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`:

```
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:

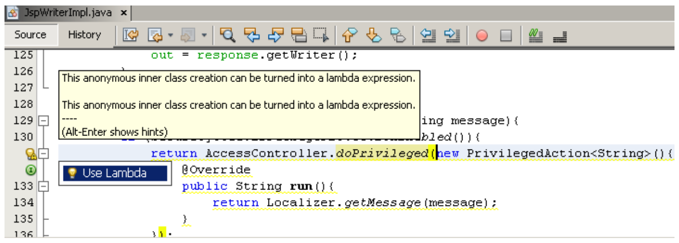
```
blocks.stream()
     .filter(b -> b.getColor() == BLUE)
     .forEach(b -> { b.setColor(RED);});
```

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



```
out = response.getWriter();
```

This anonymous inner class creation can be turned into a lambda expression.

This anonymous inner class creation can be turned into a lambda expression.

(Alt-Enter shows hints)

```
                                    ing message){
                                    bled()){
    return AccessController.doPrivileged(new PrivilegedAction<String>(){
        @Override
        public String run(){
            return Localizer.getMessage(message);
        }
    });
```

Fig. 2. LAMBDAFICATOR performs the refactoring in *Quick Hint* mode.

II. User Experience

LAMBDAFICATOR 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. LAMBDAFICATOR 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, LAMBDAFICATOR can discover and apply hundreds of refactorings in a matter of seconds. In Fig. 1 we show how LAMBDAFICATOR works in batch mode. We opted to apply the refactoring on the whole Tomcat project. LAMBDAFICATOR 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 LAMBDAFICATOR suggests.

The quick hint mode scans the file that is open in the editor in real-time. Fig. 2 shows how LAMBDAFICATOR works in quick hint mode. If LAMBDAFICATOR 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, LAMBDAFICATOR 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 ANONYMOUSTOLAMBDA and FORLOOPTOFUNCTIONAL by showing examples of refactorings that LAMBDAFICATOR 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. AnonymousToLambda

Fig. 1–left-hand side shows a common practice in multithreaded 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. LAMBDAFICATOR safely removes the code that is cluttering the intent and makes it more concise and readable.

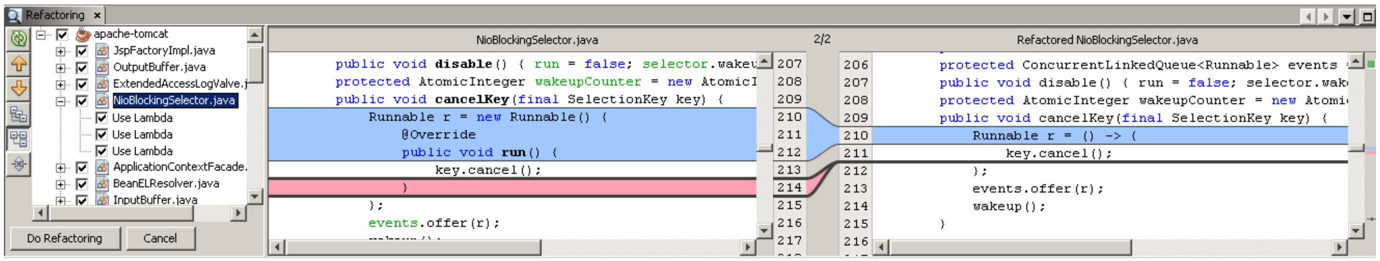
While Fig. 1 shows the most basic case, LAMBDAFICATOR analyzes the code deeper to handle several special cases. Fig. 3, adapted from the Apache Tomcat project, shows an example where the basic conversion would introduce a compilation error. The `doAction` method is overloaded and can accept two different 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. LAMBDAFICATOR detects the need for a type cast and adds it, disambiguating the type of the lambda expression.

This example also illustrates that LAMBDAFICATOR makes the resulting lambda expression even more concise. If the body of the lambda expression contains a single `return` statement, LAMBDAFICATOR 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. ForLoopToFunctional

Next we illustrate two examples of the FORLOOPTOFUNCTIONAL 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 LAMBDAFICATOR chains operations together, while expressing the semantics of each portion of the loop explicitly. LAMBDAFICATOR is able to determine the overall semantic of the loop and what operators to use. LAMBDAFICATOR 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



```
public void disable() { run = false; selector.wakeu
protected AtomicInteger wakeupCounter = new AtomicI
public void cancelKey(final SelectionKey key) {
    Runnable r = new Runnable() {
        @Override
        public void run() {
            key.cancel();
        }
    };
    events.offer(r);
```

```
protected ConcurrentLinkedQueue<Runnable> events
public void disable() { run = false; selector.wak
protected AtomicInteger wakeupCounter = new Atomi
public void cancelKey(final SelectionKey key) {
    Runnable r = () -> {
        key.cancel();
    };
    events.offer(r);
    wakeup();
}
```

Fig. 1. LAMBDAFICATOR performs the ANONYMOUSTOLAMBDA refactoring in batch mode.

```
1  String sep = doAction(new PrivilegedAction() {
2    public String run() {
3      return System.getProperty("file.separator");
4    }
5  });
6
7  String doAction(PrivilegedAction action) {...}
8  String doAction(ExceptionAction action) {...}
```
(a) An AIC

```
1  String sep = doAction((PrivilegedAction)() ->
2    System.getProperty("file.separator")
3  );
4
5
6
7  String doAction(PrivilegedAction action) {...}
8  String doAction(ExceptionAction action) {...}
```
(b) Lambda conversion requiring a type cast

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

```
public class GrammarEngineImpl implements GrammarEngine {
  ...
  private boolean isEngineExisting(String grammarName) {
        for(GrammarEngine e : importedEngines) {
            if(e.getGrammarName() == null) continue;
(1)         if(e.getGrammarName().equals(grammarName))
              return true;
        }
        return false;
    }
}
class EditorGutterColumnManager{
  ...
  public int getNumberOfErrors() {
    int count = 0;
    for (ElementRule rule : getRules()) {
(2)     if (rule.hasErrors())
          count+=rule.getErrors().size();
    }
    return count;
  }
}
```
(a)

```
class GrammarEngineImpl implements GrammarEngine {
  ...
  private boolean isEngineExisting(String grammarName) {
    return importedEngines.stream()
      .filter(e -> e.getGrammarName() != null )
      .anyMatch(e ->
          e.getGrammarName().equals(grammarName) );


  }
}

class EditorGutterColumnManager{
  ...
  public int getNumberOfErrors() {

    return getRules().stream()
        .filter(rule -> rule.hasErrors() )
        .map(rule -> rule.getErrors().size())
        .reduce(0, Integer::sum);

  }
}
```
(b)

Fig. 4. Example of FORLOOPTOFUNCTIONAL 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

semantics is preserved due to the fact that `anyMatch` uses short-circuiting to ensure that elements are iterated only until one element matches the predicate.

The second example also illustrates chaining operations together, this time to compute a map-reduce. In this example, the loop iterates over `ElementRule` objects and sums up the number of errors for each object that has errors. In order for the programmer to infer this chaining manually, she has to notice that the compound assignment represents a map-reduce operation, which may not be immediately obvious. In this transformation we used method references, a new feature in Java 8, to refer to the plus operator on `Integer`.

When performing the FORLOOPTOFUNCTIONAL refactoring, LAMBDAFICATOR considers a set of opposing constraints. First, LAMBDAFICATOR determines what operation each statement in the `for` loop represents. This involves reasoning about statements that branch the control flow and introduce side effects on local variables.

LAMBDAFICATOR also considers several differences between the original loop and the new operations. A local variable declared in the original loop is available to all subsequent statements. However, variables declared in a lambda expression are now local to that lambda. LAMBDAFICATOR builds operations in a pipeline fashion such that it maintains access to needed references. In some cases, LAMBDAFICATOR merges operations to ensure the variable references are preserved. This is due to the constraint that operations can return only one value.

On the other hand, there are several ways of chaining operations when refactoring a loop. LAMBDAFICATOR chooses the most fine-grained operations in order to make the semantic of each portion of code as explicit as possible.

Another thing to consider is that `for` loops are inherently eager constructs. LambdaFicator 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 AnonymousToLambda 1263 times, and 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, LambdaFicator 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. LambdaFicator 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. LambdaFicator 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. LambdaFicator 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 LambdaFicator 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

[1] State of the lambda. [Online]. Available: http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-4.html

[2] State of the lambda: Collections edition. [Online]. Available: http://cr.openjdk.java.net/~briangoetz/lambda/sotc3.html

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

[4] The Scala Programming Language. [Online]. Available: http://www.scala-lang.org/

[5] FluentIterable. [Online]. Available: http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/FluentIterable.html

[6] S. Okur and D. Dig, "How do developers use parallel libraries," in *FSE'12*, pp. 54–65.

[7] V. Pankratius, F. Schmidt, and G. Garretón, "Combining functional and imperative programming for multicore software: an empirical study evaluating Scala and Java," in *ICSE'12*, pp. 123–133.

[8] M. Eriksen, "Scaling Scala at Twitter," in *ACM SIGPLAN Commercial Users of Functional Programming*, ser. CUFP '10.

[9] H. Li, "Refactoring Haskell Programs," 2006.

[10] D. Y. Lee, "A case study on refactoring in Haskell programs," in *ICSE '11*, pp. 1164–1166.

[11] H. Li and S. Thompson, "Comparative study of refactoring haskell and erlang programs," in *SCAM '06*, pp. 197–206.

[12] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson, "ReLooper: refactoring for loop parallelism in Java," in *OOPSLA'09: Demo*, pp. 793–794.

[13] S. Davis and G. Kiczales, "Registration-based language abstractions," in *OOPSLA '10*, pp. 754–773.