Scala Benchmark: Invariant Verifier

Optional Semester Project

Pamela Delgado
*****
Supervisor: Aleksandar Prokopec
Professor: Martin Odersky
Programming Methods Laboratory LAMP, EPFL
Lausanne, Switzerland

January 14, 2012

1 Introduction

During the last years parallel and concurrent programming have become subjects of an increasing importance in Computer Science. On one side new technologies tend to enhance efficiency in hardware, for example by increasing the number of parallel microprocessors, on the other side the research community puts great efforts in processing data as efficiently as possible.

Alongside with this dominant paradigm, programming languages started to be designed to take advantage of these new hardware building blocks. One example is Java and its virtual machine, which provides programmers with a thread managing system underneath. Hence, it is possible to write multi-threaded and parallelized code using a high-level language that runs on the Java virtual machine, such as Java itself, or the emergent multi-paradigm Scala language.

However, parallel and concurrent programming can be difficult to test, verify and debug, since there are simultaneous and possibly interweaving executions that require to be tracked and checked. As a simple example, if a portion of a program splits the work in a number of jobs, it might be desirable that those jobs converge after each one has been processed. These kind of checks and verifications exceed the capabilities of unit tests or other standard utilities like line by line debugging, forcing programmers to do them manually, for example by printing the needed information in logs. This task is error prone, time consuming and ineffective in general.

In this project, we propose an alternative for verifying rules defined by the programmer, that should hold for a parallel program in the Scala language. We

built our approach based on the concepts of code instrumentation and the representation of rules as invariants that can be represented in an abstract form. This project has been developed under the umbrella of the Scala Benchmark suite.

We describe the main contributions of this work as follows:

i) Definition of rules as *invariant predicates* that describe a desired property of a Scala code execution.

ii) Implementation of an *instrumenter* that injects verification code to the compiled Scala classes, allowing a non-intrusive instrumentation of the code. This injected code allows later verification of the rules defined before.

iii) Implementation of a *rule verifier* that runs the instrumented classes and verifies if the resulting execution complies to the defined rules.

iv) Experimentation of the approach with common use cases.

The rest of the report is organized as follows: In Section 2 we explain the basic notions of Java and Scala code execution and instrumentation, and we present the Scala Benchmark project. In Section 3 we describe our solution, its architecture and implementation. Section 4 is dedicated to the sample use-cases for our approach, and Section 5 to the future work, before concluding in Section 6.

2 Preliminaries

In this section we introduce concepts that are essential in the context of this project, namely: how the Java Virtual Machine internally works, what are Java agents and instrumentation, and their implication with Scala programs. We also introduce the Scala Benchmark suite.

2.1 The Java Virtual Machine

The JVM is a code execution component capable of running Java bytecode. This can be generated from Java or even other programming languages. One of the main characteristics of Java is that the code can be either interpreted or compiled Just-In-Time, in fact the Java Virtual Machine contains a JIT code generator and an interpreter, as shown in Figure 1 (as JIT has proven to perform significantly better). Other features of running code with the JVM are the sandbox safety, in addition to the Bytecode verifier, and the portability to different platforms, as the JRE adapts to different Operating Systems (represented as Native OS + processor in Figure 1).

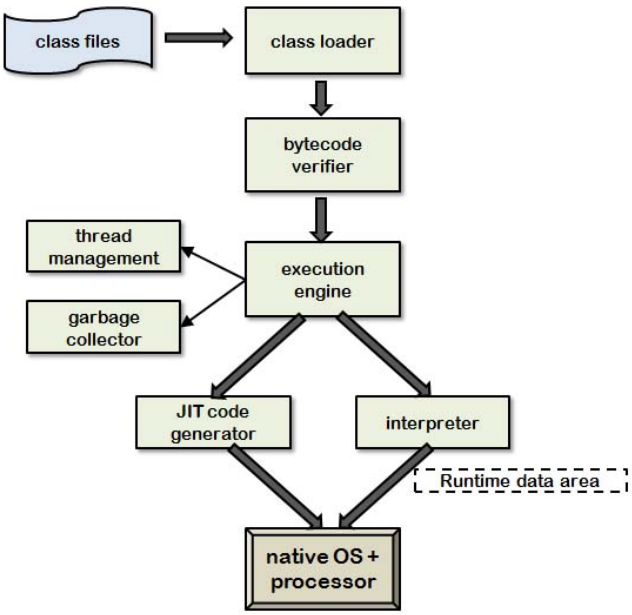

Figure 1: Java Virtual Machine overview

In Figure 2 we can see the internal architecture of the JVM, which is stack based. Once the program in class files is loaded the Java virtual machine needs to organize the memory by having several runtime data areas. The method area and the heap are shared between all threads, each new thread has its own PC register which is basically a program counter and a Java stack, which stores the state of a non native method invocations (local variables, parameters when it was invoked, return value, intermediate calculations). As for the Native method stack, the state of its native methods is implementation-dependent [6].

Although at first the JVM was meant to run Java compiled code, nowadays there are some languages which run on top of it, in order to take its advantages. Among the languages designed expressly for the JVM we find Scala, which combining object-oriented and functional paradigms is completely interoperable with Java. In the remainder of this report we will focus on Scala programs as the verification target.

2.2 Instrumentation

The fact that compiled Scala programs are Java classes allows us to perform instrumentation as if they were Java programs. For the JDK 5 a new option `-javaagent` was introduced along with the `java.lang.instrument` package. The agent specified with this parameter can be loaded statically at startup or dynamically at runtime to attach to a running process. It is possible to instru-

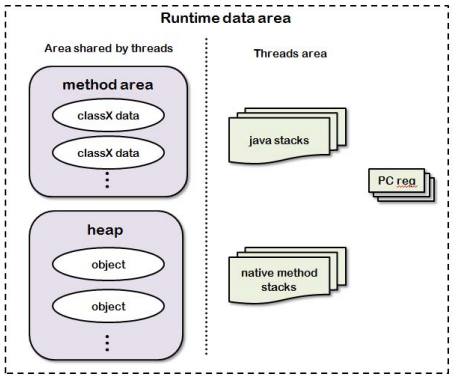

Figure 2: Java Virtual Machine internal architecture

ment the classes by modifying the bytecode of the method bodies.

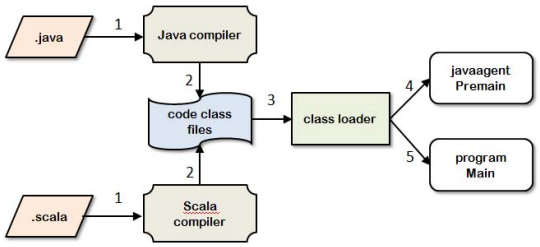

Figure 3: Instrumenting Java bytecode with javaagent.

To create a "javaagent" we need first to implement an instrumenter. Each instrumenter has to implement the `Premain` method, which as its name indicates is invoked prior to the main method of the instrumented program (See Figure 3). When used the Java agent will be statically loaded at startup. Once we have the code of the instrumenter compiled we can create a .jar file including its classes and a Manifest indicating the agent `Premain` method. In this Manifest file we can also specify if the classes can be redefined.

2.3 Scala Benchmark

As the Scala developers community starts to add more and more libraries and functionalities benchmarking them has become cumbersome to be done manually. This is why the Scala Benchmark suite was developed to assess how changes to the Scala compiler and libraries affect the space and time usage of various operations, programming patterns and idioms.

It is important to test Scala in both performance and correctness, the Scala benchmark suit follows the statistical approach mentioned in [3] by warming the JVM and includes the options Startup performance and Steady-state performance along with some profiling tasks.

3 Architecture & Implementation

In this section we describe the architecture and design of the Invariant Verifier. Its components and the way they interact with each other can be visualized in Figure 4 .

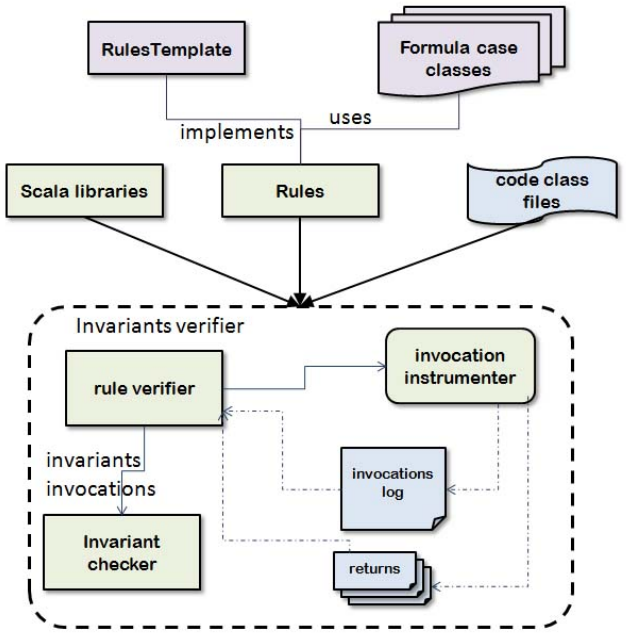

Figure 4: Invariant Verifier Architecture.

The user is provided with a Rules Template trait and the Formula case classes, he then implements the Rules Template trait by defining the list of full name methods to be instrumented and a list of Invariants, expressed as Formula, to be checked.

The path to the compiled version of this user implementation of Rules Template, we call it Rules, together with the directory of compiled user code to be tested and the Scala libraries paths are given as argument to the Rule Verifier. This component invokes Invocation Instrumenter giving it the list of methods

to be instrumented and the path to the users code.

The Invocation Instrumenter instruments and runs the user code logging information about invocations to the specified methods, if there is any return value this is also serialized and saved in a file. Finally, once the running is done, the Rule Verifier processes the log and checks the compliance of each Invariant defined by the user.

Each of the above mentioned components needed to verify user's defined invariants are described in detail in the following subsections.

3.1 Rules Template

As mentioned before, the developer needs to implement the Rules Template trait by defining a list of methods to be instrumented plus a list of invariants that must be checked on the list of invocations after running the program. The Rules Template not only clarifies and gives an order to the way rules are expressed but also is needed for the Rules Verifier to get this information using reflection.

Rules Template trait dictates that the user must specify:

- *Methods*: the list of methods as a list of String, each one of them should contain the full path to it, for example: `my.package.MyClass.mymethod`.
- *Main*: the starting point for the JVM to run the program.
- *Invariants*: this is the most important input, specified as a list of Formula, gives the user a freedom to decide how to -efficiently- express the rules.

In Scala code:

```
package ch.epfl.lamp.scalainstrument
import ch.epfl.lamp.scalainstrument.formula.Formula

trait RulesTemplate {
    def methods : List[String]
    def mainClass : String
    def invariants : List[Formula]
}
```

The following section describes Formula, the way it was designed and how to define a rule in terms of it.

3.2 Formula

Formula is an abstract concept for the way Invariants can be expressed. Its design was inspired in First Order Logic which is very powerful to express conditions and rules. Similarly to one of Scala main attractiveness, we wanted for

this DSL to be as expressive as possible from the point of view of the developer.

We started with an example of Invariant we found in Scala parallel collections. `ParIterableLike` in the standard library has a lot of collection methods inside, most of these are implemented in terms of a Task being launched, for example the filter:

```
 def filter(pred: T => Boolean): Repr = {
   executeAndWaitResult(new Filter(pred, cbfactory, splitter)
       mapResult { _.result })
 }

protected[this] class Filter[U >: T, This >: Repr](pred: T =>
    Boolean, cbf: () => Combiner[U, This], protected[this] val
    pit: IterableSplitter[T])
   extends Transformer[Combiner[U, This], Filter[U, This]] {
   @volatile var result: Combiner[U, This] = null

   def leaf(prev: Option[Combiner[U, This]]) = {
     result = pit.filter2combiner(pred, reuse(prev, cbf()))
   }

   protected[this] def newSubtask(p: IterableSplitter[T]) = new
        Filter(pred, cbf, p)

   override def merge(that: Filter[U, This]) = result = result
       combine that.result
 }
```

This Task object describes how parts of the collection are split into subsets ("split" method `newSubtask`), and then processed (through the `leaf` method). The results are then combined in the `merge` method. The idea behind the splitting of tasks is to achieve better load balancing and once the threshold is reached, "leaf" will be called to combine elements [5].

In this scenario, the following invariant should hold at all times: if there are any splits at all, then a "split" call should precede a "leaf". Expressed in another way: for all split calls there should exist a leaf such that this leaf is called after that split. If we express this rule in a First-Order logic way we have:

$$\forall s \text{ in "split"} \Rightarrow \exists p \text{ in "leaf"} \wedge s \text{ isbefore } p$$

With this example we can see some elements that can be matched with propositional and first order logic ($\forall$, $\Rightarrow$, $\exists$, $\wedge$) and some other specifically related to invocations like "in", which refers to invocation of a method, and "is before" which means that the invocation time of s is before the invocation time of p. To implement Formula we taken into account these and what we consider the most basic and useful cases to express Invariants. The following table lists

the operators implemented in Formula, each one of these correspond to a case class.

Type	First order logic	Formula equivalent
Constant	T	Constant(true)
Constant	F	Constant(false)
Logical	$\neg$	Neg
Logical	$\vee$	||
Logical	$\wedge$	&&
Logical	$\Rightarrow$	==>
Logical	$\forall$	Forall
Logical	$\exists$	Exists
Invocation specific		in
Invocation specific		Filter(true)
Invocation specific		Filter(false)
Invocation specific		isBefore

Table 1: Formula case classes and first order logic equivalence

Using this classes we can write our example of Invariant as:

```
def splitInvariant Forall(x => (x in "split")
    ==> Exists(y => (y in "leaf") && isBefore(x, y)))
```

It is important to notice that when using Formula, the user has to be wise in writing correct Invariants and take into account the effectiveness specially when dealing with a big list of invocations. Nevertheless, some optimizations were possible in the implementation of the project, they are described in the following subsection.

3.2.1 Optimizations

When implementing the Formula checking of Invocations inside Invariant checker, some optimizations were possible at the logic level. In the cases of $\vee$, $\wedge$ and $\Rightarrow$ we looked at their truth logic tables.

p	q	$p \wedge q$	p	q	$p \vee q$	p	q	$p \Rightarrow q$
T	T	T	T	T	T	T	T	T
T	F	F	T	F	T	T	F	F
F	T	F	F	T	T	F	T	T
F	F	F	F	F	F	F	F	T

Table 2: Truth tables for $\vee$, $\wedge$ and $\Rightarrow$

From this table, for the logical conjunction we know that if one predicate, for example p, is `false` the other, for example q, is irrelevant (there is no need

to evaluate it) and if one of the predicates is `true` the other one needs to be evaluated. Similarly, in the case of disjunction if one of the predicates is `true` the other is not evaluated, while if one of them is False the result is the evaluation of the other. Implication truth table is not symmetric in the sense that p and q are treated differently, the optimization here is to skip evaluation of q in the case the first predicate p is false (returning True as result).

Also the quantifiers have their optimizations, which is important for scalability given that they are evaluated for each invocation. In the case of $\forall$ we know that as soon as we have a `false` result for a given Invocation, further evaluation is not needed. In the case of $\exists$, as soon as we have a `true` result for an Invocation, the cycle stops returning `true`.

The case of `in` was a little different, because it has to be considered in $\forall$ as a filter this is why a case class Filter is returned. If the predicate `in` applied to an invocation returns `false`, the result `Filter(false)` will prevent from evaluating the rest of the implication and from returning a `false` value. However when `in` is combined with $\exists$ its `Filter` result acts like a normal `Constant`.

3.3 Invocation Instrumenter

As mentioned in Section 2, the fact that Scala compiled classes are Java bytecode let us use Java instrumentation tools and run user's instrumented code in the JVM. In Java there are basically three instrumentation approaches: instrument source files, use Java Debug Interface and instrument class files. As we want to perform a non-intrusive invariant verification we will not choose to instrument the source files even though this is the easiest way of instrumenting code. As for the JDI, this is already used in the Scala Benchmark suite and it has proved to be slow. In addition, we only want to instrument to obtain basically the order in which certain invocations have been done and probably the pre and post conditions of it. This is why we decided to instrument the source files.

Although the `java.lang.instrument` package allows access to redefine each loaded class, building instrumentation in bytecodes is cumbersome. In response, some editing tools like BCEL, ASM, SERP, Tool have been developed. For this project the tool ASM was used which comes with a bytecode viewer plugin for Eclipse, but still it was difficult or long to debug things in the instrumenter.

Following an analysis of the requirements, we determined how instrumentation is performed. Since we want to make the instrumentation as efficient as possible and introduce the lowest overhead, only methods involved in the invariants were instrumented. We inserted code using an ASM `ClassVisitorAdapter` and a `MethodAdapter`, for each invocation of a method included in user's list new bytecode is inserted before and after the execution of the body.

The bytecode inserted at the beginning calls to the Logger registering the timestamp and the class and method names (method `logInvocation`). The following code excerpt shows how we add the corresponding bytecode.

```
mv.visitMethodInsn(org.objectweb.asm.Opcodes.INVOKESTATIC,
      "ch/epfl/lamp/scalainstrument/instrumenter/Logger",
      "getInstance",
      "()Lch/epfl/lamp/scalainstrument/instrumenter/Logger;");
mv.visitLdcInsn(idTimestamp + className + methodName /*+...*/);
mv.visitMethodInsn(org.objectweb.asm.Opcodes.INVOKEVIRTUAL,
      "ch/epfl/lamp/scalainstrument/instrumenter/Logger",
      "logInvocation",
      "(Ljava/lang/String;)V");
```

While the bytecode inserted at the end sends to the Logger the new timestamp, the return object and its ID which we decided to keep as the initial timestamp (when the method was just called). The following code excerpt add the bytecode that will do that by inserting a call to the `logReturnValue` method.

```
mv.visitMethodInsn(org.objectweb.asm.Opcodes.INVOKESTATIC,
       "ch/epfl/lamp/scalainstrument/instrumenter/Logger",
       "getInstance",
       "()Lch/epfl/lamp/scalainstrument/instrumenter/Logger;");
mv.visitVarInsn(postponedOpcode, postponedVar);
mv.visitLdcInsn(endTimestamp + idTimestamp /*+ ...*/);
mv.visitMethodInsn(org.objectweb.asm.Opcodes.INVOKEVIRTUAL,
       "ch/epfl/lamp/scalainstrument/instrumenter/Logger",
       "logReturnValue",
       "(Ljava/lang/Object;Ljava/lang/String;)V");
```

The Logger itself is implemented in Java as part of the Invocation Instrumenter, it uses the Singleton pattern to ensure a unique access to the log file in case of multiple threads and it simplifies the amount of bytecode to be inserted.

An example of this instrumentation can be seen below. Suppose we want to instrument a simple method `makeTriangle` that returns an object of type `Triangle`.

```
def makeTriangle(length: Int):Triangle ={
  val triangle = new Triangle(length,0,0)
  triangle
}
```

Its corresponding byte code would be:

```
// Method descriptor #28 (I)LTriangle;
// Stack: 5, Locals: 3
public Triangle makeTriangle(int length);
0  new Triangle [41]
3  dup
```

```
4  iload_1 [length]
5  iconst_0
6  iconst_0
7  invokespecial Triangle(int, int, int) [55]
10  astore_2 [triangle]
11  aload_2
12  areturn
```

And after instrumenting the resulting bytecode of the example would be:

```
// Method descriptor #29 (I)LTriangleScala;
// Stack: 5, Locals: 3
public Triangle makeTriangle(int length);
   0  invokestatic ch.epfl.lamp.scalainstrument.instrumenter.
       Logger.getInstance() : ch.epfl.lamp.scalainstrument.
       instrumenter.Logger [58]
   3  ldc <String "1326405832880 Invoking TriangleHelloScala\
       $ makeTriangle"> [60]
   5  invokevirtual ch.epfl.lamp.scalainstrument.instrumenter
       .Logger.logInvocation(java.lang.String) : void [64]
   8  new Triangle [42]
  11  dup
  12  iload_1 [length]
  13  iconst_0
  14  iconst_0
  15  invokespecial Triangle(int, int, int) [67]
  18  astore_2
  19  invokestatic ch.epfl.lamp.scalainstrument.instrumenter.
      Logger.getInstance() : ch.epfl.lamp.scalainstrument.
      instrumenter.Logger [58]
  22  aload_2
  23  ldc <String "1326405832911 Returning 1326405832880">
      [69]
  25  invokevirtual ch.epfl.lamp.scalainstrument.instrumenter
      .Logger.logReturnValue(java.lang.Object, java.lang.
      String) : void [73]
  28  aload_2
  29  areturn
```

As reflected in the bytecode, a more complex transformation was needed to get the returning value as an object. This statefull transformation of the method, as stated in [2], memorizes the state of loading an object to the stack and `postpone` it for later. In the case of `areturn` then we can insert the bytecode as shown before, and in the case of other instruction the postponed load is executed accordingly.

During the development of Invocation Instrumenter also references [1] and [4] were helpful.

3.4 Rule Verifier

The Rule Verifier is the main orchestrator of the project, the user Rules class, the application compiled code and needed libraries directories are its input. The Rules class is first reflected to get the list of full name methods to be instrumented and the path to the main (starting point of the application) method. With this data and the code and classpath directories the Rule Verifier then invokes the Invocation Instrumenter which is written in Java.

The following is an example of how the Rule Verifier would call the instrumenter, once the list of methods has been reflected from Rules class. In this example the starting point is `TriangleHelloScala`.

```
>java -cp libs\asm-3.3.1.jar;libs\asm-tree-3.3.1.jar;
    testCases;libs\scala-library.jar;libs\scala-compiler.jar
    -javaagent:libs\InvocationInstrumenter.jar=
    methods=TriangleHelloScala.makeTriangle;
    AnotherClass.itsmethod,logpath=logPath TriangleHelloScala
```

Once the Instrumenter has finished, a log file with the list of invocations is produced and the return values (if any) of the instrumented methods are serialized and saved in files. The Rule Verifier parses the log into a list of Invocations storing the starting and ending timestamps, method and class full name and the ID name of the return serialized object file.

The final step is to verify the rules, again by reflecting Rules class the list of invariants is obtained. Each invariant is then checked with the full list of invocations.

4 Cases of study

Adding some parenthesis for clarification and a way of specifying the name of the class for the `in` operator, the rule explained in the previous section about a `split` being invoked before a `leaf` is fully expressed as:

```
def splitLeafInvariant =
    Forall(x => (x in (splitLeafClass,"split"))
    ==> Exists(y => (y in (splitLeafClass,"leaf")) &&
        isBefore(x, y)))
```

Apart from the example with splits and leafs, we found other test cases in collections. These cases of study and its corresponding Invariants are explained below.

Case Splitters. Once the Splitter is splitted it becomes invalid in the sense that after calling `split` no method can be called at all. Or formulated in another way, for all split calls it does not exist any other call to any method of that class:

```
val splitterClass = "scala.collection.parallel.Splitter"
def splitterSplitInvariant = Forall(x => (x in (splitterClass
    ,"split"))
    ==> Neg (Exists(y => (y in splitterClass) && isBefore(x,
        y))))
```

Case Builders result. Builders in collections there is an invariant, `result` method can be called only once.

```
val builderClass = "scala.collection.mutable.Builder"
def builderInvariantResult = Forall(x => (x in (builderClass,
    "result"))
    ==> Neg (Exists(y => (y in (builderClass,"result")) &&
    isBefore(y,x))))
```

Case Builders plus. Another invariant found in Builders is that there should not be a call to `+=` after calling `result` unless `clear` has been called first (between result and +=).

```
def builderInvariantPlus =  Forall(x => (x in (builderClass,"
    result"))
    ==> (Neg(Exists(y => (y in (builderClass,"+=")) &&
        isBefore(x,y))))
    || (Exists(y => (y in (builderClass,"+=")) && isBefore(x,
        y)
    && Exists(z => (z in (builderClass,"clear")) && isBefore(
        z,x)))))
```

Case Combiners result. This case is similar to Builder result invariant.

```
val combinerClass = "scala.collection.parallel.Combiner"
def combinerInvariantResult = Forall(x => (x in (
    combinerClass,"result"))
    ==> Neg (Exists(y => (y in (combinerClass,"result")) &&
        isBefore(y,x))))
```

Case Combiners plus. This case is similar to Builder plus invariant.

```
def combinerInvariantPlus =  Forall(x => (x in (combinerClass
    ,"result"))
    ==> (Forall(y => ((y in (combinerClass,"+=")) && isBefore
        (x,y))
    ==> Exists(z => (z in (combinerClass,"clear")) &&
        isBefore(z,x)))))
```

Case Iterators. This is a simple case, once `hastNext`'s result is false then `next` should not be called. In this case is useful to have the result to deserialize it.

```
val iteratorClass = "scala.collection.Iterator"
def iteratorInvariant = Forall(x => (x in (iteratorClass,"
    hasNext")) && Result[Boolean](false) ==> Neg (Exists(y =>
    (y in "next") && isBefore(x,y))))
```

Case Threads. When using threads in parallel collections calls to `update` can be specified only after or before a `filter` or `map` (not in between).

5 Future Work

Even though the information of the instrumented methods and the implementation of basic cases of Formula cover the invariants exposed before 4 that were found in parallel collections, there are some functionalities that can be added in a future work.

- In the instrumenter side we could add more information about invocations, for example save the values of the arguments passed to methods, a way of identifying each class instance and which class has invoked the method. An invariant that could use this last information is: if a stream is created from array buffer, then there should not be mutations to the array.
- In Formula it would be useful to create a transitive closure, represented as `*` for recursive calls, folds, etc. For example in `Stream` there is an invariant: since streams are not constructed eagerly, we do not want to mix lazy evaluations between mutable or immutable objects. However, when we see at the implementation of the `toStream` method we see that the way is constructed is a recursive call.
- Also in Formula some way of controlling rules correctness can be added, and some other Formula case classes together with their precedence.
- A formal proof of Formula's completeness and soundness and its PSPACE completeness with respect to satisfiability as a derivation of second order Booleans.
- Search and define test cases invariants for `Actors`.

6 Conclusion

As a conclusion we can say that this project has allowed us to succeed in three main contributions: the use of instrumentation to non-intrusively and effectively inject code in methods invocations, the introduction of `Formula` and a semi-formal way of describing invocation invariants, and finally the implementation of an Invariant checker including its optimizations.

Working in this project has been very rewarding in the sense of learning, from how to instrument code and deal with bytecodes to face some challenges and learn guidelines when designing a small Domain Specific Language.

References

[1] J. Aarniala. Instrumenting Java bytecode. In *Seminar work for the Compilerscourse, Department of Computer Science, University of Helsinki, Finland*, 2005.

[2] E. Bruneton. ASM 3.0 a Java bytecode engineering library. Technical report, 2007.

[3] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007.

[4] E. Kuleshov. Using ASM framework to implement common bytecode transformation patterns. *Proc. of the 6th AOSD, ACM Press*, 2007.

[5] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A generic parallel collection framework. *Euro-Par 2011 Parallel Processing*, pages 136–147, 2011.

[6] B. Venners. *Inside the Java virtual machine*. McGraw-Hill, Inc., 1996.