LAMBDAFICATOR: From Imperative to Functional Programming through Automated Refactoring

Lyle Franklin
Ball State University
USA
ljfranklin@bsu.edu

Alex Gyori
Politehnica University of Timisoara
Romania
gyori@cs.upt.ro

Jan Lahoda
Oracle
Czech Republic
jan.lahoda@oracle.com

Danny Dig
University of Illinois
USA
dig@illinois.edu

Abstract—Java 8 introduces two functional features: lambda expressions and functional operations like map or filter that apply a lambda expression over the elements of a Collection. Refactoring existing code to use these new features enables explicit but unobtrusive parallelism and makes the code more succinct. However, refactoring is tedious (it requires changing many lines of code) and error-prone (the programmer must reason about the control-flow, data-flow, and side-effects). Fortunately, these refactorings can be automated.

We present LAMBDAFICATOR, a tool which automates two refactorings. The first refactoring converts anonymous inner classes to lambda expressions. The second refactoring converts for loops that iterate over Collections to functional operations that use lambda expressions. In 9 open-source projects we have applied these two refactorings 1263 and 1595 times, respectively. The results show that LAMBDAFICATOR is useful. A video highlighting the main features can be found at: http://www.youtube.com/watch?v=EIyAflgHVpU

I. INTRODUCTION

Some object-oriented languages such as Smalltalk, Scala, JavaScript, Ruby supported lambda expressions from the first release. Others, like C# (v 3.0), C++ (v 11) were retrofitted with lambda expressions. Java 8 is the latest mainstream language to retrofit lambda expressions [1].

Enabled by lambda expressions, the Java 8 collections [2] provide internal iterators [3] that take a lambda expression as an argument. For example, filter takes a predicate expression and filters the elements of a collection. map maps the elements of a collection into another collection, forEach executes a block of code over each element, etc. The internal iterators enable the library developers to optimize performance, for example by providing parallel implementation, short-circuiting, or lazy evaluation.

Until now, Java did not support lambda expressions, but instead emulate its behavior with an anonymous inner class (from here on referred as AIC). An AIC typically encodes nothing more than a function. The Java class library defines several interfaces that have just one method. These are called functional interfaces and are mostly instantiated as AIC. Classic examples are Runnable – whose run method encapsulates work to be executed inside a Thread, Comparator – whose compare method imposes a total order on a collection of objects, or ActionListener – whose actionPerformed method encapsulates the behavior when an action (like pressing a GUI button) is performed.

Refactoring existing Java code to use lambda expressions brings several benefits. First, the refactoring makes the code more succinct and readable by introducing more concise expressions. Previously, using the old AIC, the programmer had to write five lines of code to encapsulate a single statement.

Second, in the refactored code it is easy to introduce explicit but unobtrusive parallelism by simply using parallel:

```java
myCollection.parallelStream()
    .map(e -> e.length());
```

Third, the refactored code makes the intent of the loop more explicit. Suppose we wanted to iterate over a collection of blocks, and color all blue blocks in red. Compared to the old style of external iterators (e.g., with a for statement), the refactored loop is:

```java
blocks.stream()
    .filter(b -> b.getColor() == BLUE)
    .forEach(b -> { b setColor(RED); });
```

This style encourages chaining the operations in a pipeline fashion, thus there is no need to store intermediate results in their own collections. Many programmers prefer this idiom, as witnessed by its popularity in Scala [4], FluentIterable [5] in Guava Google Libraries, or Microsoft PLINQ library [6].

Fourth, elements may be computed lazily: if we map a collection of a million elements, but only iterate over the results later, the mapping will happen only when the results are needed.

In this demo we will educate programmers and researchers about the new lambda-related features coming in Java 8. We are also empowering Java developers to use these features effectively by presenting our tool, LAMBDAFICATOR, that automates two refactorings. The first transforms AIC to lambda expressions and the second transforms for loops over Collections to functional operators using lambda expressions. We are the first to implement these refactorings and make them available as an extension to a widely used development environment. We are shipping both refactorings with the official release of the NetBeans IDE.

More details about the program analysis used in LAMBDAFICATOR can be found in our tech report available on the tool’s homepage: http://refactoring.info/tools/LambdaFicator

978-1-4673-3076-3/13 © 2013 IEEE 1287 ICSE 2013, San Francisco, CA, USA
Formal Demonstrations

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
II. USER EXPERIENCE

LAMBDA FACTOR provides two main workflow options, a batch and a Quick Hint mode.

The batch mode allows the programmer to invoke the refactoring automatically by selecting any file, or project open in the NetBeans IDE. LAMBDA FACTOR can automatically apply the refactoring on all files or optionally generate a preview which lists the valid transformations and provides fine-grain control over which transformations should take place. In the batch mode, LAMBDA FACTOR can discover and apply hundreds of refactorings in a matter of seconds. In Fig. 1 we show how LAMBDA FACTOR works in batch mode. We opted to apply the refactoring on the whole Tomcat project. LAMBDA FACTOR groups the changes per file, in the left side panel, so we can inspect and select each of the refactorings to apply. Alternately, we can apply all the refactorings that LAMBDA FACTOR suggests.

The quick hint mode scans the file that is open in the editor in real-time. Fig. 2 shows how LAMBDA FACTOR works in quick hint mode. If LAMBDA FACTOR finds code that meets the refactoring preconditions, it underlines the code and displays a hint in the sidebar indicating that the refactoring is available. If the programmer clicks the hint indicator, LAMBDA FACTOR applies the refactoring. This option allows the programmer to perform the refactoring without deviating from her normal workflow.

III. OVERVIEW OF REFACTORINGS

We illustrate the problems and challenges of ANONYMOUS-TO-LAMBDA and FOR-LOOP-TO-FUNCTIONAL by showing examples of refactorings that LAMBDA FACTOR performs. We designed the analysis and transformation algorithms to address the challenges for these two refactorings. These algorithms account for different scoping rules between the old and the new languages constructs and convert imperative in-place mutation into functional computations that produce new values.

A. ANONYMOUS-TO-LAMBDA

Fig. 1–left-hand side shows a common practice in multi-threaded Java code, encapsulating some asynchronous computation inside a Runnable. In this example, the developer used an AIC, avoiding the hassle of creating a separate class for running one single line of code asynchronously. Although an AIC is an improvement over an external class, the syntax is still unnecessarily verbose. The programmer must specify the name of the interface, the method signature, and finally the body of the method. Lambda expressions are a more concise solution. With lambda expressions, the compiler can infer the type of the interface as well as the method signature. The programmer only has to specify the body of the method. Fig. 1–right-hand shows a lambda expression equivalent to the AIC on the left-hand side. LAMBDA FACTOR safely removes the code that is cluttering the intent and makes it more concise and readable.

While Fig. 1 shows the most basic case, LAMBDA FACTOR analyzes the code deeper to handle several special cases. Fig. 3, adapted from the Apache Tomcat project, shows an example where the basic conversion would introduce a compilation error. The doAction method is overloaded and can accept two different interfaces, both of which define a single method run(). A naive conversion results in an ambiguous type for the lambda expression at the call site on line 1, due to the method overloading. LAMBDA FACTOR detects the need for a type cast and adds it, disambiguating the type of the lambda expression.

This example also illustrates that LAMBDA FACTOR makes the resulting lambda expression even more concise. If the body of the lambda expression contains a single return statement, LAMBDA FACTOR removes the return statement. These special cases require additional analysis and would require special attention to refactor manually.

Moreover, these changes are non-trivial. When converting AIC to lambda expressions, the programmer must first account for the different scoping rules between AIC and lambda expressions. These differences could introduce subtle bugs. For example, this or super are relative to the inner class where they are used, whereas in lambda expressions they are relative to the enclosing class. Similarly, local variables declared in the AIC are allowed to shadow variables from the enclosing class, whereas the same variables in the lambda expression will conflict with variables from the enclosing class. Moreover, converting AIC to lambda could make the resulting type ambiguous, thus it requires inferring the type of the lambda.

B. FOR-LOOP-TO-FUNCTIONAL

Next we illustrate two examples of the FOR-LOOP-TO-FUNCTIONAL refactoring in Fig. 4. The first example shows a loop that iterates over GrammarEngine objects. The loop checks whether importedEngines contains an element with a given name. The loop filters out objects with a null name, through the continue statement, and checks if the name equals the argument of the method for each non-null name. Our refactored code makes the intent explicit: it shows a non-null filter and returns true if any element’s name matches the grammarName. This example illustrates how LAMBDA FACTOR chains operations together, while expressing the semantics of each portion of the loop explicitly. LAMBDA FACTOR is able to determine the overall semantic of the loop and what operators to use. LAMBDA FACTOR is able to infer that the two operators can safely be chained and that the types match. Also it determines that the if with a continue behaves like a non-null filter and infers the right operation. In this case, the
Fig. 1. LAMBDA FICATOR performs the ANONYMOUS TO LAMBDA refactoring in batch mode.

When performing the FOR LOOP TO FUNCTIONAL refactoring, LAMBDA FICATOR considers a set of opposing constraints. First, LAMBDA FICATOR determines what operation each statement considers a set of opposing constraints. First, LAMBDA FICATOR discards the \{ \} and return tokens to make the lambda expression more concise.

Fig. 3. Example of ambiguous lambda expression due to method overloading. LAMBDA FICATOR adds a type cast to disambiguate the type of the lambda expression on line 1. In addition, LAMBDA FICATOR discarded the \{ \} and return tokens to make the lambda expression more concise.

Fig. 4. Example of FOR LOOP TO FUNCTIONAL refactoring. In column (a) you can find the original version of the program and in column (b) the refactored one. The examples are extracted from ANTLR.
Another thing to consider is that for loops are inherently eager constructs. LAMBDA\textsc{FCATOR} ensures all lazy operations get executed to preserve original semantics. Thus, it requires that the last operation in the chain be an eager operation; this will force the lazy operations to execute as needed, i.e., just before the eager operation. Notice that eager operations cannot be chained because they do not return streams.

IV. EVALUATION

We evaluated our implementations by running the two refactorings on 9 open-source projects (totaling almost 1M SLOC), invoking \textsc{AnonymousToLambda} 1263 times, and \textsc{ForLoopToFunctional} 1595 times. The results show that the refactorings are widely applicable: the first refactoring successfully converted 55% of AIC and the second refactoring converted 63% of for loops. Second, the refactorings are valuable: the first refactoring reduces the code size by 2213 SLOC, while the second refactoring infers 1093 operators and 982 chains thus making the intent of the loop explicit. Third, \textsc{Lambda\textsc{FCATOR}} saves the programmer from manually changing 3707 SLOC for the first refactoring, and 4831 SLOC for the second refactoring.

V. RELATED WORK

Pankratius et al.’s empirical study [7] shows that programmers employ a mix of functional and imperative styles when writing parallel applications. Okur and Dig [6] empirically show that functional operators provided in .NET, equivalent to those being introduced in Java 8, are widely used when writing parallel applications. LAMBDA\textsc{FCATOR} meets this need by transforming serial, imperative constructs into functional constructs, which are a precursor to parallelism. Ericksen [8] reports on Scala’s mix of functional and imperative style used in large commercial applications like Twitter.

Recently, there is a surge of interest in supporting refactorings in functional languages [9]–[11]. However, we are the first ones to help programmer retrofit functional features into an imperative program.

Our work on refactoring for parallelism contains a tool [12] that performs a related refactoring to ParallelArray rather than using lambda expressions. Our tool could benefit from the automatic thread safety analysis performed by this toolset. LAMBDA\textsc{FCATOR} improves on our previous refactoring [12] by providing increased applicability, increased readability, and operator chaining. Our tool can also infer chaining of operators, permitting the refactoring of more complex loops, such as iterations involving multiple control flow paths. LAMBDA\textsc{FCATOR} also takes advantage of built-in language features, such as lambda expressions, rather than external libraries, resulting in improved readability.

Davis and Kiczales [13] present an approach to let programmers experiment with new language extensions without requiring that the whole toolset (e.g., compiler, editor, etc) support the new extensions.

VI. CONCLUSIONS

There exists an interdependence between language features, adoption of these features in practice, and tools. On one hand, tools do not automate features that are rarely used in practice. On the other hand, language features are not used in practice if they do not have tool automation. Once we break the chicken-and-egg stalemate, tools and adoption are in a chain reaction with a positive feedback.

The concomitant release of lambda expressions in Java 8 and our release of LAMBDA\textsc{FCATOR} may be the first time when language features and refactoring tools are released together. This could be the trigger for the chain reaction that will lead to a wide adoption of functional/imperative hybrid, thus making the programmer more productive.

ACKNOWLEDGMENT

Lyle Franklin and Alex Gyori did this research while undergraduate summer interns at the Information Trust Institute at UIUC. The authors would like to thank Marius Minea, Darko Marinov, Milos Gligoric, Stas Negara, Mihai Codoban, Caius Brindescu, Yu Lin, Krzysztof Zienkiewicz, Nick Chen, and Qingzhou Luo for providing valuable feedback on earlier drafts of this paper. This research is partly funded through an NSF CCF-1219027 grant, a gift grant from Intel, and the Intel-Illinois Center for Parallelism at the University of Illinois at Urbana-Champaign. The Center is sponsored by the Intel Corporation.

REFERENCES

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 2004.