Crossing the Gap from Imperative to Functional Programming through Refactoring

Alex Gyori
University of Illinois, USA
gyori@illinois.edu

Lyle Franklin
Ball State University, USA
ljfranklin@bsu.edu

Danny Dig
Oregon State University, USA
digd@eecs.oregonstate.edu

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

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. It is also error-prone: the programmer must reason about the control-, data-flow, and side-effects. Fortunately, refactorings can be automated.

We designed and implemented 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. Using 9 open-source projects, we have applied these two refactorings 1263 and 1709 times, respectively. The results show that LambdaFicator is useful: (i) it is widely applicable, (ii) it reduces the code bloat, (iii) it increases programmer productivity, and (iv) it is accurate.

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement; D.2.3 [Software Engineering]: Coding Tools and Techniques

General Terms: Refactoring

Keywords: Java 8, Lambda Expressions, Imperative Programming, Functional Programming

1. INTRODUCTION

A lambda expression (also called an anonymous function) is a function without a name identifier. For example, `(int x, int y) -> x + y` is a lambda expression that takes two integer args and returns their sum. Lambda expressions can be conveniently passed as parameters or can be returned from functions, and are the hallmark of functional languages.

Some object-oriented languages such as Smalltalk, Scala, JavaScript, and Ruby supported lambda expressions from the first release. Others, like C# (v 3.0), C++ (v 11) were retrofitted with lambda expressions. Java 8 (to be released in 2014) is the latest mainstream language to retrofit lambda expressions [6].

The driving motivation for retrofitting lambdas in mainstream imperative languages is to make it easier to write parallel code. The hardware industry has shifted to multicore processing on all fronts: phones, tablets, laptops, desktops, etc. The software industry trend is to hide the complexity of writing parallel code behind parallel libraries. For example, the C# TPL [8] and PLINQ [3] libraries, or the C++ TBB [2] library rely heavily on lambda expressions to encapsulate functions that are passed to library APIs to be executed in parallel.

Enabled by lambda expressions, the upcoming Java 8 collections [7] provide *internal iterators* [11] that take a lambda expression as an argument. For example, `filter` takes a predicate expression and removes elements of a collection based on the predicate, `map` maps the elements of a collection into another collection, `forEach` executes a block of code over each element, etc. The internal iterators enable 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 emulated their behavior with anonymous inner classes (from here on referred as AIC). An AIC typically encodes just 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`, and `Comparator` – whose `compare` method imposes a total order on a collection.

Refactoring existing Java code to use lambda expressions brings several benefits. First, the refactoring makes the code more succinct and readable when introducing explicit but unobtrusive parallelism. The parallel code below:

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

would have taken 25 lines of code had we used the classic `Thread` and `Runnable` idioms (see example in Fig. 1).

Even when not using parallelism, the programmer can write succinct expressions when using lambdas. Previously, using the old AIC, the programmer had to write five lines of code to encapsulate a single statement.

Second, 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

ESEC/FSE '13, August 18-26, 2013, Saint Petersburg, Russia

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 [9], `FluentIterable` [1] in Guava Google Libraries, or Microsoft PLINQ library [19].

Third, 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.

To get all the benefits of lambda expressions and internal iterators, the Java programmer must refactor (i) AIC into lambda expressions and (ii) `for` loops into internal iterators. However, these refactorings are *tedious*. Java projects are riddled with many anonymous classes and external iterators. For example, ANTLRWorks, a medium-sized open-source project (97K non-blank, non-comment source lines – SLOC) contains 151 AICs and 589 external iterators. Refactoring ANTLRWorks by hand requires changing 513 SLOC for the first refactoring, and 2293 SLOC for the second refactoring.

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. Second, converting AIC to lambda could make the resulting type ambiguous, thus it requires inferring the type of the lambda.

When converting `for` loops to internal iterators there are several challenges. First, there are many ways to split loop statements into pipelined operations. Ideally the programmer would choose the most fine-grained to enable precise control of parallelism and make the intent of statements more explicit. Second, the programmer must reason about the control flow statements like `break`, `continue`, `return` and choose the appropriate operations. Third, the programmer must account for different scoping rules between the original `for` and the lambdas: a variable declared in a loop statement is available to subsequent loop statements, whereas a variable declared in one lambda expression is now local to that lambda. This requires identifying the variables that need to be passed through the pipeline. Fourth, the programmer must verify there are no side effects on local variables defined outside of the lambda. Fifth, the programmer must reason about the nature (e.g., eager vs. lazy) of operations in order to preserve the semantics of the original `for`.

To solve these challenges, we designed, implemented, and evaluated LambdaFicator, the first refactoring tool to automate the task of retrofitting functional features into imperative code. LambdaFicator currently supports two refactorings. The first refactoring, AnonymousToLambda, replaces AIC with the equivalent lambda expression. The second refactoring, ForLoopToFunctional, replaces `for` loops with their equivalent chained operations.

This paper makes the following contributions:

Problem: to the best of our knowledge, this is the first paper to describe the novel problem of converting imperative code to a functional flavor using lambdas.
Algorithms: we designed the analysis and transformation algorithms to address the challenges for two refactorings that convert AIC into lambda expressions and `for` loops into functional operations. These algorithms account for different scoping rules between the old and the new languages constructs and convert imperative in-place mutations into functional computations that produce new values.
Implementation: 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 refactoring with the official release of the NetBeans IDE.
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 1709 times. The results show that the refactorings are widely *applicable*: the first refactoring successfully converted 55% of AIC and the second refactoring converted 46% of `for` loops. Second, the refactorings are *valuable*: the first refactoring reduces the code size by 2213 SLOC, while the second refactoring infers 2681 operators and 1709 chains thus making the intent of the loop explicit. Third, LambdaFicator saves the programmer from manually changing 3707 SLOC for the first refactoring, and 12313 SLOC for the second refactoring. Fourth, when executed in batch mode on the whole projects, the AnonymousToLambda refactoring has perfect accuracy. For ForLoopToFunctional, the tool infers the most fine-grained operations more than 90% of the time.

LambdaFicator has been successfully evaluated by the ESEC/FSE artifact evaluation committee and found to exceed expectations. LambdaFicator, along with experimental data and a demo, is available at:
`http://refactoring.info/tools/LambdaFicator`
`http://www.youtube.com/watch?v=EIyAflgHVpU`

2. MOTIVATING EXAMPLES

We first illustrate the motivation behind introducing lambda expressions for parallelism. Fig. 1(a) shows a simple sequential loop from the ANTLRWorks project. The loop iterates over `ElementRule` objects and resets each object. The programmer decides to execute this loop in parallel.

Fig 1(b) shows how the programmer would traditionally refactor the original loop. She first decides the amount of parallelism (e.g., 4 parallel threads). Then she creates two loops: the first loop splits the work between the 4 parallel threads by allocating a quarter of the iterations to each worker thread. She encapsulates the parallel computation inside an anonymous instance of `Runnable`. Then she starts the threads. The second loop waits for all the worker threads to finish their work. The programmer could have also expressed the parallel computation by subclassing `Thread`, but in this case the code bloat would be even more severe.

Besides code bloat, there are several other issues with this parallel implementation. First, the amount of parallelism is hardcoded, so if she runs the code on a machine with 8 hardware threads, the code will utilize only 4. Second, just because she split the iterations equally among the worker threads, it does not mean that the running time of the 4 worker threads is equal. For example, suppose that the rules visited by the first worker thread have a much richer hier-

```
private void resetRules() {
  for ( ElementRule r : properties.getRules()) {
    r.resetHierarchy();
  }
}
```

(a) A sequential loop

```
private void resetRules() {
  int n = 4;// amount of parallelism
  Thread[] threads = new Thread[n];
  final List<ElementRule> rules =
                              properties.getRules();
  int size = rules.size();
  for ( int i = 0; i < n; i++) {
    final int from = i * size / n;
    final int to = (i + 1) * size / n;
    threads[i] = new Thread(new Runnable() {
      @Override
      public void run() {
        for ( int j = from; j < to; j++) {
          rules.get(j).resetHierarchy();
          }
        }
      });
    threads[i].start();
  }
  for ( int i = 0; i < n; i++) {
    try {
      threads[i].join();
      } catch ( InterruptedException ex) {
       //print error message
    }
  }
}
```

(b) Parallel loop with `Runnable` and `Thread` (old style)

```
private void resetRules() {
  properties.getRules().parallelStream().
    forEach((ElementRule r) -> r.resetHierarchy());
}
```

(c) Parallel loop with functional operations (new style)

Figure 1: Comparison between methods of parallelization in Java.

archy than all other rules, so the first worker thread will spend a much longer time computing. Because the work is not split evenly between the worker threads, the computation would take longer. This problem is referred in literature as dynamic load balancing [13].

By taking advantage of the parallel functional operators introduced in Java 8, the programmer can refactor the sequential loop using LambdaFicator. Fig 1(c) shows the final code, which is much more succinct than the previous parallel code, and also benefits from automatic dynamic load balancing. Notice that `parallelStream` returns a parallel view of the collection, thus `forEach` will execute in parallel.

Next, we illustrate the problems and challenges of AnonymousToLambda and ForLoopToFunctional by showing examples of refactorings that LambdaFicator performs.

Fig. 2(a) shows a common practice in any Java GUI, adding a listener to a button. In this example, the developer used an AIC, avoiding the hassle of creating a separate class for a simple button action. 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. 2(b) shows a lambda equivalent to the AIC in Fig 2(a).

```
button.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    ui.dazzle(e.getModifiers());
  }
});
```

(a) An anonymous inner class

```
button.addActionListener((ActionEvent e) -> {
  ui.dazzle(e.getModifiers());
});
```

(b) Equivalent lambda expression

Figure 2: Example of AnonymousToLambda refactoring, taken from Lambda Design Specification [6].

While Fig. 2 shows the most basic case, LambdaFicator must analyze 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 adds a type cast, disambiguating the type of the lambda expression.

This example also illustrates that LambdaFicator can make 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. We discuss additional special cases in Section 3.

Next we illustrate three 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 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. To perform this refactoring manually, a programmer would have to determine that the `if` statement followed by a `continue` behaves like a non-`null` filter, and the `return true` inside the `if` statement represents an `anyMatch`.

The second example also illustrates chaining operations together, this time to compute `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 `sum` operation on `Integer`.

The third example illustrates additional challenges of chaining operations. This loop iterates over `Entry` objects, and performs several checks before it adds an object to a collection. In this example the programmer would need to reason about the flow of data between statements to determine

```
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

Figure 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.

whether operations can be chained. At first glance, this loop appears to be a chain of `filter`, `map`, `filter`, `forEach`. However, the second `filter` operation filters out `ClassLoader` objects but the last statement needs a reference to the `entry` object; this would not be available from this `filter`, therefore, these two operations cannot be chained. Identifying when it is possible to chain operations is non trivial.

3. AnonymousToLambda REFACTORING

This section presents the AnonymousToLambda refactoring. We explain the tool's workflow, how lambda expressions are implemented in Java 8, the preconditions that determine when a conversion can take place, and special cases that differ from the basic conversion.

3.1 Workflow

The tool provides two main workflow options, a "Quick Hint" option and a batch option.

The quick hint option scans the file that is open in the editor in real-time. If LambdaFicator finds a valid conversion, it underlines the code and displays a hint in the sidebar indicating this AIC can be converted into a lambda expression. If the programmer clicks the hint indicator, LambdaFicator applies the refactoring. This option allows the programmer to refactor without deviating from her normal workflow.

The batch option allows the programmer to invoke the refactoring automatically by selecting any file, folder, or project open in the IDE. LambdaFicator can automatically apply the refactoring on all files or optionally generate a preview which lists the valid conversions and provides fine-grain control over which conversions should take place. With the batch option, LambdaFicator can discover and apply hundreds of refactorings in a matter of seconds.

3.2 Lambda Expression Implementation

In order to illustrate how LambdaFicator refactors AIC to lambda expressions, we first describe how lambda expressions are implemented in Java 8 and how they differ from lambda expressions in other programming languages. Many languages, Python for example, allow lambda expressions to be defined as follows: `sum = lambda x, y: x + y`. This expression creates a function `sum` which takes the arguments `x` and `y` and returns the result of `x + y`. Most type systems, including Python and C#, denote lambda expressions with a special `Func` type. While this implementation is highly flexible, the designers of the Java language decided it would introduce unneeded complexity to the Java type system [6]. As an alternative, the designers chose to represent lambda expressions as an instance of an interface defining a single method, i.e., a functional interface. A lambda expression in Java can be defined as follows: `BinaryOperator sum = (x,y) -> x + y`, where `BinaryOperator` is a functional interface defining a single method `op`. This function could be invoked by calling `sum.op(2, 3)`. This representation of lambda expressions is not as flexible or concise as the functional equivalent, but it fits well with the Java type system.

3.3 Preconditions

Although lambda expressions are intended as a more concise alternative to AIC, they are not a complete replacement. There are several preconditions that LambdaFicator checks before refactoring an AIC into a lambda expression. These preconditions are inherent to how lambda expressions are implemented in Java, not limitations of our tool.

(P1) AIC must instantiate from an interface. Instances of abstract or concrete classes cannot be converted to lambda expressions.

(P2) AIC must have no fields, and declare only one method. A lambda expression represents a single anonymous function; therefore, an AIC with multiple methods can not be converted to a single lambda expression.

(P3) AIC must not have references to `this` or `super`. In a lambda expression, `this` and `super` are lexically scoped, meaning they are interpreted just as they would be in the enclosing environment, e.g., as if they appeared in the statement before the lambda expression [6]. However, in an AIC they refer to the inner class itself.

(P4) AIC must not declare a recursive method. In order to perform the recursive call, we must obtain a reference to the anonymous function. While LambdaFicator could perform this refactoring, this could introduce unneeded complexity into the code and harm understandability.

3.4 Special Cases

After checking the preconditions, LambdaFicator further analyses the code to properly refactor several special cases:

(S1) Normally the body of a lambda expression is a block containing several statements. However, if the block consists of just a single return statement, the block and return keyword can be omitted. In these cases, LambdaFicator will remove the `{}` and `return` tokens. Fig. 3 shows an example. This change allows the lambda to become even more concise.

(S2) In most cases, the compiler can infer the type of the lambda expression from the surrounding context. But there are several cases which require LambdaFicator to add a cast in order for the type to be inferred. If the original AIC is being assigned to a reference of its supertype (e.g., a parent interface or type `Object`), LambdaFicator adds a cast

```
class GrammarEngineImpl implements GrammarEngine{
  ...
  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{
  ...
  int getNumberOfErrors(){
    int count = 0;
    for (ElementRule rule : getRules()){
(2)     if (rule.hasErrors())
          count+=rule.getErrors().size();
    }
    return count;
  }
}
class StandardHost{
  ...
  List<String> findReloadedContextMemoryLeaks(){
    List<String> result = new ArrayList<String>();
    for (Map.Entry<ClassLoader, String> entry :
            childClassLoaders.entrySet())
      if(isValid(entry)){
(3)     ClassLoader cl = entry.getKey();
        if (!((WebappClassLoader)cl).isStart())
          result.add(entry.getValue());
      }
    ...
  }
}
```

(a)

```
class GrammarEngineImpl implements GrammarEngine{
  ...
  boolean isEngineExisting(String grammarName){

    return importedEngines.stream()
       .filter(e -> e.getGrammarName() != null )
       .anyMatch(e ->
          e.getGrammarName().equals(grammarName));

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

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

  }
}
class StandardHost{
  ...
  List<String> findReloadedContextMemoryLeaks(){
    List<String> result = new ArrayList<String>();
    childClassLoaders.entrySet().stream()
      .filter(entry -> isValid(entry) )
      .forEach(entry -> {
        ClassLoader cl = entry.getKey();
          if (!((WebappClassLoader) cl).isStart())
            result.add(entry.getValue());});

    ...
    }
}
```

(b)

Figure 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 first two examples are extracted from ANTLRWorks and the last one is adapted from Apache Tomcat

to help the compiler determine which interface the lambda expression actually represents.

(S3) Similarly, if an AIC appears as an argument in an invocation of an overloaded method, the resulting lambda expression could be ambiguous. Consider the example in Fig. 3. Without a cast indicating the proper interface, the compiler has no way to determine which interface the lambda expression represents. LambdaFicator detects such cases and adds a cast to the original AIC, disambiguating the resulting code.

(S4) Finally, special attention must be paid to variable shadowing. Variable shadowing occurs when a variable declared in the method body or parameters of an AIC has the same name as a variable declared in the enclosing scope. In an AIC, the inner local variable declaration simply hides, or shadows, the outer variable. However, a lambda expression is not allowed to shadow a variable that is declared in the enclosing scope. In order to prevent this, LambdaFicator generates a unique variable name to disambiguate the variables inside the lambda that shadow outer variables.

4. ForLoopToFunctional REFACTORING

Next we present the ForLoopToFunctional refactoring. First we introduce the functional operations from Java 8 that we are currently supporting. Then we introduce the set of preconditions that ensure the refactoring is safe. Last we present an algorithm that infers the operator chaining and ensures the chaining is correct.

- `Stream<R> map(Function<? super T, ? extends R> mapper)`
- `Stream<T> filter(Predicate<? super T> predicate)`
- `T reduce(T identity, BinaryOperator<T> reducer)`
- `void forEach(Consumer<? super T> consumer)`
- `boolean anyMatch(Predicate<? super T> predicate)`
- `boolean noneMatch(Predicate<? super T> predicate);`

Figure 5: Method signatures of Java 8 operations.

4.1 New operations in Java 8

Java 8 introduces `interface Stream<T>` [7]. The methods introduced in it are similar in form and semantics to functional programming-style list operations. LambdaFicator currently supports `map`, `filter`, `reduce`, `forEach`, `anyMatch` and `noneMatch`. We will refer to these methods as operations. We show the method signatures of these operations in Fig. 5.

There are some fundamental semantic differences between operations. The first two operations, `map` and `filter`, are *lazy operations*, i.e., they get executed only when their result is needed. The `filter` operation returns a new `Stream<T>` containing only the elements that satisfy the `Predicate`. The `map` operation also returns a new `Stream` where each element

is mapped from the original `Stream`. The last four operations are *eager operations*, i.e., they execute when called. The `anyMatch` and `noneMatch` operations use short-circuiting to stop processing once they can determine the final result. For example, `anyMatch` will examine elements on `Stream<T>` only until it finds one for which `Predicate` is `true`.

4.2 Preconditions

Although many enhanced `for` loops can be converted to the new functional-style operations, these operations are not complete replacements. Lambda bodies cannot contain references to local variables that are not final or *effectively-final*. A variable is effectively final if its initial value is never changed. Therefore, loops that reference non-effectively final variables generally cannot be converted to operations using lambda expressions. The same holds for loops containing branching statements, such as `return`, `break`, and `continue`. However, LambdaFicator takes advantage of the properties of `anyMatch` and `noneMatch` to refactor some loops containing `return` statements, as shown in example 2 in Fig. 4. LambdaFicator also restructures the body of the loop to eliminate `continue` statements in a preprocessing step.

The following preconditions are due to inherent differences between loops and operations, not limitations of our tool. LambdaFicator checks these preconditions before applying the ForLoopToFunctional refactoring:

(P1) The enhanced `for` loop should be iterating over an instance of `java.util.Collection`, rather than an `array`, in order to be able to obtain a `stream`.

(P2) The body of the initial `for` loop does not throw checked exceptions. The lambda expression signature used by the operations does not have a `throws` clause; therefore, these loops cannot be refactored.

(P3) The body of the initial `for` loop does not have more than one reference to local non-effectively-final variables defined outside the loop. Loops having only one outside reference can be refactored if the side-effect can be externalized to a `reduce` operation, according to the heuristic in Tab. 1.

(P4) The body of the initial `for` loop does not contain any `break` statements. The semantic of `break` is inherent to a loop and cannot be handled by chaining operations together.

(P5) The body of the initial `for` loop does not contain more than one `return` statement. LambdaFicator can deal with loops with only one `return` statement as long as they return a `boolean` literal and LambdaFicator can infer that they can be refactored to an `anyMatch` or `noneMatch` operation.

(P6) The body of the initial `for` loop does not contain any labeled `continue` statement. This would introduce a `goto` point which cannot be handled by chaining operations.

4.3 Algorithm

When performing the ForLoopToFunctional refactoring, LambdaFicator needs to consider a set of opposing constraints. First, LambdaFicator must determine 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 has to consider 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 has to build operations in a pipeline fashion such that it maintains access to needed references. In some cases, LambdaFicator must merge 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. This gives the programmer finer control to specify for each operation whether it should execute `sequentially` or in `parallel`.

Finally, `for` loops are inherently eager constructs; the refactored code has to preserve the semantics of the initial code, therefore it has to get executed eagerly. LambdaFicator must ensure any lazy operations get executed by requiring 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.

Below is an overview of the chaining inference algorithm:

- Step 1: Break code into potential operations.
- Step 2: Annotate each potential operation with variable availability information.
- Step 3: Merge operations in order to maintain access to needed references.
- Step 4: Chain the operations.

In step 1, the algorithm marks each statement as a *prospective* operation. Prospective denotes that this is not the final operation, since some operations need to be merged later to meet variable availability constraints. By default, it marks all statements (but `if` statements) as `map` operations. For `if` statements that have no `else` branch and no statements after them, the algorithm marks them as `filter` operations. Finally, the algorithm marks the last statement as a prospective eager operation.

In step 2, the algorithm annotates each prospective operation with variable availability information. We introduce the notions of *Available Variables* and *Needed Variables* of a *Prospective Operation*.

Definition 1. *The set of available variables of a* Prospective Operation*,* AV_{PO}*, is:*

$AV_{PO} = F \cup L_{PO} \cup \{L_{Meth} \setminus L_{Loop}\}$ *Where:*

- F *is the set of all fields declared in the current class or inherited from superclasses and all visible fields from the imported classes.*
- L_{PO} *is the set of local variables declared in the* Prospective Operation
- L_{Meth} *is the set of all local variables of the current method*
- L_{Loop} *is the set of local variables declared within the loop.*

Definition 2. *The set of needed variables of a* Prospective Operation*,* NV_{PO}*, is:* $NV_{PO} = U_{PO} \setminus AV_{PO}$

Here U_{PO} *is the set of all variables used in the* Prospective Operation*.* AV_{PO} *is defined above.*

In step 3 the algorithm uses the sets generated in step 2 to determine if operations can be chained or need to be merged. To do so, it iterates the prospective operations

bottom up. To determine if two operations, O and O' can be chained ($O.O'$), the algorithm checks whether the variable needed in O' can be provided by the previous operation O, as expressed:

PROPOSITION 1. Prospective Operation O *can be chained with* O' *(*$O.O'$*) iff* $|NV_{O'}| = 1$ *and* $NV_{O'} \subseteq (AV_O \cup NV_O)$

If the algorithm cannot chain two operations, it merges all previously built operations into a single operation to ensure variable availability. When merging operations O and O' into O'', the algorithm computes the availability sets as: $AV_{O''} = AV_O \cup AV_{O'}$ $\quad NV_{O''} = \{NV_O \cup NV_{O'}\} \setminus AV_{O''}$

In step 4 LAMBDAFICATOR determines the correct *eager operation*. Afterwards it builds the chain from end to start by prepending the operations.

Finally, LAMBDAFICATOR prepends the expression that returns the `Stream` to the chain. If the last operation is a `reduce`, `anyMatch`, or `noneMatch`, LAMBDAFICATOR might need to assign the result to a variable or return it directly.

Next, we illustrate the algorithm by applying it to example 3 from Fig. 4. LAMBDAFICATOR first checks that all the preconditions are met. No `Exception` is thrown from the loop, and there are no `return`, `break`, or `continue` statements. LAMBDAFICATOR finds a reference to `result` but it is able to determine that the variable is only initialized; therefore, it is *effectively final* and usable from a lambda expression.

The chaining algorithm begins by breaking the list of original statements into *Prospective Operations*. It does this by iterating over the list of statements and checking if each statement conforms to the restrictions of a given operation. It marks the first statement as a *Prospective Filter* since it is an `if` with no `else` branch. It marks the second statement as a *Prospective Map* since it is not an `if` statement. It marks the third statement as *Prospective Filter*, and the last statement automatically becomes *Prospective Eager*.

Next, the algorithm computes the two sets of available and needed variables for each *Prospective Operation* and annotates each *Prospective Operation* with this information.

The algorithm iterates on the four *Prospective Operations* identified in the first step. It starts iterating from bottom up, in reversed control flow order. LAMBDAFICATOR finds that the last operation cannot be chained with the previous; this is because the last operation uses an `entry` and the prospective filter passes down a `ClassLoader` object. Therefore, it merges them into a new operation and recomputes the variable availability sets. In the next iteration, it finds that the current operation, obtained through merging, needs more than one variable from the upstream operation, i.e., `cl` and `entry`, requiring LAMBDAFICATOR to merge the two *Prospective Operations*. In iteration three, LAMBDAFICATOR finds that the *Prospective Operations* can be chained; therefore, it does not merge the operations.

In the last step, the algorithm takes the last *Prospective Operation* and, using the heuristics in Tab. 1, determines that the last operation is a `forEach`. Because the next block is a *Prospective Filter*, LAMBDAFICATOR determines that the first operation is a `filter` operation.

Note: LAMBDAFICATOR does not split `synchronized` blocks and `try-catch` blocks between multiple operations in order to avoid introducing races or changing semantics. This behaviour is omitted during the presentation of the algorithm, but LAMBDAFICATOR ensures the whole `synchronized` block or `try-catch` block is not split across operations.

Table 1 Heuristics to infer the last operation in the chain

Operator	Heuristic
`reduce`	The *Prospective Operation* contains one reference to a non-final local variable defined outside the loop and the *Prospective Operation* contains only one control flow path in it and it contains a write through a compound assignment to that local variable or a unary postfix or prefix increment/decrement. A compound assignment can be +=, -=, *=, /=, %=, |=, &=, ≪=, ≫=.
`anyMatch`	The *Prospective Operation* has only one statement, which is a `return true`.
`noneMatch`	The *Prospective Operation* has only one statement, which is a `return false`.
`forEach`	The *Prospective Operation* does not have any reference to non-effectively-final local variables defined outside the loop and no return statements.

5. EVALUATION

Research Questions. To determine if LAMBDAFICATOR is useful, we answer the following research questions:

Q1. Applicability: How applicable are the refactorings?

Q2. Value: Do the refactorings improve code quality?

Q3. Effort: How much programmer effort is saved by LAMBDAFICATOR when refactoring?

Q4. Accuracy: How accurate is LAMBDAFICATOR when performing the refactoring in batch mode?

Q5. Safety: Is LAMBDAFICATOR safe?

5.1 Experimental Setup

In order to empirically evaluate the usefulness of LAMBDAFICATOR, we ran it on 9 widely used open-source projects. We applied ANONYMOUSTOLAMBDA refactoring 1263 times, and FORLOOPTOFUNCTIONAL 1709 times. These case studies give more confidence that the proposed algorithms and implementations generalize to real-world situations.

The left-hand side of Fig. 6 shows the size of each project in terms of non-empty non-comment source lines of code (generated using David A. Wheeler's SLOCCount [5]).

For each project, we applied the ANONYMOUSTOLAMBDA refactoring to all AIC and FORLOOPTOFUNCTIONAL refactoring to all `enhanced-for` loops (the `enhanced-for` syntax was introduced by Java 5). We used the batch execution mode of LAMBDAFICATOR.

We recorded several metrics for each project. To measure the applicability, we count how many code fragments met the refactoring preconditions and thus can be refactored by LAMBDAFICATOR. We also report the number of times each precondition fails.

To measure the value of ANONYMOUSTOLAMBDA refactoring on the code quality, we report the reduction in lines of code. However, simply comparing reduced SLOC could produce misleading results: some programmers use different formatting options (e.g., curly braces on new lines) or using name identifiers of different length. To eliminate these effects, we

also calculated the percentage decrease in the number of Abstract Syntax Tree (AST) nodes:

$\frac{nodesInAIC - nodesInResultingLambda}{nodesInAIC} * 100$

The value of ForLoopToFunctional refactoring on code quality can be best gauged if we compared our refactored code with the code refactored to use `Threads` and `Runnable`. However, in the absence of a refactoring that converts sequential loops to parallel loops with `Threads`, we could not measure this. By comparing Fig. 1(b) and Fig. 1(c), it is obvious that our refactoring significantly reduces the code size and makes the code more readable. Instead, we report how many operations were inferred from the original loops. We also report usage of individual operations and of chained operations, along with the average length of chains.

To measure the effort that a programmer would spend to refactor the project manually, we report the number of files that are modified by the refactoring. We also report the number of modified SLOC, as counted by the standard diff tool from Unix (we configured the tool to ignore white spaces), and the number of special cases that require extra attention. These numbers estimate the programmer effort that is saved when refactoring with LambdaFicator.

To measure the accuracy of LambdaFicator we will use standard metrics from information retrieval, such as *precision* and *recall*. In our case, precision measures how many of the refactorings performed with LambdaFicator match the best, most fine-grained refactorings that an expert can apply. Recall measures how many of the possible refactorings LambdaFicator successfully performed. Since our corpus is large, we sampled 10% of the `for` loops in the original code. To create the ground truth set, we carefully analyzed for each input construct whether the refactoring can be applied. Thus, we created two sets: $ShouldApply_{man}$, and $ShouldNotApply_{man}$ which contain tuples of the form $\langle in, out \rangle$, where *in* represents the input code, and *out* represents the expected refactored code. Then we ran LambdaFicator on the sampled inputs and created two sets: $Applied_{tool}$ and $NotApplied_{tool}$ which also contains tuples of the same form. Note that in $ShouldNotApply_{man}$ and $NotApplied_{tool}$ the tuples are of the form $\langle in, in \rangle$. We define the *Precise* transformations set as:

$Precise = ShouldApply_{man} \cap Applied_{tool}$,

and the *Imprecise* transformations set as:

$Imprecise = \{\langle in, out \rangle \mid \langle in, out \rangle \in Applied_{tool}, \exists out'.out' != out, \langle in, out' \rangle \in ShouldApply_{man}\}$,

and the missed transformations set:

$Missed = ShouldApply_{man} \setminus Applied_{tool}$

We define precision and recall using set cardinality:

$$Precision = \frac{|Precise|}{|Precise| + |Imprecise|} \quad (1)$$

$$Recall = \frac{|Precise|}{|Precise| + |Missed|} \quad (2)$$

To answer the safety question, we ran extensive test suites (4360 tests) before and after all refactorings that we applied on our corpus. We used projects that had extensive tests[1] to help us confirm that the refactorings did not break the systems. The refactorings did not cause any new failures. We also carefully inspected 10% of all refactored elements, and found that all refactorings we sampled preserved semantics.

[1]jEdit does not contain JUnit tests, and for Hadoop/Tomcat none of the tests run under Java 8

5.2 Results

5.2.1 AnonymousToLambda Refactoring

Figure 6 tabulates results for the AnonymousToLambda.

Applicability: Column 3 shows that Java projects use AIC extensively. 55% of all AIC met the refactoring preconditions. AIC that were not converted failed a precondition, shown in columns P1-P4. The most commonly failed preconditions were not instantiating from an interface (37% of AIC) and declaring multiple methods or fields (18% of AIC). Notice that the percent of AIC that could be converted in each project had a high variability, with a standard deviation of 23.6%. Projects like ANTLRWorks (78% converted) and Apache Ivy (82% converted) heavily utilized functional interfaces, while others like JUnit (13% converted) and Hadoop (26% converted) did not. This shows that lambda expressions do not provide a complete replacement for AIC, but still have a high level of applicability.

Value: The data shows that our AnonymousToLambda refactoring produces more concise code. The SLOC Red. column shows the reduction in source lines of code after the refactoring. In total, the AnonymousToLambda refactoring reduced SLOC by 2213 lines, or 3.19 lines per refactoring. The AST Red. column shows that the refactoring reduces AST nodes by 52.8% on average. The highest reductions per refactoring were obtained in the Hadoop project. We examined this project and found that many refactored AIC contained one method with a single return statement that was significantly compacted after the refactoring.

Effort: A total of 274 files were modified by applying the refactoring, an average of 30.4 files per project (see Files Mod. column). Notice that the refactorings were not strongly clustered, with an average of just 2.5 refactorings per file. Had the programmer refactored manually, she would have had to jump across many files.

In total, the refactoring modified 3707 SLOC, with an average of 411.9 SLOC per project (see SLOC Mod. column). Notice that many of these changes are non-trivial. Columns S1-S4 show special cases, such as disambiguating shadowed variables or inferring the types for lambda expressions, which require special attention from the programmer. These special cases represent 29% of all refactorings.

By contrast, the automated AnonymousToLambda refactoring takes an average of just 11.3 seconds per project. These results show that LambdaFicator can save a lot of programmer effort.

Accuracy: By manually checking 10% of the AIC we found that LambdaFicator transformed all the AIC in a precise manner. We also found that LambdaFicator didn't miss any refactoring. The precission and the recall were 100% for the AnonymousToLambda refactoring.

5.2.2 ForLoopToFunctional Refactoring

Figure 7 shows the results for the ForLoopToFunctional refactoring. We omitted the results for the projects that did not have any enhanced `for` loops.

Applicability: On average, LambdaFicator successfully refactored 46.02% of the enhanced `for` loops present in our code corpus. The precondition that was not met most often (41% of the time) was P1. This is due to the fact that

			Applicability							Value						Effort		
Project	SLOC	Tests	#AIC	Conv.	Conv. [%]	P1	P2	P3	P4	S1	S2	S3	S4	SLOC Red.	AST Red.	Files Mod.	SLOC Mod.	Time [s]
ANTLR+Works v1.5.1	97795	674	151	118	78%	27	14	4	0	2	8	0	0	243	52.6%	41	513	9
Ant v1.8.4	129053	1637	104	38	37%	50	22	0	0	14	0	1	1	125	52.3%	21	203	15
Ivy v2.3.0	68193	944	107	88	82%	11	8	2	0	10	1	0	5	203	48.4%	27	387	9
Tomcat v7.0.29	221321	-	153	87	57%	53	32	1	0	29	3	57	0	368	52.4%	37	584	17
FindBugs v2.0.2	121988	163	338	174	51%	124	90	0	0	18	0	1	0	507	44.6%	53	853	17
FitNesse v20121220	63188	250	102	36	35%	61	23	0	0	11	0	0	18	99	45.8%	19	179	8
Hadoop v0.20.3	307016	-	31	8	26%	23	5	0	0	7	0	2	0	40	66.2%	8	60	8
jEdit v5	114899	0	201	134	67%	59	16	2	0	2	4	0	0	593	53.4%	61	873	13
JUnit v4.11	10443	692	76	10	13%	63	23	0	0	7	0	0	1	35	59.5%	7	55	6
Total	1133896	4360	1263	693	55%	471	233	9	0	100	16	61	25	2213	52.8%	274	3707	102

Figure 6: Lambda Conversion Results. Failed Preconditions - P1: Not an interface, P2: Has multiple methods or has field, P3: Has reference to 'this' or 'super', P4: Has recursive call. Special Cases - S1: Block and return were omitted, S2: Has shadowed variable, S3: Has ambiguous method overload, S4: Has assignment to supertype. AST Red. - Average percent decrease in AST nodes per conversion.

	Applicability							Value									Effort		
Project	*#for loops*	*%Refactored*	P1	P2	P3	P4	P5	P6	*#forEach*	*#anyMatch*	*#noneMatch*	*#reduce*	*#map*	*#filter*	*#Singleton*	*#Chains*	*Avg chain length*	*#Files Mod.*	*#SLOC Mod.*	*Time [s]*
ANTLR	589	60.10%	78	15	102	44	56	18	322	17	4	10	117	93	216	138	2.51	140	2293	16
FitNesse	242	48.34%	79	28	22	1	25	0	98	5	10	4	35	17	83	34	2.52	83	3336	8
Hadoop	1772	28.04%	800	600	476	130	77	27	453	9	4	30	198	65	330	167	2.56	196	2654	36
Tomcat	425	44.70%	149	42	88	29	45	3	173	1	15	1	151	22	129	61	3.83	83	1283	16
jEdit	190	32.10%	82	10	51	13	22	6	61	0	0	0	23	14	37	24	2.54	33	401	7
FindBugs	1075	40.09%	359	131	226	79	114	32	402	14	3	12	140	82	295	136	2.63	182	2346	16
jUnit	109	54.12%	27	17	9	1	9	0	48	6	3	2	12	5	48	11	2.54	32	277	4
Total	4402	46.02%	1574	843	974	297	348	86	1557	52	39	59	676	298	1138	571	2.72	717	12313	97

Figure 7: ForLoopToFunctional conversion results. Failed Preconditions: P1: Not iterating over `Collection`, P2: Throwing exception, P3: References to outer non-final variables, P4: Contains `break`, P5: Contains `return` P6: Contains labeled `continue` #Singleton shows the number of refactorings formed that have only one operator #Chains shows the number of refactorings composed of more than one operator.

the `stream` method is available in `Collection` and not in `array`. The second most frequent failed precondition was P3. It checks for outer non-effectively-final local variables referenced from within the loop. This result makes sense: code written in an imperative style would inevitably have side effects incompatible with the functional style. LambdaFicator still managed to refactor over 46% of the loops.

Value: The data shows the most common operation is `forEach`. This makes sense since this operation has the most relaxed preconditions. However, operations other than `forEach` represent 42% of all operations. These operations have descriptive names which convey intent more explicitly than the original code. Additionally, 33% of the operations were chained, with an average chain length of 2.72. These convey intent with a finer granularity than in the original code.

Effort: The data shows that on average per project the developer would need to edit 102 files and change 1759 lines. We find that the refactoring are spread across files, with a rate of 2.3 loops refactored per file. The analysis is non-trivial. For example, column P3 shows that the programmer would have had to find 974 references to non-effectively final variables and determine if they can be converted to a reducer. The programmer would have to reason about how to chain lazy operators with eager ones and make values available in a pipeline fashion 33% of the time. LambdaFicator is fast. Even on the largest project, with over 300K lines of code, LambdaFicator can determine in half a minute if the refactorings are safe and apply them on the whole project.

Accuracy: When inspecting 10% of the loops LambdaFicator has refactored, we found that it had a precision of 0.9. The cases when the refactoring is not precise enough are due to the fact that tool always tries to build the most fine grained chain. The inspection showed that in 10% of the cases a professional might chain operations in a different way. We didn't find any refactoring that changed semantics - only refactorings that a human might do differently (e.g., merge two subsequent `maps`). By considering also the loops that were not transformed by LambdaFicator we found that the recall of ForLoopToFunctional is 0.92. The misses are either due to operations that could be used – like `min` and `max` (and are trivial extensions to LambdaFicator), or the restrictiveness of the `reduce` heuristic which can be improved. Overall, we found that most of the time LambdaFicator infers the most precise chain a human could infer.

5.3 Threats to Validity

Construct Validity: Can we measure development effort by counting size and clustering of changes? Ideally, we would have observed developers while they refactor. However, given the cutting edge nature of lambda expressions

in Java, we could not find such developers. Thus, we chose to use indirect metrics. Notice that refactoring changes are non-trivial, and require reasoning about preconditions (see Sec. 3.3, 4.2). The number of files changed is relevant because it shows refactorings are widespread, so a developer would spend time searching for refactoring opportunities.

Internal Validity: How did we mitigate bias during manual inspection? We randomly sampled from the set of `for` loops and AICs in each project. Before looking at the results from LambdaFicator, we classified the sampled exemplars as refactorable or unrefactorable, and we manually performed the refactorings, thus creating the ground truth. The authors are expert users of Java lambda expressions.

External Validity: Do our results generalize? We choose a diverse corpus of 9 widely used open source projects totaling over 1 million SLOC. They are developed by very different entities, by large organizations or researchers, and covering domains such as GUIs, compute-intensive servers, IDEs, testing tools, etc.

Reliability: Is our evaluation reliable? The corpus is available on our webpage. LambdaFicator is open-source: `http://refactoring.info/tools/LambdaFicator/`.

6. DISCUSSION

With respect to the type for the arguments of the lambda expression, LambdaFicator has two modes of operation: it can make the types of the arguments explicit, or it can omit them since they can be inferred by the compiler. While omitting the types makes the code more succinct, there is value in making the types explicit. Pankratius et al. [21] find that explicit type declarations results in code that is more readable and maintainable.

Our ForLoopToFunctional refactoring takes as input an `enhanced-for` loop. We have developed a refactoring (that ships with the official release of the Eclipse IDE [4]) to convert old style `for` loops into `enhanced-for`. This refactoring is an enabler for ForLoopToFunctional refactoring.

One could ask whether it is profitable to convert Java sequential loops to the functional form that can execute in parallel. In our previous work [10], we have shown that dynamically-balanced loop parallelism (similar with the one supported by Java 8 collections) produces close to linear speedup all the way up to 24 cores. The official performance tests accompanying the Java 8 collections also show solid speedups up to 20x on 32-way parallelism. From the code produced by the ForLoopToFunctional refactoring, one simply has to add the `parallelStream` to enable parallel execution. To determine whether it is safe to run the loop iterations in parallel, we refer the reader to our static data-race detector [24] specialized for Java loops.

7. RELATED WORK

Several empirical studies describe the advantages of using functional features in imperative programming languages. Okur and Dig [19] found 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 enablers for parallelism. Pankratius et al. [21] show that programmers employ a mix of functional and imperative styles when writing parallel applications.

Ericksen [12] reports on Scala's mix of functional/imperative style used in large commercial applications like Twitter. Hundt [14] compared C++/Java/Go/Scala implementations using Scala's equivalent `map, filter, forEach` operations. He reports that Scala's concise notation and powerful language features allowed for the least complex code. LambdaFicator enables developers to use these powerful features. Oliveira and Cook [20] present methods and advantages of using a functional style of programming and ways of combining it with the object-oriented style in order to reduce abstraction overload.

Prokopec et al. [23] show that higher-order functions can simplify the programming interface of data-flow abstractions in the context of parallel `Collection`s. Odersky [18] shows how Scala's collections framework, which has the equivalent functional operations coming in Java 8, simplifies the use of parallelism when iterating over `Colection`s. Prokopec et al. [22] present and evaluate a framework to build parallel collections (similar with Java 8) and report good speedups.

LambdaFicator enables Java programmers to benefit from all the advantages reported in these studies.

Recently, there is a surge of interest [15–17,25] in supporting refactorings in functional languages. However, LambdaFicator is the first refactoring tool that helps programmers retrofit functional features into an imperative program.

8. CONCLUSIONS

This paper presents the analysis, design, implementation, and evaluation of two refactorings that enable the Java programmer to retrofit existing imperative code with two functional features: lambda expressions and functional operations. Our empirical data shows that AIC and `for` loops compatible with functional style are pervasive. Thus, Java programmers have ample opportunities to use functional language features.

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.

9. ACKNOWLEDGMENTS

Alex Gyori and Lyle Franklin conducted this research while being undergrad summer interns at the Information Trust Institute at UIUC, under the supervision of Danny Dig. The authors would like to thank Marius Minea, Darko Marinov, Milos Gligoric, Stas Negara, Mihai Codoban, Caius Brindescu, Yu Lin, Krzysztof Zienkiewicz, Nick Chen, Qingzhou Lou, and the anonymous reviewers for valuable feedback on earlier drafts of this paper. This research is partly funded through NSF CCF-1213091 and CCF-1219027 grants, a gift grant from Intel, and the Intel-Illinois Center for Parallelism at UIUC. The Center is sponsored by the Intel Corporation.

10. REFERENCES

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

[2] Intel Thread Building Blocks For Open Source. http://threadingbuildingblocks.org/.

[3] Parallel LINQ. http://msdn.microsoft.com/en-us/library/dd460688.aspx.

[4] Quick-assist to convert for to enhanced-for. http://archive.eclipse.org/eclipse/downloads/drops/R-3.1-200506271435/eclipse-news-all.html#part2.

[5] SLOCCount. http://www.dwheeler.com/sloccount/.

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

[7] State of the lambda: Collections edition. http://cr.openjdk.java.net/~briangoetz/lambda/collections-overview.html.

[8] Task Parallel Library. http://msdn.microsoft.com/en-us/library/dd537609.aspx.

[9] The Scala Programming Language. http://www.scala-lang.org/.

[10] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM.

[11] T. P. Brian Goetz. *Java concurrency in practice*. Addison-Wesley, 2006.

[12] M. Eriksen. Scaling Scala at Twitter. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 8:1–8:1, New York, NY, USA, 2010. ACM.

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

[14] R. Hundt. Loop Recognition in C++/Java/Go/Scala. In *Proceedings of Scala Days*, 2011.

[15] D. Y. Lee. A case study on refactoring in Haskell programs. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1164–1166, New York, NY, USA, 2011. ACM.

[16] H. Li. *Refactoring Haskell Programs*. PhD thesis, University of Kent, 2006.

[17] H. Li and S. Thompson. Comparative Study of Refactoring Haskell and Erlang Programs. In *Sixth IEEE International Workshop on Source Code Analysis and Manipulation, 2006. SCAM '06.*, pages 197–206, 2006.

[18] M. Odersky. Future-proofing collections: From mutable to persistent to parallel. In *Proceedings of the 20th International Conference on Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011.

[19] S. Okur and D. Dig. How do developers use parallel libraries? In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 54:1–54:11, New York, NY, USA, 2012. ACM.

[20] B. Oliveira and W. Cook. Extensibility for the masses. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 2–27. Springer Berlin Heidelberg, 2012.

[21] V. Pankratius, F. Schmidt, and G. Garretón. Combining functional and imperative programming for multicore software: an empirical study evaluating Scala and Java. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 123–133, Piscataway, NJ, USA, 2012. IEEE Press.

[22] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A generic parallel collection framework. In *Euro-Par 2011 Parallel Processing*, volume 6853 of *Lecture Notes in Computer Science*, pages 136 – 147. Springer Berlin Heidelberg, 2011.

[23] A. Prokopec, H. Miller, T. Schlatter, P. Haller, and M. Odersky. FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction. In *25th International Workshop on Languages and Compilers for Parallel Computing*, pages 158–173, 2012.

[24] C. Radoi and D. Dig. Practical Static Race Detection for Java Parallel Loops. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA'13. To Appear, 2013.

[25] S. Thompson. Refactoring functional programs. In *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 331–357. Springer Berlin Heidelberg, 2005.