Revealing Purity and Side Effects on Functions for Reusing Java Libraries

Jiachen Yang[1], Keisuke Hotta[1], Yoshiki Higo[1] and Shinji Kusumoto[1]

[1]Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka, 565-0871, Japan
{jc-yang,k-hotta,higo,kusumoto}@ist.osaka-u.ac.jp

Abstract. Reuse of software components requires the comprehension of the behavior and possible side effects among APIs of program components. Meanwhile, identifying problematic usage of these components is difficult with conventional static analysis. Purity and side effects are important properties of methods that often are neglected by the documentations of the object oriented languages such as Java. In this paper, we studied these properties by using a static analysis technique to automatically infer the state dependencies for the return value and side effects of methods. As a result, the effect information reveals purity of methods as well as well-defined state interactions between objects. We have implemented the analyzer targeting Java bytecode and tested it on some open source Java software libraries with different scale and characteristic. From our experimental results, we found that 24–44% of the methods in the evaluated open source Java libraries are pure, which indicates that a large percentage of the methods are suitable for high level refactoring. Our study can help programmers to understand and reuse these libraries.

Keywords: static analysis · pure function · state boundary · state dependency · object-oriented · design by contract

1 Introduction

It is difficult for programmers to reuse software components without fully understanding their behavior. The documentation and naming of these components usually focuses on *intent*, i.e., what these functions are required to do, but fails to illustrate their *side effects*, i.e., how these functions accomplish their task [4]. For instance, it is rare for API function[1] signatures or documentation to include information about what global and object states will be modified during an invocation. It is hard to reuse the modularized components, because of the possible side effects in API libraries. For instance, it is usually unclear for programmers

[1] We interchange the term *function* with the term *method* throughout this paper referring to the same thing. We use *function* to refer the ideas that originate from the functional paradigm, and *method* to refer the ideas that originate from object-oriented paradigm such as Java.

whether it is safe to call the APIs across multiple threads. In addition, undocumented API side effects may be changed during software maintenance, making debugging even more challenging in the future [16].

By understanding of side effects in the software libraries, programmers can perform high level refactoring on the source code that is using the functional part of the libraries. For instance, the return value of math functions such as `sin` will be the same result if the same parameter is passed, therefore the result can be cached if the same calculation is performed more than once. Moreover, the calculation without side effects are good candidates for parallelization [9]. However, the purity information is usually missing in external libraries, therefore programmers would risk introducing bugs with such refactorings, for example, caching the result of a function which depends on the mutable internal state.

In this paper, we present an approach to infer a function's purity from byte code for later use. Programmers can use effect information to understand a function's side effects in order to reuse it. For example, the approach can help to decide whether it is safe to cache or parallel a time-consuming calculation. The contributions of this research include:

- An approach to automatically infer purity and side effects,
- A concrete implementation for Java bytecode,
- Experiments on well-known open source software libraries with different scale and characteristic.

In our experiments,we found that 24–44% of the methods in the evaluated open source Java libraries are pure. Also, we observed methods that should be pure in theory but not in the implementation, and revealed tricks or potential bugs in the implementation by a case study of our approach.

We achieved the same percentage of pure functions with the existing study without a manually created white-list, and we revealed which side effects these functions were generating which would not found in the existing studies. We focused on revealing these side effect information on real world software libraries to be used by the programmers and tools.

2 Related Work

The idea of verifiable imperative programs has been used in the Euclid language [10] and its descendants [6] since the 1970s. However, modern OO languages such as Java and C# have not implemented these ideas. The proposed research checks side effects and purity in legacy source codes written in these modern OO languages.

Many previous efforts on combining pure functional style into an OO paradigm concentrate on introducing immutable restrictions on existing type systems, as in functional programming languages. Tschantz, et al. [19] 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 [8]

proposed an analyzer called Javarifier to automatically infer reference immutability in Javari syntax. Huang, et al. [7] proposed a much simpler but more restricted design, called ReIm and ReImInfer, as they only modified the type system of Java by adding three extra qualifiers in the type declaration, and their implementation is more unified in comparing Javari with Javarifier. A similar approach has been taken by extending the syntax of C# in [5]. 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 [18], which uses an inter-procedural pointer analysis [17] 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. We have compared the purity result of our approach with their study using the same benchmark in Section 5.2. JPure [14] 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 informations 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 informations. Also neither of these two studies further classify the pure functions into *Stateless* and *Stateful* as we do. Further, we provide a heuristic approach to detect cache semantics in member fields, thus eliminating the need of a manually provided white-list.

Mettler, et al. [13] take a different approach. Instead of extending the syntax of an existing language, they created a subset of Java called Joe-E. As one application of Joe-E, they proposed [3] to verify the purity of functions by only permitting immutable objects in the function signature. Kjolstad, et al. [9] proposed a technique to transform a mutable class into an immutable one. They utilized an escaping and entering analysis similar to [18]. These two studies are similar to each other as they completely eliminate the mutable states from target source codes, which is not always acceptable in general programming scenarios, thus limits their application. Comparing to these two studies, our technique can be performed on the real world software libraries, even without the source code. Therefore we are more suitable for comprehension the legacy code bases.

3 Automatic Inference of Purity and Side Effects

In this section, we first define the concepts of purity and side effects on the functions in Java. Then we present our approach to automatically infer the purity

and side effects. Lastly, we will describe how to utilize our approach during the reusing of software components.

3.1 Stateless & Stateful Purity of Functions

The notion of purity on functions does not match well with other OO paradigm concepts. In OO languages such as Java, 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 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 methods called on them simply query these internal states. We would like to distinguish these state-querying methods from those methods that modify the states. Through our research we have observed that Java libraries can contain around 24-44% of functional code that does not modify the program's state.

Based on the above observation, we defined 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 [15]. Meanwhile, many existing studies such as [11, 14, 18] share the same purity definition with us. To illustrate the difference of two definitions, we divide 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.*

All other non-*pure* functions generate side effects. Although the notion of a stateful pure function may seem like a contradiction, we can view the state of field members as extra arguments, so that they can be converted into the mathematical form of a pure function. An example of a stateful pure function is `equals` method in Java, which compares the value equality of two objects. Although they depend on an object's state, well-formed `equals` methods do not change the state.

3.2 Lexical State Accessors and Side Effects

The main purpose of this research is to reveal the effects of functions. Therefore we need to define 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.*

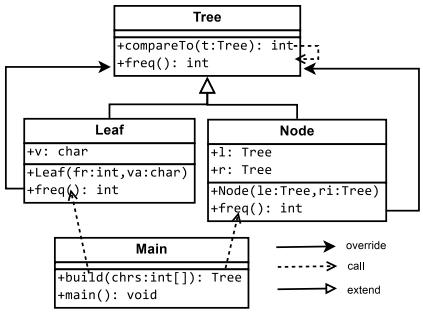

Fig. 1. Class Diagram with Call Graph

The effects of a functions are all the side effects plus the return value. According to the *single response principle* in [12], 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.*

All possible modifications to the state of a program are 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 directly or transitively calling system routines to perform I/O operations.

3.3 Call Graph and Data Analysis

The analyzer identifies method 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 methods in a class hierarchy. The call graph records the invocation instructions inside the method body, which points to another method defined in the class diagram. An example class diagram is shown in Fig. 1.

Our analyzer traverses all of the methods in the class diagram, inferring possible effects including side effects. We capture only the dependencies of lexical state accessors, 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 3.2, which are the *static field*s (shortened as S) of a class, the *member field*s (shortened as F) of an object, and the *argument*s (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 interpret the code, follow the instructions in the given function, and apply 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 a *if* or *while* statement. Next, the analyzer interprets 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 instruc-

Table 1. Transfer Functions for Values and Instructions

Type of Instuctions	Code Pattern	Reference Dependency	Value Dependency
new object	**new** τ	$\emptyset$	$\emptyset$
parameter	x	$\{x\}$	$\{x\}$
local variable	y	$\emptyset$	$\emptyset$
member field	`this.field`	{field}	{field}
static field	`Class.field`	{field}	{field}
object field	V`.field`	V_{rd}	V_{vd}
unary operation	$op\,V$	$\emptyset$	V
binary operation	$V_1\ op\ V_2$	$\emptyset$	$V_{1vd} \cup V_{2vd}$
array access	$V_1[V_2]$	V_{1rd}	$V_{1vd} \cup V_{2vd}$
type cast	$(\tau)\,V$	V_{rd}	V_{vd}
assignment	V_1 `=` V_2	V_{1rd}	V_{2vd}
return value	`return` V	$\emptyset$	$\emptyset$
merge		$V_{1rd} \cup V_{2rd}$	$V_{1vd} \cup V_{2vd}$

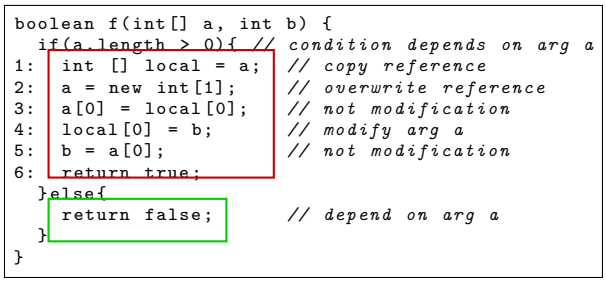

Fig. 2. Example of Data and Control Analysis

tion, and the output is the new V after the execution. Besides the reference and value dependency 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 3.4.

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.

An example of the interpretation stage is represented in Fig. 2. At the beginning of interpretation, the reference dependency of `a` is assigned as argument `a`, and the value dependency of `a` and `b` are assigned as corresponding argument names. There are two blocks in this code colorized as red (above) and green (below), which are associated with the branch condition `a.length > 0`. Since the value dependency of this condition is argument `a`, both two blocks depend on the state of `a`. Then, during the interpretation of the red block:

1. The reference of `a` is copied into `local`, which implies that the reference dependency of `local` is {`a`}
2. The reference dependency of `a` is now $\emptyset$
3. A *modification behavior* is performed on the reference dependency of `a`, which is $\emptyset$, and thus has no side effects.
4. A *modification behavior* is performed on the reference dependency of `local`, with a value dependency of {`b`}. An `@Argument` effect on `a` is generated with a data dependency on `b` and a control flow dependency on `a`.
5. A *modification behavior* is performed on $\emptyset$.
6. A *return statement* generates a `Depend` effect with a value dependency of $\emptyset$ and an value dependency of the constant `true`, which is then merged with the control dependency on `a`.

The analysis on the green block generates the same `Depend` effect, and these two `Depend` effect are then merged.

3.4 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 data flow across the invocation boundaries. Fortunately, this cross-function analysis is possible with the generated effect information on the callee, so that we do not need to examine the codes of the caller and callee at the same time.

There are two kinds of invocation instructions in Java: static and dynamic dispatch. Dynamic dispatch is used to call virtual methods, and static dispatch is used to call non-virtual methods and special cases such as calling overridden methods defined in a super class.

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 method, 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 occurred inside the caller method.

The V_{obj} is the object that owns the method, which could be `this`, `ClassName` or a certain dynamically calculated value during the interpretation. Static member methods 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 information 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 3.2 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}.

With the effect information 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 modifications 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.

3.5 Iteration to a Fix-point of Class Diagram

A function's effects depend on the effects 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 effects 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.

We also differentiate two sets of effects: *static effects* and *dynamic effects*, because we differentiate between static and dynamic dispatch invocations.

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

1. Merges the *static effects* with the result of the data analysis on this method.
2. Sets the *dynamic effects* to be the merge of *static effects* and all *dynamic effects* of the overridden methods.
3. Clears the flag on this method.
4. If the effects have changed since last analysis, marks the flags of all methods that depend on this method.

We continue iterating until none of the methods in the class diagram are marked, which means a fix-point of the analysis is reached. Note that during the execution of this algorithm, the size of both *static effects* and *dynamic effects* only increases and never decreases. There is an upper limit on the size, which is the sum of numbers of all possible modifications to the fields and arguments in the program. With the monotone increasing property and the upper bound of the algorithm, we can guarantee that it will halt.

3.6 Applications in Reusing Software Components

We have described how our analyzer infer the effect information. Next, we will briefly introduce how to use our analyzer from a programmer's point of view.

Suppose a programmer is facing a reusable software component, either in distributed binary form or in source code form, and the programmer would like to know whether it is safe to reuse this component in his new code. The programmer can apply our analyzer on the candidate component, together with all its dependent libraries, to obtain a list of side effects on each functions from the component. The programmer can then decide whether it is safe to reuse the component based on the side effects.

For example, if the programmer is writing a multi-threaded program, and the candidate component is accessing some global states, then the programmer may need to introduce a thread lock to synchronize the accesses to these global states. As another example, if the candidate function is a pure function reported by our analyzer, then it is usually safe to reuse this function in the new source code without introducing hidden data dependency.

Moreover, the output of our analyzer can help the debugging and understanding of the behavior of software components. It is reported [1] that some bug will appear only if the programmer execute the unit test separately. Understanding the side effects could reveal these bugs even before executing the test cases.

4 Implementation Details

We discuss some of the implementation details of our analyzer in this section. We chose Java bytecode defined by the Java Runtime Environment (shortened

```
class String{
  /** Cache the hash code for the string */
  private int hash; // Default to 0
  ...
  public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
      char val[] = value;
      for (int i = 0; i < value.length; i++) {
        h = 31 * h + val[i];
      }
      hash = h;
    }
    return h;
  }
}
```

Fig. 3. Example of Cache Semantic in `java.lang.String`

as JRE) version 6 as our target language, and implemented the described analyzer based on the widely used ASM library [2]. There are several advantages in targeting an intermediate language rather than source code. First, the analyzer is syntax neutral, so we can automatically analyze all languages targeting the Java Virtual Machine. Second, the analyzer can be applied on binary libraries without source code. Finally, the type safety is assured by the JRE's compiler and bytecode verifier.

4.1 Detection of Cache Semantics

Although the described analysis works well for identifying modification behaviors in theory, we find a difficulty to apply it in practice when member fields are used solely to cache the calculation results. We refer to the member fields that are used to cache the calculation results as having cache semantics. We found that the implementation of `HashMap.equals` modifies its member field `HashMap.entrySet`, and the implementation of `String.hashCode` caches the result in its member field `String.hash`, as shown in Fig. 3. By our definition, these methods change the state of internal member fields, and thus are no longer pure functions. As a result of these two methods not being pure, callers of these methods were also marked as generating side effects.

This caching semantic is not only found by us, but also described in previous literatures such as [18]. A widely accepted solution to this problem was to accept a white-list of functions from the user (called *special* methods in [18]), indicating that they are proven to be pure by the user manually. For the reason that the selection of the white-list will impose great impact on the precision of the analyzing result, and they involve human judgments, we do not consider this as an ideal solution.

To preciously and automatically analyze this kind of methods that have caching semantics, we extend our analyzer to detect the cache semantics using a heuristic approach. More precisely, we consider a member field of a class having the cache semantic if all the following preconditions are true:

P1 The field is assigned either by a constant value, or in only one member function.
P2 The non-constant assignment on the field occurs within a branch block.
P3 The right-hand value of the non-constant assignment is only depended on other fields.
P4 The branch condition of the block checks that the value of the modified member field is a constant value.

We consider the following values as constant values: constant literals, null pointers and values of `static final` member fields that have a primitive type. The assignment with a constant value is considered as re-initializing the state of the cache field. The checking with a constant value is considered as checking the initialized state. In either cases, the value of the field is determined by other fields, therefore, it cannot be used to store a mutable state of the object.

In the example of `String`, the member field `hash` is assigned by `hashCode` with a calculation result and by its constructor with a constant value, therefore P1 is true. The assignment to `hash` occurs in a `if` condition block, therefore P2 is true. The value of the assignment is depend on the member field `value`, therefore P3 is true. Lastly in the condition block, the value of `hash` is assured to be zero by the condition check `h==0`, therefore P4 are true. The member field `hash` meets all the preconditions, therefore it is considered to be a caching field by our analyzer.

The modification behavior on the detected caching fields are suppressed from the effects, and the return value dependencies on these caching fields are ignored.

5 Experiments

We implemented our analyzer with name *purano*[2], and evaluated it on real world software components in terms of accuracy, performance, and the distribution of different kinds of effects in different scale of software components. During the experimentation, we expected to answer the following research questions:
RQ1 What is the distribution of pure and side effect methods in the software libraries?
RQ2 How is the accuracy of our analysis comparing with an existing study? How is the heuristic approach in the detection of cache semantic compared to the white-list approach?
RQ3 How to utilize the revealed information during reusing the software components?

Firstly, we will answer the 2 research questions by experiments. Then we will demonstrate how would our study help programmers in RQ3 as a case study.

5.1 R1: Distribution of Effects

To show the distribution of purity and side effects of the methods in real world software libraries, we experimented on 4 target software projects, listed in Ta-

[2] We have published *purano* at https://github.com/farseerfc/purano.

ble 2. These experiments were executed on an octo-core Xeon E5520 with a 2GB heap size limitation. *purano* is the implementation of the analyzer of this paper, which includes a modified version of the ASM library. Both *htmlparser*, *tomcat* and *argouml* are well-known open source Java projects, and we used their latest stable binary distributions. Note that all of these software projects were analyzed together with the JRE standard libraries, because the analyzer need the purity and side effect information for all functions being called including the ones in the libraries. This lead to the much greater number of analyzed classes than the number of the target classes. According to the Javadoc for JRE 7, there are 3,793 public classes altogether, and more private ones in the JRE library. The analysis time of *argouml* was around 4 minutes, which is reasonable for large scale software. The number of analysis passes ranged from 16 to 22, which was depended on the longest invocation and overriding chain in all analyzed methods. Based on the analysis times in Table 2, we can conclude 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 output, we find that around 24%–44% of the methods 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. The *argouml* project contains many non-pure graphical code percentage and the htmlparser project have more pure functional code percentage.

5.2 R2: Comparison with an Existing Approach

While there are none of existing studies to identify the side effect informations within our knowledge, there are studies that only infer the purity of the functions based on different approaches. Therefore, we compare our purity result with one of the existing studies to examine the accuracy of our analysis. We ran our tool

Table 2. Experiment Target and Analysis Performance

Software	Analyzed Classes	Target Classes	Target Functions	Time (sec.)	# Passes
purano	2,942	253	2,372	148	16
htmlparser	5,795	156	1,645	112	17
tomcat	7,673	772	8,824	186	18
argouml	11,608	2,545	20,167	233	22

Table 3. Percentage of Effects

Software	Pure Functions		Side Effects	Modifying		
	Stateless	Stateful		Member	Static	Arg.
purano	382 (16.1%)	192 (8.0%)	1,798 (75.9%)	1,548	1,087	485
htmlparser	363 (22.1%)	358 (21.8%)	924 (56.2%)	679	462	143
tomcat	1,260 (14.3%)	1,861 (21.1%)	5,703 (64.6%)	4,346	3,990	1,288
argouml	5,019 (24.9%)	1,744 (8.6%)	13,404 (66.5%)	7,057	11,849	3,255

against the JOlden benchmark used in [18]. The result from the benchmark is shown in Table 4, comparing with the result from their study. Also we run our analyzer in two different configurations. One configuration is using a white-list which is similar to the configuration of [18], with the detection of cache semantic disabled. Another configuration is using the detection of cache semantics.

Their approach relies on a whole program analysis starting from a `main` entry point, and thus they covered fewer functions than our tool. They chose a set of functions for the white-list by viewing all the source code manually in advance, a time-consuming task in practice, while our approach automatically identifying the cache semantics. We were unable to compare precision and recall due to challenges in executing their tool in our environment. Therefore we compared with their result from the published literature [18]. As we can see from the result table, we achieved a similar result on the number of pure functions. In addition to the number of pure functions shown in the result, we identified all the side effects and the type of purity, which is the main purpose of our study and cannot be found in their result.

Comparing our result with different configurations, we can see that the detection of cache semantics result to a slightly lower pure percentage than the white-list approach. This is excepted, as the heuristic detection approach cannot find all the fields that are used for caching purpose without increasing the false positive rate. For example, we cannot detect the cached result within an entry of a hashmap instead of a single field. We consider the heuristic detection approach is more applicable for the existing software libraries because the programmers usually do not have a clue of which API functions are the libraries using and whether they are pure functions. Revealing this information is the main purpose of the purity analysis in the first place. An automatic technique like our approach will break the chicken or the egg dilemma and enable the purity analysis to be adopt in practice.

Table 4. Comparison on JOlden Benchmark. Function numbers are different because our approach analyzes all functions while Sălcianu's approach analyzes only the functions invoked transitively from the `main` entry point.

Application	Our (White-list)				Our (Cache Semantic)			Sălcianu's	
	Total	Stateless	Stateful	Pure	Stateless	Stateful	Pure	Total	Pure
BH	73	14	17	31	13	13	26	59	28
BiSort	15	6	0	6	5	0	5	13	5
Em3d	23	7	3	10	5	2	7	20	8
Health	29	8	1	9	8	0	8	27	13
MST	36	8	11	19	5	9	14	31	17
Perimeter	50	28	11	39	28	9	37	37	33
Power	32	2	4	6	2	4	6	29	9
TSP	16	5	1	6	4	1	5	14	5
TreeAdd	12	3	1	4	2	1	3	5	2
Voronoi	73	11	31	42	12	33	45	70	50

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

Fig. 4. A Potential Problem in `FilePermission`

5.3 RQ3 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 tool. We examined the inferred effects on two methods, namely `equals` and `hashCode`. These two methods 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 methods reflect their value equalities, and hence these return values should depend on the state of the objects. Therefore, we expect these methods to be *stateful* pure functions if they contain member fields. The purity types of these two methods are listed in Table 5.

To further understand the result, firstly we focused on the existence of *stateless* pure functions in Table 5 by manually examining their source code. Most of these methods are defined in interfaces or abstract classes. There were also 2 `equals` and 6 `hashCode` methods defined in the classes that do not have member fields. There were 9 `equals` that compares referential identities defined in classes, while these classes have member fields that are not accessed in the `equals`. 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`, 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

Table 5. Purity of `equals` and `hashCode`

Software		All	Pure Functions		Side Effects
			Stateless	Stateful	
purano	`equals`	518	19 (3.7%)	165 (31.9%)	334 (64.5%)
	`hashCode`	499	14 (2.8%)	176 (35.3%)	306 (61.9%)
htmlparser	`equals`	359	14 (3.9%)	141 (39.3%)	204 (56.8%)
	`hashCode`	355	10 (2.8%)	147 (41.4%)	198 (55.8%)
tomcat	`equals`	477	65 (13.6%)	282 (59.1%)	132 (27.7%)
	`hashCode`	473	52 (11.0%)	245 (51.8%)	176 (37.2%)
argouml	`equals`	426	55 (12.9%)	219 (51.4%)	152 (35.7%)
	`hashCode`	416	55 (12.2%)	214 (51.4%)	162 (28.9%)

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` that return a constant, whereas their corresponding `equals` compared the states of member fields. An example is shown in Fig. 4, 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.

Next we examined the functions in Table 5 that generate side effects. 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 similiar to the caching technique but will change the observable state of the object.

All of these implementation details revealed by our analyzer require special care in both development and maintenance of the software. We hope our research can aid the development in the situations like we have studied in this case study.

6 Future Work and Conclusions

The current implementation of our analyzer works on Java bytecode rather than source code. Besides all the advantages described, this decision is also made to ease the 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 the generated effect information, by programmers as well as by analysis tools. Currently we output the effect information as annotations. 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. We will apply this approach to more software projects for further evaluation.

To conclude, in this paper we presented a study on the purity and side effects of the functions in Java, helping programmers to reuse the software libraries. We proposed a technique to automatically infer the purity and side effect informations from Java bytecode. We implemented and experimented the proposed analyzer on real world Java software libraries, and found that around 24%–44% of all the methods of a Java libraries are made of pure functions. We compared the accuracy of distribution of pure functions with an existing study. Also, we demonstrated how programmers will use our technique 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. Bell, J.S., Kaiser, G.E.: Unit test virtualization with vmvm (2013)
2. Bruneton, E., Lenglet, R., Coupaye, T.: Asm: a code manipulation tool to implement adaptable systems. Adaptable and extensible component systems 30 (2002)
3. Finifter, M., Mettler, A., Sastry, N., Wagner, D.: Verifiable functional purity in java. In: Proc. of the 15th ACM conference on Computer and communications security. pp. 161–174. ACM (2008)
4. Goetz, B.: Java theory and practice: I have to document that? http://www.ibm.com/developerworks/java/library/j-jtp0821/index.html (2002)
5. Gordon, C., Parkinson, M., Parsons, J., Bromfield, A., Duffy, J.: Uniqueness and reference immutability for safe parallelism (2012)
6. Holt, R.C., Cordy, J.R.: The turing programming language. Communications of the ACM 31(12), 1410–1423 (1988)
7. Huang, W., Milanova, A., Ernst, W.: Reim & reiminfer: Checking and inference of reference immutability and method purity. OOPSLA (2012)
8. J., Q.: Javarifier: Inference of reference immutability in Java. Ph.D. thesis, Massachusetts Institute of Technology (2008)
9. Kjolstad, F., Dig, D., Acevedo, G., Snir, M.: Transformation for class immutability. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 61–70. ICSE '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1985793.1985803
10. Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., Popek, G.J.: Report on the programming language euclid. ACM Sigplan Notices 12(2), 1–79 (1977)
11. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of jml. Tech. rep., Technical Report 96-06p, Iowa State University (2001)
12. Martin, R.C.: Clean code: a handbook of agile software craftsmanship. Prentice Hall (2008)
13. Mettler, A., Wagner, D., Close, T.: Joe-e: A security-oriented subset of java. In: Network and Distributed Systems Symposium. Internet Society (2010)
14. Pearce, D.J.: Jpure: a modular purity system for java. In: Compiler Construction. pp. 104–123. Springer (2011)
15. Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 71–84. ACM (1993)
16. Raymond, C.: The importance of error code backwards compatibility. http://blogs.msdn.com/b/oldnewthing/archive/2005/01/18/355177.aspx (2005)
17. Sălcianu, A.: Pointer analysis and its applications for Java programs. Ph.D. thesis, Citeseer (2001)
18. Sălcianu, A., Rinard, M.: Purity and side effect analysis for java programs. In: Verification, Model Checking, and Abstract Interpretation. pp. 199–215. Springer (2005)
19. Tschantz, M., Ernst, M.: Javari: Adding reference immutability to Java, vol. 40. ACM (2005)