Dsl.scala: creating library-defined keywords from ad-hoc polymorphic delimited continuations

ANONYMOUS AUTHOR(S)

The semantics of a programming language are defined by its keywords, which traditionally require special treatment by the compiler. Thus, the capacity of the programming language is not extensible by libraries, unless using compiler plug-ins, AST macros, code generation, or other metaprogramming technologies.

Our goal is making the control flow of a programming language extensible. We discovered a novel approach to create library-defined keywords used in control flow. Those keywords are interpreted by libraries instead of the compiler. No metaprogramming knowledge is required for keyword authors.

Library-defined keywords are collaborative. An application developer can create a single function that contains interleaved keywords from different libraries, along with ordinary language control flows.

Additional Key Words and Phrases: type class, scala

1 INTRODUCTION

Traditionally, the capacity of a general purpose language can be extended to special domain by creating an embedded DSL (Domain-Specific Language). For example, Akka provides a DSL to create finite-state machines [Lightbend, Inc. 2017], which consists of some domain-specific operators including `when`, `goto`, `stay`, etc. Although those operators looks similar to native control flow, they are not embeddable in native **`if`**, **`while`** or **`try`** blocks, because the DSL code is split into small closures, preventing ordinary control flow from crossing the boundary of those closures. Thus, this kind of DSLs reinvent incompatible control flow to the meta-languages. TensorFlow's control flow operations [Abadi et al. 2016] and Caolan's async library [McMahon 2017] are other examples of reinventing control flow in eDSLs.

Instead of reinventing the whole set of control flow for each DSL, a more general approach is implementing a minimal interface for control flow for each domain, while other control flow operations are derived from the interface, shared between different domains. In Haskell and other functional programming language, monads are used as the generic interface of control flow [Jones and Duponcheel 1993; Wadler 1990, 1992]. Scala implementations of monads are provided by Scalaz [Yoshida et al. 2017], Cats [Typelevel 2017], Monix [Nedelcu et al. 2017] and Algebird [Twitter, Inc. 2016]. A DSL author only have to implement `bind` and `point` functions in `Monad` type class, and all the derived control flow operations like `whileM` or `ifM` are available. In addition, those monadic data type can be created and composed from **`do`** notation [Jones et al. 1998] or **`for`** comprehension [Odersky et al. 2004]. For example, you can use the same `scalaz.syntax` or **`for`** comprehension to create random value generators [Nilsson 2015] and data-binding expressions [Yang 2016], as long as there are `Monad` instances for data types `org.scalacheck.Gen` and `com.thoughtworks.binding.Binding` respectively.

An idea to avoid incompatible domain-specific control flow is converting direct style control flow to domain-specific control flow at compiler time. For example, Scala Async provides a macro to generate asynchronous control flow [Haller and Zaugg 2013], allowing normal sequential code inside a `scala.async` block to run asynchronously. This approach can be generalized to any monadic data types. ThoughtWorks Each [Yang 2015], Monadless [Brasil 2017], effectful [Crockett 2013] and !-notation in Idris [Brady 2013a] are compiler-time transformers to convert source code of direct style control flow to monadic control flow. For example, with the help of ThoughtWorks

2018. XXXX-XXXX/2018/4-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Each, Binding.scala[Yang 2016] can be used to create reactive HTML template from ordinary direct style code.

Another generic interface of control flow is continuation, which is known as the mother of all monads [Piponi 2008], where specific control flow in specific domain can be supported by specific answer types of continuations. Scala Continuations [Rompf et al. 2009] and Stateless Future [Yang 2014a] are two delimited continuation implementations in Scala. Both projects can convert direct style control flow to continuation-passing style closure chains at compiler time. For example, Stateless Future Akka [Yang 2014b], based on Stateless Future, provides a special answer type for akka actors. Unlike reinvented control flow in `akka.actor.AbstractFSM`, users can create complex finite-state machines from simple direct style control flow along with Stateless Future Akka's domain-specific operator `nextMessage`.

All the previous approaches lack of the ability to collaborate with other DSLs. Each of the above DSLs can be exclusively enabled in a code block. Scala Continuations enables calls to `@cps` method in `reset` blocks, and ThoughtWorks Each enables the magic `each` method [Yang 2015] for `scalaz.Monad` in `monadic` blocks. It was impossible to enable both DSL in one function.

[Kiselyov et al. 2013] introduced effect handlers to solve the collaboration problem. However, the solution is heavy weight, only expressions written in the special `Eff` language are able to use DSLs defined in effect handlers.

This paper describes a lighter-weight approach to resolve the collaboration problem, and presents an implementation in Scala, the framework *Dsl.scala*.

Dsl.scala allows library authors to create special keywords for language features that were usually implemented by the compiler. Those library-defined keywords (LDKs) are adaptive to the enclosing DSL, as a library user can create one function that contains interleaved LDKs from different vendors, along with ordinary Scala control flow. Unlike `Eff`, an LDK is non-intrusive, can be added into an existing function as an optional first-class feature.

Dsl.scala ships with some built-in LDKs, including:

- The `Shift` LDK for asynchronous programming, similar to the `await` and `async` keywords in C#, Python and JavaScript.
- The `Yield` LDK for generating lazy streams, similar to the `yield` keyword in C#, Python and JavaScript.
- The `Each` LDK for traversing each element of a collection, similar to **`for`**, **`yield`** keywords for Scala collections.
- The `Fork` LDK for duplicating current thread, similar to the `fork` system call in POSIX.
- The `AutoClose` LDK to automatically close resources when exiting a scope, similar to the destructor feature in C++.
- The `Monadic` LDK for creating Scalaz [Yoshida et al. 2017] or Cats [Typelevel 2017] monadic control flow, similar to the !-notation in Idris[Brady 2013a].

2 USING LIBRARY-DEFINED KEYWORDS

In this section, we will show some use cases from the perspective of the user of LDKs.

2.1 Creating generators

Suppose Alice is creating an Xorshift pseudo-random number generator [Marsaglia et al. 2003], and she wants to store the generated numbers in a lazily evaluated infinite stream.

The usage of Alice's pseudo-random number generator is shown as below:

```scala
val generatedNumbers = aliceRandomGenerator(seed = 2463534242)
println(generatedNumbers(0))
```

```
println(generatedNumbers(1))
println(generatedNumbers(2))
```

Listing 1. Using Alice's pseudo-random number generator

Alice is a functional programming language developer. She wants to avoid mutable variables in the implementation. Unfortunately, a pseudo-random number generator usually has an internal state that are changed during generate new random number.

With the help of the built-in LDK `Yield` from *Dsl.scala*, Alice can implement the generator as a recursive function that produce the next random number in each iteration.

```
import dsl.keywords.Yield
def aliceRandomGenerator(seed: Int): Stream[Int] = {
  val tmp1 = seed ^ (seed << 13)
  val tmp2 = tmp1 ^ (tmp1 >>> 17)
  val tmp3 = tmp2 ^ (tmp2 << 5)
  !Yield(tmp3)
  aliceRandomGenerator(tmp3)
}
```

Listing 2. The implementation of Alice's pseudo-random number generator

`aliceRandomGenerator` does not throw a `StackOverflowError`, because the execution of `aliceRandomGenerator` will be paused at the LDK `Yield`, and it will be resumed when the caller is looking for the next number.

`Yield` is an LDK to produce a value for a lazily evaluated `Stream`, similar to the `yield` keyword in C#, JavaScript or Python. That is to say, `Stream` is the domain where the domain-specific LDK `Yield` can be used. More generally, all LDKs are domain-specific, where the word "domain" stands for the return type of the enclosing function.

2.2 Creating generators with an additional return value

In this use case, we will demonstrate how to add logging to existing functions using the `Yield` LDK.

Suppose Bob has a function to parse JSON text. The parser is fault-tolerant, since it returns the `defaultValue` for invalid input (Listing 3).

```
import scala.util.parsing.json._
def bobParser(jsonContent: String, defaultValue: JSONType): JSONType = {
  JSON.parseRaw(jsonContent) match {
    case Some(json) =>
      callback(json)
    case None =>
      callback(defaultValue)
  }
}
```

Listing 3. The original implementation of Bob's parser

Then, Bob wants to add some logs to his existing parser. He learned from Alice's use case, and wonders if he can `Yield` log messages to a `Stream[String]` during parsing.

However, unlike Alice's case, Bob's parser should return both the parsed JSON objects and the collected logs. It's impossible in C#'s `yield`, because `yield` does not work in a method that returns a JSON object.

Bob resolves the problem by creating a delimited continuation. The parsed JSON object is handled by a callback function instead of return value. Thus the return value is still a `Stream`, allowing `Yield`ing log messages (Listing 4).

```
import dsl.Dsl.!!
def bobLoggingParser(jsonContent: String, defaultValue: JSONType): Stream[
    String] !! JSONType = { (callback: JSONType => Stream[String]) =>
  !Yield(s"I␣am␣going␣to␣parse␣the␣JSON␣text␣$jsonContent...")
  JSON.parseRaw(jsonContent) match {
    case Some(json) =>
      !Yield(s"Succeeded␣to␣parse␣$jsonContent")
      callback(json)
    case None =>
      !Yield(s"Failed␣to␣parse␣$jsonContent")
      callback(defaultValue)
  }
}
```

Listing 4. The implementation of Bob's logging parser

The return type of Bob's new parser is (`Stream[String] !! JSONType`), which is an alias to the delimited continuation (`(JSONType => Stream[String]) => Stream[String]`), indicating it produces both a `scala.util.parsing.json.JSONType` and a `Stream` of logs.

After Bob created the first version of delimited continuation, he then found that the closure can be simplified with the help of Scala's placeholder syntax (Listing 5).

```
def bobLoggingParserUnderscore(jsonContent: String, defaultValue: JSONType):
    Stream[String] !! JSONType = _ {
  !Yield(s"I␣am␣going␣to␣parse␣the␣JSON␣text␣$jsonContent...")
  JSON.parseRaw(jsonContent) match {
    case Some(json) =>
      !Yield(s"Succeeded␣to␣parse␣$jsonContent")
      json
    case None =>
      !Yield(s"Failed␣to␣parse␣$jsonContent")
      defaultValue
  }
}
```

Listing 5. The implementation of Bob's logging parser, the underscore placeholder version

Alternately, Bob can use the pre-defined function `reset` instead of the underscore placeholder (Listing 6).

```
def reset[R, A](a: => A): R !! A = _(a)

def bobLoggingParserReset(jsonContent: String, defaultValue: JSONType): Stream[
    String] !! JSONType = reset {
```

```
  !Yield(s"I␣am␣going␣to␣parse␣the␣JSON␣text␣$jsonContent...")
  JSON.parseRaw(jsonContent) match {
    case Some(json) =>
      !Yield(s"Succeeded␣to␣parse␣$jsonContent")
      json
    case None =>
      !Yield(s"Failed␣to␣parse␣$jsonContent")
      defaultValue
  }
}
```

Listing 6. The implementation of Bob's logging parser, the reset version

Then, the user of Bob's parser calls `bobLoggingParserReset` to handle both results (Listing 7):

(1) The JSON result in a callback function.
(2) The logs from return value.

```
val logs = bobLoggingParserReset("""{"key":"value"}""", JSONArray(Nil)) { json
    =>
  json should be(JSONObject(Map("key" -> "value")))
  Stream("done")
}
logs should be(
  Stream(
    "I␣am␣going␣to␣parse␣the␣JSON␣text␣{\"key\":\"value\"}...",
    "Succeeded␣to␣parse␣{\"key\":\"value\"}",
    "done"
  )
)
```

Listing 7. Using Bob's parser

The use case of Bob's parser demonstrates how to enable additional LDKs into existing ordinary functions. Generally, an LDK user can introduce a new domain that supports more LDKs to an existing method with the two changes:

(1) Inserting a `NewDomain!!` prefix to the return type (`Stream[String]!!` in Bob's case).
(2) Inserting an underscore or a `reset` before the method body.

2.3 Using multiple LDKs at once

In this use case, we will demonstrate how to use multiple library-defined keywords in one function.

Suppose Carol is creating a line splitter from a file. Carol wants to lazily read each line of a file to a `Stream`, and automatically close the file handle after reading the last line, and finally return the total number of lines.

Carol will use the `Yield` LDK to append a line to the `Stream`, and the `AutoClose` LDK to manage the life-cycle of the file handle.

`Yield` LDK is only available in a function that returns `Stream` or `(Stream[_] !! _)` as we already knows. Similarly, the `AutoClose` LDK is only available in a function that returns `(_ !! Throwable !! _)`. So, Carol makes her line splitter return `(Stream[String] !! Throwable !! Int)` to enable both LDKs (Listing 8).

```scala
import dsl.Dsl.!!
import dsl.keywords.AutoClose
import dsl.keywords.Yield
import dsl.keywords.Shift
import java.nio.file._, Files._

def carolLineSplitter(path: Path): Stream[String] !! Throwable !! Int = reset {
  val reader = !AutoClose(newBufferedReader(path))

  def loop(lineNumber: Int): Stream[String] !! Throwable !! Int = _ {
    reader.readLine() match {
      case null =>
        lineNumber
      case line =>
        !Yield(line)
        !Shift(loop(lineNumber + 1))
    }
  }

  !loop(0)
}
```

Listing 8. Carol's line splitter

Note that the return type of `loop`, `(Stream[String] !! Throwable !! Int)`, is a delimited continuation, Carol needs `Shift` LDK as the first-class delimited continuation operator[Asai 2009; Danvy and Filinski 1990] to invoke `loop` recursively.

The type `(Stream[String] !! Throwable !! Int)` returned from `carolLineSplitter` contains the following data:

- A `Stream` of each lines of the file, as the final return value.
- An optional `Throwable` of the exception thrown during reading the file, which can be handled by a callback function.
- An `Int` of the total number of lines in the file, which can be handled by another callback function.

The example code of using Carol's line splitter is shown in Listing 9.

```scala
val allLines: Stream[String] = carolLineSplitter(Paths.get("multiline.txt")) {
    numberOfLines: Int =>
  println(s"There␣are␣${numberOfLines}␣lines␣in␣multiline.txt")
  Function.const(Stream.empty)(_)
} { e: Throwable =>
  println("An␣error␣occurred␣during␣splitting␣multiline.txt")
}
```

Listing 9. Using Carol's line splitter

In this use case, Carol created a function from three library-defined keywords.

(1) `AutoClose` for resource management, similar to C++'s RAII feature.

(2) `Yield` for lazily append values to a `Stream`, similar to Python, C# or ECMAScript's `yield` keyword.
(3) `Shift` for awaiting a value from a task, similar to Python, C# or ECMAScript's `await` keyword.

What is interesting is that our library-defined keywords are more like first-class features than compiler-defined keywords. Despite the fact that Python 3.5, C# and ECMAScript do not support automatic resource management, they also do not support using both `yield` and `await` in one function, even when `yield` and `await` are supported respectively, and Python 3.6 needs a special implementation of Asynchronous Generators [Selivanov 2016] to use both `yield` and `await`, while our library-defined keywords can collaborate with arbitrary other LDKs by composing extra domains on the return type.

2.4 Fork / join in asynchronous programming

We provided a type alias **type** `Task[A] = TailRec[Unit] !! Throwable !! A` for asynchronous programming.

For example, Suppose Dave is creating an HTTP client. He can implement the HTTP protocol in the `Task` domain shown in Listing 10.

```
import dsl.task._
import dsl.keywords._, Shift.implicitShift, AsynchronousIo._
import java.io._
import java.net._
import java.nio._, channels._

def readAll(channel: AsynchronousByteChannel, destination: ByteBuffer): Task[
    Unit] = _ {
  if (destination.remaining > 0) {
    val numberOfBytesRead: Int = !Read(channel, destination)
    numberOfBytesRead match {
      case -1 =>
      case _  => !readAll(channel, destination)
    }
  } else {
    throw new IOException("The␣response␣is␣too␣big␣to␣read.")
  }
}

def writeAll[Domain](channel: AsynchronousByteChannel, destination: ByteBuffer)
    : Task[Unit] = _ {
  while (destination.remaining > 0) {
    !Write(channel, destination)
  }
}

def daveHttpClient(url: URL): Task[String] = _ {
  val socket = AsynchronousSocketChannel.open()
  try {
    val port = if (url.getPort == -1) 80 else url.getPort
    val address = new InetSocketAddress(url.getHost, port)
```

```
    !AsynchronousIo.Connect(socket, address)
    val request = ByteBuffer.wrap(s"GET␣${url.getPath}␣HTTP/1.1\r\nHost:${url.
        getHost}\r\nConnection:Close\r\n\r\n".getBytes)
    !writeAll(socket, request)
    val response = ByteBuffer.allocate(100000)
    !readAll(socket, response)
    response.flip()
    io.Codec.UTF8.decoder.decode(response).toString
  } finally {
    socket.close()
  }
}
```

Listing 10. Dave's HTTP client

Dave's HTTP client is built from `Connect`, `Read` and `Write` LDKs. Those are asynchronous Java NIO.2 IO operators defined in `dsl.keywords.AsynchronousIo`, working with `dsl.task.Task` domain.

In this example, Dave imported `implicitShift`, which is an implicit conversion, which automatically converts `Task` or other CPS functions to `Shift` LDKs. Therefore, `!Shift(writeAll(...))` becomes unnecessary, in favor of `!writeAll(...)` .

The usage of `Task` can be similar to previous examples in Section 2.3, but we also provide `blockingAwait` and some other utilities under the implicit class `dsl.task.TaskOps` to ease the usage (Listing 11).

```
val fileContent = daveHttpClient(new URL("http://ftp.debian.org/debian/")).
    blockingAwait
fileContent should startWith("HTTP/1.1␣200␣OK")
```

Listing 11. Using Dave's http client

Another useful LDK for asynchronous programming is `Fork`, which duplicate the current control flow, and the child control flow are executed in parallel, similar to the POSIX `fork` system call.

Dave puts `Fork` inside a `join` block, which collects the result of each forked control flow in parallel (Listing 12).

```
import dsl.keywords.Fork
val Urls = Seq(
  new URL("http://ftp.debian.org/debian/README.CD-manufacture"),
  new URL("http://ftp.debian.org/debian/README")
)
def parallelTask: Task[Seq[String]] = Task.join {
  val url: URL = !Fork(Urls)
  !daveHttpClient(url)
}

val Seq(fileContent0, fileContent1) = parallelTask.blockingAwait
fileContent0 should startWith("HTTP/1.1␣200␣OK")
fileContent1 should startWith("HTTP/1.1␣200␣OK")
```

Listing 12. Using Dave's http client in parallel

In addition to Fork, we also provide the Each LDK, whose type signature is identical with Fork, to sequentially execute tasks. If Dave replaces the Fork to Each, those URLs will be fetched in sequentially. Other usage of Each LDK will be introduced in Section 2.5.

The Task implemented in *Dsl.scala* is light-weight and faster. See section 4 for the performance benchmark between `dsl.task.Task`, `scala.concurrent.Future`, `scalaz.concurrent.Task` and `monix.eval.Task`.

2.5 Monadic programming

Despite LDKs directly implemented in *Dsl.scala*, we also provide some LDKs as adapters to monads and other type classes.

The built-in Monadic LDK can be used as an adapter to `scalaz.Monad`, to create monadic code from imperative syntax, similar to the !-notation in Idris.

For example, suppose Erin is creating a program that counts lines of code under a directory. She uses the Monadic LDK store the result in a Stream of line count of each file (Listing 13).

```
import java.io.File
import dsl.keywords.Monadic
import dsl.domains.scalaz._
import scalaz.std.stream._
def erinMonadicCounter(file: File): Stream[Int] = Stream {
  if (file.isDirectory) {
    file.listFiles() match {
      case null =>
        // Unable to open `file`
        !Monadic(Stream.empty[Int])
      case children =>
        // Import this implicit conversion to omit the Monadic keyword
        import dsl.keywords.Monadic.implicitMonadic
        val child: File = !children.toStream
        !erinMonadicCounter(child)
    }
  } else {
    scala.io.Source.fromFile(file).getLines.size
  }
}
```

Listing 13. Erin's line of code counter, the monadic version

The previous code requires a `toStream` conversion on `children`, because `children`'s type `Array[File]` does not fit the `F` type parameter in `scalaz.Monad.bind` [Yoshida et al. 2017].

There is a Each LDK in *Dsl.scala* to extract each element in a Scala collection, based on CanBuildFrom type class instead of monads. The Each behavior is similar to Monadic, except the collection type can vary.

Thus, Erin can extract each element from an Array with the help of Each LDK in Listing 14, even when the enclosing domain is still a Stream.

```
import java.io.File
import dsl.keywords.Monadic, Monadic.implicitMonadic
```

```scala
import dsl.keywords.Each
import dsl.domains.scalaz._
import scalaz.std.stream._
def erinMixedCounter(file: File): Stream[Int] = Stream {
  if (file.isDirectory) {
    file.listFiles() match {
      case null =>
        // Unable to open `file`
        !Stream.empty[Int]
      case children =>
        val child: File = !Each(children)
        !erinMixedCounter(child)
    }
  } else {
    scala.io.Source.fromFile(file).getLines.size
  }
}
```

Listing 14. Erin's line of code counter, mixed Monad-based and CanBuildFrom-based LDKs

As shown the erinMixedCounter, Dsl.scala allows Each and other non-monadic LDKs to work along with monads, which is impossible in Haskell's do-notation or Idris's !-notation.

However, Erin still wants to add one more feature to the LOC counter. Considering the line counter implemented in previous example may be failed for some files, due to permission issue or other IO problem, Erin wants to use an OptionT monad transformer to mark those failed file as a None (Listing 15).

```scala
import scalaz._
import java.io.File
import dsl.keywords.Monadic, Monadic.implicitMonadic
import dsl.domains.scalaz._
import scalaz.std.stream._
def erinTransformerCounter(file: File): OptionT[Stream, Int] = OptionT.some {
  if (file.isDirectory) {
    file.listFiles() match {
      case null =>
        // Unable to open `file`
        !OptionT.none[Stream, Int]
      case children =>
        val child: File = !Stream(children: _*)
        !erinTransformerCounter(child)
    }
  } else {
    scala.io.Source.fromFile(file).getLines.size
  }
}
```

Listing 15. Erin's line of code counter, using an OptionT monad transformer

Note that our LDKs are adaptive to the domain it belongs to. Thus, instead of explicit lifting as `!Monadic(OptionT.optionTMonadTrans.liftM(Stream(children: _*)))`, Erin can simply write `!Stream(children: _*)`. This implicit lifting feature looks like Idris's effect monads [Brady 2013b], though the mechanisms is different from **`implicit`** `lift` in Idris.

3 CREATING LIBRARY-DEFINED KEYWORDS

LDKs introduced in Section 2 are optional libraries, activated by common compiler-time CPS-transform rules, which are implemented as a Scala compiler plug-in. In this section, we will present the implementation of some LDKs, and the compiler-time generated code for !-notations written by LDK users.

Dsl.scala ships with a compiler plug-in that supports both nonadaptive LDKs and adapters LDKs. A nonadaptive LDK must belongs to in an exact domain, while an adaptive LDK works in various types of domains.

3.1 Nonadaptive LDKs

A Nonadaptive LDK is simply a delimited continuation, along with a syntactic `unary_!` method for !-notation. For example, the `Yield` LDK described at Section 2.1 can be implemented as shown in Listing 16.

```
import dsl.Dsl.shift
case class Yield[Element](element: Element) {

  @shift
  @compileTimeOnly("Calls␣to␣this␣method␣will␣be␣translated␣to␣cpsApply␣calls␣
      by␣the␣compiler␣plug-in")
  def unary_! : Value = ???

  @inline
  def cpsApply(handler: Unit => Stream[Element]): Stream[Element] = {
    new Stream.Cons(element, handler(()))
  }
}
```

Listing 16. The `Yield` LDK, the nonadaptive version

Calls to `unary_!` method will be translated to `cpsApply` calls by our compiler plug-in. For example, `aliceRandomGenerator` in Listing 2 will be translated to the code shown in Listing 17 by our compiler plug-in:

```
def aliceRandomGenerator(seed: Int): Stream[Int] = {
  val tmp1 = seed ^ (seed << 13)
  val tmp2 = tmp1 ^ (tmp1 >>> 17)
  val tmp3 = tmp2 ^ (tmp2 << 5)
  Yield(tmp3).cpsApply { _: Unit =>
    aliceRandomGenerator(tmp3)
  }
}
```

Listing 17. The translated code for Alice's pseudo-random number generator

And Listing 4 or Listing 5 will be translated to Listing 18:

```scala
import dsl.Dsl.!!
def bobLoggingParser(jsonContent: String, defaultValue: JSONType): Stream[
    String] !! JSONType = { (callback: JSONType => Stream[String]) =>
  Yield(s"I␣am␣going␣to␣parse␣the␣JSON␣text␣$jsonContent...").cpsApply { _:
      Unit =>
    JSON.parseRaw(jsonContent) match {
      case Some(json) =>
        Yield(s"Succeeded␣to␣parse␣$jsonContent").cpsApply { _: Unit =>
          callback(json)
        }
      case None =>
        Yield(s"Failed␣to␣parse␣$jsonContent").cpsApply { _: Unit =>
          callback(defaultValue)
        }
    }
  }
}
```

Listing 18. The translated code for Bob's parser

Our compiler plug-in performs CPS-transform in a similar approach to Scala Continuations [Rompf et al. 2009], with some minor differences.

(1) Compiler-time instruction `reset` is not necessary in our implementation, as the boundary of a delimited continuation is by default the enclosing function. Instead, `reset` can be implemented as an ordinary Scala function shown in Listing 6.
(2) All return types are kept as is in our implementation, instead of hijacking on the `@cps` type.
(3) Our implementation only performs CPS-transform explicitly on the !-notation, instead of implicit conversion between `@cps` type and ordinary type.

Because of (2), CPS-translated function produced by our compiler plug-in always has the same answer type as the return type, which is called the "domain" of a DSL. Even then, our approach still allows explicit answer type by using the underscore trick shown in Listing 5.

3.2 Adaptive LDKs

All *Dsl.scala* built-in LDKs are adaptive, which can collaborate with other LDKs. For example, `Yield` is adaptive, since it works not only in functions that return `Stream[_]`, but also `(Stream[_] !! _)`, `(Stream[_] !! _ !! _)`, etc.

Those LDKs are adaptive because they all extends the **trait** Keyword, which has a ad-hoc polymorphic cpsApply method (Listing 19).

```scala
trait Keyword[Self, Value] extends Any { this: Self =>

  @shift
  @compileTimeOnly("Calls␣to␣this␣method␣will␣be␣translated␣to␣cpsApply␣calls␣
      by␣the␣compiler␣plug-in")
  def unary_! : Value = ???

  def cpsApply[Domain](handler: Value => Domain)(implicit dsl: Dsl[Self, Domain
      , Value]): Domain = {
```

```
    dsl.interpret(this, handler)
  }

}
```

Listing 19. The ad-hoc polymorphism in Keyword

The functionality of cpsApply is implemented in type class instances of `Dsl` (Listing 20).

```
trait Dsl[Keyword, Domain, Value] {
  def interpret(keyword: Keyword, handler: Value => Domain): Domain
}
```

Listing 20. The type class to interpret cpsApply

The adaptive version of `Yield` LDK ships with a type class instance of `Dsl[Yield[Element], Stream[Element], Unit]`, allowing the `!Yield` notation in functions that return `Stream[Element]`

```
case class Yield[Element](element: Element) extends Keyword[Yield[Element],
    Unit]

object Yield {
  implicit def yieldDsl[Element]: Dsl[Yield[Element], Stream[Element], Unit] =
    new Dsl[Yield[Element], Stream[Element], Unit] {
      def interpret(keyword: Yield[Element], mapper: Unit => Stream[Element]):
          Stream[Element] = {
        new Stream.Cons(keyword.element, mapper(()))
      }
    }
}
```

Listing 21. The `Yield` LDK, the adaptive version

To make `Yield` LDK available for another domain, just provide a `Dsl` type class instance for other types.

Listing 22 shows a `Dsl` that allows an LDK be available for `Domain!!Value` as long as the LDK is available for `Domain`. [1]

```
implicit def continuationDsl[Keyword, Domain, Value, KeywordValue](
  implicit restDsl: Dsl[Keyword, Domain, KeywordValue]
): Dsl[Keyword, Domain !! Value, KeywordValue] = {
  new Dsl[Keyword, Domain !! Value, KeywordValue] {
    def interpret(keyword: Keyword, handler: KeywordValue => Domain !! Value):
        Domain !! Value = {
      (continue: Value => Domain) =>
        restDsl.interpret(keyword, { a =>
          handler(a)(continue)
        })
    }
  }
```

[1] LDKs is also adaptive to domains other than continuations, such as a heterogeneous list that contains multiple sub-domains.

```
}
```

Listing 22. The `Yield` LDK, the adaptive version

Therefore, the `Yield` LDK can be used in `(Stream[String] !! JsonType)` domain as shown in Listing 6, because the type class `Dsl[Yield[String], Stream !! JsonType, Unit]` can be implicitly resolved as `continuationDsl(yieldDsl)`.

4 BENCHMARK

We created some benchmarks to evaluate the computational performance of code generated by our compiler plug-in for LDKs, especially, we are interesting how LDKs and other direct style DSL affect the performance in an effect system that support both asynchronous and synchronous effects.

In spite of LDKs of adapters to monads or other effect systems (see Section 2.5), the preferred effect system for LDKs is `Task`, the type alias of vanilla continuation-passing style function (Listing 23):

```scala
type !![Domain, Value] = (Value => Domain) => Domain
type TaskDomain = TailRec[Unit] !! Throwable
type Task[Value] = TaskDomain !! Value
```

Listing 23. The definition of `Task`, the preferred effect system using with LDKs

Our benchmarks measured the performance of LDKs in the `Task` domain, along with other combination of effect system with direct style DSL, listed in Table 1:

Effect System	direct style DSL
vanilla continuation-passing style functions	LDKs provided by *Dsl.scala*
Scala Future [Haller et al. 2012]	Scala Async [Haller and Zaugg 2013]
Scala Continuation library [Rompf et al. 2009]	Scala Continuation compiler plug-in
Monix tasks [Nedelcu et al. 2017]	`for` comprehension
Cats effects [Typelevel 2017]	`for` comprehension
Scalaz Concurrent [Yoshida et al. 2017]	`for` comprehension

Table 1. The combination of effect system and direct style DSL being benchmarked

4.1 The performance of recursive functions in effect systems

The purpose of the first benchmark is to determine the performance of recursive functions in various effect system, especially when a direct style DSL is used.

4.1.1 The performance baseline. In order to measure the performance impact due to direct style DSLs, we have to measure the performance baseline of different effect systems at first. We created some benchmarks for the most efficient implementation of a sum function in each effect system. These benchmarks perform the following computation:

- Creating a `List[X[Int]]` of 1000 tasks, where `X` is the data type of task in the effect system.
- Performing recursive right-associated "binds" on each element to add the `Int` to an accumulator, and finally produce a `X[Int]` as a task of the sum result.
- Running the task and blocking awaiting the result.

Note that the tasks in the list is executed in the current thread or in a thread pool. We keep each task returning a simple pure value, because we want to measure the overhead of effect systems, not the task itself.

The "bind" operation means the primitive operation of each effect system. For Monix tasks, Cats effects, Scalaz Concurrent and Scala Continuations library, the "bind" operation is `flatMap`; for *Dsl.scala*, the "bind" operation is `cpsApply`, which may or may not be equivalent to `flatMap` according to the type of the current domain.

We use the !-notation to perform the `cpsApply` in *Dsl.scala*. The !-notation results the exact same Java bytecode to manually passing a callback function to `cpsApply` (Listing 24).

```scala
def loop(tasks: List[Task[Int]], accumulator: Int = 0)(callback: Int =>
    TaskDomain): TaskDomain = {
  tasks match {
    case head :: tail =>
      // Expand to: implicitShift(head).cpsApply(i => loop(tail, i +
          accumulator)(callback))
      loop(tail, !head + accumulator)(callback)
    case Nil =>
      callback(accumulator)
  }
}
```

Listing 24. The most efficient implementation of sum based on vanilla CPS function

However, direct style DSLs for other effect systems are not used in favor of raw `flatMap` calls, in case of decay of the performance. Listing 25 shows the benchmark code for Scala Futures. The code for all the other effect systems are similar to it.

```scala
def loop(tasks: List[Future[Int]], accumulator: Int = 0): Future[Int] = {
  tasks match {
    case head :: tail =>
      head.flatMap { i =>
        loop(tail, i + accumulator)
      }
    case Nil =>
      Future.successful(accumulator)
  }
}
```

Listing 25. The most efficient implementation of sum based on Scala Futures

The benchmark result is shown in Table 2 (larger score is better):

The `Task` alias of continuation-passing style function used with *Dsl.scala* is quite fast. *Dsl.scala*, Monix and Cats Effects score on top 3 positions for either tasks running in the current thread or in a thread pool.

4.1.2 The performance impact of direct style DSLs. In this section, we will present the performance impact when different syntax notations are introduced. For vanilla CPS functions, we added

Benchmark	executedIn	size	Score, ops/s	
RawSum.cats	thread-pool	1000	799.072	± 3.094
RawSum.cats	current-thread	1000	26932.907	± 845.715
RawSum.dsl	thread-pool	1000	729.947	± 4.359
RawSum.dsl	current-thread	1000	31161.171	± 589.935
RawSum.future	thread-pool	1000	575.403	± 3.567
RawSum.future	current-thread	1000	876.377	± 8.525
RawSum.monix	thread-pool	1000	743.340	± 11.314
RawSum.monix	current-thread	1000	55421.452	± 251.530
RawSum.scalaContinuation	thread-pool	1000	808.671	± 3.917
RawSum.scalaContinuation	current-thread	1000	17391.684	± 385.138
RawSum.scalaz	thread-pool	1000	722.743	± 11.234
RawSum.scalaz	current-thread	1000	15895.606	± 235.992

Table 2. The benchmark result of sum for performance baseline

one more !-notation to avoid manually passing the `callback` in the previous benchmark (Listing 26, 27). For other effect systems, we refactored the previous sum benchmarks to use Scala Async, Scala Continuation's `@cps` annotations, and **`for`** comprehension, respectively (Listing 28, 29, 30, 31, 32, 33).

```
def loop(tasks: List[Task[Int]]): Task[Int] = _ {
  tasks match {
    case head :: tail =>
      !head + !loop(tail)
    case Nil =>
      0
  }
}
```

Listing 26. Left-associated sum based on LDKs of *Dsl.scala*

```
def loop(tasks: List[Task[Int]], accumulator: Int = 0): Task[Int] = _ {
  tasks match {
    case head :: tail =>
      !loop(tail, !head + accumulator)
    case Nil =>
      accumulator
  }
}
```

Listing 27. Right-associated sum based on LDKs of *Dsl.scala*

Note that reduced sum can be implemented in either left-associated recursion or right-associated recursion. The above code contains benchmark for both cases. The benchmark result is shown in Table 3, 4:

```
def loop(tasks: List[Future[Int]]): Future[Int] = async {
  tasks match {
    case head :: tail =>
      await(head) + await(loop(tail))
    case Nil =>
      0
  }
}
```

Listing 28. Left-associated sum based on Scala Async

```
def loop(tasks: List[Future[Int]], accumulator: Int = 0): Future[Int] = async {
  tasks match {
    case head :: tail =>
      await(loop(tail, await(head) + accumulator))
    case Nil =>
      accumulator
  }
}
```

Listing 29. Right-associated sum based on Scala Async

```
def loop(tasks: List[() => Int @suspendable]): Int @suspendable = {
  tasks match {
    case head :: tail =>
      head() + loop(tail)
    case Nil =>
      0
  }
}
```

Listing 30. Left-associated sum based on Scala Continuation plug-in

```
def loop(tasks: List[() => Int @suspendable], accumulator: Int = 0): Int @
    suspendable = {
  tasks match {
    case head :: tail =>
      loop(tail, head() + accumulator)
    case Nil =>
      accumulator
  }
}
```

Listing 31. Right-associated sum based on Scala Continuation plug-in

The result demonstrates that the !-notation provided by *Dsl.scala* is faster than all other direct style DSLs in the right-associated sum benchmark. The *Dsl.scala* version sum consumes a constant

```
def loop(tasks: List[Task[Int]]): Task[Int] = {
  tasks match {
    case head :: tail =>
      for {
        i <- head
        accumulator <- loop(tail)
      } yield i + accumulator
    case Nil =>
      Task(0)
  }
}
```

Listing 32. Left-associated sum based on **for** comprehension

```
def loop(tasks: List[Task[Int]], accumulator: Int = 0): Task[Int] = {
  tasks match {
    case head :: tail =>
      for {
        i <- head
        r <- loop(tail, i + accumulator)
      } yield r
    case Nil =>
      Task.now(accumulator)
  }
}
```

Listing 33. Right-associated sum based on **for** comprehension

Benchmark	executedIn	size	Score, ops/s	
LeftAssociatedSum.cats	thread-pool	1000	707.940	± 10.497
LeftAssociatedSum.cats	current-thread	1000	16165.442	± 298.072
LeftAssociatedSum.dsl	thread-pool	1000	729.122	± 7.492
LeftAssociatedSum.dsl	current-thread	1000	19856.493	± 386.225
LeftAssociatedSum.future	thread-pool	1000	339.415	± 1.486
LeftAssociatedSum.future	current-thread	1000	410.785	± 1.535
LeftAssociatedSum.monix	thread-pool	1000	742.836	± 9.904
LeftAssociatedSum.monix	current-thread	1000	19976.847	± 84.222
LeftAssociatedSum.scalaContinuation	thread-pool	1000	657.721	± 9.453
LeftAssociatedSum.scalaContinuation	current-thread	1000	15103.883	± 255.780
LeftAssociatedSum.scalaz	thread-pool	1000	670.725	± 8.957
LeftAssociatedSum.scalaz	current-thread	1000	5113.980	± 110.272

Table 3. The benchmark result of left-associated sum in direct style DSLs

number of memory during the loop, because we implemented a tail-call detection in our CPS-transform compiler plug-in, and the `Dsl` interpreter for `Task` use a trampoline technique [Tarditi et al. 1992]. On the other hand, the benchmark result of Monix Tasks, Cats Effects and Scalaz Concurrent

Benchmark	executedIn	size	Score, ops/s	
RightAssociatedSum.cats	thread-pool	1000	708.441	± 9.201
RightAssociatedSum.cats	current-thread	1000	15971.331	± 315.063
RightAssociatedSum.dsl	thread-pool	1000	758.152	± 4.600
RightAssociatedSum.dsl	current-thread	1000	22393.280	± 677.752
RightAssociatedSum.future	thread-pool	1000	338.471	± 2.188
RightAssociatedSum.future	current-thread	1000	405.866	± 2.843
RightAssociatedSum.monix	thread-pool	1000	736.533	± 10.856
RightAssociatedSum.monix	current-thread	1000	21687.351	± 107.249
RightAssociatedSum.scalaContinuation	thread-pool	1000	654.749	± 7.983
RightAssociatedSum.scalaContinuation	current-thread	1000	12080.619	± 274.878
RightAssociatedSum.scalaz	thread-pool	1000	676.180	± 7.705
RightAssociatedSum.scalaz	current-thread	1000	7911.779	± 79.296

Table 4. The benchmark result of right-associated sum in direct style DSLs

posed a significant performance decay, because they costs O(n) memory due to the `map` call generated by **for** comprehension, although those effect systems also built in trampolines. In general, the performance of recursive monadic binds in a **for** comprehension is always underoptimized due to the inefficient `map`.

4.2 The performance of collection manipulation in effect systems

The previous sum benchmarks measured the performance of manually written loops, but usually we may want to use higher-ordered functions to manipulate collections. We want to know how those higher-ordered functions can be expressed in direct style DSLs, and how would the performance be affected by direct style DSLs.

In this section, we will present the benchmark result for computing the Cartesian product of lists.

4.2.1 The performance baseline. As we did in sum benchmarks, we created some benchmarks to maximize the performance for Cartesian product. Our benchmarks create the Cartesian product from `traverseM` for Scala Future, Cats Effect, Scalaz Concurrent and Monix Tasks. Listing 34 shows the benchmark code for Scala Future.

Scala Async or **for** comprehension is used in element-wise task `cellTask`, but the collection manipulation `listTask` is kept as manually written higher order function calls, because neither Scala Async nor **for** comprehension supports `traverseM`.

The benchmark for *Dsl.scala* is entirely written in LDKs (Listing 35):

The `Each` LDK is available here because it is adaptive. `Each` LDK can be used in not only `List[_]` domain, but also `(_ !! Coll[_])` domain as long as `Coll` is a Scala collection type that supports `CanBuildFrom` type class.

We didn't benchmark Scala Continuation here because all higher ordered functions for `List` do not work with Scala Continuation.

The benchmark result is shown in Table 5.

Monix tasks, Cats Effects and vanilla CPS functions created from *Dsl.scala* are still the top 3 scored effect systems.

4.2.2 The performance of collection manipulation in direct style DSLs. We then refactored the benchmarks to direct style DSLs. Listing 36 is the code for Scala Future, written in `ListT` monad

```
import scala.concurrent.Future
import scalaz.std.list._
import scalaz.std.scalaFuture._
import scalaz.syntax.all._

def cellTask(taskX: Future[Int], taskY: Future[Int]): Future[List[Int]] = async
     {
  List(await(taskX), await(taskY))
}

def listTask(rows: List[Future[Int]], columns: List[Future[Int]]): Future[List[
    Int]] = {
  rows.traverseM { taskX =>
    columns.traverseM { taskY =>
      cellTask(taskX, taskY)
    }
  }
}
```

Listing 34. Cartesian product for Scala Future, based on traverseM

```
def cellTask(taskX: Task[Int], taskY: Task[Int]): Task[List[Int]] = _ {
  List(!taskX, !taskY)
}

def listTask(rows: List[Task[Int]], columns: List[Task[Int]]): Task[List[Int]]
    = {
  cellTask(!Each(rows), !Each(columns))
}
```

Listing 35. Cartesian product for vanilla CPS functions, based on *Dsl.scala*

Benchmark	executedIn	size	Score, ops/s	
RawCartesianProduct.cats	thread-pool	50	136.415	± 1.939
RawCartesianProduct.cats	current-thread	50	1346.874	± 7.475
RawCartesianProduct.dsl	thread-pool	50	140.098	± 2.062
RawCartesianProduct.dsl	current-thread	50	1580.876	± 27.513
RawCartesianProduct.future	thread-pool	50	100.340	± 1.894
RawCartesianProduct.future	current-thread	50	93.678	± 1.829
RawCartesianProduct.monix	thread-pool	50	142.071	± 1.299
RawCartesianProduct.monix	current-thread	50	1750.869	± 18.365
RawCartesianProduct.scalaz	thread-pool	50	78.588	± 0.623
RawCartesianProduct.scalaz	current-thread	50	357.357	± 2.102

Table 5. The benchmark result of Cartesian product for performance baseline

transformer provided by Scalaz. The benchmarks for Monix tasks, Scalaz Concurrent are also rewritten in the similar style.

```
import _root_.scalaz.syntax.all._
import _root_.scalaz.ListT
import _root_.scalaz.std.scalaFuture._

def listTask(rows: List[Future[Int]], columns: List[Future[Int]]): Future[List[
    Int]] = {
  for {
    taskX <- ListT(Future.successful(rows))
    taskY <- ListT(Future.successful(columns))
    x <- taskX.liftM[ListT]
    y <- taskY.liftM[ListT]
    r <- ListT(Future.successful(List(x, y)))
  } yield r
}.run
```

Listing 36. Cartesian product for Scala Future, based on `ListT` transformer

With the help of `ListT` monad transformer, we are able to merge `cellTask` and `listTask` into one function in a direct style **for** comprehension, avoiding any manual written callback functions.

We also merged `cellTask` and `listTask` in the *Dsl.scala* version of benchmark (Listing 37).

```
def listTask: Task[List[Int]] = reset {
  List(!(!Each(inputDslTasks)), !(!Each(inputDslTasks)))
}
```

Listing 37. Cartesian product for vanilla CPS functions, in one function

This time, Cats Effects are not benchmarked due to lack of `ListT` in Cats. The benchmark result are shown in Table 6.

Benchmark	executedIn	size	Score, ops/s	
CartesianProduct.dsl	thread-pool	50	283.450	± 3.042
CartesianProduct.dsl	current-thread	50	1884.514	± 47.792
CartesianProduct.future	thread-pool	50	91.233	± 1.333
CartesianProduct.future	current-thread	50	150.234	± 20.396
CartesianProduct.monix	thread-pool	50	28.597	± 0.265
CartesianProduct.monix	current-thread	50	120.068	± 17.676
CartesianProduct.scalaz	thread-pool	50	31.110	± 0.662
CartesianProduct.scalaz	current-thread	50	87.404	± 1.734

Table 6. The benchmark result of Cartesian product in direct style DSLs

Despite the trivial manual lift calls in **for** comprehension, the monad transformer approach causes terrible computational performance in comparison to manually called `traverseM`. In contrast, the performance of *Dsl.scala* even got improved when `cellTask` is inlined into `listTask`.

5 DISCUSSION AND CONCLUSION

This paper presents a novel approach to build embedded DSLs in control flow. The approach is based on three assumptions:

(1) The return type is the specific domain of a DSL.
(2) A DSL feature should be adaptive to various domains.
(3) Native control flow of the meta-language should be supported in a DSL.

By combining of the three assumptions, we defined the concept LDK (Library-Defined Keyword). An LDK is merely an ad-hoc polymorphic delimited continuation, interpreted by a domain-specific type class, as described in Section 3.

But what interesting is that an LDK can be considered as a more general version of monadic bind operation as well. The interpreter of an LDK is a triple parametric `Dsl` type class (Listing 20), which contains only one `interpret` function, whose type signature is `(K, (A => D)) => D`, which is exactly as same as monadic bind operation when `K` is `F[A]` and `D` is `F[B]`. Thus, the interpreter for `Monadic` LDK (See section 2.5) can be implemented as a trivial forwarder to the bind operation, as shown in Listing 38. In contrast, the reverse adapter is quite difficult, if not impossible, to be implemented.

```
implicit def monadDsl[F[_], A, B](implicit monad: Monad[F]): Dsl[Monadic[F, A],
     F[B], A] =
  new Dsl[Monadic[F, A], F[B], A] {
    def interpret(keyword: Monadic[F, A], handler: A => F[B]): F[B] = {
      monad.bind(keyword.fa)(handler)
    }
  }
```

Listing 38. The implementation of interpreter for `Monadic` LDK

The benchmarks in Section 4 demonstrated that our approach of triple parametric polymorphism improves both the extensibility and computational performance, in comparison to ordinary delimited continuations, monads or other direct style DSLs (Table 7).

Direct style DSL	Control flow	Extensibility	Performance
LDKs provided by *Dsl.scala*	supported	automatically adapted	good
Scala Async	supported	unsupported	good
Delimited continuation	supported	unsupported	good
`for` comprehension + monad transformer	unsupported	requires manually lifting	not good
Compiler-defined keywords `yield`, `async` and `await` in C#, Python or ECMAScript	supported	unsupported	uncomparable

Table 7. The comparison of direct style DSLs

The capacity of LDKs is the superset of both monads and ordinary delimited continuations, thus LDKs can be used in various domains they can be, including asynchronous or parallel programming, lazy stream generation, collection manipulation, resource management, etc. But unlike monads or ordinary delimited continuations, an LDK user can use multiple LDKs for different domains at once, along with ordinary control flow and ordinary types. No manually lifting is required, just like first-class features.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.

Kenichi Asai. 2009. On typing delimited continuations: three new solutions to the printf problem. *Higher-Order and Symbolic Computation* 22, 3 (2009), 275–291.

Edwin Brady. 2013a. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593.

Edwin Brady. 2013b. Programming and reasoning with algebraic effects and dependent types. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 133–144.

Flavio Brasil. 2017. *Monadless: Syntactic sugar for monad composition.* http://monadless.io/

Tom Crockett. 2013. *Effectful: A syntax for type-safe effectful computations in Scala.* https://github.com/pelotom/effectful

Olivier Danvy and Andrzej Filinski. 1990. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*. ACM, 151–160.

Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, , and Vojin Jovanovic. 2012. SIP-14 - Futures and Promises. (2012). https://docs.scala-lang.org/sips/futures-promises.html

Philipp Haller and Jason Zaugg. 2013. SIP-22 - Async. (2013). http://docs.scala-lang.org/sips/pending/async.html

Mark P Jones and Luc Duponcheel. 1993. *Composing monads.* Technical Report. Technical Report YALEU/DCS/RR-1004, Department of Computer Science. Yale University.

S Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, et al. 1998. *Haskell 98 report.* Technical Report. https://www.haskell.org/onlinereport/

Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible effects: an alternative to monad transformers. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 59–70.

Lightbend, Inc. 2017. *Akka FSM.* Lightbend, Inc. https://doc.akka.io/docs/akka/2.5.10/fsm.html

George Marsaglia et al. 2003. Xorshift RNGs. *Journal of Statistical Software* 8, 14 (2003), 1–6.

Caolan McMahon. 2017. *TensorFlow Control Flow.* https://www.tensorflow.org/api_guides/python/control_flow_ops

Alexandru Nedelcu, Sorin Chiprian, Mihai Soloi, Andrei OpriÈŹan, Jisoo Park, Dawid Dworak, Omar Mainegra, Piotr GawryÅŻ, A. Alonso Dominguez, Leandro Bolivar, Ryo Fukumuro, Ian McIntosh, Denys Zadorozhnyi, and Oleg Pyzhcov. 2017. *Monix: Asynchronous, Reactive Programming for Scala and Scala.js.* https://monix.io/

Rickard Nilsson. 2015. ScalaCheck: Property-based testing for Scala. (2015). https://www.scalacheck.org/

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. *The Scala language specification.* https://www.scala-lang.org/docu/files/ScalaReference.pdf

Dan Piponi. 2008. The Mother of all Monads. (2008). https://www.schoolofhaskell.com/user/dpiponi/the-mother-of-all-monads

Tiark Rompf, Ingo Maier, and Martin Odersky. 2009. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *ACM Sigplan Notices*, Vol. 44. ACM, 317–328.

Yury Selivanov. 2016. PEP 525 – Asynchronous Generators. *Python.org* (2016). https://www.python.org/dev/peps/pep-0525/

David Tarditi, Peter Lee, and Anurag Acharya. 1992. No assembly required: Compiling Standard ML to C. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 2 (1992), 161–177.

Twitter, Inc. 2016. *Algebird: Abstract Algebra for Scala.* Twitter, Inc. https://twitter.github.io/algebird/

Typelevel 2017. *typelevel/cats: Lightweight, modular, and extensible library for functional programming*. Typelevel. https://github.com/typelevel/cats

Philip Wadler. 1990. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*. ACM, 61–78.

Philip Wadler. 1992. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1–14.

Bo Yang. 2014a. *Stateless Future.* Shenzhen QiFun Network Corp., LTD. https://github.com/qifun/stateless-future

Bo Yang. 2014b. *Stateless Future Akka.* Shenzhen QiFun Network Corp., LTD. https://github.com/qifun/stateless-future-akka

Bo Yang. 2015. *ThoughtWorks Each: A macro library that converts native imperative syntax to scalaz's monadic expressions.* ThoughtWorks, Inc. https://github.com/ThoughtWorksInc/each

Bo Yang. 2016. *Binding.scala: Reactive data-binding for Scala.* ThoughtWorks, Inc. https://github.com/ThoughtWorksInc/Binding.scala

Kenji Yoshida, Alexey Romanov, Derek Williams, Edward Kmett, Heiko Seeberger, retronym, Mark Hibberd, Nick Partridge, runarorama, Richard Wallace, void, and Tony Morris. 2017. *Scalaz: An extension to the core scala library.* https://scalaz.github.io/scalaz/