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 many applications, including automatic parallelisation, extended static checking, and more. We present a novel algorithm for inferring the purity of methods in Java. Our algorithm exploits two properties, called *freshness* and *locality*, which, when combined together, enable more precise purity analysis. Our algorithm also differs from the majority of previous attempts at purity analysis, in that it is modularly checkable. That is, the algorithm produces annotations which can be checked without the need for an expensive and costly interprocedural analysis. We evaluate our analysis against several packages from the Java Standard Library. Our results indicate that it is possible to uncover significant amounts of purity efficiently.**

I. Introduction

In traditional object-oriented programming languages, methods are free to modify global state almost at will. Whilst this provides much flexibility in their implementation, it also presents a number of challenges to both programmers and automated tools [1], [2]. This is because, to reason about a given sequence of statements, one must understand which method calls may or may not affect the visible state. Unfortunately, in modern languages like Java, this cannot easily be done without inspecting their implementations.

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 practical uses, including: specification languages [3], [4], [5], [6], model checking [7], compiler optimisations [8], [9], [10], atomicity [11], query systems [12], [13] and memoisation of function calls [14].

Several existing techniques are known for determining method purity in OO languages (e.g. [15], [16], [17]). The majority of these employ interprocedural pointer analysis as the underlying algorithm. While this yields precise results, it is less suitable in a setting where one wants to *maintain* purity information. For example, where one is writing a library and wishes to restrict all implementations of a given interface method to be pure; or, where one is working in a multi-team environment and, likewise, wishes to enforce that certain methods are always pure. Techniques based on interprocedural pointer analysis are not ideal in this setting for two reasons: firstly, they require the whole program be known in advance [18]; secondly, they require a significant amount of time to run, and this prohibits their use in normal day-to-day development.

Our solution to the problem of maintaining purity information is through the use of annotations. Here, pure methods in library code and derived programs are first annotated with `@Pure`; then, a *purity checker* is provided to help enforce the purity protocol in derived programs. That is, programmers use the tool to check their `@Pure` annotations against the library. We believe, for this approach to be practical, the purity checker must be efficient — otherwise it won't fit within normal day-to-day development. With an inefficient checker, there will always be a lag between development and purity checking, resulting in code which is out-of-synch and always in need of updating to satisfy the purity protocol. The likely effect of this, is that the purity system will be abandoned when deadlines and other pressures loom large.

A sensible way of ensuring the purity checker is efficient is to require that it be modular. That is, it can check the purity of a given method solely by looking at its implementation and the signatures of those methods/classes it depends on. We refer to purity annotations which can be checked in this fashion as being *modularly checkable*. Furthermore, we observe that the majority of previous works on purity analysis, particularly those which depend upon interprocedural pointer analysis, *do not generate modularly checkable annotations*. Hence, they cannot fit in with the approach we are proposing for maintaining purity information.

An obvious difficulty with our approach 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, and this remains a formidable obstacle. 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 be used to check these annotations are correct efficiently at compile-time. The idea behind this system is simple: users can take their existing applications, infer the `@Pure` annotations once using the (potentially expensive) purity inference, and then *maintain* them using the (efficient) purity checker.

A final requirement we believe is critical to success, is that the resulting annotations must remain sufficiently simple. This is because, once the annotations are inferred, we expect programmers to understand and respect 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 inference algorithm which generates modularly checkable annotations, and is implemented as a source-to-source translation. This uses the notions of *freshness* and *locality* to increase the number of methods which can be considered pure, and this is critical to operating with legacy systems.

2) 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.

II. A Simple Purity System

As a starting point, we begin by considering a simple purity system which is surprisingly effective in practice. A key aspect of this system is that it is designed to work in a modular fashion, as outlined in §I. We will then highlight several problems which stem from code found in the Java Standard Library; these problems will motivate our improvement on the simple approach, discussed in Sections III and IV.

A. Overview

The simple purity analysis is designed to be used on legacy code which has not previously been annotated with `@Pure` annotations. In this setting, the tool will be used in a one-off fashion and, hence, there is no requirement that it be efficient (rather, the requirement is that it generate annotations which can be checked efficiently). The aim of the analysis is to update existing Java source with `@Pure` annotations, whilst ensuring they can be checked modularly. The following characterises the meaning of purity within this 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 definition has several implications for the simple purity system. In particular, for a method to be pure, any methods it may call must also be pure — otherwise, it may indirectly assign some field. Therefore, for any call-site, we must conservatively approximate the set of methods that may be invoked as a result by generating a *call-graph*.

To determine the set of methods that may be invoked and also ensure that `@Pure` annotations are modularly checkable, we can only rely on the static information known at that point. Therefore, we employ a well-known technique based on *Static Class Hierarchy Analysis* [19]. For example, consider the following:

```
1 public void f(List<String> x) {
2   x.add("Hello")
3 }
```

When considering this method, we do not know what implementations of `List` may be supplied for `x`. We must assume any are possible and, hence, that the invocation may dispatch to any implementation of `List.add()`. Therefore, for this method to be considered pure, every implementation of `List.add()` must itself be pure.

In addition to this conservative treatment of method invocations, the simple purity analysis must follow a covariant typing protocol. This requires that, for any method which is not pure, every class or interface method which it overrides or implements cannot be annotated `@Pure`. The following illustrates:

```
1 class Parent { public void f() {} }
2
3 class Child extends Parent {
4   int x;
5   public void f() { x = 1; }
6 }
```

If we considered the method `Parent.f()` in isolation, one might conclude it to be pure (indeed, by Definition 1, it is). However, the method `Child.f()` is clearly not pure, since it assigns to field `x`, and this prevents us from annotating either method with `@Pure`.

B. Modular Checking

At this point, we can give an informal argument as to why the annotations produced by the simple purity analysis are modularly checkable. By this, we mean that the purity checker can safely check that a method annotated `@Pure` does indeed meet the requirements of Definition 1. Essentially, there are three cases we must consider:

(1) **Direct Field Assignment.** It is easy enough to determine whether a method may directly assign to a field: we simply inspect its implementation looking for such an assignment.

(2) **Indirect Field Assignment.** Likewise, determining that a method may indirectly assign a field through some method it calls is also relatively easy: we simply inspect every invocation point and check whether the called method is marked `@Pure`, based on its static type.

(3) **Method Overriding.** It is easy enough to check a given method adheres to the required covariant typing protocol: if the method is not annotated `@Pure`, we simply check no method it overrides or implements is marked `@Pure`.

The argument here is essentially that: one can check a method annotated `@Pure` is indeed pure simply by inspecting its implementation, *assuming all other annotations are correct*. The latter point is a subtle, but safe, approach. That is, we might incorrectly conclude a given method `f()` is pure because some method `h()` it calls is incorrectly annotated `@Pure`; however, this is safe because our purity checker will inevitably identify the error when checking method `h()`.

C. Problem 1 — Iterator

We now consider several problems that arose when using the simple purity system on real code. The first problem 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 Java code:

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

Here, we can see that `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.

D. 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 as expected, by copying the contents of `s` into `data` at the correct position. It's fairly clear that `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, in practice, the bytecode generated for this method indirectly calls `AbstractStringBuilder.append()` and, hence, cannot be considered pure. Clearly, we want simple methods such as this to be considered pure.

III. Our Improved Purity System

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

A. Freshness and Locality

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

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

While this method cannot be considered pure under Definition 1, there are circumstances under which one might consider it to be. For example, if the `iterator()` method always returned a *fresh* (i.e. newly allocated) object; and, if `hasNext()` and `next()` only ever *locally* updated the iterator object (i.e. only fields of the object, and not objects it referenced, were updated), then one could consider the method to be *observationally pure*.

To indicate a method returns a fresh object, or that it only ever modifies an object locally, we employ two additional annotations: `@Fresh` and `@Local`. Thus, for the `Collection` and `Iterator` interfaces, our system might generate the following:

```
interface Collection {
  @Fresh Object iterator();
  ...
}

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

Here, we see that `iterator()` returns fresh object and, furthermore, that it is pure (since `@Fresh` implies `@Pure`). Next, we see that `hasNext()` is pure and, hence, cannot have any side-effects. Finally, the `@Local` annotation on `next()` applies to the receiver object, and indicates that it maybe locally modified by `next()`. Note, `@Local` annotations can be placed on any parameter, not just the receiver. Furthermore, the presence of a `@Local` annotation indicates the only allowed side-effects are local modifications to the given parameters (hence, an *impure* method is one with no annotations).

An important question here is: *to what extent do these annotations apply to objects reachable from the annotated reference*. We will make this more concrete in the following section.

B. Understanding Locality

In order to make our informal definition of freshness and locality more precise, we need to consider exactly which parts of an object they apply to. For example, consider an `Iterator` instance returned from `iterator()`. Whilst that instance may be freshly allocated, it's almost certainly not the case that all objects reachable from it are (i.e. the items being iterated over). Thus, we need some way to be clear regarding how much of an object's reachable state is fresh and, likewise, what is considered a local modification.

Our approach is to distinguish between the *internal* and *external* state of an object. This is, in some ways, similar to the notion of *ownership* (see e.g. [20], [21]); however, we are able to exploit some counter-intuitive properties of pure methods to get a simpler, more flexible system. The following example illustrates the main idea:

```
1 class IntList {
2  private int length;
3  private @Local int[] data;
4  private IntList parent;
5
6  public IntList() {
7   length = 0;
8   data = new int[10];
9   parent = null;
10  }
11  public void copy(@Local IntList dst) {
12   dst.length = length;
13   dst.data = new int[length*2];
14   dst.parent = this;
15   for(int i=0;i!=length;++i) {
16    dst.data[i] = data[i];
17 }}}
```

Here, field `data` is annotated `@Local`, which signifies the referred object is part of `MyList`'s *locality*. Field `length` is not annotated `@Local`, but is still in `MyList`'s locality (since the fields themselves always are). Finally, field `parent` is not `@Local`, but has reference type. Here, the field itself is in `MyList`'s locality, *but the referred object is not*.

Now, in `copy()`, the object referred to by `dst` is modified to reflect **`this`** object's state. The following rules are used to determine whether these assignments are valid:

1) A local method may assign fresh objects to `@Local` fields which are either in the locality of a parameter annotated `@Local`, or a fresh object (this is necessary to preserve the *locality invariants*, discussed below). Note, we consider **`this`** a parameter, and that values of primitive type are always fresh. Therefore, the first two assignments above are safe under this rule. The fourth assignment on line 17 is also safe under this rule. This is because field `data` is annotated `@Local` and, hence, the referred object is in the locality of `dst`. Note, we treat array cells as fields of the array.
2) A local method may assign *any* reference to fields which are either in the locality of a parameter annotated `@Local`, or a fresh object, *provided the field is not itself annotated `@Local`*. Therefore, the third assignment on line 14 is safe under this rule.

At this stage, the mechanics of locality should be becoming clear; however, the justification for the above rules may seem somewhat mysterious. In the following section, we will address this in more detail.

C. Locality Invariants

The two rules for checking proper use of locality given in the previous section may seem strange. They are necessary in order to preserve the *locality invariants*, defined below:

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

Locality Invariant 2 (Preservation): When the locality of a fresh object is modified, its locality remains fresh.

Considering Locality Invariant 1 first, our aim is to isolate as much state as possible which is guaranteed to be fresh immediately after the object is constructed. Fields of reference type may contribute to this state, and the `@Local` annotation distinguishes those which do contribute from those which don't. The purpose of locality is to ensure that: *if we know an object is fresh, then it's locality can be safely modified by a method annotated `@Pure`.*

Now, let us consider Locality Invariant 2. In this case, we must preserve the notion of locality through local modifications. Without this guarantee, it is very difficult to be sure such modifications remained local. 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, we see that method `set` violates Locality Invariant 2. This causes a problem in `create` as, after `set` is first called, the locality of `c` no longer adheres to the `@Local` annotations on its fields, and those reachable from it. Therefore, the second call to `set` results in a side-effect, meaning that `create` should not be considered pure (recall `@Fresh` implies `@Pure`).

D. The Law of Locality

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

Law of Locality: one can safely assume that different objects do not share locality (even if they do).

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

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

Here, it is clear that `b` must be annotated `@Local`, since it's 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 they should. However, under the law of locality, we can assume they are not.

So, *why does the law of locality work?* Well, the only situation in which we can exploit `b`'s `@Local` annotation *is when we know `b` is fresh*. Thus, if `a` and `b` are aliased, it immediately follows that `a` is fresh — hence, neither the Locality Invariants nor Definition 1 are violated by the field assignment.

E. Modular Checking

Our purity system still adheres to the notion of purity laid out in Definition 1. However, through the notions of freshness and locality, it can guarantee a larger class of methods is pure, compared with the simple system outlined in §II. In particular, we extend the rules given in §II-B with the following:

(3b) **Method Overriding.** In addition to checking the covariant typing protocol for `@Pure` annotations, we must also perform a similar check for `@Local` and `@Fresh` annotations. In particular, a parameter annotated `@Local` can safely override one annotated `@Local`. Similarly, a method annotated `@Pure` or `@Fresh` may override one with a `@Local` parameter. However, an impure method may not override one with a `@Local` parameter. Furthermore, a return value which overrides one annotated with `@Fresh`, must itself be annotated `@Fresh`. Thus, we see that `@Fresh` follows a covariant protocol, whilst `@Local` follows a *contra-variant* protocol.

(4) **Local Modifications.** We can now check that all state updates remain local to freshly allocated objects. This is done for a given method call by checking the freshness of an argument, compared with the locality of the parameter it is supplied for.

Intermediate Language Syntax:

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

Fig. 1. 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 too illustrate the main ideas. Finally, field accesses and method calls are annotated with the static type of the field/method in question.

The mechanism for checking rule (4) above is more complex than for the other rules given in §II-B. This is because we must infer the freshness of local variables within the method, based on information flow within the method. We do this using a dataflow analysis, the details of which are outlined in the following section.

IV. Implementation

We have implemented our purity analysis from §III as part of a tool called JPure. This performs a source-to-source translation of Java source code, whilst annotating it with `@Pure`, `@Local` and `@Fresh` annotations where appropriate. The tool employs a data-flow analysis to determine the freshness and locality of variables within a method, and propagates that information interprocedurally using static class hierarchy analysis [19]. In this section, we provide more details on both aspects of our analysis, and identify some limitations of our approach.

A. 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 1. 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:

```
1 void meth(MyClass this, Object x) {
2   Object y;
3   y = new T();
4   if(x == null) goto label1;
5   y = x;
6  label1:
7   y.f();
8 }
```

Here, we can see that, for simplicity, the `this` variable is simply passed in as a normal parameter. In this case, the analysis will determine that method `f()` may be called on the object referred to by `x`. If this method is impure, the entire method will 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 will be annotated `@Pure`.

B. Intraprocedural Analysis

A simple intraprocedural analysis provides the basis for our purity system, and underpins both the checking and inference algorithms. In this section, we simply detail the mechanics of this analysis, without considering how it helps determine purity. In the following sections, we will build upon this by explaining how we use the intraprocedural analysis to infer and check purity.

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 are values from $\{?, \epsilon, \ell_{\mathtt{x}}, \ell_{\mathtt{y}}, \ldots\}$. Here, $?$ indicates an unknown object is referenced, ϵ indicates a fresh object is referenced and, finally, $\ell_x, \ell_y, \ldots$ are *named objects*. One named object is provided for each parameter, and represents the object the parameter referenced on entry.

The effect of a statement on the abstract environment is determined by its *abstract semantics*, which we describe using transition rules. These summarise the abstract store immediately after the instruction in terms of that immediately before it. The abstract semantics for the freshness analysis are given in Figure 2. Here, $\Gamma[\mathtt{v} \mapsto \phi]$ returns an abstract environment identical to Γ, except that `v` now maps to ϕ. Similarly, $\Gamma[\mathtt{v}]$ simply returns the abstract reference for `v` in Γ. Several helper methods are used in the semantics:

- `isImpure(m,T_f)` — true iff the given method (determined by its name and static type) is known to be impure. That is, it is neither annotated with `@Pure`, nor any of its parameters are marked with `@Local`.
- `isLocal(f,T)` — true iff the given field (determined by its name and static type) is currently annotated `@Local`.
- `isLocal(i,m,T_f)` — true iff the parameter at position `i` in the given method (determined by its name and static type) is annoated `@Local`.
- `isLocal(`$\{\ell_1, \ldots, \ell_n\}$`)` — true iff the parameters identify by $\{\ell_1, \ldots, \ell_n\}$ are annotated `@Local` in the method currently being analysed.

As an example, let us consider how the freshness analysis applies to the method `meth` given above. The abstract environment which holds before line 3 is:

$$\Gamma = \{\mathtt{this} \mapsto \{\ell_{\mathtt{this}}\}, \mathtt{x} \mapsto \{\ell_{\mathtt{x}}\}, \mathtt{y} \mapsto \{?\}\}$$

Intraprocedural Analysis:

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

$$\frac{}{\mathtt{tf}(\mathtt{v}=\mathtt{w}, \Gamma) \longrightarrow \Gamma[\mathtt{v} \mapsto \Gamma[\mathtt{w}]]} \quad \text{[S-V]}$$

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

$$\frac{\Gamma[\mathtt{w}] = \{\ell_1, \ldots, \ell_n\} \quad \mathtt{T} \neq \mathtt{int} \quad \mathtt{isLocal}(\mathtt{f}, \mathtt{T})}{\mathtt{tf}(\mathtt{v}=\mathtt{w}.[\mathtt{T}]\mathtt{f}, \Gamma) \longrightarrow \Gamma[\mathtt{v} \mapsto \{\ell_1, \ldots, \ell_n\}]} \quad \text{[S-F2]}$$

$$\frac{\Gamma[\mathtt{w}] = \{\epsilon\} \quad \mathtt{T} \neq \mathtt{int} \quad \mathtt{isLocal}(\mathtt{f}, \mathtt{T})}{\mathtt{tf}(\mathtt{v}=\mathtt{w}.[\mathtt{T}]\mathtt{f}, \Gamma) \longrightarrow \Gamma[\mathtt{v} \mapsto \{\epsilon\}]} \quad \text{[S-F3]}$$

$$\frac{\mathtt{T} \neq \mathtt{int} \quad \neg\mathtt{isLocal}(\mathtt{f}, \mathtt{T})}{\mathtt{tf}(\mathtt{v}=\mathtt{w}.[\mathtt{T}]\mathtt{f}, \Gamma) \longrightarrow \Gamma[\mathtt{v} \mapsto \{?\}]} \quad \text{[S-F4]}$$

$$\frac{\neg\mathtt{isLocal}(\mathtt{f}, \mathtt{T}) \quad \{?\} \not\subseteq \Gamma[\mathtt{v}] \quad \Gamma[\mathtt{v}] \neq \{\epsilon\} \implies \mathtt{isLocal}(\Gamma[\mathtt{v}])}{\mathtt{tf}(\mathtt{v}.\mathtt{f}=\mathtt{w}, \Gamma) \longrightarrow \Gamma} \quad \text{[S-W1]}$$

$$\frac{\{?\} \not\subseteq \Gamma[\mathtt{v}] \quad \Gamma[\mathtt{w}] = \{\epsilon\} \quad \mathtt{isLocal}(\mathtt{f}, \mathtt{T}) \quad \Gamma[\mathtt{v}] \neq \{\epsilon\} \implies \mathtt{isLocal}(\Gamma[\mathtt{v}])}{\mathtt{tf}(\mathtt{v}.\mathtt{f}=\mathtt{w}, \Gamma) \longrightarrow \Gamma} \quad \text{[S-W2]}$$

$$\frac{\begin{array}{c}\neg\mathtt{isImpure}(\mathtt{m}) \\ \mathtt{isFresh}(\mathtt{f}) \implies \phi = \{\epsilon\} \\ \neg\mathtt{isFresh}(\mathtt{f}) \implies \phi = \{?\} \\ \mathtt{isLocal}(\mathtt{w}_1, \mathtt{f}, \mathtt{T}_\mathtt{f}) \implies \mathtt{isLocal}(\Gamma[\mathtt{w}_1]) \\ \ldots \\ \mathtt{isLocal}(\mathtt{w}_n, \mathtt{f}, \mathtt{T}_\mathtt{f}) \implies \mathtt{isLocal}(\Gamma[\mathtt{w}_n])\end{array}}{\mathtt{tf}(\mathtt{v}=\mathtt{m}[\mathtt{T}_\mathtt{f}](\overline{\mathtt{w}}), \Gamma) \longrightarrow \Gamma[\mathtt{v} \mapsto \phi])} \quad \text{[S-M]}$$

$$\frac{\begin{array}{c}\neg\mathtt{isImpure}(\mathtt{m}) \\ \mathtt{isLocal}(\mathtt{w}_1, \mathtt{f}, \mathtt{T}_\mathtt{f}) \implies \mathtt{isLocal}(\Gamma[\mathtt{w}_1]) \\ \ldots \\ \mathtt{isLocal}(\mathtt{w}_n, \mathtt{f}, \mathtt{T}_\mathtt{f}) \implies \mathtt{isLocal}(\Gamma[\mathtt{w}_n])\end{array}}{\mathtt{tf}(\mathtt{v}=\mathtt{new}[\mathtt{T}_\mathtt{f}](\overline{\mathtt{w}}), \Gamma) \longrightarrow \Gamma[\mathtt{v} \mapsto \{\epsilon\}])} \quad \text{[S-N]}$$

$$\frac{}{\mathtt{tf}(\mathtt{return}\ \mathtt{v}, \Gamma) \longrightarrow \Gamma[\$ \mapsto \Gamma[\mathtt{v}]]} \quad \text{[S-R]}$$

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

$$\frac{}{\mathtt{tf}(\mathtt{goto}\ \mathtt{L}, \Gamma) \longrightarrow \Gamma} \quad \text{[S-G]}$$

Fig. 2. Abstract semantics of statements for the intraprocedural analysis. Some rule antecedents are shaded in gray — the meaning of this will be explained in §IV-C.

Here, both this and x refer to the objects they did on entry, whilst we know nothing yet about y. Now, the environment holding after line 3 would be:

$$\Gamma = \{\texttt{this} \mapsto \{\ell_{\texttt{this}}\}, \texttt{x} \mapsto \{\ell_{\texttt{x}}\}, \texttt{y} \mapsto \{\epsilon\}\}$$

Here, we see that y now references a fresh object, as expected. Now, the environment which holds after line 5 is:

$$\Gamma = \{\texttt{this} \mapsto \{\ell_{\texttt{this}}\}, \texttt{x} \mapsto \{\ell_{\texttt{x}}\}, \texttt{y} \mapsto \{\ell_{\texttt{x}}\}\}$$

In this case, we see that y now refers to the same named object as x following rule $\texttt{F}-\texttt{V}$. Finally, let us consider the environment which would hold immediately before line 7:

$$\Gamma = \{\texttt{this} \mapsto \{\ell_{\texttt{this}}\}, \texttt{x} \mapsto \{\ell_{\texttt{x}}\}, \texttt{y} \mapsto \{\epsilon, \ell_{\texttt{x}}\}\}$$

Here, we see that y can refer to a fresh object, as well as the named object $\ell_{\texttt{x}}$. Each of these arises from a different control-flow path, and we must conservatively assume either is possible. This means we cannot conclude that y definitely refers to a named object at this point. More precisely, we define the meet of two abstract environments as follows:

$$\begin{aligned}\Gamma_1 \sqcup \Gamma_2 = \quad & \{\texttt{x} \mapsto \phi_1 \cup \phi_2 \mid \texttt{x} \in \mathbf{dom}(\Gamma_1) \cup \mathbf{dom}(\Gamma_2) \\ & \wedge\ \phi_1 = \Gamma_1[\texttt{x}] \wedge \phi_2 = \Gamma_2[\texttt{x}]\}\end{aligned}$$

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

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

$$\Gamma{\downarrow}(\texttt{n}) = \texttt{tf}\big(\texttt{S}(\texttt{n}), \Gamma{\uparrow}(\texttt{m})\big)$$

Here, n and m represent nodes in the control-flow graph, and $\texttt{n}\to\texttt{m}$ the directed edge between them. Similarly, tf denotes the *transfer function*, whose operation is determined by the semantics of Figure 2, whilst $\texttt{S}(\texttt{n})$ gives the statement at node n. Finally, $\Gamma{\uparrow}(\texttt{n})$ determines the abstract environment that holds immediately before node n, whilst $\Gamma{\downarrow}(\texttt{n})$ that which holds immediately after it.

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

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

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.

Finally, it is relatively easy to see that the transfer function is *monotonic*. Furthermore, there are a fixed number of abstract references that can be in $\Gamma(\texttt{v})$, for any variable v. From these facts, it follows that the analysis is guaranteed to terminate. A full proof of this is omitted for brevity, however.

C. Freshness, Locality and Purity Checking

At this stage, whilst the mechanics of the intraprocedural may be understood easily, its meaning in terms of purity analysis may be less clear. We now detail how the intraprocedural analysis helps us in determining method purity.

```
1 class MyList {
2   int length;
3   @Local MyList next;
4   Object data;
5
6   MyList(int len) {
7     this.length = len;
8     if(n == 1) goto label1:
9       this.next = null;
10      goto exit1;
11    label1:
12      this.next = new MyList(len-1);
13    exit1:
14  }
15  void copy(@Local MyList dst) {
16    if(dst == null) goto exit2;
17    int l = this.length;
18    dst.length = l;
19    Object t = this.data;
20    dst.data = t;
21    t = this.next;
22    t.copy(dst.next)
23  exit2:
24  }
25  @Fresh Object clone() {
26    int tmp = this.length
27    MyList t = new MyList(tmp)
28    this.copy(t);
29    return t;
30  }
31  void set(MyList next) { this.next = next; }
32 }
```

Fig. 3. Illustrating the main features of the intraprocedural analysis.

As discussed previously, our purity system breaks into two components: a *purity inference* and a *purity checker*. The former infers @Pure, @Local, and @Fresh annotations from existing source code; the latter checks that the annotations on a given program are used correctly. *The meaning of the rules marked in gray from Figure 2 depend upon whether we are inferring or checking purity annotations*. In this section, we will begin by considering the *purity checker*, as this is the simpler component. In this case, the rules of Figure 2 should be treated just like other rules.

Figure 3 provides a more detailed example than before. Considering the constructor first, there are three assignments statements to consider. The first passes under rule S-W1 from Figure 2, which requires the assigned field is not local and the dereferenced variable to be either fresh, or referencing only named objects annotated @Local. The second two assignments pass under rule S-W2, since they assign @Local fields. In this case, we see why **null** must be regarded as a fresh object.

Considering the copy(MyList) method now. The ab-

stract environment which holds before line 17 is:

$$\Gamma\uparrow(17) = \{\texttt{this} \mapsto \ell_{\texttt{this}}, \texttt{dst} \mapsto \ell_{\texttt{dst}}, \texttt{l} \mapsto \{?\}, \texttt{t} \mapsto \{?\}\}$$

Here, we see that *the analysis assumes parameters are unaliased on entry*. Whilst this seems counterintuitive, it is safe under the law of locality (see §III-D)

The field access statement on line 18 passes under rule E-F1, giving the following environment:

$$\Gamma\downarrow(18) = \{\texttt{this} \mapsto \ell_{\texttt{this}}, \texttt{dst} \mapsto \ell_{\texttt{dst}}, \texttt{l} \mapsto \{\epsilon\}, \texttt{t} \mapsto \{?\}\}$$

From here, the field assignment on line 19 passes under rule S-W1, since parameter `dst` is annotated `@Local`. Then, the field access on line 20 passes under rule S-F2, giving:

$$\Gamma\downarrow(20) = \{\texttt{this} \mapsto \ell_{\texttt{this}}, \texttt{dst} \mapsto \ell_{\texttt{dst}}, \texttt{l} \mapsto \{\epsilon\}, \texttt{t} \mapsto \{\ell_{\texttt{this}}\}\}$$

Again, this may seem counter-intuitive, since dereferencing **`this`** gives $\ell_{\texttt{this}}$. Essentially, this should be taken to mean: `t` still references local state of **`this`**. *But, how do we know `data` still refers to local state?* For example, if method `MyList.Set()` had been called on **`this`** then `data` might point to arbitrary state. The Locality Invariants discussed in §III-C help ensure this is impossible. If we consider the sequence of method calls since the `MyList` object was created, we know that none of these could be impure; hence, the locality invariants ensure that the object's locality has been preserved upto this point. The field assignment on line 21 passes under rule S-W1 because `dst` is annotated `@Local`. Likewise, the method call on line 23 passes under rule S-M because `dst.next` returns local state.

Considering the `clone()` method, we can see the value of locality information. In this case, a fresh object is created and then (locally) modified via the `copy()` method. Thus, the `clone()` method is considered pure and, in fact, is annotated `@Fresh` since it can be shown to return a fresh object (recall that `@Fresh` implies `@Pure`).

Finally, as discussed above, method `set()` will not type check under the rules of Figure 2. This is valid, since the method is not annotated with either `@Fresh` or `@Pure`, and none of its parameters are annotated `@Local`. Therefore, the method is strictly impure.

D. Freshness, Locality and Purity Inference

We now briefly discuss the purity inference algorithm, which is also based on the intraprocedural analysis presented before. In this case, the rules shown in gray from Figure 2 must be treated slightly differently. These rules can require that certain parameters are annotated `@Local`. When undertaking purity checking, this is exactly how they are interpreted. However, when performing purity inference, they are taken as constraints; in other words, when they are needed, the purity inference simply adds the annotations as necessary. However, the purity inference must additionally propagate information across the call graph using static class hierarchy analysis. The following example illustrates:

```
1class Parent {
2 void f(Test x, Test y) { g(x) }
3 void g(Test z) { z.field = 1; }
4}
5class Child extends Parent {
6 int field;
7 void f(Test u, Test v) { }
8}
```

Here, the inference algorithm will quickly conclude that `z` must be annotated `@Local`. At this point, it will identify all potential call sites based on static class hierarchy analysis. The method `Parent.f()` contains one such call-site and, hence, `x` will be annotated `@Local` as well. Finally, the analysis must also ensure all `@Local` annotations adhere to the contravariance requirement discussed in §III-E. Therefore, it must propagate the new annotation up the class hierarchy, resulting in `u` being annotated in `Child.f()`. A similar process occurs for `@Fresh` annotations, although this adheres to a covariant protocol as discussed in §III-E.

V. Experimental Results

We have implemented our analysis as part of a tool called JPure. This is an open source tool which is 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 4. 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 4) 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 improved numbers if more of the standard library were considered in one go (i.e. because some internal methods call out to external methods).

One aspect of our system not considered thus far, is how native methods are treated. In the former case, we assume that native methods are pure. Whilst this is not ideal, it remains for us to manually identify those native methods which can have side-effects. We would not expect this to affect the data, since it should mostly relate to I/O and, in this case, methods system as `Write.write()` were inferred to be impure anyway.

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. 4. Experimental data on packages from the Java Standard Library.

VI. 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 [22], [23]. 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 [15], [16], [17]). 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. [24], [25], [26]). 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 [16]. Rountev's system is designed to work with incomplete programs [15]. 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 [27] 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. 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) [28]. 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)* [19] to approximate the call-graph, as we do. The advantage of this, as discussed in §I, is that it lends itself more easily to modular checking. Clausen developed a Java bytecode optimiser using SCHA which exploits knowledge of side-effects [8]. 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 [29]. 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 [30]. Here, only methods marked pure may be used in pre- and post-conditions. In [3] 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 §II, 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* [4]. 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 [31]. Cataño identified that it does not check `modifies` clauses, and presented an improved system [32]. However, their system ignores the effect of pointer aliasing altogether. Finally, Spec# is another specification language which requires methods called from specifications be pure [6].

There are numerous other works of relevance. In [9], 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 [10]. Whilst few details were given regarding their method, it appears similar to that outlined in §II. However, they achieved a 30% speedup on a range of benchmarks. In [11], 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 [2]. 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.

Finally, Xu *et al.* consider a dynamic notion of purity, rather than the more common static approach [33]. 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.

VII. 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.

References

[1] J. J. Dolado, M. Harman, M. C. Otero, and L. Hu, "An empirical investigation of the influence of a type of side effects on program comprehension," *IEEE TSE*, vol. 29, no. 7, pp. 665–670, 2003.

[2] M. Finifter, A. Mettler, N. Sastry, and D. Wagner, "Verifiable functional purity in Java," in *Proc. CCS*. ACM, 2008, pp. 161–174.

[3] G. T. Leavens, "Advances and issues in JML," in *Presentation at Java Verification Workshop*, 2002.

[4] M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun, "99.44% pure: Useful abstractions in specification," in *Proc. FTFJP*, 2004, pp. 11–19.

[5] K. Bierhoff and J. Aldrich, "Lightweight object specification with typestates," in *ESEC/SIGSOFT FSE*. ACM Press, 2005, pp. 217–226.

[6] Á. Darvas and K. R. M. Leino, "Practical reasoning about invocations and implementations of pure methods," in *FASE*. Springer, 2007, pp. 336–351.

[7] O. Tkachuk and M. B. Dwyer, "Adapting side effects analysis for modular program model checking," *SIGSOFT Softw. Eng. Notes*, vol. 28, no. 5, pp. 188–197, 2003.

[8] L. R. Clausen, "A java bytecode optimizer using side-effect analysis," *Concurrency - Practice and Experience*, vol. 9, no. 11, pp. 1031–1045, 1997.

[9] A. Le, O. Lhoták, and L. J. Hendren, "Using inter-procedural side-effect information in JIT optimizations," in *Proc. CC*. Springer, 2005, pp. 287–304.

[10] J. Zhao, I. Rogers, C. Kirkham, and I. Watson, "Pure method analysis within jikes rvm," in *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*, 2008.

[11] C. Flanagan, S. N. Freund, and S. Qadeer, "Exploiting purity for atomicity," in *Proc. ISSTA*. ACM Press, 2004, pp. 221–231.

[12] R. Lencevicius, U. Holzle, and A. K. Singh, "Query-based debugging of object-oriented programs," in *Proc. OOPSLA*. ACM Press, 1997, pp. 304–317.

[13] D. Willis, D. J. Pearce, and J. Noble, "Caching and incrementalisation in the java query language," in *Proc. OOPSLA*. ACM Press, 2008, pp. 1–18.

[14] A. Heydon, R. Levin, and Y. Yu, "Caching function calls using precise dependencies," in *Proc. PLDI*, 2000, pp. 311–320.

[15] A. Rountev, "Precise identification of side-effect-free methods in java," in *Proc. ICSM*. IEEE Computer Society, 2004, pp. 82–91.

[16] A. Salcianu and M. Rinard, "Purity and side effect analysis for Java programs," in *Proc. VMCAI*, 2005, pp. 199–215.

[17] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to and side-effect analyses for java," *SIGSOFT Softw. Eng. Notes*, vol. 27, no. 4, pp. 1–11, 2002.

[18] A. Rountev and B. G. Ryder, "Points-to and side-effect analyses for programs built with precompiled libraries," in *Proc. CC*. Springer-Verlag, 2001, pp. 20–36.

[19] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proc. ECOOP*. Springer, 1995, pp. 77–101.

[20] D. Clarke, J. Potter, and J. Noble, "Ownership Types for Flexible Alias Protection," in *Proc. OOPSLA*. ACM, 1998, pp. 48–64.

[21] C. Boyapati, B. Liskov, and L. Shrira, "Ownership types for object encapsulation," in *Proc. POPL*. ACM, 2003, pp. 213–223.

[22] J.-D. Choi, M. Burke, and P. Carini, "Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects," in *Proc. POPL*. ACM, 1993, pp. 232–245.

[23] W. Landi, B. G. Ryder, and S. Zhang, "Interprocedural side effect analysis with pointer aliasing," in *PLDI*, 1993, pp. 56–67.

[24] A. Rountev, A. Milanova, and B. G. Ryder, "Points-to analysis for Java using annotated constraints," in *Proc. OOPSLA*, 2001, pp. 43–55.

[25] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using Binary Decision Diagrams," in *Proc. PLDI*. ACM Press, 2004, pp. 131–144.

[26] D. J. Pearce, P. H. J. Kelly, and C. Hankin, "Efficient field-sensitive pointer analysis for C," *ACM TOPLAS*, vol. 30, 2007.

[27] A. Rountev, A. Milanova, and B. G. Ryder, "Fragment class analysis for testing of polymorphism in Java software," in *Proc. ICSE*, 2003, pp. 210–220.

[28] W. C. Benton and C. N. Fischer, "Mostly-functional behavior in java programs," in *Proc. VMCAI*. Springer, 2009, pp. 29–43.

[29] S. Cherem and R. Rugina, "A practical escape and effect analysis for building lightweight method summaries," in *Proc. CC*. Springer, 2007, pp. 172–186.

[30] 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*. ACM Press, 2000, pp. 105–106.

[31] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *Proc. PLDI*. ACM Press, 2002, pp. 234–245.

[32] N. Cataño and M. Huisman, "CHASE: A static checker for JML's assignable clause," in *Proc. VMCAI*. Springer, 2003, pp. 26–40.

[33] H. Xu, C. J. F. Pickett, and C. Verbrugge, "Dynamic purity analysis for java programs," in *Proc. PASTE*. ACM, 2007, pp. 75–82.