Towards Purity-Guided Refactoring in Java

Jiachen Yang, Keisuke Hotta, Yoshiki Higo, Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka, 565-0871, Japan
Email: {jc-yang,k-hotta,higo,kusumoto}@ist.osaka-u.ac.jp

***Abstract*—Refactoring source code requires preserving a certain level of semantic behaviors, which are difficult to be checked by IDEs. Therefore, IDEs generally check syntactic preconditions instead before applying refactoring, which are often too restrictive than checking semantic behaviors. On the other hand, there are pure functions in the source code that do not have observable side-effects, of which semantic behaviors are more easily to be checked. In this research, we propose purity-guided refactoring, which applies high-level refactoring such as memoization on pure functions that can be detected statically. By combining our purity analyzing tool *purano* with refactoring, we can ensure the preservation of semantic behaviors on these detected pure functions, which is impossible through previous refactoring operations provided by IDEs. As a case study of our approach, we applied memorization refactoring on several open-source software in Java. We observed improvements of the performance and preservation of semantics by profiling their bundled test cases.**

***Index Terms*—refactoring, static analysis, pure function, side effect analysis**

I. Introduction

Source code refactoring is generally defined as a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior [1]. Refactoring is one of many important tasks in software development and maintenance. Integrated Development Environments (shortened as IDEs) generally provide supports for common refactoring operations, which improve maintainability, performance or both, during the software development process. Machine-aid refactoring is so important that it is promoted as one of the best practices in both the Extreme programming [2] and Agile Software development [3], so that both of them heavily depend on automatic tools to perform refactoring during the development process.

It is emphasized to preserve the external behavior while performing refactoring so that they can be safely conducted without being evaluated about the breaking changes. However, it is hard to reason about the semantic behavior of a code fragment automatically by refactoring tools, due to the lack of semantic information from the traditional static analysis. Therefore, refactoring tools provided by IDEs generally take more conservative approaches by checking the syntactic structure of a given code fragment only in the pre-conditions of the refactoring operations.

Determining the possible semantic behavior from only the syntactic structure of a given code fragment puts heavy limitations on the possible refactoring patterns. Therefore, the refactoring tools provided by IDEs are limited to low-level structural restructuring on the code fragment, such as *Rename Method* or *Extract Method*. Although there are more high-level refactoring patterns widely recognized and adopted by programmers, such as *Replace Loop with Collection Closure Method* [4], currently they are not provided automatically by tools or IDEs and are applied manually by programmers.

In the other hand, preservation of semantic behavior can be statically checked for a certain part of the source code, namely the code that use pure functions. Pure functions are the functions that do not have observable side effects during the execution. With the property to be side effect free, whether refactoring changed the semantic behavior of the code fragments that use pure functions can be easily checked by tools. Therefore, more refactoring patterns become available when the purity information is inferred from the source code. 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 [5].

In our previous research [6], we adopted the definition of pure functions from functional programming languages to methods in object-oriented languages. We observed a relatively a large part of the source code, namely 24–44% of all the defined methods in our evaluated open source Java libraries are pure. This result indicates that there might be great opportunities to apply refactoring on these pure functions.

In this research, we focus on refactoring these pure functions and propose a new category of high-level automatic refactoring patterns, called *purity-guided refactoring*. In the following part of this paper, we will discuss the purity-guided refactoring. As a case study, we applied a kind of the purity-guided refactoring, namely *Memoization* refactoring on several open-source libraries in Java. We observed the improvements of the performance and the preservation of semantics by profiling the bundled test cases of these libraries.

II. Related Works

Xu et al. [7] have been studied dynamic purity analysis and developed a memoization technique during online purity analysis inside JVM. Rito et al. [8] proposed a memoization technique by using software transactional memory to find altered values during execution. These two studies shared a large part of the background with our study. However, we focused on static analysis and aim to source code level refactoring.

$$
\begin{array}{rcl}
\alpha & ::= & \overline{\alpha_r(dp)}\,\overline{\alpha_m(dp, from, target)} \\
\alpha_r & ::= & \texttt{@Depend} \mid \texttt{@Expose} \\
\alpha_m & ::= & \texttt{@Field} \mid \texttt{@Static} \mid \texttt{@Argument} \\
dp & ::= & [\texttt{dependFields} = \{s[,s]*\},] \\
 & & [\texttt{dependStatic} = \{s[,s]*\},] \\
 & & [\texttt{dependThis} = \texttt{true},] \\
 & & [\texttt{dependArguments} = \{s[,s]*\}] \\
from & ::= & [\texttt{from} = \{s[,s]*\}] \\
target & ::= & \texttt{name} = s, \texttt{type} = t, \texttt{owner} = t \\
t & ::= & \tau\texttt{.class}
\end{array}
$$

Fig. 1. BNF syntax for proposed annotations. τ is a type name. s is a string.

There are studies to convert a procedure style program to a more pure functional style program. Mettler, et al. created a subset of Java called Joe-E [9]. As one application of Joe-E, they proposed a method to verify the purity of functions by only permitting immutable objects in the function signatures [10]. Kjolstad, et al. proposed a technique to transform a mutable class into an immutable one [5]. Applying the approaches of these studies will result in more pure functions in the source code, thus they can increase the refactoring candidates of this research.

III. Purity-Guided Refactoring

We define the purity-guided refactoring as a category of refactoring patterns that utilize the purity information on code fragments. The required purity information is not readily available from the source code for object-oriented languages such as Java, therefore we need to infer the purity information from source code and document the purity information elsewhere. To achieve this purpose, we introduce *effect annotations*.

A. Effect Annotations

We introduce a set of method annotations to indicate the effects that can arise during execution. For each method, several annotations can be prepended, each representing a value effect (i.e. returning a value) or a side effect (e.g. modifying a member field). We define the syntax of these annotations as Backus-Naur Form in Figure 1. The annotations express the effects such as direct or transitive modifications to locally accessible variables, with the possible data dependency between these effects and other variables from the function.

α_r **return value annotations.** The following two annotations capture the dependency of a return value:
`@Depend` specifies the dependency of a function's return value of the function.
`@Expose` specifies the exposed state of lexical variables returned from a function.

α_m **modification annotations.** The following three annotations capture the modifications to variables. Multiple annotations of the same type are possible when there are multiple modifications within the function.
`@Field` represents the modification to member fields.
`@Static` represents the modification to static fields.
`@Argument` represents the modification to arguments, which should be a reference type so that pass-by-value semantics can affect the state of real arguments.

```
public class PreciseDateTimeField ... {
  @Depend(dependFields="long PreciseDateTimeField.
      iUnitMillis")
  public long getUnitMillis(){ return iUnitMillis; }
  @Depend(dependArguments={"long instant"},
    dependFields={"int PreciseDateTimeField.iRange",
      "long PreciseDateTimeField.iUnitMillis"})
  public int get(long instant) {
    if (instant >= 0) {
      return (int)(instant/getUnitMillis())%iRange;
    } else {
      return iRange-1+(int)(((instant+1)/
          getUnitMillis())%iRange);
    }
  }
}
```

Fig. 2. An example of `@Depend` effect annotations by *purano*

These effect annotations are not directly available from the source code because values can be passed through local variables and we need to track the value dependencies. Moreover, these annotations reflect the flattened properties flattened from other functions that are called inside the target function. Requiring the programmers to write these annotations will be a huge burden in code maintenance, therefore we proposed an analysis tool to automatically infer these annotations, which will be described in the following subsection.

B. Inference of Purity Information

We use a static analysis tool called *purano* that has been developed during our previous research [6] to infer the previously described effect annotations. *Purano* analyzes the provided bytecode of given Java software, together with all the binary dependency libraries. The output of *purano* is a set of annotations that record the purity and side effect information of all the methods in the given Java software.

An example of the output result from *purano* can be found in Figure 2. From the figure, we can see that *purano* is capable of identifying the `PreciseDateTimeField.get` method as a pure function, because its return value depends on both the value of its argument and the state of two member fields `iRange` and `iUnitMillis`.

More importantly, *purano* collects the data dependencies transitively across the function invocation boundaries. As shown in the example, the dependency on member field `iUnitMillis` is passed from the member method `getUnitMillis`. Besides the demonstrated `@Depend` annotation in the example, *purano* will also generate side effect annotations described in the previous subsection if the function may generate observable side effects during the execution.

C. Purity Queries During Refactoring

With the purity information at hand, refactoring tools can query several semantic properties on a given code fragment that is not available in its syntactic structure. These semantic properties are useful during both pre-condition checking and code restructuring. To name a few, refactoring tools can ask the following queries:

Q1 **Is this function pure?** The tools can query whether the function has modification annotations.

Q2 **Are the objects of this class immutable?** The tools can query whether the class includes a member function except for constructors that generate `@Field` side effects or returns an `@Exposed` member field.

Q3 **Which methods modify the value of this field?** The tools can query on all member methods whether the method generate a side effect of type `@Field` on the field or `@Expose` the reference to the field.

This list of queries is rather ad-hoc and far from complete. We only listed the queries that we use during the case study that will be described in the following section.

IV. Case Study: *Memoization* Refactoring

Memoization refactoring (also referred as memorization) is a refactoring pattern that caches the result of a given function and returns the same result when the function gets called afterward with the same set of arguments. The implementation will store the result of the given function into a key-value storage with the set of the arguments as the key. The purpose of this refactoring is to reduce duplicated calculations and trade the CPU time with the memory.

The idea of the *Memoization* refactoring is simple and widely recognized. But it is often error-prone to apply the refactoring in practice, as the return value of the function may depend on internal states that may change between the function calls. Traditionally it requires the programmers to manually check these internal states and identify the locations that change them, which largely prevents the programmers from adopting this refactoring. Therefore, automatically tracking these internal states is essential for this refactoring.

In this research, we focus on applying the *Memoization* refactoring on Java member methods, although we believe the same approach could be easily extended to other object-oriented languages. We will describe the pre-conditions need to be checked, the general restructuring pattern, and some optional optimizations in detail. Further, we will discuss our considerations on the preservation of the purity.

A. Pre-conditions

The following pre-conditions should be hold for the target:

P1 **The function is pure.** This can be checked by **Q1**. Also, this implies that the return type can not be `void`.

P2 **The types of the arguments are immutable.** This can be checked by **Q2** for object types and ensured for primitive types in Java. Immutability for arguments is required to store them as keys in a key-value storage. Further, we rely on the correct implementation of `hashCode` in the classes of the arguments, but currently we do not check the correctness in our refactoring implementation.

P3 **The return value depends on no static fields, no public member fields nor publicly `@Exposed` member fields.** The refactoring tools can check this by retrieving the data dependencies from a `@Depend` annotation and querying **Q3** on the `dependFields`. This ensures that only member methods in the same class can affect the return value of the target function.

B. Refactoring Steps

Our refactoring tool applies the following steps on the target functions that meet all the pre-conditions:

1) Creates a private member field that served as the key-value storage in the class of the function. The key type of the field is a tuple of all the types of the arguments. The value type of the field is the type of the return value of the function. In the implementation, we use `HashMap` as the concrete key-value type and use the tuple types from a third-party library called *javatuples*.
2) At the entry point of the function:
   a) Defines a tuple variable that captures all arguments.
   b) Checks whether the tuple is already stored in the key-value storage. If it contains the tuple, return the corresponding value stored in it.
3) At every exit point of the function:
   a) Puts the calculated value into the key-value storage, with the captured tuple as the key.
   b) Returns the value.
4) For each member function that modifies one of the `dependFields` in `@Depend`:
   a) Empties the key-value storage right after the modification happens, so that the result will be re-calculated next time.

C. Optimizations

Depending on the target functions, optimizations can be applied to avoid unnecessary overheads and improve the maintainability of the generated source code.

O1 If there are no arguments, the key-value storage can be replaced with a simple member field whose type is the type of the return value. We use a `null` pointer to represent the empty state of the key-value storage.

O2 If there is only one argument, the `Unit` tuple used as the key can be replaced as the argument directly.

D. Preservation of the Purity on Functions

After refactoring, the target functions inevitably modify the key-value storage as a side effect. Therefore, the purity of the target function changed from pure to impure, by the definition of a pure function in related studies [10], [11]. However, the observable purity of the function is actually preserved, as still the return value of the function is determined by the state of its arguments and member fields excluding the newly introduced key-value storage.

One advantage of our purity analyzer *purano* is that it adopts a heuristic approach to automatically identify this kind of the cache semantics, and automatically exclude the fields that are used in caching from the purity analysis. As a result, the purity of the target function is still reported as pure by our analyzer. Therefore, we can say the purity of the target function is preserved during the refactoring. The preservation

of purity is also held for other functions involving in the refactoring because the modification on the key-value storage is introduced if and only if there are modifications on other member fields.

We value the preservation of purity throughout our study because we view the purity as not only the output result of our analyzer but also an inherently designed property of the function. It can serve as a metric to indicate the "functional" aspect of the software implementation.

V. Experiment

We conducted an experiment on the proposed *Memoization* refactoring to evaluate our approach and to demonstrate the possibility of the *purity-guided refactoring* in general.

A. Target Software

We applied *Memoization* refactoring on 3 open-source software projects. These projects are collected from the maven repository and we use their latest available source code release. All these 3 projects can be directly built with `mvn install` command and tested with `mvn test` command by their bundled test cases. The version and statistic metrics of these 3 projects are listed in Table I, where the code coverage is measured by block unit. We chose these projects for their relatively high coverage by the test cases, as shown in Table I.

B. Experiment Methodology

In a real-world development process, refactoring is generally initiated by the programmer. The programmer may select the target functions by his knowledge and execute desired refactoring operations on them. Then refactoring tools will check the pre-conditions and perform the refactoring on the selected functions. However, we do not have the deep knowledge on the target software as their developers in this experiment. Instead, we use a profiler to provide the knowledge about the performance of the source code. We use the `hprof` java-agent library provided with the standard JDK as the profiler.

In addition, we assumed that the programmer is provided with a purity analyzer like our *purano* by the IDE so that the purity information is available for the programmer. With these conditions, we conducted the experiment in the following steps for each software target:

1) Run the test cases by the profiler with the original code.
2) Get the top-20 *hot methods* from the profiling result that take most accumulated execution time.
3) Check the pre-conditions on all *hot methods* to get a list of candidates.
4) Apply the *Memoization* refactoring on all the candidates.
5) Run the test cases by the profiler with refactored code.

We checked the pre-conditions only on the top-20 hot methods because they have the most influence on the performance. We applied refactoring on 2 candidates in HTMLParser, 2 candidates in Joda-Time and 1 candidate in PCollections.

C. Results

The total execution time and heap usage before and after the refactoring provided by the profiler is shown in Table II. We can see from the table that the performance improvements vary from 3.1% to 31.3% in these 3 projects. We measured JVM peak heap usage of test cases for these projects. The memory usage increased 23% for HTMLParser, but not changed much for Joda-Time and PCollection, because the objects doing caching are short-lived during the test cases. Meanwhile, we observed more objects are created during the test cases. For example, there are 27,991 more `String` objects created for HTMLParser and 13,893,640 more `Long` objects created for Joda-Time.

We also compare the profiling results before and after the refactorings using *hpjmeter* and draw bar graphs on top-20 most changed functions in Figure 3. In these bar graphs, each bar represents a function. The length of the bar represents the changed accumulated time between two executions. The text titles of the bars are the differences of the accumulated execution time in milliseconds and the fully-qualified names. For example, we can see from the bar graph of Joda-Time that the execution time of `IslamicChronology.isLeapYear` is reduced by 282s and the execution time of `Long.hashCode` is increased by 59s.

D. Discussion

The execution time of each function in these test cases is generally short. Therefore, as we can observed from the bar graphs, the overhead of introducing *Memoization* is relatively large. In fact, the refactoring patterns applied on HTMLParser are optimized with **O1** while the refactoring patterns applied on Joda-Time and PCollections are optimized with **O2**. As we can see, the main overhead in Joda-Time and PCollections are introduced with `HashMap` operations.

Nevertheless, we observed the improvements in performance for all 3 targets. As all these projects are well-tested and focused on speed, we considered these results still remarkable.

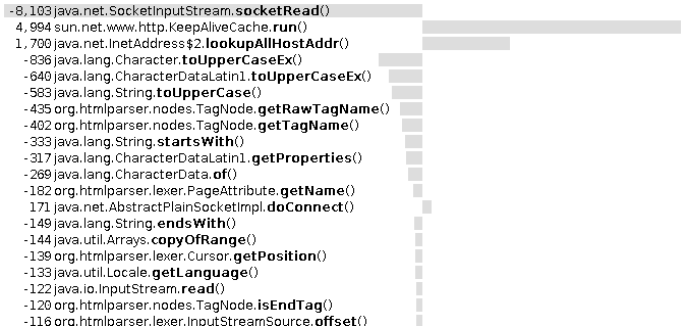
As for the preservation of semantics, the results for test cases are not changed before and after the refactoring. In fact, only a single test case for HTMLParser failed, both before and after the refactoring, because the test case retrieved a URL on the Internet and the content have been changed since the test case was written. All other test cases passed without errors.

TABLE I
Experiment Targets

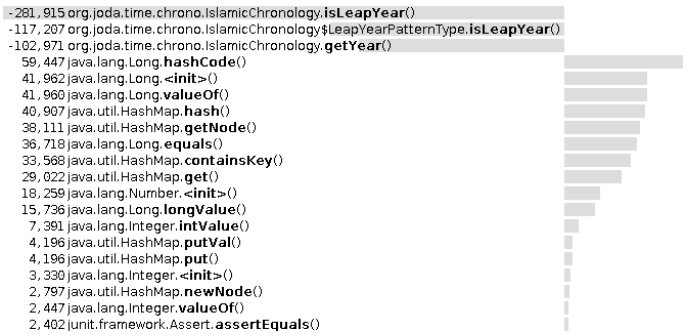
Software & Version	Class	Function	Pure	Test	Coverage
HTMLParser 2.2	157	1,643	707	472	67%
Joda-Time 2.8.1	247	4,411	1,334	4,157	90%
PCollections 2.1.3	31	282	77	22	74%

TABLE II
Profiling Results

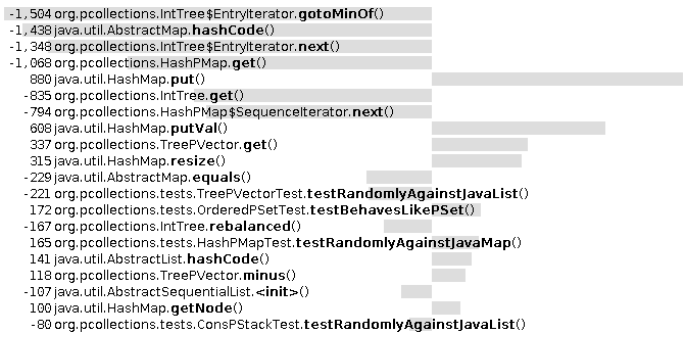
Software	Performance			Heap Usage (MB)	
	Before	After	Improve	Before	After
HTMLParser	1m44s	1m31s	12.5%	13	16
Joda-Time	60m10s	58m17s	3.1%	10	10
PCollections	16s	11s	31.3%	15	15



(a) HTMLPARSER



(b) JODA-TIME



(c) PCOLLECTIONS

Fig. 3. Top-20 accumulated execution time changed functions

VI. THREATS TO VALIDITY

The experiment depends largely on the correctness of our purity analyzer *purano*, which is only used and tested in our research group currently. There may exist bugs in the implementation of *purano*, and there are known limitations due to the decisions we made and the heuristic approaches we adopted, which are discussed in our previous paper [6]. However, we believed those decisions directly result from the goal of this research, which is to guide refactoring by the purity of the methods found in object-oriented languages.

We only experimented one kind of the *purity-guided refactoring*, namely *Memoization* refactoring, on 3 open-source Java libraries, and only tested them using test cases rather than real workloads. We are aware that the refactorings happen in real-world developments may differ from the experimental environments we have made. Meanwhile, as we tested on well-tested speed-oriented open-source projects and still observed improvements on performance, we believe that the refactorings in the real development process would have more opportunities to apply the proposed refactoring.

VII. CONCLUSION AND FUTURE WORK

In this paper, we focused on *purity-guided refactoring* that utilize the purity information of functions during the automated refactoring on object-oriented languages such as Java. We conducted an experiment on a refactoring pattern called *Memoization* that caches the calculation result for pure functions. We tested the refactoring pattern on 3 open-source Java libraries and observed improvements in performance and preservation of semantics by running a profiler on the bundled test cases on these libraries.

We are still in an early stage of this research, continually evaluating new refactoring techniques that require purity information at hand. In the future, we will develop more refactoring approaches, for example, one converts a single-threaded sequential program to a thread-pool based multi-threading program or an event-driven asynchronized program. We are planning to testify our methodology on larger programs or apply the refactorings during the development process. Furthermore, we are also looking into the maintenance burden of the proposed annotations. We will develop tools to update the annotations or check the conformance of the source code with the annotations.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Numbers 25220003 and 24680002.

REFERENCES

[1] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[2] K. Beck, *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[3] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[4] J. Fields, S. Harvie, M. Fowler, and K. Beck, *Refactoring: Ruby Edition*. Pearson Education, 2009.

[5] F. Kjolstad, D. Dig, G. Acevedo, and M. Snir, "Transformation for class immutability," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 61–70.

[6] J. Yang, K. Hotta, Y. Higo, and S. Kusumoto, "Revealing purity and side effects on functions for reusing java libraries," in *Software Reuse for Dynamic Systems in the Cloud and Beyond*. Springer, 2014, pp. 314–329.

[7] H. Xu, C. J. Pickett, and C. Verbrugge, "Dynamic purity analysis for java programs," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2007, pp. 75–82.

[8] H. Rito and J. Cachopo, "Memoization of methods using software transactional memory to track internal state dependencies," in *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*. ACM, 2010, pp. 89–98.

[9] A. Mettler, D. Wagner, and T. Close, "Joe-e: A security-oriented subset of java," in *Network and Distributed Systems Symposium, Internet Society*, vol. 10, 2010, pp. 357–374.

[10] M. Finifter, A. Mettler, N. Sastry, and D. Wagner, "Verifiable functional purity in java," in *Proc. of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 161–174.

[11] A. Sălcianu and M. Rinard, "Purity and side effect analysis for java programs," in *Verification, Model Checking, and Abstract Interpretation*. Springer, 2005, pp. 199–215.