


Theory and Practice of Coroutines with Snapshots

Aleksandar Prokopec

Oracle Labs, Zürich, Switzerland


aleksandar.prokopec@gmail.com

 <https://orcid.org/0000-0003-0260-2729>

Fengyun Liu

École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

fengyun.liu@epfl.ch

 <https://orcid.org/0000-0001-7949-4303>

Abstract

While event-driven programming is a widespread model for asynchronous computing, its inherent control flow fragmentation makes event-driven programs notoriously difficult to understand and maintain. *Coroutines* are a general control flow construct that can eliminate control flow fragmentation. However, coroutines are still missing in many popular languages. This gap is partly caused by the difficulties of supporting suspendable computations in the language runtime.

We introduce first-class, type-safe, stackful coroutines with snapshots, which unify many variants of suspendable computing. Our design relies solely on the static metaprogramming support of the host language, without modifying the language implementation or the runtime. We also develop a formal model for type-safe, stackful and delimited coroutines, and we prove the respective safety properties. We show that the model is sufficiently general to express iterators, single-assignment variables, `async-await`, actors, event streams, backtracking, symmetric coroutines and continuations. Performance evaluations reveal that the proposed metaprogramming-based approach has a decent performance, with workload-dependent overheads of $1.03 - 2.11\times$ compared to equivalent manually written code, and improvements of up to $6\times$ compared to other approaches.

2012 ACM Subject Classification Software and its engineering → Coroutines, Software and its engineering → Control structures

Keywords and phrases coroutines, continuations, coroutine snapshots, asynchronous programming, inversion of control, event-driven programming

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.3

Related Version <https://arxiv.org/abs/1806.01405>

1 Introduction

Asynchronous programming is becoming increasingly important, with applications ranging from actor systems [1, 21], futures and network programming [10, 22], user interfaces [30], to functional stream processing [32]. Traditionally, these programming models were realized either by blocking execution threads (which can be detrimental to performance [4]), or callback-style APIs [10, 22, 26], or with monads [67]. However, these approaches often feel unnatural, and the resulting programs can be hard to understand and maintain. *Coroutines* [11] overcome the need for blocking threads, callbacks and monads by allowing parts of the execution to pause at arbitrary points, and resuming that execution later.

There are generally two approaches to implement control flow constructs like coroutines: *call stack manipulation* and *program transformation*. In the first approach, the runtime is



© Aleksandar Prokopec and Fengyun Liu;

licensed under Creative Commons License CC-BY

32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Editor: Todd Millstein; Article No. 3; pp. 3:1–3:32

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

augmented with call stack introspection or the ability to swap call stacks during the execution of the program. We are aware of several such attempts in the context of the JVM runtime [14, 62], which did not become official due to the considerable changes required by the language runtime. In the second approach, the compiler transforms the program that uses coroutines into an equivalent program without coroutines. In Scheme, a similar control flow construct `call/cc` was supported by transforming the program into the continuation-passing style (CPS) [64]. The CPS transform can also be selectively applied to delimited parts of the program [3, 16, 17, 35, 56]. Mainstream languages like Python, C#, JavaScript, Dart and Scala offer suspension primitives such as generators, enumerators and `async-await`, which often target specific domains.

Coroutines based on metaprogramming. We explore a new transformation approach for coroutines that relies on the static metaprogramming support of the host language (in our case Scala), and assumes no call stack introspection or call stack manipulation support in the runtime (in our case JVM). The metaprogramming-based solution has several benefits:

- (1) The language runtime and the compiler do not need to be modified. This puts less pressure on the language and the runtime maintainers.
- (2) Since the metaprogramming API is typically standardized, the coroutine implementation is unaffected by the changes in the runtime or in the compiler.
- (3) The implementation does not need to be replicated for each supported backend. Our own implementation works with both the JVM runtime, and the Scala.JS browser backend.
- (4) Coroutines can be encapsulated as a standalone library. Our implementation in Scala is distributed independently from the standard Scala distribution.

We note that our approach is not strictly limited to metaprogramming – it can also be implemented inside the compiler. However, to the best of our knowledge, we are the first ones to implement and evaluate coroutines using a metaprogramming API.

Summary. Our coroutine model is statically typed, stackful, and delimited. Static typing improves program safety, stackfulness allows better composition, and delimitedness allows applying coroutines to selected parts of the program (this is explained further in Section 2). Regions of the program can be selectively marked as suspendable, without modifying or recompiling existing libraries. These regions represent first-class coroutines that behave similar to first-class function values. We show that the model generalizes many existing suspendable and asynchronous programming models. We extend this model with snapshots, and show that the extension allows expressing backtracking and continuations. Finally, we show that the metaprogramming approach has a reasonable performance (at most $2.11\times$ overheads compared to equivalent manually optimized code) by comparing it against alternative frameworks. We also formalize coroutines with snapshots and prove standard safety properties.

Contributions. The main novelties in this work are as follows:

- We show that *AST-level static metaprogramming support of the host language* is sufficient to implement first-class, typed, stackful, delimited coroutines that are reasonably efficient.
- We propose a new form of coroutines, namely *coroutine with snapshots*, which increases the power of coroutines. For example, it allows emulating backtracking and continuations.
- We formalize stackful coroutines with snapshots in λ_{\sim} by extending the simply typed lambda calculus, and we prove soundness of the calculus.

Some of the features of the proposed model, such as the typed coroutines and first-class composability, have been explored before in various forms [57, 34, 17, 16]. We build on earlier work, and we do not claim novelty for those features. However, to the best of our knowledge, the precise model that we describe is new, as we argue in Section 7 on related work.

Structure and organization. This paper is organized as follows:

- Section 2 describes the core primitives of the proposed programming model – coroutine definitions, coroutine instances, yielding, resuming, coroutine calls, and snapshots.
- Section 3 presents use-cases such as Erlang-style actors [66], `async-await` [23], Oz-style variables [65], event streams [20, 32], backtracking [36], and delimited continuations [56].
- In Section 4, we formalize coroutines with snapshots in $\lambda_{\rightsquigarrow}$ and prove its soundness.
- In Section 5, we describe the AST-level coroutine transformation implemented in Scala.
- In Section 6, we experimentally compare the performance of our implementation against Scala Delimited Continuations, Scala Async, iterators, and lazy streams.

Syntax. Our examples throughout this paper are in the Scala programming language [37]. We took care to make the paper accessible to a wide audience by using a minimal set of Scala features. The `def` keyword declares a method, while `var` and `val` declare mutable and final variables, respectively. Lambdas are declared with the right-arrow symbol `=>`. Type annotations go after the variable name, colon-delimited (`:`). Type parameters are put into square brackets [`]`. Parenthesis can be omitted from nullary and unary method calls.

2 Programming Model

In this section, we describe the proposed programming model through a series of examples.

Coroutine definitions. A subroutine is a sequence of statements that carry out a task. The same subroutine may execute many times during execution. When a program calls a subroutine, execution suspends at that callsite and continues in the subroutine. Execution at the callsite resumes only after the subroutine completes. For example, the program in Listing 1 declares a subroutine that doubles an integer, and then calls it with the argument 7.

■ **Listing 1** Subroutine example.

```
1 val dup = (x:Int) => { x + x }
2 dup(7)
```

■ **Listing 2** Coroutine example.

```
1 val dup =
2   coroutine { (x:Int) => x + x }
```

Upon calling `dup`, the subroutine does an addition, returns the result and terminates. When the execution resumes from the callsite, the subroutine invocation no longer exists.

Coroutines generalize subroutines by being able to suspend during their execution, so that their execution can be resumed later. In our implementation, a coroutine is defined inside the `coroutine` block. We show the coroutine equivalent of `dup` in Listing 2.

Yielding and resuming. Once started, the `dup` coroutine from Listing 2 runs to completion without suspending. However, a typical coroutine will suspend at least once. When it does, it is useful that it *yields* a value to the caller, explaining why it is suspended.

Consider the coroutine `rep` in Listing 3, which takes one argument `x`. The `rep` coroutine invokes the `yieldval` primitive to yield the argument `x` back to its caller, twice in a row.

3:4 Theory and Practice of Coroutines with Snapshots

■ **Listing 3** Yielding example.

```
1 val rep = coroutine { (x:Int) =>
2   yieldval(x)
3   yieldval(x)
4 }
```

■ **Listing 4** Execution states.

```
1 { ↑yieldval(7); yieldval(7) } =>
2 { yieldval(7); ↑yieldval(7) } =>
3 { yieldval(7); yieldval(7)↑ } =>
4 { yieldval(7); yieldval(7) }
```

Listing 4 show the states that the coroutine undergoes during its execution for $x=7$. The upward arrow (\uparrow) denotes the program counter. After getting invoked, the coroutine is paused in line 1 before the first `yieldval`. The caller resumes the coroutine (resuming is explained shortly), which then executes the next `yieldval` and yields the value 7 in line 2. The caller resumes the coroutine again, and the coroutine executes the last `yieldval` in line 3. The caller then resumes the coroutine the last time, and the coroutine terminates in line 4. The termination of a coroutine is similar to a termination of a subroutine – once the control flow reaches the end, the invocation of the corresponding coroutine no longer exists.

Delimited coroutines. As stated in the introduction, the proposed coroutine model is *delimited*. This means that the `yieldval` keyword can only occur inside a scope that is lexically enclosed with the `coroutine` keyword – a free `yieldval` results in a compiler error.

By itself, this restriction could hinder composability. Consider a hash table with closed addressing, which consists of an array whose entries are lists of elements (buckets). We would like a coroutine that traverses the elements of the hash table. Given a separately implemented `bucket` coroutine from Listing 5 that yields from a list, it is handy if a hash table coroutine can reuse this existing functionality by passing buckets from which to yield.

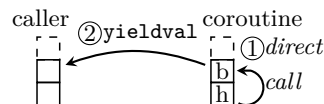
■ **Listing 5** List coroutine.

```
1 val bucket = coroutine {
2   (b: List[Int]) =>
3   while (b != Nil) {
4     yieldval(b.head)
5     b = b.tail
6   }
7 }
```

■ **Listing 6** Hash table coroutine.

```
1 val hashtable = coroutine {
2   (t: Array[List[Int]]) =>
3   var i = 0
4   while (i < t.length) {
5     bucket(t(i)); i += 1
6   }
7 }
```

Stackful coroutines. To allow composing separately written coroutines, it must be possible for one coroutine to call into another coroutine, but retain the same yielding context. The `hashtable` coroutine in Listing 6 traverses the array entries, and calls `bucket` for each entry. The two coroutines yield values together, as if they were a single coroutine.



Similar to ordinary subroutine calls, when the `hashtable` coroutine calls `bucket`, it must store its local variables and state. One way to achieve this is to use a call stack. When the program resumes the `hashtable` coroutine, it switches from the normal program call stack to a separate call stack that belongs to the resumed coroutine instance. The `hashtable` coroutine then pushes its state to the stack, and passes the control flow to the `bucket` coroutine (① in the figure below). The `bucket` coroutine stores its state to the stack and yields to the same caller that originally resumed the `hashtable` coroutine (②).

By saying that our coroutine model is *stackful*, we mean that coroutines are able to call each other, and yield back to the same resume-site [34]. In our implementation, this is enabled with an artificial call stack, as explained in Section 5.

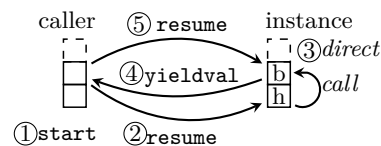
Importantly, a coroutine call can only occur in a scope that is lexically enclosed with the `coroutine` keyword (akin to `yieldval`). Only a coroutine body can call another coroutine – a free call is a compiler error. A natural question follows: how does a normal program create a new coroutine instance?

Coroutine instances. Similar to how invoking a *subroutine* creates a running instance of that subroutine, starting a *coroutine* creates a new *coroutine instance*. A subroutine's execution is not a program value – it cannot be observed or controlled. However, after creating a coroutine instance, the caller must interact with it by reading the yielded values and resuming. Therefore, it is necessary to treat the coroutine instance as a program value.

To distinguish between normal coroutine calls and creating a coroutine instance, we introduce a separate `start` keyword, as a design choice. The instance encapsulates a new call stack. After `start` returns a new coroutine instance, the caller can invoke `resume` and `value` on the instance. By returning `true`, the `resume` indicates that the instance yielded, and did not yet terminate. In this case, invoking `value` returns the last yielded value.

■ **Listing 7** Starting and resuming.

```
1 val i = hashtable.start(array)
2 var sum: Int = 0
3 while (i.resume) sum += i.value
```



Listing 7 shows how to use the hash table coroutine to compute the sum of the hash table values. A new coroutine instance is started in line 1 using the `hashtable` coroutine. In the `while` loop in line 3, the values are extracted from the instance until `resume` returns `false` (subsequent calls to `resume` result in a runtime error in our implementation). The figure on the right shows the creation of the instance, followed by two `resume` calls.

Typed coroutines. In the example in Listing 7, the `sum` variable in line 2 is integer-typed. Therefore, the right hand of the assignment in line 3 must also have the integer type. This illustrates that it is useful to reason about the type of the yielded values.

A coroutine type encodes the types P_i of its parameters, the return type R , and the *yield type* Y of the yielded values, also called the *output type* [57]. Its general form is $(P_1, \dots, P_N) \rightsquigarrow (Y, R)$ ¹. In the following, we annotate the `dup` coroutine from Listing 2:

```
val dup: Int~>(Nothing, Int) = coroutine { (x: Int) => x + x }
```

Since this coroutine does not yield any values, its yield type is `Nothing`, the bottom type in Scala. A coroutine `once`, which yields its argument once, has the yield type `Int`:

```
val once: Int~>(Int, Unit) = coroutine { (x: Int) => yieldval(x) }
```

The `bucket` and `hashtable` coroutines from Listings 5 and 6 have the following types:

```
val bucket: List[Int]~>(Int, Unit) = ...
val hashtable: Array[List[Int]]~>(Int, Unit) = ...
```

Coroutine instances have a separate type $Y \leftarrow \rightsquigarrow R$, where Y is the yield type, and R is the return type. For example, starting `once` produces an instance with the type `Int<~>Unit`:

```
val i: Int<~>Unit = once.start(7)
```

Instances of the `bucket` and `hashtable` coroutines also have the type `Int<~>Unit`.

¹ We note that Scala allows operator syntax, such as `~>`, when declaring custom data types.

Snapshots. We extend standard coroutines with the `snapshot` operation, which takes a coroutine instance and returns a duplicated instance with exactly the same execution state.

```
val i1: Int<~>Unit = once.start(7)
val i2: Int<~>Unit = i1.snapshot
```

In the previous example, the coroutine instance `i1` is duplicated to `i2`. Subsequent coroutine operations can be called independently on the two coroutine instances.

3 Use Cases

Having explained what type-safe, delimited and stackful means, we motivate our design choices with observations and concrete examples. The goal of this section is to show how specific primitives help express other kinds of suspendable computations.

► **Observation 1.** Stackful coroutines allow composing suspendable software modules.

The hash table example from Listings 5 and 6 demonstrated why composing different modules is useful, and the remaining examples in this section reinforce this.

► **Observation 2.** Stackful coroutines simplify the interaction with recursive data types.

Iterators. Data structure iterators are one of the earliest applications of coroutines [29]. We can implement an iterator coroutine `it` for binary trees as shown in Listing 8:

■ **Listing 8** Tree iterator implementation.

```
1 val it = coroutine { (t:Tree) =>
2   if (t.fst != null) it(t.fst)
3   yieldval(t.element)
4   if (t.snd != null) it(t.snd) }
```

■ **Listing 9** Symmetric coroutine.

```
1 type Sym[R] = Sym[R] <~> R
2 def run(i: Sym[R]): R =
3   if (i.resume) run(i.value)
4   else i.result
```

► **Observation 3.** Asymmetric coroutines can be used to express symmetric coroutines.

Symmetric coroutines. In a programming model with symmetric coroutines, there is no `resume` primitive. Instead, a symmetric coroutine always yields the next coroutine instance from which to continue. As shown in Listing 9, a symmetric coroutine instance can be expressed with the recursive type `Sym[R]`. This was observed before [34].

► **Observation 4.** Coroutine calls and coroutine return values, together with the `yieldval` primitive, allow encoding alternative forms of suspension primitives.

The core idea is to express a suspension primitive as a special coroutine. This coroutine yields a value that allows the resume-site to communicate back. Once resumed, the coroutine *returns another value* to its caller coroutine. We show several examples of this pattern.

Async-await. A future is an entity that initially does not hold a value, but may asynchronously attain it at a later point. Future APIs are usually callback-based – the code that handles the future value is provided as a function. In Scala, type `Future` exposes the `onSuccess` method, which takes a callback and calls it once the value is available, and `value`, which can be used to access the future value if available. `Promise` is the future’s writing end, and it exposes `success`, which sets the value of the corresponding future at most once [22].

■ **Listing 10** Callback-style API.

```

1 val token: Future[Token] =
2   authenticate()
3 token.onSuccess { t =>
4   val session: Future[Session] =
5     sessionFor(t)
6   session.onSuccess(useSession)
7 }

```

■ **Listing 11** Async-await-style API.

```

1 async {
2   val token: Token =
3     await { authenticate() }
4   val session: Session =
5     await { sessionFor(token) }
6   useSession(session)
7 }

```

In Listing 10, the `authenticate` method returns a future with a `Token` value. Once the token arrives, the callback function (passed to the `onSuccess` method) calls `sessionFor` to obtain a session future. The program continues in the `useSession` callback. The direct-style async-await version, shown in Listing 11, relies on the `async` statement, which starts asynchronous computations, and the `await`, which suspends until a value is available.

We use coroutines to replicate Scala’s Async framework [23]. The `await` method emulates the suspension primitive – it creates a coroutine that yields a future and returns its value. It assumes that the resume-site invokes `resume` only *after* the future is completed.

■ **Listing 12** The `await` primitive.

```

1 def await[T]: Future[T] ~> (Future[T], T) =
2   coroutine { (f: Future[T]) => yieldval(f); f.value }

```

The `async` method interacts with `await` from the resume-site. For a given computation `b`, it starts an instance, and executes it asynchronously in a `Future`. Whenever the computation `b` yields a future, a callback is recursively installed. If `resume` returns `false`, the resulting future is completed with the evaluation result of `b` (`async` itself must return a future).

■ **Listing 13** The `async` primitive.

```

1 def async[R](b: () ~> (Future[Any], R)): Future[R] = {
2   val i = b.start()
3   val p = new Promise[R]
4   @tailrec def loop(): Unit =
5     if (i.resume) i.value.onSuccess(loop) else p.success(i.result)
6   Future { loop() }
7   p.future
8 }

```

Erlang-style actors. Actor frameworks for the JVM are unable to implement exact Erlang-style semantics, in which the `receive` statement can be called anywhere in the actor [21]. For example, Akka exposes a top-level `receive` method [1], which returns a partial function used to handle messages. This function can be swapped with the `become` statement.

Listing 14 shows a minimal Akka actor example that implements a server. The server starts its execution in the `receive` method, which awaits a password from the user. If the password is correct, the top-level event-handling loop becomes the `loggedIn` method, which accepts GET requests. When the `Logout` message arrives, the actor stops. Listing 15 shows an equivalent Erlang-style actor, in which the control flow is more apparent. The `receive` method has the role of a suspension primitive – it pauses the actor until a message arrives.

■ **Listing 14** Akka-style actor.

```

1 class Server extends Actor {
2   def receive = {
3     case Login(pass) =>
4       assert(isCorrect(pass))

```

```

5     become(loggedIn) }
6   def loggedIn = {
7     case Get(url) => serve(url)
8     case Logout() => stop() } }

```

■ Listing 15 Erlang-style actor.

```

1 def server() = {
2   val Login(pass) = receive()
3   assert(isCorrect(pass))
4   while (true) receive() match {

```

```

5     case Get(url) => serve(url)
6     case Logout() => stop()
7   }
8 }

```

As shown in the appendix, the `receive` coroutine follows a similar pattern as `async-await` – `receive` yields an object into which the top-level loop can insert the message.

Event streams. First-class Rx-style event streams [32] expose a set of declarative transformation combinators. As an example, consider how to collect a sequence of points when dragging the mouse. The mouse events are represented as an event stream value. Dragging starts when the mouse is pressed down, and ends when released. In Listing 16, the `after` combinator removes a prefix of events, and `until` removes a suffix. The first drag event’s `onEvent` callback creates a `Curve` object, and the last event saves the curve.

■ Listing 16 Rx-style streams.

```

1 val drag = mouse.after(_ .isDown)
2   .until(_ .isUp)
3 drag.first.onEvent { e =>
4   val c = new Curve(e.x, e.y)
5   drag.onEvent(
6     e => c.add(e.x, e.y))
7 drag.last.onEvent(
8   e => saveCurve(c)) }

```

■ Listing 17 Direct-style streams.

```

1 var e = mouse.get
2 while (!e.isDown) e = mouse.get
3 val c = new Curve(e.x, e.y)
4 while (e.isDown) {
5   e = mouse.get
6   c.add(e.x, e.y)
7 }
8 saveCurve(c)

```

The equivalent direct-style program in Listing 17 uses the `get` coroutine to suspend the program until an event arrives. We show the implementation of `get` in the appendix.

Oz-style single-assignment variables. A variable in the Oz language [65] can be assigned only once. Reading a single-assignment variable suspends execution until some other thread assigns a value. Assigning a value to the variable more than once is an error.

■ Listing 18 Oz-style variable read.

```

1 @volatile var state: AnyRef =
2   List.empty()
3
4 val get = coroutine { () =>
5   if (READ(state).is[List])
6     yieldval(this)
7   READ(state).as[ElemType]
8 }

```

■ Listing 19 Oz-style variable write.

```

1 @tailrec def set(y: ElemType) {
2   val x = READ(state)
3   if (x.is[List]) {
4     if (CAS(state, x, y))
5       for (i <- x.as[List])
6         schedule(i)
7     else set(y)
8   } else throw Reassigned }

```

Internally, a single-assignment variable has some `state`, which is either the list of suspended threads or a value. When `get` from the Listing 18 is called, `state` is atomically read in line 6. If the state is a list, coroutine is yielded to the scheduler, which atomically adds the new suspended thread to the list (not shown). The coroutine is resumed when the value becomes available – method `set` in Listing 19 tries to atomically replace the list with the value of type `ElemType`. If successful, it schedules the suspended threads for execution.

► **Observation 5.** Snapshots enable backtracking and allow emulating full continuations.

Backtracking. Testing frameworks such as ScalaCheck [36] rely on backtracking to systematically explore the test parameter space. ScalaCheck uses a monadic-style API to compose the parameter generators. Listing 20 shows a ScalaCheck-style test that first creates a generator for number pairs in a Scala `for`-comprehension, and then uses the generator in a commutativity test. The `pairs` generator is created from the intrinsic `ints` generator.

In the direct-style variant in Listing 21, the `ints` generator is a coroutine that yields and captures the execution snapshot. Therefore, it can be called from anywhere inside the test.

■ **Listing 20** Monadic ScalaCheck test.

```

1 val pairs =
2   for {
3     x <- ints(0 until MAX_INT)
4     y <- ints(0 until MAX_INT)
5   } yield (x, y)
6
7 forAll(pairs) { pair =>
8   val (a, b) = pair
9   assert(a * b == b * a)
10 }
11
```

■ **Listing 21** Direct-style ScalaCheck test.

```

1 test {
2   val a = ints(0 until MAX_INT)
3   val b = ints(0 until MAX_INT)
4   assert(a * b == b * a)
5 }
```

■ **Listing 22** Positive-definite matrix test.

```

1 val pd = coroutine { (m:Mat) =>
2   val x = nonZeroVector(m.size)
3   assert(x.t * m * x > 0)
4 }
```

Moreover, generator late-binding allows modularizing the properties. Listing 22 shows a modular *positive-definite* matrix property `pd`: given a matrix M , value $x^T M x$ is positive for any non-zero vector x . The `pd` coroutine can be called from within other tests. Importantly, note that the vector x is generated *from within* the test. This is hard to achieve in the standard ScalaCheck tests, since their generators require prior knowledge about the vector x .

Consider implementing the `test` and the `ints` primitives from Listing 21. The key idea is as follows: each time a test calls a generator, it suspends and yields a list of environment setters. Each environment setter is a function that prepares a value to return from the generator. The resume-site runs each environment setter, creates a snapshot and resumes it.

■ **Listing 23** Direct-style ScalaCheck.

```

1 type Test =
2   () -> (List[() => Unit], Unit)
3
4 type Instance =
5   List[() => Unit] <-> Unit
6
7 def backtrack(i: Instance) = {
8   if (i.resume)
9     for (setEnv <- i.value) {
10      setEnv()
11      backtrack(i.snapshot)
12    }
13 }
```

■ **Listing 24** Direct-style ScalaCheck, cont.

```

1 val ints = coroutine {
2   (xs: List[Int]) =>
3     var env: Int = _
4     val setEnvs =
5       xs.map(x => () => env = x)
6     yieldval(setEnvs)
7     env
8 }
9
10 def test(t: Test) = {
11   val instance = t.start()
12   backtrack(instance)
13 }
```

In Listing 23, we first declare two type aliases `Test` and `Instance`, which represent a test coroutine and a running test instance. Their yield type is a list of environment setters `List[() => Unit]`. The `backtrack` subroutine takes a running test instance, and resumes it. If the instance yields, then the test must have called a generator, so `backtrack` traverses the environment setters and recursively resumes a snapshot for each of them. Thus, each recursive call to `backtrack` represents a node in the respective backtracking tree.

The `ints` generator in Listing 24 is a coroutine that takes a list of integers `xs` to choose from. It starts by creating a local variable `env`, and a list of functions that set `env` (one for each integer in `xs`). The generator then yields this list. Each time `ints` gets resumed, the `env` variable is set to a proper value, so it is returned to the test that called it.

$t ::=$	terms:	$T ::=$	types:
$(x:T) \Rightarrow t$	abstraction	$T \Rightarrow T$	function type
$t(t)$	application	$T \overset{T}{\rightsquigarrow} T$	coroutine type
x	variable	$T \overset{\leftarrow}{\rightsquigarrow} T$	instance type
$()$	unit value	Unit	unit type
$(x:T) \overset{T}{\rightsquigarrow} t$	coroutine	\perp	bottom type
$\text{yield}(t)$	yielding	$r ::=$	runtime terms:
$\text{start}(t, t)$	starting	i	instance
$\text{resume}(t, t, t, t)$	resuming	$\langle t, v, v, v \rangle_i$	resumption
$\text{snapshot}(t)$	snapshot	$\llbracket t \rrbracket_v$	suspension
$\text{fix}(t)$	recursion	\emptyset	empty term
i	instance	$v ::=$	values:
$\langle t, v, v, v \rangle_i$	resumption	$(x:T) \Rightarrow t$	abstraction
$\llbracket t \rrbracket_v$	suspension	$()$	unit value
\emptyset	empty term	$(x:T) \overset{T}{\rightsquigarrow} t$	coroutine
		i	instance
		\emptyset	empty term

■ **Figure 1** Syntax and types of the $\lambda_{\rightsquigarrow}$ calculus.

Continuations. Shift-reset-style delimited continuations use the `reset` operator to delimit program regions for the CPS-transform [12]. The `shift` operator, which takes a function whose input is the continuation, can be used inside these regions. We sketch the implementation of `shift` and `reset` similar to those in Scala delimited continuations [56].

```

1 type Shift = (() => Unit) => Unit
2 def reset(b: () ~> (Shift, Unit)): Unit = {
3   def continue(i: Shift <~> Unit) =
4     if (i.resume) i.value(() => continue(i.snapshot))
5     continue(b.start())
6 }
7 def shift: Shift ~> (Shift, Unit) =
8   coroutine { (b: Shift) => yieldval(b) }
```

The type alias `Shift` represents continuation handlers – functions that take continuations of the current program. The `reset` operator takes a coroutine that can yield a `Shift` value. It starts a new coroutine instance and resumes it. If this instance calls `shift` with a continuation handler, the handler is yielded back to `reset`, which creates an instance snapshot and uses it to create a continuation. The continuation is passed to the continuation handler. The use of `snapshot` is required, as the continuation can be invoked multiple times.

4 Formal Semantics

This section presents the $\lambda_{\rightsquigarrow}$ (pron. *lambda-squiggly*) calculus that captures the core of the programming model from Section 2. This calculus is an extension of the simply-typed lambda-calculus. The complete formalization, along with the proofs of the progress and preservation theorems, is given in the corresponding tech report [51].

Syntax. Figure 1 shows the syntax. The abstraction, application and variable terms are standard. The *coroutine* term represents coroutine declarations. The *yield* term corresponds

$$\begin{array}{c}
\frac{\Sigma|\Gamma, x:T_1 \vdash t_2:T_2|\perp}{\Sigma|\Gamma \vdash (x:T_1)\Rightarrow t_2:T_1\Rightarrow T_2|\perp} \quad (T\text{-ABS}) \qquad \frac{\Sigma|\Gamma \vdash t_1:T_2\Rightarrow T_1|T_y}{\Sigma|\Gamma \vdash t_2:T_2|T_y} \quad (T\text{-APP}) \qquad \frac{x:T \in \Gamma}{\Sigma|\Gamma \vdash x:T|\perp} \quad (T\text{-VAR}) \qquad \frac{\Sigma|\Gamma \vdash t:T|\perp}{\Sigma|\Gamma \vdash t:T|T_y} \quad (T\text{-CTX}) \\
\\
\frac{\Sigma|\Gamma \vdash ():\mathbf{Unit}|\perp}{(T\text{-UNIT})} \qquad \frac{\Sigma|\Gamma, x:T_1 \vdash t_2:T_2|T_y}{\Sigma|\Gamma \vdash (x:T_1) \overset{T_y}{\rightsquigarrow} t_2:T_1 \overset{T_y}{\rightsquigarrow} T_2|\perp} \quad (T\text{-COROUTINE}) \qquad \frac{\Sigma|\Gamma \vdash t_1:T_1 \overset{T_y}{\rightsquigarrow} T_2|T_w}{\Sigma|\Gamma \vdash t_2:T_1|T_w} \quad (T\text{-START}) \\
\\
\frac{\Sigma|\Gamma \vdash t:T|T}{\Sigma|\Gamma \vdash \mathbf{yield}(t):\mathbf{Unit}|T} \quad (T\text{-YIELD}) \qquad \frac{\Sigma|\Gamma \vdash t:T_y \rightsquigarrow T_2|T_w}{\Sigma|\Gamma \vdash \mathbf{snapshot}(t):T_y \rightsquigarrow T_2|T_w} \quad (T\text{-SNAPSHOT}) \qquad \frac{\Sigma|\Gamma \vdash t:T\Rightarrow T|\perp}{\Sigma|\Gamma \vdash \mathbf{fix}(t):T|\perp} \quad (T\text{-FIX}) \\
\\
\frac{\Sigma|\Gamma \vdash t_1:T_y \rightsquigarrow T_2|T_w \quad \Sigma|\Gamma \vdash t_2:T_2 \overset{T_w}{\rightsquigarrow} T_R|T_w}{\Sigma|\Gamma \vdash t_3:T_y \overset{T_w}{\rightsquigarrow} T_R|T_w} \quad \frac{\Sigma|\Gamma \vdash t_4:\mathbf{Unit} \overset{T_w}{\rightsquigarrow} T_R|T_w}{\Sigma|\Gamma \vdash \mathbf{resume}(t_1, t_2, t_3, t_4):T_R|T_w} \quad (T\text{-RESUME}) \qquad \frac{\Sigma|\Gamma \vdash t_1:T_2 \overset{T_y}{\rightsquigarrow} T_1|T_y \quad \Sigma|\Gamma \vdash t_2:T_2|T_y}{\Sigma|\Gamma \vdash t_1(t_2):T_1|T_y} \quad (T\text{-APPCOR}) \\
\\
\frac{\Sigma|\Gamma \vdash t:T|T_y \quad \Sigma|\Gamma \vdash v:T_y|\perp}{\Sigma|\Gamma \vdash \llbracket t \rrbracket_v:T|T_y} \quad (T\text{-SUSPENSION}) \qquad \frac{\Sigma(i) = T_y \rightsquigarrow T_2 \quad \Sigma|\Gamma \vdash t_1:T_2|T_y \quad \Sigma|\Gamma \vdash v_2:T_2 \overset{T_w}{\rightsquigarrow} T_R|\perp}{\Sigma|\Gamma \vdash v_3:T_y \overset{T_w}{\rightsquigarrow} T_R|\perp} \quad \frac{\Sigma|\Gamma \vdash v_4:\mathbf{Unit} \overset{T_w}{\rightsquigarrow} T_R|\perp}{\Sigma|\Gamma \vdash \langle t_1, v_2, v_3, v_4 \rangle_i:T_R|T_w} \quad (T\text{-RESUMPTION}) \\
\\
\frac{\Sigma(i) = T_y \rightsquigarrow T_2}{\Sigma|\Gamma \vdash i:T_y \rightsquigarrow T_2|\perp} \quad (T\text{-INSTANCE}) \qquad \Sigma|\Gamma \vdash \emptyset:T|\perp \quad (T\text{-EMPTY})
\end{array}$$

■ **Figure 2** Typing rules for the $\lambda_{\rightsquigarrow}$ calculus.

to the `yieldval` statement shown earlier. The `start` term is as before, but uses prefix syntax. The `resume` term encodes both the `resume` and the `value` from Section 2. This is because resuming an instance can complete that instance (in our implementation, `resume` returns `false`), it can result in a yield (previously, `true`), or fail because an instance had already completed earlier (in our implementation, an exception is thrown). In $\lambda_{\rightsquigarrow}$, the `resume` term therefore accepts four arguments: the coroutine instance, the result handler, the yield handler, and the handler for the already-completed case. The `fix` term supports recursion [39].

The calculus differentiates between *user terms*, which appear in user programs, and *runtime terms*, which only appear during the program evaluation. A label i is used to represent a coroutine instance. Each coroutine instance has an evaluation state, which changes over the lifetime of the instance. A resumed instance i is represented by the *resumption term* $\langle t, v, v, v \rangle_i$. A term that yielded a value v , and is about to be suspended, is represented by the *suspension term* $\llbracket t \rrbracket_v$. Finally, the *empty term* \emptyset is used in the store μ (shown shortly) when encoding terminated coroutines. The abstraction term $(x:T)\Rightarrow t$, the unit term $()$, the coroutine definition $(x:T) \overset{T_y}{\rightsquigarrow} t$, the instance label i and the empty term \emptyset are considered values.

Note that the calculus distinguishes between standard function types $T_1\Rightarrow T_2$ and coroutine types of the form $T_1 \overset{T_y}{\rightsquigarrow} T_2$, where T_y is the type of values that the coroutine may yield. Coroutine instances have the $T_y \rightsquigarrow T_2$ type, and a unit value has the type `Unit`. The bottom type \perp is used to represent the fact that a term does not yield.

Example. Recall the `rep` coroutine from Listing 3. We can encode it in $\lambda_{\rightsquigarrow}$ as follows:

$$(x:\mathbf{Int}) \overset{\mathbf{Int}}{\rightsquigarrow} ((u:\mathbf{Unit}) \overset{\mathbf{Int}}{\rightsquigarrow} \mathbf{yield}(x))(\mathbf{yield}(x))$$

The encoding uses a standard trick to sequence two statements [39], but instead of a regular lambda, relies on a coroutine to ignore the result of the first statement. Starting this coroutine creates an instance i , whose current term is saved in the store:

$$\mathbf{start}((x:\mathbf{Int}) \overset{\mathbf{Int}}{\rightsquigarrow} ((u:\mathbf{Unit}) \overset{\mathbf{Int}}{\rightsquigarrow} \mathbf{yield}(x))(\mathbf{yield}(x)), 7) \rightarrow i$$

Assume that we now resume this instance once. We provide three handlers to **resume**. We show the complete yield handler (here, identity), and name the other two c_2 and c_4 :

$$\begin{aligned} & \mathbf{resume}(i, c_2, (x:\mathbf{Int}) \overset{\perp}{\rightsquigarrow} x, c_4) \rightarrow \\ & \langle ((u:\mathbf{Unit}) \overset{\mathbf{Int}}{\rightsquigarrow} \mathbf{yield}(7))(\mathbf{yield}(7)), c_2, (x:\mathbf{Int}) \overset{\perp}{\rightsquigarrow} x, c_4 \rangle \rightarrow \\ & \langle ((u:\mathbf{Unit}) \overset{\mathbf{Int}}{\rightsquigarrow} \mathbf{yield}(7))(\llbracket () \rrbracket_7), c_2, (x:\mathbf{Int}) \overset{\perp}{\rightsquigarrow} x, c_4 \rangle \rightarrow \\ & \langle \llbracket ((u:\mathbf{Unit}) \overset{\mathbf{Int}}{\rightsquigarrow} \mathbf{yield}(7))(\llbracket () \rrbracket) \rrbracket_7, c_2, (x:\mathbf{Int}) \overset{\perp}{\rightsquigarrow} x, c_4 \rangle \rightarrow ((x:\mathbf{Int}) \overset{\perp}{\rightsquigarrow} x)(7) \rightarrow 7 \end{aligned}$$

Typing. Before showing the typing rules, we introduce the *instance typing* Σ , which tracks coroutine instance types, and is used alongside the standard typing context Γ .

► **Definition 6** (Instance typing). The *instance typing* Σ is a sequence of coroutine instance labels and their types $i:T$, where comma $(,)$ extends a typing with a new binding.

► **Definition 7** (Typing relation). The *typing relation* $\Sigma|\Gamma \vdash \mathbf{t}:T|\mathbf{T}_y$ in Fig. 2 is a relation between the instance typing Σ , the typing context Γ , the term \mathbf{t} , its type T , and the *yield type* \mathbf{T}_y , where \mathbf{T}_y is the type of values that may be yielded when evaluating \mathbf{t} .

We inspect the most important rules here, and refer the reader to the tech report [51] for a complete discussion. The T-ABS rule is the modification of the standard abstraction typing rule. Note that the yield type of the function body **must be** \perp . This is because $\lambda_{\rightsquigarrow}$ models *delimited coroutines* – if a **yield** expression occurs, it must instead be lexically enclosed within a coroutine term (which corresponds to the **coroutine** statement). We emphasize that $\lambda_{\rightsquigarrow}$ nevertheless models stackfulness – a **yield** can still cross coroutine boundaries at runtime if a coroutine calls another coroutine, as illustrated in Section 3, and explained shortly.

Note further, that the T-APP rule permits a non- \perp type on the subterms, since the reduction of the function and its arguments is itself allowed to yield. A non-yielding term can be assumed to yield any type by the T-CTX rule. Given a term whose type is T and yield type is also T , the T-YIELD rule types a **yield** expression as **Unit** with the yield type T .

The T-COROUTINE rule allows the body to yield a value of the type \mathbf{T}_y , which must correspond to the yield type of the coroutine. The coroutine itself gets a \perp yield type (the coroutine definition effectively swallows the yield type). Consider now the T-APPCOR rule, which is similar to the standard T-APP rule for functions. To directly call another coroutine \mathbf{t}_1 , its yield type \mathbf{T}_y must correspond to the yield type at the callsite.

Last, we examine the runtime term typing rules. The T-SUSPENSION rule requires that the yielded value \mathbf{v} has the type \mathbf{T}_y , and that the suspension has the same type and yield type as the underlying suspended term \mathbf{t} . The T-INSTANCE rule requires that the instance typing Σ contains a corresponding type for i . Finally, the T-RESUMPTION term has the type \mathbf{T}_R that corresponds to the return types of the handler coroutines \mathbf{t}_2 , \mathbf{t}_3 and \mathbf{t}_4 . The corresponding yield type is \mathbf{T}_w , which is generally different from the yield type \mathbf{T}_y that the coroutine instance evaluation \mathbf{t}_1 can yield. The empty term can be assigned any type.

► **Definition 8** (Well-typed program). A term \mathbf{t} is *well-typed* if and only if $\exists T, \mathbf{T}_y, \Sigma$ such that $\Sigma|\emptyset \vdash \mathbf{t}:T|\mathbf{T}_y$. A term \mathbf{t} is a *well-typed user program* if \mathbf{t} is well-typed and $\mathbf{T}_y = \perp$.

$$\begin{array}{c}
\frac{i \notin \text{dom}(\mu)}{\text{start}((x:T_1) \overset{T_y}{\rightsquigarrow} t, v) | \mu \rightarrow i | \mu, i \triangleright [x \mapsto v]t} \quad \frac{i_2 \notin \text{dom}(\mu) \quad i_1 \neq i_2}{\text{snapshot}(i_1) | \mu, i_1 \triangleright t \rightarrow i_2 | \mu, i_1 \triangleright t, i_2 \triangleright t} \\
\text{(E-START)} \qquad \qquad \qquad \text{(E-SNAPSHOT)} \\
\text{yield}(v) | \mu \rightarrow [()]_v | \mu \qquad \langle [t_1]_v, v_2, v_3, v_4 \rangle_i | \mu, i \triangleright [t_0]_{v'} \rightarrow v_3(v) | \mu, i \triangleright t_1 \\
\text{(E-YIELD)} \qquad \qquad \qquad \text{(E-CAPTURE)} \\
\frac{t \neq [t_0]_{\emptyset}}{\text{resume}(i, v_2, v_3, v_4) | \mu, i \triangleright t \rightarrow \langle t, v_2, v_3, v_4 \rangle_i | \mu, i \triangleright [t]_{\emptyset}} \text{(E-RESUME1)} \\
\text{resume}(i, v_2, v_3, v_4) | \mu, i \triangleright [t_0]_{\emptyset} \rightarrow v_4(()) | \mu, i \triangleright [t_0]_{\emptyset} \text{(E-RESUME2)} \\
\langle v, v_2, v_3, v_4 \rangle_i | \mu, i \triangleright [t_0]_{v'} \rightarrow v_2(v) | \mu, i \triangleright [v]_{\emptyset} \text{(E-TERMINATE)}
\end{array}$$

■ **Figure 3** A subset of evaluation rules in the $\lambda_{\rightsquigarrow}$ calculus.

Semantics. Before showing the operational semantics, we introduce the concept of a coroutine store μ , which is used to track the evaluation state of the coroutine instances.

► **Definition 9** (Coroutine store). A *coroutine store* μ is a sequence of coroutine instance labels i and their respective evaluation terms t , where the comma operator $(,)$ extends the coroutine store with a new binding $i \triangleright t$.

We only show a subset with the most important evaluation rules in Fig. 3, and present the complete set of rules in the tech report [51]. The E-START rule takes a coroutine value and an argument, and uses them to create a new coroutine instance i , where i is a fresh label. The coroutine store μ is modified to include a binding from i to the coroutine body t after substitution. Such a coroutine instance i can then be resumed by the E-RESUME1 rule, which reduces a `resume` expression into an resumption term $\langle t, v_2, v_3, v_4 \rangle_i$. Note the convention that an executing or a terminated coroutine has a suspension term $[t_0]_{\emptyset}$ in the coroutine store μ . The E-RESUME1 rule applies if and only if $t \neq [t_0]_{\emptyset}$. If the instance is terminated, the E-RESUME2 rule applies instead, which just invokes the ‘callback’ v_4 to handle that case.

Consider now what happens when a resumption term yields. By E-YIELD, the expression `yield(v)` reduces to a suspended unit value that is yielding the value v . A suspension term then spreads across the surrounding terms. The following two example reductions spread the suspension across an application. There is one such rule for each non-value syntax form.

$$[[t_1]_v(t_2)] | \mu \rightarrow [[t_1(t_2)]_v] | \mu$$

$$v_1([t_2]_v) | \mu \rightarrow [v_1(t_2)]_v | \mu$$

Once the suspension reaches the coroutine resumption term, the E-CAPTURE reduces it to a call to the yield handler v_3 , and puts the term in the suspension into the store μ .

Safety. We now state the safety properties of $\lambda_{\rightsquigarrow}$. Complete proofs are given in the corresponding tech report article [51].

► **Definition 10** (Well-typed coroutine store). A coroutine store μ is well-typed with respect to the instance typing Σ , denoted $\Sigma \vdash \mu$, if and only if it is true that $\forall i \in \text{dom}(\mu)$, $\Sigma(i) = T_y \rightsquigarrow T_2 \Leftrightarrow \Sigma | \emptyset \vdash \mu(i) : T_2 | T_y$, and $\text{dom}(\Sigma) = \text{dom}(\mu)$.

► **Theorem 11 (Progress).** *Suppose that t is a closed, well-typed term for some T and Σ . Then, either t is a value, or t is a suspension term $\llbracket t \rrbracket_v$, or, for any store μ such that $\Sigma \vdash \mu$, there is some term t' and store μ' such that $t|\mu \rightarrow t'|\mu'$.*

► **Theorem 12 (Preservation).** *If a term and the coroutine store are well-typed, that is, $\Sigma|\Gamma \vdash t:T|T_y$, and $\Sigma|\Gamma \vdash \mu$, and if $t|\mu \rightarrow t'|\mu'$, then there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma'|\Gamma \vdash t':T|T_y$ and $\Sigma'|\Gamma \vdash \mu'$.*

► **Corollary 1 (Yield safety).** *If a user program τ_u is well-typed, then it does not evaluate to a suspension term of the form $\llbracket \tau \rrbracket_v$.*

5 Implementation

This section describes our metaprogramming-based coroutine implementation in Scala, which consists of a runtime library and an AST-level transformation.

5.1 Preliminaries

Our implementation relies on Scala Macros [9], the metaprogramming API in Scala². The following are the key metaprogramming features of Scala Macros that our transformation scheme relies on. First, it must be possible to declare *macro definitions* that take ASTs as input arguments and return transformed ASTs. Second, it must be possible to invoke such macro definitions from user programs, which makes the compiler execute the macro with its argument expressions and replace the macro invocation with the resulting ASTs. Third, it must be possible to decompose and compose ASTs inside the macro definition. Finally, it must be possible to inspect the types of the expressions passed to the macro, and reason about symbol identity. The following is an example of a macro definition:

```
1 def log(msg: String): Unit = macro log_impl
2 def log_impl(c: Context)(msg: c.Tree): c.Tree =
3   q"""if (loggingEnabled) info(currentTime() + ": " + $msg)"""
```

In the above, we declared a `log` macro with the macro implementation `log_impl`, which takes the corresponding `Tree` of `msg` as argument. The `log_impl` method uses the quasiquote notation `q"""` to build an AST [61], which in turn checks if logging is enabled before constructing the command line output from the `msg` string. Values are interpolated into the AST using the `$` notation, as is the case with the expression `$msg` above.

Scala macro definitions can be packaged into libraries. Our coroutine implementation is therefore a macro library, which the user program can depend on. User coroutines implemented with our coroutine library can be similarly packaged into third party libraries.

5.2 Runtime Model

When yielding, the coroutine instance must store the state of the local variables. When resuming, that state must be retrieved into the local variables. Since coroutines are stackful, i.e. they support nested coroutine calls, it is necessary to store the entire coroutine call stack.

² We have an ongoing second implementation that relies on a newer Scala Macros API, but we are using the original Scala Macros for the evaluation purposes in this paper.

Call stacks. Our implementation uses arrays to represent coroutine call stacks. A call stack is divided into a sequence of coroutine frames, which are similar to method invocation frames. Each frame stores the following:

- (1) pointer to the coroutine object (i.e. `Coroutine` block),
- (2) the position in the coroutine where the last yield occurred,
- (3) the local variables and the return values.

A coroutine call stack can be implemented as a single contiguous memory area. Scala is constrained by the JVM platform, on which arrays contain either object references or primitive values, but not both. Hence, our implementation separates coroutine descriptors, program counters and local variables into reference and value stacks.

It would be costly to create a large call stack whenever a coroutine instance starts. Many coroutines only need a single or several frames, so we set the initial stack size to 4 entries. When pushing, a stack overflow check potentially reallocates the stack, doubling its size. In the worst case, the reallocation overhead is $2\times$ compared to an optimally sized stack.

Coroutine instance class. A coroutine instance is represented by the `Instance` class shown in Listing 25. A new instance is created by the `start` method. In addition to the call stack, a coroutine instance tracks if the instance is live (i.e. non-terminated), if a nested call is in progress, what the last yield value was, the result value (if the instance terminated), and the exception that was thrown (if any). The coroutine instance exposes the `value`, `result` and `exception` user-facing methods, which either return the value of the respective field, or throw an error if the field is not set. The instance also exposes the `resume` method, which is described shortly.

■ **Listing 25** Coroutine instance class.

```

1 class Instance[Y, R] {
2   var _live = true
3   var _call = false
4   var _value: Y = null
5   var _result: R = null
6   var _exception:
7     Exception = null
8   /* Call stack arrays */
9 }

```

Coroutine class. For each `Coroutine` declaration in the program, the transformation macro generates a new anonymous subclass of the `Coroutine` class shown in Listing 26. Each concrete `Coroutine` subclass defines several entry point methods, and implements the `_enter` method of the `Coroutine` base class. An *entry point method* is a replica of the `Coroutine` block such that it starts from either:

- (1) the beginning of the method, or
- (2) a `yieldval` statement, or
- (3) a call to another coroutine.

The `_enter` method is called when a coroutine instance resumes. It reads the current position from the coroutine instance, and dispatches to the proper entry point method with a `switch` statement. A `goto` primitive is unavailable in Scala, so the `_enter` method emulates the `goto` semantics.

■ **Listing 26** Coroutine definition base classes.

```

1 class Coroutine[Y, R] {
2   def _enter(i: Instance[Y, R]): Unit

```

```

3 }
4 class Coroutine1[T0, Y, R]
5 extends Coroutine[Y, R] {
6   def _call(
7     i: Instance[Y, R], a0: T0): Unit
8 }
9 /* One class for each arity */

```

Listing 26 also shows the abstract `Coroutine1` subclass. The `Coroutine1` subclass declares the `_call` method which stores the resume-site or callsite arguments into the proper locations in the call stack. This method is invoked by `start` and by coroutine calls. Neither JVM nor Scala support variadic templates, so we include 4 different arity classes (the same approach is used for `Function` classes in Scala, and functional interface classes in Java).

Trampolining. An entry point method ends at a position at which the original `coroutine` block has a `yieldval`, a coroutine call, or a return. Therefore, each entry point method in the `Coroutine` class is tail-recursive. Consequently, nested coroutine calls can be invoked from a trampoline-style loop.

The `resume` method in Listing 27 implements a trampoline in lines 8-10. After resetting the yield-related fields, `resume` repetitively reads the topmost coroutine from the coroutine stack `_cstack`, and invokes `_enter`. If an entry point calls another coroutine, or returns from a coroutine call, the `_call` field is set to `true`. Otherwise, if the instance yields or terminates, the `_call` field is set to `false`, and the loop ends.

■ **Listing 27** The resume trampoline.

```

1 def resume[Y, R]
2   (i: Instance[Y, R]) = {
3   if (!i._live)
4     throw sys.error()
5   i._hasValue = false
6   i._value = null
7   do {
8     i._cstack(i._ctop)
9     ._enter(i)
10  } while (i._call)
11  i._live }

```

5.3 Transformation

The transformation is performed by the following `coroutine` macro, which takes a Scala AST, typed `c.Tree`. The `coroutine` macro checks that the AST is statically a function type, and reports a compiler error otherwise. The macro returns an AST that holds a definition of an anonymous `CoroutineN` subclass (for a specific arity `N`), and a new instance of that class.

```

1 def coroutine[T, R](f: Any): Any = macro coroutine_impl[T, R]
2 def coroutine_impl[T, R](c: Context)(f: c.Tree): c.Tree = ...

```

The transformation consists of four compilation phases. First, the input AST is converted into a normal form. Second, the normalized AST is converted into a control flow graph. Third, the control flow graph is cut into segments at the points where the coroutine yields or calls other coroutines. Finally, control flow graph segments are converted back to ASTs, which represent the coroutine's entry points and are used to generate the anonymous class.

AST normalization. This phase converts the input AST with arbitrary phrases into a normalized AST. The phrases in the normalized AST are restricted to:

- (1) single operator expressions on constants and identifiers,
- (2) assignments and declarations whose right-hand side is a constant or an identifier³,
- (3) method calls on constants and identifiers,
- (4) if-statements and while-loops whose condition is a constant or an identifier and whose body is normalized,
- (5) basic blocks whose statements are normalized.

The benefit of normalization is that the subsequent phases have fewer cases to consider.

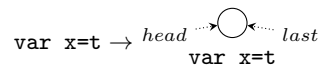
■ **Listing 28** Canonicalized list coroutine.

```

1 val bucket = coroutine {
2   (b: List[Int]) =>
3   var x_0 = b != Nil
4   var x_1 = x_0
5   while (x_1) {
6     var x_2 = b.head
7     var x_3 = yieldval(x_2)
8     var x_4 = b.tail
9     b = x_4
10    var x_0 = b != Nil
11    x_1 = x_0 }
12  () }

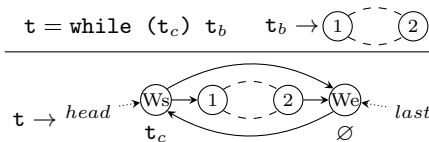
```

Example. Recall the `bucket` coroutine from Listing 5, which yields the elements of a list. After normalization, this coroutine is transformed into the coroutine in Listing 28.



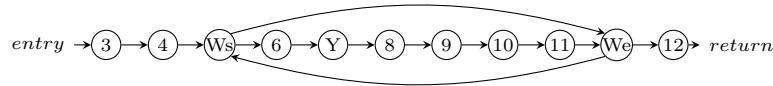
Control flow graph conversion. A normalized AST is converted to a control flow graph. The transformation is implemented as a set of mappings between the input AST and the output CFG nodes. An example rule for a variable declaration is informally shown above, where a declaration is replaced by a single node that records the AST.

The rule for `while`-loops, informally shown on the right, relies on the recursive transformation of the body t_b of the loop. Given that t_b transforms to a CFG that starts with a node 1 and ends with a node 2, a `while`-loop transforms to a pair of W_s and W_e nodes, which are connected with successor links (solid lines) as shown in the figure.



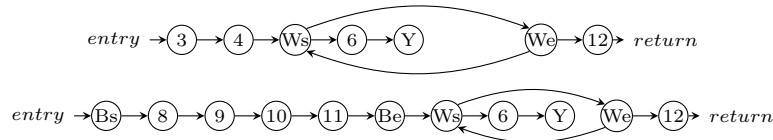
Example. Recall once more the `bucket` coroutine from Listing 5. The resulting control flow graph is shown below. Nodes that do not represent structured control flow or yielding are annotated with the line number from Listing 28, and the Y node represents the yield-site.

³ We sometimes slightly deviate from this in the examples, for better readability.



Control flow graph splitting Resuming a coroutine effectively jumps to the middle of its body. Such a jump is not possible if the target language that supports only structured programming constructs. As explained in Section 5.2, the transformation outputs multiple entry point subroutines, each containing only structured control flow. Therefore, the CFG from the previous phase is split into multiple segments, each corresponding to an entry point.

The splitting starts at the node that corresponds to the coroutine method entry, and traverses the nodes of the graph until reaching a previously unseen yield-site or coroutine call. The search is repeated from each split, taking care not to repeat the search twice. If the graph traversal encounters a control flow node (such as *We*) whose corresponding *Ws* node was not seen as part of the same segment (which can happen if there is a yield inside a while-loop), such a node is converted into a *Be* (block exit) node, followed by a loop again.



Example. The control flow graph of the `bucket` coroutine from Listing 5, produced in the previous phase, is split into the following pair of segments. Note that the second segment starts from the yield-site inside the loop, and that one loop iteration is effectively unrolled.

■ **Listing 29** Entry points of `bucket`.

```

1 def _enter(
2   i: Instance[Int, Unit]
3 ): Unit =
4   i._pstack(i._ptop) match {
5     case 0 => _ep0(i)
6     case 1 => _ep1(i)
7   }
8
9 def _ep0(
10  i: Instance[Int, Unit]
11 ): Unit = {
12   var b = i._rstack(i._rtop + 0)
13   var x_0 = b != Nil
14   var x_1 = x_0
15   while (x_1) {
16     var x_2 = b.head
17     i.value = x_2
18     i._pstack(i._ptop + 0) = 1
19     return
20   }
21   i._result = ()
22   i._cstack(i._ctop) = null
23   i._ctop -= 1
24   i._ptop -= 1 }

```

■ **Listing 30** Entry points of `bucket`, cont.

```

1 def _ep1(
2   i: Instance[Int, Unit]
3 ): Unit = {
4   var b = i._rstack(i._rtop + 0)
5   var x_1 = false
6   {
7     var x_3 = ()
8     var x_4 = b.tail
9     b = x_4
10    var x_0 = b != Nil
11    x_1 = x_0
12  }
13
14  while (x_1) {
15    var x_2 = b.head
16    i.value = x_2
17    i._rstack(i._rtop + 0) = b
18    i._pstack(i._ptop + 0) = 1
19    return
20  }
21  i._result = ()
22  i._cstack(i._ctop) = null
23  i._ctop -= 1
24  i._ptop -= 1 }

```

AST generation. This phase transforms the CFG segments back into the ASTs for the entry point methods. Each entry point starts by restoring the local variables from the call stack. At each yield-site and each coroutine call (commonly, the `exit`), the entry point method stores the local variables back to the stack. It then either stores the yield value into the

coroutine, or stores the result value. The entry point methods are placed into a `Coroutine` subclass, and the `_enter` method dispatches to the proper entry points.

Example. Listings 29 and 30 show the implementation of the entry points of the `bucket` coroutine. Note that each method starts by restoring the local variable `b` from the reference stack of the coroutine instance `i`. Each entry point ends by either storing the yield value into the `value` field of the coroutine instance, or the result value into the `result` field. Local variables are then stored onto the stack (there are two stacks – `_vstack` for primitive values and `_rstack` for references), and the program counter is stored to the `_pstack`.

■ **Listing 31** Error-handling coroutines.

```

1 val fail =
2   coroutine { (e: Error) =>
3     throw e
4   }
5 val forward =
6   coroutine { () =>
7     fail(new Error)
8   }
9 val main =
10  coroutine { () =>
11    try forward()
12    catch {
13      case e: Error =>
14        println("Failed.")
15    }
16  }
```

■ **Listing 32** Normalized `main` coroutine.

```

1 /* main */
2 var x_0: Exception = null
3 try forward()
4 catch {
5   case e => x_0 = e
6 }
7 var x_1 = x_0 != null
8 if (x_1) {
9   var x_2 =
10    x_0.isInstanceOf[Error]
11    if (x_2) {
12      println("Failed.")
13    } else {
14      throw x_0
15    }
16 }
```

5.4 Exception handling

Code inside the `coroutine` block can throw an exception. In this case, standard exception handling semantics apply – the control flow must continue from the nearest dynamically enclosing `catch` block that handles the respective exception type. To ensure this, the transformation does three things:

- (1) it normalizes the `try-catch` and `throw` ASTs,
- (2) it treats each `throw` statement as a suspension point that writes to the instance's `_exception` field,
- (3) it places an exception handler at the beginning of each entry point.

To explain these steps, we use the example in Listing 31. The `fail` coroutine takes an `Error` argument and throws it. The `forward` coroutine creates an `Error` object, and calls the `fail` coroutine without handling its exceptions. The `main` coroutine calls the `forward` coroutine inside a `try` block, and catches the subset of exceptions with the `Error` type.

Normalization. The normalized coroutine `main` is shown in Listing 32. The `catch` handler is transformed so that, once caught, the exception is immediately stored into `x_0`. The variable `x_0` is matched against the `Error` type in the subsequent `if`-statements.

Exception throws. The transformation of the `throw` statement from the `fail` coroutine is in Listing 33. The parameter is loaded into the variable `e`, and immediately stored into the `exception` field. The coroutine stack `_cstack` is then popped, and the coroutine returns.

■ **Listing 33** 1st entry point of fail.

```

1 /* fail, _ep0 */
2 var e = i._rstack(i._rtop)
3 i._exception = e
4 i._rstack(i._rtop) = null
5 i._rtop -= 1
6 return

```

■ **Listing 34** 2nd entry point of main.

```

1 /* main, _ep1 */
2 try {
3     try {
4         var x_0 = i._exception
5         if (x_0 != null) throw x_0
6     } catch { case e => x_0 = e }
7     var x_1 = x_0 != null
8     if (x_1) {
9         var x_2 =
10            x_0.isInstanceOf[Error]
11            if (x_2) println("Failed.")
12            else throw x_0
13     }
14     /* Normal exit */
15 } catch { case x_1 =>
16     /* Exceptional exit */
17 }

```

■ **Listing 35** Entry points of forward.

```

1 /* forward, _ep0 */
2 var x_0 = new Error
3 fail._call(i, x_0)
4 i._ctop += 1
5 i._cstack(i._ctop) = null
6 i._ptop += 1
7 i._pstack(i._ptop) = 0
8 i._call = true
9 return
10
11 /* forward, _ep1 */
12 try {
13     var x_0 = i._exception
14     if (x_0 != null) throw x_0
15     i._cstack(i._ctop) = null
16     i._ctop -= 1
17     i._ptop -= 1
18     return
19 } catch { case x_1 =>
20     i._exception = x_1
21     i._cstack(i._ctop) = null
22     i._ctop -= 1
23     i._ptop -= 1
24     return
25 }
26

```

Stack unwinding. The final rule is to wrap every entry point that starts at a return from a coroutine call into an *unwinding exception handler*. In addition, if the previous entry point ended inside a *user exception handler*, then a replica of that handler is added.

The *forward* coroutine does not have an exception handler, so its second entry point `_ep1` contains only the *unwinding* handler, as shown in Listing 35. On the other hand, *main*'s first entry point `_ep0` ends with a coroutine call. Listing 34 shows that the second entry point `_ep1` therefore has both the *unwinding* handler and the *user* handler. If the user handler cannot handle the exception, then the exception is rethrown.

5.5 Optimizations

An entry point method does not need to load all the local variables at the beginning, nor store all of them to the stack. For example, the entry points of the *bucket* coroutine in Listings 29 and 30 only store a subset of all the variables in the scope. In particular, `_ep0` does not store the variables `b`, `x_0`, `x_1` and `x_2`, while `_ep1` only stores `b`. In this section, we explain the optimization rules used to avoid the unnecessary loads and stores.

Scope rule. A variable does not need to be loaded or stored if it is not in scope after the exit point. This rule applies to, for example, the variables `x_3` and `x_4` from Listing 30.

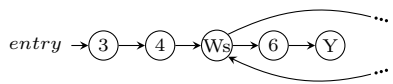
► **Definition 13.** A control flow graph node d dominates a node n if every control flow path from the begin node to n must go through d .

Must-load rule. A local variable v must be loaded from the stack if the respective entry point contains at least one read of v that is not dominated by a write of v .

Example. Consider the variable `b` of the entry point `entry` \rightarrow $\textcircled{\text{Bs}}$ \rightarrow $\textcircled{8}$ \rightarrow $\textcircled{9}$ \rightarrow $\textcircled{10}$ \rightarrow \dots `_ep1` of the `bucket` coroutine in Listing 30. In the corresponding control flow graph, the read in the node 8 precedes the write in the node 9. Consequently, there exists a read that is not dominated by any write, and `b` must be loaded.

► **Definition 14.** A control flow path is a connected sequence of CFG nodes. A control flow path is *v-live* if the variable `v` is in scope in all the nodes of that control flow path.

Was-changed rule. A local variable `v` must not be stored to the stack if there is no *v-live* control flow path that starts with a write to `v` and ends at the respective exit node.



Example. Consider the variable `b` of the entry point `_ep0` of the `bucket` coroutine in Listing 29. In the corresponding CFG, the variable `b` is read in nodes 3 and 6. However, there is no write to `v` that connects to the exit node `Y`. Therefore, at `Y`, `b` did not change its value since the begin node, so it does not have to be stored.

► **Definition 15.** We say that an exit node `x` *resumes* at an entry point `e`, if the exit node corresponds to the begin node of `e` in the original control flow graph.

► **Definition 16.** Relation $needed(x, v, e)$ between an exit node `x`, the variable `v` and an entry point method `e` holds if and only if either:

- (1) `x` resumes at `e`, and the must-load rule applies to `v` and `e`, or
- (2) `x` resumes at `e'`, and there is a *v-live* control flow path between the begin node of `e'` and some exit node `x'` of `e'`, such that $needed(x', v, e)$.

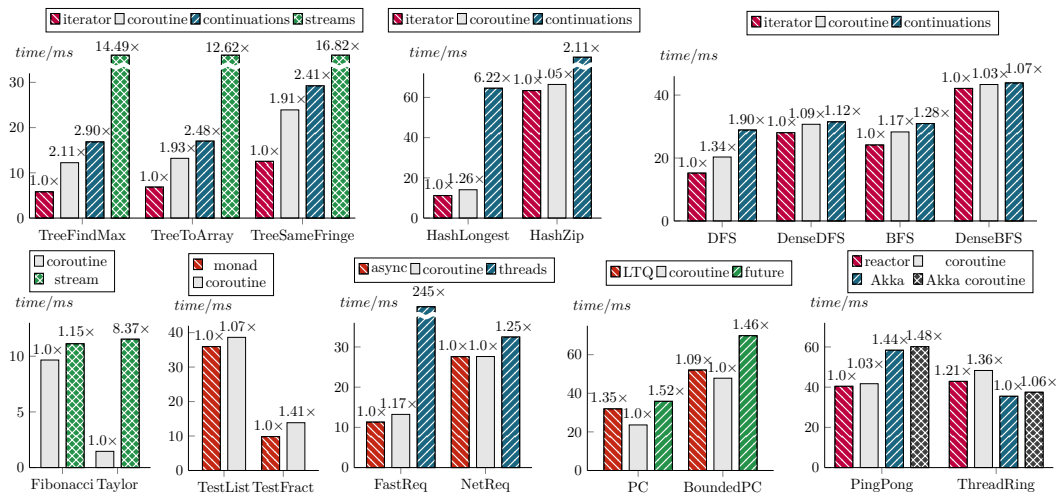
Is-needed rule. A local variable `v` must not be stored to the stack at an exit node `x` if there is no entry point method `e` such that $needed(x, v, e)$.

Example. Intuitively, this rule applies when it becomes impossible to reach (without `v` going out of scope) an entry point that would need to load `v`. This rule applies to the variable `x_2` in the entry point `_ep1` in Listing 30. Variable `x_2` is in scope at the exit point, however, it does not need to be loaded when `_ep1` is reentered, and it goes out of scope before it is needed again.

6 Performance Evaluation

The goal of the evaluation is to assess coroutine performance on a range of different use cases, most notably those from Section 3. The source code of the benchmarks is available online [44]. Evaluation was done in accordance with established JVM benchmarking methodologies [19]. We used the ScalaMeter framework [42] to repeat each benchmark 30 times, across 6 different JVM process instances, and we report the average values. We used a quad-core 2.8 GHz Intel i7-4900MQ processor with 32 GB of RAM, and results are shown in Figure 4.

Iterators. We test tree iterators from Listing 8 against a manually implemented tree iterator. The first benchmark, `TreeFindMax`, traverses a single tree and finds the largest integer, while `TreeToArray` copies the integers to an array. `TreeSameFringe` compares the corresponding integers in two trees with a different layout [18]. These benchmarks heavily modify the stack, so a coroutine is 1.9 – 2.1× slower than an iterator. A CPS-based iterator, built using Scala



■ **Figure 4** Performance of coroutines and alternative frameworks (lower is better).

delimited continuations [56], is 2.4 – 2.9 \times slower. For comparison, a lazy functional stream is 12 – 17 \times slower.

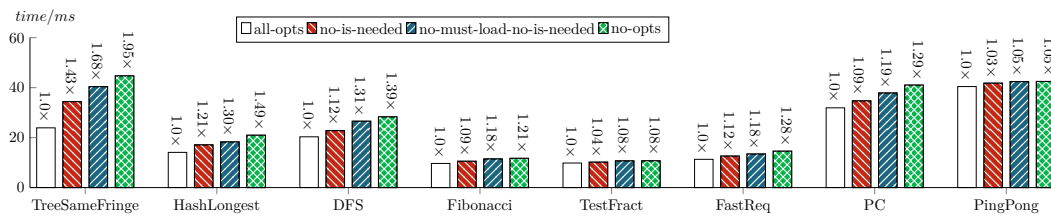
The *HashLongest* benchmark traverses a hash table to find the longest string (strings contain size information, so checks are cheap). Since most of the time is spent in the loop and not in coroutine calls, performance overhead compared to iterators is only 26%. This benchmark reveals a downside of CPS-based continuations⁴. The relative overhead of allocating continuation closures is considerable for hash table iterators, so a continuation-based iterator is 6.22 \times slower. *HashZip* simultaneously traverses two hash tables, picks an element from each pair, and inserts it into a third hash table, effectively implementing a zip operation on the hash tables. Since zipping does additional work of implementing the resulting hash table, coroutines have only a 5% overhead compared to a manually implemented iterator, while continuations are 2.1 \times slower.

DFS and *BFS* benchmarks traverse a sparse graph (degree 3) in depth-first and breadth-first order. Coroutines have an overhead of 34% and 17% compared to manually implemented iterators, and continuations 90% and 28%, respectively. Making the graphs denser (degree 16) amortizes the overhead of suspensions, resulting in overheads of 9% and 3% for coroutines.

Lazy streams. Functional lazy streams, or lazy lists, are a neat abstraction for recursively defining number series. Coroutines are also a good fit for this use-case, but are considerably faster, since a stream needs to create a node object for each number in the series.

The *Fibonacci* benchmarks generates Fibonacci numbers, and uses big integer arithmetic to do so. The overhead of a lazy-stream-based solution compared to a coroutine-based one is only 15%. The *Taylor* benchmark generates a Taylor series, using floating-point arithmetic. The work involved in a floating-point computation is much lower compared to big-integer arithmetic. Since the relative overhead of lazy streams is much more pronounced in this case, the stream-based solution is 8.4 \times slower.

⁴ In some cases, the Scala compiler can eliminate tail-calls, but typically not when invoking lambda objects that encode the continuations produced by Scala’s delimited continuations plugin.



■ **Figure 5** Impact of optimizations on performance (lower is better).

ScalaCheck. The *TestList* and *TestFract* benchmarks compare regular ScalaCheck generator-based testing [36] with backtracking from Section 3. The *TestList* benchmark checks properties of list objects, and is computation-heavy – relative backtracking overhead due to creating snapshots is only 7%. The *TestFract* benchmark checks properties of fractions and does only simple arithmetic – in this case, backtracking overhead is 41%.

Async-Await. Here, network communication is the primary use-case. *FastReq* creates an immediately completed request, and awaits it. In this case, coroutine-based implementation from Section 3 has a 17% overhead. Just for comparison, starting a new thread for each request is 245× slower. In practice, the network introduces a delay between requests and responses. *NetReq* uses a 1 ms delay, in which case coroutines have no observable overhead.

Single-assignment variables. In this benchmark, Oz-style single-assignment variables from Listing 18 are used to implement dataflow streams – a variant of cons-lists with single-assignment tails. This allows a straightforward encoding of the producer-consumer pattern [65]. The *PC* benchmark compares dataflow streams based on Scala Futures [22] with coroutine-based streams. The direct-style coroutine API has an interesting performance impact. Futures are 52% slower because every tail-read must allocate a closure and install a callback even if the value is already present, whereas a coroutine can be directly resumed when a value is available. The *LinkedTransferQueue* from the JDK blocks the thread when waiting for a value, and is 35% slower. *Bounded PC* adds an additional backpressure dataflow stream between the producer and the consumer, and has similar performance ratios.

Actors and event streams. We compare callback-style and direct-style Akka actors [1] and reactors [54, 47, 43, 45] on two benchmarks from the Savina actor benchmark suite [24]. Direct-style programs are encoded by hot-swapping the event loop, as explained in Section 3. The callback allocation in *receive* and *get* calls causes a 3% slowdown for reactors and 2.8% for actors in *PingPong*. In *ThreadRing*, slowdown is 12% for reactors and 6% for actors.

Optimization Breakdown. We show a breakdown of different optimizations from Section 5.5. We pick eight benchmarks from Figure 4 and run them after disabling different optimization combinations. We observe the highest impact on *TreeSameFringe*, *HashLongest*, *DFS* and *PC*. In Figure 5, *all-opts* shows performance with all optimizations enabled, *no-is-needed* disables the is-needed rule, *no-must-load-no-is-needed* additionally disables the must-load rule, and *no-opts* disables all optimizations. Results show that optimizations have the highest impact on *TreeSameFringe*, where disabling them causes a total slowdown of almost 2×. Here, 50% of the performance comes from the is-needed rule. In other benchmarks shown in Figure 5, total improvement from optimizations ranges from 5% to 50%.

■ **Table 1** Comparison of Suspension Primitives in Different Languages.

Name	Type-safe	First-class	Stackful	Allocation-free	Scope	Snapshots
Enumerators (C#)	✓	✗	✗	∅	delimited	✗
Generators (Python)	✗	✓	✓	✗	delimited	✗
Async (Scala, C#)	✓	✓	✓	✗	delimited	✗
Spawn-sync (Cilk)	✓	✗	✓	✓	whole program	✗
Boost (C++)	✓	✓	✓	✓	delimited	✗
CO2 (C++)	✓	✓	✗	∅	delimited	✗
Coroutines (Lua)	✗	✓	✓	✓	just-in-time	✗
Coroutines (Kotlin)	✓	✓	✓	✗	delimited	✗
Coroutines (Scala)	✓	✓	✓	✓	delimited	✓

7 Related Work

We organize the related work on coroutines into several categories. We start with the origins and previous formalization approaches, we then contrast coroutines to similar domain-specific primitives, and conclude with the related work on continuations. Where appropriate, we contrast our model with alternatives. As stated in the introduction, many features of our model have been studied already. However, our main novelty is that our delimited coroutines rely only on metaprogramming, as well as augmenting coroutines with snapshots.

Coroutine in programming languages. Table 1 is a brief comparative summary of suspension primitives in different languages. We compare type-safety, whether suspendable code blocks are first-class objects, whether coroutines are stackful, and if suspendable blocks can call each other without dynamic allocation. The scope column denotes the scope in which the primitive can be used.

The Kotlin language exposes coroutines with its `suspend` and `yield` keywords [7]. Kotlin’s implementation is delimited and CPS-based, and it translates every call to a coroutine `c` to an allocation of a coroutine class specific to `c`. This coroutine-specific class holds the state of the local variables. Instances of this class are chained and form linked-list-based callstack. Our translation approach is different in that a coroutine call modifies an array-based call stack, and does not require an object allocation. Currently, Kotlin coroutines do not allow snapshots, which makes them equivalent to one-shot continuations [34].

In the C++ community, there are two popular coroutine libraries: Boost coroutines [27] and CO2 [25]. Boost coroutines are stackful, and they expose two separate asymmetric coroutine types: push-based coroutines, where resuming takes an input value, and pull-based coroutines, where resuming returns an output value. They do not support snapshots due to problems with memory safety in copying the stack. CO2 aims to implement fast coroutines, and reports better performance than Boost, but it supports only stackless coroutines.

Origins and formalizations. The idea of coroutines dates back to Erdwinn and Conway’s work on a tape-based Cobol compiler and its separability into modules [11]. Although the original use-case is no longer relevant, other use-cases emerged. Coroutines were investigated on numerous occasions, and initially appeared in languages such as Modula-2 [68], Simula [6], and BCPL [33]. A detailed classification of coroutines is given by Moura and Ierusalimschy [34], along with a formalization of asymmetric coroutines through an operational semantics. Moura and Ierusalimschy observed that asymmetric first-class stackful coroutines have an equal expressive power as one-shot continuations, but did not investigate snapshots, which

make coroutines equivalent to full continuations. Anton and Thiemann showed that it is possible to automatically derive type systems for symmetric and asymmetric coroutines by converting their reduction semantics into equivalent functional implementations, and then applying existing type systems for programs with continuations [2]. James and Sabry identified the input and output types of coroutines [57], where the output type corresponds to the *yield* type described in this paper. The input type ascribes the value passed to the coroutine when it is resumed. As a design tradeoff, we chose not to have explicit input values in our model. First, the input type increases the verbosity of the coroutine type, which may have practical consequences. Second, as shown in examples from Section 3, the input type can be simulated with the return type of another coroutine, which yields a writable location, and returns its value when resumed (e.g. the `await` coroutine from Section 3). Fischer et al. proposed a coroutine-based programming model for the Java programming language, along with the respective formal extension of Featherweight Java [17].

Domain-specific approaches. The need for simpler control flow prompted the introduction of coroutine-inspired primitives that target specific domains. One of the early applications was data structure traversal. Push-style traversal with `foreach` is easy, but the caller must relinquish control, and many applications cannot do this (e.g. the same-fringe benchmark from Section 6). Java-style iterators with `next` and `hasNext` are harder to implement than a `foreach` method, and coroutines bridge this gap.

Iterators in CLU [29] are essentially coroutines – program sections with `yield` statements that are converted into traversal objects. C# inherited this approach – its iterator type `IEnumerator` exposes `Current` and `MoveNext` methods. Since enumerator methods are not first class entities, it is somewhat harder to abstract suspendable code, as in the backtracking example from Section 3. C# enumerators are not stackful, so the closed addressing hash table example from the Listing 6 must be implemented inside a single method. Enumerators can be used for asynchronous programming, but they require exposing `yield` in user code. Therefore, separately from enumerators, C# exposes `async-await` primitives. Some newer languages such as Dart similarly expose an `async-await` pair of primitives.

Async-Await in Scala [23] is implemented using Scala’s metaprogramming facilities. Async-await programs can compose by expressing asynchronous components as first-class `Future` objects. The Async-Await model does not need to be stackful, since separate modules can be expressed as separate futures. However, reliance on futures and concurrency makes it hard to use Async-Await generically. For example, iterators implemented using futures have considerable performance overheads due to synchronization involved in creating future values.

There exist other domain-specific suspension models. For example, Erlang’s `receive` statement effectively captures the program continuation when awaiting for the inbound message [66]. A model similar to Scala Async was devised to generate Rx’s `Observable` values [20, 32], and the event stream composition in the reactor model [46, 49], as well as callbacks usages in asynchronous programming models based on futures and flow-pools [22, 53, 52, 41, 59] can be similarly simplified. Cilk’s `spawn-sync` model [28] is similar to `async-await`, and it is implemented as a full program transformation. The Esterel language defines a `pause` statement that pauses the execution, and continues it in the next event propagation cycle [5]. Behaviour trees [31] are AI algorithms used to simulate agents – they essentially behave as AST interpreters with `yield` statements.

Generators. Dynamic languages often support *generators*, which are essentially untyped asymmetric coroutines. A Python generator instance [58] exposes only the `next` method (an

equivalent of `resume`), which throws a `StopIteration` error when it completes. In practice, Python generators are mostly used for *list comprehensions*, as programmers find it verbose to handle `StopIteration` errors. Newer Python versions allow stackful generators [15] with the `yield from` statement, which is implemented as syntactic sugar around basic generators that chains the `resume` points instead of using call stacks. ECMAScript 6 generators are similar to Python’s generators. Lua coroutines bear the most similarity with the coroutine formulation in this paper [13], with several differences. First, Lua coroutines are not statically typed. While this is less safe, it has the advantage of reduced syntactic burden. Second, Lua coroutines are created from function values dynamically. This is convenient, but requires additional JIT optimizations to be efficient.

Transformation-based continuations. Continuations are closely related to coroutines, and with the addition of `snapshot` the two can express the same programs. Scheme supports programming with continuations via the `call/cc` operator, which has a similar role as `shift` in shift-reset delimited continuations [12, 3]. In several different contexts, it was shown that continuations subsume other control constructs such as exception handling, backtracking, and coroutines. Nonetheless, most programming languages do not support continuations today. It is somewhat difficult to provide an efficient implementation of continuations, since the captured continuations must be callable more than once. One approach to continuations is to transform the program to continuation-passing style [64]. Scala’s continuations [56] implement delimited shift-reset continuations with a CPS transform. One downside of the continuation-passing style transformation is the risk of stack overflows when the tail-call optimization is absent from the runtime, as is the case of JVM.

Optimizing compilers tend to be tailored to the workloads that appear in practice. For example, it was shown that optimizations such as inlining, escape analysis, loop unrolling and devirtualization make most collection programs run nearly optimally [50, 40, 55, 63, 48]. However, abstraction overheads associated with coroutines are somewhat new, and are not addressed by most compilers. For this reason, compile-time transformations of coroutine-heavy workloads typically produce slower programs compared to their runtime-based counterparts. We postulate that targeted high-level JIT optimizations could significantly narrow this gap.

Runtime-based continuations. There were several attempts to provide runtime continuation support for the JVM, ranging from Ovm implementations [14] based on `call/cc`, to JVM extensions [62], based on the `capture` and `resume` primitives. While runtime continuations are not delimited and can be made very efficient, maintenance pressure and portability requirements prevented these implementations from becoming a part of official JVM releases. An alternative, less demanding approach relies only on stack introspection facilities of the host runtime [38]. There exists a program transformation that relies on exception-handling to capture the stack [60]. Here, before calling the continuation, the saved state is used in method calls to rebuild the stack. This works well for continuations, where the stack must be copied anyway, but may be too costly for coroutine `resume`. Bruggeman et al. observed that many use cases call the continuation only once and can avoid the copying overhead, which lead to *one-shot continuations* [8]. One-shot continuations are akin to coroutines without snapshots.

Other related constructs. Coroutines are sometimes confused with *goroutines*, which are lightweight threads in the Go language. While coroutines can be used to encode goroutines, the converse encoding is not as efficient, as goroutines involve message passing.

8 Conclusion

We described a programming model for first-class typed stackful coroutines with snapshots, along with a formalization. Our implementation relies on metaprogramming facilities of the host language. We identified the critical optimizations that need to accompany the implementation, and showed their performance impact. We identified a range of use cases such as iterators, Async-Await, Oz-style dataflow variables, Erlang-style actors, backtracking, and direct-style event streams, and we showed that they can be expressed in our model. Experimental evaluation shows that our coroutine implementation is almost as efficient as these other primitives, and in some cases has an even better performance.

Our implementation is available online [44], as an independent module that relies on metaprogramming capabilities in Scala, and works with the official language releases. This work may indicate a wider need for metaprogramming support in general purpose languages, which may be easier to provide than continuation support in the runtime. Moreover, runtime support and JIT optimizations [50] could improve the performance of our implementation even further, and we plan to investigate this in the future.

References

- 1 Akka. Akka documentation, 2011. <http://akka.io/docs/>.
- 2 Konrad Anton and Peter Thiemann. *Towards Deriving Type Systems and Implementations for Coroutines*, pages 63–79. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. doi:10.1007/978-3-642-17164-2_6.
- 3 Kenichi Asai and Chihiro Uehara. Selective cps transformation for shift and reset. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '18, pages 40–52, New York, NY, USA, 2018. ACM. doi:10.1145/3162069.
- 4 V. Beltran, D. Carrera, J. Torres, and E. Ayguade. Evaluating the scalability of java event-driven web servers. In *International Conference on Parallel Processing, 2004. ICPP 2004.*, pages 134–142 vol.1, Aug 2004. doi:10.1109/ICPP.2004.1327913.
- 5 Gérard Berry and Georges Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992. doi:10.1016/0167-6423(92)90005-V.
- 6 G.M. Birtwhistle, O.J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Chartwell-Bratt Ltd, 1979.
- 7 Andrey Breslav. Coroutines for kotlin (revision 3.2), 2017. <https://github.com/Kotlin/kotlin-coroutines/blob/master/kotlin-coroutines-informal.md>.
- 8 Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 99–107, New York, NY, USA, 1996. ACM. doi:10.1145/231379.231395.
- 9 Eugene Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 3:1–3:10, New York, NY, USA, 2013. ACM. doi:10.1145/2489837.2489840.
- 10 Mike Cantelon, Marc Harter, TJ Holowaychuk, and Nathan Rajlich. *Node.js in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2013.
- 11 Melvin E. Conway. Design of a Separable Transition-Diagram Compiler. *Commun. ACM*, 6(7):396–408, 1963. doi:10.1145/366663.366704.

- 12 Olivier Danvy and Andrzej Filinski. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 151–160, New York, NY, USA, 1990. ACM. doi:10.1145/91556.91622.
- 13 A. Lúcia de Moura, N. Rodriguez, and R. Ierusalimsky. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, July 2004.
- 14 Iulian Dragos, Antonio Cuneo, and Jan Vitek. Continuations in the Java Virtual Machine. In *Second ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2007)*, Berlin, 2007. Technische Universität Berlin.
- 15 Gregory Ewing. PEP 380 - Syntax for Delegating to a Subgenerator, 2009. <https://www.python.org/dev/peps/pep-0380/>.
- 16 Matthias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 180–190, New York, NY, USA, 1988. ACM. doi:10.1145/73560.73576.
- 17 Jeffrey Fischer, Rupak Majumdar, and Todd Millstein. Tasks: Language support for event-driven programming. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '07, pages 134–143, New York, NY, USA, 2007. ACM. doi:10.1145/1244381.1244403.
- 18 Richard P. Gabriel. The Design of Parallel Programming Languages. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*, pages 91–108. Academic Press Professional, Inc., San Diego, CA, USA, 1991. URL: <http://dl.acm.org/citation.cfm?id=132218.132225>.
- 19 Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM. doi:10.1145/1297027.1297033.
- 20 Philipp Haller and Heather Miller. RAY: Integrating Rx and Async for Direct-Style Reactive Streams. In *Workshop on Reactivity, Events and Modularity*, 2013.
- 21 Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, feb 2009. doi:10.1016/j.tcs.2008.09.019.
- 22 Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. Scala improvement proposal: Futures and promises. In *SIP-14*, 2012. URL: <http://docs.scala-lang.org/sips/pending/futures-promises.html>.
- 23 Philipp Haller and Jason Zaugg. Scala Async Repository, 2013. <https://github.com/scala/async>.
- 24 Shams M. Imam and Vivek Sarkar. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, pages 67–80, New York, NY, USA, 2014. ACM. doi:10.1145/2687357.2687368.
- 25 Jamboree. Co2: A c++ await/yield emulation library for stackless coroutine, 2017. URL: <https://github.com/jamboree/co2>.
- 26 Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988. URL: <http://www.laputan.org/drc.html>.
- 27 Oliver Kowalke. Coroutine2, 2017. URL: http://www.boost.org/doc/libs/1_66_0/libs/coroutine2.
- 28 Charles E. Leiserson. *Programming irregular parallel applications in Cilk*, pages 61–71. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. doi:10.1007/3-540-63138-0_6.
- 29 Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction Mechanisms in CLU. *Commun. ACM*, 20(8):564–576, aug 1977. doi:10.1145/359763.359789.

- 30 Ingo Maier and Martin Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, EPFL, 2012.
- 31 A. Marzotto, M. Colledanchise, C. Smith, and P. Ögren. Towards a Unified Behavior Trees Framework for Robot Control. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5420–5427, May 2014. doi:10.1109/ICRA.2014.6907656.
- 32 Erik Meijer. Your Mouse is a Database. *Commun. ACM*, 55(5):66–73, may 2012. doi:10.1145/2160718.2160735.
- 33 Ken Moody and Martin Richards. A coroutine mechanism for bcpl. *Softw., Pract. Exper.*, 10(10):765–771, 1980. doi:10.1002/spe.4380101002.
- 34 Ana Lúcia De Moura and Roberto Ierusalimsky. Revisiting Coroutines. *ACM Trans. Program. Lang. Syst.*, 31(2):6:1–6:31, 2009. doi:10.1145/1462166.1462167.
- 35 Lasse R. Nielsen and BRICS. A selective cps transformation. *Electronic Notes in Theoretical Computer Science*, 45:311–331, 2001. MFPS 2001, Seventeenth Conference on the Mathematical Foundations of Programming Semantics. doi:10.1016/S1571-0661(04)80969-1.
- 36 Rickard Nilsson. ScalaCheck Website, 2010. <https://www.scalacheck.org/>.
- 37 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language. Technical report, EPFL, 2004.
- 38 Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from Generalized Stack Inspection. *SIGPLAN Not.*, 40(9):216–227, sep 2005. doi:10.1145/1090189.1086393.
- 39 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.
- 40 A. Prokopec, D. Petrashko, and M. Odersky. Efficient lock-free work-stealing iterators for data-parallel collections. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 248–252, March 2015. doi:10.1109/PDP.2015.65.
- 41 Aleksandar Prokopec. *Data Structures and Algorithms for Data-Parallel Computing in a Managed Runtime*. PhD thesis, IC, Lausanne, 2014. doi:10.5075/epfl-thesis-6264.
- 42 Aleksandar Prokopec. Scalometer website, 2014. URL: <http://scalometer.github.io>.
- 43 Aleksandar Prokopec. Pluggable scheduling for the reactor programming model. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2016, pages 41–50, New York, NY, USA, 2016. ACM. doi:10.1145/3001886.3001891.
- 44 Aleksandar Prokopec. Scala Coroutines Website, 2016. <https://storm-enroute/coroutines>.
- 45 Aleksandar Prokopec. Accelerating by idling: How speculative delays improve performance of message-oriented systems. In Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro, editors, *Euro-Par 2017: Parallel Processing*, pages 177–191, Cham, 2017. Springer International Publishing.
- 46 Aleksandar Prokopec. Encoding the building blocks of communication. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, pages 104–118, New York, NY, USA, 2017. ACM. doi:10.1145/3133850.3133865.
- 47 Aleksandar Prokopec. Reactors.io website, 2018. URL: <http://reactors.io>.
- 48 Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. A generic parallel collection framework. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par’11, pages 136–147, Berlin, Heidelberg, 2011. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=2033408.2033425>.
- 49 Aleksandar Prokopec, Philipp Haller, and Martin Odersky. Containers and aggregates, mutators and isolates for reactive programming. In *Proceedings of the Fifth Annual Scala*

- Workshop*, SCALA '14, pages 51–61, New York, NY, USA, 2014. ACM. doi:10.1145/2637647.2637656.
- 50 Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger. Making collection operations optimal with aggressive JIT compilation. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, SCALA 2017, pages 29–40, New York, NY, USA, 2017. ACM. doi:10.1145/3136000.3136002.
 - 51 Aleksandar Prokopec and Fengyun Liu. On the soundness of coroutines with snapshots. *CoRR*, abs/1806.01405, 2018. arXiv:1806.01405.
 - 52 Aleksandar Prokopec, Heather Miller, Philipp Haller, Tobias Schlatter, and Martin Odersky. FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction, Proofs. Technical report, EPFL, 2012.
 - 53 Aleksandar Prokopec, Heather Miller, Tobias Schlatter, Philipp Haller, and Martin Odersky. Flowpools: A lock-free deterministic concurrent dataflow abstraction. In *LCPC*, pages 158–173, 2012. doi:10.1007/978-3-642-37658-0_11.
 - 54 Aleksandar Prokopec and Martin Odersky. Isolates, Channels, and Event Streams for Composable Distributed Programming. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 171–182, New York, NY, USA, 2015. ACM. doi:10.1145/2814228.2814245.
 - 55 Aleksandar Prokopec, Dmitry Petrashko, and Martin Odersky. On lock-free work-stealing iterators for parallel data structures. Technical report, EPFL, 2014.
 - 56 Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform. *SIGPLAN Not.*, 44(9):317–328, 2009. doi:10.1145/1631687.1596596.
 - 57 P. James Roshan and Amr Sabry. Yield: Mainstream delimited continuations, 2011.
 - 58 Neil Schemenauer, Tim Peters, and Magnus Hetland. PEP 255 - Simple Generators, 2001. <https://www.python.org/dev/peps/pep-0255/>.
 - 59 Tobias Schlatter, Aleksandar Prokopec, Heather Miller, Philipp Haller, and Martin Odersky. Multi-lane flowpools: A detailed look. *Tech Report*, 2012.
 - 60 Tatsuro Sekiguchi, Takahiro Sakamoto, and Akinori Yonezawa. Portable implementation of continuation operators in imperative languages by exception handling. In *Advances in Exception Handling Techniques (the Book Grows out of a ECOOP 2000 Workshop)*, pages 217–233, London, UK, UK, 2001. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=647332.722736>.
 - 61 Denys Shabalin, Eugene Burmako, and Martin Odersky. Quasiquotes for Scala. Technical report, EPFL, 2013.
 - 62 Lukas Stadler, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck, and John Rose. Lazy continuations for java virtual machines. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 143–152, New York, NY, USA, 2009. ACM. doi:10.1145/1596655.1596679.
 - 63 Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksandar Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 52–78, Berlin, Heidelberg, 2013. Springer-Verlag. doi:10.1007/978-3-642-39038-8_3.
 - 64 Gerald Jay Sussman and Guy L. Steele, Jr. Scheme: A interpreter for extended lambda calculus. *Higher Order Symbol. Comput.*, 11(4):405–439, 1998. doi:10.1023/A:1010035624696.
 - 65 Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 1st edition, 2004.

- 66 Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ER-LANG (2nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.
- 67 Philip Wadler. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK, 1995. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=647698.734146>.
- 68 N. Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, 1985. URL: <https://books.google.ch/books?id=ZVaRXPrD1AoC>.

A Additional coroutine-based implementations

In Section 3, we showed how coroutines simplify actors and Rx streams. In this section, we show the coroutine-based implementations for these use-cases.

Actors. Since JVM does not have continuations, actor frameworks are unable to implement exact Erlang-style semantics, in which the `receive` statement can be called anywhere in the actor. Instead, frameworks like Akka expose a top-level `receive` method [1], which returns a partial function used to handle messages. This function can be swapped during the actor lifetime with the `become` statement. In the example from Listing 15, we used a `receive` statement that suspends in the middle of the actor and awaits a message. We now show its implementation using coroutines.

The core idea is to implement a `recv` coroutine (we call it `recv` to disambiguate from Akka’s `receive` function), which yields a partial function that represents the continuation of the actor. The resume-site can then call `become` to hot-swap Akka’s top-level `receive` handler with the yielded partial function.

In Listing 36, we first define an auxiliary type `Rec`, which describes a partial function that can take `Any` message objects. The method `act` declares an Erlang-style actor – it takes a coroutine that may yield partial functions of the `Rec` type, which describe how to process the next message. The `act` method starts an instance of the input coroutine, and resumes it inside a recursive `loop` function. The instance potentially calls the `recv` method, which yields. When this happens, `act` extends the yielded partial function with the `andThen` combinator which recursively resumes the coroutine instance. This chained partial function is passed to `become`, which tells Akka to run the chained function when a message arrives.

The `recv` is a coroutine that yields the `Rec` function and returns a message of type `Any` – the actor definition must then match this value. The implementation of `recv` declares a local variable `res` in which the incoming message is stored by the yielded partial function. After `recv` gets resumed, `act` will have already called the yielded function, which will have written the message to `res`, so that it can be returned to the actor that invoked `recv`.

Listing 36 Erlang-style actor implementation.

```

1 type Rec = PartialFunction[Any, Unit]
2 def act(c: () ~> (Rec, Unit)) = Actor { self =>
3   val i = c.start()
4   def loop() =
5     if (i.resume) self.become(i.value.andThen(loop))
6     else self.stop()
7   loop()
8 }
9 val recv: () ~> (Rec, Any) = coroutine { () =>
10  var res: Any = _
11  yieldval({ case x => res = x })
12  res
13 }

```

Event streams. Event streams expose the `onEvent` method, similar to `onSuccess` on futures. The `onEvent` method takes a callback that is invoked when the next event arrives. As shown in Listing 17, it is much more convenient to extract an event in the direct-style by invoking a `get` statement, instead of installing a callback.

In Listing 37, we implement the method `get` on the event stream of type `Events[T]`. We declare a type alias `Install` that represents a function that installs a callback to the event stream. When an `Install` function is invoked with a function `f`, the function `f` is passed as a callback to some event stream.

The `react` method is similar to the `act` method for actors – it delimits the suspendable part of the event-driven program. The `react` method starts and resumes the coroutine. When the coroutine yields an `Install` function, the `react` method uses the `Install` function to install a callback that recursively resumes the coroutine.

The `get` method is called by the users to extract the next event out of a reactor's event stream. Its implementation yields an `Install` function that installs the callback on the event stream by calling `onEvent`. The event stream callback sets the result variable and invokes the continuation function `f`. This technique is similar to the actor use-case, but the difference is that it abstracts over what `become` is.

■ **Listing 37** Direct-style event streams.

```

1 type Install = (() => Unit) => Unit
2 def react(c: () ~> (Install, Unit)) = {
3   val i = c.start()
4   def loop(): Unit = if (i.resume) i.value(loop)
5   loop()
6 }
7 def get[T](e: Events[T]): () ~> (Install, T) = coroutine { () =>
8   var res: T = _
9   val install = (f: () => Unit) => e.onEvent(x => {
10    res = x
11    f()
12  })
13   yieldval(install)
14   res
15 }

```

We note that, in this example, it might have been more natural to yield an event stream directly, instead of yielding `Install` functions. However, the event stream is parametric in the type of events, and the coroutine would always have to yield event streams of the same event type. The `Install` function hides the event type inside the `get` function, and allows a more flexible event stream API.