implementation details as part of their API specification. As described in §3.2, all such accesses are listed at build time. It is therefore possible to check for such mutations and report them as errors in the static analysis, thus having no effect on analysis soundness.

## 5 Implementation

This section presents details that are not novel contributions of the paper, but provide important background information to understand how the system works in practice and to understand the implementation and evaluation.

### 5.1 Pointer Analysis Precision

As GraalVM Native Image aims to be a production compiler, there are stringent scalability requirements on the pointer analysis. In order to meet these requirements, by default, the pointer analysis is context insensitive and uses one abstract object for each type. As far the abstract analysis objects are concerned, the analysis is thus merely type sensitive in the default case. In the (non-default) case of allocation site sensitivity, the analysis employs merging of abstract objects for scalability. When the number of abstract objects of a particular type in a points-to set increases beyond a threshold, all objects corresponding to that type in that points-to set are merged into a single abstract object. Such merging leads to a loss in precision as it amounts to abstracting multiple objects into a single object. This is illustrated in Figure 3. The field `fld` on abstract objects `o1` and `o2` points to objects `p1` and `p2` respectively. After merging `o1` and `o2`, however the field loses precision as it now points to both objects `p1` and `p2`. Despite the fact that `var1` points only to `o1` and not `o2`, a load of `fld` on `var1` now results in both `p1` and `p2`.

### 5.2 Semantic Models and Heap Scanning

§3.2 introduced how the pointer analysis implementation in GraalVM Native Image scans the image heap to capture the effects of compile time initialization. Like in the static case, we create a corresponding abstract semantic model object every time we encounter a data structure we have a model for, when scanning the heap. In order to update the abstract semantic model objects, the model author also needs to provide a scanner for each semantically modeled data structure. This scanner iterates over the data structure and updates the points-to sets of fields, and other structures reachable from the semantic model objects. For example, a scanner for `HashMap` would iterate over the key-value pairs in it, while adding the corresponding abstract objects in the points-to sets of the fields `HashMap_Model.allKeys` and `HashMap_Model.allValues` of the semantic model object.

Iterators and view objects are handled slightly differently in this case. We saw in §4.4 how mutations on such view types are reflected in the underlying collection and vice versa. As a result, the semantic model of a view type contains a reference to the underlying collection (field `KeySet_Model.map` in Listing 3). Obtaining the abstract object corresponding to the underlying collection requires us to have a mapping between the iterator/view object and the concrete underlying collection object, which is difficult without accessing internal fields of the iterator or view object. We chose not to do so as that would make the scanning dependent on the collection implementation. As a result, we do not semantically model views and iterators allocated at build time. We did not observe significant drops in precision due to such iterators or views in the image heap for our benchmarks and hence believe that this conservative modeling is sufficient in most cases.

### 5.3 Semantic Models in the Compilation Flow

In our implementation, semantic models are implemented as regular Java classes and hence are represented like any other class in the analysis. We use Java annotations (Listing 2) to map the model to the corresponding modeled types.

Semantic models are unsound for concrete execution. As a result, we need to make sure they are not compiled or executed. After the execution of the pointer analysis, we iterate over the entire points-to graph and remove references to semantic models. Calls to semantic model methods, as well as abstract objects corresponding to semantic model types in points-to sets of fields, variables and arrays are removed. Further stages of compilation thus are not affected by semantic models and do not need any modifications. Our modifications do not hinder the compiler's functionality and our implementation can generate functioning binaries of applications, while enjoying the benefits of semantic models.

| Analysis config. | Semantic Models | Abstract objects | Context sensitivity | Merging for model types |
|---|---|---|---|---|
| INSENS | No | Per type | None | - |
| DSIMPLSENS | No | Per allocation for collections, per type for others | Callsite sensitive for calls on collections, none for others | - |
| SEMMERGE | Yes | Per allocation for models, per type for others | Callsite sensitive for calls on models, none for others | Yes |
| SEMNOMERGE | Yes | Per allocation for models, per type for others | Callsite sensitive for calls on models, none for others | No |

**Table 1.** Analysis configurations used for evaluation

| Benchmark | Jar size (KB) | #classes | Description |
|---|---|---|---|
| helidon [38] | 6548 | 4347 | Microservices framework |
| micronaut [33] | 12092 | 8387 | Microservices framework |
| quarkus [43] | 7656 | 5426 | Microservices framework |
| djbdd [30] | 3240 | 1564 | BDD library |
| jacc [24] | 108 | 76 | Parser generator |
| janino [54] | 5672 | 3599 | Java compiler |
| jatalog [49] | 508 | 315 | Datalog engine |
| jlex [2] | 56 | 26 | Lexer generator |
| jtidy [41] | 2144 | 1213 | HTML syntax checker |
| raytracer [32] | 28 | 30 | Raytracer |
| sablebdd [40] | 56 | 54 | BDD Library |
| xalan | 5580 | 3365 | XSLT processor from DaCapo [3] benchmark suite |

**Table 2.** Benchmarks used for evaluation. The jar file size and the number of classes include all dependencies except the Java standard library.

## 6 Evaluation

We measure analysis precision and performance with and without semantic models for the evaluation. We compare different analysis configurations as described in Table 1.

We provide semantic models for the following collections for the analysis configurations SEMMERGE and SEMNOMERGE,

- `java.util.ArrayList`,
- `java.util.LinkedList`,
- `java.util.HashMap`,
- `java.util.LinkedHashMap`, and
- `java.util.concurrent.ConcurrentHashMap`

These were chosen as they are some of the most commonly used collection data structures in Java.

We use the applications listed in Table 2 as benchmarks. Micronaut, Quarkus, and Helidon are all frameworks for building cloud based applications in Java. They were chosen as Java is extensively used in this domain as well as because the past work [55] on GraalVM Native Image uses them for evaluation. The remaining benchmarks are general-purpose applications spanning multiple diverse domains chosen from the set of applications the original proposal for semantic models [15] evaluates on. Given the closed world assumptions GraalVM Native Image makes, these were the benchmarks that we found we could evaluate on.

All experiments are performed on a 4-core, 8-thread Lenovo ThinkPad T480 laptop with an Intel Core i5-8350U CPU at 1.70 GHz, 16 GByte of RAM, 1 GByte of swap, running Ubuntu 19.04 with Linux kernel version 5.0.0 and Oracle Java HotSpot Virtual Machine 1.8.0_212-jvmci-19.2-b01 with JVMCI enabled. The analysis timeout is set to 1 hour.

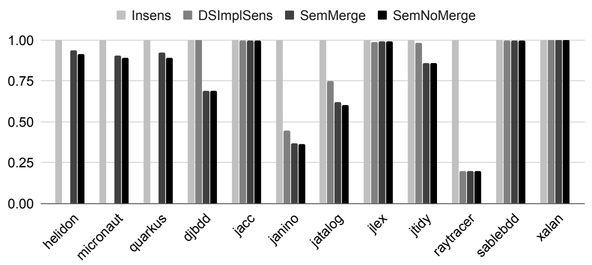
We perform the experiments on a laptop, as opposed to a more powerful workstation because we expect the pointer analysis to be run frequently during development as a part of compilation with GraalVM Native Image, on development machines, rather than server class machines. This is also the reason why the analysis timeout is set slightly lower than what is generally used in past work [28, 50].

Note that the analysis precision statistics and the analysis runtimes are collected from two different runs of the analysis. This is because statistics collection is easier when additional analysis information is maintained during the analysis. This information has no effect on the correctness of the analysis but tracking it affects the runtime of the analysis.

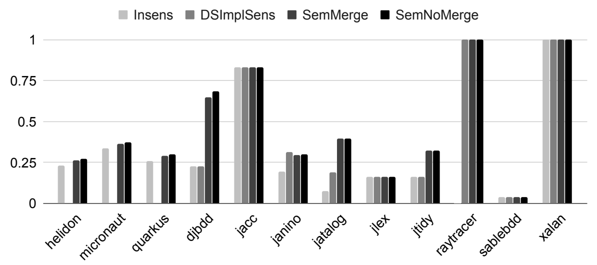
Figures 4 and 5 show the analysis precision and performance results respectively, for all the benchmarks. We also show summaries of these results for easy comprehension in Tables 3 and 4. Table 4 excludes benchmarks that timed out in any of the four analysis configurations, while Table 3 includes all benchmarks for a configuration that could be analyzed completely.

### 6.1 Analysis Precision

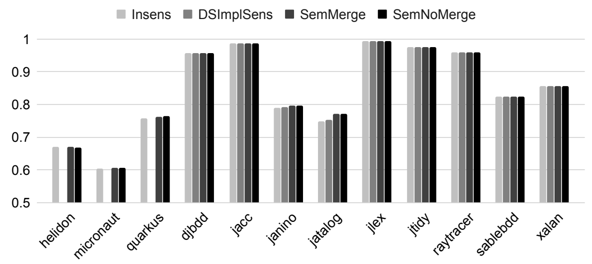
We use four metrics to evaluate analysis precision: the mean number of types in the points-to sets of variables, the fraction of `checkcast` statements that can be elided, the number of reachable methods discovered by the analysis and the fraction of virtual calls that can be devirtualized based on the analysis results. Elided casts excludes casts for exception types as these are mostly associated with exception handling. Points-to sets of variables declared in semantic models, or in the modeled classes are excluded in all of the analysis configurations. Furthermore, the number of reachable methods discovered by the analysis and the mean number of types in the points-to sets of variables are normalized with respect to the INSENS configuration. Precision results for all benchmarks are shown in Figure 4.



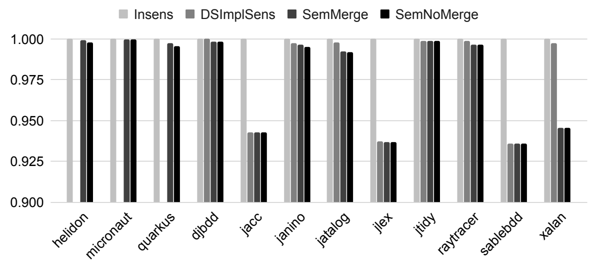
**(a)** Normalized mean number of types in points-to sets



**(b)** Fraction of checkcast statements elided



**(c)** Fraction of devirtualized calls



**(d)** Normalized reachable methods

**Figure 4.** Analysis precision results. Analysis configuration DSImplSens timed out for helidon, micronaut and quarkus. All metrics exclude methods in the Java standard library.

| Analysis Config | Normalized Analysis Time | Normalized Reachable Methods | Fraction of Devirtualized Calls | Fraction of Casts Elided | Normalized Mean #Types in Points-to Sets |
|---|---|---|---|---|---|
| Insens | 1.0 | 1.0 | 0.89 | 0.34 | 1.0 |
| DSImplSens | 43.43 | 0.98 | 0.89 | 0.36 | 0.9 |
| SemMerge | 1.09 | 0.97 | 0.9 | 0.46 | 0.82 |
| SemNoMerge | 1.17 | 0.97 | 0.9 | 0.47 | 0.81 |

**Table 3.** Summarized pointer analysis results. For a fair comparison, averages do not include benchmarks that timed out in any of the configurations.

| Analysis Config | Normalized Analysis Time | Normalized Reachable Methods | Fraction of Devirtualized Calls | Fraction of Casts Elided | Normalized Mean #Types in Points-to Sets |
|---|---|---|---|---|---|
| Insens | 1.0 | 1.0 | 0.84 | 0.32 | 1.0 |
| SemMerge | 1.19 | 0.98 | 0.85 | 0.47 | 0.79 |
| SemNoMerge | 1.83 | 0.98 | 0.85 | 0.47 | 0.78 |

**Table 4.** Summarized pointer analysis results. These results include timed out runs and hence include the three large benchmarks. DSImplSens is not included here as it times out on these. These numbers more realistically reflect overheads due to semantic models.

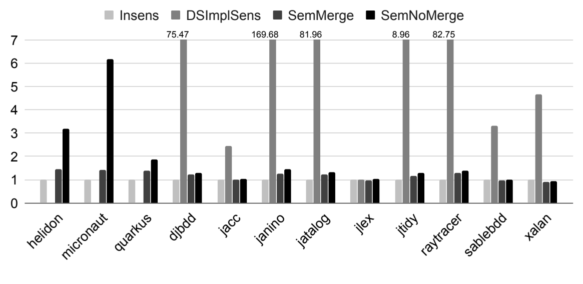


**Figure 5.** Analysis time results. Analysis configuration DSImplSens timed out for helidon, micronaut and quarkus.

As can be seen in Table 4, the mean number of types in the points-to sets decreases significantly by 21% in the SemMerge configuration and by 22% in the SemNoMerge configuration as compared to the default baseline. This increase in precision is useful as it can impact the precision as well as runtimes of downstream analyses that use the results of the points-to analysis.

Table 3 shows that precision is generally worse for DSImplSens as compared to SemMerge. Despite effectively having the same level of context sensitivity, we believe this is because there is more chance of abstract objects being merged in DSImplSens as compared to the semantic model cases.

Merging of objects corresponding to any of the multiple internal structures in the data structure might cause a loss in precision in DSImplSens. Thus semantic models appear to be more robust to accidental merging.

The analysis configurations SemMerge, SemNoMerge and DSImplSens primarily lead to an increase in precision of the results of data structure operations. Due to Java's type erasure [39], results of operations on data structures are often guarded by checkcast statements. The increase in precision in these analysis configurations thus leads to an increase in the fraction of such cast statements that can be proven safe statically and hence elided. This increase in precision however does not have a significant effect on other parts of the program, as evidenced by the fraction of calls that can be devirtualized because the knowledge of types in Java is already enough to devirtualize a significant fraction of virtual calls. We believe this is also the reason why the number of reachable methods does not decrease significantly.

## 6.2 Analysis Performance

We use analysis time as a measure of analysis scalability. We normalize the analysis time with respect to that of the Insens configuration. These results is shown in Figure 5.

Using semantic models amounts to a higher analysis cost (19% increase in the case of SemMerge on average) as compared to the imprecise baseline because there is additional analysis cost in analyzing the semantic model implementations with their higher context sensitivity. DSImplSens, however leads to a 43X increase in the analysis runtime. For three of our moderately sized benchmarks (helidon, micronaut and quarkus), DSImplSens does not even complete execution, timing out in each case.

## 6.3 Relationship between Analysis Precision and Performance

It is observed that pointer analysis cost often first drops, before rising again, when the analysis precision is increased in the form of higher context sensitivity. While deeper contexts lead to a higher number of points-to sets, their overall size decreases due to the added precision.

We however did not observe such an effect in our evaluation. This is because the baseline configuration Insens abstracts all objects of the same type to a single abstract object. The size of points-to sets is thus bounded by the total number of types in the application, which is generally quite low even for large applications. Thus the decrease in the points-to set sizes does not offset the increase in their number with higher precision in the form of semantic models.

For the evaluation, we have provided semantic models for a few commonly used data structures in the Java standard library. As a result, benefits are seen only for benchmarks that use these data structures. Some benchmarks like jacc and raytracer, for example do not extensively use any standard library data structure. This explains the high variation in the results obtained. We stress that this is not a limitation of the approach, as the user can easily add semantic models for other data structures they wish to use. This extensibility is practically useful as a number of large applications implement specific data structures tuned for their use cases [17, 52, 53].

```
class Graph_Model<Pn, Pe> {
  Node_Model<Pn> allNodes;
  Edge_Model<Pe> allEdges;
}
class Node_Model<Pn, Pe> {
  Edge_Model<Pn, Pe> allIncidentEdges;
  Pn allProperties;
}
class Edge_Model<Pn, Pe> {
  Node_Model<Pn, Pe> bothIncidentNodes;
  Pe allProperties;
}
```

**Listing 5.** Possible semantic model for a property graph. We focus on modeling data and do not show method implementations for brevity.

## 7 Models for Other Data Structures and Analyses

We believe that it is easy to generalize semantic models for other data structures, as well as other non-data structure classes. We briefly discuss how this could be done for graphs. Graphs are commonly used to represent various kinds of data [10, 21]. Such graph data is generally organized as a set of graph nodes, each of which has edges to other nodes in the graph. Nodes and edges can have labels and properties associated with them. Depending on the actual implementation of the graph under consideration, a semantic model for such a graph could look like the one shown in Listing 5.

Similarly, since most static analyses abstract the concrete semantics of the underlying languages in some way, we believe that it should be possible to write semantic models for other kinds of static analysis. For example, [19] showed how pointer and taint analyses can be unified in a single framework, thus making it straightforward to use semantic models for static taint analysis. Depending on how an analysis abstracts concrete semantics, the models might differ from our models for pointer analysis. For example, semantic models written for an analysis for race detection may retain code for synchronization, which we elide for a pointer analysis.

Semantic models can also be used for modeling hard-to-analyze features such as reflection and native code accesses. For example, field loads using reflection can be presented to the analysis as normal field loads using semantic models.

## 8 Related Work

**Semantic Models and Data Structure Awareness:** As discussed in the earlier sections, semantic models were proposed in [15]. §4 explains how our work significantly simplifies semantic models. As a result of this simplification, we also believe we make them more general and easier to write for the library developer, or even the end user.

Our simplifications to semantic models, specifically the fact that data structures are no longer first class objects in the compiler leads to a certain loss in precision as far as key-value maps are concerned. This is because the analysis can no longer maintain a mapping between abstract key and value objects. We believe that this loss in precision not be significant, especially because the abstract objects in the pointer analysis in GraalVM Native Image are coarser grained (§5.1) as compared to the ones in [15]. In fact, this simpler design allows us to be more general as we do not have to modify the compiler IR or accompanying analyses for adding semantic models for additional data structures.

Cohen et al. [7] propose a container aware alias analysis. However, their proposal is not as easily extensible in terms of new data structures as ours is. Their analysis is designed for the special case of container traversals and is geared more towards autoparallelization and related optimizations.

Shape analysis [5, 16, 45, 56] can identify data structures, infer properties about them and use these for a variety of applications. The approach of semantic models differs from shape analysis in that the necessary inferences about data structures are provided to the analysis externally in the form of semantic models thus reducing analysis effort in performing these inferences.

**Procedure Summaries:** Semantic models are similar to procedure summaries [47, 57] as both of them use summarized representations of the actual implementation. Procedure summaries, however, are automatically generated by the analysis and hence cannot aggressively simplify the implementation like users can when they write semantic models.

In [13], the authors are able to infer API specifications like described in §2 using machine learning. However, the inferred specifications could be unsound, and possibly require expensive ML training for every new library, or data structure, that needs to be added. Obtaining training programs using new and custom data structures might not be feasible either. Writing semantic models for data structure would therefore be significantly easier and less time consuming.

**Manual Specifications, Annotations and API Stubs:** There has been a significant amount of work exploring manual or external specifications or annotations for static analysis and program verification [4, 6, 23, 25, 26]. Semantic models also involve the user specifying the behavior of data structures for an analysis to use. One difference between semantic models and the previous work on manual specifications is that the latter required the user to specify program behavior in some logic, or a similarly constructed language. However semantic models can be written in Java without any modifications to the language. Also, semantic models focus more on modeling the heap-related behavior of the program, while specifications can be quite general. This contributes to the simplicity of semantic models. Also, because we analyze the actual data structure implementation as well, semantic models need not be sound with respect to all program behaviors.

Stubbing out APIs is well-known as an analysis engineering technique, used often to model APIs the source code for which is not available during analysis. However, our work goes beyond simple function-effect summaries because we can allow models (or "stubs") that simplify the actual implementation, skipping some effects or aspects of the algorithm if they are not relevant to the analysis. We can do this because we are aware of what the analysis is actually trying to derive. We can thus often omit complex data-structure optimizations, or fast-paths, or other engineering details.

The Jahob system for program verification [26] has a notion of data structures and allows the user to write specifications which the system can verify. While similar to semantic models, the specifications required for the system to work are more complex as compared to the semantic models.

**Domain Specific Languages (DSLs):** In domain specific languages or frameworks [9, 22, 42], the compiler can optimize programs based on a high level knowledge about domain specific data structures and operations on them. Semantic models also, in some sense, make the compiler aware of (a part of) the semantics of data structures. This is done without restricting the user to a particular DSL.

## 9 Conclusions

This work evaluates the feasibility of employing past work on semantic models in the context of a pointer analysis in a production compiler. We believe that the results obtained are encouraging and therefore suggest the usefulness of such approaches in production settings. As discussed in §7, we believe that approaches similar to semantic models can be fruitfully employed to improve the precision as well as scalability of a variety of static analyses other than pointer analysis. Encouraged by the results obtained, we believe that semantic models could similarly improve the precision and/or scalability of other such analyses.

## Acknowledgments

We thank Chris Fallin, Todd Mowry, Phil Gibbons, and Codrut Stancu for their support, contributions, and feedback on this paper.

## References

[1] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Technical Report.

[2] Elliot Joel Berk and C. Scott Ananian. 2003. JLex: A Lexical Analyzer Generator for Java. https://www.cs.princeton.edu/~appel/modern/java/JLex/

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications.* ACM Press, 169–190. https://doi.org/10.1145/1167473.1167488

[4] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A Type and Effect System for Deterministic Parallel Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA ’09).* ACM Press, New York, NY, USA, 97–116. https://doi.org/10.1145/1640089.1640097

[5] Cristiano Calcagno, Dino Distefano, Peter W. O’Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6, Article 26 (Dec. 2011), 66 pages. https://doi.org/10.1145/2049697.2049700

[6] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. 2015. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. *ACM Trans. Comput. Syst.* 32, 4, Article 10 (Jan. 2015), 47 pages. https://doi.org/10.1145/2699681

[7] Albert Cohen, Peng Wu, and David Padua. 2001. Pointer Analysis for Monotonic Container Traversals.

[8] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages.* ACM Press, 238–252. https://doi.org/10.1145/512950.512973

[9] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. 2011. Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’11).* ACM Press, New York, NY, USA, Article 9, 12 pages. https://doi.org/10.1145/2063384.2063396

[10] Georgios Drakopoulos, Andreas Kanavos, and Athanasios Tsakalidis. 2016. Evaluating Twitter Influence Ranking with System Theory. https://doi.org/10.5220/0005811701130120

[11] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Speculation Without Regret: Reducing Deoptimization Meta-data in the Graal Compiler. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ ’14).* ACM Press, New York, NY, USA, 187–193. https://doi.org/10.1145/2647508.2647521

[12] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages (VMIL ’13).* ACM Press, New York, NY, USA, 1–10. https://doi.org/10.1145/2542142.2542143

[13] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin Vechev. 2019. Unsupervised Learning of API Aliasing Specifications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019).* Association for Computing Machinery, New York, NY, USA, 745–759. https://doi.org/10.1145/3314221.3314640

[14] Josef Eisl, Matthias Grimmer, Doug Simon, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Trace-based Register Allocation in a JIT Compiler. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ ’16).* ACM Press, New York, NY, USA, Article 14, 11 pages. https://doi.org/10.1145/2972206.2972211

[15] Christopher Fallin. 2019. *Finding and Exploiting Parallelism with Data-Structure-Aware Static and Dynamic Analysis.* Ph.D. Dissertation. Pittsburgh, PA, USA.

[16] Rakesh Ghiya and Laurie J. Hendren. 1996. Is It a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-directed Pointers in C. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages.* ACM Press, 1–15. https://doi.org/10.1145/237721.237724

[17] Google. 2019 (accessed October 17, 2019). Gson. https://github.com/google/gson/blob/master/gson/src/main/java/com/google/gson/internal/LinkedTreeMap.java

[18] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2018. Shooting from the Heap: Ultra-scalable Static Analysis with Heap Snapshots. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018).* ACM Press, New York, NY, USA, 198–208. https://doi.org/10.1145/3213846.3213860

[19] 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

[20] Samuel Z. Guyer and Calvin Lin. 2005. Error checking with client-driven pointer analysis. *Science of Computer Programming* 58, 1 (2005), 83 – 114. https://doi.org/10.1016/j.scico.2005.02.005 Special Issue on the Static Analysis Symposium 2003.

[21] Nathan Hawes, Ben Barham, and Cristina Cifuentes. 2015. FrappÉ: Querying the Linux Kernel Dependency Graph. In *Proceedings of the GRADES’15 (GRADES’15).* ACM Press, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/2764947.2764951

[22] Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII).* ACM Press, New York, NY, USA, 349–362. https://doi.org/10.1145/2150976.2151013

[23] Daniel J. Quinlan, Markus Schordan, Richard Vuduc, and Qing Yi. 2006. Annotating user-defined abstractions for optimization. *20th International Parallel and Distributed Processing Symposium, IPDPS 2006* 2006. https://doi.org/10.1109/IPDPS.2006.1639722

[24] Mark P Jones. 2019 (accessed August 16, 2019). Jacc: Just another compiler compiler for Java. http://web.cecs.pdx.edu/~mpj/jacc/.

[25] Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, and Keshav Pingali. 2011. Exploiting the Commutativity Lattice. *SIGPLAN Not.* 46, 6 (June 2011), 542–555. https://doi.org/10.1145/1993316.1993562

[26] Viktor Kuncak and Martin Rinard. 2006. An Overview of the Jahob Analysis System: Project Goals and Current Status. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS’06).* IEEE Computer Society, Washington, DC, USA, 285–285. http://dl.acm.org/citation.cfm?id=1898699.1898807

[27] David Leopoldseder, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. 2018. A Cost Model for a Graph-based Intermediate-representation in a Dynamic Compiler. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL 2018).* ACM Press, New York, NY, USA, 26–35. https://doi.org/10.1145/3281287.3281290

[28] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. 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 (ESEC/FSE 2018)*. ACM Press, New York, NY, USA, 129–140. https://doi.org/10.1145/3236024.3236041

[29] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14 (SSYM'05)*. USENIX Association, Berkeley, CA, USA, 18–18. http://dl.acm.org/citation.cfm?id=1251398.1251416

[30] Diego J. Romero López. 2019 (accessed August 16, 2019). DJBDD: Java BDD implementation based on hashmaps. https://github.com/diegojromerolopez/djbdd

[31] 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

[32] I. Mokhtarzada. 2019 (accessed August 16, 2019). A Simple Ray Tracer written in Java. https://github.com/idris/raytracer.

[33] Object Computing. 2019 (accessed August 16, 2019). Micronaut Framework. https://micronaut.io

[34] Oracle. 2019 (accessed August 15, 2019). GraalVM Native Image. https://www.graalvm.org/docs/reference-manual/aot-compilation/

[35] Oracle. 2019 (accessed August 15, 2019). GraalVM Native Image Source Code. https://github.com/oracle/graal/tree/master/substratevm

[36] Oracle. 2019 (accessed August 16, 2019). ConcurrentHashMap Implementation. http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/concurrent/ConcurrentHashMap.java

[37] Oracle. 2019 (accessed August 16, 2019). HashMap Implementation. http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/HashMap.java

[38] Oracle. 2019 (accessed August 16, 2019). Helidon Project: Java libraries crafted for Microservices. https://helidon.io/

[39] Oracle. 2019 (accessed October 31, 2019). The Java Tutorials: Type Erasure. https://docs.oracle.com/javase/tutorial/java/generics/erasure.html..

[40] F. Qian. [n.d.]. SableBDD: A Java Binary Decision Diagram Package. http://www.sable.mcgill.ca/~fqian/SableJBDD/.

[41] A. Quick. 2019 (accessed August 16, 2019). JTidy: HTML Parser and Pretty Printer in Java. http://jtidy.sourceforge.net/index.html.

[42] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 519–530. https://doi.org/10.1145/2491956.2462176

[43] RedHat. 2019 (accessed August 16, 2019). Quarkus: Supersonic Subatomic Java. https://quarkus.io

[44] Radu Rugina and Martin Rinard. 1999. Automatic Parallelization of Divide and Conquer Algorithms. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '99)*. ACM Press, New York, NY, USA, 72–83. https://doi.org/10.1145/301104.301111

[45] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 2002. Parametric Shape Analysis via 3-valued Logic. *ACM Trans. Program. Lang. Syst.* 24, 3 (May 2002), 217–298. https://doi.org/10.1145/514188.514190

[46] Luc Roger Semeria. 2001. *Applying Pointer Analysis to the Synthesis of Hardware from C*. Ph.D. Dissertation. Advisor(s) Micheli, Giovanni De. AAI3026902.

[47] M Sharir and A Pnueli. 1978. *Two approaches to interprocedural data flow analysis*. New York Univ. Comput. Sci. Dept., New York, NY. https://cds.cern.ch/record/120118

[48] 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

[49] Werner Stoop. 2019 (accessed August 16, 2019). Jatalog: Java Datalog Engine with Semi-Naive Evaluation and Stratified Negation. https://github.com/wernsey/Jatalog.

[50] 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.

[51] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata. *SIGPLAN Not.* 52, 6 (June 2017), 278–291. https://doi.org/10.1145/3140587.3062360

[52] The Apache Software Foundation. 2019 (accessed October 17, 2019). Apache Dubbo. https://github.com/apache/dubbo/blob/master/dubbo-filter/dubbo-filter-cache/src/main/java/org/apache/dubbo/cache/support/expiring/ExpiringMap.java

[53] The Apache Software Foundation. 2019 (accessed October 17, 2019). ElasticSearch. https://github.com/elastic/elasticsearch/blob/master/server/src/main/java/org/elasticsearch/common/collect/CopyOnWriteHashMap.java

[54] Arno Unkrig. 2019 (accessed August 16, 2019). Janino: A super-small, super-fast Java compiler. http://janino-compiler.github.io/janino/

[55] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize Once, Start Fast: Application Initialization at Build Time. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 184 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360610

[56] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. 2008. Scalable Shape Analysis for Systems Code. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV '08)*. Springer-Verlag, Berlin, Heidelberg, 385–398. https://doi.org/10.1007/978-3-540-70545-1_36

[57] Greta Yorsh, Eran Yahav, and Satish Chandra. 2008. Generating Precise and Concise Procedure Summaries. *SIGPLAN Not.* 43, 1 (Jan. 2008), 221–234. https://doi.org/10.1145/1328897.1328467

[58] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 2010. Level by Level: Making Flow- and Context-sensitive Pointer Analysis Scalable for Millions of Lines of Code. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10)*. ACM Press, New York, NY, USA, 218–229. https://doi.org/10.1145/1772954.1772985

[59] Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. 2014. Hybrid Top-down and Bottom-up Interprocedural Analysis. *SIGPLAN Not.* 49, 6 (June 2014), 249–258. https://doi.org/10.1145/2666356.2594328