JPure: A Modular Purity System for Java

David J. Pearce

School of Engineering and Computer Science,
Victoria University of Wellington, New Zealand
Email: djp@ecs.vuw.ac.nz

Abstract. Purity Analysis is the problem of determining whether or not a method may have side-effects. This has applications in automatic parallelisation, extended static checking, and more. We present a novel purity system for Java that employs purity annotations which can be checked modularly. This is done using a flow-sensitive, intraprocedural analysis. The system exploits two properties, called *freshness* and *locality*, to increase the range of methods that can be considered pure. JPure also includes an inference engine for annotating legacy code. We evaluate our system against several packages from the Java Standard Library. Our results indicate it is possible to uncover significant amounts of purity efficiently.

1 Introduction

Methods which don't update program state may be considered *pure* or *side-effect free*. Knowing which methods are pure in a program has a variety of applications, such as: specification languages [19, 1, 3, 10], model checking [31], compiler optimisations [9, 18, 36], atomicity [13], query systems [21, 34] and memoisation of function calls [15].

Several existing techniques are known for determining purity in OO languages (e.g. [26, 30, 22, 23]). The majority employ interprocedural pointer analysis as the underlying algorithm. While this yields precise results, there are a number of drawbacks: firstly, it requires the whole program be known in advance [29]; secondly, it takes a significant amount of time to run, which is prohibitive in normal day-to-day development.

A useful alternative is to use annotations. Here, pure methods are first annotated with `@Pure`; then, a *purity checker* is provided to help enforce the purity protocol in programs. For this approach to be practical, the purity checker must be efficient to fit within normal day-to-day development. A sensible way of ensuring this is to require that the annotations be *modularly checkable*. That is, the purity annotations on one method can be checked in isolation from others. Unfortunately, the majority of previous works on purity analysis, particularly those which depend upon interprocedural pointer analysis, do not generate modularly checkable annotations.

An obvious difficulty with any annotation-based purity system is the vast amount of legacy code that would first need to be annotated. In particular, the Java standard library has not been annotated to identify pure methods. To address this, we present a purity system that is split into two components: a *purity inference* and a *purity checker*. The purity inference operates as a source-to-source translation, taking in existing Java code and adding modularly checkable `@Pure` annotations (and any auxiliary annotations required). The purity checker can then check these annotations are correct efficiently at compile-time. The idea behind this is simple: users take their existing applications, infer the `@Pure` annotations once using the (potentially expensive) purity inference; then, they *maintain* them using the (efficient) purity checker.

An important requirement for an annotation-based purity system is that the annotations themselves must be simple to use. This is because, once the annotations are inferred, we expect programmers to use and understand them. Our system uses only three annotations, `@Pure`, `@Local` and `@Fresh`, but remains sufficiently flexible for many real-world examples.

The contributions of this paper are as follows:

1. We present a novel purity system which employs modularly checkable annotations. This exploits *freshness* and *locality* to increase the number of methods which can be considered pure. The system comprises a *purity checker* and a *purity inference*. The latter operates as a source-source translation for annotating legacy code.

2. We formalise the intraprocedural analysis that underpins both the purity checking and purity inference algorithms.

3. We report on experiments using our system on several packages from the Java Standard Library. Our results indicate that at least 40% of methods in these packages are pure.

2 A Simple Purity System

We start by considering a simple purity system which is surprisingly effective in practice and, crucially, employs modularly checkable annotations. We then highlight several problems which stem from code found in the Java Standard Library. Later, in §3, we refine this simple system to resolve these problems.

2.1 Overview

In the simple purity system, pure methods are indicated by a `@Pure` annotation. The following characterises the meaning of purity within the system:

Definition 1 (Pure Method). *A method is considered pure if it does not assign (directly or indirectly) to any field or array cell that existed before it was called.*

This implies that, for a method to be pure, any method it calls must also be pure. Therefore, for any call-site, we must conservatively approximate the set of methods that may be invoked. To check annotations modularly, we can only rely on the static type information available at a given call-site. For example:

```java
public void f(List<String> x) { x.add("Hello"); }
```

As we do not know what implementations of `List` may be supplied for `x`, we must assume any is possible and, hence, that the invocation may dispatch to any implementation of `List.add()`. Thus, for method `f()` above to be considered pure, every implementation of `List.add()` must itself be pure.

The simple purity system must also follow a covariant typing protocol. This requires that pure methods can only be overridden by methods which are also pure. The following illustrates:

```
class Parent {
 @Pure void f() {}
}

class Child extends Parent {
 int x;
 void f() { x=1; }
}
```

If we considered `Parent.f()` in isolation, one would conclude it is pure. However, `Child.f()` is clearly not pure, since it assigns field `x`. Thus, following the covariant typing protocol, the purity checker must reject this code.

2.2 Modular Checking

We now give an informal argument as to why the annotations produced by the simple purity system are modularly checkable. Essentially, there are three cases to consider:

(1) **Direct Field Assignment.** A method annotated `@Pure` cannot assign to fields. This can be easily checked by inspecting its implementation.

(2) **Indirect Field Assignment.** A method annotated `@Pure` may only call methods which are themselves pure. This is checked by ensuring, for each call-site, the method invoked is annotated `@Pure`. Since this is determined using static type information, it may not be the actual method invoked (due to dynamic dispatch). However, it is safe as the covariant typing protocol ensures all overriding methods must be pure.

(3) **Method Overriding.** A method annotated `@Pure` can only be overridden by methods which are also annotated `@Pure`. This ensures the covariant typing protocol is followed and is easily checked by comparing the annotations on a method with those it overrides.

The argument here is that: one can check a method annotated `@Pure` is indeed pure simply by inspecting its implementation, and those signatures of methods it calls or overrides. This follows the general approach to type checking, as adopted by e.g. Java's type checker.

2.3 Problem 1 — Iterator

We now consider several problems that arose when using the simple purity system on real code. The first is that of `java.util.Iterator`. The following illustrates:

```
public Test {
 private List<String> items;
 boolean has(String x) {
  for(String i : items) {
   if(x == i) { return true; }
  }
 return false;
}}
```

At a glance, method `Test.has()` appears pure. But, this is not the case because the **`for`**-loop uses an iterator. Roughly speaking, the loop is equivalent to the following:

```
boolean has(String x) {
 Iterator iter = items.iterator();
 while(iter.hasNext()) {
  String i = iter.next();
  if(x == i) { return true; }
 }
 return false;
}
```

Here, `iter.next()` is called to get the next item on the list. However, this method updates its `Iterator` object and, hence, cannot be considered pure. In section §3, we resolve this by extending the system to make explicit the fact that `items.iterator()` only ever returns *fresh* — i.e. newly allocated — objects.

2.4 Problem 2 — Append

The second kind of problem one encounters with the simple system is illustrated by the following, adapted from `java.lang.AbstractStringBuilder`:

```
class AbstractStringBuilder {
 char[] data;
 int count;   // number of items used
 AbstractStringBuilder append(String s){
  ...
  ensureCapacity(count + s.length());
  s.getChars(0, s.length(), data, count);
  ...
  return this;
}}
```

Here, `getChars()` works by copying the contents of `s` into `data` at the correct position. Because of this, `getChars()` and, hence, `append()` cannot be considered pure under Definition 1. *So, why is this a problem?* Well, consider the following:

```
String f(String x) { return x + "Hello"; }
```

Again, at a glance, this method appears pure. However, the bytecode generated for this method indirectly calls `AbstractStringBuilder.append()` and, hence, it cannot be considered pure.

Clearly, we want methods such as `f()` above to be considered pure. In section §3, we achieve this by extending the system to determine that: firstly, the `StringBuilder` object created for the string append is fresh; secondly, that the array referred to by `data` is in the *locality* of that `StringBuilder` object. When an object (the child) is in the locality of another (the parent), we have a guarantee that the child is fresh if the parent is fresh. Thus, assignments to the array referenced by `data` are permitted as it is known to be fresh (since the `StringBuilder` object is fresh).

3 Our Improved Purity System

In this section, we detail our purity system which improves upon the simple approach outlined in §2. Our system is implemented in a tool called JPure, and we report on experiments using it in §5.

3.1 Freshness and Locality

Let us recall the first problem encountered with the simple purity system, as discussed in §2.3:

```
@Pure boolean has(String x) {
 Iterator iter = items.iterator();
 while(iter.hasNext()) {
  String i = iter.next();
  if(x == i) { return true; }
 }
 return false;
}
```

To show that this method is pure under Definition 1 requires two guarantees:

1. That `items.iterator()` always returns a *fresh* (i.e. newly allocated) object.

2. That `iter.next()` can only modify the *localilty* (i.e. local state) of the `Iterator` object.

Intuitively, the idea is that, when an object is fresh, so is its locality. Then, by Definition 1, state in its locality can be modified as it did not exist prior to the method (i.e. `has()`) being called. In §3.2 we will give a more precise definition of locality.

To indicate a method returns a fresh object, or that it only ever modifies an object's locality, we employ two additional annotations: `@Fresh` and `@Local`. Thus, for the `Collection` and `Iterator` interfaces, we would add the following annotations:

```
interface Collection {
  @Fresh Object iterator();
  ...
}
interface Iterator {
  @Pure boolean hasNext();
  @Local Object next();
  ...
}
```

Here, `@Fresh` implies all implementations of `iterator()` must return a fresh object and, furthermore, must be pure (as, for brevity, we say `@Fresh` implies `@Pure`). Likewise, since `hasNext()` is annotated `@Pure`, its implementations must be pure. Finally, the `@Local` annotation on `next()` implies its implementations may only modify the `Iterator`'s locality.

3.2 Understanding Locality

To make our notions of freshness and locality more precise, we must consider which parts of an object they apply to. For example, an `Iterator` instance returned from `iterator()` may be freshly allocated; but, it is almost certainly not the case that all objects reachable from it are (e.g. the items being iterated over). Thus, we need some way to determine how much of an object's reachable state is included in its locality.

Definition 2 (Locality). *The* locality *of an object includes every field defined by its class and, for those annotated `@Local`, the locality of the referenced object.*

Fields of primitive type are always in the locality of their containing object. For fields of reference type, the field itself is always in the locality, but the referenced object may or may not be (depending on whether the field is annotated `@Local` or not). Our definition of locality is, in some ways, similar to the notion of *ownership* (see e.g. [4, 8]); however, we are able to exploit some counter-intuitive properties of pure methods to get a simpler, more flexible system.

Figure 1 illustrates the main ideas. For an instance of `TypedList`, its locality includes the fields `length`, `data` and `elementType`. The locality of the object referenced by `data` is also included, whilst that referred to by `elementType` is not. The presence of an `@Local` annotation on `copy()` indicates it is a *local method*:

Definition 3 (Local Method). *A local method may modify the locality of any parameter annotated `@Local` but, in all other respects, must remain pure. The method receiver (i.e. **`this`**) is treated as a special parameter, with `@Local` placed on the method itself.*

By Definition 3, `copy()` may modify the locality of **`this`** and, by Definition 1, any state created during its execution.

The following rules clarify in more detail what a local method conforming to Definition 3 is permitted to do:

Rule 1 *A local method may assign fresh objects to any field in the locality of a parameter annotated `@Local`.*

```
 1 class TypedList {
 2  private int length;
 3  private @Local Object[] data;
 4  private Type elementType;
 5
 6  @Local public TypedList(Type type, int maxSize) {
 7    length = 0;
 8    data = new Object[maxSize];
 9    elementType = type;
10  }
11
12  @Local public void copy(TypedList dst) {
13    length = dst.length;
14    type = dst.type;
15    data = new Object[dst.length];
16    for(int i=0;i!=length;++i) { data[i] = dst.data[i]; }
17 } }
```

Fig. 1. An example illustrating the main aspects of locality.

Rule 2 *A local method may assign* any *reference to a field in the locality of a parameter annotated `@Local`,* provided that field is not itself annotated `@Local`.

To better understand these rules, consider them in the context of Figure 1. The assignment to `data` on Line 15 is permitted under Rule (1) above. Similarly, the assignments to `length` and `type` on Lines 13 and 14 are permitted under Rule (2) since they are both in the locality of **`this`**, but are not annotated `@Local`. Finally, the assignment to elements of `data` on Line 16 is permitted under Rule (2). This is because elements of an array object are treated as though they were fields. Since these "fields" cannot be annotated with `@Local`, Rule (2) must apply.

3.3 Locality Invariants

The rules for checking local methods given in the previous section may seem strange, but they are needed to preserve the *locality invariants*:

Locality Invariant 1 (Construction) *When a new object is constructed, its locality is always fresh.*

The purpose of Locality Invariant 1 is to ensure we can safely modify the locality of fresh objects within pure methods.

Locality Invariant 2 (Preservation) *When the locality of a fresh object is modified, its locality must remain fresh.*

The purpose of Locality Invariant 2 is to ensure that the locality of a fresh object remains fresh, even after modifications to it. Without this guarantee, we become limited in how we can subsequently modify this locality. For example:

```
 1 class Link {
 2  private @Local Link next;
 3
 4  public @Local void set(Link link) {
 5   this.next = link; // violates invariant 2
 6  }
 7  public @Fresh Link create() {
 8   Link c = new Link();
 9   c.set(this.next);
10   c.next.set(null); // problem
11   return c;
12 }}
```

Here, method `set()` violates Locality Invariant 2 by assigning an arbitrary reference to field `next`. This is a problem because the locality of a fresh `Link` may no longer be fresh after this assignment (i.e. if the `link` assigned is not fresh). This problem manifests itself in `create()` as, after `set()` is called on Line 9, the locality of the object referenced by `c` is no longer entirely fresh (i.e. `c.next` is not fresh). As a result, the call to `set()` on Line 10 causes a side-effect — meaning `create()` should not be considered pure (recall `@Fresh` implies `@Pure`).

3.4 The Law of Locality

Locality can be regarded as a simplified form of ownership suited to purity analysis. Unlike ownership we can be more flexible regarding object aliasing. In particular, the seemingly counter-intuitive *law of locality* is useful:

Definition 4 (Law of Locality). *When checking `@Local` annotations, one can safely assume parameters are not aliased.*

This law seems strange, but it relates to the overall goal of purity analysis. To understand it better, consider the following:

```
void f(T a, @Local T b) { b.field = 0; }
```

Here, it is clear that `b` must be annotated `@Local`, since its locality is modified. However, the question is: should `a` be annotated `@Local` as well? Given that `a` and `b` could be aliased on entry, it seems as though we should assume they are. However, under the law of locality, we can assume they are not.

So, *why does the law of locality work?* Well, the key lies in the way `@Local` annotations are used to show methods are pure. Consider the following:

```
@Pure void g() { T x = new T(); f(x,x); }
```

This method is considered pure precisely because the object passed in for parameter `b` is fresh. Thus, if `a` and `b` are aliased on entry to `f()` we have one of two things: either, the caller is impure (in which case it doesn't matter); or, the object passed in for `b` is fresh. In the latter, it immediately follows that `a` is fresh — hence, neither the Locality Invariants nor Definition 1 are violated by the assignment in `f()`.

4 Implementation

We have implemented the purity system outlined in §3 as part of a tool called JPure, which supports *purity checking* and *purity inference*. The former can be done efficiently in a modular fashion. The latter performs a source-to-source translation of Java source code, whilst annotating it with `@Pure`, `@Local` and `@Fresh` annotations where appropriate. Both tools employ an intraprocedural analysis to determine the freshness and locality of variables within a method. The purity inference propagates that information interprocedurally using *Static Class Hierarchy Analysis* [11] to ensure annotations remain modularly checkable. In this section, we formalise the dataflow analysis which underpins both modes of operation.

4.1 Intermediate Language

Before presenting the details of our analysis, we first introduce an Intermediate Language (IL) to base this on. The IL is small and compact, and we deliberately omit many features of the Java language. Despite this, it provides a useful vehicle for presenting the key aspects of our analysis.

The syntax of our intermediate language is given in Figure 2. The IL uses unstructured control-flow, and employs only very simple statements. We also assume our variables are class references, and ignore other types altogether (since they are of no concern here). Likewise, we provide only very limited forms of expression in **`if`** conditions. A simple IL program is given below:

```
void meth(Object x) {
  Object y;
  y = new MyClass();
  if(x == null) goto label1;
  y = x;
 label1:
  y.f();
}
```

Our intraprocedural analysis will determine that method `f()` may be called on the object referred to by `x`. If this method is impure, the entire method must be impure; or, if this method has a `@Local` receiver, then `x` will be annotated `@Local`; otherwise, if `f()` is `@Pure` then the whole method may be annotated `@Pure` (i.e. provided the `MyClass` constructor is).

4.2 Overview

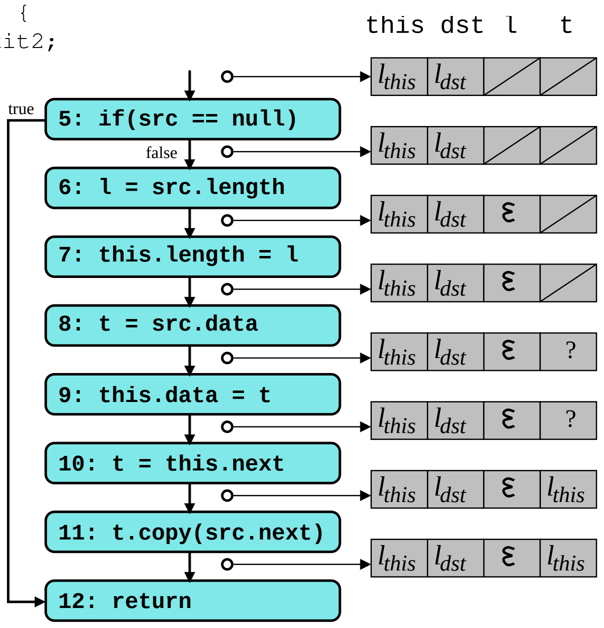
The intraprocedural analysis employs an *abstract environment*, Γ, which conservatively models the freshness and locality of visible objects. This maps variables to a set of *abstract references* which range over $\{?, \epsilon, \ell_{\mathtt{x}}, \ell_{\mathtt{y}}, \ldots\}$. Here, $?$ indicates an unknown object, ϵ indicates a fresh object or primitive value and, finally, $\ell_x, \ell_y, \ldots$ are *named objects*. One named object is provided for each parameter, and represents the object it

Intermediate Language Syntax:

$$
\begin{array}{rl}
\mathtt{M} ::= & \mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}\ \mathtt{x}})\ \{\ \overline{\mathtt{Object}\ \mathtt{v}}\ \ \overline{[\mathtt{L:}]\ \mathtt{S}}\ \} \\
\mathtt{S} ::= & \mathtt{v} = \mathtt{w} \mid \mathtt{v} = \mathtt{c} \mid \mathtt{v}.[\mathtt{T}]\mathtt{f} = \mathtt{w} \mid \mathtt{v} = \mathtt{w}.[\mathtt{T}]\mathtt{f} \mid \mathtt{v} = \mathtt{u}.\mathtt{m}[\mathtt{T_f}](\overline{\mathtt{w}}) \mid \mathtt{v} = \mathtt{new}\ [\mathtt{T_f}](\overline{\mathtt{w}}) \\
 & \mid \mathtt{return}\ \mathtt{v} \mid \mathtt{if}(\mathtt{v}{=}{=}\mathtt{w})\ \mathtt{goto}\ \mathtt{L} \mid \mathtt{goto}\ \mathtt{L} \\
\mathtt{c} ::= & \mathtt{null}, \ldots, -1, 0, 1, \ldots \\
\mathtt{T} ::= & \mathtt{C} \mid \mathtt{int} \\
\mathtt{T_f} ::= & \overline{\mathtt{T}} \rightarrow \mathtt{T}
\end{array}
$$

Fig. 2. Syntax for a simple intermediate language. Here, `C` represents a valid class name, whilst `c` represents a constant. We only consider class reference and **`int`** types, since these are sufficient to illustrate the main ideas. Finally, field accesses and method calls are annotated with the static type of the field/method in question.

```
1  class LL {  int length;  Object data;  @Local LL next;
2   void set(LL next) { this.next = next; }
3
4   @Local void copy(LL src) {
5    if(src == null) goto exit2;
6     int l = src.length;
7     this.length = l;
8     Object t = src.data;
9     this.data = t;
10    t = this.next;
11    t.copy(src.next);
12   exit2:
13  }
14  @Fresh Object clone() {
15   int tmp = this.length;
16   LL t = new LL(tmp);
17   t.copy(this);
18   return t;
19  }
20  @Local LL(int n) {
21   this.length = n;
22   if(n > 0) goto label1:
23     this.next = null;
24     goto exit1;
25   label1:
26     this.next = new LL(n-1);
27   exit1:
28 }}
```



this dst l t
5: if(src == null)
true
false
6: l = src.length
7: this.length = l
8: t = src.data
9: this.data = t
10: t = this.next
11: t.copy(src.next)
12: return
l_{this} l_{dst}
l_{this} l_{dst}
l_{this} l_{dst} ε
l_{this} l_{dst} ε
l_{this} l_{dst} ε ?
l_{this} l_{dst} ε ?
l_{this} l_{dst} ε l_{this}
l_{this} l_{dst} ε l_{this}

Fig. 3. A simple linked list example which contains (among other things) a local method `copy`, which updates the locality of the receiver **`this`**. Alongside, the result of our intraprocedural analysis are shown for this method.

referenced on entry. The analysis tracks how these flow through the method, in a manner similar to an intraprocedural pointer analysis.

Figure 3 illustrates the analysis operating on a simple local method, `copy()`. Recall from Definition 3 that, since the method is annotated `@Local`, it is entitled to modify the locality of the receiver (i.e. **`this`**), but in all other respects must remain pure.

The intraprocedural analysis assumes the following abstract environment holds on entry to `copy()` (i.e. immediately after Line 4):

$$\Gamma{\downarrow}(4) = \{\texttt{this} \mapsto \{\ell_{\texttt{this}}\}, \texttt{src} \mapsto \{\ell_{\texttt{src}}\}\}$$

At this stage, `l` and `t` are undefined and, hence, not present in the abstract environment. Thus, we see that **`this`** references named object $\ell_{\texttt{this}}$, and `src` references $\ell_{\texttt{src}}$. The abstract environment immediately following the assignment to `l` on Line 6 is:

$$\Gamma{\downarrow}(6) = \{\texttt{this} \mapsto \{\ell_{\texttt{this}}\}, \texttt{src} \mapsto \{\ell_{\texttt{src}}\}, \texttt{l} \mapsto \{\epsilon\}\}$$

Here, `l` maps to the special value ϵ which represents both primitive values (as in this case) and fresh objects. The abstract environment immediately following the assignment to `t` on Line 8 is:

$$\Gamma{\downarrow}(8) = \{\texttt{this} \mapsto \{\ell_{\texttt{this}}\}, \texttt{src} \mapsto \{\ell_{\texttt{src}}\}, \texttt{l} \mapsto \{\epsilon\}, \texttt{t} \mapsto \{?\}\}$$

In the above, ? represents an *unknown object reference*, and indicates that there is no information available at this point about the object `t` refers to. Nevertheless, this unknown reference can be safely assigned to **`this`**`.data` on Line 9 under Rule (2) from §3.2.

The second assignment to `t` on Line 10 is treated differently from the first, because field `next` is annotated `@Local`. The abstract environment immediately following this assignment is:

$$\Gamma{\downarrow}(10) = \{\texttt{this} \mapsto \{\ell_{\texttt{this}}\}, \texttt{src} \mapsto \{\ell_{\texttt{src}}\}, \texttt{l} \mapsto \{\epsilon\}, \texttt{t} \mapsto \{\ell_{\texttt{this}}\}\}$$

This captures the fact that `t` still refers to an object in the locality of $\ell_{\texttt{this}}$. This is needed to determine that the subsequent invocation, `t.copy(src.next)`, is safe. That is, since `copy()` is permitted to modify the locality of its receiver and, at this point, `t` refers to an object within this locality, the invocation `t.copy(src.next)` is permitted.

4.3 Abstract Semantics

The effect of a statement on an abstract environment is determined by its *abstract semantics*, which we describe using transition rules. These summarise the abstract environment immediately after the instruction in terms of that immediately before it. The abstract semantics for the intraprocedural analysis are given in Figure 4. Here, $\Gamma[\texttt{v} \mapsto \phi]$ returns an abstract environment identical to Γ, except that `v` now maps to ϕ. Similarly, $\Gamma[\texttt{v}]$ returns the abstract reference for `v` in Γ. Several helper functions and constants are used in the semantics:

$$\frac{\mathtt{c} \in \{\mathtt{null}, \ldots, -1, 0, 1, \ldots\}}{\mathtt{tf}(\mathtt{v}=\mathtt{c}, \Gamma) \longrightarrow \Gamma[\mathtt{v} \mapsto \{\epsilon\}]}\ \text{[S-C]} \qquad \frac{}{\mathtt{tf}(\mathtt{v}=\mathtt{w}, \Gamma) \longrightarrow \Gamma[\mathtt{v} \mapsto \Gamma[\mathtt{w}]]}\ \text{[S-V]}$$

$$\frac{\begin{array}{c} \neg\mathtt{isImpure}(\mathtt{m}, \mathtt{T_f}) \\ \mathtt{isFresh}(\mathtt{m}, \mathtt{T_f}) \Longrightarrow \phi = \{\epsilon\} \quad \neg\mathtt{isFresh}(\mathtt{m}, \mathtt{T_f}) \Longrightarrow \phi = \{?\} \\ \mathtt{isLocal}(\mathtt{u}, \mathtt{m}, \mathtt{T_f}) \Longrightarrow \mathtt{isLocalOrFresh}(\Gamma[\mathtt{u}]) \\ \mathtt{isLocal}(\mathtt{w_1}, \mathtt{m}, \mathtt{T_f}) \Longrightarrow \mathtt{isLocalOrFresh}(\Gamma[\mathtt{w_1}]) \\ \ldots \\ \mathtt{isLocal}(\mathtt{w_n}, \mathtt{m}, \mathtt{T_f}) \Longrightarrow \mathtt{isLocalOrFresh}(\Gamma[\mathtt{w_n}]) \end{array}}{\mathtt{tf}(\mathtt{v}=\mathtt{u.m}[\mathtt{T_f}](\overline{\mathtt{w}}), \Gamma) \longrightarrow \Gamma[\mathtt{v} \mapsto \phi]}\ \text{[S-M]}$$

$$\frac{\mathtt{T} = \mathtt{int}}{\mathtt{tf}(\mathtt{v}=\mathtt{w}.[\mathtt{T}]\mathtt{f}, \Gamma) \longrightarrow \Gamma[\mathtt{v} \mapsto \{\epsilon\}]}\ \text{[S-F1]} \qquad \frac{\mathtt{T} \neq \mathtt{int} \quad \neg\mathtt{isLocal}(\mathtt{f}, \mathtt{T})}{\mathtt{tf}(\mathtt{v}=\mathtt{w}.[\mathtt{T}]\mathtt{f}, \Gamma) \longrightarrow \Gamma[\mathtt{v} \mapsto \{?\}]}\ \text{[S-F2]}$$

$$\frac{\begin{array}{c} \mathtt{isLocalOrFresh}(\Gamma[\mathtt{w}]) \\ \mathtt{T} \neq \mathtt{int} \quad \mathtt{isLocal}(\mathtt{f}, \mathtt{T}) \end{array}}{\mathtt{tf}(\mathtt{v}=\mathtt{w}.[\mathtt{T}]\mathtt{f}, \Gamma) \longrightarrow \Gamma[\mathtt{v} \mapsto \Gamma[\mathtt{w}]]}\ \text{[S-F3]} \qquad \frac{\begin{array}{c} \mathtt{isLocalOrFresh}(\Gamma[\mathtt{v}]) \\ \mathtt{isLocal}(\mathtt{f}, \mathtt{T}) \Longrightarrow \Gamma[\mathtt{w}] = \{\epsilon\} \end{array}}{\mathtt{tf}(\mathtt{v}.[\mathtt{T}]\mathtt{f}=\mathtt{w}, \Gamma) \longrightarrow \Gamma}\ \text{[S-W]}$$

$$\frac{\begin{array}{c} \neg\mathtt{isImpure}(\$, \mathtt{T_f}) \\ \mathtt{isLocal}(\mathtt{w_1}, \$, \mathtt{T_f}) \Longrightarrow \mathtt{isLocalOrFresh}(\Gamma[\mathtt{w_1}]) \\ \ldots \\ \mathtt{isLocal}(\mathtt{w_n}, \$, \mathtt{T_f}) \Longrightarrow \mathtt{isLocalOrFresh}(\Gamma[\mathtt{w_n}]) \end{array}}{\mathtt{tf}(\mathtt{v}=\mathtt{new}[\mathtt{T_f}](\overline{\mathtt{w}}), \Gamma) \longrightarrow \Gamma[\mathtt{v} \mapsto \{\epsilon\}]}\ \text{[S-N]}$$

$$\frac{}{\mathtt{tf}(\mathtt{if}(\mathtt{v}{==}\mathtt{w})\ \mathtt{goto}\ \mathtt{L}, \Gamma) \longrightarrow \Gamma}\ \text{[S-I]} \qquad \frac{}{\mathtt{tf}(\mathtt{goto}\ \mathtt{L}, \Gamma) \longrightarrow \Gamma}\ \text{[S-G]}$$

$$\frac{\mathtt{isFresh}(\mathtt{this_{meth}}) \Longrightarrow \Gamma[\mathtt{v}] = \epsilon}{\mathtt{tf}(\mathtt{return}\ \mathtt{v}, \Gamma) \longrightarrow \Gamma}\ \text{[S-R]}$$

Fig. 4. Abstract semantics for checking the correctness of methods annotated `@Pure`, `@Fresh` or `@Local`. The rules assume the method being analysed has at least one of these annotations.

– `isFresh(m,`$\mathtt{T_f}$`)` — true iff the given method (determined by its name `m` and static type $\mathtt{T_f}$) is annotated `@Fresh`.

– `isImpure(m,`$\mathtt{T_f}$`)` — true iff the given method (determined by its name `m` and static type $\mathtt{T_f}$) is impure. That is, it is neither annotated with `@Pure`, nor any of its parameters are marked with `@Local`. Note, \$ indicates a constructor.

– `isLocal(f,T)` — true iff the given field (determined by its name `f` and static type `T`) is annotated `@Local`.

– `isLocal(i,m,`$\mathtt{T_f}$`)` — true iff the parameter at position `i` in the given method (determined by its name `m` and static type $\mathtt{T_f}$) is annotated `@Local`.

– `isLocalOrFresh(ls)` — true iff the parameters identified in $\mathtt{ls} \subseteq \{\ell_1, \ldots, \ell_n, \epsilon\}$ are annotated `@Local` in the method being analysed. Note, $? \in \mathtt{ls}$ is not permitted.

– $\mathtt{this_{meth}}$ — expands to $(\mathtt{m}, \mathtt{T_f})$ where `m` is the name of the method being analysed, and $\mathtt{T_f}$ is its type.

As another example, let us consider how the intraprocedural analysis applies to the method `clone()` from Figure 3. This is annotated `@Fresh` which implies: firstly, it must return an object that did not exist prior to its invocation; secondly, it may not modify any state that existed prior to its invocation (since `@Fresh` implies `@Pure`).

The intraprocedural analysis assumes the following abstract environment holds on entry to `clone()`:

$$\Gamma{\downarrow}(14) = \{\mathtt{this} \mapsto \{\ell_{\mathtt{this}}\}\}$$

Then, by application of rule S-F1, it computes the following to hold immediately after Line 15:

$$\Gamma{\downarrow}(15) = \{\mathtt{this} \mapsto \{\ell_{\mathtt{this}}\}, \mathtt{tmp} \mapsto \{\epsilon\}\}$$

At this point, a call to the `LL(`**`int`**`)` constructor is encountered. Constructors are treated like other methods, and may be annotated with `@Pure`, `@Local` or not at all (i.e. if they are impure). The `LL(`**`int`**`)` constructor is annotated with `@Local`, and is treated in the same way as a local method. Hence, it is permitted to modify the locality of **`this`** (but must remain pure in all other respects). Therefore, rule S-N applies here, and so the following is determined to hold immediately after Line 16:

$$\Gamma{\downarrow}(16) = \{\mathtt{this} \mapsto \{\ell_{\mathtt{this}}\}, \mathtt{tmp} \mapsto \{\epsilon\}, \mathtt{t} \mapsto \{\epsilon\}\}$$

Recall that `clone()` must be pure (since `@Fresh` implies `@Pure`), and that the `LL(`**`int`**`)` constructor is `@Local` (hence, it may modify state). Rule S-N safely embodies these requirements as, in a constructor, **`this`** is (by definition) fresh.

Finally, the analysis applies rule S-M to determine the abstract environment immediately after Line 17. This is identical to $\Gamma{\downarrow}(16)$ as `copy(LL)` has no return value. The analysis then applies rule S-R to confirm the return value is indeed fresh.

4.4 Dataflow Equations

We formalise the intraprocedural analysis in the usual way by providing *dataflow equations* over the control-flow graph:

$$\Gamma\!\uparrow\!(\mathtt{n}) = \bigsqcup_{\mathtt{m}\to\mathtt{n}} \Gamma\!\downarrow\!(\mathtt{m})$$

$$\Gamma\!\downarrow\!(\mathtt{n}) = \mathtt{tf}\big(\mathtt{S}(\mathtt{n}), \Gamma\!\uparrow\!(\mathtt{n})\big)$$

Here, $\mathtt{m}$ and $\mathtt{n}$ represent nodes in the control-flow graph, and $\mathtt{m}\to\mathtt{n}$ the directed edge between them. Similarly, $\mathtt{tf}$ denotes the *transfer function*, whose operation is determined by the semantics of Figure 4, whilst $\mathtt{S}(\mathtt{n})$ gives the statement at node $\mathtt{n}$. The abstract environment immediately before node $\mathtt{n}$ is given by $\Gamma\!\uparrow\!(\mathtt{n})$, whilst that immediately after is given by $\Gamma\!\downarrow\!(\mathtt{n})$. Finally, we define the meet of two abstract environments as follows:

$$\Gamma_1 \sqcup \Gamma_2 = \{\mathtt{x} \mapsto \phi_1 \cup \phi_2 \mid \mathtt{x} \in \mathbf{dom}(\Gamma_1) \cup \mathbf{dom}(\Gamma_2) \wedge\ \phi_1 = \Gamma_1[\mathtt{x}] \wedge \phi_2 = \Gamma_2[\mathtt{x}]\}$$

To be complete, we must detail the initial store used in the dataflow analysis. This is defined as follows:

$$\Gamma\!\uparrow\!(0) = \{\mathtt{x} \mapsto \{\ell_{\mathtt{x}}\} \mid \mathtt{x} \in \mathtt{Params}\} \cup \{\mathtt{this} \mapsto \epsilon\}$$

Here, Params is the set of all parameters accepted by the method being analysed. Furthermore, we assume that node 0 is the entry point of the control-flow graph. Observe that the analysis assumes parameters are unaliased on entry. Whilst this may seem unsound, it is safe under the law of locality (see §3.4).

4.5 Purity Checking

As discussed previously, our purity system breaks into two components: a *purity checker* and a *purity inference*. The former checks the annotations in a given program are used correctly; the latter infers @Pure, @Local, and @Fresh annotations on legacy code.

In this section, we consider the *purity checker* in more detail. This checks each method in isolation from others using the rules from §3.2 and the intraprocedural analysis discussed earlier. For any method m, the purity checker ensures a covariant typing protocol is followed for @Fresh and @Pure annotations, and a contra-variant protocol is followed for @Local annotations. This is done by examining the annotations on those methods overridden by m (which is the same approach used in the Java compiler for checking generic types).

For methods annotated with @Pure, @Fresh or @Local, the intraprocedural analysis is used to check the annotations are properly adhered to. In this case, the rules of Figure 4 are treated in a similar manner to normal typing rules. If the analysis can construct a valid abstract environment before and after each statement, then the method is considered safe (with respect to its purity annotations). Or, if it is unable to do this, an error is reported.

4.6 Purity Inference

The purity inference is also based on the intraprocedural analysis; however, the rules from Figure 4 are treated differently and information may be propagated interprocedurally. For example, some rules (e.g. S-F3) require that fields be annotated `@Local`, whilst others (e.g. S-W) can require that parameters be annotated `@Local`. Other rules (e.g. S-R) are indifferent on whether an annotation actually has to be present or not.

In order to uncover as much purity as possible, the inference adopts a greedy approach. Initially, it assumes all methods are annotated `@Fresh` or `@Pure` (depending on their return type) and all fields are annotated `@Local`. Then, it processes each method in turn and, upon encountering something which invalidates these assumptions, downgrades them as necessary. For example, consider the following method:

```
1 void f(Counter t) { t.count = 1 }
```

The inference initially assumes `f()` is `@Pure`. Upon encountering the assignment on Line 2 this assumption becomes untenable under rule S-W (which requires `t` be fresh or annotated `@Local`). Since `t` is not fresh, it removes the `@Pure` annotation on `f()`, and replaces it with a `@Local` annotation on `t`.

When the annotations on a method are downgraded, this can have a knock-on effect for other methods. In particular, if one method `g()` calls `f()` and, subsequently, it transpires that `f()` is not `@Pure`, then this implies `g()` is no longer `@Pure`. To address this, the purity inference propagates information interprocedurally using static class hierarchy analysis. The following example illustrates:

```
class Parent {
  void f(Test x, Test y) { g(x) }
  void g(Test z) { z.field = 1; }
}
class Child extends Parent {
  int field;
  void f(Test u, Test v) { }
}
```

Let us assume the inference initially visits `Parent.f()`, then `Parent.g()`. The inference will conclude that `Parent.f()` is `@Pure`, since `Parent.g()` is assumed `@Pure`. However, when subsequently examining `Parent.g()` it will realise that `z` must be annotated `@Local`. At this point, it identifies all potential call sites of `Parent.g()` using *Static Class Hierarchy Analysis* [11]. The method `Parent.f()` contains one such call-site and, hence, is re-examined. As a result, `Parent.f()` is downgraded from being `@Pure` and, instead, `x` is annotated `@Local`. The inference must ensure all `@Local` annotations adhere to a contravariant typing protocol. Therefore, it propagates the new `@Local` annotation up the class hierarchy, resulting in `u` being annotated `@Local` in `Child.f()`.

A similar strategy is employed for propagating information about other annotations, such as `@Fresh` and `@Local` on fields, interprocedurally. The inference will continue doing this until no further changes are necessary (i.e. it has reached a fixed-point). Finally, since the inference only relies on static class hierarchy analysis, the resulting annotations are guaranteed to be modularly checkable.

5 Experimental Results

We have implemented our analysis as part of a tool called JPure. This is open source and freely available from `http://www.ecs.vuw.ac.nz/~djp/jpure`. Our main objective with the tool is to develop a set of modularly checkable purity annotations for the Java Standard Library. We are interested in this because it represents the first, and most difficult, obstacle facing any purity system based on annotations.

Our experimental data is presented in Figure 5. Here, column "#Method" counts the total number of (non-synthetic) methods; "#Pure" counts the total number of pure methods (i.e. those annotated `@Pure`, or those annotated with `@Fresh` but with no `@Local` parameters); "#Local" counts the total number of methods with one or more parameters annotated `@Local`, compared with the total number accepting one or more parameters of reference type; "#Fresh" counts the total number of methods guaranteed to return fresh objects, compared with the total number which return a reference type.

When generating this data, our system assumed all classes being inferred (i.e. all those in the packages shown in Figure 5) were *internal*, and all others were *external*. Then, since annotations were not generated for external classes, their methods were conservatively regarded as impure. Thus, we would expect to see greater amounts of purity if more of the standard library were considered in one go (i.e. because some internal methods call out to external methods).

One issue is the treatement of native methods, which our inference assumed were pure. Whilst this is not ideal, it remains for us to manually identify native methods with side-effects. We would not expect this to affect the data since it mostly relates to I/O, and methods such as `Writer.write()` were inferred as impure anyway.

5.1 Discussion

The results presented in Figure 5 show surprising amounts of purity can be uncovered using our purity inference and (modularly) checked with our purity checker. Recall the inference assumed methods in external packages (i.e. packages other than those listed) were impure. By analysing more of the standard library in one go, we may uncover more purity in those packages listed (since they call methods in external packages).

An interesting question is whether or not we could more purity in these packages by further extending our system. By manually inspecting the inferred annotations, we found a few surprises. In particular, neither `java.lang.Object.equals()` nor `java.lang.Object.hashCode()` were inferred as `@Pure`. This was surprising as: firstly, we did not expect implementations of these methods to have side-effects; secondly, these methods are so widely used that their impurity must be having a large knock-on effect. Through a detailed examination of methods which override `java.lang.Object.equals()`, we identified various reasons why it could not be annotated `@Pure`. For example, `java.util.GregorianCalendar.equals()` has side-effects. A common pattern in such methods is to use one or more fields as a cache. The first time the method is called, objects are created and assigned to these fields, whilst subsequent invocations reuse them. This causes a problem for our system, since the field assignment forces the method to be local or — worse still — impure (e.g. if the fields are **`static`**).

pkg	#Methods	#Pure		#Local		#Fresh	
java.lang	1624	995	(61.2%)	103 / 599	(17.1%)	113 / 520	(21.7%)
java.util.prefs	202	75	(37.1%)	5 / 125	(4.0%)	25 / 80	(31.2%)
java.lang.management	130	105	(80.7%)	0 / 16	(0.0%)	34 / 60	(56.6%)
java.lang.instrument	15	15	(100.0%)	0 / 9	(0.0%)	3 / 5	(60.0%)
java.util.concurrent	525	142	(27.0%)	16 / 242	(6.6%)	15 / 164	(9.1%)
java.util.regex	371	181	(48.7%)	32 / 181	(17.6%)	24 / 70	(34.2%)
java.util	2151	647	(30.0%)	171 / 1043	(16.3%)	108 / 745	(14.4%)
java.util.concurrent.atomic	170	41	(24.1%)	8 / 80	(10.0%)	3 / 21	(14.2%)
java.util.concurrent.locks	98	39	(39.7%)	1 / 35	(2.8%)	8 / 15	(53.3%)
java.io	1017	374	(36.7%)	111 / 491	(22.6%)	22 / 153	(14.3%)
java.util.zip	255	131	(51.3%)	36 / 90	(40.0%)	6 / 23	(26.0%)
java.lang.annotation	17	10	(58.8%)	1 / 8	(12.5%)	2 / 10	(20.0%)
java.util.jar	134	39	(29.1%)	3 / 75	(4.0%)	8 / 44	(18.1%)
java.util.logging	238	49	(20.5%)	8 / 140	(5.7%)	2 / 69	(2.8%)
Total	6947	2843	(40.9%)	495 / 3134	(15.7%)	373 / 1979	(18.8%)

Fig. 5. Experimental data on packages from the Java Standard Library.

6 Related Work

Interprocedural side-effect analysis has a long history, with much of the early work focused on compiler optimisation for languages like C and FORTRAN [7, 17]. The use of pointer analysis as a building block quickly became established, and remains critical for many modern side-effect and purity systems (e.g [26, 30, 22]). In such cases, the precision and efficiency of the side-effect analysis is largely determined by that of the underlying pointer analysis. Numerous pointer analyses have been developed which offer different precision-time trade-offs (see e.g. [27, 33, 25]). Almost all of these perform *whole-program analysis* and, as such, are inherently unmodular.

There are several good examples of side-effect systems built on top of pointer analysis. Salcianu and Rinard employ a combined pointer and escape analysis, and generate regular expressions to characterise externally mutated heap locations [30]. Rountev's system is designed to work with incomplete programs [26]. It assumes a library is being analysed, and identifies methods which are observationally pure to its clients. Side-effects are permitted on objects created within library methods, provided they do not escape. The system uses fragment analysis [28] to approximate the possible information flow and is parameterised on the pointer analysis algorithm. Thus, it could be considered a modularly checkable system, provided the underlying pointer analysis was. A critical difference from our work, is the lack of a concept comparable to locality for succinctly capturing side-effects. Instead, raw points-to information feeds the analysis, meaning that any modularly checkable annotations used would necessarily reflect this — making them cumbersome for a human to maintain. In experiments conducted with this system, around 22% of methods were found to be side-effect free. Nguyen and Xue adopt a similar approach for dealing with dynamic class loading [23]. Benton and Fischer present a lightweight type and effect system for Java, which characterises initialisation effects (i.e. writes to object state during construction) and quiesing fields (i.e. fields which are never written after construction) [2]. Their approach is parameterised on the pointer analysis algorithm. As above, this means that, while it could be considered modularly checkable, it would require cumbersome annotations that were hard to maintain. Finally, they demonstrate that realistic Java programs exhibit a high-degree of mostly functional behaviour.

Systems have also been developed which do not rely on interprocedural analysis. Instead, they typically rely on *Static Class Hierarchy Analysis (SCHA)* [11] to approximate the call-graph, as we do. The advantage of this, as discussed in §1, is that it lends itself more easily to modular checking. Clausen developed a Java bytecode optimiser using SCHA which exploits knowledge of side-effects [9]. In particular, it determines whether a method's receiver and parameters are *pure*, *read-only*, *write-only* or *read-write*. Here, *pure* is taken to mean: *is not accessed at all*. Cherem and Rugina describes a mechanism for annotating methods with summaries of heap effects in Java programs [6]. In principle, these could be checked modularly, although they did not directly address this. An interprocedural, context-sensitive analysis is also provided for infering summaries. This differs from our work, in that it is more precise, but generates larger, and significantly harder to understand, annotations.

Aside from compiler optimisations, another important use of purity information lies with specification and assertion languages. The issue here is that, in the specification of a method, one cannot invoke other methods unless they are pure. The Java Modelling Language (JML) provides a good example [20]. Here, only methods marked pure may be used in pre- and post-conditions. In [19] a simple approach to checking the purity of such methods is given — they may not assign fields, perform I/O or call impure methods. However, as discussed in §2, this is insufficient for real-world code, such as found in Java's standard libraries. Barnett *et al.* also considered this insufficient in practice and, instead, proposed a notion of *observational purity* [1]. Thus, a pure method may have side-effects, provided they remain invisible to callers. They permit field writes in pure methods, provided those fields are annotated as secret. JML supports an annotation — `modifies` — which identifies the locations a method may modify. The ESC/Java tool attempts to statically check JML annotations [14]. Cataño identified that it does not check `modifies` clauses, and presented an improved system [5]. However, their system ignores the effect of pointer aliasing altogether. Spec# is another specification language which requires methods called from specifications be pure [10]. Finally, Nordio *et al.* employ pure methods to model pre- and post-conditions for function objects [24].

There are numerous other works of relevance. In [18], an interprocedural pointer analysis is used to infer side-effect information for use in the Jikes RVM. This enabled upto a 20% improvement in performance for a range of benchmarks. Zhao *et al.* took a simpler approach to infering purity within Jikes [36]. Whilst few details were given regarding their method, it appears similar to that outlined in §2. However, they achieved a 30% speedup on a range of benchmarks. In [13], pure methods are used in verify atomocity of irreducible procedures. However, no mechanism for checking their purity was given, and instead the authors assume an existing analysis that annotates methods appropriately. Finifter *et al.* adopt a stricter notion of purity, called *functional purity*, within the context of Joe-E — a subset of Java [12]. A method is considered functionally pure if it is both side-effect free, and deterministic. Here, methods are allowed to return different objects for the same inputs, provided that they are equivalent, and their reachable object graphs are isomorphic. The authors report on their experiences identifying (manually) pure methods in several sizeable applications. DPJizer infers method effect summaries and annotates the program accordingly [32]. This is used to help porting of Java programs to DPJ — a language for writing safe parallel programs [16]. DPJ provides a type system that guarantees noninterference of parallel tasks.

Finally, Xu *et al.* consider a dynamic notion of purity, rather than the more common static approach [35]. They examined the number of methods which exhibit pure behaviour on a given program run. They considered different strengths of purity, and found that, while weak definitions exposed significant purity, this information was not always that useful in practice.

7 Conclusion

We have presented a novel purity system that is specifically designed to generate and maintain modularly checkable purity annotations. The system employs only three annotations, `@Pure`, `@Local` and `@Fresh`, but remains sufficiently flexible for many real-world examples. The key innovation lies in the concepts of locality and, particularly, in the locality invariants and the law of locality. We have evaluated our system against several packages from the Java Standard Library, and found that over 40% of methods were inferred as pure.

Acknowledgments. Thanks to Art Protin for useful feedback on an earlier draft.

References

1. M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. 99.44% pure: Useful abstractions in specification. In *Proc. FTFJP*, pages 11–19, 2004.
2. W. C. Benton and C. N. Fischer. Mostly-functional behavior in Java programs. In *Proc. VMCAI*, pages 29–43. Springer, 2009.
3. K. Bierhoff and J. Aldrich. Lightweight object specification with typestates. In *ESEC/SIGSOFT FSE*, pages 217–226. ACM Press, 2005.
4. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Proc. POPL*, pages 213–223. ACM Press, 2003.
5. N. Cataño and M. Huisman. CHASE: A static checker for JML's assignable clause. In *Proc. VMCAI*, pages 26–40. Springer, 2003.
6. S. Cherem and R. Rugina. A practical escape and effect analysis for building lightweight method summaries. In *Proc. CC*, pages 172–186. Springer, 2007.
7. J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proc. POPL*, pages 232–245. ACM Press, 1993.
8. D. Clarke, J. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *Proc. OOPSLA*, pages 48–64. ACM Press, 1998.
9. L. R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency - Practice and Experience*, 9(11):1031–1045, 1997.
10. Á. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE*, pages 336–351. Springer, 2007.
11. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. ECOOP*, pages 77–101. Springer, 1995.

12. M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable functional purity in Java. In *Proc. CCS*, pages 161–174. ACM Press, 2008.
13. C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. In *Proc. ISSTA*, pages 221–231. ACM Press, 2004.
14. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI*, pages 234–245. ACM Press, 2002.
15. A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *Proc. PLDI*, pages 311–320, 2000.
16. R. L. B. Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *Proc. OOPSLA*, pages 97–116. ACM Press, 2009.
17. W. Landi, B. G. Ryder, and S. Zhang. Interprocedural side effect analysis with pointer aliasing. In *PLDI*, pages 56–67, 1993.
18. A. Le, O. Lhoták, and L. J. Hendren. Using inter-procedural side-effect information in JIT optimizations. In *Proc. CC*, pages 287–304. Springer, 2005.
19. G. T. Leavens. Advances and issues in JML. In *Presentation at Java Verification Workshop*, 2002.
20. G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA Companion*, pages 105–106, 2000.
21. R. Lencevicius, U. Holzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *Proc. OOPSLA*, pages 304–317. ACM Press, 1997.
22. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. *SIGSOFT Softw. Eng. Notes*, 27(4):1–11, 2002.
23. P. H. Nguyen and J. Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In *Proc. ACSC*, pages 9–18, 2005.
24. M. Nordio, C. Calcagno, B. Meyer, P. Müller, and J. Tschannen. Reasoning about function objects. In *Proc. TOOLS*, pages 79–96. Springer, 2010.
25. D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis for C. *ACM TOPLAS*, 30, 2007.
26. A. Rountev. Precise identification of side-effect-free methods in Java. In *Proc. ICSM*, pages 82–91. IEEE Computer Society, 2004.
27. A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Proc. OOPSLA*, pages 43–55, 2001.
28. A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java software. In *Proc. ICSE*, pages 210–220, 2003.
29. A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *Proc. CC*, pages 20–36. Springer, 2001.
30. A. Salcianu and M. Rinard. Purity and side effect analysis for Java programs. In *Proc. VMCAI*, pages 199–215, 2005.
31. O. Tkachuk and M. B. Dwyer. Adapting side effects analysis for modular program model checking. *SIGSOFT Softw. Eng. Notes*, 28(5):188–197, 2003.
32. M. Vakilian, D. Dig, R. L. Bocchino, J. Overbey, V. S. Adve, and R. Johnson. Inferring method effect summaries for nested heap regions. In *Proc. ASE*, pages 421–432, 2009.
33. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using Binary Decision Diagrams. In *Proc. PLDI*, pages 131–144. ACM Press, 2004.
34. D. Willis, D. J. Pearce, and J. Noble. Caching and incrementalisation in the java query language. In *Proc. OOPSLA*, pages 1–18. ACM Press, 2008.
35. H. Xu, C. J. F. Pickett, and C. Verbrugge. Dynamic purity analysis for Java programs. In *Proc. PASTE*, pages 75–82. ACM Press, 2007.
36. J. Zhao, I. Rogers, C. Kirkham, and I. Watson. Pure method analysis within jikes rvm. In *Proc. ICOOOLPS*, 2008.