Alias Analysis for Object-Oriented Programs

Manu Sridharan[1], Satish Chandra[1], Julian Dolby[1], Stephen J. Fink[1], and Eran Yahav[2]

[1] IBM T.J. Watson Research Center
[2] Technion
{msridhar,satishchandra,dolby,sjfink}@us.ibm.com,yahave@cs.technion.ac.il

Abstract. We present a high-level survey of state-of-the-art alias analyses for object-oriented programs, based on a years-long effort developing industrial-strength static analyses for Java. We first present common variants of points-to analysis, including a discussion of key implementation techniques. We then describe flow-sensitive techniques based on tracking of access paths, which can yield greater precision for certain clients. We also discuss how whole-program alias analysis has become less useful for modern Java programs, due to increasing use of reflection in libraries and frameworks. We have found that for real-world programs, an under-approximate alias analysis based on access-path tracking often provides the best results for a variety of practical clients.

1 Introduction

Effective analysis of pointer aliasing plays an essential role in nearly all non-trivial program analyses for object-oriented programs. For example, computing a precise inter-procedural control-flow graph, a necessity for many program analyses, often requires significant pointer reasoning to resolve virtual dispatch. Furthermore, any program analysis attempting to discover non-trivial properties of an object must reason about mutations to that object through pointer aliases.

Building alias analyses that simultaneously scale to realistic object-oriented programs and libraries while providing sufficient precision has been a longstanding challenge for the program analysis community. The twin goals of scalability and precision often conflict with each other, leading to subtle tradeoffs that make choosing the right alias analysis for a task non-obvious. Moreover, as large object-oriented frameworks (e.g., Eclipse[3] for desktop applications or Spring[4] for server-side code) have proliferated, achieving scalability and precision has become increasingly difficult.

In this work, we give a high-level survey of the alias-analysis techniques that we have found most useful during a years-long effort developing industrial-strength analyses for Java programs. We focus on two main techniques:

[3] http://www.eclipse.org
[4] http://www.springsource.org

1. *Points-to analysis*, specifically variants of Andersen's analysis [3] for Java. A points-to analysis result can be used to determine *may-alias* information, i.e., whether it is possible for two pointers to be aliased during program execution.
2. Flow-sensitive tracking of the *access paths* that name an object, where an access path is a variable and a (possibly empty) sequence of field names (see Section 5 for details). Access-path tracking enables determination of *must-alias* information, i.e., whether two pointers must be aliased at some program point.

We also aim to explain particular challenges we have encountered in building analyses that scale to modern Java programs. We have found that as standard libraries and frameworks have grown, difficulties in handling reflection have led us to reduce or eliminate our reliance on traditional points-to analysis. Instead, we have developed an under-approximate approach to alias analysis based on on type-based call graph construction and tracking of access paths. We have found this approach to be more effective for analyzing large Java programs, though traditional points-to analysis remains relevant in other scenarios. Our experiences may shed light on issues in designing analyses for other languages, and in designing future languages to be more analyzable.

This chapter is *not* intended to be an exhaustive survey of alias analysis. Over the past few decades, computer scientists have published hundreds of papers on alias-analysis techniques. The techniques vary widely depending on myriad analysis details, such as policies for flow sensitivity, context sensitivity, demand-driven computation, and optimization tradeoffs. We cannot hope to adequately cover this vast space, and the literature grows each year. Here we focus on alias analyses that we have significant experience implementing and applying to real programs. To the best of our knowledge, the presented alias analyses are the state-of-the-art for our desired analysis clients and target programs. For many of the analyses described here, a corresponding implementation is available as part of the open-source Watson Libraries for Analysis (WALA) [69].

Organization This chapter is organized as follows. In Section 2, we motivate the alias-analysis problem by showing the importance of precise aliasing information for analysis clients. Then, we discuss points-to analysis for Java-like languages: Section 3 gives formulations of several variants of Andersen's analysis [3], and Section 4 discusses key implementation techniques. Section 5 discusses must-alias analysis based on access-path tracking, which provides greater precision than a typical points-to analysis. In Section 6, we describe challenges in applying points-to analysis to modern Java programs and how under-approximate techniques can be used instead. Finally, Section 7 concludes and suggests directions for future work. Some of the material presented here has appeared in previous work by the authors [20, 62, 67].

2 Motivating Analyses

Many program analyses for object-oriented languages rely on an effective alias analysis. Here we illustrate a number of alias analysis concerns in the context of an analysis for detecting resource leaks in Java programs, and discuss how these concerns also pertain to other analyses.

2.1 Resource Leaks

```
1 public void test(File file, String enc) throws IOException {
2   PrintWriter out = null;
3   try {
4     try {
5       out = new PrintWriter(
6               new OutputStreamWriter(
7                   new FileOutputStream(file), enc));
8     } catch (UnsupportedEncodingException ue) {
9       out = new PrintWriter(new FileWriter(file));
10    }
11    out.append('c');
12  } catch (IOException e) {
13  } finally {
14    if (out != null) {
15      out.close();
16    }
17  }
18 }
```

Fig. 1. Example of a resource leak.

While garbage collection frees the programmer from the responsibility of memory management, it does not help with the management of finite *system resources*, such as sockets or database connections. When a program written in a Java-like language acquires an instance of a finite system resource, it must release that instance by explicitly calling a dispose or close method. Letting the last handle to an unreleased resource go out of scope *leaks* the resource. Leaks can gradually deplete the finite supply of system resources, leading to performance degradation and system crashes. Ensuring that resources are always released, however, is tricky and error-prone.

As an example, consider the Java program in Fig. 1, adapted from code in Apache Ant.[5] The allocation of a `FileOutputStream` on line 7 acquires a stream, which is a system resource that needs to be released by calling `close()` on the stream handle. The acquired stream object then passes

[5] `http://ant.apache.org`

into the constructor of `OutputStreamWriter`, which remembers it in a private field. The `OutputStreamWriter` object, in turn, passes into the constructor of `PrintWriter`. In the `finally` block, the programmer calls `close()` on the `PrintWriter` object. This `close()` method calls `close()` on the "nested" `OutputStreamWriter` object, which in turn calls `close()` on the nested `FileOutputStream` object. By using `finally`, it would appear that the program closes the stream, even in the event of an exception.

However, a potential resource leak lurks in this code. The constructor of `OutputStreamWriter` might throw an exception: notice that the programmer anticipates the possibility that an `UnsupportedEncodingException` may occur. If it does, the assignment to the variable `out` on line 5 will not execute, and consequently the stream allocated on line 7 is never closed. A *resource leak analysis* aims to statically detect leaks like this one.

2.2 Role of alias analysis

A resource leak analysis should report a potential leak of the stream allocated at line 7 of Fig. 1. But a bug finding client should not *trivially* report all resource acquisitions as potentially leaking, as that would generate too many false positives. Hence, the key challenge of resource leak analysis is in reasoning that a resource in fact does *not* leak, and it is this reasoning that requires effective alias analysis. Here we will consider what it takes to prove that the resource allocated at line 7 does *not* leak along the exception-free path.

Figure 2 shows the relevant parts of the CFG for Fig. 1, along with the CFGs of some of the called methods. We introduced temporary variables `t1` and `t2` when constructing the CFG (a).

Consider the program path 7-6-5-11-14-15 (the exception-free path), starting with the resource allocation on line 7. The constructor on line 6 stores its argument into an instance field `a`; see CFG (b). Likewise, the constructor on line 5 stores its argument into an instance field `b`; see CFG (c). The call `out.close()` on line 15 transitively calls `close()` on expressions `out.b` and `out.b.a` (notice that `this` in CFGs (d) and (e) would be bound appropriately), the last one releasing the tracked resource as it is equal to `t1`. At this point, the (same) resource referred to by the expressions `t1`, `t2.a`, and `out.b.a` is released.

What reasoning is needed for an analysis to prove that the resource allocated on line 7 is definitely released along the path 7-6-5-11-14-15?

1. Data flow must be tracked inter-procedurally, as the call to the `close()` method of the `FileOutputStream` occurs in a callee. For this reason, an accurate *call graph* must be built.
2. The analysis must establish that `this.a` in CFG (e), when encountered along this path *must* refer to the same object assigned to `t1` in CFG (a).

Both cases demand effective alias analysis.

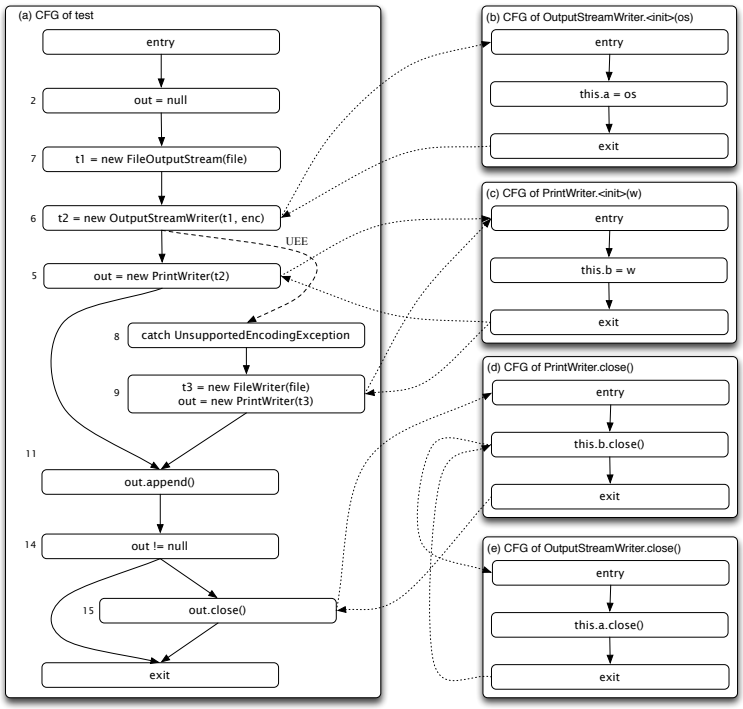

Fig. 2. Control-flow graph of the procedure shown in Fig. 1. Numbers to the left are line numbers. Dotted edges represent inter-procedural control transfers.

Call Graph Construction A *call graph* indicates the possibly-invoked methods at each call site in a program. In object-oriented languages, virtual calls make building a call graph non-trivial. Consider CFG (d). Say the field `b` in class `PrintWriter` has declared type `Writer`, which has a number of different subtypes, one of which is `OutputStreamWriter`. A call graph based solely on the program's class hierarchy (a *class hierarchy analysis* [13]) allows implementations of `close()` in all possible `Writer` subtypes to be potential targets of the call `this.b.close()`. In principle, some subtype of `Writer` other than `OutputStreamWriter` could implement `close()` in a way that does not close the resource, causing a false positive here.[6]

A *points-to analysis* (see Section 3) creates an over-approximation of all the heap values that can possibly flow into each reference by tracing data flow through assignments. In this program, the only heap values that flow into field

[6] Rapid type analysis (RTA) [4], which only considers allocated types, is generally more accurate than class hierarchy analysis [66], but it would still cause a false positive if the bad `Writer` subtype were allocated anywhere in the program. We have found that the difference between RTA and class hierarchy analysis tends to vanish in large framework-dependent programs.

`b` of `PrintWriter` are of type `OutputStreamWriter` (via the call at line 5) or `FileWriter` (via line 9). The class `FileWriter` inherits its `close()` method from `OutputStreamWriter`. Thus, a call graph based on points-to analysis can correctly narrow down the call target of `this.b.close()` to the `close()` method in the class `OutputStreamWriter`, as shown in edges from CFG (d) to CFG (e), yielding greater precision in the resource leak analysis.

Note that call graph construction and alias analysis are often inter-dependent. In our example, the assignment to the `b` field of `PrintWriter` occurs in a callee (the constructor `<init>`) via the calls at lines 5 and 9. Hence, the points-to analysis must know that this `<init>` method is in the call graph to properly trace the flow of an `OutputStreamWriter` into the `b` field, which in turn implies that `OutputStreamWriter.close()` can be called from in CFG (d). Multiple approaches exist to address this inter-dependency, to be discussed in Section 3.

Equality Recall that to prove leak freedom for the path of interest, the analysis needs to show that the object referenced by `this.a` in CFG (e) refers to the same object pointed by `t1` is CFG (a), to ensure that the resource is released. However, this fact cannot be proved using points-to analysis alone, as it only provides *may-alias* information, i.e., it can only state that `this.a` *may* refer to the same object as `t1`. Given may-alias information alone, the analysis must consider a case where `this.a` does not alias `t1` on the path, and a false leak will be reported.

To avoid this false report, *must-alias* information is needed, indicating alias relationships that *must* hold at a program point. This reasoning can be accomplished by tracking *must access paths* naming the resource as part of the resource-leak analysis. Much as in the informal reasoning described previously, must access paths are expressions of the form `t1`, `t2.a`, and `out.b.a`, with the property that in the current program state, they must equal the tracked resource. By tracking access paths along the control-flow path of interest, the equality of `this.a` and `t1` in our example can be established, and the false leak report is avoided. As we shall show in Section 5, access-path tracking can provide useful aliasing information for a number of important client analyses.

2.3 Other Analysis Clients

Many client analyses share some or all of the alias analysis needs shown for the resource leak analysis above. Any static analysis performing significant reasoning across procedure boundaries (quite a large set) is likely to benefit from a precise call graph produced via alias analysis, due to the pervasiveness of method calls and virtual dispatch in Java-like languages. Other analyses can be rather directly formulated in terms of possible heap data flow and aliasing, for example, static race detection [43] and taint analysis [68]. Access-path tracking is most often used for analyses that need to track changing properties of objects, like resource leak analysis or typestate verification [12, 20], but other analyses may also benefit from the additional precision.

3 Formulating Points-To Analysis

Here we formulate several common variants of Andersen's points-to analysis [3] for Java-like object-oriented languages; implementation techniques will be discussed in Section 4. We begin with a standard formulation of context-insensitive Andersen's analysis that captures its essential points. Then, we extend the formulation with a generic template for context sensitivity, and we present various context-sensitive analyses in the literature as instantiations of the template.

3.1 Context-Insensitive Formulation

A *points-to analysis* computes an over-approximation of the heap locations that each program pointer may point to. Pointers include program variables and also pointers within heap-allocated objects, e.g., instance fields. The result of the analysis is a *points-to relation* pt, with $pt(p)$ representing the *points-to set* of a pointer p. For decidability and scalability, points-to analyses must employ abstraction to finitize the possibly-infinite set of pointers and heap locations arising at runtime. In particular, a *heap abstraction* represents dynamic heap locations with a finite set of *abstract locations*.

Andersen's points-to analysis [3] has the following properties:

- *Flow insensitive*: The analysis assumes statements can execute in any order and any number of times.
- *Subset based*: The analysis models directionality of assignments, i.e., a statement `x = y` implies $pt(y) \subseteq pt(x)$. In contrast, an equality-based analysis (e.g., that of Steensgaard [65]) would require $pt(y) = pt(x)$ for the same statement, a coarser approximation.

As is typical for Java points-to analyses, we also desire *field sensitivity*, which requires separate reasoning about each instance field of each abstract location. *Field-based* analyses for Java, in which instance field values are merged across abstract locations, may provide sufficient precision for certain clients [32, 64]. However, field sensitivity typically adds little expense to a context-insensitive analysis [32], and for context-sensitive analyses (to be discussed in Section 3.2), field sensitivity is essential for precision.

Table 1 gives a standard formulation of context-insensitive, field-sensitive Andersen's analysis for Java, equivalent to those appearing elsewhere in the literature [32, 52, 64, 72].[7] Canonical statements for the analysis are given in the first column. In order, the four statement types enable object allocation, copying pointers, and reading and writing instance fields. More complex memory-access statements (e.g., `x.f = y.g.h`) are handled through suitable introduction of

[7] Points-to analysis has also been formulated as an abstract interpretation [11] (e.g., by Might et al. [41]), yielding a systematic characterization of the analysis result in terms of the target program's concrete semantics. See Might et al. [41] for details of such a formulation and a discussion the relationship of context-sensitive points-to analysis to control-flow analysis for functional languages [56].

Statement	Constraint	
`i: x = new T()`	$\{o_i\} \subseteq pt(x)$	[New]
`x = y`	$pt(y) \subseteq pt(x)$	[Assign]
`x = y.f`	$\dfrac{o_i \in pt(y)}{pt(o_i.\mathtt{f}) \subseteq pt(x)}$	[Load]
`x.f = y`	$\dfrac{o_i \in pt(x)}{pt(y) \subseteq pt(o_i.\mathtt{f})}$	[Store]

Table 1. Canonical statements for context-insensitive Java points-to analysis and the corresponding points-to set constraints.

temporary variables. Array objects are modeled as having a single field `arr` that may point to any value stored in the array (so, `x[i] = y` is modeled as `x.arr = y`). Section 3.2 discusses handling of method calls.

The inference rules in Table 1 describe how each statement type affects the corresponding points-to sets. Note that since the analysis is field sensitive, points-to sets are maintained both for variables (e.g., $pt(x)$) and for instance fields of abstract locations (e.g., $pt(o_i.\mathtt{f})$). Also note that in the New rule, the abstract location o_i is named based on the statement label `i`, the standard heap abstraction used in Andersen's analysis.

Example Consider the following program (assume type `T` has a field `f`):

```
1 a = new T();
2 b = new T();
3 a.f = b;
4 c = a.f;
```

The following is a derivation of of $o_2 \in pt(c)$ according to the rules of Table 1, with rule applications labeled by the corresponding program statement's line number:

$$\dfrac{\dfrac{\dfrac{}{o_2 \in pt(b)}\,\text{L2} \quad \dfrac{\dfrac{}{o_1 \in pt(a)}\,\text{L1}}{pt(b) \subseteq pt(o_1.\mathtt{f})}\,\text{L3}}{o_2 \in pt(o_1.\mathtt{f})} \quad \dfrac{\dfrac{}{o_1 \in pt(a)}\,\text{L1}}{pt(o_1.f) \subseteq pt(c)}\,\text{L4}}{o_2 \in pt(c)}$$

3.2 Context Sensitivity

We now extend our points-to analysis formulation to incorporate context-sensitive handling of method calls. We formulate context sensitivity in a generic manner

and then show how to instantiate the formulation to derive standard analysis variants.

A *context-sensitive* points-to analysis separately analyzes a method m for each *calling context* that arises at call sites of m. A calling context (or, simply, a *context*) is some abstraction of the program states that may arise at a call site. Separately analyzing a method for each context removes imprecision due to conflation of analysis results across its invocations.

For example, consider the following program:

```
1 id(p) { return p; }
2 x = new Object(); // o1
3 y = new Object(); // o2
4 a = id(x);
5 b = id(y);
```

A context-*insensitive* analysis conflates the effects of all calls to `id`, in effect assuming that either object `o1` or `o2` may be passed as the parameter at the calls on lines 4 and 5. This assumption leads to the imprecise conclusions that `a` may point to `o2` and `b` to `o1`. Now, consider a context-sensitive points-to analysis that uses a distinct context for each method call site. This analysis will process `id` separately for its two call sites, thereby precisely concluding that `a` may only point to `o1` and `b` only to `o2`.

Formulation Our generic formulation of context-sensitive points-to analysis appears in Table 2. Compared to Table 1, the two additional statement types respectively allow for invoking and returning from procedures. We assume that a method m has formal parameters m_{this} for the receiver and $m_{p_1}, \ldots, m_{p_n}$ for the remaining parameters, and we use a pseudo-variable m_{ret} to hold its return value.[8]

The analysis formulated in Table 2 maintains a set $contexts(m)$ of the contexts that have arisen at call sites of each method m. For each local pointer variable x, the analysis maintains a separate abstract pointer $\langle x, c\rangle$ to represent x's possible values when its enclosing method is invoked in context c. Abstract locations $\langle o_i, c\rangle$ are similarly parameterized by a context. Finally, note that each constraint in the second column of Table 1 is written under the assumption that the corresponding statement in the first column is from method m.

Our formulation is parameterized by two key functions, which together specify a *context-sensitivity policy*:

- The *selector* function, which determines what context to use for a callee at some call site, and
- The *heapSelector* function, which determines what context c to use in an abstract location $\langle o_i, c\rangle$ at allocation site i.

[8] We elide static fields (global variables) and static methods from our formulation, as their handling is straightforward. Since method contexts cannot be applied to global variables, their usage may blunt precision gains from context sensitivity.

Statement in method m	Constraint	
`i: x = new T()`	$\dfrac{c \in contexts(m)}{\langle o_i, heapSelector(c)\rangle \in pt(\langle x, c\rangle)}$	[New]
`x = y`	$\dfrac{c \in contexts(m)}{pt(\langle y, c\rangle) \subseteq pt(\langle x, c\rangle)}$	[Assign]
`x = y.f`	$\dfrac{c \in contexts(m) \qquad \langle o_i, c'\rangle \in pt(\langle y, c\rangle)}{pt(\langle o_i, c'\rangle.\mathtt{f}) \subseteq pt(\langle x, c\rangle)}$	[Load]
`x.f = y`	$\dfrac{c \in contexts(m) \qquad \langle o_i, c'\rangle \in pt(\langle x, c\rangle)}{pt(\langle y, c\rangle) \subseteq pt(\langle o_i, c'\rangle.\mathtt{f})}$	[Store]
`j: x = r.g(a_1,...,a_n)`	$\dfrac{\begin{array}{c} c \in contexts(m) \qquad \langle o_i, c'\rangle \in pt(\langle r, c\rangle) \\ m' = dispatch(\langle o_i, c'\rangle, \mathtt{g}) \\ argvals = [\{\langle o_i, c'\rangle\}, pt(\langle a_1, c\rangle), \ldots, pt(\langle a_n, c\rangle)] \\ c'' \in selector(m', c, j, argvals) \end{array}}{\begin{array}{c} c'' \in contexts(m') \\ \langle o_i, c'\rangle \in pt(\langle m'_{this}, c''\rangle) \\ pt(\langle a_k, c\rangle) \subseteq pt(\langle m'_{p_k}, c''\rangle),\ 1 \le k \le n \\ pt(\langle m'_{ret}, c''\rangle) \subseteq pt(\langle x, c\rangle) \end{array}}$	[Invoke]
`return x`	$\dfrac{c \in contexts(m)}{pt(\langle x, c\rangle) \subseteq pt(\langle m_{ret}, c\rangle)}$	[Return]

Table 2. Inference rules for context-sensitive points-to analysis.

We first present the inference rules for the analysis without specifying these functions. Then, we show how standard variants of context-sensitive points-to analysis can be expressed by instantiating *selector* and *heapSelector* appropriately.

Inference Rules For the first four statement types, the inference rules in Table 2 are modified from those in Table 1 to include appropriate parameterization with contexts. In each rule, a pre-condition chooses a context c from those that have been created for the enclosing method m, and c is used in the rule's conclusions. The *heapSelector* function from the context-sensitivity policy is used in the New rule to obtain contexts for abstract locations. The final Return rule in Table 2 models `return` statements by simulating a copy from the returned variable x to the pseudo-variable m_{ret} for the method. So, $pt(\langle m_{ret}, c\rangle)$ will include all abstract objects possibly returned by m in context c.

By far, the Invoke rule is the most complex. The first two lines of the rule model reasoning about virtual dispatch. Given a location $\langle o_i, c'\rangle$ that the receiver argument $\langle r, c\rangle$ may point to, a *dispatch* function is invoked to resolve the virtual

dispatch of `g` on $\langle o_i, c' \rangle$ to a target method m'. (For Java, *dispatch* would be implemented based on the type hierarchy and the concrete type of $\langle o_i, c' \rangle$.) This direct reasoning about virtual dispatch implies that the analysis computes its call graph *on-the-fly* [32, 52, 72], rather than relying on a call graph computed with some less precise analysis (recall the inter-dependence of points-to analysis and call graph construction, discussed in Section 2). The tradeoffs between on-the-fly call graph construction and using a pre-computed call graph have been explored extensively in the literature [21, 32]; we have found that on-the-fly call graph construction usually improves both precision *and* performance (see Section 4.4).

Once the target m' of the virtual call is discovered, the analysis uses the *selector* function from the context-sensitivity policy to determine which context(s) to use for this call of m'. *selector* can discriminate contexts for the target method m' based on the caller's context c, the call site id j, and a list of possible parameter values *argvals*. Given a context c'' returned by *selector*, the first conclusion of the INVOKE rule ensures that c'' is in the set of observed contexts for m'. The final three conclusions of the rule model parameter passing and return-value copying for the call.

Entrypoints Points-to analyses with on-the-fly call graph construction must be provided with a set of *entrypoint* methods E that may be invoked by the environment to begin execution (e.g., a `main` method for a standard Java program); these methods are assumed to be reachable by the analysis. Given E, the *contexts* sets referenced in Table 1 should be initialized as follows:

- If $m \in E$, then $\mathit{contexts}(m) = \{\texttt{Default}\}$, where `Default` is a special dummy context value.
- If $m \notin E$, then $\mathit{contexts}(m) = \emptyset$.

With these initial conditions, results will only be computed for methods deemed reachable by the analysis itself, as desired.

Note that an entrypoint method may rely on initialization being performed before it is invoked, e.g., the creation of the `String[]` array parameter for a `main` method. In WALA [69], such behavior is modeled in a synthetic "fake root method" that serves as a single root for the call graph and contains invocations of the real entrypoints. The fake root method includes code to pass objects to entrypoint parameters based on customizable heuristics (e.g., passing an object whose concrete type matches the parameter's declared type). In general, precisely modeling how an environment initializes objects before executing an entrypoint can be quite difficult (e.g., for framework-based applications [59]), and this modeling can be critical to getting useful results from a points-to analysis.

Context Sensitivity Variants In this section, we discuss several standard variants of context-sensitive Andersen's-style points-to analysis, and we show how the analyses can be expressed by instantiating the *selector* and *heapSelector* functions used in Table 2. Note that a context-*insensitive* analysis (with on-the-fly call graph construction) can be expressed using the dummy `Default` context

(we use '_' for an unused argument):

$$\begin{aligned} \mathit{selector}(\_,\_,\_,\_) &= \{\texttt{Default}\} \\ \mathit{heapSelector}(\_) &= \texttt{Default} \end{aligned}$$

Call Strings A standard technique to distinguish contexts is via *call strings* [55], which abstract the possible call stacks under which a method may be invoked. Call strings are typically represented as a sequence of call site identifiers, corresponding to a (partial) call stack. The following *selector* function gives a call-string-sensitive context for a callee at site j, given the caller context $[j_0, \ldots, j_n]$:

$$\mathit{selector}(\_, [j_0, j_1, \ldots, j_n], j, \_) = \{[j, j_0, j_1, \ldots, j_n]\} \tag{1}$$

For full precision, the *heapSelector* function should simply re-use the contexts provided by *selector*, i.e., $\mathit{heapSelector}(c) = c$. This choice of *heapSelector* yields a *context-sensitive heap abstraction*.

As a simple example, consider the following program.

```
1 Object f1(T x) { return x.f; }
2 Object f2(T x) { return f1(x); }
3 ...
4 p = f2(q);
5 r = f2(s);
```

Given *selector* as defined above, method `f2()` will be analyzed in contexts $[s_4]$ and $[s_5]$ (we write s_i for the call site on line i), and `f1()` will be analyzed in contexts $[s_2, s_4]$ and $[s_2, s_5]$.

Unfortunately, the naïve *selector* function above could cause non-termination in the presence of method recursion, as call strings may grow without bound. Even without recursion, analysis time grows exponentially in the number of methods in the worst case, as the worst-case number of paths in a program's call graph is exponential in the number of methods. In practice, over 10^{14} possible call strings have been observed for a medium-sized program [72], making straightforward use of long call strings intractable.

A standard method for improving scalability of the call-string approach in practice is k-limiting [55], where the maximum call-string length is bounded by a small constant k. This approach has been employed in various previous systems, though the consensus seems to be that bounded object sensitivity (discussed below) provides greater precision for the same or less cost [34]. Instead of using k-limiting in *selector*, Whaley and Lam [72] achieve scalability by using compact BDD data structures (see Section 4) and a context-*insensitive* heap abstraction, i.e., with $\mathit{heapSelector}(c) = \texttt{Default}$. (In essence, k-limiting is performed in the heap selector, with $k = 0$.) While the scalability of their analysis was impressive, later work showed that its precision was lacking for typical clients due to the coarse heap abstraction [34].

For certain classes of program analyses, a result equivalent to using arbitrary-length call strings can be computed efficiently, for example, so-called IFDS prob-

lems [51].[9] For this level of precision, the analysis result is typically computed using a summary-based approach [51, 55] that is not directly expressible in the formulation of Table 2. However, Reps has shown that full context sensitivity for a field-sensitive analysis is undecidable [50]. While summary-based points-to analyses have been developed [73, 74], we are unaware of any such analysis that scales to large Java programs.

Object Sensitivity Rather than distinguishing a method's invocations based on call strings, an *object-sensitive* analysis [42][10] uses the (abstract) objects passed as the receiver argument to the method. The intuition behind object sensitivity is that in typical object-oriented design, the state of an object is accessed or mutated via its instance methods (e.g., "setter" and "getter" methods for instance fields). Hence, by using receiver objects to distinguish contexts, an object-sensitive analysis can avoid conflation of operations performed on distinct objects.

In terms of our Table 2 formulation, object-sensitive analysis and more recent variants [58] can be expressed via the following *selector* function:

$$selector(\_, \_, \_, argvals) = \bigcup_{\langle o,c\rangle \in argvals[0]} locToContext(\langle o, c\rangle) \tag{2}$$

locToContext converts an abstract location (which includes context information) into a context. For standard object sensitivity [42],[11] a context is a list of allocation sites, and *locToContext* simply adds to that list:

$$locToContext(\langle o_i, l\rangle) = cons(o_i, l) \tag{3}$$

(As in Lisp, $cons(o_i, [o_1, o_2, \ldots]) = [o_i, o_1, o_2, \ldots]$.) As discussed previously, using $heapSelector(c) = c$ yields a context-sensitive heap abstraction.

To illustrate object sensitivity, consider the following example:

```
1 class A { B makeB() { return new B(); } }
2 class B { Object makeObj() { return new Object(); } }
3 ...
4 A a1 = new A();
5 A a2 = new A();
6 B b1 = a1.makeB();
7 B b2 = a2.makeB();
8 Object p1 = b1.makeObj();
9 Object p2 = b2.makeObj();
```

[9] In the literature, an analysis computing such a result is often termed "context-sensitive," but we avoid that usage, as we consider contexts other than call strings.

[10] While Milanova's work [42] introduced the term "object sensitivity," similar ideas were employed in earlier work on object-oriented type inference [1, 45].

[11] While alternate object-sensitivity definitions have appeared [35], Smaragdakis et al. [58] showed that Milanova's definition [42] is most effective.

With object-sensitive analysis defined by the *selector* and *heapSelector* function above, `makeB()` will be analyzed in contexts $[o_4]$ and $[o_5]$ (with abstract objects labeled by allocating line number), and we have $pt(b_1) = \{\langle o_1, [o_4]\rangle\}$ and $pt(b_2) = \{\langle o_1, [o_5]\rangle\}$ due to the context-sensitive heap abstraction. Similarly, `makeObj()` is analyzed in contexts $[o_1, o_4]$ and $[o_1, o_5]$, $pt(p_1) = \{\langle o_2, [o_1, o_4]\rangle\}$, and $pt(p_2) = \{\langle o_2, [o_1, o_5]\rangle\}$.

As with call-string sensitivity, k-limiting, either in *selector* or *heapSelector*, is necessary to achieve scalability to realistic programs. The literature contains inconsistent definitions of what exactly it means to limit object-sensitive contexts with a particular value of k; see Smaragdakis et al. [58] for an extended discussion.

In general, the precision of an object-sensitive analysis is incomparable to that of a call-string-sensitive analysis [42]. Object sensitivity can lose precision compared to call-string sensitivity by merging across call sites that pass the same receiver object, but it may gain precision by using multiple contexts at a single call site (when multiple receiver objects are possible). Work by Lhoták and Hendren [34] has shown that for small values of k, object-sensitive analysis yields more precise results for common clients than a call-string-sensitive analysis. In practice, a mix of object- and call-string sensitivity is often used, e.g., with call-string sensitivity being employed only for static methods (which have no receiver argument).

Recently, Smaragdakis et al. [58] have identified *type sensitivity* as a useful technique for obtaining much of the precision of object sensitivity with greater scalability. In one variant of type sensitivity, a context is a list of types rather than allocation sites, and *locToContext* converts the allocation site from the abstract location into a type:

$$locToContext(\langle o_i, l\rangle) = cons(enclosingClass(o_i), l) \tag{4}$$

Rather than using the concrete type of o_i in the context, the concrete type of the enclosing class for the allocation site is used, yielding greater precision in practice [58]. Another variant of type sensitivity allows for one abstract location to remain in the context:

$$locToContext(\langle o_i, cons(o_j, l)\rangle) = cons(o_i, cons(enclosingClass(o_j), l)) \tag{5}$$

Again, k-limiting is required for scalability of either of these schemes. Smaragdakis et al. [58] show how k-limited versions of these type-sensitive analyses provide much of the precision of standard object-sensitive analysis with significantly less cost.

Rather than limiting attention to the receiver argument, the *cartesian product algorithm* (CPA) [1] distinguishes contexts based on the objects passed in all

argument positions. We can define the *selector* function for CPA as follows:

$$cartProd(argvals) = \prod_{i=0}^{n} argvals[i]$$

$$selector(\_, \_, \_, argvals) = \bigcup_{l \in cartProd(argvals)} locListToContext(l) \quad (6)$$

The *cartProd* function computes the (generalized) cartesian product of all entries in *argvals*, yielding a set of lists of abstract locations. Each such list l is converted to a context using *locListToContext*, analogous to the use of *locToContext* for object sensitivity. In fact, the object-sensitive analysis variants described above can also be formulated as a special case of CPA, by only using values for the receiver argument in *locListToContext*. The *locToContext* used in Equation 3 for standard object sensitivity can be generalized to handle all argument positions:

$$locListToContext([\langle o_{i_0}, c_0\rangle, \ldots, \langle o_{i_n}, c_n\rangle]) = (cons(o_{i_0}, c_0), \ldots, cons(o_{i_n}, c_n)) \quad (7)$$

As formulated above, CPA creates many more contexts per method than the equivalent object-sensitive analysis, a significant scalability barrier. In its original formulation [1], CPA was used for type inference, and the heap abstraction consisted of types rather than allocation sites, making scalability more feasible. While full CPA based on allocation sites may not scale, we believe that contexts based on arguments other than the receiver may still prove useful.

Unification-Based Approaches Some previous approaches to context-sensitive points-to analysis have employed equality constraints for assignments [16, 31, 44], which cannot be expressed as a context-sensitivity policy in the formulation of Table 2.[12] In this approach, statement `x = y` is modeled with constraint $pt(y) = pt(x)$ instead of $pt(y) \subseteq pt(x)$, enabling the use of fast union-find data structures to represent equal points-to sets. While this approach has been shown to scale for C++ programs [31], we are unaware of a scalable implementation for Java-like languages. In particular, the increased use of virtual dispatch in Java negatively affects the scalability of the equality-based approach [44].

4 Implementing Points-To Analysis

Here we present techniques for efficiently implementing the points-to analyses formulated in Section 3. Over the past two decades, advances in implementation techniques (and hardware advances) have shown that some of these variants can scale to relatively large programs (papers reporting analysis of millions of lines of code are now commonplace). We present basic techniques for implementing an

[12] For languages like Java and C#, a context-*insensitive* equality-based approach like Steensgaard's analysis [65] does not work—since all objects are passed as the `this` parameter to the constructor of the root object type, the analysis would conclude that all points-to sets are equal.

Andersen's-style analysis, and then briefly review some of the most prominent advanced techniques which have appeared in the literature.

Unfortunately, the pointer analysis literature contains several different formalisms for describing analyses and implementations. Presentations use various mathematical frameworks, including set constraints [17], context-free-language reachability [49], and Datalog [72]. Each framework elucidates certain issues most clearly, and the choice of framework depends on the best match between the input language, the analysis variant, and the author's taste.

This section discusses implementation techniques based on old-fashioned algorithmic description of imperative code based on fixed-point iteration. A previous paper [62] presented an algorithmic analysis of this algorithm, which sheds some light on the performance issues which arise in practice. We restate some of the key points from that work [62] here. The WALA pointer analysis implementation [69] follows this algorithm directly.

4.1 Algorithm

Here we present an algorithm for Andersen's analysis for Java, as specified in Table 1 in Section 3. The algorithm is most similar to Pearce et al.'s algorithm for C [47] and also resembles existing algorithms for Java (e.g., that of Lhoták and Hendren [32]). We do not give detailed pseudocode for implementing a context-sensitive analysis with on-the-fly call-graph construction, as formulated in Table 2, but we discuss some of the key implementation issues later in the section.

The algorithm constructs a flow graph G representing the pointer flow for a program and computes its (partial) transitive closure, a standard points-to analysis technique (e.g., see [17, 25, 27]). G has nodes for variables, abstract locations, and fields of abstract locations. At algorithm termination, G has an edge $n \to n'$ iff one of the following two conditions holds:

1. n is an abstract location o_i representing a statement `x = new T()`, and n' is `x`.
2. $pt(n) \subseteq pt(n')$ according to some rule in Table 1.

Given a graph G satisfying these conditions, it is clear that $o_i \in pt(x)$ iff x is reachable from o_i in G. Hence, the transitive closure of G—where only abstract location nodes are considered sources—yields the desired points-to analysis result. Since flow relationships for abstract-location fields depend on the points-to sets of base pointers for the corresponding field accesses (see the LOAD and STORE rules referencing $pt(o_i.f)$ in Table 1), certain edges in G can only be inserted after some reachability has been determined, yielding a *dynamic transitive closure* (DTC) problem.

Pseudocode for the analysis algorithm appears in Figure 3. The DOANALYSIS routine takes a set of program statements of the forms shown in Table 1 as input. (We assume suitable data structures that, given a variable `x`, yield all load statements `y = x.f` and store statements `x.f = y` in constant time per statement.) The algorithm maintains a flow graph G as just described and computes

```
DoAnalysis()
 1  for each statement i: x = new T() do
 2          pt_Δ(x) ← pt_Δ(x) ∪ {o_i}, o_i fresh
 3          add x to worklist
 4  for each statement x = y do
 5          add edge y → x to G
 6  while worklist ≠ ∅ do
 7          remove n from worklist
 8          for each edge n → n' ∈ G do
 9                  DiffProp(pt_Δ(n), n')
10          if n represents a local x
11            then for each statement x.f = y do
12                                  for each o_i ∈ pt_Δ(n) do
13                                          if y → o_i.f ∉ G
14                                            then add edge y → o_i.f to G
15                                                 DiffProp(pt(y), o_i.f)
16                  for each statement y = x.f do
17                                  for each o_i ∈ pt_Δ(n) do
18                                          if o_i.f → y ∉ G
19                                            then add edge o_i.f → y to G
20                                                 DiffProp(pt(o_i.f), y)
21          pt(n) ← pt(n) ∪ pt_Δ(n)
22          pt_Δ(n) ← ∅

DiffProp(srcSet, n)
1  pt_Δ(n) ← pt_Δ(n) ∪ (srcSet − pt(n))
2  if pt_Δ(n) changed then add n to worklist
```

Fig. 3. Pseudocode for the points-to analysis algorithm.

a points-to set $pt(x)$ for each variable x, representing the transitive closure in G from abstract locations. Note that abstract location nodes are eschewed, and instead the relevant points-to sets are initialized appropriately (line 2).

The algorithm employs *difference propagation* [18, 32, 47] to reduce the work of propagating reachability facts. For each node n in G, $pt_\Delta(n)$ holds those abstract locations o_i such that (1) the algorithm has discovered that n is reachable from o_i and (2) this reachability information has not yet propagated to n's successors in G. $pt(n)$ holds those abstract locations for which (1) holds and propagation to successors of n is complete. The DiffProp routine updates a difference set $pt_\Delta(n)$ with those values from *srcSet* not already contained in $pt(n)$. After a node n has been removed from the worklist and processed, all current reachability information has been propagated to n's successors, so $pt_\Delta(n)$ is added to $pt(n)$ and emptied (lines 21 and 22).

Theorem 1 DoAnalysis *terminates and computes the points-to analysis result specified in Table 1.*

Proof. (*Sketch*) DoAnalysis terminates since (1) the constructed graph is finite and (2) a node n is only added to the worklist when $pt_\Delta(n)$ changes (line 2 of DiffProp), which can only occur a finite number of times. For the most part, the correspondence of the computed result to the rules of Table 1 is straightforward. One subtlety is the handling of the addition of new graph edges due to field accesses. When an edge $y \rightarrow o_i.f$ is added to G to handle a putfield statement (line 14), only $pt(y)$ is propagated across the edge, not $pt_\Delta(y)$ (line 15). This operation is correct because if $pt_\Delta(y) \neq \emptyset$, then y must be on the worklist, and hence $pt_\Delta(y)$ will be propagated across the edge when y is removed from the worklist. A similar argument holds for the propagation of $pt(o_i.f)$ at line 20. □

4.2 Complexity

A simple algorithmic analysis shows that the algorithm in Figure 3 has worst-case cubic complexity. Note that difference propagation is required to ensure the cubic complexity bound for this worklist-style algorithm [46].

In practice, many papers have reported scaling behavior significantly better than cubic. Two of the authors have published an analysis [62] that explains why this pointer analysis usually runs in quadratic time on strongly-typed languages such as Java. The key insight is that Java's strong type system restricts the structure of the graph G to be relatively sparse for most pointer assignments. By bounding the sparsity of this graph, we can show that the algorithm usually runs in quadratic time. We refer the reader to the previous paper [62] for more details.

The previous work [62] also compares the expected behavior with the observed behavior in the WALA pointer analysis implementation. The paper reports results that show that the WALA implementation scales roughly quadratically with program size on Java programs, as predicted. As a rough characterization of overall scalability, [62] reports that the WALA implementation can usually perform this analysis on programs with a few hundred thousand lines of code in a few minutes. However, this scalability can vary widely, in particular depending on implementation details inside the standard library, to be discussed further in Section 6.

4.3 Optimizations

In practice, an implementation can use several techniques in conjunction with the code in Figure 3 to improve performance by significant constant factors.

Type Filters In strongly-typed languages, *type filters* provide a simple but highly effective optimization which improves both precision and (usually) performance [21, 32].

Consider, for example, the following Java code:

```
Integer i = new Integer(0);
Double d = new Double(0.0);
Object o = new Random().nextBoolean() ? i : d;
Object p = (o instanceof Integer) ? (Integer)o : null;
o.toString();
```

The basic algorithm of Figure 3, which ignores the cast statement, would conclude imprecisely that `p` may alias `d`.

Slightly less obviously, consider the alias relation for the receiver (`this` pointers) in the methods `Integer.toString()` and `Double.toString()`. Due to the `o.toString()` invocation, the basic algorithm would conclude that since `o` may alias either `i` or `d`, then so may the receiver for each `toString()` method. However, this is imprecise, since the semantics of virtual dispatch ensure that the receiver of `Integer.toString()` cannot point to an object of type `Double`.

Type filters provide a simple technique to build these language constraints into the points-to analysis, in order to improve precision. We describe the technique informally as follows. In Figure 3, we add labels to the edges in the graph G. Each label represents a type in the source language – a label T can indicate either a "cone type" (any subtype of T) or a "point type" (only objects of concrete type T).

We modify the algorithm to add labels to the graph based on the source code. For example, for an assignment `x = (T) y`, we label the edge $y \rightarrow x$ with (cone type) T. We add similar labels for edges that arise from assignments from actual parameters to formal parameters, to capture type constraints imposed by virtual dispatch. Finally, we would modify the DiffProp routine to only add appropriately typed objects to points-to sets. This can be accomplished with a bit-vector intersection, updating the bit vector for each type as allocation sites are discovered.

Cycle elimination Consider the following Java code snippet:

```
Object a = ...
Object b = ...
Object c = ...
while (...) {
  if (?) a = b;
  else if (?) b = c;
       else c = b;
}
```

It should be clear that a points-to-analysis will compute the same points-to-set for `a`, `b`, and `c`. This arises from a *cycle* in the flow graph.

Cycles arise relatively frequently in flow graphs for flow-insensitive points-to analysis, especially for weakly-typed languages like C. When a cycle arises in the flow graph, a points-to analysis implementation can collapse the cycle in the flow graph and use a single representative points-to set for all variables in the cycle. This optimization can drastically reduce both space consumption (fewer points-to sets and constraints), and also time (less propagation to a fixed point). A key

challenge with cycle elimination is identifying cycles as they arise dynamically (due to flow graph edge additions), and a large body of work studies efficient cycle detection for C points-to analysis [17, 23, 27].

To our knowledge, the results with cycle elimination for Java points-to analysis have been much less impressive than those for C. We personally experimented with implementing cycle elimination in WALA and found it to provide little benefit. Paradoxically, cycle elimination works best for cases where the points-to analysis is often unable to distinguish between related points-to-sets. Recall that when analyzing Java, we can use type filters to achieve a more precise solution than typical for untyped C programs. Effectively, type filters "break cycles," since a labeled edge breaks the invariant that all variables in a cycle have the same points-to-set. It seems that for analyses with richer abstractions, cycle elimination becomes less effective, since the existence of huge cycles relies on a coarse abstraction that fails to distinguish locations.

Method-Local State In WALA, if a variable's points-to set is determined entirely by statements in the enclosing method, the points-to set is computed on-demand rather than via the global constraint system. Consider the following example:

```
void m(T x) {
  Object y = new Object();
  Object z = y;
  Object w = x.f;
}
```

For this case, $pt(y)$ and $pt(z)$ would be computed only when required, while the constraint system would be used to compute $pt(w)$ (since it depends on a field of parameter `x`). Though it complicates the implementation, we have found this optimization to yield significant space savings in practice. Separate handling of local state has been employed in other previous work [71, 73].

4.4 Handling Method Calls

On-the-fly call-graph construction (see discussion in Section 3.2) has a significant impact on real-world points-to analysis performance. If constraint generation costs are ignored, on-the-fly call graph reasoning can slow down analysis, as more iterations are required to reach a fixed point [72]. However, if the costs of constraint generation are considered (which we believe is a more realistic model), on-the-fly call graph building improves performance, since constraints need not be generated for unreachable library code. Also, on-the-fly call graph reasoning can make the flow graph for a program more sparse, improving performance.

As discussed in Section 3.2, context-sensitive points-to analysis can often give much more precise results for object-oriented programs than context-insensitive analysis. The most straightforward strategy for implementing context sensitivity is via *cloning*. Recall from Section 3.2 that context-sensitive analysis computes a different solution (points-to set) for local variables that arise in different contexts.

With cloning, the implementation simply creates a distinct copy of the relevant program structures for each context distinguished.

For example, consider a context-sensitivity policy employing k-limited call-string contexts with $k = 1$. For this policy, a cloning-based analysis would clone each method for each possible call site, and compute a separate solution for each clone. Intuitively, this can effectively blow up the program size by a quadratic factor — if there are N methods, each might be cloned N times, resulting in N^2 clones.

Data structures Cloning for context-sensitivity exacerbates the demand for both time and space. Much work over the last decade has improved techniques to exploit redundancy in the pointer analysis structures to mitigate these factors. The algorithm of Figure 3 must maintain two data structures, each of which grows super-linearly with program size:

- the set of constraints that represent the flow graph (G), and
- the points-to sets for each program variable.

A straightforward analysis implementation would represent the points-to sets using bit vectors, as commonly presented in textbooks for dataflow analysis [2]. The implementation can map each abstract object (e.g. allocation site) to a natural number, and then use these as identifiers in bit vector indices. At first glance, this seems like a compact representation, since it appears to devote roughly one bit of space to each unique piece of information in the output.

However, better solutions have been developed. Several key advances in pointer analysis implementation have relied on clever data structures to reduce the space costs of constraints and points-to sets by exploiting redundancy. The key insight is that many points-to sets are similar, due to the patterns by which values flow between variables in real programs. So, several works have proposed data structures to exploit these redundancies.

The WALA implementation uses a clever "shared-bit vector" representation presented by Heintze [24]. This implementation exploits the commonalities in bit vector contents, resulting in a bit vector representation that shares large common subsets. Each bit vector is represented as the union of a shared common base and a relatively small delta.

In our experience, the Heintze shared bit vectors can dramatically reduce the space costs of a cloning-based pointer analysis implementation, and allows some limited context sensitivity policies to scale to relatively large programs. However, these techniques cannot suffice for aggressive context-sensitivity policies, such as full call-string context sensitivity for variables [72]. For these policies, the number of clones grows exponentially with program size, to the point where even one bit per clone would demand more memory than there are flip-flops in the universe.

Several groups have presented solutions based on exploiting binary decision diagrams (BDDs), which potentially allow a system to explore an exponential space using a tractable implicit representation. This technique has been used extensively in explicit-state model checking [10], and several papers indicate that

similar techniques can work for certain flavors of context-sensitive pointer analysis [6, 7, 33, 72, 76]. Compared with shared-bit vectors, BDDs have the advantage of employing the same compact representation for both input constraints and the output points-to relation. Compact constraint representation makes aggressive policies like full call-string sensitivity for variables possible [72]. On the other hand, performance of BDD-based analyses can be fragile with respect to variable orderings [70], and using BDDs requires representing all relevant analysis state in BDD relations, making integration with other systems more difficult (though work has been done to ease this integration [35]).

Employing difference propagation exhaustively as in Figure 3 may double space requirements and hence represent an unattractive space-time tradeoff. A set implementation that enables propagation of abstract locations in parallel, like shared-bit vectors, lessens the need for exhaustive difference propagation in practice. In our experience, the key benefit of difference propagation lies in operations performed for each abstract location in a points-to set, e.g., edge adding (see lines 12 and 17 in Figure 3). To save space, WALA [69] only uses difference propagation for edge adding and for handling virtual call receivers (since with on-the-fly call graph construction, each receiver abstract location may yield a new call target). Also note that the best data structure for the $pt_\Delta(x)$ sets may differ from the $pt(x)$ sets to support smaller sets and iteration efficiently; see [32, 46] for further discussion.

4.5 Demand-Driven Analysis

The previous discussion focused on computing an *exhaustive* points-to analysis solution, i.e., computing all points-to sets for a program. However, recall that the primary motivation for pointer analysis is to enable some client, which performs some higher-level analysis such as for program understanding, verification, or optimization. For many such clients, computing the full solution is not required. The client will demand information for only a few program variables, and so it makes sense to compute the information requested *on-demand*.[13]

Heintze and Tardieu presented a highly influential paper describing a demand-driven version of context-insensitive Andersen's analysis, showing performance benefits for a client resolving C function pointers [26]. A demand-driven analysis formulation can also be obtained from a context-free-language reachability formulation of Andersen's analysis [49, 64] via the magic-sets transformation [48].[14]

Additional precision benefits can be obtained via *refinement* of the analysis abstraction where relevant to client queries. Guyer and Lin [22] showed the benefits of such an approach for various C points-to analysis clients. Sridharan et al. [60, 64] gave a refinement-based points-to analysis that exhibited significant precision and scalability improvements for several Java points-to analysis clients.

[13] On-demand computation of purely-local points-to sets was discussed in Section 4.3; here we extend the discussion to on-demand computation of any points-to set.

[14] For C, applying the magic-sets transformation to the Melski-Reps formulation [49] yields an equivalent analysis to that of Heintze and Tardieu [26].

Finally, recent work by Liang et al. [36, 37, 38] has shown that precision for clients can be improved with local improvements to the heap abstraction of a points-to analysis.

5 Must-Alias Analysis

Heretofore, we have concentrated on flow-insensitive alias analyses. These analyses produce a statically bounded (abstract) representation of the program's runtime heap. The pointer analysis solution indicates which abstract objects each pointer-valued expression in the program may denote.

Unfortunately, these scalable analyses have serious disadvantage when used for verification: they can answer only *may-alias* questions, that is, whether two variable may potentially refer to the same object. They cannot in general answer *must-alias* questions, that is, whether to two variables must always refer to the same object.

May-alias information requires a verifier to model any operation performed through a pointer dereference as an operation that may or may not be performed on the *possible* target abstract objects identified by the pointer analysis – this is popularly known as a "weak update" as opposed to a "strong update" [8].

In this section, we present a flow-sensitive must-alias analysis that is based on dynamic partition of the heap, and show how its greater precision is used to verify typestate properties.

5.1 On the Importance of Strong Updates

```
1  File makeFile {
2    return new File(); //  ⟨o1, init⟩, ⟨o2, init⟩
3  }
4  File f = makeFile(); //  ⟨o1, init⟩, ⟨o2, init⟩
5  File g = makeFile(); //  ⟨o1, init⟩, ⟨o2, init⟩
6  if(?)
7    f.open(); //  ⟨o1, open⟩, ⟨o2, init⟩
8  else
9    g.open();
10 f.read(); //  ⟨o1, open⟩, ⟨o2, init⟩
11 g.read(); //  ⟨o1, open⟩, ⟨o2, err⟩
12 }
```

Fig. 4. Concrete states for a program reading from two `File` objects allocated at the same allocation site. The example shows states for an execution in which the condition evaluates to true.

Consider a `File` type which requires invoking `open()` on a `File` object before invoking `read()`, and consider the simple example program of Fig. 4. The allo-

```
1 File makeFile {
2   return new File(); //  ⟨A, init⟩
3 }
4 File f = makeFile(); //  ⟨A, init⟩
5 File g = makeFile(); //  ⟨A, init⟩
6 if(?)
7   f.open(); //  ⟨A, open⟩
8 else
9   g.open(); //  ⟨A, open⟩
10 f.read(); //  ⟨A, open⟩
11 g.read(); //  ⟨A, open⟩
12 }
```

Fig. 5. Unsound update of abstract states for the example of Fig. 4.

cation statement in Line 2 allocates two `File` objects in some initial state *init*. In the figure, we write $\langle o, st\rangle$ to denote that an object o is in state st.

In this example, a typical points-to analysis will represent *both* objects allocated at Line 2 by a single abstract object A. The abstract state at Line 2 would therefore be $\langle A, init\rangle$, representing an *arbitrary* number of `File` objects that have been allocated at this point, all of which are in their initial state.

Now, consider the operation `f.open` invoked on the abstract object $\langle A, init\rangle$. What should be the effect of this operation? The abstract object $\langle A, init\rangle$ represents all objects allocated at Line 2. Assuming that the invocation of `f.open` yields the state $\langle A, open\rangle$ is equivalent to assuming that the state of *all* files represented by A has turned to *open*, which is unsound in general. For example, Fig. 5 shows the unsoundness of this scheme for the example from Fig. 4—the possible *err* state at Line 11 is not represented by the abstract states.

To guarantee soundness, the effect of `f.open()` would have to represent the possibility that some concrete objects represented by A remain in their initial state. As a result, the abstract state after `f.open()` should reflect both possibilities: $\langle A, open\rangle$, where the object is in its open state, and $\langle A, init\rangle$ where the object remains in its initial state. Such an update is referred to as a *weak update*, as it maintains the old state ($\langle A, init\rangle$) as part of the updated state. Using weak updates, however, would fail to verify even a simple program such the one shown in Fig. 6.

Addressing this issue requires knowing *must-alias* information. For Fig. 6, the analysis must prove that at Line 2, `f` *must* point to the object allocated by Line 1; with this knowledge, the analysis can show that the object can only be in the *open* state after the call, enabling verification. While computing this must-alias information would be straightforward for Fig. 6, in general the problem is much more challenging, due to language features like loops and method calls.

The literature contains many approaches for must-alias analysis, ranging from relatively simple abstractions such as the recency abstraction [5] and random isolation [29], to full-fledged shape analysis [53]. We next review a particular

```
1 File f = new File(); //  ⟨A, init⟩
2 f.open(); //  ⟨A, init⟩, ⟨A, open⟩
3 f.read(); //  ⟨A, err⟩, ⟨A, open⟩
```

Fig. 6. Simple correct example that cannot be verified directly using weak updates.

abstraction framework to combine may and must-alias analysis information, developed for typestate verification [20, 40, 57, 75]. The framework is based on maintaining must and must-not points-to information based on *access paths*.

5.2 Access Paths

Abstractions based on allocation sites impose a *fixed* partition on memory locations. We next present an abstraction to allow the name of an abstract object to change *dynamically* based on the program variables (and paths) that point to it. Specifically, we define the notion of an *access path*, a sequence of references that points to a heap allocated object, and name an abstract object by the set of access paths that may/must refer to it.

Concrete Semantics We assume a standard concrete semantics which defines a program state and evaluation of an expression in a program state. The semantic domains are defined in a standard way as follows:

$$\begin{aligned}
L^\natural &\in 2^{objects^\natural} \\
v^\natural &\in Val = objects^\natural \cup \{null\} \\
\rho^\natural &\in Env = VarId \to Val \\
h^\natural &\in Heap = objects^\natural \times FieldId \to Val \\
\sigma^\natural = \langle L^\natural, \rho^\natural, h^\natural \rangle &\in States = 2^{objects^\natural} \times Env \times Heap
\end{aligned}$$

where $objects^\natural$ is an unbounded set of dynamically allocated objects, *VarId* is a set of local variable identifiers, and *FieldId* is a set of field identifiers. We generally use the $\natural$ superscript to denote concrete entities.

A *program state* keeps track of the set of allocated objects ($L^\natural$), an environment mapping local variables to values ($\rho^\natural$), and a mapping from fields of allocated objects to values ($h^\natural$).

We also define the notion of an access path as follows: A *pointer path* $\gamma \in \Gamma = FieldId^*$ is a (possibly empty) sequence of field identifiers. The empty sequence is denoted by ϵ. We use the shorthand f^k where $f \in FieldId$ to mean a sequence of length k of accesses along a field f. An *access path* $p \equiv x.\gamma \in VarId \times \Gamma$ is a pair consisting of a local variable x and a pointer path γ.

We denote by *APs* all possible access paths in a program. The l-value of access path p, denote by $\sigma^\natural[p]$, is recursively defined using the environment and heap mappings, in the standard manner. Given a concrete object $o^\natural$ in a state $\sigma^\natural$, we denote by $AP^\natural(o^\natural)$ the set of access paths that point to o .

Maintaining Must Points-to Information To describe our abstraction, we first assume that a preliminary flow-insensitive points-to analysis has run. This analysis generates an abstract points-to graph based on a static set of abstract memory locations. For this discussion, we call each abstract memory location from the preliminary points-to analysis an *instance key*.

The more precise analysis performs a flow-sensitive, context-sensitive interprocedural propagation of abstract states. Each abstract state represents a set of concrete states that may arise during execution, and encodes information regarding certain aliasing relationships which these concrete states share. We represent aliasing relationships with tuples of the form $\langle o, \mathit{unique}, AP_{\text{M}}, \mathit{May}, AP_{\text{MN}} \rangle$ where:

- o is an instance key.
- *unique* indicates whether the corresponding allocation site has a single concrete live object.
- AP_{M} is a set of access paths that must point-to o.
- *May* is a boolean that indicates whether there are access paths (not in the must set) that may point to o.
- AP_{MN} is a set of access paths that do *not* point-to o.

This parameterized abstract representation has four dimensions, for the *length* and *width* of each access path set (must and must-not). The length of an access path set indicates the maximal length of an access path in the set, similar to the parameter k in k-limited context-sensitivity policies. The width of an access path set limits the number of access paths in this set.

An abstract state is a set of tuples. We observe that a conservative representation of the concrete program state must obey the following properties:

1. An instance key can be indicated as unique if it represents a single object for this program state.
2. The access path sets (the must and the must-not) do not need to be complete. This does not compromise the soundness of the abstraction, since other elements in the tuple can indicate the existence of other possible aliases.
3. The must and must-not access path sets can be regarded as another heap partitioning which partitions an instance key into the two sets of access paths: those that a) must alias this abstract object, and b) definitely do not alias this abstract object. If the must-alias set is non-empty, the must-alias partition represents a single concrete object.
4. If $\mathit{May} = \mathit{false}$, the must access path is complete; it contains all access paths to this object.

This can be formally stated as follows:

Definition 1 *A tuple* $\langle o, \mathit{unique}, AP_{\text{M}}, \mathit{May}, AP_{\text{MN}} \rangle$ *is a* sound representation *of object* $o^\natural$ *at program state* $\sigma^\natural$ *when the following conditions hold:*

- $o = \mathit{ik}(o^\natural)$
- $\mathit{unique} \Rightarrow \{x^\natural \in \mathit{live}(\sigma^\natural) \mid \mathit{ik}(x^\natural) = o\} = \{o^\natural\}$
- $AP_{\text{M}} \subseteq AP^\natural(o^\natural)$

– $(\neg May \Rightarrow (AP_{\text{M}} = AP^{\natural}(o^{\natural})))$
– $AP_{\text{MN}} \cap AP^{\natural}(o^{\natural}) = \emptyset$

where ik is an abstraction mapping a concrete object to the instance key that represents it, and $live(\sigma^{\natural})$ *is defined to be* $\{x^{\natural} \mid AP^{\natural}(x^{\natural}) \neq \emptyset\}$.

Definition 2 *An abstract state* σ *is a* sound *representation of a concrete state* $\sigma^{\natural} = \langle L^{\natural}, \rho^{\natural}, h^{\natural} \rangle$ *if for every object* $o^{\natural} \in L^{\natural}$ *there exists a tuple in* σ *that provides a sound representation of* $o^{\natural}$.

Abstract Transformers Table 3 shows how a tuple is transformed by the interpretation of various statements. The effect of a transfer function on a given abstract state is defined by taking the union of applying the tuple transfer functions of Table 3 to each tuple in the abstract state.

The interpretation of an allocation statement "`v = new T()`" with instance key o will *generate* a tuple $\langle o, true, \{v\}, false, \emptyset \rangle$ representing the newly allocated object. When *May* is false, the AP_{MN} component is redundant and, hence, initialized to be empty.

When a tuple reaches the allocation site that created it, we generate two tuples, one representing the newly created object, and one representing the incoming tuple. We change the uniqueness flag to false for reasons explained earlier. For assignment statements, we update the AP_{M} and AP_{MN} as appropriate.

Note that since we place a finite bound on access path lengths, there are a finite number of possible abstract states, so fixed-point iteration terminates. The number of possible abstract states is exponential in the access path bound.

To use this aliasing information in a client analysis, we can extend the abstract transformers of Table 3 to also maintain the abstract state of an object being tracked.

For example, consider a simple typestate analysis to verify that only *open* files are *read*. We extend the tuple to track the abstract state of a `File` object, which can be *init* (just initialized), *open*, or *closed*.

```
1 Collection files = ...
2 while (...) {
3  File f = new File(); // (⟨A, true, {f}, false, ∅⟩, init), (⟨A, false, ∅, true, {f}⟩, open)
4  files.add(f); // (⟨A, true, {f}, true, ∅⟩, init), (⟨A, false, ∅, true, {f}⟩, open)
5  f.open(); // (⟨A, true, {f}, true, ∅⟩, open), (⟨A, false, ∅, true, {f}⟩, open)
6  f.read(); // (⟨A, true, {f}, true, ∅⟩, open), (⟨A, false, ∅, true, {f}⟩, open)
7 }
```

Fig. 7. Illustration of a strong update to the state of a `File` object using access paths.

Consider the example of Fig. 7. In this example, the abstraction is able to capture the fact that at Line 5, `f` must point to the object allocated by the most

Stmt S	**Resulting abstract tuples**
$v =$ `new` T() where $o =$ Stmt S	$\langle o, false, AP_{\text{M}} \setminus startsWith(v, AP_{\text{M}}), May, AP_{\text{MN}} \cup \{v\}\rangle$ $\langle o, true, \{v\}, false, \emptyset\rangle$
$v =$ `null`	$\langle o, unique, AP'_{\text{M}}, May, AP'_{\text{MN}}\rangle$ $AP'_{\text{M}} := AP_{\text{M}} \setminus startsWith(v, AP_{\text{M}})$ $AP_{\text{MN}} := AP_{\text{MN}} \cup \{v\}$
$v.f =$ `null`	$\langle o, unique, AP'_{\text{M}}, May, AP'_{\text{MN}}\rangle$ $AP'_{\text{M}} := AP_{\text{M}} \setminus \{e'.f.\gamma \mid mayAlias(e', v), \gamma \in \Gamma\}$ $AP'_{\text{MN}} := AP_{\text{MN}} \cup \{v.f\}$
$v = e$	$\langle o, unique, AP'_{\text{M}}, May, AP'_{\text{MN}}\rangle$ $AP'_{\text{M}} := AP_{\text{M}} \cup \{v.\gamma \mid e.\gamma \in AP_{\text{M}}\}$ $AP'_{\text{MN}} := AP_{\text{MN}} \setminus \{v \mid e \notin AP_{\text{MN}}\}$
$v.f = e$	$\langle o, unique, AP'_{\text{M}}, May', AP'_{\text{MN}}\rangle$ $AP'_{\text{M}} := AP_{\text{M}} \cup \{v.f.\gamma \mid e.\gamma \in AP_{\text{M}}\}$ $May' := May \vee \exists v.f.\gamma \in AP'_{\text{M}}.\exists p \in AP.mayAlias(v, p) \wedge p.f.\gamma \notin AP'_{\text{M}}$ $AP'_{\text{MN}} := AP_{\text{MN}} \setminus \{v.f \mid e \notin AP_{\text{MN}}\}$
$startsWith(v, P) = \{v.\gamma \mid \gamma \in P\}$	

Table 3. Transfer functions for statements indicating how an incoming tuple $\langle o, unique, AP_{\text{M}}, May, AP_{\text{MN}}\rangle$ is transformed, where $pt(e)$ is the set of instance keys pointed-to by e in the flow-insensitive solution, $v \in VarId$. $mayAlias(e_1, e_2)$ iff pointer analysis indicates e_1 and e_2 may point to the same instance key.

recent execution of line 3, and its state can be therefore update to *open*. This means that `read()` can be safely invoked on the object pointed to by `f` at Line 6.

When a typestate method is invoked, we can (1) use the AP_{MN} information to avoid changing the typestate of the tuple where possible, (2) use the AP_{M} information to perform strong updates on the tuple where possible, and (3) use the uniqueness information also to perform strong updates where possible.

There are several more powerful tricks which can use this information to improve precision – notably a *focus* operation which performs a limited form of case splitting to improve abstraction precision. We refer the reader to [20] for further discussion of techniques using these abstractions.

As explained earlier, we enforce limits on the length and the number of access paths allowed in the AP_{M} and AP_{MN} components to keep the number of tuples generated finite. We designed this abstract domain specifically to discard access-path information soundly, allowing heuristics that trade precision for performance but do not sacrifice soundness. This feature is crucial for scalability; the analysis would suffer an unreasonable explosion of dataflow facts if it soundly tracked every possible access path, as in much prior work [9, 14, 15, 30].

However, the abstraction just presented still relies on a preliminary sound points-to analysis. The abstraction introduces machinery designed to exploit the points-to analysis, in order to maintain a sound (over-approximate) representation of the set of possible concrete states.

In practice, modern large Java programs introduce substantial barriers to this style of sound verification. As we discuss next, certain features of these programs introduce prohibitive obstacles to running a preliminary, sound points-

to analysis. However, we show that access-path tracking in the style described here is still useful in the context of under-approximate analyses, which do not guarantee coverage of all program states.

6 Analyzing Modern Java Programs

In our most recent work, we have found that large libraries and reflection usage in modern Java programs and libraries have made points-to analysis (as described in Sections 3 and 4) a poor basis for alias reasoning. In this section, we describe in more detail why points-to analysis does not work well for modern Java programs and libraries, and how we have worked around this issue with under-approximate techniques based on type-based call graph construction and access-path tracking.

We note that though we have not found points-to analysis to work well for modern desktop and server Java applications, it remains relevant in other domains. Scalability issues with points-to analysis may be less severe in cases where applications tend to have less code and use smaller libraries, e.g., mobile phone applications. Furthermore, for languages where a type-based approach does not yield a reasonable call graph (e.g., JavaScript), points-to analysis remains the most scalable technique for reasoning about indirect function calls [61].

6.1 Points-to Analysis Difficulties

In Java-like languages, *reflection* allows for meta-programming based on string names of program constructs like classes or methods. For example, the following code reflectively allocates an object of the type named by `x`:

```
class Factory {
  Object make(String x) {
    return Class.forName(x).newInstance();
  }
}
```

Analyzing this code with the assumption that `x` may name any type yields extremely imprecise results, as is most cases only a few types may be allocated by code like the above. In some cases, tracking string constants flowing into `x` and only considering those types can help. However, many common idioms make this difficult or ineffective, such as use of string concatenation or reading the string from a configuration file.

Previous work has suggested handling code like the above by exploiting uses of the allocated object [7, 39], since the object is often cast to a more specific type before it is used. By tracking data flow of reflectively-created objects to casts and optimistically assuming that the casts succeed, the set of allocated types can often be narrowed significantly. For example, consider the following use of the previously-shown `Factory` class:

```
Factory f = new Factory();
String widgetClass = properties.get("widgetClassName");
IWidget w = (IWidget) f.make(widgetClass);
```

In this case, the analysis treats the reflective code as allocating any subtype of `IWidget`.

Unfortunately, techniques like the above cannot save points-to analysis from reflection in general. Sometimes, reflective creations can flow to interfaces like `java.io.Serializable`, which is implemented by many types. In other cases, reflection is used without any downcasts, e.g., reflective method calls via Java's `Method.invoke()`. As standard libraries and frameworks have grown, the cost of imprecise reflection handling has increased dramatically, since many more types and methods may be (imprecisely) deemed reachable. In some cases, certain parts of libraries may be manually excluded based on knowledge of the application; e.g., GUI libraries can be excluded when analyzing a program with no graphical interface (this approach is used when running WALA regression tests). However, for cases like server applications that are themselves packaged with many libraries (we have observed more than 75 `.jar` files in some cases), manual exclusions are not suitable. We are unaware of any automatic technique that is able to handle reflection across large Java applications with sufficient precision, and others have also observed this problem [58, Section 5.1].

6.2 Under-Approximate Techniques

Given the aforementioned difficulties with over-approximate alias analysis, under-approximate techniques present an attractive alternative in cases where sound analysis is not required (e.g., in a bug detection tool). When first exploring this area, we tried to base our approach on a points-to analysis modified to be under-approximate, either via reduced reflection handling or use of a partial result computed in some time bound. However, we found this approach to be unsatisfactory: ignoring reflection often led to missed, important behaviors (particularly in framework-based applications [59]), and time bounds required complex heuristics to ensure that the points-to analysis explored desired parts of the program early. Instead, we have turned to an approach of (i) using a variant of the access-path tracking described in Section 5 to track must-alias information under-approximately, and (ii) employing domain-specific modeling of reflection as needed. We describe these techniques in turn.

Under-Approximation by using Only Must Information Section 5 described an access-path analysis based on states of $\langle o, \mathit{unique}, AP_{\text{M}}, \mathit{May}, AP_{\text{MN}} \rangle$ tuples, designed to achieve high precision while relying on a pre-computed points-to analysis for soundness. In the under-approximate setting, we can define a simpler analysis with tuples of the form $\langle o, AP_{\text{M}} \rangle$, carrying only must-alias information. As earlier, each time the transformation of must-access-paths set is computed, we must limit the size of the resulting set. It is necessary to do so for two reasons: (i) in the presence of loops (or recursion), it is possible for access paths to grow without a bound (ii) even loop free code might inflate the sets to needlessly large sizes, compromising efficiency. The transformers over $\langle o, AP_{\text{M}} \rangle$ tuples follow the update for AP_{M} as described in Table 3.

In the following examples, we demonstrate how this abstraction can be used for identifying resource leaks (see Section 2), and consider tuples of the form $\langle o, R, AP_{\text{M}} \rangle$ where R is a resource type. R can be acquired via statement type `p = acquire R` and released via `release R r` (where `r` points to the resource). At a high level, we use the must-alias access-path abstraction to detect resource leaks as follows (see Torlak and Chandra [67] for full details):

- At statement `p = acquire R`, the analysis generates tuple $\langle p, R, \{p\} \rangle$ (we name resource objects by the variable to which they are first assigned).
- Must aliases are updated as shown in Table 3.
- If statement `release R r` is reached with tuple $t = \langle p, R, a \rangle$ and $r \in a$, then the analysis kills t.
- If method exit is reached with a tuple $\langle p, R, \{\} \rangle$, then a leak is reported, as no aliases exist to release the resource.
- For additional precision, conditionals performing null checks are interpreted. If a conditional checking $v = \mathit{null}$ is reached with tuple $t = \langle p, R, a \rangle$ and $v \in a$, t is killed on the true branch, as a must-alias for a resource cannot be null. $v \neq \mathit{null}$ is handled similarly.

Example 1 Consider the code fragment shown below. We show the facts accumulated by our analysis after each statement to the right.

`p = acquire R`	$\langle p, R, \{p\} \rangle$
`q.f = p`	$\langle p, R, \{p, q.f\} \rangle$
`r = q.f`	$\langle p, R, \{r, p, q.f\} \rangle$
`branch (r == null) L1`	T: *none*, F: $\langle p, R, \{r, p, q.f\} \rangle$
`release R r`	*none*
`L1:`	*none*

At the `branch` statement, the analysis concludes that only the fall-through successor is feasible: r, being a must-alias to a resource, cannot be a null pointer. At the `release` statement, the analysis uses the AP_{M} set to establish that r must-alias the resource. Consequently, no fact makes it to L1, and no error is reported.

Had the analysis not interpreted the `branch` statement, fact $\langle p, R, \{r, p, q.f\} \rangle$ would have reached L1. Local variables `p`, `q`, and `r` would then be dropped from the state, giving the fact $\langle p, R, \{\} \rangle$ at the exit. This fact would lead the analysis to conclude that resource p is unreachable at exit, resulting in a false positive.

Example 2 Consider the leaky code fragment shown below. It allocates a resource in a loop, but frees only the last allocated instance. The `branch * L2` has a non-deterministic condition which cannot be interpreted by the analysis.

p_1 = null	
L1	$\langle p_3, R, \{p_3\}\rangle$, $\langle p_3, R, \{p_2\}\rangle$
$p_2 = \phi(p_1, p_3)$	$\langle p_3, R, \{p_2, p_3\}\rangle$, $\langle p_3, R, \{\}\rangle$
branch * L2	
p_3 = acquire R	$\langle p_3, R, \{p_3\}\rangle$, $\langle p_3, R, \{p_2\}\rangle$
branch true L1	
L2	$\langle p_3, R, \{p_2, p_3\}\rangle$, $\langle p_3, R, \{\}\rangle$
release R p_2	$\langle p_3, R, \{\}\rangle$

This fragment also illustrates the treatment of ϕ nodes. Consider the path taken through the loop two times and then exiting to L2. The analysis generates $\langle p_3, R, \{p_3\}\rangle$ after the `acquire`. The generated fact flows to L1, and the analysis generates $\langle p_3, R, \{p_2, p_3\}\rangle$ after the ϕ, using the effect of $p_2 = p_3$. This, in turn, flows out to L2 via the `branch`, where it is killed by the `release`.

In the next loop iteration, the `acquire` statement overwrites p_3, so the analysis kills the occurrence of p_3 in $\{p_2, p_3\}$, generating the new fact $\langle p_3, R, \{p_2\}\rangle$. After propagation on the back edge, this last fact is transformed by the ϕ statement to $\langle p_3, R, \{\}\rangle$. Finally, when the transformed fact flows out to L2, it cannot be killed by `release` since the must-alias set is empty. The fact reaches method exit, and a leak is reported.

Method calls We have not yet addressed how an under-approximate analysis like the resource leak detector reasons about method calls; as discussed in Sections 2 and 3, call graph construction and alias analysis are inter-dependent. We have found that an under-approximate call graph based on the class hierarchy is sufficient for bug-finding tools like the leak detector. In using the class hierarchy, all available code is considered, so certain issues related to insufficient reflection handling are avoided. For call sites with a very large number of possible targets, a subset is chosen heuristically, with the heuristics tunable by the client analysis. For example, in the leak detector [67], the heuristics were tuned to prefer code performing resource allocation.

Domain-specific Reflection Modeling In certain cases, key application behaviors are implemented using reflection, necessitating modeling of those reflective behaviors. For example, server-side web applications written in Java are typically built atop Java EE[15] and other frameworks, and the application code is only invoked via reflective calls from the framework. To effectively detect security vulnerabilities in such applications using taint analysis [68], the analysis must have visibility into how these reflective calls invoke application code (e.g., to see how untrusted data is passed). For taint analysis of web applications, recent work describes Framework for Frameworks (F4F) [59], a system that eases modeling the security-relevant behaviors of web-application frameworks. We expect similar modeling to be required in other domains where complex, reflection-heavy frameworks are employed.

[15] `http://www.oracle.com/technetwork/java/javaee/index.html`

7 Conclusions and Future Work

We have presented a high-level overview of state-of-the-art may- and must-alias analyses for object-oriented programs, based on our experiences implementing production-quality static analyses for Java. The sound alias-analysis techniques presented here work well for medium-sized programs, while for large-scale Java programs, an under-approximate alias analysis based on access-path tracking currently yields the most useful results.

We see several potentially fruitful directions for future work, for example:

Reflection Improved reflection handling could significantly increase the effectiveness of various alias-analysis techniques. Approaches based on analyzing non-code artifacts like configuration files [59] or introducing more analyzable language constructs [28] seem particularly promising.

Dynamically-Typed Languages As scripting languages like JavaScript gain in popularity, there is an increasing need for effective alias analyses for such languages. Analyzing such languages poses significant challenges, as use of reflective code constructs is even more pervasive than in Java, and optimizations based on the type system (see Section 4.3) may no longer be effective in improving scalability [61].

Developer Tool Integration Some initial work has been done on developer tools that make significant use of alias analysis [19, 54, 63], but we believe there is significant further scope for tools to help developers reason about data flow and aliasing in their programs. Better tools for reasoning about aliasing are particularly important since trends indicate increasing usage of dynamically-typed languages and large frameworks, both of which can obscure aliasing relationships in programs.

Bibliography

[1] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP*, 1995.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools with Gradiance*. Addison-Wesley Publishing Company, USA, 2nd edition, 2007.

[3] Lars O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.

[4] David Bacon and Peter Sweeney. Fast static analysis of C++ virtual function calls. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), San Jose, CA*, October 1996.

[5] Gogul Balakrishnan and Thomas Reps. Recency-abstraction for heap-allocated storage. In *IN SAS*, pages 221–239, 2006.

[6] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[7] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM.

[8] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 296–310, New York, NY, 1990. ACM Press.

[9] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, pages 232–245, 1993.

[10] Edmund Clarke. Model checking. In S. Ramesh and G Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *Lecture Notes in Computer Science*, pages 54–56. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0058022.

[11] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 269–282, New York, NY, 1979. ACM Press.

[12] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 57–68, New York, NY, USA, 2002. ACM.

[13] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, August 1995.

[14] N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *ISSTA*, pages 12–22, 2004.

[15] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256, New York, NY, USA, 1994. ACM Press.

[16] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[17] Manuel Fändrich, Jeffrey S. Foster, Zhendong Su, and Alex Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada*, June 1998.

[18] Christian Fecht and Helmut Seidl. Propagating differences: an efficient new fixpoint algorithm for distributive constraint systems. *Nordic J. of Computing*, 5(4):304–329, 1998.

[19] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for JavaScript. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 119–138, New York, NY, USA, 2011. ACM.

[20] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology*, 17(2):1–34, 2008.

[21] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, 2001.

[22] Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In *International Static Analysis Symposium (SAS), San Diego, CA*, June 2003.

[23] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290–299, 2007.

[24] Nevin Heintze. Analysis of Large Code Bases: The Compile-Link-Analyze Model. Draft of November 12, 1999.

[25] Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. *SIGPLAN Not.*, 32(5):261–272, 1997.

[26] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah*, June 2001.

[27] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Conference on Programming Language Design and Implementation (PLDI)*, June 2001.

[28] Shan Shan Huang and Yannis Smaragdakis. Morphing: Structurally shaping a class by reflecting on others. *ACM Trans. Program. Lang. Syst.*, 33:6:1–6:44, February 2011.

[29] Nicholas Kidd, Thomas W. Reps, Julian Dolby, and Mandana Vaziri. Finding concurrency-related bugs using random isolation. *STTT*, 13(6):495–518, 2011.

[30] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 235–248, New York, NY, USA, 1992. ACM Press.

[31] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 278–289, New York, NY, USA, 2007. ACM.

[32] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction (CC), Warsaw, Poland*, April 2003.

[33] Ondřej Lhoták and Laurie Hendren. Jedd: a BDD-based relational extension of Java. In *Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[34] Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18:3:1–3:53, October 2008.

[35] Ondřej Lhoták and Laurie Hendren. Relations as an abstraction for BDD-based program analysis. *ACM Trans. Program. Lang. Syst.*, 30:19:1–19:63, August 2008.

[36] Percy Liang and Mayur Naik. Scaling abstraction refinement via pruning. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 590–601, New York, NY, USA, 2011. ACM.

[37] Percy Liang, Omer Tripp, and Mayur Naik. Learning minimal abstractions. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 31–42, New York, NY, USA, 2011. ACM.

[38] Percy Liang, Omer Tripp, Mayur Naik, and Mooly Sagiv. A dynamic evaluation of the precision of static heap abstractions. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 411–427, New York, NY, USA, 2010. ACM.

[39] V. Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for Java. In *APLAS*, pages 139–160, 2005.

[40] Alexey Loginov, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, and M. G. Nanda. Verifying dereference safety via expanding-scope analysis. In *ISSTA '08: International Symposium on Software Testing and Analysis*, 2008.

[41] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 305–315, New York, NY, USA, 2010. ACM.

[42] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[43] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *PLDI*, pages 308–319, 2006.

[44] Robert O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie Mellon University, November 2000.

[45] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, OOPSLA '91, pages 146–161, New York, NY, USA, 1991. ACM.

[46] David J. Pearce. *Some directed graph algorithms and their application to pointer analysis*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, 2005.

[47] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Online cycle detection and difference propagation for pointer analysis. In *Proceedings of the third international IEEE Workshop on Source Code Analysis and Manipulation*, 2003.

[48] Thomas Reps. Solving demand versions of interprocedural analysis problems. In *International Conference on Compiler Construction (CC), Edinburgh, Scotland*, April 1994.

[49] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, November/December 1998.

[50] Thomas Reps. Undecidability of context-sensitive data-independence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, 2000.

[51] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1995.

[52] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Tampa Bay, Florida*, October 2001.

[53] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24:217–298, May 2002.

[54] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Refactoring Java programs for flexible locking. In *Proceeding of the 33rd International Conference on Software Engineering*, ICSE '11, pages 71–80, New York, NY, USA, 2011. ACM.

[55] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[56] O. Shivers. Control flow analysis in Scheme. In *Conference on Programming Language Design and Implementation (PLDI)*, 1988.

[57] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 174–184, New York, NY, USA, 2007. ACM.

[58] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL*, pages 17–30, 2011.

[59] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4F: taint analysis of framework-based web applications. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 1053–1068, New York, NY, USA, 2011. ACM.

[60] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[61] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *ECOOP*, 2012.

[62] Manu Sridharan and Stephen J. Fink. The complexity of Andersen's analysis in practice. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 205–221, Berlin, Heidelberg, 2009. Springer-Verlag.

[63] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 112–122, New York, NY, USA, 2007. ACM.

[64] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

[65] Bjarne Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1996.

[66] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Minneapolis, MN*, October 2000.

[67] Emina Torlak and Satish Chandra. Effective interprocedural resource leak detection. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 535–544, New York, NY, USA, 2010. ACM.

[68] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *PLDI*, 2009.

[69] T.J. Watson Libraries for Analysis (WALA). `http://wala.sf.net`.

[70] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In *The Third Asian Symposium on Programming Languages and Systems*, 2005.

[71] John Whaley and Monica S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *International Static Analysis Symposium (SAS), Madrid, Spain*, September 2002.

[72] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[73] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 1999.

[74] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 1995.

[75] Eran Yahav and Stephen Fink. The SAFE experience. In *Engineering of Software*, pages 17–33. Springer Berlin Heidelberg, 2011.

[76] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *Conference on Programming Language Design and Implementation (PLDI)*, 2004.