ReImInfer: Method Purity Inference for Java

Wei Huang Ana Milanova
Rensselaer Polytechnic Institute
110 8th Street, Troy NY
{huangw5, milanova}@cs.rpi.edu

ABSTRACT

Method purity inference, also known as side-effect analysis, is an important problem. It has many applications including compiler optimization, model checking, memoization of function calls, atomicity, etc. Surprisingly, despite the long history of this problem, we know of no purity inference tool that scales to large codes and analyzes both whole programs and libraries.

We build a purity inference tool called ReImInfer on top of a type inference and checking framework. ReImInfer infers method purity for Java. It is modular and compositional, produces precise results and scales to large programs.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming

Keywords

method purity, type inference

1. INTRODUCTION

A method is *pure* (or side-effect free) when it has no visible side effects. Knowing which methods are pure has a number of applications:

- Compiler optimization: Clausen presents Cream for optimizing Java bytecode using side-effect analysis [3]. The evaluation in [3] shows that the optimizations benefit from side-effect analysis. Le et al. use side-effect information to improve performance in a just-in-time (JIT) compiler [9]. They make use of side-effect analysis in local common sub-sexpression elimination, heap SSA, redundant load elimination and loop-invariant code motion.
- Model checking: Tkachuk and Dwyer adapt side-effect analysis to model side-effecting values to improve precision of model checking via *return sensitive* and *must* side-effect analyses [13].
- Memoization of function calls: Heydon et al. use caching pure function calls to improve runtime performance [6].
- Atomicity: Flanagan et al. exploit purity for reasoning about atomicity [5]. They present a static analysis for verifying the atomicity by applying reduction to a program abstraction based on purity and instability.
- Universe types inference: Universe types [4] require method purity information. In previous work [7], we present an inference approach for Universe types which uses ReImInfer to decide the purity of methods.

In order to facilitate these applications, purify inference must be precise and scalable. Purity inference (also known as side-effect analysis) has a long history. The two best-known publicly available tools are JPPA by Sălcianu and Rinard [12] and JPure by Pearce [11]. Unfortunately, both tools are unsuitable. JPPA is a whole-program analysis and cannot handle libraries. Additionally, it crashes frequently due to the fact that it is based on a custom compiler. JPure is modular and scalable by design. Unfortunately, it is fragile. We were able to analyze only the smallest programs from our suite (after contacting JPure's author, David Pearce, for help).

We build ReImInfer, a tool that infers method purity for Java. The tool was built because of our concrete needs for method purity inference and only after spending considerable time trying to make JPPA and JPure work. We needed purity inference for Universe type inference [7], flow-sensitive local type inference, etc. We envision other applications as well.

ReImInfer is *modular* and *compositional*. It is modular in the sense that it can analyze any given set of classes L. If there are unknown callees in L, the analysis assumes appropriate default typings. Callers of L can be analyzed separately and composed with L without re-analysis of L.

ReImInfer is *scalable*. The purity inference in ReImInfer is based on reference immutability inference, which has $O(n^2)$ worst-case complexity and works linearly in practice. It took less than 1 minutes in our largest benchmark xalan, which has more than 300KLOC, on a server with Intel® Xeon® CPU X3460 @2.80GHz and 8 GB RAM (the maximal heap size is set to 2 GB).

2. METHOD PURITY INFERENCE

In this section, we discuss the analysis behind ReImInfer. ReImInfer infers method purity based on *reference immutability* which ensures that an immutable reference is not used to

SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.

modify the referenced object. The key idea is to decide the mutability of references passed into the method. If all such references are immutable, we conclude that the method is pure.

2.1 Reference Immutability

We propose a context-sensitive type system for reference immutability named ReIm [8]. ReIm is similar to Javari [14], the state of the art in reference immutability, but differs in important points of design and implementation. ReIm is designed with purity inference in mind. It forgoes the assignability feature of Javari in order to ensure the desired semantics for purity. One key novelty in ReIm is the use of *viewpoint adaptation* to encode context sensitivity; Javari uses templating for this purpose. The use of viewpoint adaptation contributes to the better scalability of ReImInfer compared to Javarifier, Javari's inference tool.

Immutability Qualifiers.

Qualifiers are applied to fields, local variables, formal parameters, and return values in ReIm. There are three immutability qualifiers in ReIm:

- mutable: A mutable reference can be used to mutate the referenced object; this is the implicit and only option in standard object-oriented languages.
- readonly: A readonly reference x cannot be used to mutate the referenced object nor anything it references. For example, x.f = z is not allowed as this assignment mutates the object referred to by x.
- polyread: polyread expresses polymorphism over immutability. A polyread reference is immutable in the current context, but it may or may not be mutable in other context. The interpretation of polyread depends on the context, i.e. it can be instantiated to either mutable or readonly. For example, x.f = null, where x is polyread, is not allowed, but z = id(y); z.f = null, where id is defined as polyread X id(polyread X p) {return p;}, is allowed when y and z are mutable. In the latter case, the polyread return value of id is instantiated to mutable.

The subtyping relation between the qualifiers is

$$\text{mutable} <: \text{polyread} <: \text{readonly}$$

where $q_1 <: q_2$ denotes q_1 is a subtype of q_2.

Context Sensitivity.

Consider the DateCell example in Figure 1. The return value of getDate is mutated at line 6 through md. A context-insensitive type system would type the return type of getDate as mutable, which would force the this of getDate to be mutable. As a result, this of cellGetHours has to be mutable as well because of the call at line 9 and only mutable references can be called on cellGetHours. However, we can see that there is no object mutation in either getDate or cellGetHours. A context-insensitive type system imposes an unnecessary restriction on method calls. Furthermore, our purity inference algorithm (Section 2.2) would conclude that cellGetHours is impure, because its implicit parameter this is mutable.

We express context sensitivity using qualifier polyread and viewpoint adaptation [4]. Viewpoint adaptation of a type

```
1  class DateCell {
2    Date date;
3    Date getDate(DateCell this) { return this.date; }
4    void cellSetHours(DateCell this) {
5      Date md = this.getDate();
6      md.setHours(1);              // md is mutated
7    }
8    int cellGetHours(DateCell this) {
9      Date rd = this.getDate();
10     int hour = rd.getHours();   // rd is readonly
11     return hour;
12   }
13 }
```

Figure 1: A Date cell. The formal parameter this is shown explicitly for readability.

q' from the point of view of another type q, results in the adapted type q''. This is written as $q \triangleright q' = q''$. Viewpoint adaptation is applied at field accesses and method calls, where the "context" changes. We define $\triangleright$ for ReIm as follows:

$$\begin{array}{rcl} \_ \triangleright \text{mutable} & = & \text{mutable} \\ \_ \triangleright \text{readonly} & = & \text{readonly} \\ q \triangleright \text{polyread} & = & q \end{array}$$

where the underscore denotes a "don't care" value. For example, the type of field access y.f, where y is readonly and f is polyread, is the not declared type of f. Instead, it is the adapted type from the viewpoint of y: $q_y \triangleright q_f =$ readonly $\triangleright$ polyread = readonly.

ReIm overcomes the limitation of context insensitivity. If we type rd at line 9 as readonly, the return type and this of getDate, as well as field date as polyread, we have:

```
polyread Date date;
polyread Date getDate(polyread DateCell this) {
  return this.date;
}
```

At line 9, getDate is called in readonly context (rd is not modified). We adapt the type of this of getDate from the viewpoint of rd:

$$q_{\text{rd}} \triangleright q_{\text{this}} = \text{readonly} \triangleright \text{polyread} = \text{readonly}$$

As a result this of cellGetHours can be typed as readonly. ReIm overcomes the limitations of context insensitivity by using polyread qualifier and viewpoint adaptation. This improves the precision of purity inference — cellGetHours would be inferred as pure by our method purity inference algorithm.

Immutability Inference.

We implement an efficient algorithm to infer reference immutability types. Given a set of classes (a whole-program or a library), our inference algorithm decides mutability for each reference in the set of classes, including local variables, fields, return values, and formal parameters. The key idea of the algorithm is to map each reference to a set of all immutability qualifiers, i.e. {mutable, polyread, readonly}, and iteratively remove infeasible qualifiers for each reference according to ReIm's typing rules defined in [8]. The detailed inference algorithm is described in [8].

```
1  class List {
2    Node head;
3    int len;
4    void add(Node n) {
5      n.next = this.head;
6      this.head = n;
7      this.len++;
8    }
9    void reset() {
10     this.head = null;
11     this.len = 0;
12   }
13   int size() {
14     return this.len;
15   }
16 }
```

Figure 2: A simple linked list

2.2 Purity Inference

ReImInfer adopts the definition of purity given by Sălcianu and Rinard [12]: a method is *pure* if it does not mutate any object that exists in *prestates*. Thus, a method is pure if (1) it does not mutate prestates reachable through parameters, and (2) it does not mutate prestates reachable through static fields. The definition allows a pure method to create and mutate local objects, as well as return a newly constructed object as a result.

For a method that does not access static fields, the prestates it can reach are the objects reachable from the actual arguments and the method receiver. Therefore, if any of the formal parameters of m or implicit parameter this, is inferred as mutable by reference immutability inference, m is impure. Otherwise, i.e., if none of the parameters is inferred as mutable, m is pure.

Consider the implementation of List in Figure 2. For method add, reference immutability inference infers that both n and this are mutable, i.e. the objects referred to by them are mutated in add. When there is a method invocation lst.add(node), we know that the prestates referred to by the actual argument node and the receiver lst are mutated. As a result, we can infer that method add is impure. We can also infer that method reset is impure because the implicit parameter this is inferred as mutable by reference immutability inference. Method size is inferred as pure because its implicit parameter this is inferred as readonly and it has no other formal parameters.

ReImInfer also considers prestates that are from static fields. The technical details can be found in [8].

3. COMPARISON WITH OTHER TOOLS

We have evaluated ReImInfer on 13 Java benchmarks, including 4 whole-program applications and 9 Java libraries, comprising 766,053 lines of code in total. To evaluate analysis precision, we compare with JPPA by Sălcianu and Rinard [12] and JPure by Pearce [11]. ReImInfer scales well to large programs and shows better precision compared to JPPA and JPure in almost all cases. Furthermore, ReImInfer, which is based on the stable and well-maintained Checker Framework [10], is more robust than JPPA and JPure, both of which are based on custom compilers. These results suggest that ReImInfer can be useful in practice as a wide variety of clients require purity analysis. The detailed comparison can be found in [8].

4. CONCLUSION

Method purity inference has a number of applications. We have presented ReImInfer, a modular and compositional tool for method purity inference for Java. We have evaluated ReImInfer on 13 large Java programs and Java libraries and shown that ReImInfer achieves both scalability and precision.

5. REFERENCES

[1] JOlden benchmark suite. `http://osl-www.cs.umass.edu/DaCapo/benchmarks.html`.

[2] HTML Parser. `http://htmlparser.sourceforge.net/`, 2006.

[3] L. R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(April):1031–1045, 1997.

[4] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[5] C. Flanagan, S. Freund, and S. Qadeer. Exploiting purity for atomicity. *IEEE Transactions on Software Engineering*, 31(4):275–291, Apr. 2005.

[6] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *PLDI*, pages 311–320, 2000.

[7] W. Huang, W. Dietl, A. Milanova, and M. D. Ernst. Inference and Checking of Object Ownership. In *ECOOP*, pages 181–206, 2012.

[8] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *OOPSLA*, 2012.

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

[10] M. M. Papi, M. Ali, T. L. Correa Jr, J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212, 2008.

[11] D. Pearce. JPure: a modular purity system for Java. In *CC*, pages 104–123, 2011.

[12] A. Sălcianu and M. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, pages 199–215, 2005.

[13] O. Tkachuk and M. B. Dwyer. Adapting side effects analysis for modular program model checking. In *ESEC/FSE*, volume 28, pages 188–197, Sept. 2003.

[14] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, 2005.

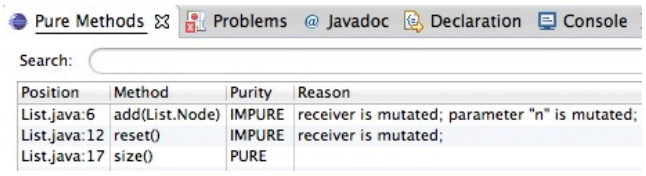
APPENDIX

We demonstrate ReImInfer in three usage scenarios: (1) whole program (i.e., a program which has main method), (2) an incomplete program (i.e., a library, or any set of classes without a main method), and (3) composition of a pre-analyzed library with user code. In addition, we also build an Eclipse plugin for ReImInfer. The source and binary can be downloaded from `http://code.google.com/p/type-inference/`, where the installation instructions are also available.

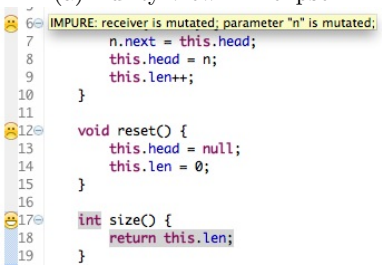
A. WHOLE PROGRAM

ReImInfer can infer method purity on whole programs. The following steps show how to use ReImInfer to infer method purity for the BH program in the Jolden benchmark suite [1].

1. Make sure ReImInfer is installed correctly by following the instructions.



(a) Purity view in Eclipse



(b) Purity markers in Eclipse

Figure 3: Using ReImInfer in Eclipse

2. Change the console into the BH directory:
   cd ./benchmarks/jolden/bh
3. Run the javai-reim command on the source:
   javai-reim BH.java Body.java Cell.java MathVector.java Node.java Tree.java
4. ReImInfer outputs debug information and stores the inference results into the following three files:
   - result.jaif The reference immutability results for all references except local variables in JAIF format.
   - reim-result.csv The references immutability results for all references in <filename> <linenum> format. For example, bh/Body.java 10 vel @Mutable MathVector denotes that the vel reference at line 10 in file bh/Body.java is mutable.
   - pure-methods.csv The pure methods inferred by ReImInfer. The pure methods are in the format of <absolute-class>.<method-signature>. For example, bh.Body.subindex(bh.Tree,int) denotes the method subindex(bh.Tree,int) in class bh.Body is pure. If a method does not appear in this file, that means it is inferred as impure.

JPPA works on this whole program. There are differences between ReImInfer's result and JPPA's result due to limitations in JPPA as discussed in [8]. However, JPure crashes on this simple program, throwing an exception that we were not able to correct.

B. INCOMPLETE PROGRAM

ReImInfer can also infer method purity on incomplete programs (i.e., libraries without a main method). We demonstrate how to infer method purity for htmlparser, a Java library for parsing HTML [2].

1. Make sure ReImInfer is installed correctly by following the instructions.
2. Change the console into the htmlparser directory:
   cd ./benchmarks/htmlparser
3. Run javai-reim command on the java source:
   find . -name '*.java' | xargs javai-reim

   The above commands first find all Java source files in the current directory and then pass these file names as input to javai-reim.
4. ReImInfer outputs the inference result into three files as described above.

JPPA does not work on this incomplete program because it is a whole-program analysis and requires a main method. JPure crashes on htmlparser; the underlying compiler issues an error which we were unable to correct.

C. COMPOSING A LIBRARY WITH USER CODE

In the case when the source code for library methods is unavailable, ReImInfer conservatively assumes that all library methods are impure. ReImInfer provides an option, -AlibPureMethods which allows to incorporate inference result for libraries. Suppose we wanted to infer method purity for BH in Jolden by incorporating the inference result of Java libraries. The following steps show how to compose libraries with user code.

1. Make sure ReImInfer is installed correctly by following the instructions.
2. Download the JDK source code from http://download.java.net/openjdk/jdk6/.
3. Unzip the package.
4. Run javai-reim on java.lang package and java.util package (or any other JDK packages we want to include):
   find ./java/lang ./java/lang -name '*.java' | xargs javai-reim
5. Now pure-methods.csv contains all pure methods in java.lang and java.util packages. Rename it to jdk-pure-methods.csv:
   mv pure-methods.csv jdk-pure-methods.csv
6. Change the console into the BH directory:
   cd ./benchmarks/jolden/bh
7. Run javai-reim on the source with the -AlibPureMethods option:
   javai-reim -AlibPureMethods ../../../jdk-pure-methods.csv BH.java Body.java Cell.java MathVector.java Node.java Tree.java
8. ReImInfer outputs the inference results into three files as discussed above.

The inference result for BH is more precise compared with the result from Section A, because it incorporates the purity result for the java.lang and java.util packages. In the case that an impure user method overrides a pure library method, ReImInfer issues a warning that the user code does not comply to the subtyping constraints that are expected by the library. JPPA and JPure cannot compose library and user code.

D. ECLIPSE PLUGIN

We can use ReImInfer for Java projects in Eclipse by using a plugin. The plugin would generate a view showing the purity of all methods for the selected Java project, as well as insert markers in the source code to indicate the purity of each method. Figure 3 shows two screenshots of the plugin.