A Combined Pointer and Purity Analysis for Java Programs

Alexandru Sălcianu and Martin Rinard
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139
{salcianu, rinard}@csail.mit.edu

ABSTRACT

We present a new method purity analysis for Java programs. A method is pure if it does not mutate any location that exists in the program state right before method invocation. Our analysis is built on top of a combined pointer and escape analysis for Java programs and is capable of determining that methods are pure even when the methods do heap mutation, provided that the mutation affects only objects created after the beginning of the method.

Because our analysis extracts a precise representation of the region of the heap that each method may access, it is able to provide useful information even for methods with externally visible side effects. In particular, it can recognize *read-only* parameters (a parameter is read-only if the method does not mutate any objects transitively reachable from the parameter) and *safe* parameters (a parameter is safe if it is read-only and the method does not create any new externally visible paths in the heap to objects transitively reachable from the parameter). The analysis can also generate regular expressions that characterize the externally visible heap locations that the method mutates.

We have implemented our analysis and used it to analyze several data structure implementations. Our results show that our analysis effectively recognize a variety of pure methods, including pure methods that allocate and mutate complex auxiliary data structures. Even if the methods are not pure, our analysis can provide information which may enable developers to usefully bound the potential side effects of the method.

Keywords

Static analysis, purity analysis, effect inference

1. INTRODUCTION

Methods in object-oriented languages often update the objects that they access, including the "this"/"self" object. Accurately characterizing these updates is important for many tasks. Many program analyses, for example, need to understand how the execution of invoked methods may affect the information that the analysis maintains [15,17,20]. Accurate side effect information is also useful for program understanding and documentation [16, 23]. The knowledge that a method is *pure*, or has no externally visible side effects, is especially useful because it guarantees that invocations of the method will not inadvertently interfere with other computations. Researchers in a variety of fields have identified method purity as a useful concept. For example, when model checking Java programs [10, 11, 35], it is important to know that methods are pure because this information allows the model checker to reduce the search space by removing irrelevant interleavings.

This paper presents a new method purity analysis for Java programs. This analysis is built on top of a combined pointer and escape analysis that accurately extracts a representation of the region of the heap that each method may access. Our analysis conservatively tracks object creation, updates to the local variables and updates to the object fields. This information enables our analysis to distinguish objects allocated within the computation of the method from objects that existed before the method was invoked. It also allows the analysis to recognize *captured objects* whose lifetime is contained within the lifetime of their allocating method.

Therefore, our analysis can check that a method is pure, in the sense that it does not mutate any object that exists in the pre-state, i.e., the program state right before the method invocation; this is also the definition of purity adopted in the Java Modeling Language [23]. This definition allows a method to perform mutation on newly allocated objects and/or construct objects and return them as a result.

Our analysis applies a more flexible purity criterion than previously implemented purity analyses, e.g., [8], that consider a method to be pure only if it does not perform any writes on heap locations at all, and does not invoke any impure method. The increased precision of our analysis enables pure methods to use important programming constructs such as iterators and complex auxiliary data structures.

Purity Generalizations

Even when a method is not pure, it may have some useful generalized purity properties. For example, our analysis can recognize *read-only* parameters; a parameter is read-only if the method does not write the parameter or any objects reachable from the parameter. It can also recognize *safe* parameters; a parameter is safe if it is read-only and the method does not create any new externally visible paths in the heap to objects reachable from the parameter.

The *read-only* and *safe* parameter properties do not consider the (unknown) aliasing from the calling context. E.g., if a method is invoked in a context where a read-only parameter is aliased with a non-read-only parameter, mutation

This research was supported by DARPA Contract F33615-00-C-1692, NSF Grant CCR-0086154, NSF Grant CCR-0073513, NSF Grant CCR-0209075, and the Singapore-MIT Alliance.

MIT CSAIL Technical Report, MIT-CSAIL-TR-949, May 2004

can occur on the object pointed to by the read-only parameter, through the non-read-only alias. This is the common approach in detecting and specifying read-only annotations for Java [2].

A program verifier should use both the safe parameter information inferred by the analysis *and* the aliasing information at the call site. Here is an example scenario: a typestate checker, e.g., [15], is a tool that tracks the typestate of objects; one important application is checking complex, finite state machine-like API usage protocols. The typestate checker can precisely track only the state of the objects for which all aliasing is statically known. Each time such an object is passed as a safe parameter, the typestate checker can rely on the fact that the method call does not change the state of the object, and it does not introduce new aliasing to the object. As the typestate checker knows all aliasing to the tracked object, it can check that the tracked object is not aliased with any object transitively reachable from a non-safe argument at the call site. This example illustrates the use of safe parameter information for increasing the effectiveness of other static analyses.

Finally, our analysis is capable of generating regular expressions that completely characterize the externally visible heap locations that a method mutates. These regular expressions identify paths in the heap that start with a parameter or static class field and end with a potentially mutated object field. This side effect information can provide many of the same benefits as a purity analysis because it enables other program analyses and developers to usefully bound the potential effects of the method.

Contributions

This paper makes the following contributions:

- **Purity Analysis:** We present a new analysis for finding pure methods in unannotated Java programs. Unlike previously implemented purity analyses, we track variable and field updates, and allow mutations on newly allocated data structures. Our analysis therefore supports the use of important programming constructs such as iterators in pure methods.
- **Supporting Pointer Analysis:** We show how to place this purity analysis on top of an underlying pointer analysis. We use an updated version of the Whaley and Rinard pointer analysis [36]. The updated version retains the ideas of the original analysis, but is better structured in order to allow the analysis correctness proof from [31].
- **Experience:** We present our experience using our analysis to find pure methods in a number of benchmark programs. We found that our analysis was able to recognize the purity of methods that 1) were known to be pure, but 2) were beyond the reach of previously implemented purity analyses because they allocate and mutate complex internal data structures.
- **Read-Only/Safe Parameters:** Our analysis detects *read-only* parameters; the execution of the method does not mutate objects reachable from these parameters. The analysis can also detect *safe* parameters, i.e., read-only parameters such that the execution of the method does not produce any new externally visible heap paths to the objects reachable from these parameters.
- **Write Effect Inference:** We show how to use the results of our analysis to generate regular expressions that conservatively approximate heap paths to all externally visible locations that an impure method mutates.

Paper Structure: Section 2 illustrates the execution of our analysis on an example. Section 3 presents the mathematical notations we use in this paper, and Section 4 describes the representation of the analyzed programs. Section 5 gives a formal presentation of our analysis, and Section 6 shows how to interpret the analysis results. Section 7 presents experimental results. Section 8 discusses some related work, and Section 9 concludes.

2. EXAMPLE

2.1 Example Overview

Figure 1 presents sample Java source code that implements a singly linked list in class `List`; the list implementation uses list cells of class `Cell`. Our lists support two operations: `add(e)` adds object `e` to a list, and `iterator()` returns an iterator over the list elements.[1] We also define a class `Point` for modeling bidimensional points, and two static methods that process lists of `Point`s. `Main.sumX(list)` returns the sum of the `x` coordinates of all points from `list`, and `Main.flipAll(list)` flips the `x` and `y` coordinates of all points from `list`.

Method `sumX` iterates over all the list elements, by repeatedly invoking the `next()` method on the list iterator. The method `next()` is impure, because it mutates the state of the iterator; in our implementation, it mutates the field `cell` of the iterator. However, the iterator is an auxiliary object that did not exist at the beginning of `sumX`. As we present in this section, our analysis is able to infer that `sumX` is pure, in spite of the mutation on the iterator. Our analysis is also able to infer that the impure method `flipAll` mutates only locations that are accessible in the prestate[2] along paths that match the regular expression `list.head.next*.data.(x|y)`.

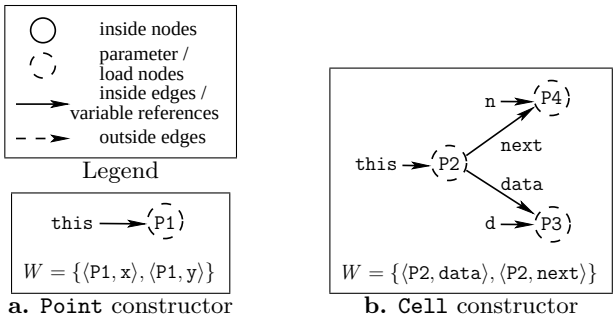
2.2 Intuitive Description of the Analysis

For each method m and for each program point inside m, the analysis computes a points-to graph that models the part of the heap that the method m accesses up to that program point. The nodes from the points-to graphs model heap objects: the *inside nodes* model the objects created by the analyzed method, the *parameter nodes* model the objects passed as arguments, and the *load nodes* model the objects read from outside the method. The analysis uses edges to model heap references; each edge is labeled with the field it corresponds to. We write $\langle n_1, f, n_2 \rangle$ to denote an edge from n_1 to n_2, labeled with the field f; intuitively, this edge models a reference from an object that n_1 models to a node that n_2 models, along field f. The analysis uses two kinds of edges: the *inside edges* model the heap references created by the analyzed method, while the *outside edges* model the heap references read by the analyzed method from escaped objects. An object escapes if it is reachable from outside the analyzed method (e.g., from one of the parameters);

[1] Normally, the classes `Cell` and `ListItr` would be implemented as inner classes of `List`; for simplicity, our examples uses a flat format.
[2] We use the term *prestate* to denote the state of the program right before the execution of an invoked method.

```
1  class List {
2    Cell head = null;
3    void add(Object e) {
4      head = new Cell(e, head);
5    }
6    Iterator iterator() {
7      return new ListItr(head);
8    }
9  }
10
11 class Cell {
12   Cell(Object d, Cell n) {
13     data = d; next = n;
14   }
15   Object data;
16   Cell   next;
17 }
18
19 interface Iterator {
20   boolean hasNext();
21   Object next();
22 }
23
24 class ListItr implements Iterator {
25   ListItr(Cell head) {
26     cell = head;
27   }
28   Cell cell;
29   public boolean hasNext() {
30     return cell != null;
31   }
32   public Object next() {
33     Object result = cell.data;
34     cell = cell.next;
35     return result;
36   }
37 }
38
39 class Point {
40   Point(float x, float y) {
41     this.x = x; this.y = y;
42   }
43   float x, y;
44   void flip() {
45     float t = x; x = y; y = t;
46   }
47 }
48
49 class Main {
50   static float sumX(List list) {
51     float s = 0;
52     Iterator it = list.iterator();
53     while(it.hasNext()) {
54       Point p = (Point) it.next();
55       s += p.x;
56     }
57     return s;
58   }
59
60   static void flipAll(List list) {
61     Iterator it = list.iterator();
62     while(it.hasNext()) {
63       Point p = (Point) it.next();
64       p.flip();
65     }
66   }
67
68   public static void main(String args[]) {
69     List list = new List();
70     list.add(new Point(1,2));
71     list.add(new Point(2,3));
72     sumX(list);
73     flipAll(list);
74   }
75 }
```

Figure 1: Sample Code for Section 2.



a. `Point` constructor **b.** `Cell` constructor

Figure 2: Analysis results for two simple methods: the constructors of `Point` and `Cell`. In each case, we present the points-to graph at the end of the corresponding method, and the set W of externally visible modified abstract fields.

otherwise, the object is captured. An outside edge always ends in a load node.

For each method, the analysis also computes a set of modified abstract fields. An abstract field is a field of a specific node, i.e., a pair of the form $\langle n, f \rangle$.

The analysis examines methods starting with the leaves of the call graph. The analysis examines each method m without knowing m's calling context; instead, the analysis computes a parameterized result that it later instantiates to take into account the aliasing relation at each call site that may invoke m.

Section 5 contains a complete, formal presentation of the analysis.

2.3 Analysis of the Example

Figure 2.a presents the analysis results for the constructor of the class `Point`. The analysis uses the parameter node `P1` to model the object that the parameter `this` points to. The analysis records the fact that the `Point` constructor mutates fields `x` and `y` of the parameter node `P1`.

Figure 2.b presents the analysis results for the constructor of the class `Cell`. The analysis uses the parameter nodes `P2`, `P3`, and `P4` to model the objects that the three parameters, `this`, `d`, and `n`, point to. The analysis uses inside edges to model the references that the `Cell` constructor creates from `P2` to `P3` and `P4`. The constructor of `Cell` mutates the fields `data` and `next` of the parameter node `P2`.

Figure 3.c presents the analysis results for the method `List.add(Object e)`. The method reads the `head` field of the `this` parameter. The analysis does not know what `this.head` points to in the calling context. Instead, the analysis uses the load node `L1` to model the loaded object and adds the outside edge $\langle$P5, `head`, L1$\rangle$. Next, the method allocates a new `Cell`, that we model with the inside node `I1`, and invokes the cell constructor with the arguments `I1`, `P6`, and `L1`. Based on the points-to graph before the call, and the points-to graph for the invoked constructor (Figure 2.b), the analysis maps each parameter node from the `Cell` constructor to one or more corresponding nodes from the calling context. In this case, `P2` maps to (i.e., stands for) `I1`, `P3` maps to `P6`, and `P4` maps to `L1`. The analysis uses the node mapping to incorporate information from the points-to graph of the `Cell` constructor: the inside edge $\langle$P2, `data`, P3$\rangle$ translates into the inside edge $\langle$I1, `data`, P6$\rangle$. Similarly, we have the inside edge $\langle$I1, `next`, L1$\rangle$. As `P1` stands for `I1`, the

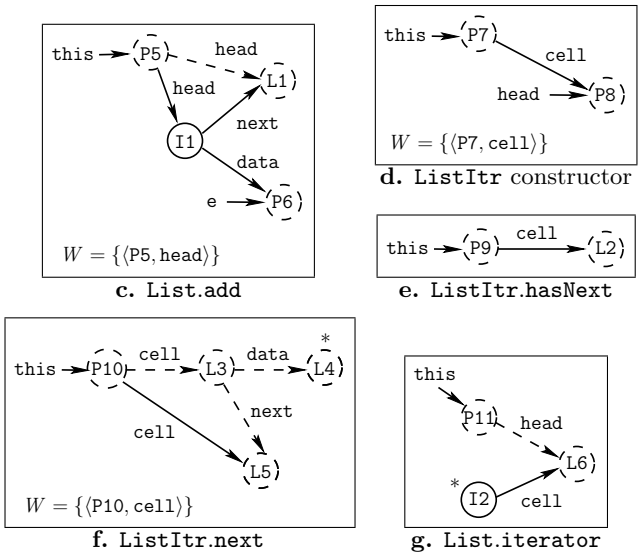


Figure 3: Analysis results for several other simple methods. We use the same conventions as in Figure 2. In addition, we use an asterisk ("*") to mark nodes returned from the analyzed method.

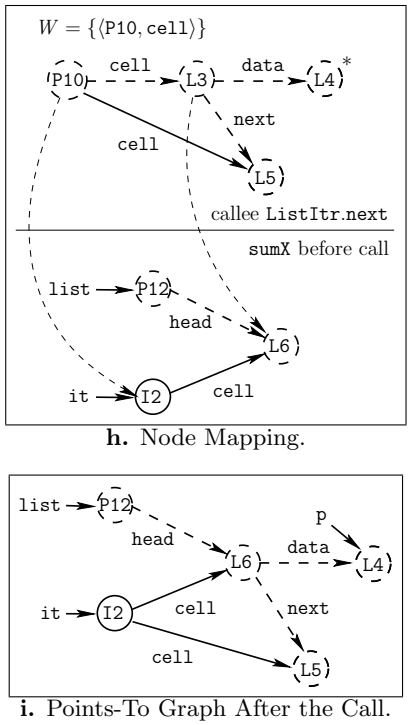


Figure 4: Inter-procedural analysis for the call to `ListItr.next` from line 54, in the first iteration over the loop from lines 53-56.

analysis knows that the two fields of `I1` are mutated. However, `I1` represents a new object, that did not exist in the prestate; hence, we can ignore the mutation of `I1`. Finally, the analysis adds an inside edge from `P5` to `I1`, and records the mutation on the field `head` of `P5`.

Similarly, the analysis examines four other simple methods and obtains the information from parts d, e, f and g of Figure 3.

The analysis of method `Main.sumX(list)` starts with formal parameter `list` pointing to the corresponding parameter node `P12`. The method `sumX` calls `list.iterator()` to obtain an iterator over the list. The analysis takes the analysis result for the `iterator()` method (Figure 2.g), maps `P11` to `P12`, and produces the points-to graph after the call, shown in the lower half of Figure 4.h. The local variable `it` points to the node `I2` returned from `iterator()`. Next, the analysis iterates over the loop from lines 53-56 until it reaches a fixed point.

Figure 4 presents the inter-procedural analysis for the call to `next()`, in the first iteration over the loop body. Initially, the analysis maps the parameter node `P10` to the actual argument `I2`. The analysis matches the callee outside edge $\langle$P10, `cell`, L3$\rangle$ with the inside edge $\langle$I2, `cell`, L6$\rangle$ from before the call, and maps `L3` to `L6`. This ilustrates a key element of our analysis: maching outside edges (read operations) against inside edges (write operations) to detect the nodes that load nodes stand for. Figure 4.i presents the points-to graph after the call: the outside edges from `L3` generated two outside edges from `L6`; we also put local variable `p` to point to the returned node `L4`.[3] The analysis detects the mutation on $\langle$`I2`, `cell`$\rangle$, but, as usual, it ignores all mutation on inside nodes. Section 5.3 contains a complete, formal presentation of the inter-procedural analysis.

Figure 5 presents the inter-procedural analysis for the call to `next()`, in the second iteration over the loop body. The analysis proceeds as in the first iteration, except that we now have more edges and mappings. Figure 5.k presents the resulting points-to graph after the call. As `L3` maps to `L5` (among other nodes), the callee outside edge $\langle$L3, `next`, L5$\rangle$ generates the "loop" outside edge $\langle$L5, `next`, L5$\rangle$. Loop edges occur during the analysis of methods that construct/traverse recursive data structures.

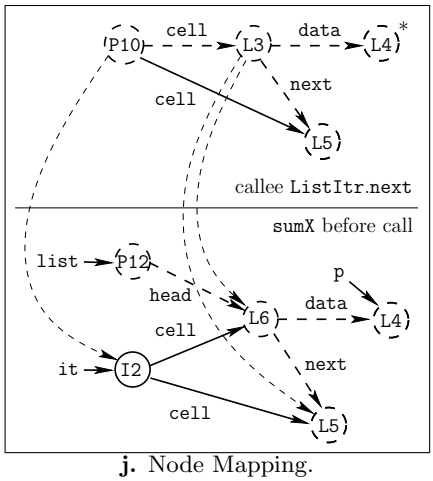
Future iterations over the loop body do not produce new information. The analysis of `Main.flipAll(list)` proceeds almost identically to the analysis of `Main.addX(list)`, obtains the same points-to graphs, but detects mutations on the fields `x` and `y` of the node `L4`.

2.4 Analysis Results

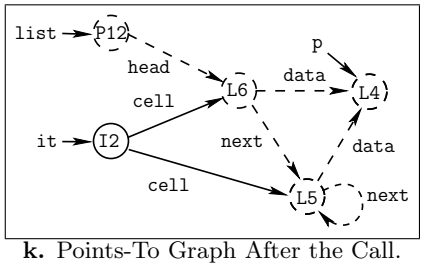
For the method `Main.sumX`, the analysis did not detect any mutation on the only parameter node `P12`, or on any of the load nodes reachable from it. Therefore, the analysis guarantees that the method `sumX` is pure. On the other hand, the analysis detects that the method `Main.flipAll` is not pure, due to the mutations on the node `L4` that is transitively loaded from the parameter `P12`. In addition, the analysis is able to conservatively describe the set of modified locations.

In the points-to graph from Figure 5.k, the outside edges model the references read from nodes reachable from the parameters. Furthermore, as the method `flipAll` does not create any `head`, `next`, and `data` references, it reads only references that exist in the prestate. Therefore, to describe

[3]The attentive reader may be confused by the absence of an outside edge from I2 in Figure 4.i. The inter-procedural analysis is more complex than we present in this simple example. In particular, as we explain in Section 5.3, the inter-procedural analysis has an internal step that simplifies the resulting points-to graph by removing unnecessary edges and nodes.



j. Node Mapping.



k. Points-To Graph After the Call.

Figure 5: Inter-procedural analysis for the call to `ListItr.next` from line 54, in the second iteration over the loop from lines 53-56.

the set of locations that the method `flipAll` modifies, it suffices to describe the paths from `P10` (the only parameter) to `L4`, along outside nodes. These paths are generated by the regular expression `head.next*.data`. Hence, `flipAll` may modify only the prestate locations reachable along a path that matches `list.head.next*.data.(x|y)`. Section 6.2 explains how to compute the regular expressions automatically.

2.5 Using the Analysis Results

Knowing that `Main.sumX(list)` is pure allows us to propagate information about `list` across calls to `Main.sumX(list)`. It also allows us to freely use `sumX` in assertions and specifications.

Even if `Main.flipAll(list)` is impure, we know that it modifies only locations covered by heap paths that match the regular expression `list.head.next*.data.(x|y)`. Therefore, we can still propagate information across calls to `flipAll`, as long as the information refers only to other locations. For example, as none of the list cells matches the aforementioned regular expression (by a simple type reasoning), the list spine itself is not affected, and we can propagate non-emptyness of `list` across calls to `flipAll`.

3. GENERAL MATHEMATICAL NOTATIONS

This paper uses the following notations: "$\{a_0, a_1, \ldots, a_k\}$" represents the set of distinct elements $a_0, a_1, \ldots, a_k$; $\emptyset$ denotes the empty set and $\uplus$ denotes the disjoint set union operator. For any set A, $\mathcal{P}(A)$ is the set of all subsets of A, i.e., $\mathcal{P}(A) = \{B \mid B \subseteq A\}$. "$\{a_i \mapsto b_i\}_{i \in I}$" denotes a partial function f such that $f(a_i) = b_i, \forall i \in I$, and f is undefined in the other points; in particular, "$\{\}$" denotes a partial function that is not defined in any point. If $f : A \to B$ is a function from A to B, $a \in A$, and $b \in B$, "$f\,[a \mapsto b]$" denotes the function that has the value b in the point a, and behaves exactly like f in the other points of the domain A. If $\mu \subseteq A \times B$ is a relation from A to B, and $a \in A$, $\mu(a) = \{b \mid \langle a, b\rangle \in \mu\}$. Furthermore, if $S \subseteq A$, $\mu(S) = \bigcup_{a \in S} \mu(a)$.

4. PROGRAM REPRESENTATION

We present our analysis in the context of a small but non-trivial subset of Java. It is straightforward to extend the analysis to handle the full Java language.

A program consists of a set of classes, *Class*, and a set of methods, *Method*. Each method has a list of parameters, a set of local variables, and a body consisting of a list of instructions. Method m has $k = \mathit{arity}(m)$ parameters: $p_0, p_1, \ldots, p_{k-1}$; the first parameter p_0 is the "this" parameter. *Var* denotes the set of all local variables and parameters. For simplicity, we suppose that all parameters, local variables, object fields and return values have object type; in the analysis implementation, it is straightforward to ignore parameters, local variables, etc. of primitive values.

Each class $C \in \mathit{Class}$ has a set of fields $\mathit{fields}(C) = \{f_0, f_1, \ldots, f_{q-1}\}$. Some fields are static, i.e., attached to a class C, not to a specific instance of C; static fields act as global variables. We distinguish between static and non-static fields by using different instructions for manipulating them.

Figure 6 presents the statements from the programs we analyze. We suppose that prior to the analysis, each application is preprocessed to contain only these statements.

An IF instruction may alter the normal control flow by branching to a specific address in the same method. A CALL instruction "$v_R = v_0.s(\ldots)$" calls the method named s from the class C of the object pointed to by v_0. The parameter passing semantics is call-by-value. Although we did not give any mechanism for calls to native methods, the analysis handles the more general case of *unanalyzable* calls, i.e., calls to methods whose code is unavailable or too expensive to analyze.

In Java, threads are instances of the class `java.lang.Thread`; a thread is started by calling a special native method (`java.lang.Thread.start()`) on the thread object. The body of the newly started thread is the `run` method of the thread object. Equivalently, in our language, we start a thread by executing the THREAD START instruction "`start` v".

CFG_m denotes the *control flow graph* of method m. CFG_m contains an arc from label lb_1 to label lb_2 for every lb_1 and lb_2 that might be consecutive on an execution path inside method m. CFG_m has an isolated entry point entry_m, and a single exit point exit_m. Given a label lb from a method m, $\mathit{pred}(lb)$ is the set of direct predecessors of lb in CFG_m, and $\mathit{succ}(lb)$ is the set of direct successors of lb in CFG_m.

We assume that we have a *conservative call graph* CG: for a given CALL at label lb, $CG(lb)$ contains *all* methods that may be called by that CALL.

In our representation, exceptions are handled explicitly: e.g., there is a null pointer check before each pointer dereference; if we detect an exceptional condition, we allocate an appropriate exception object and we transfer the control to the appropriate exception handler (if any), or to the end

Statement Name	Statement Format	Informal Semantics
COPY	$v_1 = v_2$	copy one local variable into another
NEW	$v =$ `new` C	create a new object of class C; all fields of the new object are initialized to `null`
NEW ARRAY	$v =$ `new` $C[k]$	create an array of k references to objects of class C; all array cells are initialized to `null`
STORE	$v_1.f = v_2$	store a reference into an object field
STATIC STORE	$C.f = v$	store a reference into a static field
ARRAY STORE	$v_1[i] = v_2$	store a reference into an array cell
LOAD	$v_1 = v_2.f$	load a reference from an object field
STATIC LOAD	$v = C.f$	load a reference from a static field
ARRAY LOAD	$v_2 = v_1[i]$	load a reference from an array cell
IF	`if` $(\ldots)$ `goto` a_t	conditional transfer of control to address a_t from the same method (the condition is irrelevant for our analysis; we just suppose it has no heap side effects)
CALL	$v_R = v_0.s(v_1, \ldots, v_j)$	call method named s of object pointed to by v_0
RETURN	`return` v	return from the currently executed method with the result v
THREAD START	`start` v	start the thread pointed to by v

Figure 6: Relevant Statements in the Analyzed Program.

of the current method (otherwise). There is a check after each call, to propage the exception thrown from the invoked method.

5. ANALYSIS PRESENTATION

The analysis processes individual methods from the analyzed program, from the leaves of the call graph to the main method. During the analysis of method m, the scope of the analysis is the method m plus the methods it transitively calls. For each program point inside m, the analysis computes a points-to graph that models the heap at that point. More specifically, for each label lb from method m, the analysis of m computes the points-to graph $\circ A(lb)$ for the program point right before lb, and the points-to graph $A\circ(lb)$ for the program point right after lb. In addition, for each method m, the analysis computes the set W_m that contains all the externally visible abstract fields mutated by m. An abstract field is a pair $\langle n, f\rangle$ that models a mutation of the field f of the parameter/load node n; since we are interested only in externally visible mutations, we ignore mutations on inside nodes (an inside node models objects allocated by the current invocation of the analyzed method).

We express the analysis of method m as a set of standard dataflow equations:

$$\begin{array}{rcl} \circ A(lb) & = & \begin{cases} G^m_{\text{init}} & \text{if } lb = entry_m \\ \bigsqcup\{A\circ(lb') \mid lb' \in pred(lb)\} & \text{otherwise} \end{cases} \\ A\circ(lb) & = & [\![lb]\!](\circ A(lb)) \end{array}$$

In the above equations, G^m_{init} is the points-to graph for the beginning of method m, and $[\![lb]\!]$ is the transfer function attached to the label lb.

Each transfer function $[\![lb]\!]$ takes the points-to graph for the program point right before lb, and produces the points-to graph for the program point right after lb. At the beginning of the analysis of method m, $W_m = \emptyset$. The transfer function $[\![lb]\!]$ may have the side effect of adding a few new elements to the set W_m.

The rest of this section is organized as follows. Section 5.1 defines the abstractions used by the analysis. Section 5.2 presents the intra-procedural analysis, i.e., the initial points-to graph G^m_{init} and the transfer functions $[\![lb]\!]$ for labels lb that do not correspond to analyzable CALLs. Section 5.3 presents the inter-procedural analysis, i.e., the transfer functions $[\![lb]\!]$ for labels that correspond to analyzable CALLs. Section 5.4 gives a high-level description of an algorithm that computes the least fixed point of the analysis equations.

$$n \in Node = INode \uplus PNode \uplus LNode \uplus \{n_{\texttt{GBL}}\}$$

$$\begin{array}{rll} n^I_{lb} & \in INode & \text{inside nodes} \\ n^P_{m,i} & \in PNode & \text{parameter nodes} \\ n^L_{lb} & \in LNode & \text{load nodes} \end{array}$$

$$\langle n, f\rangle \in AField = Node \times Field; \quad \text{abstract fields}$$

$$\begin{array}{rl} I \in IEdges = & \mathcal{P}(Node \times Field \times Node) \\ O \in OEdges = & \mathcal{P}(Node \times Field \times LNode) \\ L \in LocVar = & Var \rightarrow \mathcal{P}(Node) \end{array}$$

$$G \in PTGraph = \quad IEdges \times OEdges \times LocVar \times \mathcal{P}(Node)$$

Figure 7: Sets and notations used by the analysis.

5.1 Sets and Notations

Figure 7 presents the sets and the notations that we use in the analysis presentation.

The analysis uses *nodes* to model objects from the analyzed program. We introduce one *inside* node n^I_{lb} for each label lb that corresponds to a NEW or an ARRAY NEW instruction. n^I_{lb} models *all* objects created by the analyzed scope by executing the instruction from label lb.

There is one *parameter* node $n^P_{m,i}$ for each formal parameter p_i of the analyzed method m. For a given invocation of a method, a parameter node models a single object: the object pointed to by the actual argument.

Some LOAD instructions read references from escaped objects, i.e., objects accessible from outside the analyzed scope. As the analysis examines each method once, without knowing its calling context, the analysis does not know what the

other parts of the program may have written in the fields of the escaped objects. Instead, for each label lb that corresponds to a LOAD/ARRAY LOAD statement that reads from escaped objects, the analysis introduces a *load* node n^L_{lb}; n^L_{lb} models the objects read at label lb from escaped objects.

The parameter/load nodes are essential for our ability to analyze m without knowing the heap at the point where m is called. In the presence of complete information about the calling context, the inside nodes would be enough for modeling the heap. A parameter/load node n is a placeholder for the inside nodes associated with the objects that n models. For each call to method m, the inter-procedural analysis computes a node mapping that disambiguates these placeholders, according to the current calling context.

The special node $n_{\texttt{GBL}}$ models objects that may be accessed by the entire program. We use it to model objects that are read from a static field and objects returned by unanalyzable CALLs.

An abstract field is a pair $\langle n, f\rangle$; it conservatively represents the field f of all objects that n models.

A points-to graph $G \in PTGraph$ is a tuple $G = \langle I, O, L, E\rangle$, consisting of a set of *inside* edges I, a set of *outside* edges O, an *abstract state of local variables* L, and a set of *globally escaped* nodes E.

The edges from the points-to graph model the points-to relation between objects. The *inside edges* from I model the heap references created by the analyzed scope. The *outside edges* from O model the heap references read by the analyzed scope from escaped objects. An outside edge always ends in a load node.

Arrays are just a special kind of objects, and are modeled by nodes too. If an array has elements of a non-primitive type, the values stored in the array cells are addresses of objects. We use edges to represent these heap references. We do not distinguish between individual array cells: the special field [] represents *all* cells of an array.

L models the state of the local variables of the analyzed method: $L(v)$ is the set of nodes that the local variable v may point to. To keep track of the objects returned from an analyzed method m, we introduce a special variable v_{ret}: $L(v_{ret})$ is the set of nodes that m might return.

The last component of a points-to graph, E, contains: 1) the nodes whose address is stored in static fields, 2) the nodes that correspond to started threads, and 3) the nodes passed as arguments to unanalyzable CALLs. These nodes escape the analyzed scope: they are potentially reachable from the entire program; we say that they escape *globally*.

Other escaped nodes include the parameter nodes, the returned nodes, and the special node $n_{\texttt{GBL}}$ ($n_{\texttt{GBL}}$ models the objects returned from unanalyzable CALLs and/or read from a static field). In addition, any node reachable from an escaped node along a path of inside and/or outside edges escapes too. Formally,

DEFINITION 1. *Given a points-to graph $G = \langle I, O, L, E\rangle$, let $e(G)$ be the following boolean predicate on nodes: $e(G)(n)$ is true iff n is reachable from a node from $PNode \cup L(v_{ret}) \cup E \cup \{n_{\texttt{GBL}}\}$, along a (possibly empty) path of edges from $I \cup O$. If $e(G)(n)$, n* escapes from G. *Otherwise, n* is captured in G.

The ordering relation between sets (e.g., I, O, E) is the set inclusion; the associated join operation is the set union. For elements from the set $LocVar$, we use the classic elementwise ordering between functions: $L_1 \sqsubseteq L_2$ iff $\forall v \in Var$, $L_1(v) \subseteq L_2(v)$. The associated join operation is $L_1 \sqcup L_2 = \lambda v.\,(L_1(v) \cup L_2(v))$. Points-to graphs are ordered componentwise:
$\langle I_1, O_1, L_1, E_1\rangle \sqsubseteq \langle I_2, O_2, L_2, E_2\rangle$ iff $I_1 \subseteq I_2$, $O_1 \subseteq O_2$, $L_1 \sqsubseteq L_2$, and $E_1 \subseteq E_2$. $\langle PTGraph, \sqsubseteq\rangle$ is a join semi-lattice with the join operator

$$\langle I_1, O_1, L_1, E_1\ \rangle \sqcup \langle I_2, O_2, L_2, E_2\ \rangle = \\ \langle\ I_1 \cup I_2,\ O_1 \cup O_2,\ L_1 \sqcup L_2,\ E_1 \cup E_2\rangle$$

and the least element $\bot_{PTGraph} = \langle \emptyset,\ \emptyset,\ \lambda v.\emptyset,\ \emptyset\rangle$.

5.2 Intra-procedural Analysis

The points-to graph for the beginning of m is:

$$G^m_{\text{init}} = \langle \emptyset,\ \emptyset,\ \{p_i \mapsto n^P_{m,i}\}_{0 \le i \le k-1},\ \emptyset\rangle$$

where $p_0, p_1, \ldots, p_{k-1}$ are the k parameters of m. Each parameter p_i points to the corresponding parameter node $n^P_{m,i}$; G^m_{init} is otherwise empty. At the beginning of the analysis of m, the method-wide set W_m of mutated abstract fields is initialized to be empty.

Figure 8 presents the transfer functions associated with the labels from the analyzed program; Figure 10 presents an informal graphic representation for some of the transfer functions.

The transfer function $[\![lb]\!]$ takes as argument the points-to graph for the program point just before label lb and returns the points-to graph for the program point right after lb. $[\![lb]\!]$ may also have the side effect of adding a few new elements to W_m, if the instruction from label lb mutates a field of one or more nodes. We define the functions $[\![lb]\!]$ on a case by case basis, based on the instruction from label lb. Figure 8 does not cover the case of an analyzable CALL; we study this case later in Section 5.3.

As a general rule, assignments to variables are *destructive*, i.e., assigning something to v "removes" all the previous values of $L(v)$, while assignments to node fields are *non-destructive*[4]: assigning something to $n_1.f$ does not remove the existing edges that start from n_1. The reason is that a node might represent multiple objects and so, updating $n_1.f$ might not overwrite the edge $\langle n_1, f, n_2\rangle$ because the update instruction and the edge might concern different objects.

The two special labels $entry_m$ and $exit_m$ do not correspond to any concrete instruction. The transfer function for them is naturally the identity function. This is also the case for the labels that correspond to IF, or some other instruction that does not manipulate pointers.

A COPY instruction "$v_1 = v_2$" makes v_1 point to all the nodes that v_2 might point to. As previously mentioned, the analysis "forgets" the previous value of $L(v_1)$. The transfer function for a label lb that corresponds to a NEW instruction "$v = \texttt{new}\ C$" makes v point to the inside node attached to the label lb, n^I_{lb}. Notice that object creation does not generate any effect in our analysis: we are interested only in mutation on the objects from the prestate, i.e., objects that existed at the beginning of the method.

For a STORE instruction "$v_1.f = v_2$", the analysis introduces an f-labeled inside edge between each node pointed to by v_1, and each node pointed to by v_2. The analysis also updates the set W_m to record the mutation on the field f of all non-inside nodes pointed to by v_1. The case of an

[4]An equivalent term is "*weak updates*."

Instruction from label lb	$[\![lb]\!](\langle I, O, L, E\rangle)$	Set of abstract fields added to W_m
$v_1 = v_2$	$\langle I, O, L\left[v_1 \mapsto L(v_2)\right], E\rangle$	$\emptyset$
$v =$ `new` C	$\langle I, O, L\left[v \mapsto \{n^I_{lb}\}\right], E\rangle$	$\emptyset$
$v =$ `new` $C[k]$	$\langle I, O, L\left[v \mapsto \{n^I_{lb}\}\right], E\rangle$	$\emptyset$
$v_1.f = v_2$	$\langle I \cup (L(v_1) \times \{f\} \times L(v_2)), O, L, E\rangle$	$(L(v_1) \setminus \mathit{INode}) \times \{f\}$
$v_1[i] = v_2$	$\langle I \cup (L(v_1) \times \{[]\} \times L(v_2)), O, L, E\rangle$	$(L(v_1) \setminus \mathit{INode}) \times \{[]\}$
$C.f = v$	$\langle I, O, L, E \cup L(v)\rangle$	$\{\langle n_{\texttt{GBL}}, f\rangle\}$
$v_1 = v_2.f$	$\mathit{process\_load}(G,\ v_1,\ v_2,\ f,\ lb)$	$\emptyset$
$v_1 = v_2[i]$	$\mathit{process\_load}(G,\ v_1,\ v_2,\ [],\ lb)$	$\emptyset$
$v = C.f$	$\langle I, O, L\left[v \mapsto \{n_{\texttt{GBL}}\}\right], E\rangle$	$\emptyset$
`if` $(\ldots)$ `goto` a_t	$\langle I, O, L, E\rangle$ (unmodified)	$\emptyset$
$v_R = v_0.s(v_1, \ldots, v_j)$	**Case 1:** analyzable call — studied later in Section 5.3.	
	Case 2: unanalyzable call $\langle I, O, L\left[v_R \mapsto n_{\texttt{GBL}}\right], E \cup \bigcup_{i=0}^{j} L(v_i)\rangle$	$\emptyset$
`return` v	$\langle I, O, L\left[v_{ret} \mapsto L(v)\right], E\rangle$	$\emptyset$
`start` v	$\langle I, O, L, E \cup L(v)\rangle$	$\emptyset$

Figure 8: functions $[\![lb]\!], lb \in \mathit{Label}$. $[\![lb]\!]$ takes the points-to graph $G = \langle I, O, L, E\rangle$ for the program point right before label lb, and produces the points-to graph for the program point after lb. See Figure 10 for an informal graphic representation of some of the transfer functions.

$$
\begin{array}{l}
\mathit{process\_load}(\langle I, O, L, E\rangle,\ v_1,\ v_2,\ f,\ lb) = \\
\quad \textbf{let}\ \ A \ = \ \{n \in \mathit{Node} \mid \exists n_1 \in L(v_2), \langle n_1, f, n\rangle \in I\} \\
\qquad\ \ B \ = \ \{n \in L(v_2) \mid e(G)(n)\}\ \ \textbf{in} \\
\qquad \textbf{if}\ (B = \emptyset) \\
\qquad\quad \textbf{then}\ \langle I,\ O,\ L\left[v_1 \mapsto A\right],\ E\rangle \\
\qquad\quad \textbf{else}\ \ \langle I,\ O \cup (B \times \{f\} \times \{n^L_{lb}\}),\ L\left[v_1 \mapsto \left(A \cup \{n^L_{lb}\}\right)\right],\ E\rangle
\end{array}
$$

Figure 9: *process_load*. Its arguments are, in order, the points-to graph before the load ($G = \langle I, O, L, E\rangle$), the variable v_1 we load into, the variable v_2 we load from, the loaded field f, and the label lb of the LOAD instruction "$v_1 = v_2.f$". It returns the points-to graph after the instruction.

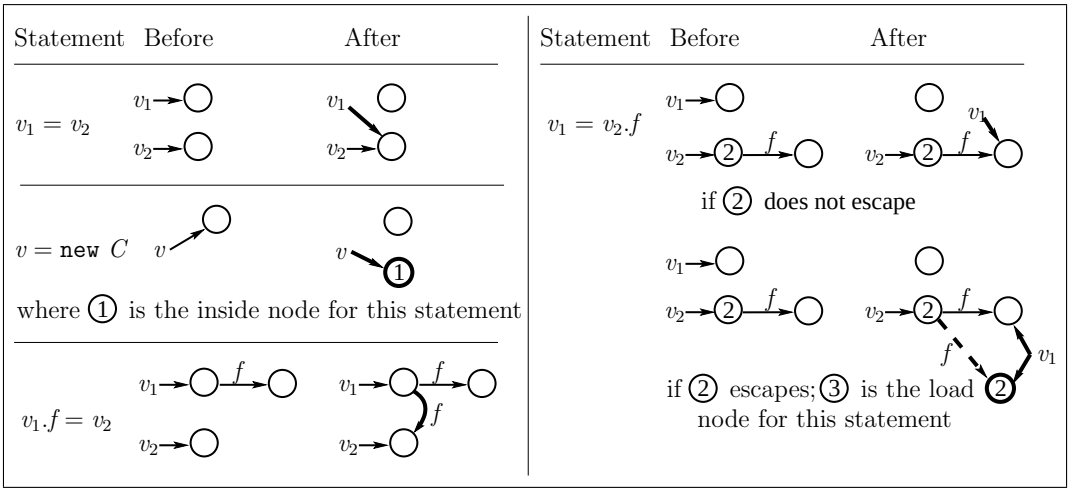


Figure 10: Informal graphic representation of the transfer functions for non-call statements. We use continuous circles for all types of nodes, continuous lines for the inside edges, and dashed lines for the outside edges. We use bold circles/lines for potentially new nodes/edges.

ARRAY STORE instruction "$v_1[i] = v_2$" is similar, except that we use the special field [] that models the references coming from all the cells of the array. The analysis records the mutation on the special field [] for all concerned nodes. For a STATIC STORE "$C.f = v$," we add all nodes pointed to by v to the set of globally escaped nodes E, and we record a mutation on the static field f.

The transfer function for a LOAD instruction "$v_1 = v_2.f$" uses the auxiliary function *process_load* from Figure 9. After the instruction, v_1 points to all nodes pointed to by f-labeled inside edges starting from nodes in $L(v_2)$; in Figure 9, we collect these nodes in the set A. If we load from nodes that escape the analyzed scope, i.e., $B \neq \emptyset$ in Figure 9, v_1 also points to the load node n^L_{lb}. In this case, the analysis introduces an f-labeled outside edge from every escaped node we read from to n^L_{lb}. Later, when we analyze calls to m, we use these outside edges to detect the nodes that the placeholder n^L_{lb} stands for. The transfer function for an ARRAY LOAD instruction is identical, except that it uses the special field [].

An unanalyzable CALL "$v_R = v_0.s(v_1, \ldots, v_j)$" makes its arguments reachable from unanalyzed parts of the program. Therefore, the analysis adds all nodes pointed to by $v_0, \ldots, v_j$ to E, the set of globally escaped nodes. Also, in the points-to graph after the unanalyzable CALL, v_R points to the special node $n_{\texttt{GBL}}$ that models objects potentially reachable from the entire program. Similarly, for a START THREAD instruction "`start` v", the analysis adds all nodes pointed to by v to E. Finally, for a RETURN instruction "`return` v", the special variable v_{ret} is set to point to the returned nodes.

5.3 Inter-procedural Analysis

Consider a CALL instruction at label lb, "$v_R = v_0.s(v_1, \ldots, v_j)$", and let $callee \in CG(lb)$ be one of the possible callees. The inter-procedural analysis uses the points-to graph G before the CALL and the points-to graph $G_{callee} = \circ A(exit_{callee})$ for the end of *callee*, and computes a points-to graph for the program point after the CALL, valid in the case when *callee* is called. If there are several possible callees, we conservatively join the points-to graphs computed for each of them.

The inter-procedural analysis has four steps:

1. We compute a mapping relation $\mu' \subseteq Node \times Node$ that maps nodes from G_{callee} to nodes that appear in the final graph; μ' disambiguates as many parameter and load nodes as possible.

2. We use the mapping μ' to combine G and G_{callee}. An important aspect of this step is that each node n from G_{callee} is projected through the mapping μ', i.e., intuitively, n is replaced with the nodes from $\mu'(n)$.

3. We simplify the resulting points-to graph by removing superfluous load nodes and outside edges.

4. We use the information about the abstract fields mutated by the callee (i.e., the set W_{callee}) to update the set W_m of abstract fields mutated by m.

We describe these steps in the next paragraphs.

Construction of the Node Mapping

We start by computing a "core" mapping μ that disambiguates as many parameter and load nodes from the calee as possible. Let $G = \langle I, O, L, E\rangle$, and $G_{callee} = \langle I_{callee}, O_{callee}, L_{callee}, E_{callee}\rangle$. We define μ as the least fixed point of the following constraints:

$$L(v_i) \subseteq \mu(n^P_{callee,i}),\ \forall i \in \{0, 1, \ldots j\} \tag{1}$$

$$\frac{\langle n_1, f, n_2\rangle \in O_{callee},\ \langle n_3, f, n_4\rangle \in I,\ n_3 \in \mu(n_1)}{n_4 \in \mu(n_2)} \tag{2}$$

$$\frac{\begin{array}{c}\langle n_1, f, n_2\rangle \in O_{callee},\ \langle n_3, f, n_4\rangle \in I_{callee},\\ (\mu(n_1) \cup \{n_1\})\ \cap\ (\mu(n_3) \cup \{n_3\}) \neq \emptyset,\\ (n_1 \neq n_3) \vee (n_1 \in LNode)\end{array}}{\mu(n_4) \cup (\{n_4\} \setminus PNode) \subseteq \mu(n_2)} \tag{3}$$

Constraint 1 maps each parameter node $n^P_{callee,i}$ to the nodes pointed to by v_i, the i^{th} argument passed to *callee*; these are the nodes from the set $L(v_i)$. The other two constraints extend the mapping by matching outside edges (i.e., references read by LOAD instructions) against inside edges (i.e., references created by STORE instructions).[5]

Constraint 2 handles the case when the callee reads references created by the caller. It matches an outside edge $\langle n_1, f, n_2\rangle \in O_{callee}$ from the callee against an inside edge $\langle n_3, f, n_4\rangle \in I$ from the caller, in the case when n_1 might represent n_3, i.e., $n_3 \in \mu(n_1)$. Figure 11.a presents a graphic representation of this situation. As n_1 might be n_3, the outside edge read from n_1 might be the inside edge $\langle n_3, f, n_4\rangle$, and the load node n_2 might be the node n_4. Hence, the analysis maps n_2 to n_4, i.e., $n_4 \in \mu(n_2)$

Constraint 3 matches an outside edge from the callee against an inside edge from the callee, i.e., from the same scope. This constraint deals with the aliasing present in the calling context. Consider an outside edge $\langle n_1, f, n_2\rangle \in O_{callee}$ and an inside edge $\langle n_3, f, n_4\rangle \in I_{callee}$. Figure 11.b presents a graphic representation of the case where n_1 and n_3 might represent the same node n_5. In this case, n_2 might be n_4. Therefore, we enforce $n_4 \in \mu(n_2)$. Also, as n_4 is a node from the callee, it might be a node placeholder that represents some other nodes. Therefore, node n_2 might represent not only n_4 but also the nodes represented by n_4. The analysis updates the mapping to record this too: $\mu(n_4) \subseteq \mu(n_2)$. The same reasoning is valid if n_1 might represent n_3, i.e.,

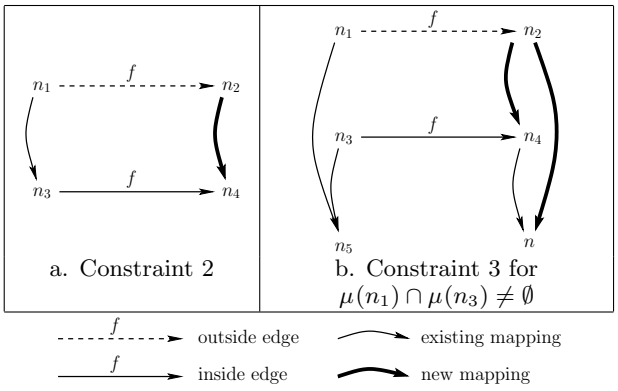


Figure 11: Graphic representation of Constraint 2 and Constraint 3.

[5] A real implementation would use Constraint 1 to initialize the mapping and would next iterate only over constraints 2 and 3 until a fixed point is reached.

$n_3 \in \mu(n_1)$, or n_3 might represent n_1, i.e., $n_1 \in \mu(n_3)$. Constraint 3 unifies these three cases in a single condition:

$$(\mu(n_1) \cup \{n_1\}) \cap (\mu(n_3) \cup \{n_3\}) \neq \emptyset$$

The third part of the precondition, "$(n_1 \neq n_3) \vee (n_1 \in LNode)$", reduces the applicability of Constraint 3, and avoids fake mappings. The correctness proof from [31] shows that this condition does not prevent the analysis from detecting all real mappings.

We compute the final mapping μ' by extending the "core" mapping μ with a mapping from each non-parameter node to itself:

$$\forall n,\ \mu'(n) = \mu(n) \cup (\{n\} \setminus PNode)$$

μ' maps nodes from G_{callee} to nodes that appear in the points-to graph after the call. Inside nodes model objects created by the analyzed scope and should appear in the points-to graph after the call; for inside nodes, μ' is the identity relation. We already know all the nodes that the parameter nodes stand for. Hence, the parameter nodes are unnecessary in the resulting points-to graph, and the map extension ignores them. This is also the reason why in Constraint 3, instead of $\mu(n_4) \cup \{n_4\} \subseteq \mu(n_2)$, we use $\mu(n_4) \cup (\{n_4\} \setminus PNode) \subseteq \mu(n_2)$: as we do not want the callee parameter nodes to appear in the resulting points-to graph, we avoid creating mappings to them.

Unlike parameter nodes, load nodes are generally not fully disambiguated. Each load node is a placeholder for the nodes that a specific LOAD instruction loads from an escaped node. That escaped node might remain an escaped node even in the points-to graph after the CALL. In a first phase, we preserve all load nodes. We show later in this section how to remove some of them.

Combining the Points-to Graphs

After we obtain the node mapping μ', we use it to combine the points-to graph G before CALL with the points-to graph G_{callee} for the end of *callee*. We obtain the points-to graph after the CALL, valid if *callee* is called. Formally, we construct the points-to graph $G_2 = \langle I_2, O_2, L_2, E_2 \rangle$ defined by the following equations:

$$I_2 = I \cup \bigcup_{\langle n_1, f, n_2 \rangle \in I_{callee}} \mu'(n_1) \times \{f\} \times \mu'(n_2)$$

$$O_2 = O \cup \bigcup_{\langle n, f, n^L \rangle \in O_{callee}} \mu'(n) \times \{f\} \times \{n^L\}$$

$$L_2 = L\left[v_R \mapsto \mu'(L_{callee}(v_{ret}))\right]$$

$$E_2 = E \cup \mu'(E_{callee})$$

The above equations require some explanation. The heap references that existed before the call might exist after the call too. Hence, all inside edges from G appear in the points-to graph after the call. In addition, if *callee* created the heap edge $\langle n_1, f, n_2 \rangle$, where n_1 may be any node from $\mu'(n_1)$, and n_2 may be any node from $\mu'(n_2)$, then *callee* might have created any of the inside edges from the set $\mu'(n_1) \times \{f\} \times \mu'(n_2)$. All these edges appear in the points-to graph after the call, as well.

Similarly, the set of outside edges after the call is the union of the set of outside edges right before the call, O, and the *semi*-projection through μ' of the outside edges from the end of *callee*: we project only the start node of an outside edge, the target remains unmodified. Here's why: an outside edge $\langle n, f, n^L_{lb} \rangle$ from the callee models the action of loading the field f from the escaped node n, action done by the LOAD instruction at label lb; the load node n^L_{lb} models the objects read in that instruction. As n may be any of the nodes from $\mu'(n)$, the read operation may take place from any of these nodes, hence the need for projecting the node n. However, n^L_{lb} has the same meaning: it models the objects read in the instruction at label lb. Hence, the analysis does not project n^L_{lb} through μ'.

The abstract state of the local variables after the call is similar to the state L before the call, except that now v_R — the local variable that receives the value returned from the callee — points to the nodes returned from *callee*, i.e., $\mu'(L_{callee}(v_{ret}))$. The projection is necessary because some of the returned nodes may be placeholders from *callee* that μ' disambiguates. Finally, the set of globally escaped nodes is the union of the set of directly escaped nodes before the call, E, and the set of nodes that directly escape in callee, $\mu'(E_{callee})$.

Points-to Graph Simplification

We simplify the points-to graph for the program point after the call by removing all captured load nodes (together with all adjacent edges), as well as all outside edges that start in a captured node.

Intuitively, a load node is a placeholder for the (unknown) nodes loaded from an escaped node. There is no need for a load node when we load from captured nodes: as only the analyzed scope access a captured node, the analysis knows all the nodes loaded from it. If a points-to graph G contains a captured load node n^L_{lb}, all the nodes that we loaded n^L_{lb} from[6] are captured too. Therefore, we can remove all captured load nodes. Similarly, we can remove all outside edges that start from a captured node.

Modified Abstract Fields

We update the set W_m of modified abstract fields from the caller m by adding to it all elements from the following set:

$$\bigcup_{\langle n, f \rangle \in W_{callee}} ((\mu'(n) \setminus INode) \cap N) \times \{f\}$$

where N is the set of nodes that appear in the simplified points-to graph. We use the mapping μ' to project each node modified by the callee. As usual, we ignore inside nodes; we also use the set intersection "$\cap N$" to ignore nodes that have been removed by the points-to graph simplification.

5.4 Analysis Algorithm

As we prove in [31], all transfer functions are monotonic. For each analyzed program, the number of nodes we can define is bounded: we have one parameter node for each formal parameter, one inside node for each NEW instruction, at most one load node for each LOAD instruction, etc. By consequence, $PTGraph$ is finite and there is no infinite ascending chain in $\langle PTGraph, \sqsubseteq \rangle$. Hence, we can solve the dataflow equations with an iterative fixed point algorithm.

We recommend using a variant of the "Iterating Through Strong Components" algorithm [30, Chapter 6]. Our algorithm contains an outer loop for the inter-procedural analysis, and nested inside it, an inner loop for the intra-procedural

[6] I.e., the nodes that point to n^L_{lb} through some outside edge.

analysis.

The inter-procedural computation processes the strongly connected components of the call graph, i.e., the groups of mutually recursive methods,[7] in increasing topological order, i.e., from the leaves of the call graph to the main method. For each such set of mutually recursive methods, the algorithm uses a worklist to iterate over the set of methods until it reaches the least fixed point. At the beginning of the processing for a strongly connected component, the worklist contains all the methods from that component. In each iteration, the algorithm takes a method from the worklist and calls the inner computation, i.e., the intra-procedural computation, to analyze the method. If the points-to graph for the end of the method changed, all the possible callers of the method that are in the current strong component are added to the worklist. The inter-procedural computation for a component terminates when the worklist is empty.

The intra-procedural computation is similar to the inter-procedural computation: it processes the strongly connected components of the control flow graph for the analyzed method in decreasing topological order, i.e., from the beginning of the method toward its end, and iterates over each component by using a worklist.

Complexity: Let n be the size of the analyzed program. The analysis computes points-to graphs for $\mathcal{O}(n)$ program points. The height of the lattice of points-to graphs is $\mathcal{O}(n_a^2 n_f)$, where n_a, the number of nodes, and n_f, the total number of fields, are both $\mathcal{O}(n)$. Most transfer functions can easily be implemented in polynomial complexity. The only difficult operation is the construction of the inter-procedural node mapping. However, notice that the inter-procedural analysis monotonically computes mappings of at most n_a^2 elements. Therefore, the worst-case complexity is big, at least $\mathcal{O}(nn_a^4 n_f)$, but still polynomial in the size of the program. In practice, we have used this pointer analysis to analyze all SpecJVM applications, stack allocate captured objects, and remove synchronization on thread-local objects [36].

6. USE OF ANALYSIS RESULTS

After the analysis terminates, for each analyzable method m, we can use the points-to graph $G = \langle I, O, L, E\rangle$ for the end of m, and the set W_m of modified abstract fields to infer method purity, write effects, read-only parameters and safe parameters. We explain each such application in the next paragraphs.

6.1 Method Purity

To check whether m is pure, we compute the set A of nodes that are reachable in G from parameter nodes, along outside edges. We also compute the set B of all globally escaped nodes, i.e., nodes that are reachable from $E \cup \{n_{\texttt{GBL}}\}$; these are the nodes that are potentially reachable, and hence mutated, from the entire program, e.g., by native methods, hemce, we cannot guarantee anything about them.

The method m is pure iff for any node $n \in A$, 1) n does not escape globally ($n \notin B$) and 2) no field of n is mutated, i.e., there is no field f such that $\langle n, f\rangle \in W_m$.

For constructors, we can follow the JML convention of allowing a pure constructor to mutate fields of the "this" object: it suffices to ignore all modified abstract fields for the parameter node $n^P_{m,0}$ that models the "this" object.

[7] In practice, many of these groups are singletons.

6.2 Write Effects

We can infer regular expressions that describe all the prestate locations modified by m as follows: we construct a finite state automaton F with the following states: 1) all the nodes from the points-to graph G, 2) an initial state s, and 3) an accepting state t. Each outside edge from G generates a transition in F, labeled with the field that labels the outside edge. For each parameter p_i of m, we create a transition from s to the corresponding parameter node, and label it with the parameter p_i. Also, if $n_{\texttt{GBL}}$ appears in G, we create a transition from s to $n_{\texttt{GBL}}$ and label it with the empty string.[8] For each mutated abstract field $\langle n, f\rangle$, we add a transition from n to the accepting state t, and label it with the field f.

In addition, for each globally lost node n (see above), we add a transition from n to t, and label it with the special field REACH. If P is a heap path (i.e., a series of fields, separated by dots, starting in a parameter or a static field), P.REACH matches all objects that are transitively reachable from an object that matches P.

The regular expression that corresponds to the constructed automaton F describes all modified prestate locations. We can use automaton-minimization algorithms to try to reduce the size of the generated regular expression.

Note: The generated regular expression is valid if G does not contain an inside edge and a load edge with the same label. This condition guarantees that the heap references modeled by the outside edges exist in the prestate (the regular expressions are supposed to be interpreted in the context of the prestate). An interesting example that exhibits this problem is presented in [32]. If this "bad" situation occurs, we conservatively generate a regular expression that covers all nodes reachable from all parameters, with the help of the REACH field. In practice, we found that this case is very rare: most of the methods do not read and mutate the same field.

6.3 Read-Only Parameters

A parameter p_i is *read-only* iff none of the locations transitively reachable from p_i is mutated. To check this, we compute the set S_1 that contains the corresponding parameter node $n^P_{m,i}$, and all the load nodes reachable from $n^P_{m,i}$ along outside edges. Parameter p_i is read-only iff there is no abstract field $\langle n, f\rangle \in W_m$ such that $n \in S_1$.

6.4 Safe Parameters

A parameter is *safe* if it is read-only and the method m does not create any new externally visible heap paths to an object transitively reachable from the parameter.

Suppose p_i is a read-only parameter. To detect whether p_i is also safe, we compute, as before, the set S_1 that contains the corresponding parameter node $n^P_{m,i}$, and all the load nodes reachable from $n^P_{m,i}$ along outside edges. Because p_i is a read-only parameter, none of the nodes from S_1 escapes globally or is mutated.

We also compute the set S_2 of nodes reachable from the parameter nodes and/or from the returned nodes, along inside/outside edges. Notice that S_2 contains all those nodes from G that may be reachable from the caller after the end of m. Therefore, to create a new externally visible path to an object transitively reachable from p_i, one needs to create an edge that starts in an object modeled by a node from S_2

[8] In an automaton, such a transition can always be performed, without consuming any input.

and ends in an object modeled by a node from S_1. Hence, parameter p_i is safe if there is no inside edge from a node in S_2 to a node in S_1.

7. EXPERIENCE

7.1 Implementation

We implemented our analysis in the MIT Flex compiler infrastructure [1], a static compiler for Java bytecode. To increase the analysis precision (i.e., prevent edges that do not correspond to any heap references), we manually provide the points-to graphs for several common native methods. Also, we attach type information to nodes, in order to prevent type-incorrect edges, and avoid inter-procedural mappings between nodes of conflicting types.

7.2 Checking Purity of Data Structure Consistency Predicates

We ran our analysis on several benchmarks borrowed from the Korat project [3, 27]. Korat is a tool that generates non-isomorphic test cases up to a finite bound. Korat's input consists of 1) a type declaration of a data structure, 2) a finitization (e.g., at most 10 objects of type `A`, and 5 objects of type `B`), and 3) `repOk`, a pure boolean predicate written in Java that checks the consistency of the internal representation of the data structure. Given these inputs, Korat generates all non-isomorphic data structures that satisfy the `repOk` predicate. Korat does so efficiently, by monitoring the execution of the `repOk` predicate and back-tracking only over those parts of the data structure that `repOk` actually reads.

Korat relies on the purity of the `repOk` predicates but cannot statically check this. In general, writing `repOk`-like predicates is considered good software engineering practice; during the development of the data structure, programmers can write assertions that invoke `repOk` and check the consistency of the data structure at runtime. Of course, programmers do not want assertions to change the semantics of the program, other than aborting the program when it violates an assertion. Therefore, the use of `repOk` in assertions provides additional motivation for checking the purity of `repOk` methods.

We analyzed the `repOk` methods for the following data structures:

`BinarySearchTree` Binary tree that implements a set of comparable keys.

`DisjSet` Array-based implementation of the fast union-find data structure, using path compression and rank estimation heuristics to improve efficiency of find operations.

`HeapArray` Array-based implementation of the heap (priority queues) data structure.

`BinomialHeap` and `FibonacciHeap` Dynamic data structures that also implement heaps, but differ in complexity for certain operations.

`LinkedList` Implementation of doubly-linked lists in the Java Collections Framework, a part of the standard Java libraries.

`TreeMap` Implementation of the `Map` interface using red-black trees.

`HashSet` Implementation of the `Set` interface, backed by a hash table.

Classic textbooks on algorithms and data structures, e.g., [12], present a detailed algorithmic description of all these data structures. `LinkedList`, `TreeMap`, and `HashSet` are from the standard Java Library. The only change the Korat developers performed was to add the corresponding `repOk` methods. We present the `repOk` method for `BinarySearchTree` in Appendix A. The source code for the other `repOk` methods has similar complexity. As the example from Appendix A shows, the `repOk` methods use complex auxiliary data structures: sets, linked lists, wrapper objects, etc. Checking the purity of these methods is beyond the reach of simple purity checkers that prohibit pure methods to call impure methods, or to do any heap mutation.

The first problem we faced while analyzing the data structures is that our analysis is a whole-program analysis that operates under a closed world assumption: in particular, it needs to know the entire class hierarchy in order to infer the call graph. Therefore, we should either 1) give the analysis a whole program (clearly impossible in this case), or 2) describe the rest of the world to the analysis. In our case, we need to describe to the analysis the objects that can be put in the data structures. The methods that our data structure implementations invoke on the data structure elements are overriders of the following methods:

```
java.lang.Object.equals
java.lang.Object.hashCode
java.util.Comparable.compareTo
java.lang.Object.toString
```

We call these methods, and all methods that override them, *special* methods. We specified to the analysis that all special methods are pure. Moreover, these methods do not introduce new externally visible aliasing: any new externally visible path requires either creating an edge from the prestate (thus violating purity), or creating a path from a returned object. The methods `hashCode`, `equals`, and `compareTo` return primitive data, not objects; the method `toString` returns an immutable `java.lang.String` that, we assume, cannot generate new aliasing.

Therefore, the aforementioned special methods (and their overriders) are pure, and do not create new externally visible paths. Hence, the analysis can simply ignore calls to these methods (even dynamically dispatched calls).

We ran the analysis and analyzed the `repOk` methods for all the data structures, and all the methods transitively called from these methods. The analysis was able to verify that all `repOk` methods mutate only new objects, and are therefore pure. On a Pentium 4 @ 2.8Ghz with 1Gb RAM, our analysis took between 3 and 9 seconds for each analyzed data structure.

Of course, our results are valid only if all of the special methods are indeed pure. Our tool tries to verify that this is indeed true for all special methods that the analysis encountered. Unfortunately, some of these methods use caches for performance reasons, and are not pure. For example, several classes cache their hashcode; other classes cache more complex data, e.g., `java.util.AbstractMap` caches its set of keys and entries (these caches are nullified each time a map update is performed).

Fortunately, our analysis can tell us which memory locations the mutation affects. We manually examined the output of the analysis, and checked that all the fields mutated by impure special methods correspond to caching.

7.3 Discussion

From a theoretical point of view, our analysis is sound. However, in order to analyze complex data structures that use the real Java library, we had to sacrifice soundness to obtain a practical tool. More specifically, we had to trust that the caching mechanism used by several classes from the Java library is sound, i.e., it is just a performance issue. We believe that making reasonable assumptions about the unknown code in order to check complex known code is a good tradeoff. As our experience shows, knowing why exactly a method is impure is very useful in practice: this feature allows us to identify (and ignore) benign mutation related to caching.

7.4 Pure Methods in the Java Olden Benchmark Suite

We also ran the purity analysis on all the applications from the Java Olden benchmark suite [6,7]. Table 1 presents a short description of the applications we analyzed. They are all standalone applications. On a Pentium 4 @ 2.8Ghz with 1Gb RAM, the analysis time ranges from 3.4 seconds for `TreeAdd` to 7.2 seconds for `Voronoi`. In each case, the analysis processed all methods, user and library, that may be transitively invoked from the `main` method.

Table 2 presents the results of our purity analysis. For each application, we counted the total number of methods (user and library), and the total number of user methods. For each category, we present the percentage of pure methods, as detected by our analysis. Following the JML convention, we consider that constructors that mutate only fields of the "this" objects are pure. As the data from Table 2 shows, our analysis is able to find large numbers of pure methods in Java applications. Most of the applications have similar percentages of pure methods, because most of them use the same library methods. The variation is much larger for the user methods, ranging from 31% for `Power` to 89% for `Perimeter`.

Application	Description
`BH`	Barnes-Hut N-body solver
`BiSort`	Bitonic Sort
`Em3d`	Models the propagation of electromagnetic waves through three dimensional objects
`Health`	Simulates a health-care system
`MST`	Computes the minimum spanning tree in a graph using Bentley's algorithm
`Perimeter`	Computes the perimeter of a region in a binary image represented by a quadtree
`Power`	Maximizes the economic efficiency of a community of power consumers
`TSP`	Solves the traveling salesman problem using a randomized algorithm
`TreeAdd`	Recursive depth-first traversal of a tree to sum the node values
`Voronoi`	Computes a Voronoi diagram for a random set of points

Table 1: Applications from the Java Olden Benchmark Set.

Application	All Methods		User Methods	
	count	% pure	count	% pure
`BH`	264	55%	59	47%
`BiSort`	214	57%	13	38%
`Em3d`	228	55%	20	40%
`Health`	231	57%	27	48%
`MST`	230	58%	31	54%
`Perimeter`	236	63%	37	89%
`Power`	224	53%	29	31%
`TSP`	220	56%	14	35%
`TreeAdd`	203	58%	5	40%
`Voronoi`	308	62%	70	71%

Table 2: Percentage of Pure Methods in the Java Olden benchmarks. For each application, we present the total number of user and library methods, the percentage of them that are pure, the number of user methods, and the percentage of pure user methods.

8. RELATED WORK

Modern research on effect inference stems from the seminal work of Gifford, Lucassen, and Jouvelot on type and effect systems [19,26]. Most of the previous work on effect inference was done in the context of type systems and/or type inference, and mostly for functional languages. In contrast, we apply dataflow analysis techniques for purity checking of Java programs. Still, there are many common techniques, e.g., the construction of inter-procedural node mapping from our algorithm has a flavor of the unification algorithm used in type inference.

Although the original work of Gifford and Luccasen was motivated by applications in program parallelization, most of the current work on effects is done in the context of program specification and verification. Two very popular such projects are the Java Modeling Language (JML) [23], and the Extended Static Checker for Java (ESC/Java) [17].

JML is a Behavioral Interface Specification Language for Java, used as a common specification language in many research projects [5]. The annotations provided by the user are used for static program verification [34] or for generating runtime assertions. Methods can be invoked from the JML annotations, provided they are pure. JML also allow the user to specify "`assignable`" locations, i.e., locations that a method can mutate [29]. Currently, the purity and `assignable` clauses are either not checked at all or are checked using very conservative analyses: a method is pure iff 1) it does not do I/O, 2) it does not write any heap field,[9] and 3) it does not invoke impure methods [22].

ESC/Java is a tool for checking properties of Java programs. ESC/Java requires annotations in a specification language that is almost identical to JML. ESC/Java uses a theorem prover to do modular checking of the provided annotations. While checking a method body, ESC/Java assumes that the callers of the method satisfy their specification. Since ESC/Java also checks these callers, it ensures that all methods satisfy their specifications. A major source

[9]Constructors are treated in a special way.

of unsoundness in ESC/Java is the fact that the tool uses purity and modifies annotations, but does not check them.

There are two categories of approaches to solve this problem: the first category relies on user-provided annotations; the second category, including our approach, relies on program analysis. An interesting approach from the first category is the work of Leino et al. on data groups [21, 24]. Other approaches in this category use region types [14, 33] and/or ownership types [4, 9] to specify effects at the granularity of regions, respectively at the granularity of ownership boundaries. In general, annotation based approaches are well suited for modular checking; they also provide abstraction mechanisms to hide representation details.

The analysis-based approach is appealing because it does not require additional user annotations. Even in situations where annotations are desired (e.g., to facilitate modular checking), static analysis can still be used to give the user a hint of what the annotations should look like. We briefly discuss two related static analyses.

ChAsE [8] is a syntactic tool designed by Cataño and Huisman for modular checking of JML `assignable` clauses. For each method, the tool traverses the method code and collects write effects, using the `assignable` clauses from the specification of the invoked methods. Although lightweight and useful in many practical situations, ChAsE is an *unsound* syntactic tool; in particular, unlike our analysis, it does not keep track of the values / points-to relation of variables and fields, and ignores all aliasing.

Spoto and Poll [32] propose an abstract interpretation [13] based static analysis that detects mutated locations. Their paper [32] contains compelling evidence that a static analysis for this purpose should propagate not only the set of mutated locations, but also information about the new values stored in those locations; otherwise, the analysis results are either unsound or overly-conservative. Our analysis uses the set of inside edges to keep track of the new value of pointer fields. Unfortunately, we are unaware of an implementation of the analysis of Spoto and Poll.

The Fugue [15] protocol checker is another tool that could benefit from the use of our analysis. Fugue tracks the correct usage of finite state machine-like protocols. Fugue requires user annotations in a rich type system that specifies the state of the tracked objects on method entry/exit. All aliasing to the tracked objects must be statically known. Many library methods 1) do not do anything relevant to the checked protocol, and 2) are too tedious to annotate. In addition, to promote code reuse, Fugue attempts to support library methods that work with both 1) tracked objects, and 2) objects whose aliasing may not be fully known at compile time. Therefore, Fugue tries to find "[`NonEscaping`]" parameters that are equivalent to our safe parameters. The current analysis/type checking algorithm from Fugue is very conservative as it does not allow a reference to a "[`NonEscaping`]" object to be stored in fields of locally captured objects (e.g., iterators).

Model checking of Java programs [10, 35] could also benefit from our analysis. For example, the interleavings of two pure methods from two distinct threads are irrelevant.[10] The model checker can use this information to reduce the search space. Corbett [11] uses a related shape analysis to reduce the finite state models of multithreaded Java programs by identifying thread-local objects. Interleavings of operations on thread-local objects are irrelevant.

Ernst and Birka [2] proposed Javari, i.e., "Java with reference immutability", an extension to Java that allows the programmer to specify read-only parameters (called `const` in Javari). A type checker then checks the programmer annotations. Read-only annotations for parameters are a great documentation asset, and can catch many practical bugs related to unintended mutation. To cope with caches in real applications, Javari allows the programmer to declare `mutable` fields. Such fields can be mutated even when they belong to a `const` object. Of course, the `mutable` annotation must be used with extreme caution. We encountered the same problem when analyzing real Java programs: many methods are impure simply because they use caching for performance issues. To make the tool practical, we expose the mutation on caches to the programmer, and allow the programmer to judge whether this mutation is allowed or not. Our tool could be a perfect companion for Javari: one can imagine using our tool to infer the read-only parameters for legacy code. A programmer can then refine these annotations and/or do small program changes to increase the number of read-only parameters.

Other researchers, e.g. [18, 28], have already considered the use of pointer analysis while infering side effects. However, unlike previous analyses, our analysis uses separate abstractions (the inside node) for the objects allocated by the current invocation of the analyzed method. Therefore, our analysis focuses on prestate mutation, and supports pure methods that mutate newly allocated objects.

9. CONCLUSIONS AND FUTURE WORK

Recognizing method purity is important for a variety of program analysis and understanding tasks. We present the first implemented method purity analysis that is capable of recognizing pure methods that mutate newly allocated objects, including encapsulated objects that do not escape their creating method. Because this analysis produces a precise characterization of the accessed region of the heap, it can also recognize generalized purity properties such as read-only and safe parameters.

Our experience using our implemented analysis indicates that it can effectively recognize many pure methods. It therefore provides a useful tool that can support a range of important program analysis and software engineering tasks.

The most important future work direction concerns making the analysis better suited to the analysis of incomplete programs and libraries; to make this possible, one should have a specification for the missing parts of the program. The `assignable` clauses from JML are not sufficient: according to [32], we have to specify (in an abstract way) not only the mutated locations, but also the new aliasing created by the mutation. The points-to graphs contain this information, but they are too hairy to be used as a specification language. In Section 7, we used an ad-hoc solution in order to analyze consistency predicates for data structures; however, a more systematic solution is required. Ideally, the specification language should respect abstraction boundaries, i.e., it should not reveal private implementation details.

[10] For this to be true, we also have to treat synchronizations as memory writes.

Acknowledgements

The authors would like to thank several people whose help made this paper possible. Darko Marinov and Viktor Kuncak provided many research suggestions; Darko also gave us the Korat testcases. Suhabe Bugrara wrote a regular expression Java package, and Brian Demsky proof-read parts of the paper.

10. REFERENCES

[1] C. Scott Ananian. MIT FLEX compiler infrastructure for Java. http://www.flex-compiler.lcs.mit.edu.

[2] Adrian Birka. Compiler-enforced immutability for the Java language. Technical Report MIT-LCS-TR-908, MIT Laboratory for Computer Science, June 2003. Revision of Master's thesis.

[3] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis*, pages 123–133, July 2002.

[4] Chandrasekhar Boyapati and Martin C. Rinard. A parameterized type system for race-free Java programs. In *Proc. 16th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001.

[5] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications. Technical Report NII-R0309, Computing Science Institute, Univ. of Nijmegen, March 2003.

[6] Brendon Cahoon and Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proc. 10th International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[7] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[8] Nestor Cataño and Marieke Huismann. ChAsE: a static checker for JML's assignable clause. In *Proc. 4th International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2575 of *LNCS*, January 2003.

[9] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 292–310. ACM Press, 2002.

[10] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, June 2000.

[11] James C. Corbett. Using shape analysis to reduce finite-state models of concurrent java programs. *Software Engineering and Methodology*, 9(1):51–93, 2000.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms (Second Edition)*. MIT Press and McGraw-Hill, 2001.

[13] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th POPL*, 1977.

[14] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proc. 26th ACM POPL*, 1999.

[15] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *Proc. 18th ECOOP*, June 2004.

[16] Brian Demsky and Martin Rinard. Role-Based Exploration of Object-Oriented Programs. In *Proc. 2002 International Conference on Software Engineering*, 2002.

[17] Cormac Flanagan, K. Rustan M. Leino, Mark Lilibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proc. ACM PLDI*, 2002.

[18] Michael Hind and Anthony Pioli. Which pointer analysis should I use? In *Proc. International Symposium on Software Testing and Analysis*, 2000.

[19] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Proc. 18th ACM POPL*, 1991.

[20] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Proc. 29th POPL*, 2002.

[21] Viktor Kuncak and K. Rustan M. Leino. In-place refinement for effect checking. In *Second International Workshop on Automated Verification of Infinite-State Systems (AVIS'03), Warsaw, Poland*, April 2003.

[22] Gary T. Leavens. Advances and issues in JML. Presentation at the Java Verification Workshop, January 2002.

[23] Gray T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML. Technical Report 96-06p, Iowa State University, 2001.

[24] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proc. ACM PLDI*, 2002.

[25] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[26] John M. Lucassen and David K. Gifford. Polymorphic effect systems. pages 47–57. ACM Press, 1988.

[27] Darko Marinov, Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Martin Rinard. An evaluation of exhaustive testing for data structures. Technical Report MIT-LCS-TR-921, MIT CSAIL, Cambridge, MA, September 2003.

[28] Ana Milanova, Atanas Routev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proc. International Symposium on Software Testing and Analysis*, July 2002.

[29] Peter Mueller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in JML. Technical Report TR 02-02, Iowa State University, February 2002.

[30] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[31] Alexandru Salcianu. Pointer analysis and its applications to Java programs. Master's thesis, MIT Laboratory for Computer Science, 2001.

[32] Fausto Spoto and Erik Poll. Static analysis for JML's `assignable` clauses. In *Proc. 10th Workshop on Foundations of Object-Oriented Languages*, 2003.

[33] M. Tofte and L. Birkedal. A region inference algorithm. *Transactions on Programming Languages and Systems*, 20(4), July 1998.

[34] J.A.G.M. van der Berg and B.P.F. Jacobs. The LOOP compiler for Java and UML. Technical Report CSI-R0019, Computing Science Institute, Univ. of Nijmegen, December 2000.

[35] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *Int. Conf. on Automated Software Engineering, 2000*, 2000.

[36] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

APPENDIX

A. EXAMPLE `repOk`

Figure 12 contains an example `repOk` method for a binary search tree that implements a set. Each object of the class `BinarySearchTree` represents a binary search tree. The `size` field contains the number of nodes in the tree. Objects of the inner class `Node` represent nodes of the trees. The elements of the set are stored in the `info` fields. The elements must implement the interface `Comparable`, which provides the method `compareTo` for comparisons.

In this example, `repOk` checks if the input is a valid binary search tree with the correct `size`. First, `repOk` checks if the tree is empty. If not, `repOk` checks that there is no sharing in the underlying graphs of nodes reachable from `root` along the `left` and `right` fields. It then checks that the number of nodes reachable from `root` is `size`. It finally checks that all elements in the left (right) subtree of a node are smaller (larger) than the element in that node.

The method `isTree` uses a breadth-first traversal to check if the underlying object graph is a tree. This traversal keeps a set of `visited` nodes and a `workList` of nodes that still need to be traversed. Notice that the nodes are put in the set using the `Wrapper` class. We need this class because the standard `java.util.Set` compares the elements using their `equals` methods, whereas we want to compare the nodes in the set using comparison by object identity, `==`. The use of the wrapper class is a typical way [25] to achieve this behavior.

Our analysis finds that all three auxiliary methods–`isTree`, `numNodes`, and `isOrdered`–are pure, and also that `repOk` is pure. It is easy to establish that `numNodes` and `isOrdered` are pure, because they do not update any heap location. However, `isTree` writes several heap locations: it modifies the `visited` set and the `workList` list. Additionally, it creates `Wrapper` objects for putting the nodes as elements of the set. Our analysis is precise enough to determine that all mutation occurs on new objects.

```
import java.util.*;

class BinarySearchTree {
    Node root; // root node
    int size;  // number of nodes in the tree

    static class Node {
        Node left;  // left child
        Node right; // right child
        Comparable info; // data
    }

    static final class Wrapper {
        Object o;
        Wrapper(Object o) {
            this.o = o;
        }
        public boolean equals(Object o) {
            if (!(o instanceof Wrapper)) return false;
            return this.o == ((Wrapper)o).o;
        }
        public int hashCode() {
            return System.identityHashCode(o);
        }
    }

    boolean repOk() {
        // checks that empty tree has size zero
        if (root == null) return size == 0;
        // checks that the input is a tree
        if (!isTree()) return false;
        // checks that size is consistent
        if (numNodes(root) != size) return false;
        // checks that data is ordered
        if (!isOrdered(root, null, null)) return false;
        return true;
    }

    boolean isTree() {
        Set visited = new HashSet();
        visited.add(new Wrapper(root));
        LinkedList workList = new LinkedList();
        workList.add(root);
        while (!workList.isEmpty()) {
            Node current = (Node)workList.removeFirst();
            if (current.left != null) {
                // checks that the tree has no sharing
                if (!visited.add(new Wrapper(current.left)))
                    return false;
                workList.add(current.left);
            }
            if (current.right != null) {
                // checks that the tree has no sharing
                if (!visited.add(new Wrapper(current.right)))
                    return false;
                workList.add(current.right);
            }
        }
        return true;
    }

    int numNodes(Node n) {
        if (n == null) return 0;
        return 1 + numNodes(n.left) + numNodes(n.right);
    }

    boolean isOrdered(Node n, Comparable min, Comparable max) {
        if ((min != null && n.info.compareTo(min) <= 0) ||
            (max != null && n.info.compareTo(max) >= 0))
            return false;
        if (n.left != null)
            if (!isOrdered(n.left, min, n.info))
                return false;
        if (n.right != null)
            if (!isOrdered(n.right, n.info, max))
                return false;
        return true;
    }

    /* binary tree methods */
}
```

Figure 12: Code for Appendix A: binary search tree implementation with `repOk` method.