社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

Identifying Cross-Function Side Effects using Static Analysis

Jiachen YANG†, Keisuke HOTTA†, Yoshiki HIGO†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University, 1-5, Yamadaoka, Suita-shi, Osaka, 565-0871, Japan

E-mail: †{jc-yang,k-hotta,higo,kusumoto}@ist.osaka-u.ac.jp

Abstract Side effects are modifications done to the state of the objects in Object-Oriented Programming languages such as Java. Side effects can happen across the boundary of the functions, therefore they are important properties that often be neglected by the documentations. In this paper, we studied the side effects of functions by using a static analysis method to automatically infer the state dependencies for the return value and side effects of functions. We also present a set of annotations to document these state dependencies. As a result, the annotations presented in this paper reveals well-defined state interactions between objects. These annotations can be used in further investigations by both programmers and tools. We have implemented the analysis method targeting Java bytecode and tested it on vary-sized open source Java software libraries. From our experimental results, we found that 25–33% of the functions in the evaluated open source Java libraries are pure functions, which indicates that a large percentage of the functions is suitable for high level refactoring. And we present a case study of equals and hashCode functions to show the importance of our method in real world programming tasks.

Key words state boundary, state dependency, object-oriented, effect annotation, static analysis

1. Introduction

It is difficult for programmers to use software components without fully understanding their behavior. The documentation and naming of these components usually focuses on *intent*, i.e., what the functions are required to do, but fails to illustrate their *side effects*, i.e., how these functions accomplish their task [1]. Because of the possible side effects in API libraries, it is hard to reuse the modularized components. In addition, undocumented API side effects may be changed during software maintenance, making debugging even more challenging in the future [7]. With the understanding of side effects in the software libraries, programmers can perform high level refactoring on the functional part of the source code. However, the purity information is usually missing in external libraries, therefore programmers would risk introducing bugs with such refactorings.

In this paper, we present an approach to infer a function's purity from byte code and automatically document effect annotations for later use by programmers and static processing tools. Programmers can use effect annotations to understand a function's side effects, whereas static processing tools can use them for static checking, optimizing or refactoring.

The contributions of this research include:

- An extended definition of purity as *stateless* or *stateful* in object-oriented(in short, OO) languages such as Java.
- An approach to automatically infer effect annotations, as well as a concrete implementation for Java bytecode.
- A set of function annotations that document the details of effects such as return value dependencies or variable state modifications, for programmers to understand the effects.
- Experiments on well-known open source software libraries with varies size of code bases. In our experiments, we observed modifier functions that should be pure, revealing tricks or potential bugs in the implementation.

2. Purity and Effect Annotations

In this section, we firstly discuss our definition of the purity and side effects. Secondly, we define a set of annotations to document these informations. Lastly, we discuss the rules that should be followed by these annotations.

2.1 Stateless & Stateful Purity of Functions

The notion of purity on functions does not match well with other OO programming paradigm concepts. In OO languages, program states are usually encapsulated within objects, which use well-defined boundary functions called methods to interact with each other. This is the opposite of a pure functional paradigm where the states of the program are passing through function arguments.

Moreover, we noticed that most objects have a life span pattern of creation, use and destroy. Many objects will not change their states after properly created, and the functions called on them simply query these internal states. We would like to distinguish these state-querying functions from those functions that modify the states. Through our research we have observed that OO libraries can still contain around 25–33% of functional code that do not modify the program's state (e.g., tree traversals).

Based on the above observation, we define a function as *pure* if it does not generate side effects such as modifying the state outside the object. Note that this definition is slightly different from the traditional definition of pure functions by return value dependencies [6]. Meanwhile, many existing studies such as [3, 5, 9] share the same purity definition with us. To illustrate the difference of two definitions, we divide the our definition of a *pure* function into *stateless* and *stateful* functions:

[Definition 1] (Stateless) If the return value of a pure function is only determined by the state of its arguments.

[Definition 2] (Stateful) If the return value of a pure function is also determined by the states of member fields or static fields.

2. 2 Lexical State Accessors and Side Effects

The main purpose of this research is to reveal the side effects of functions. Therefore, we need to define what is an effect and what is a side effect of a function.

[Definition 3] (Effect) We define the effects of a function as the modifications to the states of the program, including the return value.

[Definition 4] (Side Effect) We define the side effects of a function as the modifications to the states of the objects or performing I/O operations.

The effects of a function are all the side effects including the return value. According to the *single response principle* in [4], a function should have exactly one effect, either calculating a value and return it, or doing one kind of modification to the state of the program. Disobeying this practice usually leads to problematic, unmaintainable coding style.

[Definition 5] (Lexical State Accessor) We define a *lexical state accessor* to be any variable that is directly accessible within a function's lexical scope before the execution.

In statically-typed object-oriented languages such as Java, lexical state accessors of a function include the possible `this` pointer, the arguments, the member fields within the same class, and the static fields in any arbitrary classes. Note that local variables defined inside a function are excluded in the definition of lexical accessors, because they do not exist outside the function's body. We focus on lexical variables because they can be easily identified and understood from the function definition by programmers.

All possible modifications to the state of a program are

```
class Tree extends Comparable {
  @Depend(dependThis=true,
    dependFields= {"Tree Node.r", "int Leaf.f",
                   "Tree Node.l"}
    from = {"int Leaf.freq()","int Node.freq()"})
  int freq() { return 0; }
  @Depend(dependThis=true,
    dependArguments= {"Tree tree"},
    dependFields= {"Tree Node.r", "int Leaf.f",
                   "Tree Node.l"})
  int compareTo(Tree t) { return freq() - t.freq(); }
}
class Leaf extends Tree {
  char v; int f;
  @Field(type=int.class, owner=Leaf.class,
    name="f", dependArguments= {"int fr"})
  @Field(type=char.class, owner=Leaf.class,
    name="v", dependArguments= {"char va"})
  Leaf(int fr, char va) { f = fr; v = va; }
  @Depend(dependThis=true,
    dependFields= {"int Leaf.f"})
  int freq() { return f; }
}
class Node extends Tree {
  Tree l, r;
  @Field(type=Tree.class, owner=Node.class,
    name="l", dependArguments= {"Tree le"})
  @Field(type=Tree.class, owner=Node.class,
    name="r", dependArguments= {"Tree ri"})
  Node(Tree le, Tree ri) {
    f = le.f + ri.f;
    l = le;
    r = ri;
  }
  @Depend(dependThis=true,
    dependFields= {"Tree Node.r", "Tree Node.l"})
  int freq() { return l.freq() + r.freq(); }
}
class Main{
  @Depend(dependArguments= {"int[] chrs"})
  Tree build(int[] chrs) {
    PriQueue q = new PriQueue();
    for(int i = 0;i < chrs.length; i = i + 1){
      if (chrs[i] > 0){
        q.offer(new Leaf(chrs[i], (char)i));
      }
    }
    for (;q.size() > 1;) {
      q.offer(new Node(q.poll(),q.poll()));
    }
    return q.poll();
  }
  void main(String[] args) {
    String test = "this is an example";
    int[] chrs = new int[256];
    for(int i = 0;i < test.length(); i = i + 1){
      char c = test.getChar(i);
      chrs[c] = chrs[c]+1;
    }
    Tree tree = build(chrs);
  }
}
```

Figure 1 Annotated Huffman Algorithm

achieved by accessing the aforementioned lexical state accessors. There are two forms of modification: changing the values of these accessors directly, or modifying indirectly though the use of lexical state accessors. These modifications are considered to be the side effects of executing the function. Additional side effects include calling system routines to perform I/O operations directly or transitively.

2. 3 Effect Annotations

We introduce a set of function annotations to indicate the effects that can arise during invocation. For each function, several annotations can be prepended, each representing a side effect that for example modifies one member field. The proposed annotations express the effects such as direct or transitive modifications to lexical state accessors, with the possible data dependency between these effects and other lexical state accessors from the function. The data dependencies are captured in annotation records such as `dependThis`, `dependArguments`, `dependFields` and `dependStatic`, with detailed information such as types and owner classes of the fields. Although the `this` pointer is not a mutable variable in the context of a target function, it is possible to compare the identity by using `this` pointer to other pointers or expose `this` pointer as return value of the function, hence the `this` pointer is included in data dependency.

For example, the annotated version of the source code of the Huffman algorithm is represented in Figure 1. The effect annotations are intended to be used by both programmers and tools that process the program, as a contract describing the given function. This contract can be viewed as a complimentary to the function signature and exception specification that imposes restrictions on the implementation of the function.

3. Automatic Inference of Effect Annotations

Asking developers to manually annotate effect annotations is tedious, error-prone, and infeasible for third party libraries. In this section, we present our approach to automatically inferring effect annotations.

The analyzer identifies function targets by using a class diagram and call graph. The class diagram records the inheritance relationship of classes (including interfaces) and the overriding relationship between functions in a class hierarchy. The call graph records the invocation instructions inside the function, which points to another function defined in the class diagram.

3.1 Data and Control Analysis

Our analyzer traverses all of the functions in the class diagram, inferring possible effects including side effects. We capture only the dependencies of lexical state accessors, which are defined in in Subsection 2.2, during these three analysis stages:

- **data flow analysis** estimates the return value dependency.
- **reference alias analysis** identifies possible modifications to lexical state accessors that are side effects.
- **control flow analysis** supports data dependence calculations on conditional branches.

There are three kinds of lexical state accessors as defined in Section 2.2, which are the *static fields* (shortened as S) of a class, the *member fields* (shortened as F) of an object, and the *arguments* (shortened as A) passed to the function.

[Definition 6] (Data Dependency Set) We define a data dependency as the value of a lexical state accessor before a function executes, and a *dependency set* (DS) as the set of data dependencies such that $DS \subset \{x | x \in S \cup F \cup A\}$.

The above definition of *dependency set* is used in both our data flow analysis and reference alias analysis. The difference between the *dependency sets* used in these two analyses is that we only consider reference type dependencies in reference alias analysis, and value type dependencies in data flow analysis. All dependencies suitable in reference alias analysis are also suitable in data flow analysis, but not vice versa. We define two *dependency sets* used in these two stages of analysis as:

- **reference dependency** (rd) is a DS of the possible reference aliases.
- **value dependency** (vd) is a DS that affects the value.

Our analyzer interprets the code, follow the instructions in the given function, and applies the aforementioned three analysis. The analyzer begins its interpretation by breaking the code of a given function into statement *block*s using control flow analysis, where we define a *block* to be a sequence of statements. The *block* can be associated with a value of its condition if it is nested in an *if* or *while* statement. Next, the analyzer interprets the each *block*'s instructions to evaluate the value dependencies and obtain a list of effects. During the interpretation stage, each value is represented as a triplet of its static type, a reference-dependency set, and a value dependency set ($V = (\text{type}, rd, vd)$).

At the beginning of the interpretation of the given function, the argument values are assigned with value and reference dependencies of themselves. Next we interpret each instructions of the function by following the transfer functions in Table 1. The input of a transfer function is V before the execution of the instruction, and the output is a new V after the execution. Besides the reference and value depen-

Table 1 Transfer Functions for Values and Instructions

Type	Code Pattern	RD	VD
vn	**new** τ	$\emptyset$	$\emptyset$
vp	x	$\{x\}$	$\{x\}$
vl	y	$\emptyset$	$\emptyset$
vt	`this.field`	{field}	{field}
vs	`Class.field`	{field}	{field}
vf	V`.field`	V_{rd}	V_{vd}
vu	$op\, V$	$\emptyset$	V
vb	$V_1\, op\, V_2$	$\emptyset$	$V_{1vd} \cup V_{2vd}$
va	$V_1[V_2]$	V_{1rd}	$V_{1vd} \cup V_{2vd}$
vc	$(\tau)\, V$	V_{rd}	V_{vd}
assign	V_1 = V_2	V_{1rd}	V_{2vd}
return	`return` V	$\emptyset$	$\emptyset$
merge		$V_{1rd} \cup V_{2rd}$	$V_{1vd} \cup V_{2vd}$

dency sets in this table, the static types of these values should also be calculated as defined in the language specifications. Note that the "merge" instruction in this table merges the branches of statements during the interpretation. Besides the instructions listed in the table, there is another important kind of instructions, the function invocations, described in Section 3. 2. During interpretation, possible function effects are collected when processing assignment instructions. We initially mark two kinds of dependencies: *modification behavior* for reference dependencies and *return statement* for value dependencies. Both dependencies are merged with the value dependency set for the current block.

3. 2 Effects from Function Invocations

We refer to the function containing an invocation as a *caller*, and the function being called as a *callee*. When the analyzer sees a function invocation instruction during interpretation, it generates possible effects by examining the date flow across the invocation boundaries. Fortunately, this cross-function analysis is possible with the generated effect annotations on the callee, so that we do not need to examine the codes of the caller and callee at the same time. When the effect annotations on the callee are not available during analysis of a caller, the analyzer simply ignores the invocation, pretends callee has no effects, and then refreshes the result when the annotations on the callee become available.

All of the invocation instructions share the same form as V_{obj}`.function(`$\overline{V_{arg}}$`)`. All side effects on static fields are transferred from callee to caller. If there are argument effects generated on the callee function, i.e., when the callee is modifying the state of a passed argument, then the analyzer will generate a modification behavior on the reference dependencies of corresponding position, as if the modification occurs inside the caller function.

The V_{obj} is the object that owns the function, which could be `this`, `ClassName` or a certain dynamically calculated value during the interpretation. Static member functions on `ClassNames` are guaranteed not to generate modification side effects on member fields. If a reference dependency of V_{obj} is `this`, all the modification side effect annotations on member fields will be copied, otherwise a single modification effect on the reference dependency of the current V_{obj} will be recorded. This behavior of analyzer follows the definition of lexical state accessors described in Section 2. 2, to distinguish between directly and transitively accesses of these accessors.

Finally, if the interpreted invocation expression returns a value, we need to determine the reference and value dependency of its return value. The reference dependency of the invocation expression is the reference dependency of return value from callee, and the value-dependency of this expression is the merged value dependencies of all V_{arg}s.

With the effect annotations on the functions, we can simply determine whether a function is a *pure function*, and further, whether it is *stateful* or *stateless*. A function that has no modification annotations is considered to be a *pure function*. A pure function whose return value depends only on arguments is considered to be a *stateless* pure function.

3. 3 Iteration to a Fix-point of Class Diagram

A function's effect annotations depend on the annotations of its callees as well its overriding functions, potentially causing a function to be analyzed several times. In addition, recursive functions may also be analyzed multiple times. We continue analyzing until the effect annotations are inferred. We set a flag in each function on the class diagram to indicate whether the effects for this function need to be inferred or updated.

Firstly, we initialize all functions in the class diagram with *effects* as $\emptyset$. Next we mark the flags for all of these functions as "need to be analyzed". Then, for each function whose flag is marked, the analyzer:

（1） Merges the *effects* with the result of the data analysis on this function.

（2） Clears the flag on this function.

（3） If the effects have changed since last analysis, marks the flags of all functions that depend on this function.

We continue the iteration until none of the functions in the class diagram are marked, which means the reach of a fix-point of the analysis.

4. Experiments

We evaluated our analyzer on real world software components in terms of accuracy, performance, and the distribution of different kinds of effects in different scale of software components.

4. 1 Distribution of Effect Annotations

To show the distribution of purity and side effects of the functions in real world software components, we experimented on 4 target software projects, listed in Table 2. These experiments were executed on an octa-core Xeon E5520 CPU with a 16GB heap size limitation. *purano* is the implementation of the analyzer presented in this paper, which includes a modified version of the ASM library. Both *htmlparser*, *tomcat* and *argouml* are well-known open source Java libraries

Table 2 Experiment Target and Analysis Performance

Software	All Classes	User Classes	User Functions	Time (sec.)	Pass
purano	3,541	232	2,120	215	19
htmlparser	4,843	158	1,621	344	22
tomcat	6,658	763	8,662	665	23
argouml	11,234	2,544	20,274	2,217	28

```java
package javax.swing.text;
public class DefaultCaret extends Rectangle ... {
  /* Compares this object to the specified object.
   * The superclass behavior of comparing
   * rectangles is not desired, so this is changed
   * to the Object behavior.
   */
  public boolean equals(Object obj) {
    return (this == obj);
  }
}
```

Figure 2 A Special Design in `DefaultCaret`

```java
package java.io;
public final class FilePermission ... {
  public boolean equals(Object obj) {
    if (obj == this) return true;
    if (!(obj instanceof FilePermission))
      return false;
    FilePermission that = (FilePermission) obj;
    return (this.mask == that.mask) &&
      this.cpath.equals(that.cpath) &&
      (this.directory == that.directory) &&
      (this.recursive == that.recursive);
  }
  public int hashCode() {
    return 0;
  }
}
```

Figure 3 A Potential Problem in `FilePermission`

and we use their latest stable binary distributions. Based on the analysis times in Table 2, we determine that the performance of our analyzer is reasonable within a daily programming environment, although it could be further optimized by caching the result of the standard libraries.

The purity of functions of the experimental result is listed in Table 3. From the annotation output, we find that around 25–33% of the functions in these software projects were marked as *stateless* or *stateful* pure functions. We manually confirmed the generated result for *purano* to make sure it matched our expectation.

4.2 A Case Study: Purity of `equals` and `hashCode`

Different programmers may use our tool for their own usages. Therefore, we conducted a case study to illustrate one possible usage of our study. We examined the inferred annotations on two functions, namely `equals` and `hashCode`. These two functions are related with the value equality of objects in Java, and they are used by collection classes such as `HashMap`. The programmer must ensure that the return values of these functions reflect their value equalities, and hence these return values should depend on the state of the objects. Therefore, we expect these functions to be *stateful* pure functions if they contain member fields. The purity types of these two functions are listed in Table 4. We can see from the distribution that nearly 70% of these functions were annotated as *stateful pure*, as expected.

To further understand the result, firstly we focused on the existence of *stateless* pure functions in Table 4 by manually examining their source code. Most of these functions are defined interfaces or abstract classes. There were also 2 `equals` and 6 `hashCode` functions defined in classes without member fields. There were 9 `equals` that compare referential identities defined in classes with member fields. These were used in unusual cases when comparing by referential identity rather than value identity is desired. An example of this kind of special design can be found in `DefaultCaret.equals`, shown in Figure 2, where the author explicitly documented in the Javadoc as "The superclass behavior of comparing rectangles is not desired, so this is changed to the Object behavior". In addition, most of these classes are inner classes in Java with their names containing a "`$`" character. These inner classes are supposed to be used internally, where programmers control the creation of all objects. We found 3 `hashCode` functions that return a constant, whereas their corresponding `equals` functions compared the states of member fields. An example is shown in Figure 3, that `FilePermission.hashCode` will always return 0. The user of these classes must be aware of their respective behaviors, in order to avoid putting them in a `HashSet` or `HashMap`, or comparing them using equals. With the introduction of effect annotations as documentation, the user of these classes can notice the special behavior.

Next we examined the functions in Table 4 that generate side effects. Most of these functions cached the result of calculations inside the object, similar to functions on the white-list. Some classes such as `Date` and `Calendar` normalized their internal representation before comparing equality or calculating the hash code. Classes used in reflection at runtime, such as `java.lang.reflect.Class`, used a lazy loading technique to optimize general performance, which is a variant of caching technique.

5. Related Work

Many previous efforts on combining pure functional style into an object-oriented paradigm concentrate on introducing immutable restrictions on existing type systems, as in functional programming languages. Tschantz, et al. [10]

Table 4 Purity of `equals` (as `e`) and `hashCode` (as `h`)

Software		All	Pure Functions		Side Effects
			Stateless	Stateful	
purano	e	335	13 (3.9%)	237 (70.7%)	85 (25.4%)
	h	316	25 (7.9%)	224 (70.9%)	67 (21.2%)
htmlparser	e	290	19 (6.6%)	198 (68.3%)	73 (25.2%)
	h	284	23 (8.1%)	199 (70.1%)	62 (21.8%)
tomcat	e	407	23 (5.7%)	277 (68.1%)	107 (26.3%)
	h	399	30 (7.5%)	268 (67.4%)	101 (25.3%)
argouml	e	358	24 (6.7%)	239 (66.8%)	95 (26.5%)
	h	344	28 (8.1%)	248 (72.1%)	68 (19.8%)

Table 3 Percentage of Effect Annotations

Software	Pure Functions		Side Effects	Modifying		
	Stateless	Stateful		Member Fields	Static Fields	Arguments
purano	319 (15.0%)	203 (9.6%)	1,598 (75.4%)	1,425 (67.2%)	1,054 (49.7%)	430 (20.3%)
htmlparser	286 (17.6%)	257 (15.9%)	1,078 (66.5%)	718 (44.3%)	782 (48.2%)	145 (8.9%)
tomcat	894 (10.3%)	1,782 (20.6%)	5,986 (69.1%)	4,684 (54.1%)	4,318 (49.8%)	1,238 (14.3%)
argouml	4,239 (20.9%)	1,486 (7.3%)	14,549 (71.8%)	8,345 (41.2%)	12,824 (63.3%)	3,902 (19.2%)

proposed Javari as a new programming language that adds `readonly` and other keywords into Java syntax to indicate the reference immutably of variables. Based their work, Quinonez [2] proposed an analyzer called Javarifier to automatically infer reference immutability in Javari syntax. All these type-system-based approaches require syntax modification of the source code. Although they can be applied in newly developed projects, it is much more difficult for these approaches to be adopted in legacy libraries, and existing tools such as IDE support need to be extended to accept their new syntax.

There are studies of automatic purity analyzers on unmodified syntax. Sălcianu, et al. present a purity analyzer for Java in [9], which uses an inter-procedural pointer analysis [8] and escape analysis to infer reference immutability. Similar to our approach, they verify the purity of functions, but their pointer and escape analysis relies on a whole program analysis starting from a `main` entry point, which is not always available for software libraries. JPure [5] eliminated the need for reference immutability inference by introducing `pure`, `fresh` and `local` annotations, which lead to a more restrictive definition of purity, and loses the exact information for effects. Both studies focus on analyzing of purity only, and does not expose effects information outside their toolchain. Compared with these studies, our approach uses lexical state accessor analysis, which will hopefully combine the modularity of JPure by illuminating the need for inter-procedure analysis, and the flexibility of reference immutability with the availability of effect information. Also neither of these two studies further classify the pure functions into *Stateless* and *Stateful* as we do.

6. Future Work and Conclusions

The current implementation of our analyzer works on Java bytecode rather than source code. analysis tool at Besides all the advantages described, this target language decision is made to ease the early development, because it is easy to generate bytecode from source code by a compiler but not vice versa. However, targeting source code format is still important for integrating as an IDE plugin. We plan to add a source code analyzer in the future.

Moreover, we plan to further evaluate the usability of these effect annotations, by programmers as well as by analysis tools. The format of these annotations needs to be more readable and understandable to be used by programmers. We will also further investigate the applications of these effect annotations other than identification of pure functions.

To conclude, in this paper we presented a study on the purity and side effects of the functions in OO languages such as Java, helping programmers to understand the software libraries. We proposed a method to automatically infer the purity and side effect information from Java bytecode. We implemented and experimented the proposed method on real world Java software libraries, and found that around 25–33% of all the function of the Java libraries is made of pure functions. We compared the accuracy of distribution of pure functions with existing study. And we demonstrated how programmers will use our method to understand the behavior of library APIs by a case study.

Acknowledgment

This work was supported by MEXT/JSPS KAKENHI 25220003, 24650011, and 24680002.

References

[1] B. Goetz. Java theory and practice: I have to document that? `http://www.ibm.com/developerworks/java/library/j-jtp0821/index.html`, 2002.

[2] Q. J. *Javarifier: Inference of reference immutability in Java.* PhD thesis, Massachusetts Institute of Technology, 2008.

[3] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of jml. Technical report, Technical Report 96-06p, Iowa State University, 2001.

[4] R. C. Martin. *Clean code: a handbook of agile software craftsmanship.* Prentice Hall, 2008.

[5] D. J. Pearce. Jpure: a modular purity system for java. In *Compiler Construction*, pages 104–123. Springer, 2011.

[6] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84. ACM, 1993.

[7] C. Raymond. The importance of error code backwards compatibility. `http://blogs.msdn.com/b/oldnewthing/archive/2005/01/18/355177.aspx`, 2005.

[8] A. Sălcianu. *Pointer analysis and its applications for Java programs.* PhD thesis, Citeseer, 2001.

[9] A. Sălcianu and M. Rinard. Purity and side effect analysis for java programs. In *Verification, Model Checking, and Abstract Interpretation*, pages 199–215. Springer, 2005.

[10] M. Tschantz and M. Ernst. *Javari: Adding reference immutability to Java*, volume 40. ACM, 2005.