Bridging the Gap Between Functional and Imperative Programming through Refactoring

Alex Gyori

Bachelor Thesis
Presented at
Politehnica University of Timisoara
Romania

Recommended for Acceptance
by the Department of
Computer and Software Engineering
Coadvised by: Danny Dig
And Marius Minea

June 2013
Abstract

We present a technique that shows it is feasible to automatically refactor imperative code to a functional style. Our technique is implemented and integrated in the official release of the NetBeans IDE.

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 designed and implemented LAMBDAFICATOR, a tool which automates a refactoring that converts for loops that iterate over Collections to functional operations that use lambda expressions. In 9 open-source projects we have applied this refactoring 1709 times. 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.

Keywords

refactoring, Java 8, lambda expressions, imperative programming, functional programming
Acknowledgements

I learned a lot from many people, directly or indirectly, while working to complete this research. I apologize in advance to those who I will not mention here due to lack of space or the bad memory I have. You all have my deepest gratitude for your support.

I would like to thank my advisor, Marius Minea, for providing the support for completing this thesis throughout my bachelor studies. He was my first contact with research and the experience and discussions we had determined me to head towards a career in research. I was extremely fortunate to have his support and advice when applied for research internships, summer/winter schools, and PhD programs. I would like to thank him for his patience even in the moments when I asked the most trivial questions - he created a truly great learning environment in his research lab. I appreciated the freedom Marius gave me in my research, but also the many hours he spent with me when I was stuck or needed guidance. I thank Marius for believing in me and encouraging me continuously and I hope that I rose to at least a fraction of his expectations. I thank Marius for his critical opinions, those helped me the most; I wouldn’t have made it without his guidance.

I would like to thank Danny Dig, who advised me during this project, for his guidance during my research internship at Illinois, for showing high expectations and being critical with me - this motivated me to succeed. Danny gave me the idea for this research and I am very thankful to him. I would like to thank Danny for taking me to Illinois, I hope I rose to his expectations. Danny was a truly great advisor; our discussions about the role of research and our vision about it has formed me as a researcher. He put me in touch with people that I wouldn’t have dreamed to meet while reading their books, articles, etc. Thanks Danny for teaching me to dream big and taking me to ICSE. I hope that our collaboration in the future will continue to be as fruitful as it has been so far.

I would also like to thank Cosmin Rădoi for his advice and guidance during my stay at Illinois - he made me explore the field of race detection, and I came to really enjoy it; our talks on the topic were great. Lyle Franklin and I worked together during the internship while being roommates; I would like to thank him for the advice he gave me while working towards our paper submission and for the insightful discussions that we had. Thanks to Lyle also for his advice helping me adapt to the US.

Thanks to Petru Mihancea and Dan Cosma with whom I spent a lot of time doing research as an undergrad. They helped me with both personal and professional advice during my undergrad studies. Without them, the research environment wouldn’t have been the same. Thanks go to Casandra for her energy; the research lab wouldn’t have been as vivid as it was without her - she always entertained us.

I would like to thank Prof. Radu Marinescu for being motivational, providing real world advice to his students, and for his well thought irony. Also thanks Radu for his advice before going to ICSE and in choosing grad schools. I would like to thank also Prof. Emilia Petrişor for constantly encouraging me and providing advice; her classes always gave me food for thought and energy.
Thanks to all my friends with whom I shared a lot of time in and out of school. I would like to thank Matyi and Cristi for the insightful discussions we always ended up having. I would like to thank Cătălina, Sergiu, Bogdan, Mihai, and Cosmin - my colleagues from the Microsoft Student Partners program - for creating a very fruitful environment that stimulated my growth in software engineering. Thanks to Radu Stoia for coaching me as a debater, this sparked my interest towards an academic career.

I would like to thank Cristina for her patience and the understanding she showed for me and my work; she has been my closest companion in my journey for half a decade already. She constantly asked the meta-questions about my dreams and taught me to be humble and understanding. Last but not least, I would like to thank my parents for their patience and support during my studies. They always supported me and trusted me; that was reassuring.
Being tall helps you become a star in basketball, but how many of us have a shot at playing in the NBA? It’s not about what you’re born with; it’s about what you do.
- Seth Godin
## Contents

Abstract ................................. iii  
Acknowledgements ........................ iv  
List of Figures ........................... ix  

1 Introduction .......................... 1  
1.1 Introduction to Lambda Expressions .... 1  
1.2 Motivating Example .................... 2  
1.3 Challenges .......................... 3  
1.4 Contributions ........................ 4  
1.5 Structure ........................... 5  

2 Related Work .......................... 6  
2.1 Refactoring Research ................... 6  
2.2 Empirical Studies ..................... 7  
2.3 Functional Programming Support ...... 7  

3 ForLoopToFunctional refactoring ........ 8  
3.1 Functional Operations introduced in Java 8 .... 8  
3.2 Working Examples ..................... 9  
3.3 Preconditions ........................ 11  
3.4 Functional Operations Patterns ........ 11  
3.4.1 Filter .......................... 12  
3.4.2 Map ............................ 12  
3.4.3 Reduce .......................... 13  
3.4.4 AnyMatch and NoneMatch .......... 13  
3.4.5 ForEach .......................... 13  
3.5 Algorithm ........................... 14  
3.6 Discussion ........................... 19  
3.7 LambdaFicator usage scenarios ......... 21  

4 Implementation Overview ................. 23  
4.1 Abstract Syntax Trees .................. 23  
4.2 General Architecture ................... 24  
4.3 Implementation Details ............... 24  
4.4 NetBeans Integration .................. 25
# List of Figures

1.1 Comparison between methods of parallelization in Java. ................ 3  
3.1 Method signatures of Java 8 operations. ......................... 8  
3.2 Example of `ForLoopToFunctional` refactoring. .................. 10  
3.3 Control Flow pattern for `filter` ............................ 12  
3.4 Control Flow pattern for `map` ............................... 13  
3.5 Control Flow pattern for `anyMatch` and `noneMatch` ......... 14  
3.6 Refactoring `enhancedFor` loops to operations on `Stream<T>`  .... 15  
3.7 Showing how the algorithm builds the chaining of operations for example 3 in Fig. 3.2. .......................................................... 18  
3.8 Data flow graph for loop in example 3 in Fig. 3.2. .............. 19  
3.9 Data flow graph for loop with no merging needed. .............. 20  
3.10 Data flow graph for illustrating two upstream dependencies. .... 21  
3.11 Data flow graph for loop with total merging. ................. 21  
3.12 `LambdaFicator` performs the `ForLoopToFunctional` refactoring in batch mode. ................................................................. 22  
3.13 `LambdaFicator` performs the refactoring in `Quick Hint` mode. .... 22  
4.1 Simplified abstract syntax tree of second loop from Fig. 3.2 ... 23  
4.2 High-Level Architecture of `LambdaFicator` ........................ 24  
5.1 `ForLoopToFunctional` conversion results. .......................... 28
Chapter 1

Introduction

Shorter versions of this work were presented at ICSE’13 and FSE’13 on the tool demo and research track respectively [17,20]. Those papers contain also supplemental work which is not part of this dissertation.

1.1 Introduction to Lambda Expressions

A lambda expression (also called an anonymous function) is a function without a name identifier. For example, \((\text{int } x, \text{ int } y) \rightarrow x + y\) 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 have supported lambda expressions from the first release. Others, like C# (v 3.0), C++ 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 [4] 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 Java 8 collections [7] provide internal iterators [18] that take a lambda expression as an argument. For example, \text{filter} takes a predicate expression and removes elements of a collection based on the predicate, \text{map} applies a function to the elements of a collection creating a new collection, \text{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.
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:

```java
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. 1.1).

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. This style makes explicit the selection of `BLUE` blocks, through the `filter` operation, and the fact that each of them are colored in `RED`:

```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 [10], `FluentIterable` [1] in Guava Google Libraries, or Microsoft PLINQ library [32].

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.

### 1.2 Motivating Example

We illustrate the motivation behind introducing lambda expressions for parallelism. Fig. 1.1(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 1.1(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 hierarchy than all other rules, so the first worker thread will spend a much longer time computing. Because the work is not
Figure 1.1: Comparison between methods of parallelization in Java.

split evenly between the worker threads, the computation would take longer. This problem is referred in literature as dynamic load balancing [19].

By taking advantage of the parallel functional operators introduced in Java 8, the programmer can refactor the sequential loop using LAMBDAFICATOR. Fig 1.1(c) shows the final code, which is much more succinct then 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.

1.3 Challenges

To get all the benefits of lambda expressions and internal iterators, the Java programmer must refactor for loops into internal iterators. However, these refactorings are tedious. Java projects are riddled with many external iterators. For example,
ANTLRWorks, a medium-sized open-source project (42K non-blank, non-comment source lines – SLOC) contains 261 external iterators. Refactoring ANTLRWorks by hand requires changing 881 SLOC.

When converting for loops to internal iterators there are several challenges. First, there are many ways to split loop statements into pipelined operations. The developer has to first decide how to split the code between functional operations and what operation the selected code maps to. This is not always obvious. Ideally the programmer would pick the most fine-grained to enable precise control of parallelism and make the intent of statements more explicit.

Second, the programmer must reason about control flow statements like break, continue, return and choose the appropriate operations. These are constructs inherent to loops and the imperative style; integrating their behavior in a functional style is not straightforward. Sometimes code containing such constructs cannot be refactored, but LambdaFicator can refactor easily several cases through the heuristics we have developed.

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 a lambda expression is now local to that lambda. This requires identifying the variables that need to be passed through the pipeline for the next operation to use. LambdaFicator analyzes the data flow and decides what variable to pass down the stream and how to split the original code so that it is able to pass everything needed to the next functional operation.

Fourth, the programmer must verify there are no side effects on local variables defined outside of the lambda. This is due to Java’s restriction that you cannot refer non-effectively-final local variables from within lambda expressions. Some of the side effects can be propagated through a reduce operation and LambdaFicator does the analysis needed to determine when this is feasible.

Fifth, the programmer must reason about the nature (e.g., eager vs. lazy) of operations in order to preserve the semantics of the original code. A for loop is an eager construct and the refactoring has to preserve this characteristic.

### 1.4 Contributions

We present a technique that shows it is feasible to automatically refactor imperative code to a functional style. Our technique is implemented and integrated in the NetBeans IDE.

To solve the challenges described, we have designed, implemented, and evaluated LambdaFicator, the first refactoring tool to automate the task of retrofitting functional features into imperative code. Our refactoring, ForLoopToFunctional, replaces for loops with their equivalent chained operations.

This dissertation makes the following contributions:

**Problem:** to the best of our knowledge, we are the first to describe the novel problem of converting imperative code to a functional flavor using lambdas.
**Algorithms:** we have designed the analysis and transformation algorithms to address the challenges that arise when refactoring *for* loops into functional operations. The algorithm accounts for different scoping rules between the old and the new language 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 our refactoring with the official release of the NetBeans IDE.

**Evaluation:** we have evaluated \textsc{LambdaFicator} on 9 open-source projects (totaling almost 1M SLOC), invoking \textsc{ForLoopToFunctional} 1709 times. The results show that the refactoring is widely *applicable*: the refactoring converted 46% of *for* loops. Second, the refactorings are *valuable*: the refactoring infers 2681 operators and 1709 chains thus making the intent of the loop explicit. Third, \textsc{LambdaFicator} is *efficient*: saves the programmer from manually changing 12k SLOC. Fourth, \textsc{LambdaFicator} is *precise*: when executed in batch mode on the whole projects, the tool infers the best operations more than 90% of the time.

1.5 Structure

The rest of this dissertation is organized as follows: In chapter two we present the state of the practice discussing empirical studies, refactoring research, and functional programming research that is directly related to our work. In the third chapter we present the detailed design of our algorithm, the preconditions to apply this refactoring, and the usage scenarios. Chapter four contains an overview of the implementation and some low level details. In chapter five we present the framework in which we evaluated our tool and we provide interpretation for our findings. We conclude in chapter six, by discussing impact and future work.
Chapter 2

Related Work

We group related work in three areas (2.1) Refactoring Research, (2.2) Empirical Studies, and (2.3) Functional Programming tools.

2.1 Refactoring Research

There is a lot of research in the area of refactoring for parallelism. The research of Dig et al. [15] on refactoring for parallelism contains a tool, Relooper, that performs a related refactoring to ParallelArray. This tool performs static analysis to ensure loops are thread safe and uses ParallelArray to parallelize operations in the loop. LAMBDAFICATOR could benefit from the automatic thread safety analysis of Relooper. If these checks were made, an option could be introduced to include the parallel() method call, as discussed in Ch. 3.5.

LAMBDAFICATOR improves on Relooper by providing increased applicability and increased readability through operator chaining. By allowing more relaxed preconditions and better operator inference heuristics, our FOR LOOP TO FUNCTIONAL refactoring is more widely applicable. LAMBDAFICATOR also takes advantage of built-in language features, such as lambda expressions, rather than external libraries, resulting in improved readability. Our tool can also infer chaining of operators, permitting the refactoring of more complex loops, such as iterations involving multiple control flow paths.

Chen and Johnson [11] describe JFlow, a tool that enables developers to use Flow-Based parallelism. LAMBDAFICATOR is similar because it enables developers to use parallelism; the difference is that LAMBDAFICATOR uses standard Java language libraries as opposed to JFlow. Dig et al. [14] provide a set of refactorings for thread safety utilizing thread-safe constructs from java.util.concurrent. All these are enablers for parallelism, but the mentioned research focuses on thread safety, leaving the transformations for parallelism to developer. We focus on enabling the developer to parallelize the code through the transformations we provide. While our approach is different we all aim at enabling parallelization of written code.
Recently, there is a surge of interest in supporting refactorings in functional languages [27, 28, 30, 39]. However, we are the first ones to help programmers retrofit functional features into an imperative program.

2.2 Empirical Studies

Ericksen [16] reports on Scala’s mix of functional/imperative style used in large commercial applications like Twitter. Hundt [24] 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 while achieving parallelism. Oliveira and Cook [33] 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. Dig et al. [13] report in a survey of common refactorings to introduce parallelism that loop parallelism is one of the frequent changes employed by developers to increase throughput. We see LambdaFicator as an enabler for parallelism.

Prokopec et al. [37] show that higher-order functions can simplify the programming interface of data-flow abstractions in the context of parallel Collections. Odersky [31] shows how Scala’s collections framework, which has the equivalent functional operations coming in Java 8, simplifies the use of parallelism when iterating over Collections. Prokopec et al. [36] present and evaluate a framework to build parallel collections (similar with Java 8) and report good speedups.

Hinsen [22] finds that the functional programming style can make programs more easily parallelizable. He also determines that using the style requires more effort. We address this by providing automatic transformations that retrofit the functional style into imperative Java 8 programs.

2.3 Functional Programming Support

Davis and Kiczales [12] 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. Pareja et al. [35] presents an IDE for functional programming which employs visualization techniques in order to facilitate functional programming understanding. This shows that understandability problems can arise in functional style programs, and there is a need for tools that facilitate this. While we do not address this problem, we acknowledge that for developers not accustomed with the functional style understandability issues may arise; our refactored code can benefit from such techniques.

Li, Reinke, and Thomson [29] explore the prospects of refactoring functional programs. They have built a tool that shows the utility of refactorings in the functional realm. While our tool is not aimed towards functional languages, we show the utility of retrofitting functional language constructs in object oriented languages, taking advantage of lambda expressions.
Chapter 3
ForLoopToFunctional refactoring

3.1 Functional Operations introduced in Java 8

Java 8 introduces interface Stream<T> [7]. This interface exposes methods that enable internal iteration. The methods introduced in it are similar in form and semantics to functional programming-style list operations. LAMBDATICATOR currently supports map, filter, reduce, forEach, anyMatch and noneMatch. We will refer to these methods as functional operations. We show the method signatures of these functional operations in Fig. 3.1.

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. map takes a lambda expression that transforms each element of a collection. This transformation is generally from type T to type R (but the two can be the same under some circumstances). The map operation returns a new Stream where each element is mapped from the original Stream. This enables chaining operations together in a pipeline fashion.

The filter operation returns a new Stream<T> containing only the elements that satisfy the Predicate. Notice that the filter operation returns elements from the original Stream.

- 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 3.1: Method signatures of Java 8 operations.
The last four operations are *eager operations*, i.e., they execute when called. Notice that they do not return a Stream, therefore these operations can only be the last in a chain of operations.

The `reduce` operation takes a binary operator and an accumulator. It applies the `BinaryOperator` to each element of the stream and the current value of the accumulator; the resulting value is passed forward. For example below you can find an example of `reduce` used to sum everything together:

\[
\text{Stream.reduce ( sum, 0, [1, 2, 3] ) = (((0 + 1) + 2) + 3)}
\]

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` and return `true`. In contrast, `noneMatch` will return `false` and stop processing the stream whenever the `Predicate` is matched.

The `forEach` operation consumes elements from the `Stream` eagerly. It applies a lambda expression with a `void` return type to each element of the `Stream`.

Streams do not store intermediate values; they carry values from a source through a chain of operations. This means chaining such operation is cheap in terms of storage. It is important to note that operations on streams are functional in nature and they do not modify the underlying data source, e.g., `Collection`.

### 3.2 Working Examples

We illustrate three examples of the `ForLoopToFunctional` refactoring in Fig. 3.2. 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` argument. 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`. `LAMBDAFICATOR` analyzes the control flow and determines this fact easily.

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. `LAMBDAFICATOR` externalizes the imperative style, in-place, mutation on the local variable. The local variable is updated only with the final value, when the reducer finishes. In order to infer this, `LAMBDAFICATOR` has to analyze data flow and detect writes on local variables; then infer if the mutation can be externalized.
class GrammarEngineImpl implements GrammarEngine {
    boolean isEngineExisting(String grammarName) {
        for (GrammarEngine e : importedEngines) {
            if (e.getGrammarName() == null) continue;
            if (e.getGrammarName().equals(grammarName))
                return true;
        }
        return false;
    }
}

class EditorGutterColumnManager {
    int getNumberOfErrors() {
        int count = 0;
        for (ElementRule rule : getRules()) {
            count += rule.getErrors().size();
        }
        return count;
    }
}

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).isStarted())
                result.add(entry.getValue());
        });
    }
}

(a)

(b)

Figure 3.2: 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.

In this transformation we have 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 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 while the last statement needs a reference to the entry object. Notice the variable scoping change in this case: the entry is now local to the first lambda expression in the filter. The next filter needs a ClassLoader object so in between the filters we would have a map. But at this point we have no means to pass down the entry object to the last operation; this would not be available from the last filter, therefore, these two operations cannot be chained. LambdaFicator performs this kind of variable availability analysis to ensure the functional operations are chained correctly. Identifying when and how to chain operations is nontrivial. LambdaFicator merges operations together in order to ensure the operations can be chained.
3.3 Preconditions

Although many enhanced for loops, like the ones you saw previously, 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 cannot generally 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. 3.2. 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 functional operations, not limitations of our tool. LambdaFicator checks these preconditions before applying the ForLoopToFunctional refactoring:

(P1) The enhanced for loop must iterate over an instance of java.util.Collection, rather than an array for example. This constraint is due to the fact that only Collections provide a convenient way to obtain a Stream.

(P2) The body of the initial for loop does not throw checked exceptions. The lambda expression signature used by the functional operations does not have a throws clause; therefore, loops that might throw checked exceptions cannot be refactored. Wrapping the code in a try-catch does not preserve the semantics of the original program.

(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 described in Sec. 3.4.3.

(P4) The body of the initial for loop does not contain any break statements. The semantics of break is inherent to a loop and cannot be handled by chaining operations together, considering the currently available functional operations.

(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.

3.4 Functional Operations Patterns

We have designed a series of patterns to determine the proper functional operation to use. In this section we present in detail what control flow patterns we use in order to determine the suitable functional operation. When we discuss control flow, we refer to the in-loop control flow.
3.4.1 Filter

A filter corresponds to a statement that branches control flow in one empty control flow path and a non empty one, as illustrated in Fig. 3.3. The non-empty one may be further split by other if statements and create more control flow paths. What is important though, is the empty path. A filter selects just the elements that satisfy the predicate, and passes them down the Stream, to be processed. Therefore, if there is an else branch or statements after the if, then all the elements need to be passed downstream in order to be processed, and therefore a filter cannot be used. Notice that if only a continue is present inside the if, the control flow looks exactly like the one for an if with no else branch, therefore we will correctly match such patterns with a filter.

![Figure 3.3: Control Flow pattern for filter](image)

3.4.2 Map

A map corresponds to any statement that does not branch the control flow, as illustrated in Fig. 3.4. A map is defined as a function that takes one object as input and returns another object. More generally, a map corresponds to a block with one entry point and one exit point. This can correspond to one statement or to several that have this property.
3.4.3 Reduce

A reducer corresponds to a statement that writes through a compound assignment a local variable defined outside the loop. A compound assignment can be $+=, -=, *=, /=, %=, |=, &=, ≪=, ≫=$. Because a reducer does not return a Stream, we can use a reduce operation only as a last element in the pipeline. This implies that we can only refactor loops that write to local variables defined outside the loop in their last statement.

3.4.4 AnyMatch and NoneMatch

Both anyMatch and noneMatch represent an if with only a return true or false in it. Because these operations return a boolean literal and therefore have to be the last operation in the chain, only ifs that are the last in the control flow can be refactored. We refactor such code by using the predicate from the if in the anyMatch or noneMatch operation. If the return statement returns true then we use anyMatch with the predicate; otherwise we use noneMatch with the negated predicate.

3.4.5 ForEach

A forEach corresponds to the last statement in the control flow. A forEach cannot have writes to non-effectively final local variables nor return statements.
3.5 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 functional operations to execute as needed, i.e., just before the eager operation. LambdaFicator checks the preconditions and if the preconditions pass it applies the chaining inference algorithm. The following is a high-level view of the chaining inference algorithm:

- Step 1: Break code into potential operations.
- Step 2: Annotate potential operations with variable availability information.
Algorithm for Inferring Operation Chaining

1: function refactorStatements(original)
2:   // step1
3:   prospectiveOperators = breakIntoProspective(original)
4:   // step2
5:   for p in prospectiveOperators do
6:     annotateWithAvailabilityInfo(p)
7:   end for
8:   // step3
9:   composites = mergeIntoComposites(prospectiveOperators)
10:  // step4
11:  chain = asEager(composites.getLast())
12:  // Iterate in reverse order, reverse CF order
13:  for prospective in composites do
14:    chain.prepend(asLazy(prospective));
15:  end for
16:  return prependNeededStatements(chain);
17: end function

18: function mergeIntoComposites(annotated)
19:   // iterate over the list from end to start,
20:   // i.e., reverse Control Flow order
21:   for current, prev in annotated do
22:     if ! areComposable(current, prev) then
23:       Merge prev with current into prev
24:     end if
25:   end for
26: end function

Figure 3.6: Refactoring enhancedFor loops to operations on Stream<T>.

- Step 3: Merge operations in order to maintain access to needed references.
- Step 4: Chain the operations.

Fig. 3.6 shows an outline of the algorithm in pseudocode.

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 (except 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 ($O \cdot O'$), 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 $O'$ ($O \cdot O'$) iff

$$|NV_{O'}| = 1 \text{ and } NV_{O'} \subseteq (AV_{O} \cup NV_{O})$$

If the algorithm cannot chain two operations, it applies a merging strategy. If the two operations are prospective map operations then it merges the two operations. Otherwise it merges all previously built operations into a single operation to ensure variable availability. When merging an eager operation with another prospective operation the resulting operation is a prospective eager operation. When merging operations $O$ and $O'$ into $O''$, the algorithm computes the availability sets as:

$$AV_{O''} = AV_{O} \cup AV_{O'}$$

$$NV_{O''} = \{NV_{O} \cup NV_{O'}\} \setminus AV_{O''}$$
This is a fixed point algorithm. It keeps iterating over the list of prospective operations until all operations can be chained. Notice that the algorithm always stops; at each step it either merges operations or it’s done. The number of loop statements is finite, therefore the number of prospective operations is finite; at each step the number of prospective operations either decreases (through merging) or the algorithm is done because all prospective operations can be chained.

In step 4 \textsc{LambdaFicator} determines the correct \textit{eager operation}. It might also be the case that there is no such operation. Due to merging, the \textit{reduce} operation might not be suitable anymore. Afterwards it builds the chain from end to start by prepending the operations.

Finally, \textsc{LambdaFicator} prepends the expression that returns the \textit{Stream} to the chain. If the last operation is a \textit{reduce}, \textit{anyMatch}, or \textit{noneMatch}, \textsc{LambdaFicator} might need to assign the result to a variable or return it directly.

Next, we demonstrate the algorithm by applying it to example 3 from Fig. 3.2. \textsc{LambdaFicator} first checks that all the preconditions are met. In this case no \texttt{Exception} is thrown from the loop, and there are no \texttt{return}, \texttt{break}, or \texttt{continue} statements. \textsc{LambdaFicator} finds a reference to \texttt{result} but it is able to determine that the variable is only initialized; therefore, it is \textit{effectively final} and usable from a lambda expression. We show in Fig. 3.7 a step by step run of the algorithm from example 3.

The chaining algorithm begins by breaking the list of original statements into \textit{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. The first statement is an \texttt{if} with no \texttt{else} branch, so it is marked as a \textit{Prospective Filter}. The second statement is not an \texttt{if} statement, so it is marked as a \textit{Prospective Map}. The third statement is marked as another \textit{Prospective Filter}. The last statement is automatically marked as a \textit{Prospective Eager}. The results of this step are shown in column 2.

Next, the algorithm computes the two sets of available and needed variables for each \textit{Prospective Operation} and annotates each \textit{Prospective Operation} with this information. This is shown in column 3.

The algorithm iterates on the four \textit{Prospective Operations} identified in the first step. It starts iterating from bottom up, in reversed control flow order. \textsc{LambdaFicator} finds that the last operation cannot be chained with the previous one; the prospective eager needs an \texttt{entry} but the upstream \texttt{filter} can only return \texttt{cl}, a \texttt{ClassLoader} object. Therefore, it merges them into a new operation and recomputes the variable availability sets. This is shown in the \textit{First iteration} column. In the next iteration it finds that the current operation needs more than one variable from the upstream operation, i.e., \texttt{cl} and \texttt{entry}, requiring \textsc{LambdaFicator} to merge the two \textit{Prospective Operations}. In iteration three, \textsc{LambdaFicator} finds that the \textit{Prospective Operations} can be chained; therefore, it does not merge the operations.

In the last step, the algorithm takes the last \textit{Prospective Operation} and, using the patterns described in Sec. 3.4, determines that the last operation is a \texttt{forEach}. Because the next block is a \textit{Prospective Filter}, \textsc{LambdaFicator} determines that the first operation is a \texttt{filter} operation.
<table>
<thead>
<tr>
<th>Iteration 1</th>
<th>Iteration 2</th>
<th>Iteration 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>Prospective filter</td>
<td>Prospective eager</td>
</tr>
<tr>
<td>If(entry)</td>
<td>NV: (entry)</td>
<td>NV: (entry)</td>
</tr>
<tr>
<td></td>
<td>AV: (result)</td>
<td>AV: (result, cl)</td>
</tr>
<tr>
<td>Prospective filter</td>
<td>NV: (entry)</td>
<td>NV: (entry)</td>
</tr>
<tr>
<td></td>
<td>AV: (result)</td>
<td>AV: (result, cl)</td>
</tr>
<tr>
<td>Prospective map</td>
<td>NV: (entry)</td>
<td>NV: (entry)</td>
</tr>
<tr>
<td></td>
<td>AV: (result, cl)</td>
<td>NV: (result, cl)</td>
</tr>
<tr>
<td>Prospective eager</td>
<td>NV: (entry)</td>
<td>NV: (cl, entry)</td>
</tr>
<tr>
<td></td>
<td>AV: (result)</td>
<td>AV: (result)</td>
</tr>
<tr>
<td>Prepend Iterable</td>
<td>If( (cl) )</td>
<td>If(entry)</td>
</tr>
<tr>
<td></td>
<td>NV: (cl)</td>
<td>NV: (entry)</td>
</tr>
<tr>
<td></td>
<td>AV: (result)</td>
<td>AV: (result)</td>
</tr>
</tbody>
</table>

Figure 3.7: Showing how the algorithm builds the chaining of operations for example 3 in Fig. 3.2.
Note: **LambdaFicator** does not split **synchronized** blocks and **try-catch** blocks between multiple functional 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 functional operations.

### 3.6 Discussion

In this section we illustrate data flow patterns that our tool handles but also the limitations that arise when refactoring more complex data flow patterns.

![Figure 3.8: Data flow graph for loop in example 3 in Fig. 3.2.](image)

Fig. 3.8 (a) shows an example of a loop that we discussed previously in Sec. 3.5. In Fig. 3.8 (b) we can see the data flow graph of this loop. Nodes represent instructions and arrows indicate the flow. Data flow in this case is not composed of a single flow. The prospective filter that the second *if* represents, filters *ClassLoader* objects, but the next statement needs an *Entry* object. An idiom frequently used in functional languages is to box the two values in a tuple and take advantage of the pattern matching facilities of such languages. Java does not provide neither easy boxing and unboxing of tuples nor pattern matching facilities, therefore we need to restructure the potential operations we inferred in order to preserve the availability of the needed dependencies. To achieve this, **LambdaFicator** merges some prospective operations.

The dotted green box in (a) illustrates how **LambdaFicator** merges operation in order to obtain the data flow graph in Fig. 3.8 (c). The circled final node symbol denotes that the operation was obtained through merging. The now merged operation results in a data flow pattern that can be refactored; all data flows on a single flow. On the other hand, there is a more elegant way of handling this. We could simply inline the call to *entry.getKey* instead of using an auxiliary variable. This would allow for a more fine grained functional operation inference in this case. In this example it’s safe to do this because the only use of the *cl* variable is inside the *if* and we do not reorder any instruction while doing the inlining. In the general case, when there are multiple uses, only side effect free method calls can be inlined. Proving a method is side effect free is not always feasible, efficient, or precise and in the general case undecidable [21, 23, 25, 26, 38]. Doing this analysis would have required a program...
analysis infrastructure and for the sake of efficiency and integration in NetBeans we decided not to implement this kind of pointer analysis. This does not affect the applicability of LAMBDAFICATOR, but it could result in better precision when inferring functional operations.

```java
for (Entry<String, WebXml> entry : fragments.entrySet()) {
    if (!requestedOrder.contains(entry.getKey())) {
        WebXml fragment = entry.getValue();
        if (fragment != null) {
            orderedFragments.add(fragment);
        }
    }
}
```

Figure 3.9: Data flow graph for loop with no merging needed.

We present in Fig. 3.9 a loop from the Apache Tomcat project. On the right hand side of the code you can see a data flow graph for the presented loop. Notice that this graph has only one flow in it. This enables LAMBDAFICATOR to refactor this code without any further merging. This data flow graph illustrates very well the kind of dependencies LAMBDAFICATOR aims to achieve through merging. In terms of graph theory, LAMBDAFICATOR needs to merge operations until it reaches a graph with only one flow. This includes the broader constraint that each functional operation has at most one upstream dependency.

Fig. 3.10 shows an example where one node has two dependencies. In this case, the predicate in the if needs two data items from upstream. LAMBDAFICATOR has to merge such operations, because it cannot return multiple values at the same time. While functional languages like ML or F# offer support for easy boxing of tuples and pattern matching, Java does not. Our algorithm could be easily adapted to take advantage of such features if they become available; for the moment LAMBDAFICATOR only allows for one value to be passed through the pipeline. Of course, this example also illustrates a case where inlining could be used, but deeper analysis is needed to determine the safety of this transformation.

Look at another example in Fig. 3.11. This illustrates a more complex data dependencies graph. Notice that this flow does not represent any actual mapping or filtering pattern. Although from a data flow perspective it might seem chaotic, from the developer’s point of view it might make a lot of sense. Because the dependencies are spread across the loop in all directions, LAMBDAFICATOR has only one choice, i.e., merge everything into a forEach operation. We represented the end-result in the green dotted box.
3.7 LambdaFicator usage scenarios

LambdaFicator has 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, package, 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. 3.12 we show how LambdaFicator works in batch mode. We applied the refactoring on the whole ANTLRWorks project. The preview pane shows the code before and after the refactoring; we show a chain of four operations. 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.
Figure 3.12: **LAMBDAFICATOR** performs the **ForLoopToFunctional** refactoring in batch mode.

The quick hint mode scans the file that is open in the editor in real-time. Fig. 3.13 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.

```java
public List<IdeaAction> ideaProviderGetActions(int position) {
    List<EditorInspectorItem> items = window.editorInspector.getActiveItems();
    for (EditorInspectorItem item : items) {
        List<IdeaAction> itemActions = item.getActiveActions();
        if (itemActions != null)
            actions.addAll(itemActions);
    }
    return actions;
}
```

Figure 3.13: **LAMBDAFICATOR** performs the refactoring in *Quick Hint* mode.
Chapter 4

Implementation Overview

We implemented our tool as a module integrated in the official release of NetBeans.

4.1 Abstract Syntax Trees

We used the Java Source API which exposes the Abstract Syntax Tree (AST) of Java programs and the NetBeans API that exposes type analyses on the source. The class hierarchy representing the AST is a direct reflection of the Java language grammar described in Java Language Specification [3].

In Fig. 4.2 you can see a simplified version of the AST of the second loop from Fig. 3.2. We represented nonterminals from the right hand side of a grammar rule production as overlapping nodes over the left hand side non terminal. For example, an EnhancedFor has a parameter, in this case ElementRule rule, an expression that evaluates to a Collection, getRules(), and a Body, the actual if.

Figure 4.1: Simplified abstract syntax tree of second loop from Fig. 3.2
4.2 General Architecture

**LambdaFicator** takes as input an AST of an *enhanced for* loop. First, checks the set of preconditions described in Sec. 3.3. If it determines that the loop satisfies the needed preconditions, it starts the functional operation algorithm inference. The original AST is then replaced with the refactored one.

![Figure 4.2: High-Level Architecture of LambdaFicator](image)

4.3 Implementation Details

For scanning preconditions we use a Visitor pattern [18] to traverse the AST. While some preconditions are localized and we only need to look for statements, e.g., `break`, `return`, others are more involved and we need to analyze the control flow and data flow.

To check the first precondition, that the loop iterates over `Collection` objects, we check the type of the *expression* present in the **EnhancedFor** node in the AST (see Fig. 4.2). To check the second precondition, the body of the loop does not throw checked exceptions, we use the analyses available in NetBeans to ensure it. To check the third precondition, no more than one non-effectively-final local variable referenced from the loop, we analyze all the writes to local variables and look to find more than one use of a non-effectively final variable. We analyze the control flow to decide if the variable that is written is compatible with our heuristic for a reducer. We start with the writing instruction as a leaf and go up in the tree looking for instruction that branch control flow. If we find such instructions, for example the **if** in Fig. 4.2, we check if the branching instruction creates more than one non-empty control-flow path; if this is the case we cannot externalize this write. For the fourth precondition, we visit `break` statements in the AST; if any such statements are found, the precondition fails. For the fifth precondition, we scan for return statements, if they return a **Boolean** literal the precondition passes; otherwise it fails. For the sixth precondition, we scan for `continue` statements and if they have a label the precondition fails.

For the merging we do a recursive, depth first, traversal of the AST and we create a **ProspectiveOperation** for each statement. We also build the **AvailableVariables** and **NeededVariables** sets for each prospective operation. Next we iterate the list and we merge operations until we reach a fixed point, i.e., all prospective operations are composable. At this point we perform the rewriting of the AST with the refactored code.
4.4 NetBeans Integration

We have integrated our tool as a NetBeans hint module. This allows NetBeans to use our module for both quickhint and batch mode of operation. In order to specify that our hint needs an enhanced for loop as input we use an annotation @TriggerTreeKind(Tree.Kind.ENHANCED_FOR_LOOP). Our module is called whenever the NetBeans scanner detects an enhanced for loop while the developer is typing or when it calls Inspect and Transform to use the tool in batch mode.
Chapter 5

Empirical Evaluation

5.1 Experimental setup

In order to empirically evaluate the usefulness of **LambdaFicator**, we ran it on 7 widely used open-source projects. We applied **ForLoopToFunctional** 1709 times. These case studies give confidence that the proposed algorithms and implementations generalize to real-world situations.

The left-hand side of Fig. 5.1 shows the size of each project in terms of non-empty non-comment source lines of code (generated using David A. Wheeler’s 'SLOC-Count'. [5]).

For each project, we applied the **ForLoopToFunctional** refactoring to all enhanced **for** loops using 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.

The value of **ForLoopToFunctional** refactoring on code quality can be best gouged 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. 1.1(b) and Fig. 1.1(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. We consider that the named operations make the semantics of the loop more explicit. Moreover, fine grained functional operations inference provide for fine grained control over the execution of each part of the code, i.e., **sequential** or in **parallel** as explained in 3.5.

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 Unix **diff** tool (we configured the tool to ignore white spaces). These numbers provide a rough estimate of 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 golden standard, we carefully analyzed for each input construct whether the refactoring can be applied. Thus, we created two sets: ShouldApply\textsubscript{man}, and ShouldNotApply\textsubscript{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\textsubscript{tool} and NotApplied\textsubscript{tool} which also contains tuples of the same form. Note that in ShouldNotApply\textsubscript{man} and NotApplied\textsubscript{tool} the tuples are of the form \(\langle in, in \rangle\). We define the Precise transformations set as:

\[
Precise = \text{ShouldApply}_{\text{man}} \cap \text{Applied}_{\text{tool}},
\]

and the Imprecise transformations set as:

\[
\text{Imprecise} = \{\langle in, out \rangle | \langle in, out \rangle \in \text{Applied}_{\text{tool}}, \exists out'.out' \neq out, \langle in, out' \rangle \in \text{ShouldApply}_{\text{man}} \},
\]

and the missed transformations set:

\[
\text{Missed} = \text{ShouldApply}_{\text{man}} \setminus \text{Applied}_{\text{tool}}.
\]

We define precision and recall using set cardinality:

\[
\text{Precision} = \frac{\left| \text{Precise} \right|}{\left| \text{Precise} \right| + \left| \text{Imprecise} \right|}
\]

\[
\text{Recall} = \frac{\left| \text{Precise} \right|}{\left| \text{Precise} \right| + \left| \text{Missed} \right|}
\]

5.2 Aplicability

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 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. However, almost half of the loops were successfully refactored by LAMBDAFICATOR.
<table>
<thead>
<tr>
<th>Project</th>
<th>SLOC</th>
<th>#for loops</th>
<th>%Refactored</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>#forEach</th>
<th>#anyMatch</th>
<th>#noneMatch</th>
<th>#reduce</th>
<th>#map</th>
<th>#filter</th>
<th>#Singleton</th>
<th>#Chains</th>
<th>Avg chain length</th>
<th>#Files Mod.</th>
<th>#SLOC Mod.</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANTLR+Works</td>
<td>97795</td>
<td>589</td>
<td>60.10%</td>
<td>78</td>
<td>15</td>
<td>102</td>
<td>44</td>
<td>56</td>
<td>18</td>
<td>322</td>
<td>17</td>
<td>4</td>
<td>10</td>
<td>117</td>
<td>93</td>
<td>216</td>
<td>138</td>
<td>2.51</td>
<td>140</td>
<td>2293</td>
<td>16</td>
</tr>
<tr>
<td>FitNesse</td>
<td>63188</td>
<td>242</td>
<td>48.34%</td>
<td>79</td>
<td>28</td>
<td>22</td>
<td>1</td>
<td>25</td>
<td>0</td>
<td>98</td>
<td>5</td>
<td>10</td>
<td>4</td>
<td>35</td>
<td>17</td>
<td>83</td>
<td>34</td>
<td>2.52</td>
<td>83</td>
<td>3336</td>
<td>8</td>
</tr>
<tr>
<td>Hadoop</td>
<td>307016</td>
<td>1772</td>
<td>28.04%</td>
<td>800</td>
<td>600</td>
<td>476</td>
<td>130</td>
<td>77</td>
<td>27</td>
<td>453</td>
<td>9</td>
<td>4</td>
<td>30</td>
<td>198</td>
<td>65</td>
<td>330</td>
<td>167</td>
<td>2.56</td>
<td>196</td>
<td>2654</td>
<td>36</td>
</tr>
<tr>
<td>Tomcat</td>
<td>221321</td>
<td>425</td>
<td>44.70%</td>
<td>149</td>
<td>42</td>
<td>88</td>
<td>29</td>
<td>45</td>
<td>3</td>
<td>173</td>
<td>1</td>
<td>15</td>
<td>1</td>
<td>151</td>
<td>22</td>
<td>129</td>
<td>61</td>
<td>3.83</td>
<td>83</td>
<td>1283</td>
<td>16</td>
</tr>
<tr>
<td>jEdit</td>
<td>114899</td>
<td>190</td>
<td>32.10%</td>
<td>82</td>
<td>10</td>
<td>51</td>
<td>13</td>
<td>22</td>
<td>6</td>
<td>61</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>23</td>
<td>14</td>
<td>37</td>
<td>24</td>
<td>2.54</td>
<td>33</td>
<td>401</td>
<td>7</td>
</tr>
<tr>
<td>FindBugs</td>
<td>121988</td>
<td>1075</td>
<td>40.09%</td>
<td>359</td>
<td>131</td>
<td>226</td>
<td>79</td>
<td>114</td>
<td>32</td>
<td>402</td>
<td>14</td>
<td>3</td>
<td>12</td>
<td>140</td>
<td>82</td>
<td>295</td>
<td>136</td>
<td>2.63</td>
<td>182</td>
<td>2346</td>
<td>16</td>
</tr>
<tr>
<td>jUnit</td>
<td>10443</td>
<td>109</td>
<td>54.12%</td>
<td>27</td>
<td>17</td>
<td>9</td>
<td>1</td>
<td>9</td>
<td>0</td>
<td>48</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>12</td>
<td>5</td>
<td>48</td>
<td>11</td>
<td>2.54</td>
<td>32</td>
<td>277</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>936650</td>
<td>4402</td>
<td>46.02%</td>
<td>1574</td>
<td>843</td>
<td>974</td>
<td>297</td>
<td>348</td>
<td>86</td>
<td>1557</td>
<td>52</td>
<td>39</td>
<td>59</td>
<td>676</td>
<td>298</td>
<td>1138</td>
<td>571</td>
<td>2.72</td>
<td>717</td>
<td>12313</td>
<td>97</td>
</tr>
</tbody>
</table>

Figure 5.1: **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.
5.3 Value

The data shows the most common operation is `forEach`. This makes sense since this operation has the most relaxed preconditions. Next on the list are `map` and `filter`. This tells us that developers transform values in loops – because that’s what `map` corresponds to. Also, we found numerous instances of filtering if statements, i.e., statements with no `else` branch or instructions following them. 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 and also control over parallelization.

5.4 Effort

The data shows that on average per project the developer would need to edit 93 files and change 1573 lines. We find that the refactoring are not clustered, but spread across files, with a rate of 2.3 loops refactored per file. 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. The analysis to determine all this is non-trivial. 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.

5.5 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 incorrect refactoring that changed semantics – only refactorings that a human might do differently (e.g., merge two subsequent `maps`). By considering how many loops were not transformed by LambdaFicator, we found that the recall of ForLoopToFunctional is 0.92. These 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 operation heuristic which can be improved. Overall, we found that most of the time LambdaFicator determines correctly the opportunity to refactor and the most precise chain a human could infer.
5.6 Safety

To answer the safety question, we ran extensive test suites (4360 tests) before and after all refactorings that we applied on our corpus projects. 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.

5.7 Threats to validity

Construct Validity:

Why did we measure the number of functional operations as value for ForLoopTo\-Functional? The more fine grained the chain of functional operations we infer, the more explicit the semantic of the expressed code; also it gives more control to the programmer over the individual execution of each operation. This brings value.

Why did we measure effort through number of modified lines of code and files? Ideally we would have asked developers to refactor this manually and measure the time needed, but this would introduce even more variables (expertise, choosing the code to refactor, etc.). The number of lines of code modified gives a good insight on the time needed to perform the changes manually, if we consider that changing any line takes some effort. The number of files changed is relevant because it proves refactorings are not clustered but widespread. This would raise, in a real project, ownership and understandability issues for the developers performing the changes.

Internal Validity:

How did we mitigate bias during manual inspection? We choose the loops to inspect randomly from the set of for loops in the project. We classified the inspected exemplars as refactorable or unrefactorable before looking at the results from LambdaFicator. We performed the transformations manually before looking at the refactored code to prevent bias. We classified as imprecise any refactoring that differed from the manually created one.

External Validity: Do our results generalize? Obviously, any project is unique in some way. We chose 9 widely used open source projects totaling over 1 million SLOC. Since the projects are developed by others we could not influence or bias the applicability of LambdaFicator in anyway. We tried to choose as diverse projects as possible. The projects in our corpus are developed by very different entities, ranging from large organizations like Apache to academia projects, e.g., FindBugs; we choose old projects, e.g., jEdit, and newer ones, e.g., Tomcat. Our corpus contains compute-intensive application like Apache Hadoop and also more visual apps, e.g., ANTLRWorks.

Reliability: Is our evaluation reliable? The subject programs we used in our evaluation are freely available. LambdaFicator is also freely available to use and so is its source code at http://refactoring.info/tools/LambdaFicator/.

---

\(^1\)jEdit does not contain JUnit tests, and for Hadoop/Tomcat none of the tests run under Java 8
Chapter 6

Conclusion

6.1 Impact

This research aims at making a difference for the software engineering community. We believe that there is a strong dependency between language tool support and adoption of language features. LAMBDAFICATOR will provide early tool support for developers, helping them to embrace the power of lambda expressions and the functional style. We made our tool available to millions of developers by shipping LAMBDAFICATOR with the official release of the NetBeans IDE.

We have designed, implemented, and evaluated an algorithm. Our results are good in terms of applicability, value provided, effort saved, and safety. The source code of our tool is freely available under a Common Development and Distribution License (CDDL) v1.0 and the GNU General Public License (GPL) v2 in the official NetBeans repository [9]. We look forward to the official launch of Java 8 in March 2014 and the public release of NetBeans. We look forward to receive feedback from users of our tool.

6.2 Future Work

LAMBDAFICATOR can be extended to handle more of the operations in the Collections API, e.g., max or into. This would extend its applicability. LAMBDAFICATOR can be integrated with a race detection tool to automatically parallelize operations that are safe to be run in parallel; also the transformation algorithm can be made more precise, as discussed in Sec. 3.6, by employing a side-effect analysis.

More broadly, work can be done in the area of refactoring imperative code to a functional style. The introduction of lambda expressions and the ability to pass functions around in Java opens the way to a new range of lambda-enabled refactorings. We look forward to see more support promoting the use of a functional style in imperative programs.

Empirical studies on how object oriented developers use functional features could guide future refactorings. More and more languages employ a mix of paradigms and encourage it, but we know very few things about how we should build systems
in such languages. There is contradicting data regarding development in functional languages Hundt [24] has acknowledged fast and simple development while Pankratius found the opposite [24, 34]. While we don’t know much about the challenges that arise when developing code in a mixed paradigm language, we know even less about what are the challenges that arise in understanding and maintaining such code. We acknowledge the need for empirical studies to understand what are the development, understandability, and maintenance issues that arise when mixing imperative with functional programming in object oriented languages.
Bibliography


[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 2004.


