Purity and Side Effect Analysis for Java Programs

Alexandru Sălcianu and Martin Rinard

Massachusetts Institute of Technology
{salcianu,rinard}@csail.mit.edu

Abstract. We present a new purity and side effect analysis for Java programs. A method is pure if it does not mutate any location that exists in the program state right before the invocation of the method. Our analysis is built on top of a combined pointer and escape analysis, and is able to determine that methods are pure even when the methods mutate the heap, provided they mutate only new objects.

Our analysis provides useful information even for impure methods. 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 heap paths 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 applications. Our results show that our analysis effectively recognizes a variety of pure methods, including pure methods that allocate and mutate complex auxiliary data structures.

1 Introduction

Accurate side effect information has several important applications. For example, many program analyses need to understand how the execution of invoked methods may affect the information that the analysis maintains [14,15,18]. In program understanding and documentation, 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 do not inadvertently interfere with other computations. Pure methods can safely be used in program assertions and specifications [3,21]. As a final example, when model checking Java programs [11,12,32,34], the model checker can reduce the search space by ignoring irrelevant interleavings between pure methods, or, more generally, between methods that access disjoint parts of the heap.

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. We use an updated version of the Whaley and Rinard pointer analysis [35].

The updated analysis retains many ideas from the original analysis, but has been completely redesigned in order to allow the analysis correctness proof from [29]. 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 execution of a method from objects that existed before the method was invoked.

Therefore, our analysis can check that a method is pure, in the sense that it does not mutate any object that exists in the prestate, i.e., the program state right before the method invocation; this is also the definition of purity adopted in the Java Modeling Language (JML) [21]. This definition allows a pure method to perform mutation on temporary objects (e.g., iterators) and/or construct complex object structures and return them as a result.

Our analysis applies a more flexible purity criterion than previously implemented purity analyses, e.g., [8,20], 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.

Other researchers have used different pointer analyses to infer side effects [9, 16,25,28]. While our pointer analysis is not the only choice for the basis of a side effect analysis, it has several advantages that recommend it for this task. First, the analysis abstraction distinguishes between prestate objects and newly allocated objects, enabling the support of a more general purity property. Second, the additional information that the analysis computes can identify other useful side effect information (see below). Third, our underlying pointer analysis has already been proved correct [29], implemented, and used for a variety of tasks, including optimizations like stack allocation and synchronization removal [35], and modular reasoning about aspect oriented programs [27].

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 mutate any object 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 heap paths to objects reachable from the parameter.

For compositionality reasons, our analysis examines each method once, under the assumption that objects from the calling context are maximally unaliased. The intraprocedural analysis computes a single parameterized result for each method; the interprocedural analysis instantiates this result to take into account the aliasing at each call site. Similarly, the clients of the analysis should use the read-only/safe parameter information in the context of the aliasing information at each call site[1]. For example, to infer that a call to an impure method does not mutate a specific object, one needs to check that the object is unreachable from parameters that are not read-only. This is the common approach in detecting and specifying read-only annotations for Java [2].

[1] Our underlying pointer analysis already provides such aliasing information.

Here is an example scenario for using the safe parameter information: a typestate checker, e.g., [14], is a tool that tracks the state of objects and usually checks the correct usage of finite state machine-like protocols. The typestate checker can precisely track only the state of the objects for which all aliasing is statically known. Consider the case of a method invocation that uses a tracked object in the place of a safe parameter. As the typestate checker knows all aliasing to the tracked object, it can check whether the tracked object is not aliased with objects transitively reachable from non-safe arguments at the call site. In that case, the typestate checker can rely on the fact that the method call does not change the state of the object, and that it does not introduce new aliasing to the object.

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.

The side effect information that our analysis computes for impure methods – read-only/safe parameters and the aforementioned regular expressions – can provide many of the same benefits as the purity information because it enables other program analyses and developers to bound the potential effects of an impure method.

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 pure methods to mutate newly allocated data structures. Our analysis therefore supports the use of important programming constructs such as iterators in pure methods.
- **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.
- **Beyond Purity:** Our analysis detects *read-only* and *safe* parameters. In addition, our analysis generates regular expressions that conservatively approximate all externally visible locations that an impure method mutates.

Paper Structure: Section 2 introduces our analysis through an example. Section 3 presents our analysis, and Section 4 shows how to interpret the raw analysis results to infer useful side effect information. Section 5 presents experimental results, Section 6 discusses related work, and Section 7 concludes.

2 Example

Figure 1 presents a sample Java program that manipulates singly linked lists. Class `List` implements a list using cells of class `Cell`, and supports two operations: `add(e)` adds object `e` to a list, and `iterator()` returns an iterator over the list elements[2]. We also define a class `Point` for 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`.

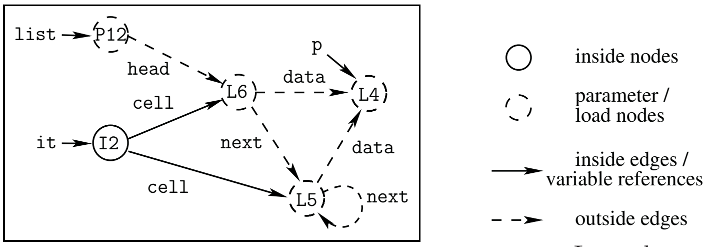
```
class List {
  Cell head = null;
  void add(Object e) {
    head = new Cell(e, head);
  }
  Iterator iterator() {
    return new ListItr(head);
  }
}

class Cell {
  Cell(Object d, Cell n) {
    data = d; next = n;
  }
  Object data;
  Cell   next;
}

interface Iterator {
  boolean hasNext();
  Object next();
}

class ListItr implements Iterator {
  ListItr(Cell head) {
    cell = head;
  }
  Cell cell;
  public boolean hasNext() {
    return cell != null;
  }
  public Object next() {
    Object result = cell.data;
    cell = cell.next;
    return result;
  }
}

class Point {
  Point(float x, float y) {
    this.x = x; this.y = y;
  }
  float x, y;
  void flip() {
    float t = x; x = y; y = t;
  }
}

class Main {
  static float sumX(List list) {
    float s = 0;
    Iterator it = list.iterator();
    while(it.hasNext()) {
      Point p = (Point) it.next();
      s += p.x;
    }
    return s;
  }

  static void flipAll(List list) {
    Iterator it = list.iterator();
    while(it.hasNext()) {
      Point p = (Point) it.next();
      p.flip();
    }
  }

  public static void main(String args[]) {
    List list = new List();
    list.add(new Point(1,2));
    list.add(new Point(2,3));
    sumX(list);
    flipAll(list);
  }
}
```

Fig. 1. Sample Code for Section 2.

Method `sumX` iterates over the list elements by repeatedly invoking the `next()` method of 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`. 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[3] along paths that match the regular expression `list.head.next*.data.(x|y)`.

[2] In real code, the classes `Cell` and `ListItr` would be implemented as inner classes of `List`; we use a flat format for simplicity.

[3] I.e., the state of the program right before the execution of an invoked method.



a. `Cell` constructor Legend

Fig. 2. Points-To Graph for the end of `Main.sumX(List)`.

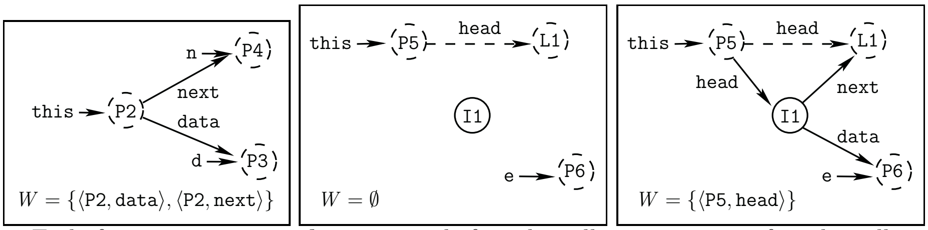
2.1 Analysis Overview

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. During the analysis of method m, the analysis scope contains m and its transitive callees. Figure 2 presents the points-to graph for the end of `Main.sumX(List)`.

The nodes from the points-to graphs model heap objects. The *inside nodes* model the objects created by the analyzed method; there is one inside node for each allocation site; this node models all objects allocated at that site during the current execution of the analyzed method. The *parameter nodes* model the objects passed as arguments; there is one parameter node for each formal parameter of object type (i.e., not an `int`, `boolean`, etc.). The *load nodes* model the objects read from outside the method; there is at most one load node for each load instruction. In Fig. 2, the parameter node P12 models the `List` object pointed by the formal parameter `list`, the inside node I2 models the iterator allocated to iterate over the list, and the load node L6 represents the first list cell (read from P12 by the invoked method `List.iterator`, at line 7). For each analyzed program, the number of nodes is bounded, ensuring the termination of our fixed-point computations.

The edges from the points-to graphs model heap references; each edge is labeled with the field it corresponds to. We write $\langle \mathtt{n_1}, f, \mathtt{n_2} \rangle$ 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); otherwise, the object is *captured.* An outside edge always ends in a load node. In Fig. 2, the outside edge $\langle \mathtt{P12}, \mathtt{head}, \mathtt{L6} \rangle$ models a reference read from the escaped node P12; the inside edges $\langle \mathtt{I2}, \mathtt{cell}, \mathtt{L6} \rangle$ and $\langle \mathtt{I2}, \mathtt{cell}, \mathtt{L5} \rangle$ model the references created by `sumX`[4] from the iterator I2 to the first, respectively to the next list cells. "Loop"

[4] Indirectly, through the iterator-related methods it invokes.


n
P4
this
P2
next
data
d
P3
$W = \{\langle \texttt{P2}, \texttt{data}\rangle, \langle \texttt{P2}, \texttt{next}\rangle\}$
this
P5
head
L1
I1
e
P6
$W = \emptyset$
this
P5
head
L1
head
next
I1
data
e
P6
$W = \{\langle \texttt{P5}, \texttt{head}\rangle\}$

a. End of `Cell` constructor **b.** `List.add` before the call to the `Cell` constructor **c.** `List.add` after the call to the `Cell` constructor

Fig. 3. Analysis results for several simple methods. We use the conventions from Fig. 2.

edges like $\langle \texttt{L5}, \texttt{next}, \texttt{L5}\rangle$ are typical for methods that manipulate recursive data structures.

For each method m, the analysis also computes a set W_m containing the modified abstract fields that are externally visible (the term will become clear later in this section). An abstract field is a field of a specific node, i.e., a pair of the form $\langle n, f\rangle$. There are no externally visible modified fields for `Main.sumX`.

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 uses parameter/load nodes to abstract over unknown nodes, and computes a single parameterized result for m. This result is later instantiated for the aliasing relation at each call site that may invoke m; the interprocedural analysis contains an algorithm that disambiguates parameter/load nodes. Normally, the analysis processes each method once; still, recursive methods may require several analysis passes in order to reach a fixed point.

2.2 Analysis of the Example

Figure 3.a presents the analysis results for the end of the constructor of class `Cell`. The analysis uses the parameter nodes `P2`, `P3`, and `P4` to model the objects that the three parameters – `this`[5], `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`.

Parts b and c of Fig. 3 present the analysis results at different points inside the method `List.add`. The analysis of method `List.add` uses the parameter node P5 to model the `this` object (the list we add to), and the parameter node P6 to model the object to add to the list. The method reads the field `this.head`. 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 $\langle \texttt{P5}, \texttt{head}, \texttt{L1}\rangle$. Next, the method allocates a new `Cell`, that we model with the inside node `I1` (see Fig. 3.b), and calls the `Cell` constructor with the arguments `I1`, `P6`, and `L1`. Based on the points-to graph before the call, and

[5] For each non-static Java method, the parameter `this` points to the receiver object.

the points-to graph for the invoked constructor (Fig. 3.a), 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 $\langle \texttt{P2}, \texttt{data}, \texttt{P3} \rangle$ translates into the inside edge $\langle \texttt{I1}, \texttt{data}, \texttt{P6} \rangle$. Similarly, we have the inside edge $\langle \texttt{I1}, \texttt{next}, \texttt{L1} \rangle$. As `P2` stands for `I1`, the analysis knows that the fields `data` and `next` 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`. This illustrates two features of our analysis: 1) the analysis propagates mutations interprocedurally, using the mappings for the callee nodes and 2) the analysis ignores mutations on inside nodes. Finally, the analysis of `List.add` adds the inside edge $\langle \texttt{P5}, \texttt{head}, \texttt{I1} \rangle$, and records the mutation on the field `head` of `P5`. Figure 3.c presents the result for the end of the method.

The analysis of the rest of the program proceeds in a similar fashion (see [30, Section 2] for the full details). Figure 2 presents the points-to graph for the end of `Main.sumX` (the set of modified abstract fields is empty). The results for `Main.flipAll` are similar to those for `Main.sumX`, with the important difference that the method `flipAll` mutates the fields `x` and `y` or node `L4`.

Analysis Results: For the method `Main.sumX`, the analysis does not detect any mutation on the prestate. Therefore, the method `sumX` is pure, and we can freely use it in assertions and specifications.

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`. Still, the analysis is able to conservatively describe the set of modified prestate locations: these are locations that are reachable from `P12` (the only parameter), along paths of outside edges. 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)`. 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 list non-emptiness of across calls to `flipAll`.

3 Analysis

This section continues the presentation of the analysis that we started in Sec. 2.1. Due to space constraints, we give an informal presentation of the analysis. A formal presentation is available in a companion technical report [30].

In addition to the points-to relation, each points-to graph records the nodes that *escape globally*, i.e., those nodes that are potentially accessed by unknown code: nodes passed as arguments to native methods and nodes pointed from static fields; in addition, any node that is transitively reachable from these nodes along inside/outside edges escapes globally too. The analysis has to be very conservative about these nodes: in particular, they can be mutated by unknown code. We use the additional special node $n_{\texttt{GBL}}$ as a placeholder for other unknown globally escaping nodes: nodes loaded from a static field and nodes returned from an unanalyzable/native method.

3.1 Intraprocedural Analysis

At the start of each method, each object-type parameter (i.e., not an `int`, `boolean`, etc.) points to the corresponding parameter node. Next, our analysis propagates information along the control flow edges, using transfer functions to abstractly interpret statements from the analyzed program. At control flow join points, the analysis merges the incoming points-to graphs: e.g., the resulting points-to graph contains any edge that exists in one or more of the incoming points-to graphs. The analysis iterates over loops until it reaches a fixed point.

As a general rule, we perform strong updates on variables, i.e., assigning something to a variable removes its previous values, and weak updates on node fields, i.e., the analysis of a store statement that creates a new edge from $n_1.f$ leaves the previous edges in place. Because n_1 may represent multiple objects, all of these edges may be required to correctly represent all of the references that may exist in the heap.

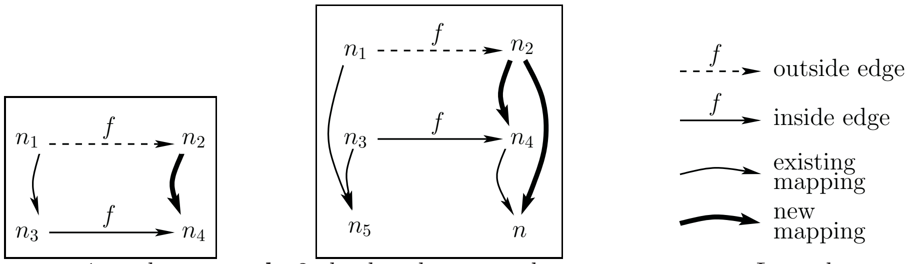
A copy statement "$v_1 = v_2$" makes v_1 point to all nodes that v_2 points to. A new statement "$v = \texttt{new}\ C$" makes v point to the inside node attached to that statement. For a store statement "$v_1.f = v_2$", the analysis introduces an f-labeled inside edge from each node to which v_1 points to each node to which v_2 points.

The case of a load statement "$v_1 = v_2.f$" is more complex. First, after the load, v_1 points to all the nodes that were pointed by an inside edge from $v_2.f$. If one of the nodes that v_2 points to, say n_2, escapes, a parallel thread or an unanalyzed method may create new edges from $n_2.f$, edges that point to objects created outside the analysis domain. The analysis represents these objects using the load node n_L attached to this load statement. The analysis sets v_1 to point to n_L too, and introduces an outside edge from n_2 to n_L. The interprocedural analysis uses this outside edge to find nodes from the calling context that may have been loaded at this load statement.

3.2 Interprocedural Analysis

For each call statement "$v_R = v_0.s(v_1, \ldots, v_j)$", the analysis uses the points-to graph G before the call and the points-to graph G_{callee} from the end of the invoked method *callee* to compute a points-to graph for the program point after the call. If there are multiple possible callees (this may happen because of dynamic dispatch), the analysis considers all of them and merges the resulting set of points-to graphs.

The interprocedural analysis operates in two steps. First, the analysis computes a node mapping that maps the parameter and load nodes from the callee



a. 1st rule **b.** 2nd rule, when n_1 and n_3 are mapped to the common node n_5 Legend

Fig. 4. Rules for the construction of the interprocedural node mapping.

to the nodes they may represent. Next, the analysis uses the node mapping to project G_{callee} and merge it with the points-to graph from before the call.

Due to space constraints, we describe only the construction of the node mapping, and we refer the reader to [30] for an in-depth description of the second step. Intuitively, the second step involves projecting the callee graph through the node mapping, and next merging the result with the graph before the call.

Initially, the analysis maps each parameter node to the nodes to which the corresponding actual argument points. It then repeatedly applies the two rules from Fig. 4 to match outside edges (from read operations) against inside edges (from corresponding write operations) and discover additional node mappings, until a fixed point is reached. The first rule matches outside edges from the callee against inside edges from the caller. This rule handles the case when the callee reads data from the calling context. If node n_1 maps to node n_2, we map each outside edge $\langle n_1, f, n_3 \rangle$ from G_{callee} against each inside edge $\langle n_2, f, n_4 \rangle$ from G, and add a mapping from n_3 to n_4. The second rule maps outside and inside edges from the callee. This rule handles the unknown aliasing introduced at the calling context. If nodes n_1 and n_2 have a common mapping, or one of them is mapped to the other one, they may represent the same location. This potential aliasing was unknown during the analysis of the callee, and we have to handle it now. Therefore, we match each callee outside edge $\langle n_1, f, n_3 \rangle$ from G_{callee} against each callee inside edge $\langle n_2, f, n_4 \rangle$ and map n_3 to n_4, and to all nodes that n_4 maps to.

3.3 Effect Analysis

We piggy-back the side-effect analysis on top of the pointer analysis described in the previous two sections. For each analyzed method m, the analysis maintains a set W_m containing the abstract fields (pairs of nodes and fields) that m mutates. The set W_m is initialized to the empty set. Each time the analysis of m encounters an instruction that writes a heap field, it records into W_m the relevant field and node(s). For example, the analysis of the `Cell` constructor records the mutations of $\langle$`P2`, `data`$\rangle$ and $\langle$`P2`, `next`$\rangle$.

The analysis propagates effects interprocedurally as follows: when the analysis of method m encounters a call instruction, it uses the interprocedural node mapping to project the effects of the callee and include these effects in the set W_m. For example, when the analysis of `List.add` encounters the call to the `Cell` constructor, as `P2` from the constructor maps to `I1`, the constructor's effects $\{\langle \texttt{P2}, \texttt{data}\rangle, \langle \texttt{P2}, \texttt{next}\rangle\}$ are projected into $\{\langle \texttt{I1}, \texttt{data}\rangle, \langle \texttt{I1}, \texttt{next}\rangle\}$. However, these abstract fields are not added to $W_{\texttt{List.add}}$ because of the following additional rule: the analysis does not record mutations on inside nodes – these nodes represent new objects that do not exist in the prestate.

4 Inferring the Side Effect Information

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

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. These nodes represent prestate objects read by the method. The method m is pure iff $\forall n \in A$, 1) n does not escape globally, and 2) no field of n is mutated, i.e., $\forall f. \langle n, f\rangle \notin W_m$.

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.

Read-Only Parameters. A parameter p is *read-only* iff none of the locations transitively reachable from p is mutated. To check this, consider the corresponding parameter node n_p, and let S_1 be the set that contains n_p and all the load nodes reachable from n_p along outside edges. Parameter p is read-only iff 1) there is no abstract field $\langle n, f\rangle \in W_m$ such that $n \in S_1$, and 2) no node from S_1 escapes globally.

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. To detect whether a read-only parameter p is safe, we compute, as before, the set S_1 that contains the corresponding parameter node n_p and all the load nodes reachable from n_p along outside edges. We also compute the set S_2 of nodes reachable from the parameter nodes and/or from the returned nodes, along inside/outside edges; S_2 contains all nodes from G that may be reachable from the caller after the end of m. To check the absence of a new externally visible path to an object reachable from p, it suffices to check the absence of any inside edges from nodes in S_2 to nodes in S_1.

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 of m, we create a transition from s to the corresponding parameter node, and label it with the parameter p. 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, we add a transition from n to t, and label it with the special field REACH. The heap path P.PATH 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 prestate). An interesting example that exhibits this problem is presented in [31]. 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 most of the methods do not read and mutate the same field.

5 Experience

We implemented our analysis in the MIT Flex compiler infrastructure [1], a static compiler for Java bytecode. To increase the analysis precision (e.g., by reducing the number of nodes that are mistakenly reported as globally escaped and therefore mutated) 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.

5.1 Checking Purity of Data Structure Consistency Predicates

We ran our analysis on several benchmarks borrowed from the Korat project [3, 24]. 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. Writing `repOk`-like predicates is considered good software engineering practice; during the development of the data structure, programmers can write assertions that use `repOk` to check the data structure consistency. Programmers do not want assertions to change the semantics of the program, other than aborting the program when it violates an assertion. 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 heaps (priority queues).
`BinomialHeap` and `FibonacciHeap` - Alternative heap implementations.
`LinkedList` - Doubly-linked lists from the the Java Collections Framework.
`TreeMap` - Implementation of the `Map` interface using red-black trees.
`HashSet` - Implementation of the `Set` interface, backed by a hash table.

`LinkedList`, `TreeMap`, and `HashSet` are from the standard Java Library. The only change the Korat developers performed was to add the corresponding `repOk` methods. The `repOk` methods use complex auxiliary data structures: sets, linked lists, wrapper objects, etc. (see [30, Appendix A] for an example). 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`, and `java.lang.Object.toString`.

We call these methods, and all methods that override them, *special* methods. We specified to the analysis that these methods are pure and all their parameters are safe[6]. Therefore, these methods do not mutate their parameters and do not introduce new externally visible aliasing. Hence, the analysis can simply ignore calls to these methods (even dynamically dispatched calls)[7].

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

[6] These assumptions correspond to the common intuition about the special methods. E.g., we do not expect `equals` to change the objects it compares.

[7] Additional processing is required to model the result of the `toString` special methods: as `String`s are supposed to be values, each call to `toString` is treated as an object creation site. The other special methods return primitive values.

Table 1. Java Olden benchmark applications.

Application	Description
`BH`	Barnes-Hut N-body solver
`BiSort`	Bitonic Sort
`Em3d`	Simulation of electromagnetic waves
`Health`	Health-care system simulation
`MST`	Bentley's algorithm for minimum spanning tree in a graph
`Perimeter`	Computes region perimeters in an image represented as a quad-tree
`Power`	Maximizes the economic efficiency of a community of power consumers
`TSP`	Randomized alg. for the traveling salesman problem
`TreeAdd`	Recursive depth-first traversal of a tree to sum the node values
`Voronoi`	Voronoi diagram for random set of points

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 our assumptions about the special methods are true. Our tool tries to verify our assumptions for all the 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.

Discussion. In order to analyze complex data structures that use the real Java library, we had to sacrifice soundness. More specifically, we had to trust that the caching mechanism used by several classes from the Java library has only a performance impact, and is otherwise semantically preserving. 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 useful in practice: this feature allows us to identify (and ignore) benign mutation related to caching.

5.2 Pure Methods in the Java Olden Benchmark Suite

We also ran the purity analysis on the applications from the Java Olden benchmark suite [6, 7]. Table 1 presents a short description of the Java Olden 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. Percentage of Pure Methods in the Java Olden benchmarks.

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 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`.

6 Related Work

Modern research on effect inference stems from the seminal work of Gifford et al on type and effect systems [17, 23] for mostly functional languages. More recent research on effects is usually done in the context of program specification and verification. JML is a behavioral specification language for Java [5] that allows annotations containing invocations of pure methods. JML also allows the user to specify "`assignable`" locations, i.e., locations that a method can mutate [26]. Currently, the purity and `assignable` clauses are either not checked or are checked using very conservative analyses: e.g., a method is pure iff 1) it does not do I/O, 2) it does not write any heap field, and 3) it does not invoke impure methods [20]. ESC/Java is a tool for statically checking JML-like annotations of Java programs. ESC/Java uses a theorem prover to do modular checking of the provided annotations. A major source of unsoundness in ESC/Java is the fact that the tool uses purity and modifies annotations, but does not check them.

Several approaches to solve this problem rely on user-provided annotations; we mention here the work on data groups from [19, 22], and the use of region types [13, 33] and/or ownership types [4, 10] for specifying effects at the granularity of regions/ownership boundaries. In general, annotation based approaches are well suited for modular checking; they also provide abstraction mechanisms to hide representation details.

Analysis-based approaches like ours are appealing because they do 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 several related analyses.

ChAsE [8] is a syntactic tool for modular checking of JML `assignable` clauses. For each method, the tool traverses the method code and collects write effects; for method invocation, ChAsE uses the `assignable` clauses from the callee specification. 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. Some of these problems are discussed in [31]. [31] 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 proposed in [31].

Other researchers [9, 16, 25, 28], have already considered the use of pointer analysis while inferring side effects. Unlike these previous analyses, our analysis uses a separate abstraction (the inside nodes) 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. [28] offers evidence that almost all pure methods can be detected using a very simple pointer analysis. However, the method purity definition used in [28] is more rigid than ours; for example, pure methods from [28] are not allowed to construct and return new objects.

Fugue [14] is a tool that tracks the correct usage of finite state machine-like protocols. Fugue requires annotations that specify 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. Hence, 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).

Javari [2] is an extension to Java that allows the programmer to specify `const` (i.e., read-only) parameters and fields. A type checker checks the programmer annotations. 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. Our solution is to expose the mutation on caches to the programmer, and let the programmer judge whether the mutation is allowed or not. Our tool could complement Javari by inferring read-only parameters for legacy code.

7 Conclusions

Recognizing method purity is important for a variety of program analysis and understanding tasks. We present the first implemented method purity analysis for Java that is capable of recognizing pure methods that mutate newly allocated objects. 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.

Acknowledgements

We thank Darko Marinov and Suhabe Bugrara for their help. This research was supported by the DARPA Cooperative Agreement FA 8750-04-2-0254, DARPA Contract 33615-00-C-1692, the Singapore-MIT Alliance, and the NSF Grants CCR-0341620, CCR-0325283, and CCR-0086154.

References

1. C. S. Ananian. MIT FLEX compiler infrastructure for Java. Available from http://www.flex-compiler.lcs.mit.edu, 1998-2004.
2. A. 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. C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. ISSTA*, 2002.
4. C. Boyapati and M. C. Rinard. A parameterized type system for race-free Java programs. In *Proc. 16th OOPSLA*, 2001.
5. L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. Technical Report NII-R0309, Computing Science Institute, Univ. of Nijmegen, 2003.
6. B. Cahoon and K. 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. M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proc. 5th PPoPP*, 1995.
8. N. Cataño and M. Huismann. ChAsE: a static checker for JML's assignable clause. In *Proc. 4th VMCAI*, 2003.
9. J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proc. 20th POPL*, 1993.
10. D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proc. 17th OOPSLA*, 2002.
11. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd ICSE*, 2000.

12. J. C. Corbett. Using shape analysis to reduce finite-state models of concurrent java programs. *Software Engineering and Methodology*, 9(1), 2000.
13. K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proc. 26th POPL*, 1999.
14. R. DeLine and M. Fähndrich. Typestates for objects. In *Proc. 18th ECOOP*, 2004.
15. C. Flanagan, K. R. M. Leino, M. Lilibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proc. PLDI*, 2002.
16. M. Hind and A. Pioli. Which pointer analysis should I use? In *Proc. ISSTA*, 2000.
17. P. Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *Proc. 18th POPL*, 1991.
18. V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proc. 29th POPL*, 2002.
19. V. Kuncak and K. R. M. Leino. In-place refinement for effect checking. In *2nd Intl. Workshop on Automated Verification of Infinite-State Systems*, 2003.
20. G. T. Leavens. Advances and issues in JML. Presentation at the Java Verification Workshop, 2002.
21. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML. Technical Report 96-06p, Iowa State University, 2001.
22. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Proc. PLDI*, 2002.
23. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proc. 15th POPL*, 1988.
24. D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical Report MIT-LCS-TR-921, MIT CSAIL, Cambridge, MA, 2003.
25. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proc. ISSTA*, 2002.
26. P. Mueller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. Technical Report TR 02-02, Iowa State University, 2002.
27. M. Rinard, A. Sălcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proc. 12th FSE*, 2004.
28. A. Rountev. Precise identification of side-effect-free methods in Java. In *IEEE International Conference on Software Maintenance*, 2004.
29. A. Salcianu. Pointer analysis and its applications to Java programs. Master's thesis, MIT Laboratory for Computer Science, 2001.
30. A. Salcianu and M. Rinard. A combined pointer and purity analysis for Java programs. Technical Report MIT-CSAIL-TR-949, MIT CSAIL, 2004.
31. F. Spoto and E. Poll. Static analysis for JML's `assignable` clauses. In *Proc. 10th FOOL*, 2003.
32. O. Tkachuk and M. B. Dwyer. Adapting side effects analysis for modular program model checking. In *Proc. 11th FSE*, 2003.
33. M. Tofte and L. Birkedal. A region inference algorithm. *Transactions on Programming Languages and Systems*, 20(4), 1998.
34. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. 15th ASE*, 2000.
35. J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. 14th OOPSLA*, 1999.