A Coroutine Extension to Java

Adam Ohlsson
D13, Lund University, Sweden
zba10aoh@student.lu.se

Erik Leffler
Pi16, Lund University, Sweden
er1110le-s@student.lu.se

Abstract

This paper present an extension to the ExtendJ Java compiler that enables the use of asymmetric, first-class, stackless coroutines. The extension perform syntactic sugaring and generate standard Java code, acting as a translator. The extension currently provide support for a fairly large subset of Java but does not provide error checking or error messages.

1 Introduction

The subject of coroutines was first introduced by Melvin Conway in 1963 [1]. A Coroutine can be described as a generalisation of a subroutine. The key separating factor is that a coroutine can be paused and reentered several times during its life time. Each time, execution picks up from where it previously left off. More specifically, inside a coroutine one may specify a control statement, `detach`, that returns control to the caller, similar to a `return` statement for a subroutine. A coroutine will always resume execution directly after the previous call to detach in subsequent activations.

The lifetime of a routine could be thought of as the time during which its state can be referenced in a meaningful way. In a language using a stack based model of computation where arguments are passed on a stack, the lifetime of a routine would then be equal to that of its corresponding stack frame. Out of neccessity, a coroutine therefore require its own stack (or equivalent construct) in order for it to be able to operate with the required flexibility.

Indeed, this is the approach used by Stadler et al. providing native support for coroutines in the Java HotSpot™ VM [7]. Though there clearly are benefits in changing the underlying machinery to support coroutines, as reported, this approach has well known drawbacks in terms of portability and thus also support.

It is possible to simulate the behaviour of coroutines using standard Java but the complexity of the resulting code does not scale very well. The main challenge for the developer is to keep track of different code paths as well as managing resources available to the routine which have to be exposed to the class in order to avoid destruction on detach. The resulting code is difficult to manage and unsafe since any method in the same class may manipulate the coroutine state.

The goal in this report is to present an alternative method by which coroutines can be supported in Java, taking a defensive approach by extending the *ExtendJ*[3] compiler using JastAdd, a meta-compilation system[5]. The extension of Java to support coroutines result in a language that is similar to Java which we will call Javasim.

The extension supports the following:

- Relocation of coroutine local state.
- Inlining of method calls that utilize the coroutine API.
- Translation from Javasim into equivalent standard Java code.

Introducing coroutines using this method has several advantages. Standard Java code is portable, and thus also the code generated by the extension. The extension can also be integrated with other extensions developed using the same toolset.

Coroutines are introduced in the Java language with little or no impact on existing code and runtime environment while enabling the corresponding features to the developer.

A limitiation of our implementation is that coroutines are stackless and does therefore not allow the user to define recursive coroutines.

2 Background

2.1 Coroutines

When the coroutine concept first was introduced by Conway he described it as "an autonomous program which communicates with adjacent modules as if they were input or output subroutines"[1]. A more concrete definition was encapsulated by Merlin [6] as follows:

- "the values of data local to a coroutine persist between successive calls;"
- “the execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage.”

More specific ways of classifying coroutines has since been proposed by Moura et al. [2] where the three distinguishing properties were stated as:

- "the control-transfer mechanism, which can provide symmetric or asymmetric coroutines;"
- "whether coroutines are provided in the language as first-class objects, which can be freely manipulated by the programmer, or as constrained constructs;"
- "whether a coroutine is a stackful construct, i.e., whether it is able to suspend its execution from within nested calls."

The concept of asymmetric coroutines entail that control is returned to the caller of a coroutine by use of a *detach* statement. From within a symmetric coroutine however, control

Course paper, EDAN70, Lund University, Sweden
January 15, 2018.

is passed by invoking yet another coroutine and hence, does not necessarily need to return to the caller. The Java extension presented in this paper enable asymmetric, first-class, stackless coroutines.

In this implementation, a coroutine is represented by a class that implements an interface called `Coroutine`. The `Coroutine` interface contains two methods, one is a default method called `detach` and the other `doActivate`. The `doActivate` method is meant to be overridden by the user to define the behaviour of a coroutine and is also the entry point. The `detach` method is used to return control directly to the process that activated the coroutine through its `doActivate` method, a mechanism that is required for asymmetric coroutines.

When the `doActivate` method in the example shown in listing 1 is called, the first print statement is executed after which execution is returned to the caller. Following a second call to `doActivate`, execution is resumed from where the method previously detached and the second print statement is executed.

```
public class Example implements Coroutine {

  public Example () {
  }

  public void doActivate() {
    System.out.println("First");
    detach();
    System.out.println("Second");
  }

}
```

Listing 1. A simple coroutine example.

2.2 JastAdd

JastAdd is a Java based compiler construction framework [5]. JastAdd is used for specifiying the structure of the Abstract Syntax Tree (AST). JastAdd enable both declarative techniques using Reference Attribute Grammars and imperative techniques for modifying and performing computations upon the AST. The AST is represented by a class hierarchy where every node is an instance of a specific class. For a complete and comprehensive introduction to JastAdd we refer you to the official documentation.

2.3 ExtendJ

ExtendJ is a JastAdd based Java compiler [3]. The underlying frameworks of the compiler allow for extensions and modifications to be realized through the addition of modules. ExtendJ currently support Java versions 4 through 8 and is being developed and maintained at Lund University. We refer you the the official ExtendJ website for more information.

3 Motivating example

In this section we will demonstrate the usefulness of coroutines by showing an example of a hospital simulation. The simulation is meant to model the work flow of a simplified hospital. The simulation will consist of the following objects and events:

- A Hospital, which will generate patients according to some temporal distribution;
- Patients, that arrive at the hospital, whilst waiting for available doctors, are stationed in a waiting room;
- Doctors that, after having finished treating a patient, will either treat the next patient in line for a random amount of time or, if the wait room is empty, have a break in the coffee room.

The `Doctor` class will be implemented as a coroutine. This allow the doctors to, either whilst waiting in the coffe room or treating a patient, return control back to the simulation environment. In listing 2 we demonstrate what such a class might look like.

```
public class Doctor implements Coroutine {
  private Hospital h;
  private Patient p;

  public Doctor(Hospital h) {
    this.h = h;
  }

  private boolean waitRoomIsEmpty() {
    return h.waitRoomEmpty();
  }

  private void waitInCoffeeRoom() {
    h.addToCoffeeRoom(this);
    detach();
  }

  private void treatNextPatient() {
    Patient p = h.nextPatient()
    System.out.println("Treating Patient: " + p);
    h.makeUnavailable(this, p.treatmentTime());
    detach();
    System.out.println("Treatment done: " + p);
  }

  public void doActivate() {
      while(true) {
      if (waitRoomIsEmpty()) {
        waitInCoffeeRoom();
      } else {
        treatNextPatient();
      }
    }
  }
}
```

Listing 2. The Doctor class

Inside of the doctor coroutine body, `doActivate`, there are calls to two *detaching* helper methods, `waitInCoffeRoom` and `treatNextPatient`. The `waitInCoffeRoom` method is meant to simply add the doctor to the coffee room (implemented as a list) and then return control back to the caller. The `treatNextPatient` method retrieves the next patient in the waiting room, specifies a time for which the doctor is unavailable due to treating the patient, and then returns control back to the caller

The use of coroutines in this example allow the doctor class to easily pass control back to the simulation environment whilst still maintaining all of its runtime state.

4 Implementation

This section describes our implementation of the coroutine extension to ExtendJ.

Before going into the implementation details, lets review the goal of this extension. The goal is to read source files containing Javasim code and produce the equivalent standard Java source code as output. Hence, the compiler extension presented in this article simply provide syntactic sugar on top of the Java language.

An important design choice made early-on is that the Javasim language is syntactically equivalent to standard Java. If the compiler does not recognize extension specific triggers in the input file, the resulting output will be equal to the input.

This implementation effectively provide asymmetric coroutines as first-class objects but not stackful. A single method, `doActivate` is used as the global (re-)entry point for a coroutine. Method calls in the body of `doActivate` that at some point leads to a call to `detach` are inlined. This effectively vanquish the requirement for a stack but at the same time prohibit the interpretation of recursive coroutines.

There are two aspects (not entirely independent) of a coroutine implementation which requires special attention. The first is data and state management and the second is control flow [2].

The local state of a coroutine has to be stored in such a way that it does not fall out of scope as the coroutine detaches. Since each coroutine is meant to be implemented in a separate class, the straightforward choice is to move all local state defined in detaching methods into its body as protected field declarations. State includes local variable declarations as well as method parameters. Constructors may not detach and are therefore always copied as they are defined by the user.

Now then, since recursion or any form of circular activation scheme is prohibited, uses of the parameters and local variables of a detaching methods never nest. It is therefore safe to create a single set of parameters and local variables as field declarations for each detaching method in each coroutine class.

In order to be able to invoke an overridden version of a detaching method, the field declarations representing the parameters must be visible in derived classes. Since, such methods must also be able to reference local state when inlined, local state and parameters should be declared with protected access when transferred to the coroutine class as field declarations.

We will now consider the method by which the coroutine body, the `doActivate` method, is translated from Javasim into standard Java and how control flows within a coroutine. Note that Javasim has the same semantics as standard Java, but also contains the control statement `detach` which is realized as a method invokation to the default method `detach` in the Coroutine interface.

A coroutine body is translated to a switch-statement wrapped in a while(true)- statement in standard Java. The switch-statement has a state variable as expression and can therefore be used to access a specific case. The while statement makes sure that several cases can be accessed during each activation. At the end of each case, the state variable is updated to specify the next block to be executed, followed by either a break or a return statement to continue to the next case or to detach, respectively.

To generate a sequence of switch-cases from a coroutine body, all control constructs, control statements, and detaching method accesses each must be dissassembled into one or more blocks which can be collected and concatenated to form a single list of blocks for the whole body. A detaching method call is inlined by replacing it by the corresponding sequence of blocks that makes up the associated method body. Since control statements cause control to be transferred to new switch-cases, these must be treated with just as much care as calls to `detach`.

In order to convert Javasim statements into blocks, we introduce the abstract `CoStmt` which is used to represent blocks of Javasim statements. From `CoStmt`, separate classes for control structures and control statements are derived each of which can represent the corresponding Javasim statement as a sequence of CoStmts. All Javasim statements that do not require any special care with regards to control flow is represented by a derived class called `CoInst` which can represent a sequence of Javasim statements. Thus, the whole body of `doActivate` can be represented by a sequence of CoStmts.

For example, the following Javasim coroutine body:

```
void doActivate() {
    if (isTrue()) {
        detach();
        System.out.println("Hello, World!");
    }
    System.out.println("Finished!");
}
```

is converted into a `CoIf` followed by a `CoInst`. `CoIf` contains a `CoStmt` representing the condition followed by a sequence of CoStmts representing the body of the statement. The following standard Java code is generated:

```
void doActivate() {
    while (true) {
        switch (statevar) {
        case 0: {
            statevar = (isTrue()) ? 1 : 3;
            break;
        }
        case 1: {
            statevar = 2;
            return; // detach
        }
        case 2: {
            System.out.println("Hello, World!");
            statevar = 0;
        }
        case 3: {
                System.out.println("Finished!");
                statevar = 0;
                break;
        }}
    }
}
```

A desugaring procedure is also applied in the conversion to CoStmts. Desugaring makes sure that local variable declarations are converted to variable assignments of the corresponding field declarations and that uses of those symbols are substituted accordingly. Also, detaching method calls are transformed into assignments of corresponding parameter field declarations followed by a `CoStmt` representing the body of the inlined method.

Control statements all get their own derived `CoStmt`-class and are transformed directly into instances of these. These are `CoBreak`, `CoContinue`, `CoReturn`. Calls to `detach` are transformed into `CoDetach`.

At the time of writing, `while`, `for`, and `if` statements (including `else if`- and `else`-variations) are considered by the implementation along-side the control statements mentioned above. If-statements are converted into `CoIf`, for-statements into `CoFor`, and while-statements into `CoWhile`, respectively.

On these new node types, declared using abstract syntax, attributes are defined in the abstract syntax tree allowing us to reason in terms of `CoStmt` instead of standard Java statements. Basically, each `CoStmt` has an attribute which determine its state (its index in the resulting list of `CoStmt`) which can be used as a case label in the generated switch-statement. Combining this with another attribute which determine the next block to be executed in the sequence (preserving the semantics of standard Java extended with the detach statement) the state variable can be updated accordingly at the end of each block.

In order to link this tree of `CoStmts` with the abstract syntax tree produced by the parser, a *non-terminal attribute* (nta) is placed on the AST-class ClassDecl defined by ExtendJ. The nta evaluates to a new desugared version of type ClassDecl which is used to generate the desugared code.

Code is generated by refining the `prettyPrint` method for ClassDecl (defined in aspect `PrettyPrint` of Java4) to evaluate the nta placed on ClassDecl described above and to *prettyPrint* the desugared version of the class declaration instead.

The `prettyPrint` method is used to convert an abstract syntax tree into its corresponding source code.

5 Evaluation

5.1 Hospital simulation

A hospital simulation similar to the one demonstrated in the motivating example section was provided to us by the computer science department at Lund University at the start of this project. The simulation was written in regular Java code and has now been rewritten to utilize our compiler extension. The result is code that is more concise and easy to read. Listing 3 show the original `doActivate` method in the `Doctor` class. The same method is displayed in listing 4, rewritten to use the extension presented in this paper.

```
public void doActivate() {
  while (true) {
    switch (pCase) {
    case "initial":
      if (h.waitRoom.isEmpty()) {
        h.coffeeRoom.add(this); // wait(
            coffeeRoom)
        return;
      } else {
        currentPatient = h.waitRoom.removeFirst
            ();
        h.log("Patient " + currentPatient + "
            started treatment by " + this);
        hold(Rand.exp(1/Hospital.treatmentTime))
            ;
        pCase = "step2";
        return;
      }
    case "step2" : {// my timer, i.e. patient
        treated
      h.log("Patient " + currentPatient + "
          ended treatment by " + this);
      pCase = "initial";
      }
    }
  }
}
```

Listing 3. The `Doctor` coroutine body prior to utilizing the presented extension.

```
public void doActivate() {
  if (!h.waitRoom.isEmpty()) {
    currentPatient = h.waitRoom.removeFirst();
    h.log("Patient " + currentPatient + "
        started treatment by " + this);
    hold(Rand.exp(1/Hospital.treatmentTime));
    detach();
    h.log("Patient " + currentPatient + " ended
        treatment by " + this);
  } else {
    h.coffeeRoom.add(this);
    detach();
  }
}
```

Listing 4. The `Doctor` coroutine body after to utilizing the presented extension.

Both simulations were executed with a seeded random function and identical results were obtained. This strengthen the validity of the compiler extension that is presented.

6 Related work

Today, coroutines are available in varying forms in several modern general purpose programming and scripting languages. Alternatives compatible with Java are for example through a language called Kotlin which claims interoperability and a 100 % compatibility with existing Java-based technology stacks, or alternatively by using a modified JVM such as described by Stadler et al.[7].

7 Conclusion

A method for enabling the use of asymmetric coroutines in Java was presented. By using a uniform activation method and an inlining technique we allow the user to define stackless (non-recursive) coroutines. Whether the consequences of stacklessness are acceptable is of course arguable, though, statements have been made in the past that recursive coroutines are exceedingly rare [4] . Despite this, coroutines manifest as classes and are therefore first-class objects and can be moved around freely by the user.

There are several improvements that can be made on this implementation. For example, only a handful of Java statements have been considered and transformed into CoStmts. More can easily be implemented by deriving new classes from *CoStmt* and define a small number of attributes for them. New Constructs can be added in a methodical way with only a small overhead in terms of attributes and equations.

The current release generate a large number of states where in many places adjacent states could be concatenated into a single state. Such a state reduction algorithm is partly implemented though, due to time constraints, have not been completed.

The most pressing improvement that is required for the implementation to be taken seriously is the addition of error checking and informative error messages. At the time of writing, no such messages are given to the user and the user is assumed to know the implementation in and out.

Generic constructs have not been considered at all.

Acknowledgments

We want to thank our supervisor Görel Hedin for her support and guidance during the execution of the project that lead up to this report. We also thank Boris Magnusson for explaining the coroutine concept to us, providing us with material and discussing the desired functionality.

References

[1] Melvin E Conway. 1963. Design of a separable transition-diagram compiler. *Commun. ACM* 6, 7 (1963), 396–408. https://doi.org/10.1145/366663.366704

[2] Ana Lúcia de Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31, 2 (2009), 6.

[3] Torbjörn Ekman and Görel Heding. 2007. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. ACM, Montreal, Canada, 1–18. https://doi.org/10.1145/2500828.2500843

[4] Dick Grune. 1977. A View of Coroutines. *SIGPLAN Not.* 12, 7 (July 1977), 75–81. https://doi.org/10.1145/954639.954644

[5] Görel Hedin and Eva Magnusson. 2001. JastAdd - a Java-based system for implementing front ends. *Electronic Notes in Theoretical Computer Science* 44 (2001), 59–78. https://doi.org/10.1016/S0167-6423(02)00109-0

[6] Christopher D Marlin. 1980. *Coroutines: a programming methodology, a language design and an implementation*. Number 95. Springer Science & Business Media.

[7] Lukas Stadler, Thomas Würthinger, and Christian Wimmer. 2010. Efficient Coroutines for the Java Platform. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ '10)*. ACM, New York, NY, USA, 20–28. https://doi.org/10.1145/1852761.1852765