Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams

Raffi Khatchadourian[a,*], Yiming Tang[b], Mehdi Bagherzadeh[c]

[a] *Department of Computer Science, City University of New York (CUNY) Hunter College, 695 Park Avenue, Room HN 1008, New York, NY 10065 USA*
[b] *Department of Computer Science, City University of New York (CUNY) Graduate Center, 365 5th Ave, New York, NY 10016 USA*
[c] *Department of Computer Science & Engineering, Oakland University, Rochester, MI 48309 USA*

Abstract

Streaming APIs are becoming more pervasive in mainstream Object-Oriented programming languages and platforms. For example, the Stream API introduced in Java 8 allows for functional-like, MapReduce-style operations in processing both finite, e.g., collections, and infinite data structures. However, using this API efficiently involves subtle considerations such as determining when it is best for stream operations to run in parallel, when running operations in parallel can be less efficient, and when it is safe to run in parallel due to possible lambda expression side-effects. In this paper, we present an automated refactoring approach that assists developers in writing efficient stream code in a semantics-preserving fashion. The approach, based on a novel data ordering and typestate analysis, consists of preconditions and transformations for automatically determining when it is safe and possibly advantageous to convert sequential streams to parallel and unorder or de-parallelize already parallel streams. The approach was implemented as a plug-in to the popular Eclipse IDE, uses the WALA and SAFE analysis frameworks, and was evaluated on 18

*Corresponding author.
Email addresses: `raffi.khatchadourian@hunter.cuny.edu` (Raffi Khatchadourian), `ytang3@gradcenter.cuny.edu` (Yiming Tang), `mbagherzadeh@oakland.edu` (Mehdi Bagherzadeh)
URL: `http://cs.hunter.cuny.edu/~Raffi.Khatchadourian99` (Raffi Khatchadourian), `http://linkedin.com/in/gracetang1993` (Yiming Tang), `http://mbagherz.bitbucket.io` (Mehdi Bagherzadeh)

Java projects consisting of ∼1.65M lines of code. We found that 116 of 419 candidate streams (27.68%) were refactorable, and an average speedup of 3.49 on performance tests was observed. The results indicate that the approach is useful in optimizing stream code to their full potential.

Keywords: refactoring, static analysis, automatic parallelization, performance improvement, interprocedural static analysis, typestate analysis, operation and data order analysis, side-effect analysis, Java 8, streams

---

1. Introduction

Streaming APIs are widely-available in today's mainstream, Object-Oriented programming languages and platforms [1], including Scala [2], JavaScript [3], C# [4], Java [5], and Android [6]. These APIs incorporate MapReduce-like [7] operations on native data structures such as collections. Below is a "sum of even squares" example in Java [1], which accepts a λ-expression (unit of computation) and results in the list element's square. The λ-expression argument to `filter()` evaluates to true iff the element is even:

```
list.stream().filter(x -> x % 2 == 0).map(x -> x * x).sum();
```

MapReduce, which helps reduce the complexity of writing parallel programs by facilitating big data processing [8] on multiple nodes using succinct functional-like programming constructs, is a popular programming paradigm for writing a specific class of parallel programs. It makes writing parallel code easier, as writing such code can be difficult due to possible data races, thread interference, and contention [9–13]. For instance, the code above can execute in parallel simply by replacing `stream()` with `parallelStream()`.

MapReduce, though, traditionally operates in a highly-distributed environment with no concept of shared memory, while Java 8 Stream processing operates in a single node under multiple threads or cores in a shared memory space. In the latter case, because the data structures for which the MapReduce-like operations execute are on the local machine, problems may arise from the close intimacy between shared memory and the operations being performed. Developers, thus, must *manually* determine whether running stream code in parallel results in an efficient yet interference-free program [14] and ensure that no operations on different threads interleave [15].

Despite the benefits [16, Ch. 1], using streams efficiently requires many subtle considerations. For example, it is often not straight-forward if running an operation in parallel is more optimal than running it sequentially due to potential side-effects of λ-expressions, buffering, etc. Other times, using *stateful* λ-expressions, i.e., those whose results depend on any state that may change during execution, can undermine performance due to possible thread contention.

In general, these kinds of errors can lead to programs that undermine concurrency, underperform, and are inefficient. Moreover, these problems may not be immediately evident to developers and may require complex interprocedural analysis, a thorough understanding of the intricacies of a particular stream implementation, and knowledge of situational API replacements. Manual analysis and/or refactoring (semantics-preserving, source-to-source transformation) to achieve optimal results can be overwhelming and error- and omission-prone. This problem is exacerbated by the fact that 419 total candidate streams[1] across 18 projects[2] were found during our experiments (section 4), a number that can increase over time as streams rise in popularity. In fact, Mazinanian et al. [17] found an increasing trend in the adoption of λ-expressions, an essential part of using the Java 8 stream API, with the number of λ-expressions being introduced increasing by two-fold between 2015 and 2016. And, a recent GitHub search by the authors yielded 350K classes importing the `java.util.stream` package.

The operations issued per stream may be many; we found an average of 4.14 operations per stream. Permutating through operation combinations and

[1]Stream candidacy is determined by several analysis parameters that involve performance trade-offs as described in sections 4.2 and 4.3.

[2]A stream instance approximation is defined as an invocation to a stream API returning a stream object, e.g., `stream()`, `parallelStream()`.

subsequently assessing performance, if such dedicated tests even exist, can be burdensome. (Manual) interprocedural and type hierarchy analysis may be needed to discover ways to use streams in a particular context optimally.

Previously, attention has been given to retrofitting concurrency on to existing sequential (imperative) programs [18–20], translating imperative code to MapReduce [21], verifying and validating correctness of MapReduce-style programs [22–25], studying the use of λ-expressions [17,26–28] and streams [29], and improving performance of the underlying MapReduce framework implementation [30–33]. Little attention, though, has been paid to mainstream languages utilizing functional-style APIs that facilitate MapReduce-style operations over native data structures like collections. Furthermore, improving imperative-style MapReduce code that has either been handwritten or produced by one the approaches above has, to the best of our knowledge, not been thoroughly considered. Tang et al. [14] only briefly present preliminary progress towards this end, while Khatchadourian et al. [34] discuss engineering aspects.

The problem may also be handled by compilers or run times, however, refactoring has several benefits, including giving developers more control over where the optimizations take place and making parallel processing explicit. Refactorings can also be issued multiple times, e.g., prior to major releases, and, unlike static checkers, refactorings transform source code, a task that can be otherwise error-prone and involve nuances.

We propose a fully-automated, semantics-preserving refactoring approach that transforms Java 8 stream code for improved performance.[3] The approach is based on a novel data ordering and typestate analysis. The ordering analysis involves inferring when maintaining the order of a data sequence in a particular expression is necessary for semantics preservation. Typestate analysis is a program analysis that augments the type system with "state" and has been

[3]Our approach is categorized as a refactoring due to the transformations being semantics-preserving as opposed to a more general program transformation that may not preserve semantics.

traditionally used for preventing resource errors [35,36]. Here, it is used to identify stream usages that can benefit from "intelligent" parallelization, resulting in more efficient, semantically-equivalent code.

Typestate was chosen to track state changes of streams that may be aliased and to determine the final state following a terminal (reduction) operation. Non-terminal (intermediate) operations may return the receiver stream, in which case traditional typestate applies. However, we augmented typestate to apply when a *new* stream is returned in such situations (cf. sections 3.3 and 3.5). Our approach interprocedurally analyzes relationships between types. It also discovers possible side-effects in λ-expressions to safely transform streams to either execute sequentially or in parallel, depending on which refactoring preconditions, which we define, pass. Furthermore, to the best of our knowledge, it is the first automated refactoring technique to integrate typestate.

The refactoring approach was implemented as an open-source Eclipse [37] plug-in that integrates analyses from WALA [38] and SAFE [39]. The evaluation involved studying the effects of our plug-in on 18 Java projects of varying size and domain with a total of $\sim$1.65M lines of code. Our study indicates that (i) given its interprocedural nature, the (fully automated) analysis cost is reasonable, with an average running time of 70.26 minutes per candidate stream and 34.04 seconds per thousand lines of code, (ii) despite their ease-of-use, parallel streams are not commonly (manually) used in modern Java software, motivating an automated approach, and (iii) the proposed approach is useful in refactoring stream code for greater efficiency despite its conservative nature. This work makes the following contributions:

Precondition formulation and algorithm design. We present a novel refactoring approach for maximizing the efficiency of Java 8 stream code by automatically determining when it is safe and possibly advantageous to execute streams in parallel, when running streams in parallel can be counterproductive, and when ordering is unnecessarily depriving streams of optimal performance. Our minimally invasive transformation algorithm approach refactors streams for greater parallelism while maintaining original semantics.

Generalized typestate analysis. Streams necessitate several generalizations of typestate analysis, including determining object state at arbitrary points and support for immutable object call chains. Reflection is also combined with (hybrid) typestate analysis to identify initial states.

Implementation and experimental evaluation. To ensure real-world applicability, the approach was implemented as an Eclipse plug-in built on WALA and SAFE and was used to study 18 Java programs that use streams. Our technique successfully refactored 27.68% of candidate streams, and we observed an average speedup of 3.49 during performance testing. The experimentation also gives insights into how streams are used in real-world applications, which can motivate future language and/or API design. These results advance the state of the art in automated tool support for stream code to perform to their full potential.

A shorter version of this work originally appeared in Khatchadourian et al. [40]. In this article, we add critical details of the approach, including adding a transformation algorithm, handling of advanced stream operations such as concatenation, more thorough treatments of the analyses involved, and an augmented motivating example. We also expand the experimentation by adding 63.64% more subjects, which help increase the generality of the experiments performed.

2. Motivation, Background, and Insight

We present a running example that highlights some of the challenges associated with analyzing and refactoring streams for greater parallelism and increased efficiency. Listing 1 depicts a simplified, hypothetical widget class [5]. `Widget`s have a `Color` (lines 2–3) and a real `weight` (line 4). A constructor is provided

Listing 1 A hypothetical widget class.

```java
1 class Widget {
2   enum Color {RED, BLUE, GREEN, /*...*/};
3   private Color color;
4   private double weight;
5   public Widget(Color color, double weight) {this.color = color; this.weight = weight;}
6   public Color getColor() {return this.color;}
7   public double getWeight() {return this.weight;}
8   /* override equals() and hashCode() ... */}
```

Listing 2 Sorting `Widget`s by weight.

(a) Stream code snippet before refactoring.

```java
1 Collection<Widget> unorderedWidgets =
2   new HashSet<>();
3
4 List<Widget> sortedWidgets =
5   unorderedWidgets
6     .stream()
7     .sorted(Comparator
8       .comparing(Widget::getWeight))
9     .collect(Collectors.toList());
```

(b) Improved stream code via refactoring.

```java
1 Collection<Widget> unorderedWidgets =
2   new HashSet<>();
3
4 List<Widget> sortedWidgets =
5   unorderedWidgets
6     .stream()parallelStream()
7     .sorted(Comparator
8       .comparing(Widget::getWeight))
9     .collect(Collectors.toList());
```

(line 5), as well as accessor methods (lines 6–7). `Object` methods `equals()` and `hashCode()` are appropriately overridden (not shown).

Listing 2 portrays code that uses the Java 8 Stream API to process collections of `Widget`s with `weight`s. Listing 2a is the original version, while listing 2b is the improved (but semantically-equivalent) version after our refactoring. In listing 2a, a `Collection` of `Widget`s is declared (line 1) that does not maintain element ordering as `HashSet` does not support it [41]. Note that ordering is dependent on the run time type.

A `stream` (a view representing element sequences supporting MapReduce-style operations) of `unorderedWidgets` is created on line 6. It is sequential, meaning its operations will execute serially. Streams may also have an *encounter order*, which can be dependent on the stream's source. In this case, it will be unordered since `HashSet`s are unordered.

Listing 3 Unoptimizable code collecting `weights` over 43.2 into a `Set` in parallel.

```
10 Collection<Widget> orderedWidgets =  new ArrayList<>();
11
12 Set<Double> heavyWidgetWeightSet =
13   orderedWidgets
14     .parallelStream()
15     .map(Widget::getWeight)
16     .filter(w -> w > 43.2)
17     .collect(Collectors.toSet());
```

On lines 7–8, the stream is `sorted` by the corresponding *intermediate* operation, the result of which is a (possibly) new stream with the encounter order rearranged accordingly. `Widget::getWeight` is a method *reference* denoting the method that should be used for the comparison. Intermediate operations are deferred until a *terminal* operation is executed like `collect()` (line 9). The `collect()` operation is a special kind of (mutable) reduction that aggregates results of prior intermediate operations into a given `Collector`. In this case, it is one that yields a `List`. The result is a `Widget List` sorted by `weight`.[4]

It may be possible to increase performance by running this stream's "pipeline" (i.e., its sequence of operations) in parallel.[5] Listing 2b, line 6 displays the corresponding refactoring with the stream pipeline execution in parallel (removed code is ~~struck through~~, while the added code is underlined). Note, however, that had the stream been *ordered*, running the pipeline in parallel may result in worse performance due to the multiple passes and/or data buffering required by *stateful* intermediate operations like `sorted()`. Because the stream is *unordered*, the reduction can be done more efficiently as the framework can employ a divide-and-conquer strategy [5].

In contrast, line 10 of listing 3 instantiates an `ArrayList`, which maintains element ordering. Furthermore, a parallel stream is derived from this collec-

[4] The `collect()` operation is only one kind of terminal operation; a full list is portrayed in table 3, column **`t. operation`**.

[5] A pipeline can only be executed via invoking a terminal operation.

Listing 4 Unoptimizable code sequentially `collect`ing into a `List`, `skip`ping first 1000.

```java
List<Widget> skippedWidgetList =
  orderedWidgets
    .stream()
    .skip(1000)
    .collect(Collectors.toList());
```

Listing 5 Collecting the first green `Widget`s into a `List`.

(a) Stream code snippet before refactoring.

```java
List<Widget> firstGreenList =
  orderedWidgets
    .stream()
    .filter(w->w.getColor()==Color.GREEN)
    .unordered()
    .limit(5)
    .collect(Collectors.toList());
```

(b) Improved stream code via refactoring.

```java
List<Widget> firstGreenList =
  orderedWidgets
    .stream()parallelStream()
    .filter(w->w.getColor()==Color.GREEN)
    .unordered()
    .limit(5)
    .collect(Collectors.toList());
```

tion (line 14), with each `Widget` mapped to its weight, each weighted filtered (line 16), and the results `collect`ed into a `Set`. Unlike the previous example, however, no optimizations are available here as a stateful intermediate operation is not included in the pipeline and, as such, the parallel computation does not incur the aforementioned possible performance degradation.[6]

Listing 4 creates a list of `Widget`s gathered by (sequentially) skipping the first thousand from `orderedWidgets`. Like `sorted()`, `skip()` is also a stateful intermediate operation. Unlike the previous example, though, executing this pipeline in parallel could be counterproductive because, as it is derived from an *ordered* collection, the stream is ordered. It may be possible to unorder the stream (via `unordered()`) so that its pipeline would be more amenable to parallelization. In this situation, however, unordering could alter semantics as the data is assembled into a structure maintaining ordering. As such, the stream remains sequential as element ordering must be preserved.

[6] Although no transformations are suggested in this example, a thorough analysis may still be necessary in some cases to determine when optimizations are not available.

Listing 6 Collecting `distinct Widget weights` into a `TreeSet`.

(a) Stream code snippet before refactoring.

```
31 Set<Double> distinctWeightSet =
32   orderedWidgets
33     .stream()
34     .parallel()
35     .map(Widget::getWeight)
36     .distinct()
37     .collect(Collectors
38       .toCollection(TreeSet::new));
```

(b) Improved stream code via refactoring.

```
31 Set<Double> distinctWeightSet =
32   orderedWidgets
33     .stream()
34     .parallel()
35     .map(Widget::getWeight)
36     .distinct()
37     .collect(Collectors
38       .toCollection(TreeSet::new));
```

In listing 5, the first five green `Widget`s of `orderedWidgets` are sequentially `collect`ed into a `List`. As `limit()` is a stateful intermediate operation, performing this computation in parallel could have adverse effects as the stream is ordered (with the source being `orderedWidgets`). Yet, on line 27, the stream is `unordered`[7] *before* the `limit()` operation. Because the stateful intermediate operation is applied to an unordered stream, to improve performance, the pipeline is refactored to parallel on line 26 in listing 5b. Although similar to the refactoring on line 6, it demonstrates that stream ordering does not solely depend on its source.

A distinct widget weight `Set` is created in listing 6. Unlike the previous example, this collection *already* takes place in `parallel`. Note though that there is a possible performance degradation here as the stateful intermediate operation `distinct()` may require multiple passes, the computation takes place in *parallel*, *and* the stream is *ordered*. Keeping the parallel computation but unordering the stream may improve performance but we would need to determine whether doing so is safe, which can be error-prone if done manually, especially on large and complex projects.

Our insight is that, by analyzing the type of the resulting reduction, we may be able to determine if unordering a stream is safe. In this case, it is a (mutable) reduction (i.e., `collect()` on lines 37–38) to a `Set`, of which sub-

[7]The use of `unordered()` is deliberate despite nondeterminism.

Listing 7 Collecting `distinct Widget colors` into a `HashSet`.

(a) Stream code snippet before refactoring.

```
39 Set<Color> distinctColorSet =
40   orderedWidgets
41     .parallelStream()
42     .map(Widget::getColor)
43     .distinct()
44     .collect(HashSet::new, Set::add,
45       Set::addAll);
```

(b) Improved stream code via refactoring.

```
39 Set<Color> distinctColorSet =
40   orderedWidgets
41     .parallelStream()
42     .map(Widget::getColor)
43     .unordered().distinct()
44     .collect(HashSet::new, Set::add,
45       Set::addAll);
```

classes that do not preserve ordering exist. If we could determine that the resulting `Set` is unordered, unordering the stream would be safe since the collection operation would not preserve ordering. The type of the resulting `Set` returned here is determined by the passed `Collector`, in this case, `Collectors.toCollection(TreeSet::new)`, the argument to which is a reference to the default constructor. Unfortunately, since `TreeSet`s preserve ordering, we must keep the stream ordered. Here, to improve performance, it may be advantageous to run this pipeline, perhaps surprisingly, *sequentially* (line 34, listing 6b).

Listing 7 maps, in parallel, each `Widget` to its `Color`, filter those that are `distinct`, and `collect`ing them into a `Set`. To demonstrate the variety of ways mutable reductions can occur, a more direct form of `collect()` is used rather than a `Collector`, and the collection is to a `HashSet`, which does *not* maintain element ordering. As such, though the stream is originally ordered, since the (mutable) reduction is to an *unordered* destination, we can infer that the stream can be safely unordered to improve performance. Thus, line 43 in listing 7b shows the inserted call to `unordered()` immediately before `distinct()`. This allows `distinct()` to work more efficiently under parallel computation [5].

Streams can also be stored in variables. Lines 50–53 of listing 8 sum the weight of *all* distinct `Widget`s. Two streams are created from each of the `Widget` collections (lines 46–47), with the former being unordered and the latter ordered (due to their sources) and parallel. The streams are composed via a concatenation operation on line 48, which produces an ordered stream iff *both* of the constituent streams are ordered and a parallel stream if *either* of the streams

Listing 8 Unoptimizable code obtaining the total `weight` of all `distinct` `Widgets`.

```
46 Stream<Widget> unorderedStream = unorderedWidgets.stream();
47 Stream<Widget> orderedStream = orderedWidgets.parallelStream();
48 Stream<Widget> concatStream = Stream.concat(unorderedStream, orderedStream);
49 double distinctWeightSum =
50   concatStream
51     .distinct()
52     .mapToDouble(w -> w.getWeight())
53     .sum();
```

Listing 9 Collecting `Widget colors` matching a regex.

```
54 Pattern pattern = Pattern.compile(".*e[a-z]");
55 ArrayList<String> results = new ArrayList<>();
56 orderedWidgets.stream()
57   .map(w -> w.getColor())
58   .map(c -> c.toString())
59   .filter(s -> pattern.matcher(s).matches())
60   .forEach(s -> results.add(s));
```

are parallel [42]. Here, the resulting stream is unordered and parallel, and the computation (lines 50–53) needs no further optimization.

Lastly, in listing 9, `Widget` colors matching a particular regular expression are sequentially accumulated into an `ArrayList`. The code proceeds by `mapping` each widget to its `Color` (line 57), each `Color` to its `String` representation (line 58), filtering matching strings (lines 59–59), and `forEach`, `adding` them to the resulting `ArrayList` via the λ-expression `s -> results.add(s)` (line 60). The stream is *not* refactored to parallel because of the λ-expression's possible side-effects. Otherwise, the unsynchronized `ArrayList` could cause incorrect results due to thread scheduling, possibly altering semantics. Adding synchronization would solve that problem but cause thread contention, undermining the benefit of parallelism [5].

Manual analysis of stream code can be complicated, even as seen in this simplified example. It necessitates a thorough understanding of the intricacies of the underlying computational model, a problem which can be compounded in

Table 1: Convert Sequential Stream to Parallel preconditions. Column **execution** is the stream pipeline execution mode. Column **ordering** is the ordering attribute of the stream in question, i.e., whether the stream is associated with an encounter order. Column **se** is *true* iff any behavioral parameters (λ-expressions) associated with any operations in the stream's pipeline have side-effects. Column **SIO** stands for *Stateful Intermediate Operations* and is *true* iff any intermediate operation contained within the stream's pipeline is stateful. Column **ROM** stands for *Reduce Ordering Matters* and is *true* iff ordering of the result produced by the (terminal) reduction operation must be preserved. Column **transformation** is the refactoring action to employ when the corresponding precondition passes. Cells whose value is N/A may be either *true* or *false*.

	execution	ordering	se	SIO	ROM	transformation
P1	sequential	unordered	F	N/A	N/A	Convert to parallel.
P2	sequential	ordered	F	F	N/A	Convert to parallel.
P3	sequential	ordered	F	T	F	Unorder, convert to parallel.

more extensive programs. As streaming APIs become more pervasive, it would be extremely valuable to developers, particularly those not previously familiar with functional programming, if automation can assist them in writing efficient stream code.

3. Optimization Approach

3.1. Intelligent Parallelization Refactorings

We propose two new refactorings, i.e., Convert Sequential Stream to Parallel and Optimize Parallel Stream. The first deals with determining if it is possibly advantageous (performance-wise, based on type analysis) and safe (e.g., no race conditions, semantics alterations) to transform a sequential stream to parallel. The second deals with a stream that is *already* parallel and ascertains the steps (transformations) necessary to possibly improve its performance, including *unordering* and *converting* the stream to sequential.

3.1.1. Converting Sequential Streams to Parallel

Table 1 portrays the preconditions for our proposed CONVERT SEQUENTIAL STREAM TO PARALLEL refactoring. It lists the conditions that must hold for the transformation to be both semantics-preserving as well as possibly advantageous, i.e., resulting in a possible performance gain. Column **execution** denotes the stream's execution mode, i.e., whether, upon the execution of a terminal operation, its associated pipeline will execute sequentially or in parallel. Column **ordering** denotes whether the stream is associated with an encounter order, i.e., whether elements of the stream *must* be visited in a particular order ("ord" is ordered and "unord" is unordered). Column **se** represents whether any behavioral parameters (λ-expressions) that will execute during the stream's pipeline have possible side-effects. Column **SIO** constitutes whether the pipeline has any stateful intermediate operations. Column **ROM** (**R**eduction **O**rder **M**atters) represents whether the encounter order must be preserved by the result of the terminal operation. A T denotes that the reduction result depends on the encounter order of a previous (intermediate) operation. Conversely, an F signifies that any ordering of the input operation to the reduction need not be preserved. Column **transformation** characterizes the transformation actions to take when the corresponding precondition passes (note the conditions are mutually exclusive). *N/A* is either T or F.

A stream passing P1 is one that is sequential, unordered, and has no side-effects. Because this stream is already unordered, whether or not its pipeline contains a stateful intermediate operation is inconsequential. Since the stream is unordered, any stateful intermediate operations can run efficiently in parallel. Moreover, preserving the ordering of the reduction is also inconsequential as no original ordering exists. Here, it is both safe and possibly advantageous to run the stream pipeline in parallel. The stream derived from `unorderedWidgets` on line 6, listing 2 is an example of a stream passing P1. A stream passing P2 is also sequential and free of λ-expressions containing side-effects. However, such streams are ordered, meaning that the refactoring only takes place if no

Table 2: Optimize Parallel Stream preconditions. Column **execution** is the stream pipeline execution mode. Column **ordering** is the ordering attribute of the stream in question, i.e., whether the stream is associated with an encounter order. Column **SIO** stands for *Stateful Intermediate Operations* and is *true* iff any intermediate operation contained within the stream's pipeline is stateful. Column **ROM** stands for *Reduce Ordering Matters* and is *true* iff ordering of the result produced by the (terminal) reduction operation must be preserved. Column **transformation** is the refactoring action to employ when the corresponding precondition passes.

	execution	ordering	SIO	ROM	transformation
P4	parallel	ordered	*T*	*F*	Unorder.
P5	parallel	ordered	*T*	*T*	Convert to sequential.

stateful intermediate operations exist. P3, on the other hand, will allow such a refactoring to occur, i.e., if a stateful intermediate operation exists, *only* if the ordering of the reduction's result is inconsequential, i.e., the reduction ordering need not be maintained. As such, the stream can be *unordered* immediately before the (first) stateful intermediate operation (as performed on line 43, listing 7b). The stream created on line 20, listing 4 is an example of a stream failing this precondition.

3.1.2. Optimizing Parallel Streams

Table 2 depicts the preconditions for the Optimize Parallel Stream refactoring. Here, the stream in question is *already* parallel. A stream passing either precondition is one that is ordered and whose pipeline contains a stateful intermediate operation. Streams passing P4 are ones where the reduction does not need to preserve the stream's encounter order, i.e., reduce ordering matters (**ROM**) is *F*. An example is depicted on line 41, listing 7. Under these circumstances, the stream can be explicitly unordered immediately before the (first) stateful intermediate operation, as done on line 43 of listing 7b. Streams passing P5, on the other hand, are ones that the reduction ordering *does* matter, e.g.,

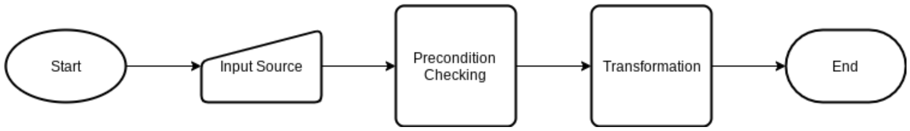


Figure 1: High-level flowchart.

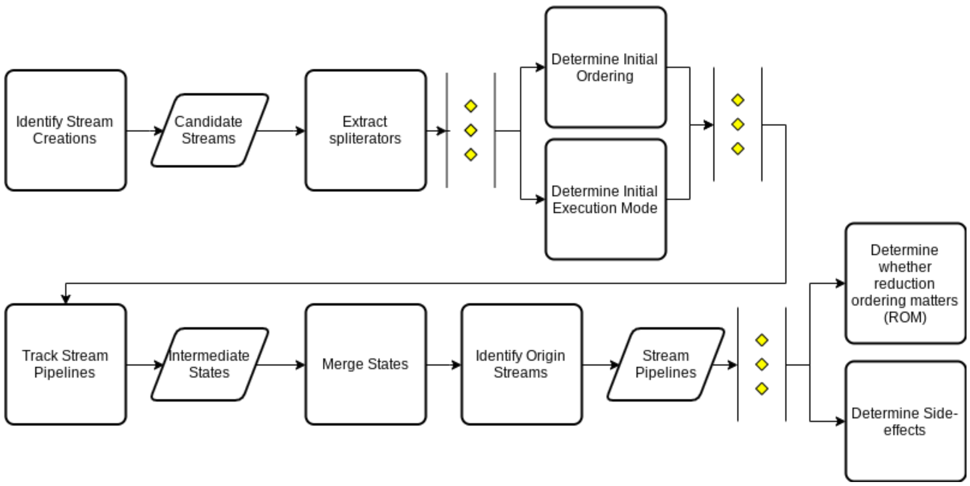


Figure 2: Precondition checking flowchart.

the stream created on line 33. To possibly improve performance, such streams are transformed to *sequential* (line 34, listing 6b).[8]

3.2. Overview

Figure 1 depicts the high-level flowchart for our approach. The process begins with input source code. Preconditions are checked on the constituent stream declarations (sections 3.3 to 3.7). Those passing preconditions are then transformed to either parallel or sequential or unordered (section 3.8).

The precondition checking process from fig. 1 is further expanded in fig. 2. First, stream creation expressions are identified (section 3.3), producing the

[8]Unlike table 1, side-effects are not considered here as our approach is a *performance*-based refactoring. De-parallelizing streams with possible side-effects would be considered a possibly semantics violating *correctness*-based transformation and is out of scope w.r.t. this work.

streams that are candidates for transformation. Next, stream attributes are analyzed (section 3.4), initially by extracting and subsequently examining their `Spliterator` [43]. This is performed to determine initial stream execution mode (section 3.4.1) and ordering (section 3.4.2). Once starting stream states have been determined, state changes are tracked through stream pipelines (section 3.5), producing intermediate streams (section 3.5.1). The states of such streams are then merged (section 3.5.2) and associated with an origin stream (section 3.5.3). The pipelines are then determined to have side-effects (section 3.6), as well as whether the terminating expression actually makes use of the stream's ordering, if applicable (section 3.7).

3.3. Identifying Stream Creation

Identifying where in the code streams are created is imperative for several reasons. First, streams are typically derived from a source (e.g., a collection) and take on its characteristics (e.g., ordering). This is used in tracking stream attributes across their pipeline (section 3.4). Second, for streams passing preconditions, the creation site serves a significant role in the transformation (section 3.8). In other words, it helps locate where the transformation should take place.

There are several ways to create streams, including being derived from `Collection`s, being created from arrays (e.g., `Arrays.stream()`), and via static factory methods (e.g., `IntStream.range()`). Streams may also be directly created via constructors. However, it is not typical of streaming API client applications, as they generally use creation APIs such as `Stream.of()`, which are our focus, as opposed to streaming API frameworks and their implementations. We consider stream creation point approximations as any expression evaluating to a type implementing the `java.util.stream.BaseStream` interface, which is the top-level stream interface. We exclude, however, streams emanating from intermediate operations, i.e., instance methods whose receiver *and* return types implement the stream interface, as such methods are not likely to produce new streams but rather ones derived from the receiver but with different attributes.

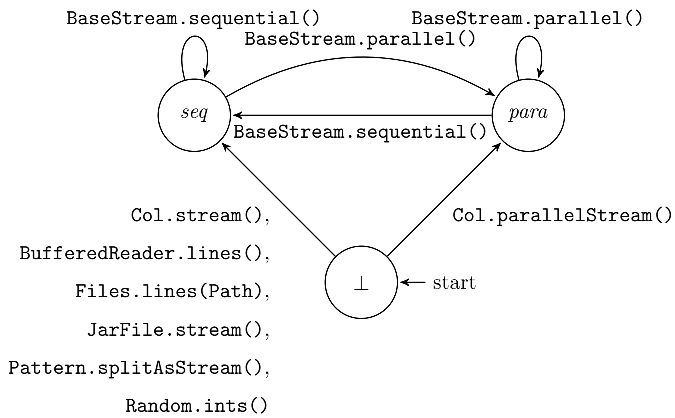


Figure 3: A proper subset of the relation $E_{\rightarrow}$ in the labeled transition system $E = (E_S, E_\Lambda, E_{\rightarrow})$. The relation depicts valid transitions between stream execution modes. The $\perp$ state is a phantom initial state immediately prior to stream creation. States "seq" is sequential and "para" is parallel.

This exclusion is part of the scheme to identify stream creation from the perspective of client applications. It does not limit the input but rather enables accurate identification.

3.4. Tracking Streams and Their Attributes

We discuss our approach to tracking streams and their attributes (i.e., state) using a series of labeled transition systems (LTSs). The LTSs are used in the typestate analysis (section 3.5).

3.4.1. Execution Mode

Definition 1. The LTS E is a tuple $E = (E_S, E_\Lambda, E_{\rightarrow})$ where $E_S = \{\perp, \mathit{seq}, \mathit{para}\}$ is the set of states, E_Λ is a set of labels, and $E_{\rightarrow}$ is a set of labeled transitions.

The labels E_Λ corresponds to method calls that either create or transform the execution mode of streams. We denote the initial stream ("phantom") state as $\perp$. Different stream creation methods may transition the newly created stream to one that is either sequential or parallel. Figure 3 portrays a proper subset of the relation $E_{\rightarrow}$ (`Col` is `Collection`, "seq" is sequential and "para" is parallel).

Transitions stemming from the $\perp$ state represent the numerous stream creation methods (section 3.3). Although it is possible to create streams directly via a constructor, Java 8 Streams are normally created from either existing data structures (such as is the case with `Collection.stream()`) or various factory methods, as shown in figs. 3 and 4.

As an example, the stream created on line 6, listing 2a would transition from $\perp$ to the *seq* state, while the stream created at line 33 would transition from *seq* to the *para* state as a result of the corresponding call on line 34. The rules governing these transitions are illustrated in fig. 3.

3.4.2. Ordering

Whether a stream has an encounter order depends on the stream source (run time) type and the intermediate operations. Certain stream sources (e.g., `List`, arrays) are intrinsically ordered, whereas others (e.g., `HashSet`) are not. Some intermediate operations (e.g., `sorted()`) may impose an encounter order on an otherwise unordered stream, and others may render an ordered stream unordered (e.g., `unordered()`). Further, some terminal operations may ignore encounter order (e.g., `forEach()`) while others (e.g., `forEachOrderer()`) abide by it [5]. The LTS for tracking stream ordering is shown in definition 2.

Definition 2. The LTS O for tracking stream ordering is the tuple $O = (O_S, O_\Lambda, O_\rightarrow)$ where $O_S = \{\perp, \mathit{ord}, \mathit{unord}\}$ and other components are in line with definition 1.

Figure 4 portrays a proper subset of the relation $O_\rightarrow$, which depicts valid transitions between stream ordering modes ("ord" is ordered and "unord" is unordered). As with E_S, $\perp$ is a phantom initial state immediately before stream creation. For presentation, the static method `Stream.concat(Stream,Stream)` is modeled as an instance method where the receiver represents the first parameter, i.e., the origin state is that of the first parameter, and the *state* of the second parameter is the sole explicit parameter (an example of stream concatenation is shown in listing 8 and discussed in the surrounding text).

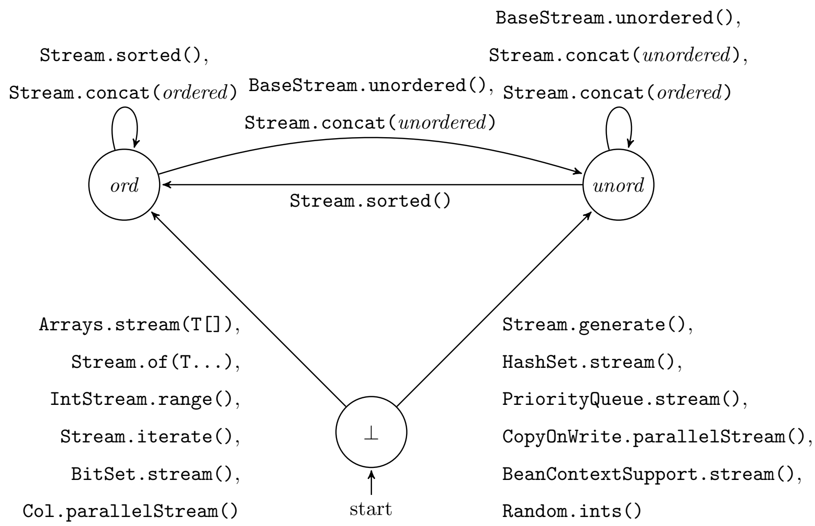


Figure 4: A proper subset of the relation $O_{\rightarrow}$ in $O = (O_S, O_\Lambda, O_{\rightarrow})$. The relation depicts valid transitions between stream ordering modes. $\perp$ is a phantom initial state immediately prior to stream creation. The static method `Stream.concat(Stream, Stream)` is modeled as an instance method where the first parameter is the receiver and the *state* of the second parameter is the sole explicit parameter. States "ord" is ordered and "unord" is unordered.

For instance, the stream created on line 6, listing 2a would transition from $\perp$ to the *unord* state due to the call to `HashSet.stream()`. Although the compile-time type of `unorderedWidgets` is `Collection` (line 1), we use an interprocedural type inference algorithm (explained next) to approximate `HashSet`. The stream created at line 33, on the other hand, would transition from $\perp$ to the *ord* state as a result of `orderedWidgets` having the approximated run time type of `ArrayList` (line 10). The rules for these transitions appear in fig. 4.

Approximating Stream Source Types and Characteristics. The fact that stream ordering can depend on the run time type of its source necessitates that its type be approximated. As shown in fig. 4, from $\perp$, a call to the instance method `BitSet.stream()` would transition us to the *ord* state, whereas a call to `HashSet.stream()` would transition us to the *unord* state. For this, we use an interprocedural type inference algorithm via points-to analysis [44], more details of which can be found in section 4.1, that computes the possible run time types of the receiver from which the stream is created (see section 3.3). Once the type is obtained, whether source types produce ordered or unordered streams is determined via reflection. While details are in section 4.1, briefly, the type is reflectively instantiated and its `Spliterator` [43] extracted. Then, stream characteristics, e.g., ordering, are queried [43]. This is enabled by the fact that collections and other types supporting streams do not typically change their ordering characteristics dynamically. For example, during program execution, an `ArrayList` would never transition from a container that maintains ordering to one that does not. In fact, developers choose which container classes to instantiate based on such characteristics, which are predetermined and documented.

Using reflection in this way amounts to a kind of *hybrid* typestate analysis where initial states are determined via dynamic analysis. If reflection fails, e.g., an abstract type is inferred, the default is to ordered and sequential. This choice is safe considering that there is no net effect caused by our proposed transformations, thus preserving semantics. Furthermore, to prevent ambiguity in state transitions, it is required that each inferred type have the same attributes. Note that abstracting the possible types to, for example, the least common super type would not be adequate as sibling types may not share the same attributes, and a receiver may not be able to take on the type of all siblings. The situation where a receiver has multiple possible run time types that are not all related to the same ordering attribute conservatively results in a refactoring precondition failure for the particular input stream creation expression. Moreover, we conservatively require that each possible (inferred) type be a leaf in the type hierarchy; this guarantees that the stream's source cannot be of a subtype that does not share the same attribute with its super type. Mistakenly inferring that a stream is unordered could have disastrous consequences in terms of semantics preservation as our performance improvements could inevitably change program behavior.

The following is an example of a stream creation expression that fails preconditions due to its possible run time types having inconsistent ordering attributes:

```java
1 void foo(int bar) {
2     Set set = null;
3     if (bar > 0) set = new HashSet();
4     else set = new TreeSet();
5     set.parallelStream();
6 }
```

On line 5, the receiver `set`, using intraprocedural analysis, has the possible types $\{$`HashSet`, `TreeSet`$\}$, meaning that the stream can be either ordered (in the case of `TreeSet`) or unordered (in the case of `HashSet`), creating a transition ambiguity per fig. 4. A similar situation could arise with execution mode in fig. 3.

3.5. Tracking Stream Pipelines

Tracking stream pipelines is essential in determining satisfied preconditions. Pipelines can arbitrarily involve multiple methods and classes as well as be data-dependent (i.e., spanning multiple branches). This kind of complication is shown in listing 8, where streams are stored in variables and can thus be passed to methods as parameters, stored in fields, and aliased. In fact, during our evaluation (section 4), we found many real-world examples that use streams interprocedurally.

Our automated refactoring approach involves developing a variant of typestate analysis [35,36] to track stream pipelines and determine stream state when a terminal operation is issued. Typestate analysis is a program analysis that augments the type system with “state” information and has been traditionally used for prevention of program errors such as those related to resource usage. It works by assigning each variable an initial ($\perp$) state (cf. figs. 3 and 4). Then, (mutating) method invocations change the object’s state. A lattice represents states, and LTSs represent possible transitions. If each method invocation sequence on the receiver does not eventually change the object back to the $\perp$ state, the object may be left in a nonsensical state, indicating the potential presence of a bug.

Our typestate analysis makes use of a call graph, which is created via a k-CFA call graph construction algorithm [45], making our analysis both object and context sensitive (the context being the k-length call string). In other words, it adds context so that method calls to an object creation site (`new` operator) can be distinguished from one another [46, Ch. 3.6]. It is used here to consider client-side invocations of API calls as object creations. Setting $k = 1$ would not suffice as the analysis would not consider the client contexts as stream creations. As such, at least for streams, k must be $>= 2$. Although k is flexible in our approach, we use $k = 2$ as the default for streams and $k = 1$ elsewhere. Section 4.2.1 discusses how k was set during our experiments, as well as a heuristic to help guide developers in choosing a sufficient k.

We formulate a variant of typestate since operations like `sorted()` return (possibly) new streams derived from the receiver stream with their attributes altered. Definition 3 portrays the formalism capturing the concept of typestate analysis used in the remainder of this section. Several generalizations are made to extract typestate at a particular program point.

Definition 3 (Typestate Analysis)**.** Define $\mathit{TState}_{\mathit{LTS}}(i_s, \mathit{exp}) = S$ where LTS is a labeled transition system, i_s a stream instance, exp an expression, and S the possible states of i_s at exp according to LTS.

In definition 3, exp, an expression in the Abstract Syntax Tree (AST), is used to expose the internal details of the analysis. Typically, typestate is used to validate complete statement sequences. Regarding definition 3, this would be analogous to exp corresponding to a node associated with the last statement of the program. In our case, we are interested in typestates at particular program points; otherwise, we may not be able to depict typestate at the execution of the terminal operation accurately. For example, let i_s be the stream on line 6, listing 2a and exp the method call `collect()` at line 9. Then, $\mathit{TState}_{O}(i_s, \texttt{collect(..)}) = \{\mathit{ord}\}$ as depicted in fig. 4.

Traditional typestate analysis is used with (mutating) methods that alter object state. The Stream API, though, is written in an immutable style where each operation returns a stream reference that may refer to a new object. A naïve approach may involve tracking the typestates of the returned references from intermediate operations. Doing so, however, would produce an undesirable result as each stream object would be at the starting state.

Section 3.4 treats intermediate operations as being (perhaps void returning) methods that mutate the state of the receiver. This makes the formalism concise. However, in actuality, intermediate operations are *value returning methods*, returning a reference to the same (general) type as the receiver. As such, the style of this API is that of immutability, i.e., "manipulating" a stream involves creating a new stream based on an existing one. In such cases, the receiver is then considered *consumed*, i.e., any additional operations on the receiver would result in a run time exception, similar to linear type systems [47].

Our generalized typestate analysis works by tracking the state of stream instances as follows. For a given expression, the analysis yields a set of possible states for a given instance following the evaluation of the expression. Due to the API style, a typestate analysis that has a notion of instances that are *based on* other instances is needed. As such, we compute the typestate of individual streams and proceed to *merge* the typestates to obtain the *final* typestate when a terminal operation consumes the stream. The final typestate is derived at this point because that is when all of the (queued) intermediate operations will execute. Moreover, the final typestate is a *set* due to dataflow analysis of multiple paths.

3.5.1. Intermediate Streams

A stream is created via APIs calls stemming from the $\perp$ state as discussed in section 3.4. Recall that intermediate operations may or may not also create streams based on the receiver. We coin such streams as *intermediate* streams as they are used to progress the computation to a final result. Moreover, intermediate streams cannot be instantiated alone; they must be based on (or derived from) existing ones. If an intermediate stream is derived from another intermediate stream, then, there must exist a chain of intermediate stream cre-

Listing 10 Sequencing stream instance derivations.

(a) Before refactoring.

```java
1  void m(int x) {
2    Stream s1 =
3      o.stream();//1
4    Stream s2 = null;
5    if (x > 0)
6      s2 = s1.filter(..);//2
7    else
8      s2 = s1.parallel()
9        .filter(..);//3
10   int c = s2.count();}
```

(b) After refactoring.

```java
1  void m(int x) {
2    Stream s1 =
3      o.stream()parallelStream();
4    Stream s2 = null;
5    if (x > 0)
6      s2 = s1.filter(..);
7    else
8      s2 = s1.parallel()
9        .filter(..);
10   int c = s2.count();}
```

ations that starts at a non-intermediate stream. Due to conditional branching and polymorphism, there may be multiple such (possible) chains. Intermediate streams must be appropriately arranged so that the correct final state may be computed.

To sequence stream instances, we require a "predecessor" function $\mathit{Pred}(i_s) = \{i_{s_1}, \ldots, i_{s_n}\}$ that maps a stream i_s to a set of streams that may have been used to create i_s. $\mathit{Pred}(i_s)$ is computed by using the points-to set of the reference used as the receiver when i_s was instantiated.

We now demonstrate the predecessor function using the code in listing 10a. Suppose we would like to know the state of the stream referred to by `s2` before the commencement of the terminal operation `count()` on line 10. The points-to set of `s2` consists of the objects created by each of the `filter()` operations on lines 6 and 9, respectively. These allocation sites have been numbered in comments in the source code using comments.[9] As such, we have that $\mathit{PointsTo}(\texttt{s2}) = \{\texttt{filter()}_2, \texttt{filter()}_3\}$.[10] For the first call to `filter()`, `s1` refers to the receiver. Because $\mathit{PointsTo}(s1) = \{\texttt{stream()}_1\}$ (from line 3), we

[9] For presentation purposes, we treat API calls as abstract object creation sites instead of the traditional `new` operators as in [36]. However, setting $k > 1$ and using call-string context sensitivity is how this effect is actually achieved.

[10] We purposely use API-level allocation sites so as to remain as implementation-neutral as possible.

have that $\mathit{Pred}(\texttt{filter()}_2) = \texttt{stream()}_1$. Finally, because `stream()` is *not* an intermediate operation, we have that $\mathit{Pred}(\texttt{stream()}_1) = \varnothing$.

Conversely, for the call to `filter()` on line 9, the receiver is the result of `s1.parallel()`. Interestingly, no allocation takes place here as `parallel()` simply sets a field value in the receiver and returns its reference, i.e., `s1`. Since $\mathit{PointsTo}(\texttt{s1}) = \{\texttt{stream()}_1\}$, we also have that $\mathit{Pred}(\texttt{filter()}_3) = \{\texttt{stream()}_1\}$. Definition 4 describes this function more generally.

Definition 4 (Predecessor Objects)**.** Define $\mathit{Pred}(\texttt{o.m()}) = \{i_1, i_2, \ldots, i_n\}$ where `o` is an object reference, `m` a method, `o.m()` results in an object reference, and $i_k \in \{i_1, i_2, \ldots, i_n\}$ for $1 \leq k \leq n$ an abstract heap object identifier:

$$\mathit{Pred}(\texttt{o.m()}) = \begin{cases} \varnothing & \text{if } \texttt{m()} \text{ is not intermediate.} \\ \mathit{PointsTo}(\texttt{o}) & \text{o.w.} \end{cases}$$

3.5.2. Typestate Merging

Since intermediate operations possibly create new streams based on the receiver, the typestate analysis will generate different states for any stream produced by an intermediate operation. We are interested in, however, the final state just before the commencement of the terminal operation, which results in stream consumption. Recall from section 3.4.1 that $\perp$ models an initial state. As such, $\perp$ will symbolize the initial state of intermediate streams. In other words, although an intermediate stream may "inherit" state from the stream from which it is derived, in our formalism, we use $\perp$ as a placeholder until we can derive what exactly the state should be. To this end, we introduce the concept of *typestate merging*.

First, we define a state selection function that results in the first state if it is not $\perp$ and the second state otherwise:

Definition 5 (State Selection)**.** Define $\mathit{Select}\colon\ S \times S \rightarrow\ S$ to be the state selection function:

$$\mathit{Select}(s_i, s_j) = \begin{cases} s_j & \text{if } s_i = \bot \\ s_i & \text{o.w.} \end{cases}$$

Definition 5 "selects" the "most recent" state in the case that the typestate analysis determines it for the instance under question and a previous state otherwise. For example, let $s_i = \bot$ and $s_j = \mathit{para}$. Then, $\mathit{Select}(s_i, s_j) = \mathit{para}$. Likewise, let $s_i = \mathit{unord}$ and $s_j = \mathit{ord}$. Then, $\mathit{Select}(s_i, s_j) = \mathit{unord}$.

Next, we define the state merging function, which allows us to merge two sets of states, as follows:

Definition 6 (State Merging)**.** Define $\mathit{Merge}(S_i, S_j) = S$ to be the typestate merging function:

$$\mathit{Merge}(S_i, S_j) = \begin{cases} S_i & \text{if } S_j = \varnothing \\ S_j & \text{if } S_i = \varnothing \\ \{\mathit{Select}(s_i, s_j) \mid s_i \in S_i \wedge s_j \in S_j\} & \text{o.w.} \end{cases}$$

As an example, let $S_i = \{\bot\}$ and $S_j = \{\mathit{seq}, \mathit{para}\}$. Then, $\mathit{Merge}(S_i, S_j) = \{\mathit{seq}, \mathit{para}\}$. Likewise, let $S_i = \{\mathit{ord}, \mathit{unord}\}$ and $S_j = \{\mathit{ord}, \mathit{unord}\}$. Then, $\mathit{Merge}(S_i, S_j) = \{\mathit{unord}, \mathit{ord}\}$.

Finally, we define the notation of merged typestate analysis:

Definition 7 (Merged Typestate Analysis)**.** Define $\mathit{MTState}_{\mathit{LTS}}(i_s, \mathit{exp}) = S$ where LTS is a labeled transition system, i_s a stream, exp an expression, to be the typestate analysis merging function:

$$\mathit{MTState}_{\mathit{LTS}}(i_s, \mathit{exp}) =$$
$$\begin{cases} \mathit{TState}_{\mathit{LTS}}(i_s, \mathit{exp}) & \text{if } \mathit{Pred}(\texttt{o.m()}) = \varnothing \\ \displaystyle\bigcup_{i_{s_k} \in \mathit{Pred}(i_s)} \mathit{Merge}(\mathit{TState}_{\mathit{LTS}}(i_s, \mathit{exp}), \mathit{MTState}_{\mathit{LTS}}(i_{s_k}, \mathit{exp})) & \text{o.w.} \end{cases}$$

This final function aggregates typestate over the complete method call chain until the terminal operation after exp. For example, let $i_s = \texttt{filter()}_2 \in \mathit{PointsTo}(\texttt{s2})$ and $exp = \texttt{s2.count(..)}$ from listing 10a. Then, $\mathit{MTState}_E(i_s, exp)$

$$
\begin{aligned}
&= \{\mathit{Merge}(\mathit{TState}_E(i_s, exp), \mathit{MTState}_O(\texttt{stream()}_1, exp))\} \\
&= \{\mathit{Merge}(\mathit{TState}_E(i_s, exp), \mathit{TState}_O(\texttt{stream()}_1, exp))\} \\
&= \{\mathit{Merge}(\{\bot\}, \{seq, para\})\} \\
&= \{\mathit{Select}(\bot, seq), \mathit{Select}(\bot, para)\} \\
&= \{seq, para\}
\end{aligned}
$$

3.5.3. Identifying Origin Streams

Once a stream's merged typestate at the terminal operation has been determined, the relationship between this stream and the original (non-intermediate) stream is examined. Because a series of intermediate operations can form a chain of streams starting at a non-intermediate stream, the stream being consumed by a terminal operation may not be the original stream, i.e., it may be one of the derived, intermediate streams. We denote original streams in the computation as *origin* streams. In terms of definition 7, origin streams are those processed in the base case.

An intermediate stream may have multiple origin streams due to branching, polymorphism, etc. Identifying origin streams is important in tracking the complete stream pipeline, as well locating potential areas where refactoring transformations may take place (as in section 3.8). Moreover, identify the stream origin as, e.g., initial stream ordering is dependent on the type from which it was derived or the (static) method that was used to create it. In other words, it is needed to determine the transitions from the start states in figs. 3 and 4. We define the concept of origin objects more generally as follows:

Definition 8 (Origin Objects)**.** Define $\mathit{Origins}(\texttt{o.m()}) = \{i_1, i_2, \ldots, i_n\}$ where `o` is an object reference, `m()` a method, `o.m()` results in an object reference, and

$i_k \in \{i_1, i_2, \ldots, i_n\}$ for $1 \leq k \leq n$ an abstract heap object identifier:

$$\mathit{Origins}(\texttt{o.m()}) \quad = \quad \begin{cases} \varnothing & \text{if } \texttt{o.m() == null}. \\ \{\texttt{o.m()}\} & \text{if } \mathit{Pred}(\texttt{o.m()}) = \varnothing \\ \displaystyle\bigcup_{i_j \in \mathit{Pred}(\texttt{o.m()})} \mathit{Origins}(i_j) & \text{o.w.} \end{cases}$$

To illustrate, consider the code in listing 10a. We have that $\mathit{Origins}(\texttt{s2.count()})$

$$\begin{aligned} &= \mathit{Origins}(\texttt{filter()}_2) \cup \mathit{Origins}(\texttt{filter()}_3) \\ &= \mathit{Origins}(\texttt{stream()}_1) \cup \mathit{Origins}(\texttt{stream()}_1) \\ &= \{\texttt{stream()}_1\} \cup \{\texttt{stream()}_1\} \\ &= \{\texttt{stream()}_1\} \end{aligned}$$

3.6. Inferring Behavioral Parameter Side-effects

In this section, we more formally define what it means for behavioral parameters (λ-expressions) that will execute as part of a stream's pipeline to possibly contain side-effects. Side-effect considerations are part of the refactoring precondition checks in table 1 and are an essential part of determining whether a sequential stream can be safely converted to one whose pipeline executes in parallel. The following more formally defines the λ-expressions associated with streams:

Definition 9 (Stream λ-expressions)**.** Define the function $\lambda(i_s) = \lambda\mathit{exp}$ that maps a streams instance i_s to a λ-expression $\lambda\mathit{exp}$ used in creating i_s. If no λ-expression is used creating i_s, then $\lambda\mathit{exp} = \bullet$, an "empty" expression not associated with any meaningful instruction (no-op).

Let i_s be the stream created as a result of the `filter()` operation on line 16 of listing 3. Then, $\lambda(i_s) =$ `w -> w > 43.2`. Likewise, let i_s be the stream that results from `skip()` on line 21. Then, $\lambda(i_s) = \bullet$.

Next, we describe the meaning of λ-expressions to contain side-effects. Note that this function must be approximated since the analysis takes place at compile time; section 4.1 discusses how the analysis is implemented in our tool:

Definition 10 (λ-expression Side-effects)**.** Define predicate $\mathit{LSideEffects}(\lambda \mathit{exp})$ on λ-expressions to be true iff $\lambda \mathit{exp}$ modifies a heap location.

For instance, let $\lambda \mathit{exp}$ represent `w -> w > 43.2` from above. Then, we have $\neg \mathit{LSideEffects}(\lambda \mathit{exp})$ since `w` does not represent a heap location. Let $\lambda \mathit{exp}$ represent `s -> results.add(s)` from line 60 of listing 9. Then, we have $\mathit{LSideEffects}(\lambda \mathit{exp})$ since `result` is a heap object add `add()` is a mutating method.

Definition 11 (Stream Side-effects)**.** Define the predicate $\mathit{SSideEffects}(i_s)$ on streams to be true iff i_s is associated with a pipeline whose operations contain a λ-expression with possible side-effects:

$$
\begin{aligned}
&\mathit{SSideEffects}(i_s) \equiv \mathit{LSideEffects}(\lambda(i_s)) \vee \\
&\qquad \exists \texttt{o.m(p)}[i_s \in \mathit{PointsTo}(\texttt{o}) \wedge \texttt{m} \text{ is a term op} \wedge \texttt{p} \text{ is a } \lambda\text{-exp} \wedge \\
&\qquad\qquad \mathit{LSideEffects}(p)] \vee \bigvee_{i_{s_j} \in \mathit{Pred}(i_s)} \mathit{SSideEffects}(i_{s_j})
\end{aligned}
$$

Informally, a stream instance i_s has possible side-effects, i.e., $\mathit{SSideEffects}(i_s)$, iff either a λ-expression used in building i_s, i.e., $\lambda(i_s)$, has side-effects, i.e., $\mathit{LSideEffects}(\lambda(i_s))$, or there exists a call `o.m(p)` such that `o` refers to i_s, i.e., $i_s \in \mathit{PointsTo}(\texttt{o})$, `m` is a terminal operation, and parameter `p` is a λ-expression with possible side-effects, i.e., $\mathit{LSideEffects}(p)$, or if there is a predecessor stream instance i_{s_j} of i_s, i.e., $i_{s_j} \in \mathit{Pred}(i_s)$, that has possible side-effects, i.e., $\mathit{SSideEffects}(i_{s_j})$.

Let i_s be the stream created on lines 59–59 of listing 9, i.e., `filter(s -> pattern.matcher(s).matches())`. Assume that the λ-expression does not contain side-effects. Then, we have:

$$\neg \mathit{LSideEffects}(\lambda(i_s)) \equiv \neg \mathit{LSideEffects}(\texttt{s -> pattern.matcher(s).matches()}).$$

However, consider the terminal operation called on line 60, i.e., `forEach(s -> results.add(s))`. We have that *LSideEffects*(`s -> result.add(s)`). Thus, we have that *SSideEffects*(i_s).

3.7. Determining Whether Reduction Ordering Matters

To obtain a result from stream computations, a terminal (reduction) operation must be issued. Determining whether the ordering of the stream immediately before the reduction matters (ROM) equates to discovering whether the reduction result is the same regardless of whether the stream is ordered or not. In other words, the result of the terminal operation does not depend on the ordering of the stream for which the operation is invoked, i.e., the value when the stream is ordered is equal to the value when the stream is unordered. Some reductions (terminal operations) do not return a value, i.e., they are void returning methods. In these cases, the *behavior* rather than the resulting value should be the same.

Terminal operations fall into two categories, namely, those that produce a result, e.g., `count()`, and those that produce a side-effect, normally by accepting a λ-expression, e.g., `forEach()` [5]. These situations are separately considered, as shown in fig. 5. Here, solid arrows represent data-flow, while dashed arrows are annotations. Figures 5a and 5b describe the two situations.

3.7.1. Non-scalar Result Producing Terminal Operations

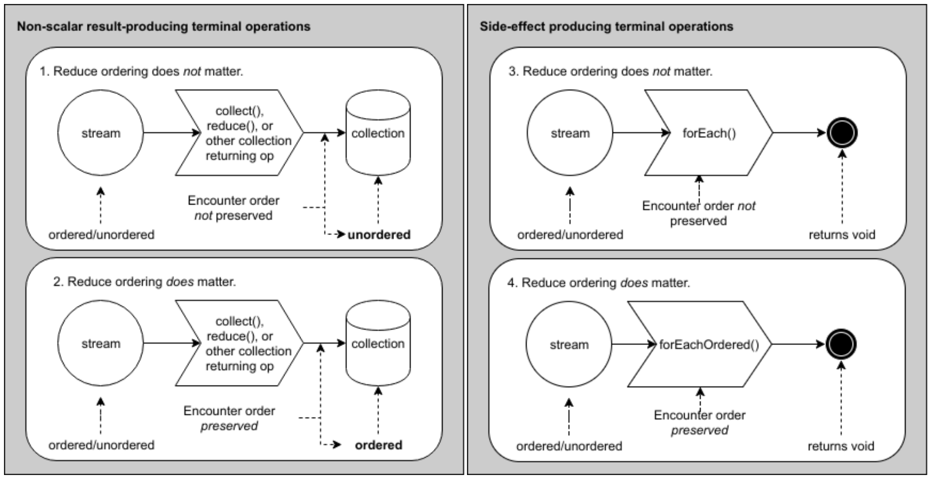
In the case of non-scalar return values, whether the return type maintains ordering is determined by reusing the reflection technique described in section 3.4.2. Specifically, a stream is reflectively derived from an instance of the non-scalar return (run time) type approximations and its characteristics examined. And, from this, whether reduction order matters is determined as follows. If it is impossible for the returned non-scalar type to maintain an element ordering, e.g., it is a `HashSet`, then, the result ordering cannot make a difference in the program's behavior. If, on the other hand, the returned type *can* maintain an ordering, we conservatively determine that the reduction ordering *does* matter.

Table 3: "Reduction ordering matters" (ROM) lookup table. Column **r. type** is the declared return type of the terminal operation in question. Column **ord** is the ordering attribute of the return type. Column **t. operation** is the terminal operation corresponding to the reduction. Column **ROM** is an abbreviation for *Reduce Ordering Matters* and is *true* iff ordering of the result produced by the (terminal) reduction operation must be preserved. Cells whose value is N/A may be either *true* or *false*. A value of '?' represents an unknown value.

r. type	ord	t. operation	ROM
non-scalar	unord	N/A	*F*
non-scalar	ord	N/A	*T*
void	N/A	`forEach()`	*F*
void	N/A	`forEachOrdered()`	*T*
scalar	N/A	`sum()`[*]	*F*
scalar	N/A	`min()`	*F*
scalar	N/A	`max()`	*F*
scalar	N/A	`count()`	*F*
scalar	N/A	`average()`[*]	*F*
scalar	N/A	`summaryStatistics()`[*]	*F*
scalar	N/A	`anyMatch()`	*F*
scalar	N/A	`allMatch()`	*F*
scalar	N/A	`noneMatch()`	*F*
scalar	N/A	`findFirst()`	*T*
scalar	N/A	`findAny()`	*F*
scalar	N/A	`collect()`	?
scalar	N/A	`reduce()`	?

[*] Only applicable to numeric streams.

As before, if there is any inconsistencies between the ordering characteristics of the approximated types, the default is ordered. This is captured in fig. 5a and table 3 under the *non-scalar* rows (column **r. type** is return type). The N/A in column **t. operation** indicates any terminal operation and, in this case, any such operation returning a non-scalar type. The term "collection" refers to any non-scalar type such as those implementing `java.util.Collection` as well as arrays, which are inherently ordered.



(a) For non-scalar result-producing terminal operations.

(b) For side-effect producing terminal operations.

Figure 5: Scenarios for whether reduce ordering matters (ROM).

3.7.2. Side-effect Producing Terminal Operations

When there is a void return value, as is the case with side-effect producing terminal operations, then, we need to know the order in which the stream elements are "served" to the λ-expression argument producing the side-effect. Currently, void terminal operations that maintain element ordering are also a parameter to our analysis. As with determining stateful intermediate operations, a more sophisticated analysis would be needed to possibly approximate this characteristic. In the current Java 8 Stream API, there are only two such methods, namely, `forEach()` and `forEachOrdered()`, as seen in fig. 5b and table 3 under the "void" return type rows.

3.7.3. Scalar Result Producing Terminal Operations

The last case is perhaps the most difficult. While discussing whether non-scalar types (e.g., containers) maintain element ordering seems natural, when the reduction is to a scalar type, it is challenging to determine whether or not the element ordering used to produce the resulting value had any influence over it. Another view of the problem involves determining whether or not the operation(s) “building” the result from the stream are associative. Examples of associative operations include numeric addition, minimum, and maximum, and string concatenation [5]. To address this, we divide the problem into determining the associativity of specialized and general reduction operations.

Specialized Reduction Operations. Luckily, the number and associativity property of specialized reduction operations are fixed. As such, the list of specialized operations along with their associativity property is input to the approach. The reduction order matters (ROM) values compiled by the authors via API documentation examination for the Java 8 Stream API is listed in table 3 under the “scalar” return type rows.

General Reduction Operations. The remaining general reduction operations are `reduce()` and `collect()`. We have already covered the cases where these operations return non-scalar types in the first two rows of table 3. What remains is the cases when these operations return scalar types. Due to the essence of `collect()`, in practice, the result type will most likely fall into the non-scalar category. In fact, `collect()` is a specialization of `reduce()` meant for mutable reductions. Recall from section 2 that such operations collect results in a container such as a collection [5].

The generality of these reduction operations make determining whether ordering matters difficult. For example, even a simple sum reduction can be difficult for an automated approach to analyze. Consider the following code [5] that adds `Widget` weights together using `reduce()`:

```
widgets.stream().reduce(0, (sum, b) -> sum + b.getWeight(), Integer::sum);
```

The first argument is the *identity* element; the second an *accumulator* function, adding a `Widget`’s weight into the accumulated sum. The last argument combines two integer sums by adding them. The question is how, in general, can we tell that this is performing an operation that is associative like summation? In other words, how can we determine that the reducer computation is independent of the order of its inputs? It turns out that this is precisely the *reducer* *commutativity* problem [22]. Unfortunately, this problem has been shown to be undecidable by Chen et al. [22]. While we will consider approximations and/or heuristics as future work, currently, our approach conservatively fails preconditions in this case as indicated by the question marks in table 3. During our experiments detailed in section 4, these failures only accounted for 8.06%.

3.8. Transformation

Once a stream has passed preconditions, there may be multiple possible ways to carry out the corresponding transformation. However, not all transformations may be ones that an expert human developer would have chosen. Here, we strive to select transformations that are (i) semantically equivalent to the original, (ii) exposing the most possible parallelism, and (iii) minimal, i.e., requiring the least amount of code changes. This last point reduces invasiveness.

Stream pipelines, i.e., method call chains of intermediate operations ending in a terminal operation, can be complex with chains possibly spanning multiple branches, methods, and even files. To assist in the transformation, we leverage the *Pred* relation from definition 4 by building a *predecessor tree* PT, where each node represents a stream instance (call site), an edge between nodes n_i and n_j exists iff $n_j \in \mathit{Pred}(n_i)$, and the root is a node n_0 such that $\forall n \in PT[n_0 \in \mathit{Origins}(n)]$ (see definition 8). A separate tree exists for each for each origin stream in the program. Origin streams are also those that are identified for transformation, thus, the transformation algorithm begins at the root of each tree if a transformation applies to the stream represented by the root.

3.8.1. Execution Mode

Figure 6a depicts a predecessor tree for the code snippet in listing 10a, while algorithm 1 depicts the algorithm for transforming a stream to parallel (transformation to sequential is similar). Steps for already parallel streams are shown for completeness. The action at line 10 is valid because intermediate operations like `parallel()` are processed lazily, i.e., when a terminal operation has been issued. As such, "[t]he most recent sequential or parallel mode setting

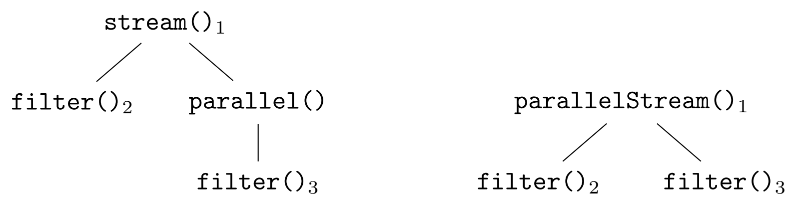
Algorithm 1 Convert stream to parallel

```
for all n ∈ PT such that n is a leaf do
  curr ← n
  while curr ≠ NIL do
    if Method(curr) = sequential() then
      Schedule curr for removal.
    else if Method(curr) = parallel() then
      if ∀a ∈ Ancestors(curr)[|Children(a)| > 1] then {Nodes up from curr to the root
      have multiple children}
        Schedule curr for removal. {To avoid redundancy.}
      else {There is a straight-line "chain" from curr to the root}
        break {parallel() remains with no further modification.}
      end if
    else if Method(curr) = stream() then {Parent(curr) = NIL}
      Schedule curr to be replaced by parallelStream().
    else if Method(curr) ≠ parallelStream() then {curr is not already parallel}
      Schedule parallel() to be inserted immediately after curr.
    end if
    curr ← Parent(curr)
  end while
end for
Execute all scheduled transformations.
```

applies to the execution of the entire stream pipeline" [48]. *Ancestors* is defined on a node n as follows:

$$Ancestors(n) = \begin{cases} \varnothing & \text{if } n = \textbf{NIL} \vee \\ & Parent(n) = \textbf{NIL} \\ Parent(n) \cup Ancestors(Parent(n)) & \text{o.w.} \end{cases}$$

Figure 6b shows the resulting predecessor tree after applying algorithm 1 to the predecessor tree in fig. 6a, while listing 10b is the transformed code.



(a) Before transformation. (b) After conversion to parallel.

Figure 6: Predecessor tree for listing 10a.

3.8.2. Unordering

Unordering a stream, i.e., actions taken for streams passing P3 (table 1) or P4 (table 2), is somewhat similar to altering its execution mode (above) but with some important differences and special considerations. Firstly, although stream execution mode can be changed at the origin stream by replacing the appropriate API call (e.g., `stream()` to `parallelStream()`), since stream ordering can be dependent on its source collection type, for semantics preservation and to limit refactoring invasiveness, unordering does not occur in a similar way. Instead, unordering transformations always take place via a call to the `unordered()` intermediate operation (e.g., line 43 in listing 7b).

While the unordering transformation can be accomplished similar to algorithm 1 by substituting `parallel()` with `unordered()` and `sequential()` with `sorted()`, there are some special considerations regarding the insertion of `unordered()`. For instance, to maximize efficient parallel computation, such calls are inserted before all stateful intermediate operations. This can be seen on line 43 in listing 7b, where `unordered()` is placed before `distinct()`, a stateful intermediate operation.

4. Evaluation

4.1. Implementation

Our approach was implemented as a publicly available, open source Eclipse IDE [37] plug-in [34] and built upon WALA [38] and SAFE [39]. Eclipse is leveraged for its extensive refactoring support [49] and that it is completely open-source for all Java development. WALA is used for static analyses such as side-effect analysis (ModRef), and SAFE, which depends on WALA, for its typestate analysis. SAFE was altered for programmatic use and "intermediate" typestates (cf. section 3.5.2). For the refactoring portion, Eclipse ASTs with source symbol bindings are used as an intermediate representation (IR), while the static analysis consumes a Static Single Assignment (SSA) [50] form IR.

Per the discussion in section 3.4.2, since stream ordering may depend on the stream's source run time type, to determine stream ordering, our implementation interprocedurally approximates (using a points-to analysis) the run time type of stream sources via type propagation using the iterative fixed-point solver available in WALA. If the type cannot be determined accurately in this way, the type's ordering is defaulted to ordered. Although this may cause missed optimization opportunities, an ordered attribute will not cause our approach to take action, guaranteeing semantics preservation.

Once the possible stream source type(s) has been obtained, reflection is used to determine ordering attributes. First, built-in reflection mechanisms are utilized (i.e., `Class.newInstance()`). However, this can be problematic when either a default (no-arg) constructor does not exist or is not accessible. In such cases, Objenesis [51], a tool normally used for Mock Objects, is used to bypass constructor calls. Ordering is retrieved by obtaining a stream from an instance of type (again, via reflection) and subsequently calling the `characteristics()` method on the newly created stream instance's `Spliterator` [43].

The tool maintains a list of stateful intermediate operations and whether reduction order matters for terminal operations (table 3). This may hinder the tool's extensibility in the case that future API versions include additional operations and where third-party stream libraries are used. Section 6 discusses plans to have this done more flexibly.

As discussed in section 3.5, our approach utilizes a k-CFA call graph construction algorithm. To make our experiments tractable and to treat client-side API invocations as stream creations (since the focus of this work is on manipulation of client code), we made k an input parameter to our analysis (with

$k = 2$ being the default as it is the minimum k value to consider client-code) for methods returning streams and $k = 1$ elsewhere. Recall that k amounts to the call string length in which to approximate object instances, thus, $k = 1$ would consider constructor calls as object creation locations, while $k = 2$ would consider calls to methods calling constructors as ("client") object creation sites. The tool currently uses a heuristic to inform developers when k is too small via a precondition failure. It does so by checking that call strings include at least one client method starting from the constructor call site. Future work involves automatically determining an optimal k, perhaps via stochastic optimization. The call graph used in the typestate analysis is pruned by removing nodes that do not have reaching stream definitions.

4.2. Experimental Evaluation

Our evaluation involved studying 18 open source Java applications and libraries of varying size and domain (table 4). Subjects were also chosen such that they are using Java >= 8 and have at least one stream declaration (i.e., a call to a stream API) that is reachable from an entry point (i.e., a candidate stream). Column **KLOC** denotes the thousands of source lines of code, which ranges from ∼1K for monads to ∼586 for elasticsearch. Column **eps** is the number of entry points. For non-library subjects, all main methods were chosen, otherwise, all unit test methods were chosen as entry points. Column **k** is the maximum k value used (see section 4.2.1). Subjects compiled correctly and had identical unit test results and compiler warnings before and after the refactoring.

The analysis was executed on an Intel Xeon E5 machine with 16 cores and 60GB RAM and a 55GB maximum heap size. Column **tm (m)** is the running time in minutes, averaging ∼34.04 secs/KLOC. The running time ranges from 0.05m to 590.43m, with the latter being for spring-framework. We consider spring-framework to be an outlier regarding running time as it is an abnormally large and complex framework. Furthermore, because it has the largest amount of entry points (which correspond to unit tests for frameworks) at 5,981, we

Table 4: Experimental results. Column **subject** is the studied project, column **KLOC** is the project's thousands of source lines of code, column **eps** is the total number of entry points used in the analysis, column **k** is the maximum k in the subject used to build the nCFA, column **str** is the total number of syntactic streams, i.e., those appearing in the source code, column **cnd** is the total number of (origin) streams reachable from the entry points, column **rft** is the total number of (origin) streams that are optimizable, columns **P*** are streams passing the respected preconditions, and column **t (m)** is the total processing time in minutes.

subject	KLOC	eps	k	str	cnd	rft	P1	P2	P3	t (m)
bootique	4.91	1,391	4	68	34	10	2	8	0	144.72
cryptomator	7.99	148	3	13	4	0	0	0	0	2.26
dari	64.86	3	2	19	4	0	0	0	0	1.76
elasticsearch	585.71	141	13	250	80	12	0	12	0	118.00
htm.java	41.14	21	4	189	34	10	0	10	0	1.85
JabRef	138.83	76	6	305	79	6	0	6	0	9.41
JacpFX	23.79	195	4	54	4	3	3	0	0	2.31
jdp[a]	19.96	25	4	38	28	15	1	13	1	31.88
jdk8-exp[b]	3.43	134	4	55	26	4	0	4	0	0.78
jetty	354.48	106	4	65	21	7	3	4	0	17.85
JetUML	20.95	660	2	7	7	2	0	2	0	0.76
jOOQ	154.01	43	4	24	5	1	0	1	0	12.94
koral	7.13	51	3	8	6	6	0	6	0	1.06
monads	1.01	47	2	3	1	1	0	1	0	0.05
retroλ	5.14	1	4	12	8	6	3	3	0	0.66
spring[c]	188.46	5,981	4	61	54	29	0	29	0	590.43
streamql	4.01	92	2	22	22	2	0	2	0	0.72
threeten[d]	27.53	36	2	2	2	2	0	2	0	0.51
Total	1,653.35	9,151	13	1,160	419	116	12	103	1	937.94

[a] jdp is java-design-patterns.

[b] jdk8-exp is jdk8-experiments.

[c] spring is a portion of spring-framework.

[d] threeten is threeten-extra.

hypothesize that this, along with 188.46K LOC and 54 streams, contributed to a substantially larger running time than the other subjects.

Thus, not including spring-framework, the average run time in secs/KLOC is ~14.23. In our original conference paper [40], where only 11 subjects were studied, this value was ~6.60. While examining this discrepancy more closely, we found that the 6 (non-spring-framework) subjects had significantly more entry points that the original 11. The average number of entry points per subject for the initial corpus and the new corpus was ~68.27 and ~176.11, respectively. Again, we suspect that the increase in entry points caused the additional run time per KLOC. After a closer investigation, we found that the secs/KLOC/entry point to be comparable between the two corpora, namely, ~0.00879 for the original and ~0.00449 for the new subjects.

Lastly, an examination of three of the subjects revealed that over 80% of the running time was for the typestate analysis, which is performed by SAFE. This analysis incorporates aliasing information and can be lengthy for larger applications. Unfortunately, SAFE is not actively being maintained, and it is difficult to say whether its performance can be improved. However, since our approach is automated, it can be executed on a nightly basis or before major releases.

4.2.1. Setting k for the k-CFA

As discussed in section 3.5, our approach takes as input a maximum call string length parameter k, which is used to construct the call graph using nCFA. Each call graph node is associated with a *context*, which, in our case, is the call string. This allows our analysis to approximate stream object creation in the *client* code rather than in the framework, where the stream objects are instantiated. Otherwise, multiple calls to the same API methods that create streams would be considered as creating one new stream.

During our experiments, a default k value of 2 was used. This is the minimum k value that can be used to distinguish client code from framework stream creation. However, depending on which stream framework methods are utilized in a particular project, this value may be insufficient. We detect this situation via a heuristic of examining the call string and determining whether any client code exists. If not, k may be too small.

Setting k constitutes a trade-off. A k that is too small will produce correct results but may miss streams. A larger k may enable the tool to detect and subsequently analyze more streams but may increase run time. Thus, an optimal k value can be project-specific. In our experiments, however, we determined k empirically based on a balance between run time and the ratio between total (syntactically available) streams and candidate streams (i.e., those detected by the typestate analysis). Notwithstanding, in keeping k between 2 and 4 (cf. table 4), good results and reasonable runtime were observed for most projects. Thus, it was not difficult to find an "effective" k.

4.2.2. Intelligent Parallelization

Streams are still relatively new, and, as they grow in popularity, we expect to see them used more widely. Nevertheless, we analyzed 419 (origin) candidate streams reachable from entry points (column **cnd**; column **str** is the number of syntactically available streams, which include unreachable streams) across 18 subjects. Of those, we automatically refactored ∼27.68% (column **rft** for *refactorable*) despite being highly conservative. These streams are the ones that have passed all preconditions; those not passing preconditions were not transformed (cf. table 5).

Columns **P1–3** are the streams passing the corresponding preconditions (cf. tables 1 and 2). Columns **P4–5** have been omitted as all of their values are 0. The number of transformations can be derived from these columns as preconditions are associated with transformations, amounting to $12+103+(1*2) = 117$.

4.2.3. Refactoring Failures

Table 5 categorizes reasons why streams could not be refactored (column **failure**), some of which correspond directly to preconditions (column **pc**). Column **cnt** depicts the count of failures in the respective category and further

Table 5: Refactoring failures.Column **failure** is the failure category, column **pc** is the corresponding precondition from tables 1 and 2, and column **cnt** is the count of precondition failures in the corresponding category.

failure	pc	cnt
F1. Inconsistent possible execution modes		1
F2. No stateful intermediate operations	P5	1
F3. No terminal operation		18
F4. Reduce ordering matters	P3	19
F5. Indeterminable reduction ordering		25
F6. Has side-effects	P1	4
	P2	85
F7. Currently not handled		156
Total		310

categorized by precondition, if applicable. Nontrivial reasons streams were not refactorable include λ-expression side-effects (F6, 28.71%) and that the reduction ordering is preserved by the target collection (F4, 6.13%; c.f. section 2). Not only do the refactoring failures shed light on the applicability of the approach to real-world software, but they also provide insight into the attributes of the software and how developers write code that is either amenable or not amenable to parallelization.

The majority of the refactoring failures were due to cases currently not handled by our tool (F7, 50.32%), which are rooted in implementation details related to model differences between representations [34]. For example, streams declared inside inner (embedded) classes are problematic as such classes are part of the outer AST but the instruction-based IR is located elsewhere. Though we plan to develop more sophisticated mappings in the future, further investigation revealed that 76.28% of the failures stemmed from only two subjects, namely, JabRef and elasticsearch. For the remaining subjects, this failure only encompassed an average of 2.64%. Moreover, our tool was still able to refactor 18 streams over JabRef and elasticsearch.

Table 6: Average run times of JMH benchmarks.Column **benchmark** is the benchmark name. Column **orig** is the original code in seconds per operation. Column **refact** is the refactored code, also in seconds per operation. Column **su** is the speedup.

#	benchmark	orig (s/op)	refact (s/op)	su
1	shouldRetrieveChildren	0.011 (0.001)	0.002 (0.000)	6.57
2	shouldConstructCar	0.011 (0.001)	0.001 (0.000)	8.22
3	addingShouldResultInFailure	0.014 (0.000)	0.004 (0.000)	3.78
4	deletionShouldBeSuccess	0.013 (0.000)	0.003 (0.000)	3.82
5	addingShouldResultInSuccess	0.027 (0.000)	0.005 (0.000)	5.08
6	deletionShouldBeFailure	0.014 (0.000)	0.004 (0.000)	3.90
7	specification.AppTest.test	12.666 (5.961)	12.258 (1.880)	1.03
8	CoffeeMakingTaskTest.testId	0.681 (0.065)	0.469 (0.009)	1.45
9	PotatoPeelingTaskTest.testId	0.676 (0.062)	0.465 (0.008)	1.45
10	SpatialPoolerLocalInhibition	1.580 (0.168)	1.396 (0.029)	1.13
11	TemporalMemory	0.013 (0.001)	0.006 (0.000)	1.97

Other refactoring failures include F3 (5.81%), where stream processing does not end with a terminal operation in all possible executions. This amounts to "dead" code as any queued intermediate operations will never execute. F5 corresponds to the situation described in section 3.7.3 (8.06%), F1 to the situation where execution modes are ambiguous on varying execution paths (0.32%), and F2 means that the stream is already optimized (0.65%).

4.2.4. Performance Evaluation

Many factors can influence performance, including dataset size, number of available cores, JVM and/or hardware optimizations, and other environmental activities. Nevertheless, we assess the performance impact of our refactoring. Although this assessment is focused on our specific refactoring and subject projects, in the general case, it has been shown that a similar refactoring done manually has improved performance by 50% on large datasets using four cores [52, Ch. 6].

Existing Benchmarks. We assessed the performance impact of our refactoring on the subjects listed in table 4. One of the subjects, htm.java [53], has formal performance tests utilizing a standard performance test harness, namely, the Java

Microbenchmark Harness (JMH) [54]. Using such a test harness is important in isolating causes for performance changes to the code changes themselves [52, Ch. 6.1]. As such, subjects with JMH tests will produce the best indicators of performance improvements. Two such tests were included in this subject.

Converted Benchmarks. Although the remainder of the subjects did not include formal performance tests, they did include a rich set of unit tests. For one subject, namely, java-design-patterns [55], we methodically transformed existing JUnit tests that covered the refactored code to proper JMH performance tests. This was accomplished by annotating existing `@Test` methods with `@Benchmark`, i.e., the annotation that specifies that a method is a JMH performance test. We also moved setup code to `@Before` methods, i.e., those that execute before each test, and annotated those with `@Setup`. This ensures that the test setup is not included in the performance assessment. Furthermore, we chose unit tests that did not overly involve I/O (e.g., database access) to minimize variability. In all, nine unit tests were converted to performance tests and made our changes available to the subject developers.

Augmenting Dataset Size. As all tests we designed for continuous integration (CI), they executed on a minimal amount of data. To exploit parallelism, however, we augmented test dataset sizes. For existing benchmarks, this was done under the guidance of the developers [56]. For the converted tests, we chose an N (dataset size) value that is consistent with that of the largest value used by Naftalin [52, Ch. 6]. In this instance, we preserved the original unit test assertions, which all passed. This ensures that, although N has increased, the spirit of the test, which may reflect a real-life scenario, remains intact.

Results. Table 6 reports the average run times of five runs in seconds per operation following five warm-up runs. Rows 1–9 are for java-design-patterns, while rows 10–11 are for htm.java; benchmark names have been shortened for brevity. Column **orig** is the original program, **refact** is the refactored program, and **su** is the speedup ($runtime_{old}/runtime_{new}$). Values associated with parentheses are averages, while the value in parenthesis is the corresponding standard deviation. The average speedup resulting from our refactoring is 3.49.

4.2.5. Discussion

The findings of Naftalin [52, Ch. 6] using a similar manual refactoring, that our tool was able to refactor 27.68% of candidate streams (table 4), and the results of JMH tests on the refactored code (table 6) combine to form a reasonable motivation for using our approach in real-world situations. Moreover, this study gives us insight into how streams, and in a broader sense, concurrency, are used, which can be helpful to language designers, tool developers, and researchers.

As mentioned in section 4.2.2, columns **P4–5** in table 4 all have 0 values. Interestingly, this means that no (already) parallel streams were refactored by our tool. Only 13 candidate streams, stemming from only two subjects, namely, htm.java and JabRef, were originally parallel. This may indicate that developers are either timid to use parallel streams because of side-effects, for example, or are (manually) unaware of when using parallel streams would improve performance [52]. This further motivates our approach for automated refactoring in this area.

From table 5, besides F7, F4 and F6 accounted for one of the largest percentage of failures (34.84%). For the latter, this may indicate that despite that "many computations where one might be tempted to use side-effects can be more safely and efficiently expressed without side-effects" [5], in practice, this is either not the case or more developer education is necessary to avoid side-effects when using streams. This motivates future work in refactoring stream code to avoid side-effects if possible. Section 6 discusses future work to mitigate F7 and F5.

Imprecision is also a possibility as we are bound by the conservativeness of the underlying ModRef analysis provided by WALA. To investigate, we manually examined 45 side-effect failures and found 11 false positives. Several subject developers, on the other hand, confirmed correct refactorings, as discussed in section 4.2.6. As for the former, a manual inspection of these sites may be necessary to confirm that ordering indeed must be preserved. If not, developers can rewrite the code (e.g., changing `forEachOrdered()` to `forEach()`) to exploit more parallelism opportunities.

The average speedup of 1.55 obtained from htm.java (benchmarks 10–11) most likely reflects the parallelism opportunities available in computationally intensive programs [57]. Benchmarks 1–6, which had good speedups as well, also mainly deal with data. Benchmark 7 had the smallest speedup at 1.03. The problem is that the refactored code appears in areas that "will not benefit from parallelism" [58], demonstrating a limitation of our approach that is rooted in its problem scope. Specifically, our tool locates sites where stream client code is safe to refactor and is possibly optimizable based on language semantics but does not assess optimizability based on input size/overhead trade-offs.

4.2.6. Pull Request Study

To assess our approach's usability, we also submitted several pull requests (patches) containing the results of our tool to the subject projects. Assessing the usefulness of our approach through pull requests, although insightful, has its challenges for program transformation. Particularly, it has been shown that developers cannot always estimate the impact of a transformation [59]. Furthermore, developers generally perceive refactorings and other transformations as a fault-prone activity [60–62]. As such, developers may not accurately decipher the value of the presented transformations immediately. Still, we performed this assessment but only as a part of the overall evaluation.

As of this writing, eight requests were made, with three pending (e.g., [56]) and five rejected. One rejected request [58] is discussed in section 4.2.5. Others (e.g., [55]) confirmed a correct refactoring but only wanted parallel streams when performance is an observed problem. Although three of the requests are still pending, at least one of them has had ongoing discussions.

4.3. Threats to Validity

The subjects may not represent the stream client code usage. To mitigate this, subjects were chosen from diverse domains as well as sizes, as well as those used in previous studies (e.g., [63,64]). Although java-design-patterns is artificial, it is a reference implementation similar to that of JHotDraw, which has been studied extensively (e.g., [65]).

Entry points may not be correct, which would affect which streams are deemed as candidates, as well as the performance assessment as there is a trade-off between scalability and number of entry points. Standard entry points were chosen (see section 4.2), representing a super set of practically true entry points. For the performance test (see table 6), the loads may not be representative of real-world usage. However, we conferred with developers regarding this when possible [56]. For the performance tests we manually generated from unit tests, a systematic approach to the generation was taken using the same parameters (N) on both the original and refactored versions.

The focus of our approach is on client code, i.e., our analysis is agnostic to a particular stream API implementation so as long as it upholds the API specifications. Particular stream API implementations, however, may be fined-tuned to particular platforms (e.g., server vs. application, GPUs [66,67]). As such, developers must manually consider the context in which their streams will execute and the particular stream API implementation they are using, especially if they need fine-grained performance tuning. In general, developers should consider several factors when deciding on a stream execution mode, including execution context, workload, and spliterator and collector performance [52, Ch. 6.2]. Although Java is a portable language, future work consists of incorporating more developer input as to the expected factors governing the execution of the code into the refactoring algorithm in order to make more informed decisions in transforming stream execution modes automatically.

5. Related Work

Automatic parallelization can occur on several levels, including the compiler [68,69], run time [70], and source [19]. The general problem of full automatic parallelization by compilers is extremely complex and remains a grand challenge [71]. Many attempt to solve it in only certain contexts, e.g., for divide and conquer [72], recursive functions [73], distributed architectures [74], graphics processing [75], matrix manipulation [76], asking the developer for assistance [77], and speculative strategies [78]. Our approach focuses on MapReduce-style code over native data containers in a shared memory space using a mainstream programming languages, which may be more amenable to parallelization due to more explicit data dependencies [18]. Moreover, our approach can help detect when it is *not* advantageous to run code in parallel, and when unordering streams can possibly improve performance.

Techniques other than ours enhance the performance of streams as well. Hayashi et al. [66] develop a supervised machine-learning approach for building performance heuristics for mapping Java applications onto CPU/GPU accelerators via analyzing parallel streams. Ishizaki et al. [67] translate λ-expressions in parallel streams into GPU code and automatically generates run time calls that handle low-level operations. While all these approaches aim to improve performance, their input is streams that are *already* parallel. As such, developers must still *manually* identify and transform sequential streams. Nonetheless, these approaches may be used in conjunction with ours. Khatchadourian et al. [29] focus on the use of streams by studying their amenability to parallelization in particular contexts, the kinds operations invoked on streams, and bugs specific and tangential to using streams.

Harrison [79] develops an interprocedural analysis and automatic parallelization of Scheme programs. While Scheme is a multi-paradigm language, and shared memory is modeled, their transformations are more invasive and imperative-focused, involving such transformations as eliminating recursion and loop fusion. Nicolay et al. [80] have a similar aim but are focused on analyzing side-effects, whereas we analyze ordering constraints.

Many approaches use streams for other tasks or enhance streams in some way. Cheon et al. [81] use streams for JML specifications. Biboudis et al. [1] develop "extensible" pipelines that allow stream APIs to be extended without changing library code. Stein et al. [82] use a type-based approach that statically ensures the thread-safety of streams that access UI threads. Other languages, e.g., Scala [2], JavaScript [3], C# [4], also offer streaming APIs. While we focus on Java 8 streams, the concepts set forth here may be applicable to other situations, especially those involving statically-typed languages, and is a topic for future work.

Other approaches refactor programs to either utilize or enhance modern construct usage. Gyori et al. [18] refactor Java code to use λ-expressions instead of imperative-style loops. Tsantalis et al. [83] transform clones to λ-expressions. Khatchadourian and Masuhara [84] refactor skeletal implementations to default methods. Tip et al. [85] use type constraints to refactor class hierarchies, and Gravley and Lakhotia [86] and Khatchadourian [87] refactor programs to use enumerated types.

Typestate has been used to solve many problems. Mishne et al. [88] use typestate for code search over partial programs. Garcia et al. [89] integrate typestate as a first-class citizen in a programming language. Padovani [90] extends typestate oriented programming (TSOP) for concurrent programming. Other approaches have also used hybrid typestate analyses. Bodden [91], for instance, combines typestate with residual monitors to signal property violations at run time, while Garcia et al. [89] also make use of run time checks via gradual typing [90].

6. Conclusion & Future Work

Our automated refactoring approach "intelligently" optimizes Java 8 stream code. It automatically deems when it is safe and possibly advantageous to run stream code either sequentially or in parallel and unorder streams. The approach was implemented as an Eclipse plug-in and evaluated on 18 open source programs, where 116 of 419 candidate streams (27.68%) were refactored. A performance analysis indicated an average speedup of 3.49.

In the future, we plan to handle several issues between Eclipse and WALA models, i.e., to consistently map SSA instructions to AST nodes. One insight is that a machine learning model can be trained to accurately match an SSA instruction with a corresponding AST node but only for cases, e.g., anonymous inner classes, where the lookup failures using our currently heuristics. We also plan to incorporate more kinds of (complex) reductions like those involving maps, details of which have been published in an accompanying technical report [92]. Implementation challenges here deal with extending the ordering inference approach to deal with so-called "embedded" collections, e.g., `Map`s may have multiple orderings, that of the map entries themselves and that of the value in the case that it is *also* a collection.

Other plans include examining approximations to combat the problems set forth by Chen et al. [22], perhaps using a conservative data-flow analysis to track λ-expressions involved in reductions. Approximating stateful intermediate operations and whether reduction ordering matters may also involve heuristics, e.g., dealing with the underlying stream framework code or analysis of API documentation. We will also explore applicability to other streaming frameworks and languages. Furthermore, we will explore how the generalized typestate analysis presented in section 3 can more broadly apply to other fluent APIs [93, Ch. 4.1].

There is a possibility that the refactored code, as a result of the imposed transformation, can be further optimized to reduce redundant and unnecessary code to improve comprehension and maintainability. For example, in listing 10b, both the `if` and `else` branches contain exactly the same code. As such, the conditional statements can be eliminated, leaving behind a single "then" portion. Consequently, the parameter `x` is also unneeded. We intend to explore the application of composite refactorings (e.g., Remove Duplicate Code, Remove Unused Parameter), perhaps by applying the techniques of Fontana et al. [94], in the future to further improve the refactored code.

Acknowledgments

We would like to thank Atanas Rountev, Eric Bodden, Eran Yahav, Ameya Ketkar, and anonymous reviewers for their insightful feedback and for referring us to related work. Support for this project was provided by PSC-CUNY Award #61793-00 49, jointly funded by The Professional Staff Congress and The City University of New York. Bagherzadeh was supported by Oakland University.

References

[1] Aggelos Biboudis, Nick Palladinos, George Fourtounis, and Yannis Smaragdakis. "Streams a la carte: Extensible Pipelines with Object Algebras". In: *European Conference on Object-Oriented Programming*. 2015, pp. 591–613. DOI: `10.4230/LIPIcs.ECOOP.2015.591`.

[2] EPFL. *Collections–Mutable and Immutable Collections–Scala Documentation*. 2017. URL: `http://scala-lang.org/api/2.12.3/scala/collection/index.html` (visited on 08/24/2018).

[3] Refsnes Data. *JavaScript Array map() Method*. 2015. URL: `http://w3schools.com/jsref/jsref_map.asp`.

[4] Microsoft. *LINQ: .NET Language Integrated Query*. 2018. URL: `http://msdn.microsoft.com/en-us/library/bb308959.aspx` (visited on 08/24/2018).

[5] Oracle. *java.util.stream (Java SE 9 & JDK 9)*. 2017. URL: `http://docs.oracle.com/javase/9/docs/api/java/util/stream/package-summary.html`.

[6] James Lau. "Future of Java 8 Language Feature Support on Android". In: *Android Developers Blog* (Mar. 14, 2017). URL: `https://android-developers.googleblog.com/2017/03/future-of-java-8-language-feature.html` (visited on 08/24/2018).

[7] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. DOI: `10.1145/1327452.1327492`.

[8] Mehdi Bagherzadeh and Raffi Khatchadourian. "Going Big: A Large-scale Study on What Big Data Developers Ask". In: *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. ACM. Tallinn, Estonia: ACM, 2019, pp. 432–442. ISBN: 978-1-4503-5572-8. DOI: `10.1145/3338906.3338939`. URL: `http://doi.acm.org/10.1145/3338906.3338939`.

[9] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. “Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2008, pp. 329–339. DOI: 10.1145/1346281.1346323.

[10] Syed Ahmed and Mehdi Bagherzadeh. “What Do Concurrency Developers Ask About?: A Large-scale Study Using Stack Overflow”. In: *International Symposium on Empirical Software Engineering and Measurement*. 2018, 30:1–30:10. DOI: 10.1145/3239235.3239524.

[11] Mehdi Bagherzadeh and Hridesh Rajan. “Order Types: Static Reasoning About Message Races in Asynchronous Message Passing Concurrency”. In: *International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. 2017, pp. 21–30. DOI: 10.1145/3141834.3141837.

[12] Yuheng Long, Mehdi Bagherzadeh, Eric Lin, Ganesha Upadhyaya, and Hridesh Rajan. “On Ordering Problems in Message Passing Software”. In: *International Conference on Modularity*. MODULARITY 2016. Málaga, Spain: ACM, 2016, pp. 54–65. ISBN: 978-1-4503-3995-7. DOI: 10.1145/2889443.2889444. URL: http://doi.acm.org/10.1145/2889443.2889444.

[13] Mehdi Bagherzadeh and Hridesh Rajan. “Panini: A Concurrent Programming Model for Solving Pervasive and Oblivious Interference”. In: *International Conference on Modularity*. MODULARITY 2015. Fort Collins, CO, USA: ACM, 2015, pp. 93–108. ISBN: 978-1-4503-3249-1. DOI: 10.1145/2724525.2724568. URL: http://doi.acm.org/10.1145/2724525.2724568.

[14] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, and Syed Ahmed. “Towards Safe Refactoring for Intelligent Parallelization of Java 8 Streams”. In: *International Conference on Software Engineering: Companion Proceedings*. ICSE ’18. ACM/IEEE. Gothenburg, Sweden: ACM, May 2018, pp. 206–207. ISBN: 978-1-4503-5663-3. DOI: 10.1145/3183440.3195098.

[15] Oracle. *Thread Interference*. 2017. URL: http://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html (visited on 04/16/2018).

[16] Richard Warburton. *Java 8 Lambdas: Pragmatic Functional Programming*. 1st ed. Apr. 7, 2014, p. 182. ISBN: 1449370772.

[17] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. "Understanding the Use of Lambda Expressions in Java". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017), 85:1–85:31. ISSN: 2475-1421. DOI: 10.1145/3133909.

[18] Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. "Crossing the Gap from Imperative to Functional Programming Through Refactoring". In: *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ACM SIGSOFT. 2013, pp. 543–553. DOI: 10.1145/2491411.2491461.

[19] Danny Dig, John Marrero, and Michael D. Ernst. "Refactoring sequential Java code for concurrency via concurrent libraries". In: *International Conference on Software Engineering.* IEEE, 2009, pp. 397–407. DOI: 10.1109/ICSE.2009.5070539.

[20] Etienne Brodu, Stéphane Frénot, and Frédéric Oblé. "Transforming JavaScript Event-loop into a Pipeline". In: *Symposium on Applied Computing.* 2016, pp. 1906–1911. DOI: 10.1145/2851613.2851745.

[21] Cosmin Radoi, Stephen J. Fink, Rodric Rabbah, and Manu Sridharan. "Translating Imperative Code to MapReduce". In: *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.* ACM. 2014.

[22] Yu-Fang Chen, Chih-Duo Hong, Nishant Sinha, and Bow-Yaw Wang. "Commutativity of Reducers". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* 2015, pp. 131–146. DOI: 10.1007/978-3-662-46681-0_9.

[23] Tian Xiao, Jiaxing Zhang, Hucheng Zhou, Zhenyu Guo, Sean McDirmid, Wei Lin, Wenguang Chen, and Lidong Zhou. "Nondeterminism in MapReduce Considered Harmful? An Empirical Study on Non-commutative Aggregators in MapReduce Programs". In: *ICSE Companion.* 2014, pp. 44–53. DOI: 10.1145/2591062.2591177.

[24] Christoph Csallner, Leonidas Fegaras, and Chengkai Li. "Testing MapReduce-style Programs". In: *Joint Meeting on European Software Engineering Confer-*

ence and Symposium on the Foundations of Software Engineering. 2011, pp. 504–507. DOI: 10.1145/2025113.2025204.

[25] Fan Yang, Wen Su, Huibiao Zhu, and Qin Li. "Formalizing MapReduce with CSP". In: *International Conference and Workshops on the Engineering of Computer-Based Systems*. IEEE. Oxford, UK, Mar. 2010, pp. 358–367. DOI: 10.1109/ECBS.2010.50.

[26] Sebastian Nielebock, Robert Heumüller, and Frank Ortmeier. "Programmers Do Not Favor Lambda Expressions for Concurrent Object-oriented Code". In: *Empirical Softw. Engg.* 24.1 (Feb. 2019), pp. 103–138. ISSN: 1382-3256. DOI: 10.1007/s10664-018-9622-9.

[27] Walter Lucas, Rodrigo Bonifácio, Edna Dias Canedo, Diego Marcílio, and Fernanda Lima. "Does the Introduction of Lambda Expressions Improve the Comprehension of Java Programs?" In: *Brazilian Symposium on Software Engineering*. SBES 2019. Salvador, Brazil: ACM, 2019, pp. 187–196. ISBN: 978-1-4503-7651-8. DOI: 10.1145/3350768.3350791.

[28] Shubham Sangle and Sandeep Muvva. "On the Use of Lambda Expressions in 760 Open Source Python Projects". In: *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: ACM, 2019, pp. 1232–1234. ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3342499.

[29] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Baishakhi Ray. "An Empirical Study on the Use and Misuse of Java 8 Streams". In: *International Conference on Fundamental Approaches to Software Engineering*. FASE 2020. To appear. ETAPS. Springer, Apr. 2020. URL: http://academicworks.cuny.edu/hc_pubs/610.

[30] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. "Improving MapReduce Performance in Heterogeneous Environments". In: *Operating Systems Design and Impl.* 2008, pp. 29–42.

[31] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. "HaLoop: Efficient Iterative Data Processing on Large Clusters". In: *Proc. VLDB Endow.* 3.1-2 (Sept. 2010), pp. 285–296. DOI: 10.14778/1920841.1920881.

[32] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. "Automatic Optimization for MapReduce Programs". In: *Proc. VLDB Endow.* 4.6 (Mar. 2011), pp. 385–396. DOI: 10.14778/1978665.1978670.

[33] Rong Gu, Xiaoliang Yang, Jinshuang Yan, Yuanhao Sun, Bing Wang, Chunfeng Yuan, and Yihua Huang. "SHadoop: Improving MapReduce performance by optimizing job execution mechanism in Hadoop clusters". In: *Journal of Parallel and Distributed Computing* 74.3 (2014), pp. 2166–2179. DOI: 10.1016/j.jpdc.2013.10.003.

[34] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. "A Tool for Optimizing Java 8 Stream Software via Automated Refactoring". In: *International Working Conference on Source Code Analysis and Manipulation.* IEEE. Sept. 2018, pp. 34–39. DOI: 10.1109/SCAM.2018.00011.

[35] Robert E Strom and Shaula Yemini. "Typestate: A programming language concept for enhancing software reliability". In: *IEEE Transactions on Software Engineering* SE-12.1 (Jan. 1986), pp. 157–171. DOI: 10.1109/tse.1986.6312929.

[36] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. "Effective Typestate Verification in the Presence of Aliasing". In: *ACM Transactions on Software Engineering and Methodology* 17.2 (May 2008), pp. 91–934. DOI: 10.1145/1348250.1348255.

[37] Eclipse Foundation. *Eclipse IDEs.* Aug. 2018. URL: http://eclipse.org (visited on 08/24/2018).

[38] WALA Team. *T.J. Watson Libraries for Analysis.* June 2015. URL: http://wala.sf.net (visited on 01/18/2017).

[39] Eran Yahav. *SAFE typestate analysis engine.* Aug. 2018. URL: http://git.io/vxwBs (visited on 08/24/2018).

[40] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. "Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams". In: *International Conference on Software Engineering.* ICSE '19. IEEE Press, 2019, pp. 619–630. DOI: 10.1109/ICSE.2019.00072.

[41] Oracle. *HashSet (Java SE 9) & JDK 9.* 2017. URL: http://docs.oracle.com/javase/9/docs/api/java/util/HashSet.html (visited on 04/07/2018).

[42] Oracle. *Stream (Java Platform SE 9 & JDK 9)–Concatenation*. 2017. URL: http://docs.oracle.com/javase/9/docs/api/java/util/stream/Stream.html#concat-java.util.stream.Stream-java.util.stream.Stream-.

[43] Oracle. *Spliterator (Java SE 9 & JDK 9)*. 2017. URL: https://docs.oracle.com/javase/9/docs/api/java/util/Spliterator.html (visited on 08/24/2018).

[44] Bjarne Steensgaard. "Points-to analysis in almost linear time". In: *Principles of Programming Languages*. 1996, pp. 32–41.

[45] Olin Grigsby Shivers. "Control-flow Analysis of Higher-order Languages of Taming Lambda". PhD thesis. Pittsburgh, PA, USA: Carnegie Mellon University, 1991.

[46] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. 2nd ed. Secaucus, NJ, USA: Springer-Verlag, 2004. ISBN: 3540654100.

[47] Philip Wadler. "Linear types can change the world". In: *IFIP TC*. Vol. 2. 1990, pp. 347–359.

[48] Oracle. *java.util.stream (Java SE 9 & JDK 9)–Parallelism*. 2017. URL: https://docs.oracle.com/javase/9/docs/api/java/util/stream/package-summary.html#Parallelism.

[49] Dirk Bäumer, Erich Gamma, and Adam Kiezun. "Integrating refactoring support into a Java development tool". Oct. 2001. URL: http://people.csail.mit.edu/akiezun/companion.pdf.

[50] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Global Value Numbers and Redundant Computations". In: *Symposium on Principles of Programming Languages*. ACM SIGPLAN-SIGACT. 1988, pp. 12–27. DOI: 10.1145/73560.73562.

[51] EasyMock. *Objenesis*. 2017. URL: http://objenesis.org (visited on 08/24/2018).

[52] Maurice Naftalin. *Mastering Lambdas: Java Programming in a Multicore World*. McGraw-Hill, 2014. ISBN: 0071829628.

[53] Numenta. *Hierarchical Temporal Memory implementation in Java*. 2018. URL: http://git.io/fNbnK.

[54] Oracle. *JMH*. 2018. URL: http://openjdk.java.net/projects/code-tools/jmh (visited on 03/30/2018).

[55] Ilkka Seppälä. *Design patterns implemented in Java*. 2018. URL: `http://git.io/v5lko` (visited on 08/24/2018).

[56] David Ray. *Pull Request #539–numenta/htm.java*. Mar. 2018. URL: `http://git.io/fAqDq` (visited on 03/21/2018).

[57] M. Kumar. "Measuring parallelism in computation-intensive scientific/engineering applications". In: *IEEE Transactions on Computers* 37.9 (Sept. 1988), pp. 1088–1098. ISSN: 0018-9340. DOI: `10.1109/12.2259`.

[58] Esko Luontola. *Pull Request #140–orfjackal/retrolambda*. Mar. 2018. URL: `http://git.io/fAqHz`.

[59] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. "Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells". In: *International Conference on Software Maintenance and Evolution*. ICSME '14. USA: IEEE Computer Society, 2014, pp. 101–110. ISBN: 9781479961467. DOI: `10.1109/ICSME.2014.32`.

[60] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. "When Does a Refactoring Induce Bugs? An Empirical Study". In: *International Working Conference on Source Code Analysis and Manipulation*. SCAM '12. USA: IEEE Computer Society, 2012, pp. 104–113. ISBN: 9780769547831. DOI: `10.1109/SCAM.2012.20`.

[61] Sarra Habchi, Romain Rouvoy, and Naouel Moha. "On the Survival of Android Code Smells in the Wild". In: *International Conference on Mobile Software Engineering and Systems*. MOBILESoft '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 87–98.

[62] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. "An Empirical Study of Refactoring Challenges and Benefits at Microsoft". In: *IEEE Transactions on Software Engineering* 40.7 (July 2014), pp. 633–649. ISSN: 0098-5589. DOI: `10.1109/TSE.2014.2318734`.

[63] Raffi Khatchadourian and Hidehiko Masuhara. "Proactive Empirical Assessment of New Language Feature Adoption via Automated Refactoring: The Case of Java 8 Default Methods". In: *International Conference on the Art, Science, and Engineering of Programming*. 2018, 6:1–6:30. DOI: `10.22152/programming-journal.org/2018/2/6`.

[64] Ameya Ketkar, Ali Mesbah, Davood Mazinanian, Danny Dig, and Edward Aftandilian. "Type Migration in Ultra-large-scale Codebases". In: *International Conference on Software Engineering*. ICSE '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 1142–1153. DOI: 10.1109/ICSE.2019.00117.

[65] Marius Marin, Leon Moonen, and Arie van Deursen. "An Integrated Crosscutting Concern Migration Strategy and its Application to JHotDraw". In: *International Working Conference on Source Code Analysis and Manipulation*. 2007.

[66] Akihiro Hayashi, Kazuaki Ishizaki, Gita Koblents, and Vivek Sarkar. "Machine-Learning-based Performance Heuristics for Runtime CPU/GPU Selection". In: *Principles and Practices of Programming on The Java Platform*. 2015, pp. 27–36. DOI: 10.1145/2807426.2807429.

[67] K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar. "Compiling and Optimizing Java 8 Programs for GPU Execution". In: *International Conference on Parallel Architecture and Compilation*. 2015, pp. 419–431. DOI: 10.1109/PACT.2015.46.

[68] Michael Wolfe. "Parallelizing Compilers". In: *ACM Comput. Surv.* 28.1 (Mar. 1996), pp. 261–262. DOI: 10.1145/234313.234417.

[69] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A Padua. "Automatic program parallelization". In: *Proceedings of the IEEE* 81.2 (1993), pp. 211–243.

[70] Bryan Chan and Tarek S Abdelrahman. "Run-time support for the automatic parallelization of Java programs". In: *The Journal of Supercomputing* 28.1 (2004), pp. 91–117.

[71] Geoffrey C Fox, Roy D Williams, and Guiseppe C Messina. *Parallel computing works!* Morgan Kaufmann, 2014.

[72] Radu Rugina and Martin Rinard. "Automatic parallelization of divide and conquer algorithms". In: *ACM SIGPLAN Notices*. Vol. 34. 8. ACM. 1999, pp. 72–83.

[73] Manish Gupta, Sayak Mukhopadhyay, and Navin Sinha. "Automatic parallelization of recursive procedures". In: *International Journal of Parallel Programming* 28.6 (2000), pp. 537–562.

[74] Mahesh Ravishankar, John Eisenlohr, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. "Automatic Parallelization of a Class of Irregular Loops for Distributed Memory Systems". In: *ACM Trans. Parallel Comput.* 1.1 (Oct. 2014), 7:1–7:37. ISSN: 2329-4949. DOI: 10.1145/2660251.

[75] Alan Leung, Ondřej Lhoták, and Ghulam Lashari. "Automatic parallelization for graphics processing units". In: *Principles and Practice of Programming in Java*. ACM. 2009, pp. 91–100.

[76] Shigeyuki Sato and Hideya Iwasaki. "Automatic parallelization via matrix multiplication". In: *ACM SIGPLAN Notices*. Vol. 46. 6. ACM. 2011, pp. 470–479.

[77] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. "The Paralax infrastructure: automatic parallelization with a helping hand". In: *International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2010, pp. 389–399.

[78] J Gregory Steffan and Todd C Mowry. "The potential for using thread-level data speculation to facilitate automatic parallelization". In: *International Symposium on High-Performance Computer Architecture*. IEEE. 1998, pp. 2–13.

[79] Williams Ludwell Harrison. "The interprocedural analysis and automatic parallelization of Scheme programs". In: *LISP and Symbolic Computation* 2.3 (Oct. 1989), pp. 179–396. ISSN: 1573-0557. DOI: 10.1007/BF01808954.

[80] Jens Nicolay, Coen de Roover, Wolfgang de Meuter, and Viviane Jonckers. "Automatic Parallelization of Side-Effecting Higher-Order Scheme Programs". In: *International Working Conference on Source Code Analysis and Manipulation*. 2011, pp. 185–194. DOI: 10.1109/SCAM.2011.13.

[81] Yoonsik Cheon, Zejing Cao, and Khandoker Rahad. *Writing JML Specifications Using Java 8 Streams*. Tech. rep. UTEP-CS-16-83. 500 West University Avenue, El Paso, Texas 79968-0518, USA: University of Texas at El Paso, Nov. 2016.

[82] Benno Stein, Lazaro Clapp, Manu Sridharan, and Bor-Yuh Evan Chang. "Safe Stream-based Programming with Refinement Types". In: *International Conference on Automated Software Engineering*. ASE 2018. Montpellier, France: ACM, 2018, pp. 565–576. ISBN: 978-1-4503-5937-5. DOI: 10.1145/3238147.3238174.

[83] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. “Clone Refactoring with Lambda Expressions”. In: *International Conference on Software Engineering*. 2017, pp. 60–70. DOI: `10.1109/ICSE.2017.14`.

[84] Raffi Khatchadourian and Hidehiko Masuhara. “Automated Refactoring of Legacy Java Software to Default Methods”. In: *International Conference on Software Engineering*. May 2017, pp. 82–93. DOI: `10.1109/ICSE.2017.16`.

[85] Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. “Refactoring Using Type Constraints”. In: *ACM Transactions on Programming Languages and Systems* 33.3 (May 2011), pp. 91–947. ISSN: 0164-0925. DOI: `10.1145/1961204.1961205`.

[86] John M. Gravley and Arun Lakhotia. “Identifying Enumeration Types Modeled with Symbolic Constants”. In: *Working Conference on Reverse Engineering*. 1996, p. 227. ISBN: 0-8186-7674-4.

[87] Raffi Khatchadourian. “Automated refactoring of legacy Java software to enumerated types”. In: *Automated Software Engineering* (Dec. 2016), pp. 1–31. DOI: `10.1007/s10515-016-0208-8`.

[88] Alon Mishne, Sharon Shoham, and Eran Yahav. “Typestate-based Semantic Code Search over Partial Programs”. In: *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2012, pp. 997–1016. DOI: `10.1145/2384616.2384689`.

[89] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. “Foundations of Typestate-Oriented Programming”. In: *ACM Trans. Program. Lang. Syst.* 36.4 (Oct. 2014), 12:1–12:44. ISSN: 0164-0925. DOI: `10.1145/2629609`.

[90] Luca Padovani. “Deadlock-Free Typestate-Oriented Programming”. In: *International Conference on the Art, Science, and Engineering of Programming*. AOSA, Apr. 2018. DOI: `10.22152/programming-journal.org/2018/2/15`.

[91] Eric Bodden. “Efficient Hybrid Typestate Analysis by Determining Continuation-equivalent States”. In: *International Conference on Software Engineering*. ACM, 2010, pp. 5–14. DOI: `10.1145/1806799.1806805`.

[92] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. *Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams*. Tech. rep. 544. 695 Park Ave, New York, NY 10065 United States: City University of New York (CUNY) Hunter College, July 2019. URL: `http://academicworks.cuny.edu/hc_pubs/544`.

[93] Martin Fowler. *Domain-Specific Languages*. English. 1st ed. Addison-Wesley Signature Series (Fowler). Addison-Wesley, Oct. 3, 2010. ISBN: 9780131392809.

[94] Francesca Arcelli Fontana, Marco Zanoni, and Francesco Zanoni. "A Duplicated Code Refactoring Advisor". In: *Agile Processes in Software Engineering and Extreme Programming*. Ed. by Casper Lassenius, Torgeir Dingsøyr, and Maria Paasivaara. Cham: Springer International Publishing, 2015, pp. 3–14. ISBN: 978-3-319-18612-2.