IMPROVING CODE READABILITY BY TRANSFORMING LOOPS TO HIGHER ORDER FUNCTION CONSTRUCTS

Anthony DePaul[1] Chen Huo[2]

Software Engineering
Shippensburg University of Pennsylvania
{ad5146[1], chuo[2]}@cs.ship.edu

ABSTRACT

Loops can be complex to write and even harder for others to comprehend. Researchers from the software maintenance community have been working on automatically generating describing comments for loops from real world software projects. As these studies point out, their generated comments can serve as bridges to further convert the loops to higher order function constructs since they are declarative themselves. Examples of higher order functions include the map function over lists. In this paper, we present our technique which can comprehend common loop constructs and directly transform loops to higher order function constructs by manipulating abstract syntax trees. Our technique can recognize and transform three common loop patterns and also has a strategy for general loops. Our technique can also transform higher order functions back to loops. We implemented our technique as an Eclipse plugin to help developers and students make the transition from loops to higher order function constructs.

KEY WORDS

Program Transformation, Software Maintenance, Software Engineering

1. Introduction

A recent study [1] surveyed 14,317 real world projects and found only less than 20% of loops are documented to help readers. Loops without describing comments are hard to understand in general. The authors developed a technique which automatically generates describing comments for a specific kind of loop. The authors suggested that developers can further transform loops to Java Stream APIs themselves. In this paper, we take a step further. Our novel technique can directly transform 4 kinds of loops to Java Stream code. The idiom of Java Stream code is to use "higher order functions" such as `map`, `filter`, `flatMap`, etc. But first, what magic do these "higher order functions" have?

Higher order functions, functions that take in or return other functions, have originally been the primary constructs for functional programming languages. The idea of using functions to program can be traced back to Church's lambda calculus in the early 30's. Church's lambda calculus is Turing equivalent and can be the building blocks of a modern programming language [2]. An example of higher order functions is the `map` function which takes in another function and applies it to each element in a collection. This idea exists in many modern programming languages, sometimes with slightly different names. For example, C++ has `for_each`, Ruby has `each`, Python has `map`, Scala has `map` and so on. The counterpart for Java had been missing until 2014 with the release of Java 8. Java does not provide `map` for general collections but rather only gives it to the Stream class and allows other collections to be converted to a Stream. The evolution of Java exhibits the "climate change" [3] in contemporary software development that there is a shift from desktop applications to web applications, from single core to concurrent computing. Higher order functions like `map` are more adapted to the current climate. For example, they generally can work without side-effects which is crucial to concurrent computing. Google's MapReduce model is a successful example of this paradigm [4]. Moreover, higher order functions have solid theoretic foundations [5] and can be reasoned for optimizations through rigorous proofs [6].

One may say that traditional loops can do whatever these higher order functions can do. This is true without any doubt. However, not only loops can be hard to parallelize because of mutability, but also can be a software engineering nightmare as they are hard to read. The community of software evolution and maintenance has expressed such concerns in several studies, e.g., [1, 7]. The community believes that loops need describing comments to be readable. However, one recent study [1] showed that among 14,317 projects, less than 20% of loops are documented to help readers. So the same study developed an automatic method to generate describing comments for some specific loops in real world projects. Contrarily, higher order functions are considered more "declarative" as

opposed to being "imperative". For example, if one writes `lst.filter(isEven)`, it should be intuitive enough even for non-developers. In fact, many declarative languages such as MySQL are designed to be used by people who don't know much about general programming. Other advantages of higher order functions include an easy access to data level parallelism [3], the processing of large amounts of data across multiple processors.

Given the advantages of using higher order functions, efforts have been made to teach them as early as in introductory programming courses. For example, *How to Design Programs* [8] uses Racket, a functional language which is a variant of Scheme. This text has been used in several major universities with active community pedagogical support. However, overall it is still a minor choice. For/while loops still dominate in introductory programming courses as the way to express repetitive constructs. For students that learned loops in their "native" (programming) languages, it is even harder for them to adopt higher order functions than absolute beginners. This is due to the inherent difference between a loop and a higher order function like `map` — one relies on mutation and one does not.

For that matter, we propose to provide automated assistance to convert loops to declarative `map`s in Java for the purpose of software evolution and maintenance. It can also help developers or students make the transition from imperative loops. Unlike the existing commenting approaches, e.g., [1, 7, 9, 10, 11], our technique converts loops directly to the equivalent higher order function constructs, a source to source program transformation technique. This could be the ultimate goal for some of the commenting techniques [1]. To be specific, our technique recognizes and transforms the most common loop patterns to Java Stream constructs. Common loop patterns without sophisticated structures take a surprisingly large portion of loops in real world projects. For example, a study showed that 26% of loops in the 14,317 real world projects have the simple while-if construct, that is, a while loop with a nested if statement [1]. To make our technique more accessible, we developed an Eclipse plugin that can make the transformation within the project.

To our best knowledge, our technique is the fist attempt to convert loops to streams in Java for maintenance and pedagogical purposes. Our contributions include

(1) a technique that can recognize four loop patterns and transform them to Java Stream constructs automatically so as to improve the readability of related code

(2) a technique that can transform common Java Stream constructs back to loop constructs automatically

(3) an Eclipse plugin that can help developers and students convert loops to Java Stream constructs so as to learn about their relations within the project

The rest of the paper is organized as follows: Section 2 discusses related work, Section 3 demonstrates how our technique works with an example, Section 4 explains the details of our technique, Section 5 demonstrates the prototype of our Eclipse plugin, Section 6 summarizes our research and possible future work.

2. Related Work and Background

We start this section with recent works concerning readability and maintenance of loops from real world projects in the software engineering community. We then point out the necessity of introducing higher order functions to developers and students by showing the evolution of the Java language. Next we discuss the connections between loops and lists. We show that the theories of lists have been well studied in decades and have solid applications in real world problems. Finally we discuss the background of program refactoring and transformation techniques.

2.1 Readability of Loops

Abelson and Sussman [12] contended that programs must be written for people to read, and only incidentally for machine to execute. More specifically, it is reported that the ratio of time spent reading versus writing is well over 10 to 1 [13]. The software maintenance community has concerns about code readability in real world projects and has taken measures to automatically generate summaries given snippets of code, e.g., [1, 7, 9, 10, 11].

In research [1], the authors first conducted a survey of 14,317 real world projects and found only less than 20% of loops are documented to help readers. The authors extract "action units" and mine patterns of loops from real world projects. Their technique generates and only generates describing comments for loops that contain exactly one conditional statement. It may seem very specific but they found that 26% of loops are loop-if's. The authors stated that their technique can help developers identify loops that can be refactored to using the new Java 8 Stream APIs. For example, a large number of loops with an if-statement inside can be described as finding a specific element in a collection. This can be done exactly with higher order functions provided by Stream APIs. In this sense, our work is one step ahead such that our technique directly converts various patterns of loops to Stream operations. Such transformations can also preserve the readability. What makes the difference is that our technique parses code snippets to abstract syntax trees, manipulate the trees, and can easily convert the trees back to working code but not just textual comments.

An earlier technique [7] identifies loops that find elements which are maximum by some calculations and generate corresponding comments to help readers understand them. In both cases, Java Stream's `find` and `maxBy` are the higher order equivalents to the above two loop patterns. It is easy to see that such higher order functions are descriptive

themselves for the semantics.

2.2 Evolution of Java

Early Java had C-like constructs except for the object oriented features. During the last two decades, Java has evolved to have more abstract constructs. For example, generic types were introduced in Java 5 [14]. Generic types allow developers to abstract over types, e.g., `A` in `List<A>` is a type variable. `Future` and `Executor` were introduced to provide abstraction over the raw threads [15]. It rescues developers from writing error-prone code involving `Thread`. `Stream`, our star in this paper, was introduced in 2014. Another new member is `Optional` which models absence of values to replace the error prone `null`. Furthermore, `Future` was enhanced to support composition with higher order functions similar to map and flat-map. The `Flow` API was added in Java 9 to support reactive programming which is the backbone of Netflix's streaming service [3].

We can see the process of how Java attempts to catch up to the contemporary software development climate. However, these new constructs cannot be covered in one introductory course. We believe that one of contributions of our technique is to introduce students into this new realm of Java in a gentle manner by starting with the transformation from loops to streams.

2.3 Higher Order Functions For Lists

Even when a for/while loop does not operate directly on lists, the values of its control variables will form a sequence that resembles a list. From this point of view, every loop can be interpreted as some list processing operation.

If a programming language adopts higher order functions for collection types, it usually offers not only `map` but also other ones in a bundle. Typical companions include `filter`, `maxBy`, `groupBy`, `reduce`, `foldLeft`, among others. Due to their declarative nature, it is not hard for one to derive what they do in conjunction with type information. Bird [6] provides a thorough discussion on the properties of such higher order functions. For example, `map` should distribute over function composition, i.e., $map\ (f \circ g) = map\ f \circ map\ g$. Another example is the commutativity of `filter`s.

The distributivity of `map` turns out to be the property of a functor in category theory. Spivey generalizes list operations to category theory concepts [5]. "List" construct can be considered as a "free monoid functor" given an underlying set. The list functor not only promotes a regular set to a set of lists but also promotes regular functions to handle lists (i.e., `map`). If list is viewed as a functor, concatenation, flattening, and reduction of lists are natural transformations. Finally some natural transformations can form an *adjunction*.

Such high level interpretation gives lots of potential for us to further understand higher order functions on lists. In [6], the properties described in the study was used to optimize a text processing procedure from $O(n^3)$ to $O(n)$.

2.4 Program Transformation

Program transformation can be applied at different levels such as source to source or intermediate levels inside a compiler. Program transformation is often used to obtain better performance. One of the source to source transformation techniques is called constant folding [16]. The technique substitutes constants for variable expressions with the help of the reaching definition analysis. For example, assignment statements `x=10;y=x+10` will be transformed to `x=10;y=10+10`. Here we can save one variable lookup. Examples of intermediate level transformation techniques are loop jamming and loop unrolling as part of compiler optimization process [17].

Another application of source to source program transformation is automatic code refactoring. To help developers, typical Integrated Development Environment (IDE)s provide various automatic code refactoring utilities such as variable renaming, extracting methods, etc. For example, when the developer decides to choose a better name for a variable, the IDE can identify the scope of the variable and change all related occurrences at the same time when the developer simply changes the declared name. These commonly used code refactoring techniques are automatic in the sense that the developer identifies what to change and the necessary changes will be applied by the IDE. With more advanced data flow analysis, the identification of possible code refactoring can also be automated, e.g., [18].

3. Motivating Example

As an example of transforming loop source into an equivalent that utilizes Java streams, we will be using a common goal of loops which is to modify every value in a list by a common function. In this example, we will be demonstrating this by multiplying every double value in a list by two. The code to accomplish this using loops is shown in Figure 1a while its stream equivalent is shown in Figure 1b. Some lines in these figures have been split into multiple lines to better describe the various pieces.

3.1 Map

The original list, loop control variable, result variable, and current element are common elements within the for-loop structures we will be transforming. The original list, if the pattern has one, is the list that is providing the main source of information for the loop. In Figure 1a, the original list is `lst`. The loop control variable is the variable in the for loop initializer that is assigned a value. In this example, `i` is the loop control variable. The result variable represents the final result of a loop and will be the last variable being assigned a value or receiving a method call inside a for-loop. In this example, the result variable is `ans`. Finally, we

refer to the current element as a `get` method invocation of the original list that uses the loop control variable as its argument. For this example, `lst.get(i)` is the current element.

Next, we will analyze Figure 1a as a developer who is seeing the code for the first time to demonstrate the greater challenge of reading loops. Line 1 tells the reader that the result variable `ans` will begin as an empty list of the same typed stored in the original list, double. On line 2, the reader sees that the there is a for-loop that uses the loop control variable, `i`, that has an initial value of zero. Line 3 shows that the condition of the for-loop is that the loop control variable `i` is less than the length of the original list. Finally, on line 4, the reader sees that the loop updates `i` by incrementing it by one on every iteration. These three statements within the for-statement tells the reader that the loop will iterate the loop control variable from zero to one less than the length of the original list. Line 5 contains the method invocation `ans.add`. At this point, the developer knows that a new element will be added to the result list every iteration of this loop. In line 6, the reader can see what exactly is being added. It takes the current variable and multiplies it by two every time. To completely understand what is happening in this for-loop, the developer must follow the loop control variable throughout the loop. First, it is given its initial value, then the loop condition is checked, then the body is executed using the loop control variable, the loop control variable is updated, and the process repeats by checking the loop-condition again until it is no longer satisfied. Only after combining all of this information can the developer deduce what this loop accomplishes. The reader now knows that, because the loop will take each element in the list, multiply it by 2, and add it to the originally empty result list in order, the result will be the original list with every value doubled.

Next, we will analyze the example that utilizes streams demonstrated in Figure 1b. Line 1 tells the reader that the result variable will be using streams to find its value. This line does not provide a whole lot of insight into what the resulting value is going to be so it can mostly be ignored. Line 2 is the map method invocation. Because the developer knows that this method will take the stream and apply the lambda function argument to each element, it is understood that each element in the stream will be mapped to a new value. On line 3, the reader can see exactly what is going to happen to each element. For every value in the stream, the resulting stream will use the that value multiplied by two. Already the reader knows that the stream will contain the elements from the original list multiplied by 2. Finally, line 4 takes the stream that `map` produced and collects the values into a new list to be used as the result variable. Once again, this line does not provide much information about the goal of this loop and can largely be ignored by the reader.

3.2 Our Transformation Technique

Now, we have to be able to transform the example in Figure 1a to its equivalent stream pattern in Figure 1b. To begin, we have created abstract syntax tree (AST) diagrams for both code snippets shown in Figure 2. Looking at these trees, we can see an easily identifiable structure in the loop example that can be used to identify it as a map pattern. The root of this AST is a for-loop that has an initializer, condition, and updater that makes its loop control variable iterate from zero to one less than the size of the original list.

In that for-statement's then block, there is a list add method invocation and it uses the current variable as its argument. By detecting this pattern we can tell that this structure has an equivalent structure that utilizes streams. The AST that uses streams is very different than the original AST excluding one piece. The expression used for the argument of the add method is used in the lambda expression for the map method argument. Before this piece can be copied over, modifications must first be made. The expression from the add method has references to the current variable `i`. This variable does not exist in the stream pattern so it cannot be used. To solve this, every occurrence of `list.get(i)` is replaced with its equivalent, the lambda parameter `x`. After this modification has been made, this expression can then be copied over to the new AST. Now that the expression used in the map method has been properly modified and copied over, the AST for the stream pattern is now complete. Finally, the source can updated using the new AST and the old pattern is replaced.

3.3 Potentials

While parallelizing using Java streams does not always guarantee performance gains directly, it is a much easier task for developers to parallelize using streams than to parallelize a loop manually using Java threads. When a for-loop is parallelized using Java threads, a developer must create multiple threads and distribute the loop's workload. After adding these optimizations, a developer reading the code for the first time will have a much harder time understanding the intention of the loop as they would be burdened with the additional task of deciphering how the work was distributed throughout the threads. To use the parallelized version of streams, readability is not reduced as `stream()` is simply replaced with `parallelStream()`.

In the case of the example used in Figure 1b, parallelizing does not provide any immediate performance improvements. The operation that map is performing is too simple and the overhead from using streams overpowers it. When the expression used for the reduction is more CPU intensive, the benefits of parallelization begin to make themselves more apparent. To demonstrate this, a benchmark was conducted using modified versions of what is shown in Figure 1a and Figure 1b. `stream` is replaced with `parallelStream`, and, instead of multiplying every value by two, every value was multiplied by a double

```
ans = new LinkedList<Double>();
for (int i = 0;
    i < lst.size();
    i++) {
  ans.add(
      lst.get(i) * 2);
}
```

(a) **For-loop that doubles every element in a list.**

```
ans = lst.stream()
    .map(x ->
        x * 2)
    .collect(Collectors.toList());
```

(b) **Use of Java streams to double every element in a list.**

```
// 0 < seed < 1
static double expensiveCalc(double seed) {
    double res = 1;
    for (int i = 0; i < 100; i++) {
        res = res * seed + 1;
    }
    return res;
}
```

(c) **The CPU expensive calculation used to demonstrate parallelization gains.**

Figure 1: Code snippets used to demonstrate Java stream benefits.

generated by the more computationally expensive method shown in Figure 1c. The seed argument of this method used for every test is a random number between zero and one and is created before a test pass is run. The same seed is used for every call to this method for one pass of these tests. This means that the resultant list for each pattern will be identical and only the load being put on the CPU is changing with this modification.

The benchmark generates a list of one million random strings and then times the two map patterns described above. On a CPU with six cores and over 100 test passes, it completed the map using the single threaded for-loop in 123.16ms on average. The map using parallelized streams completed in 30.81ms on average. Performance gains from using `parallelStreams` is entirely situational and this amount of improvement should not be expected in every real world application. Despite this, this benchmark shows how simple and readable parallelizing with streams can be. Parallelizing the original for-loop would make the code immensely more challenging to read while parallelizing the stream pattern required only one method invocation to be renamed.

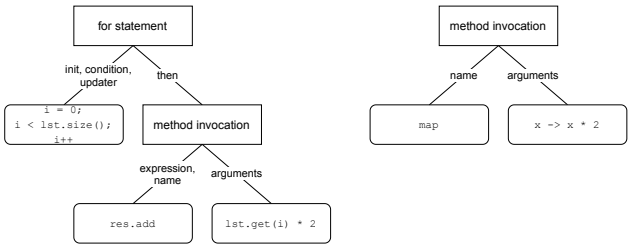


Figure 2: Map for-loop and stream ASTs.

4. Methodology

Here we will discuss the technique our tool uses to both detect for-loop patterns and transform them back and forth between their for-loop structures and equivalent stream utilizing structures. We will describe how descriptive features can be extracted from a for-loop to match the loop to the correct pattern it transforms into. Finally, we will describe each of the four transformations our tool can perform.

4.1 Detecting Patterns

In order to analyze the original source, the text is first converted into an AST which better represents the structure of the code than text. All pattern detection and transformations analyze and mutate ASTs before converting them back into the resultant source code.

Before a for-loop can be transformed into its stream utilizing equivalent, we must first identify which stream pattern matches the for-loop. There are some requirements that each of the transformation patterns need from the original for-loop. All of the transformation patterns share these requirements, so every for-loop is checked against these requirements before moving forward. The requirements are listed below.

Requirements

(1) The loop must be a for-loop

(2) Maximum of one nested if-statement

(3) Original list must be parameterized

(4) If the result variable is a list, it must be parameterized

(5) The for-loop can only have one initializer

After a for-loop's AST structure has been found to pass these common requirements, the next step is to identify some common loop features. By looking at these common features, we are able to match a pattern to a loop's set of features. We do so by assigning values to the following 5 feature definitions.

Feature Values

- F1 | If-statement: none (0), comparison using result variable and current variable (1), or other (2)
- F2 | Increments by one (1) or not (0)

- F3 | Range: 0->list length (0), 0->k (1), other (2)
- F4 | Result variable type: type stored in original list (0), new list (1), original list (2), other (3), none (4)
- F5 | Breaks inside if (1) or not (0)

The research paper that generated comments from for-loops utilized a similar pattern to identify features of a for-loop [1]. Finally, by identifying all of these features of a for-loop, it can now be matched to a pattern. Table 1 can be used to match a set of for-loop features to its stream pattern equivalent.

Pattern \ Feature	F1	F2	F3	F4	F5
Map	0	1	0	1,2	0
Reduce	1	1	0	0	0
Find	2	1	0	0,3	1
General					0

Table 1: Matching features to patterns. Empty cells are used to represent all possible values for a feature.

When transforming a stream pattern back into a for-loop, the detection is much simpler. Because the stream patterns are always strings of method invocations, only the names of those method invocations and their arguments need to be checked. For example, to find the map pattern we need to locate a list being assigned to a string of method invocations: `stream`, `map`, and `collect`.

Table 2 shows for-loop patterns and their equivalent stream versions. Note that the variables `i`, `lst`, and `res` represent the loop control variable, original list, and result variable respectively. It is not required that those variables use those exact names.

4.2 Map

To convert a for-loop that matches the map pattern, the entire for-loop is replaced with the stream structure as shown in Table 2 and its AST in Figure 3a. Much of what is needed for the stream pattern never changes. The method invocations `stream`, `map`, `collect`, and `Collectors.toList` are always going to stay the same. The pieces that vary case by case are the result variable, original list, and expression used in the `map` method invocation. These have to be found and inserted into the new structure. The result variable and original list are relatively simple to locate in the for-loop and, once they are found, get copied over to the new AST. The expression that is originally used in the for-loop structure to add elements to the result list (marked `expression` in Table 2) requires modifications before it can be used in the stream version. This is due to the fact that the original expression used here uses `lst.get(i)` to refer to the current element. The list control variable does not exist in the stream version so it must be replaced. To accomplish this, all occurrences of the current element that uses the list `get` method must be replaced by the lambda parameter `x` used in the map argument. If the loop control variable is used in any other way except for finding the current element, then the for-loop cannot be converted. Finally, with these pieces copied over to the new AST, the original for-loop can be replaced and the new source code generated.

This pattern, and all the others, can be transformed from the stream version back into a for-loop. To do so, the structure of the for-loop is created and segments are copied over in the other direction. Modifications, where required, must be applied in reverse.

4.3 Reduce

The next pattern transformation involves reducing a list down to a single element in that list by using a comparison. Table 2 and Figure 3b show the source and AST structures respectively. Once again, the structure of the method invocations `stream`, `reduce`, and `get` will always remain the same for this pattern. The original list and condition used, however, must be found within the original for-loop. While the original list can be copied over from the for-loop as is, the if condition must be modified because it contains both the current element and result variable. These need to be changed because, in the stream version, the loop control variable and result variable cannot be accessed. Because of this, the current element is replaced with the lambda parameter `x` and the result variable is replaced with the lambda parameter `y`. After the new AST is generated, it can be used to replace the original for-loop.

4.4 Find

The find pattern transforms a for-loop that sets the result variable to the first element in a list that satisfies a condition. The original for-loop and its transformation can be found in Table 2 and Figure 3c shows the AST transformation. The string of method invocations, `stream`, `filter`, `findFirst`, and `get`, is always used in the stream pattern so that structure will always be used. The pieces that have to be copied over from the original for-loop structure are the original list and the condition used in the if-statement. Like before, the condition in this if-statement will use the loop control variable to refer to the current element. As there is no loop control variable in the stream pattern, the way of referring to the current element must be changed. When copying the condition over to the filter method invocation argument, every occurrence of the current element using `lst.get(i)` is replaced with the lambda parameter `x`.

4.5 General

The final general pattern is only used if a for-loop does not match any of the other three patterns and does not contain a `break` or `return` statement within the loop body. Table 2 presents the original for-loop and its transformation while Figure 3d demonstrates the AST transformation. Like the other patterns, the general stream pattern uses a string of method invocations, `Stream.iterate`, `takeWhile`, and

	Loop	Stream

Map	`for (int i = 0; i < lst.size(); i++) {`
	`  res.add([expression]);`
	`}`
		`res = lst.stream()`
		`  .map(x -> [expression])`
		`  .collect(Collectors.toList());`

Reduce	`for (int i = 0; i < lst.size(); i++) {`
	`  if ([condition])`
	`  res = lst.get(i);`
	`}`
		`res = lst.stream()`
		`  .reduce((x, y) ->`
		`  [condition] ? x : y)`
		`  .get();`

Find	`for (int i = 0; i < lst.size(); i++) {`
	`  if ([condition]) {`
	`  res = lst.get(i);`
	`  break;`
	`  }`
	`}`
		`res = lst.stream()`
		`  .filter(x -> [condition])`
		`  .findFirst()`
		`  .get();`

General	`for (int i = [init];`
	`  [condition];`
	`  [updater]) {`
	`  [body]`
	`}`
		`Stream.iterate([init], i -> [updater])`
		`  .takeWhile(i -> [condition])`
		`  .forEach(i -> [body]);`

Table 2: For-loop and stream pattern code equivalencies.

`forEach`, that defines the overall structure. For this pattern, the stream version does have a loop control variable. This means that `init`, `condition`, and `body` can be copied over to the new AST as is with no modifications required. `updater`, on the other hand, does need to be modified. The updater in the for-loop updates the loop control variable with an expression while the lambda in the iterate method invocation behaves like an assignment. This means that the updater has to be modified. For example, if `i++` is used, the equivalent expression would be `i + 1`. After this is done, the original for-loop AST can be replaced with the new one and, following that, the new source code.

5. Prototype

Our prototype tool is implemented using the Eclipse Java development tools (JDT) framework. We first use the JDT parser to generate the abstract syntax tree of a code snippet. Then we analyze the AST using the methodology described in Section 4 to transform the AST. Finally we use JDT parser to generate source code from the transformed AST.

We developed an Eclipse plugin that can take user inputs from a prompt window as shown in Figure 4. The plugin provides two options for the user. One is to transform from a loop to a stream construct and the other is to transform a stream to a loop. In either way, the user can input the code snippet on the top window. The input will be read in by the plugin and fed to our prototype. Finally the tool outputs the result in the bottom window. Acceptable inputs, loops or streams, are restricted to the ones listed in Table 2. Our plugin will remind the user that it encounters a yet not convertible input instead of returning an erroneous output. The benefit of using an Eclipse plugin is that the user can try out the new stream paradigm whenever the user wants while working on a Java project. This can motivate the transition from loops to higher order functions.

Both the transformation tool and the Eclipse plugin are available for the public to check out and use. (Links to the repositories redacted here for the double blind review process.)

6. Conclusion and Future Work

In this paper, we present our technique which can transform three common loop patterns and a general loop pattern to Java Stream constructs. We implemented the transformation process using JDT and developed an Eclipse plugin for developers to use in a more accessible manner. Our technique and prototype implementation can be used to improve the readability of loops and to evolve the loops from real world projects. Our technique conducts source to source transformation which is different from the current commenting technique which aim to improve the readability of loops. Our Eclipse plugin can help developers and students make the transition from loops to streams as a pedagogical tool. Our insight is that

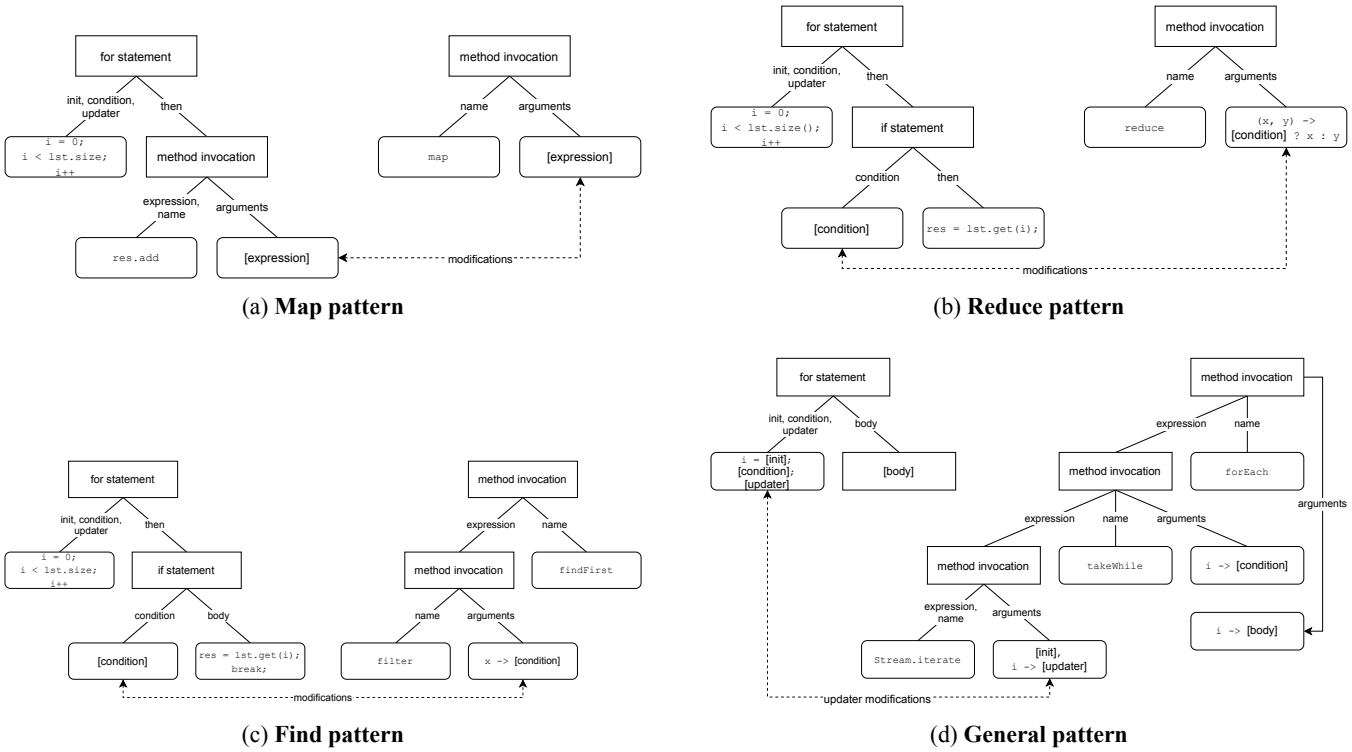


Figure 3: For-loop and stream pattern AST equivalencies and modifications.

the plugin is more helpful if it shows not only how to transform loops to streams but also the other way around so as to give the learners a full view of the paradigm. Interestingly, the fact that transformation from streams to loops is relatively simpler provides evidence that loops are harder to understand even from the view of an automated tool.

Our future work includes introducing data flow analysis to our transformation process. With data flow analysis, our technique could better understand the meaning of the loops and provide more accurate transformation to readable higher order function constructs in general cases. Another direction of future work is to apply our refined technique to real world projects for evolution and maintenance purposes. Unlike [1, 7] that requires human evaluation for new comments, the correctness of the transformations can be automatically verified by the tests that come with the projects.

References

[1] X. Wang, L. Pollock & K. Vijay-Shanker, Developing a Model of Loop Actions by Mining Loop Characteristics from a Large Code Corpus, *Proceedings of 31st International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2015, 35–44.

[2] G. Michaelson, *An Introduction to Functional Programming Through Lambda Calculus* (Dover, 2011).

[3] R.-G. Urma, M. Fusco & A. Mycroft, *Modern Java in Action: Lambdas, Streams, Functional and Reactive Programming* (Manning Publications Co., 2018).

[4] J. Dean & S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation - Volume 6*, OSDI'04, San Francisco, CA: USENIX Association, 2004, 10.

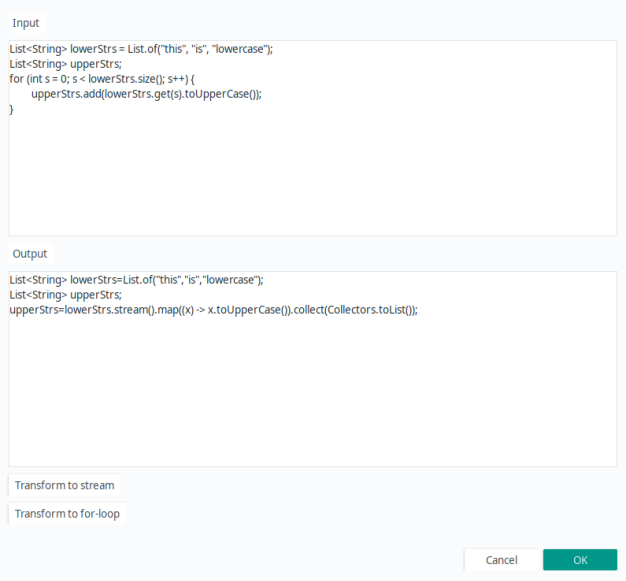
[5] M. Spivey, A categorical approach to the theory of lists, *Mathematics of Program Construction*, ed. by J. L. A. van de Snepscheut, Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, 399–408.

[6] R. S. Bird, An Introduction to the Theory of Lists, *Logic of Programming and Calculi of Discrete Design*, ed. by M. Broy, Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, 5–42.

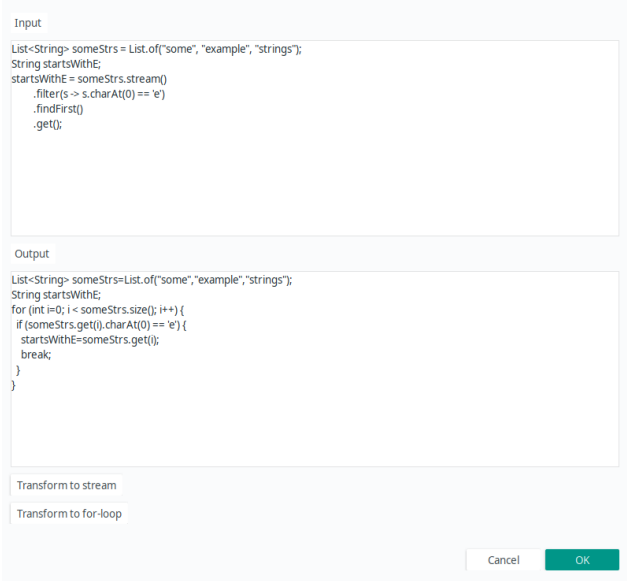
[7] G. Sridhara, L. Pollock & K. Vijay-Shanker, Automatically Detecting and Describing High Level Actions Within Methods, *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, Waikiki, Honolulu, HI, USA: ACM, 2011, 101–110.

[8] M. Felleisen et al., *How to Design Programs: An Introduction to Programming and Computing* (The MIT Press, 2018).

[9] E. Hill, L. Pollock & K. Vijay-Shanker, Automatically capturing source code context of NL-queries for software maintenance and reuse, *2009 IEEE 31st International Conference on Software Engineering*, May 2009, 232–242.



(a) **Converting a loop to a stream.**



(b) **Converting a loop to a stream.**

Figure 4: Illustrating the use of our Eclipse plugin.

[10] M. Ohba & K. Gondow, "Toward Mining "Concept Keywords" from Identifiers in Large Software Projects", in: *SIGSOFT Softw. Eng. Notes* 30.4 (May 2005), 1–5.

[11] L. Moreno et al., Automatic generation of natural language summaries for Java classes, *2013 21st International Conference on Program Comprehension (ICPC)*, May 2013, 23–32.

[12] H. Abelson, G. Sussman & J. Sussman, *Structure and Interpretation of Computer Programs* (McGraw-Hill, Inc., 1996).

[13] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship* (Prentice Hall PTR, 2008).

[14] M. Naftalin & P. Wadler, *Java Generics and Collections* (O'Reilly Media, Inc., 2006).

[15] T. Peierls et al., *Java Concurrency in Practice* (Addison-Wesley Professional, 2005).

[16] F. Nielson, H. R. Nielson & C. Hankin, *Principles of Program Analysis* (Springer Publishing Company, Incorporated, 2010).

[17] A. V. Aho et al., *Compilers: Principles, Techniques, and Tools (2nd Edition)* (Addison-Wesley Longman Publishing Co., Inc., 2006).

[18] T. Sharma, Identifying Extract-Method Refactoring Candidates Automatically, *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, Rapperswil, Switzerland: Association for Computing Machinery, 2012, 50–53.