

# Ad-hoc polymorphic delimited continuations

unifying monads and continuations

ANONYMOUS AUTHOR(S)

We designed and implemented a framework for creating extensible domain-specific languages that consists of library-defined keywords. First-class language features in other programming languages can be implemented as libraries with the help of our framework.

The core concept in our framework is the type class `Dsl`, which can be considered as both the ad-hoc polymorphic version of a delimited continuation and a more generic version of `Monad`. Thus it can be also used as a statically typed extensible effect system that is more efficient and more concise than existing `Monad`-based effect systems.

Additional Key Words and Phrases: type class, scala, delimited continuation, monad, haskell

## 1 INTRODUCTION

Traditionally, the capacity of a general purpose language can be extended to a special domain by creating an embedded Domain-Specific Language (eDSL) [Fowler 2010]. 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 designing a common protocol for control flow operators of all domains. In Haskell and other functional programming language, monads are used as the generic protocol of control flow operators [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 has to implement `>>=` and `return` operators 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 those domain-specific monadic data types respectively.

An idea to avoid incompatible domain-specific control flow is converting direct style control flow to domain-specific control flow at compile 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 2013] are compiler-time transformers to convert source code of direct style control flow to monadic control flow. For example, with the help of ThoughtWorks Each, `Binding.scala` [Yang 2016] can be used to create reactive HTML template from ordinary direct style code.

Another generic protocol of control flow is delimited 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 [Asai and Kameyama 2007]. 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 compile 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.

Monad transformers [Liang et al. 1995] is a popular technique to solve the collaboration problem. The basic idea is to use a `lift` function to convert different monadic type into the same transformed monadic type. Thus a `do` block of a transformed monadic type can contain different DSL operations as long as they can be `lifted`. With the help of additional type classes, those `lift` operations can be performed automatically.

However, a deeply nested transformed monad was considered inefficient due to the nested `lift`. An alternative approach proposed by [Kiselyov et al. 2013] is effect handlers. In the effect handler approach, the DSL “script” is written in a universal monadic type `Eff`, which allows for multiple DSLs in one `do` block. Each DSL is considered as an effect, which is dispatched by `Eff` to the specific Handler. This approach is heavy-weight, since only expressions written in `Eff` script are able to use DSLs defined in effect handlers. Additional conversion is required to retrieve the “raw” data type from an `Eff do` block.

This paper proposes a new type class `Dsl`, which can be considered as both the ad-hoc polymorphic version of a delimited-continuation and a more generic version of `Monad`. The Scala definition of the type class is shown in listing 1.

---

```

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

```

---

Listing 1. The definition of `Dsl` type class

Because `Dsl` is more generic than `Monad`, it allows a code block to contain interleaved heterogeneous Keywords, interpreted by different `Dsl` type class instances. Instead of returning an intermediate script type like `Eff` [Kiselyov et al. 2013], the return type of a DSL code block can vary as long as it supports operations used in it. No intermediate `Monad` for dispatching is used. The difference of architecture between effect handler approach and our approach is shown in figs. 1 and 2.

Ordinary delimited continuations [Danvy and Filinski 1989] can be defined as a CPS (Continuation-Passing Style) function to register a callback function (listing 2), which is similar to the signature of `Dsl` type class.

---

```

type Continuation[Domain, Value] = (Value => Domain) => Domain

```

---

Listing 2. The definition of a delimited continuation

A `Continuation` itself contains the implementation of an operation. As a result, a delimited continuation can only be used in a function that returns the specified `Domain`. In contrast, in our

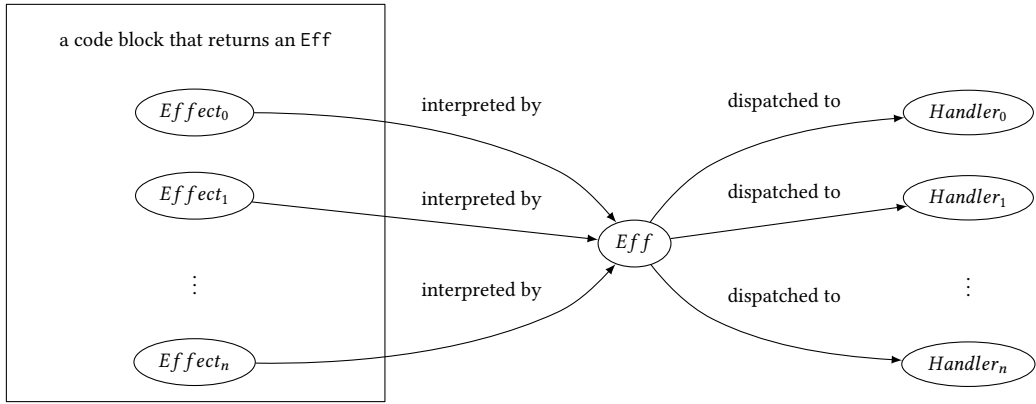


Fig. 1. The architecture of Eff approach

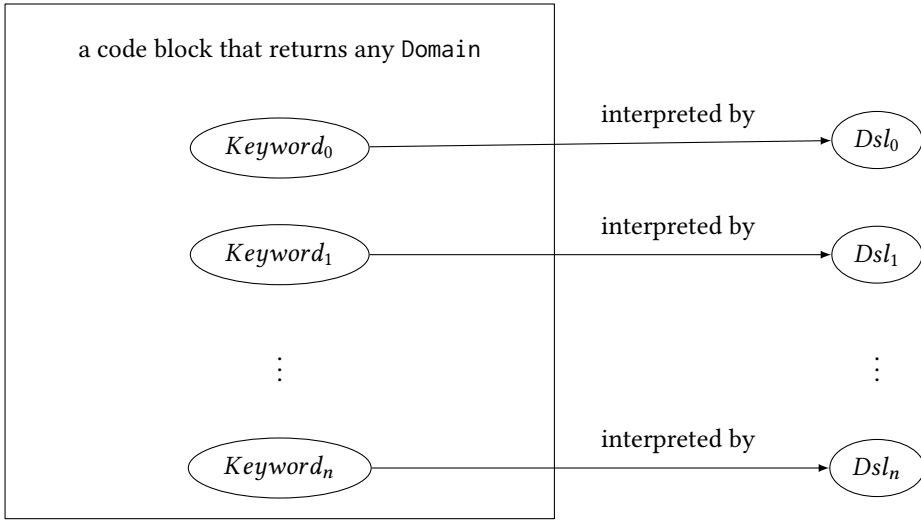


Fig. 2. The architecture of Dsl approach

approach, each Keyword is ad-hoc polymorphic to the Domain, thus it can be interpreted differently according to the enclosing Domain.

In the remaining sections of this paper, we will present the design and use cases of Dsl type class, including:

- (1) Simulating some first-class features in Python, C#, ECMAScript and C++, as library-defined keywords;
- (2) Simulating Monad to create imperative code blocks;
- (3) Composing delimited continuations with less closure creation than Monad for continuations;
- (4) Making Continuation stack safe, in a non-intrusive way;
- (5) Using any combination of the features of items 1 to 4, in a single code block.

All code examples except section 7 are written in our Scala library *Dsl.scala*, which provides an implementation of `Dsl` type class, along with a Scala compiler plug-in to perform a CPS-transformation. The compiler plug-in avoids the “callback hell” problem, allowing Idris-like `!`-notation [Brady 2013] direct style DSL in Scala, which can be used for not only monadic data type but also other operations.

## 2 FROM DELIMITED CONTINUATION TO THE DSL TYPE CLASS

Our goal is making the control flow of a programming language to be extensible. In this section, we will introduce the `Dsl` type class and the concept of name-based CPS transformation and demonstrate how to use these techniques to port first class Python language features to Scala, as library-defined keyword (LDK) <sup>1</sup>. The term LDK denotes language features implemented by libraries. No metaprogramming knowledge is required for either LDK authors or LDK users <sup>2</sup>, while, in other languages, they are used to be implemented as compiler built-in first-class features.

The remaining parts of this section are organized as following. Firstly, in sections 2.1 and 2.2, we will present how to port **yield** to Scala in the ordinary delimited continuation approach. Then in section 2.3, we will present how to port **await** to Scala in a monad-like interface. Finally, in sections 2.4 and 2.5, we will introduce the type class `Dsl` to unifying all the previous approaches, and in addition, allowing for the use of multiple LDKs like **yield** and **await** together.

### 2.1 Implementing LDKs as ordinary delimited continuations

In Python, ECMAScript, and C#, a generator is a function that returns an `Iterator` or an `IEnumerator`. The **yield** keyword is available inside the generator to lazily produce one element, which can be consumed by the `Iterator` / `IEnumerator` user. Listing 3 is a Python example to create an xorshift [Marsaglia et al. 2003] pseudo-random number generator that returns an infinite iterator of generated numbers. Note that `NumPy` <sup>3</sup> is used for 32-bit integers, and type hinting <sup>4</sup> is used for clarity.

---

```
def xor_shift_random_generator(seed: np.uint32) -> Iterator[np.uint32]:
    tmp1 = np.uint32(seed ^ (seed << 13))
    tmp2 = np.uint32(tmp1 ^ (tmp1 >> 17))
    tmp3 = np.uint32(tmp2 ^ (tmp2 << 5))
    yield tmp3
    yield from xor_shift_random_generator(tmp3)

generated_numbers = xor_shift_random_generator(seed = np.uint32(2463534242))

print(generated_numbers.__next__()) // The first generated random number
print(generated_numbers.__next__()) // The second generated random number
```

---

Listing 3. An Xorshift pseudo-random number generator in Python 3.5+

This generator feature can be ported to Scala as an LDK. In our LDK-based generator, the return type is replaced to `Stream`, which can be considered as the immutable version of `Iterator`, and the compiler-defined keyword **yield** is replaced to library-defined keyword `Yield`. Listing 4 is

<sup>1</sup>Code listings shown in section 2 are not exactly the same as the implementation in *Dsl.scala*, instead, these implementations of LDKs are modified or simplified for the purpose of introducing the concept of the LDK approach more clearly.

<sup>2</sup> Though, Scala LDKs need the common compiler plug-ins to perform CPS transformation and Haskell LDKs need `RebindableSyntax` described in section 7

<sup>3</sup><http://www.numpy.org/>

<sup>4</sup><https://docs.python.org/3/library/typing.html>

an example to create an Xorshift [Marsaglia et al. 2003] pseudo-random number generator that returns an infinite stream of generated numbers.

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

---

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

val generatedNumbers = xorShiftRandomGenerator(seed = 2463534242)

println(generatedNumbers(0)) // The first generated random number
println(generatedNumbers(1)) // The second generated random number
```

---

Listing 4. An Xorshift pseudo-random number generator with the help of the LDK `Yield`

Despite of the implementation of `Yield`, which will be revealed in upcoming sections, the above use case demonstrates some basic concepts in our approach:

- (1) `xorShiftRandomGenerator`, and any functions that contains nested continuation-passing style (CPS) calls, are considered as written in some kind of eDSL.
- (2) The word “domain” in the term “Domain-Specific Language” stands for the return type of the enclosing function. For example, `Stream[Int]` is the domain of `xorShiftRandomGenerator`.
- (3) The domain-specific language used by the enclosing function consists of some domain-specific “library-defined keywords” (LDK). For example, `Yield` is an LDK available for `Stream` domains.
- (4) Along with LDK, DSLs written in *Dsl.scala* also support native Scala control flows and expressions.

For a simple use case such as `xorShiftRandomGenerator`, LDKs can be implemented as ordinary delimited continuations. Listing 5 shows an implementation of the `Yield` LDK, as a delimited continuation, in which the `Yield` LDK creates infinite Streams by capturing handler into a lazily evaluated `Stream.Cons`.

---

```
case class Yield[A](element: A) extends Continuation[Stream[A], Unit] {
  def apply(handler: Unit => Stream[A]): Stream[A] = {
    new Stream.Cons(element, handler(()))
  }
}
```

---

Listing 5. Implementing `Yield` LDK as an ordinary delimited continuation

## 2.2 Auto-reset name-based CPS transformation

The syntax of listing 4 differs from first-class generators in Python, as the code block contains some manually created CPS closures. Ideally, the “rest” program after a `Yield` operation should be

indented at the same level of `Yield`, not in a nested closure. This coding style can be achieved by the `!`-notation provided by *Dsl.scala*'s built-in compiler plug-ins. The function `xorShiftRandomGenerator` can be written as listing 6 with the help of the `!`-notation plug-ins.

---

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

---

Listing 6. TheXorshift pseudo-random number generator, in the style of `!`-notation

Our compiler plug-ins performs CPS-transform in a similar approach to `reset/shift` control operators in Scala Continuations [Rompf et al. 2009]. The `!` prefix corresponds to the `shift` control operator, and the `reset` control operator will be automatically injected to every function body. Thus the above `xorShiftRandomGenerator` is equivalent to listing 7 in Scala Continuations.

---

```
def xorShiftRandomGenerator(seed: Int): Stream[Int] = reset {
  val tmp1 = seed ^ (seed << 13)
  val tmp2 = tmp1 ^ (tmp1 >>> 17)
  val tmp3 = tmp2 ^ (tmp2 << 5)
  shift(Yield(tmp3))
  xorShiftRandomGenerator(tmp3)
}
```

---

Listing 7. TheXorshift pseudo-random number generator, in Scala Continuations

Because of the automatically injected `reset` control operator, the boundary of a delimited continuation can never be escaped from a function in our approach. Therefore, our plug-ins are able to eliminate the internal context of delimited continuations. Scala Continuations' `ControlContext` and `cps` type annotations are not necessary any more.

There is another difference between our compiler plug-ins and Scala Continuation. Our compiler plug-ins are name-based instead of type-based, allowing CPS-transformation in monadic blocks, which will be discussed in next section.

### 2.3 Monadic blocks

In previous sections, we have demonstrated how to port the compiler-defined keyword `yield` to Scala, as a library-defined keyword. In this section, we will demonstrate how to import another compiler-defined keyword, `await`, to Scala, as a library-defined keyword.

`await` is available in Python, ECMAScript, or C#, to compose multiple asynchronous tasks into one task. The compiler-defined keyword `await` in Python is available in functions marked as `async`. Each `await` pauses the execution until the awaiting operation is complete, and each `return` keyword in an `async` function will turn the return value into an `Awaitable`. An example of creating an `Awaitable` to download two web pages by `aiohttp`<sup>5</sup> is shown in listing 8.

---

```
async def download_two_pages() -> Awaitable[Tuple[bytes, bytes]]:
  session = aiohttp.ClientSession()
  response1 = await session.get('http://example.com')
```

---

<sup>5</sup><https://docs.aiohttp.org/>

```

295 content1 = await response1.read()
296 response2 = await session.get('http://example.net')
297 content2 = await response2.read()
298 return (content1, content2)

```

---

Listing 8. Asynchronously downloading two web pages in Python

When porting **await** feature to Scala, we replaced the compiler-defined keyword **await** to a library-defined keyword **Await**, and replaced **Awaitable** to **Future**<sup>6</sup> as shown in listing 9. Note that **ByteString**, **Http**, **HttpRequest**, **HttpMethods**, **HttpRequest** in **downloadTwoPages** are asynchronous HTTP library provided by Akka<sup>7</sup> and Akka HTTP<sup>8</sup>.

```

306 def downloadTwoPages(): Future[(ByteString, ByteString)] = {
307   Await(Http().singleRequest(HttpRequest(HttpMethods.GET, "http://example.com")
308     )) { response1 =>
309     Await(response1.entity.toStrict(timeout = 5.seconds)) { content1 =>
310       Await(Http().singleRequest(HttpRequest(HttpMethods.GET, "http://example.
311         net")) { response2 =>
312         Await(response2.entity.toStrict(timeout = 5.seconds)) { content2 =>
313           Future((content1.data, content2.data))
314         }
315       }
316     }
317   }
318 }

```

---

Listing 9. Asynchronously downloading two web pages in *Dsl.scala*

**Await** should accept a handler to handle the incoming value in an asynchronous **Future**, and it can be implemented as a forwarder of **flatMap** on **Future**, as shown in listing 10.

```

324 case class Await[A](future: Future[A]) {
325   def apply[B](handler: A => Future[B])(implicit ec: ExecutionContext): Future[
326     B] = {
327     future.flatMap(handler)
328   }
329 }

```

---

Listing 10. Implementing **Await** LDK as a forwarder to **flatMap**

Similar to CPS-transformation in listing 6, the nested callback functions registered to **Await** in the **downloadTwoPages** method can be replaced to **!**-notation with the help of our compiler plug-ins. The direct style version of **downloadTwoPages** is shown in listing 11.

```

336 def downloadTwoPages(): Future[(ByteString, ByteString)] = Future {
337   val response1 = !Await(Http().singleRequest(HttpRequest(HttpMethods.GET, "
338     http://example.com"))))
339   val content1 = !Await(response1.entity.toStrict(timeout = 5.seconds))

```

---

<sup>6</sup><https://docs.scala-lang.org/overviews/core/futures.html>

<sup>7</sup><https://akka.io/>

<sup>8</sup><https://akka.io/akka-http/>



```

344     val response2 = !Await(Http().singleRequest(HttpRequest(HttpMethods.GET, "
345         http://example.net")))
346     val content2 = !Await(response2.entity.toStrict(timeout = 5.seconds))
347     (content1.data, content2.data)
348 }

```

---

Listing 11. Asynchronously downloading two web pages, in the style of !-notation

Note that listing 11 are unable to be expressed in Scala Continuation because the shift control operator accepts only CPS-functions, while the signature of `flatMap` differs from CPS-functions, due to the additional type parameter `B` and the additional implicit parameter of `ExecutionContext`.

Fortunately our CPS-transformation compiler plug-ins are name-based. Given any expression  $e_0$ ,  $e_1, \dots, e_n$ , variable name  $v_0, v_1, \dots, v_n$  and the final expression  $r$  in a function  $f$ , as shown in listing 12, it will be converted to listing 13 by our compiler plug-ins. The plug-ins convert ! prefixes to callback functions registrations, regardless what the signatures of those expressions are. Both delimited continuation and monad-like operations are supported. The behavior of our CPS-transformation compiler plug-ins is similar to !-notation in Idris or `do`-notation with `RebindableSyntax` in Haskell.

---

```

361 def f = {
362     val v0 = !e0;
363     val v1 = !e1;
364     ...
365     val vn = !en;
366     r;
367 }

```

---

Listing 12. A function with !-notation

---

```

370 def f = {
371     e0 { v0 =>
372         e1 { v1 =>
373             ...
374             en { vn =>
375                 r
376             }
377         }
378     }
379 }

```

---

Listing 13. The code converted from !-notation by our name-based CPS-transformation plug-ins

While `Await` implemented in listing 10 can “extract” the value of a `Future`, it can be generalized to any `Monads` as shown in listing 14.

---

```

385 trait Monad[F[_]] {
386     def bind[A, B](fa: F[A])(f: A => F[B])
387     def point[A](a: A): F[A]
388 }
389 object Monad {
390     implicit def futureMonad(implicit ec: ExecutionContext) = new Monad[Future] {
391         def bind[A, B](fa: Future[A])(f: A => Future[B]) = fa.flatMap(f)

```



```

393     def point[A](a: A): Future[A] = Future(a)
394   }
395 }
396
397 case class Monadic[F[_], A](fa: F[A]) {
398   def apply[B](handler: A => F[B])(implicit monad: Monad[F]): F[B] = {
399     monad.bind(fa)(handler)
400   }
401 }

```

---

Listing 14. Implementing Monadic LDK as a forwarder to Monad

Monadic is an LDK more generic than Await, able to “extract” any monadic value, not only future, as long as the corresponding Monad type class instance exists.

## 2.4 Collaborative library-defined keywords

In previous sections, we ported Python’s compiler-defined keywords **yield** and **await** to Scala, as library-defined keywords. However, those keywords are not collaborative. LDK Yield and Await implemented in previous sections cannot be present in the same function, while Python 3.5 allows using **yield** and **await** together to create asynchronous generators [Selivanov 2016].

In this section, we will present a use case of Python’s **yield** and **await** in one function, and then modify the previous implementation of LDK Yield and Await to gain the same ability of collaboration as Python.

---

```

415 async def download_two_pages_generator() -> AsyncGenerator[bytes, None]:
416   session = aiohttp.ClientSession()
417   response1 = await session.get('http://example.com')
418   content1 = await response1.read()
419   yield content1
420   response2 = await session.get('http://example.net')
421   content2 = await response2.read()
422   yield content2

```

---

Listing 15. Downloading two web pages as an asynchronous generator in Python

Listing 15 shows an example of downloading two web pages with combination of **yield** and **await**. In Python, when an **async** function like `download_two_pages_generator` contains both **yield** and **await** keywords, the return type becomes `AsyncGenerator`.

The corresponding type of `AsyncGenerator[bytes, None]` in Scala could be `Stream[Future[ByteString]]`, which should be the return type of `apply` in the modified version of Yield and Await. Therefore, the modified version of Yield and Await can be implemented as listings 16 and 17, and the usage of asynchronous generator with `!`-notation is shown in listing 18.

---

```

433 case class Yield[A](element: A) {
434   def apply(handler: Unit => Stream[Future[A]])(implicit ec: ExecutionContext):
435     Stream[Future[A]] = {
436       new Stream.Cons(Future(element), handler(()))
437     }
438 }

```

---

Listing 16. Implementing modified version of Yield LDK for creating asynchronous generators

---

```

442 case class Await[A](future: Future[A]) {
443   def apply[B](handler: A => Stream[Future[B]])(implicit ec: ExecutionContext):
444     Stream[Future[B]] = {
445     val ff = future.map(handler)
446     new Stream.Cons(ff.flatMap(_.head), result(ff, Duration.Inf).tail)
447   }
448 }
449 
```

---

Listing 17. Implementing a modified version of Await LDK for creating asynchronous generators

---

```

453 def downloadTwoPagesGenerator(): Stream[Future[ByteString]] = {
454   // The following Await and Yield LDKs will create a Future to download the
455   // page at example.com, as the first element of the output Stream
456   val response1 = !Await(Http().singleRequest(HttpRequest(HttpMethods.GET, "
457     http://example.com")))
458   val content1 = !Await(response1.entity.toStrict(timeout = 5.seconds))
459   !Yield(content1.data)
460
461   // The following Await and Yield LDKs will create a Future to download the
462   // page at example.net, as the second element of the output Stream
463   val response2 = !Await(Http().singleRequest(HttpRequest(HttpMethods.GET, "
464     http://example.net")))
465   val content2 = !Await(response2.entity.toStrict(timeout = 5.seconds))
466   !Yield(content2.data)
467
468   // Remaining elements after yielded futures
469   Stream.empty[Future[ByteString]]
470 }

```

---

Listing 18. Downloading two web pages as an asynchronous generator, in the style of !-notation

---

Semantically, each Yield LDK “prepend” a value at the head of the output Stream, and the remaining parts of the output Stream is a Stream.empty. Any asynchronous Await operations performed before a Yield are collected as the asynchronous Future for the yielded element.

The modified version of Yield and Await LDKs are collaborative, as they are both available for the domain of Stream[Future[ByteString]], thus they can be used together in one function.

## 2.5 Adaptive library-defined keywords

In previous sections, we presented two different implementations of Yield in listings 5 and 16 and two different implementations of Await in listings 10 and 17 for creating asynchronous value and asynchronous generators, respectively. However, the semantics the Yield and Await are not automatically determined by their context like Python. In this section, we will introduce the type class Dsl for creating adaptive library-defined keywords to solve the collaboration problem.

In *Dsl.scala*, the Dsl type class as defined in listing 1 is usually used along with Keyword (listing 19), which should be the super type of all adaptive LDKs.

---

```

489 trait Keyword[Self, Value] { this: Self =>

```

---

```

491     @inline def cpsApply[Domain](handler: Value => Domain)(implicit dsl: Dsl[Self
492         , Domain, Value]): Domain = {
493         dsl.cpsApply(this, handler)
494     }
495
496     def apply[Domain](handler: Value => Domain)(implicit dsl: Dsl[Self, Domain,
497         Value]): Domain = cpsApply(handler)
498 }

```

Listing 19. Keyword, the super type of all adaptive LDKs

An apply call is an alias of cpsApply, which registers a callback to handle the Value, and finally returns a Domain. The self type (Self) and the value of the keyword (Value) are defined in sub types of Keyword. The actual implementation of a keyword is resolved by the multi-parameter type class Dsl, which varies according to Domain, the return type of the enclosing function of the keyword's call site. For example, the adaptive version of Yield and Await can be defined as listings 20 and 21.

```

507 case class Yield[A](element: A) extends Keyword[Yield[A], Unit]

```

Listing 20. The Yield LDK, the adaptive version

```

511 case class Await[Value](future: Future[Value]) extends Keyword[Await[Value],
512     Value]

```

Listing 21. The Await LDK, the adaptive version

When performing !-notation on a Keyword to produce a Value inside a function whose return type is Domain, the type class instance of Dsl[Keyword, Domain, Value] is required. For example, adaptive version of LDKs in listings 5, 10, 16 and 17 requires Dsl instances implemented in listings 22 to 25.

```

520 implicit def yieldDsl[A, B >: A]: Dsl[Yield[A], Stream[B], Unit] =
521     new Dsl[Yield[A], Stream[B], Unit] {
522         def cpsApply(keyword: Yield[A], mapper: Unit => Stream[B]): Stream[B] = {
523             new Stream.Cons(keyword.element, mapper())
524         }
525     }

```

Listing 22. The Dsl type class instance of Yield for creating generators

```

528 implicit def futureYieldDsl[A, B >: A]: Dsl[Yield[A], Stream[Future[B]], Unit]
529     =
530     new Dsl[Yield[A], Stream[Future[B]], Unit] {
531         def cpsApply(keyword: Yield[A], handler: Unit => Stream[Future[B]]): Stream
532             [Future[B]] = {
533             new Stream.Cons(Future.successful(keyword.element), handler())
534         }
535     }

```

Listing 23. The Dsl type class instance of Yield for creating asynchronous generators

---

```

540
541 implicit def awaitDsl[A, B](implicit ec: ExecutionContext): Dsl[Await[A],
542     Future[B], A] =
543     new Dsl[Await[A], Future[B], A] {
544         def cpsApply(keyword: Await[A], handler: A => Future[B]): Future[B] = {
545             keyword.future.flatMap(handler)
546         }
547     }

```

---

Listing 24. The Dsl type class instance of Await for creating asynchronous values

---

```

550
551 implicit def streamAwaitDsl[A, B](implicit ec: ExecutionContext): Dsl[Await[A],
552     Stream[Future[B]], A] =
553     new Dsl[Await[A], Stream[Future[B]], A] {
554         def cpsApply(keyword: Await[A], handler: A => Stream[Future[B]]): Stream[
555             Future[B]] = {
556             val ff = keyword.future.map(handler)
557             new Stream.Cons(ff.flatMap(_.head), result(ff, Duration.Inf).tail)
558         }
559     }

```

---

Listing 25. The Dsl type class instance of Await for creating asynchronous generators

By introducing the type class Dsl, the calls to Keyword are ad-hoc polymorphic to the specific domain of the call site. As a result, library-defined keywords like Yield and Await are now adaptive like first-class keywords.

### 3 IMPLEMENTATION

We implemented the LDK approach in the framework *Dsl.scala*, which consists of the following parts:

**The core library** contains the definition of the Dsl type class and Keyword, the common super type of LDKs. They are slightly different from the definition in listings 1 and 19:

- There is an additional dummy method unary\_! annotated as @shift defined in Keyword. The unary\_! method (or any other @shift-annotated methods) will be specially treated by our compiler plug-ins, thought it will be considered as an ordinary method from the point view of type checker when our compiler plug-ins is not enabled. The definition of unary\_! method is especially useful for IntelliJ IDEA<sup>9</sup>'s built-in type checker, preventing the edit window in the IDE from being red marked.
- Keyword is a universal trait<sup>10</sup>, allowing its subtypes to be value classes, which involves lower memory overhead in most of LDK use cases.

**Compiler plug-ins** performs CPS-transformation as described in section 2.2. There are two compiler plug-ins in *Dsl.scala*: ResetEverywhere and BangNotation. The ResetEverywhere plug-in adds a hidden @reset annotation to the block of every method in source code, and the BangNotation plug-in perform CPS-transformation according to the unary\_! method (or any method annotated as @shift) and @reset annotation, which are equivalent to shift and reset control operators [Danvy and Filinski 1989], respectively.

<sup>9</sup><https://www.jetbrains.com/idea/>

<sup>10</sup><https://docs.scala-lang.org/overviews/core/value-classes.html>

In addition to block expressions mentioned in listing 13, all other first-class control flows in Scala<sup>11</sup> are transformed to CPS form by the `BangNotation` plug-in in the metacontinuation [Danvy and Filinski 1990] approach.

Unlike other typed delimited continuation implementations, the `BangNotation` plug-in performs name-based CPS-transformation. Each `!`-notation in a transformed function can be converted to an arbitrary `cpsApply` method call as long as it accepts a callback function parameter. Type checking for the transformed function will be performed once the transformation is done.

Although the `Dsl` type class does not allow changing the domain of a DSL code block, the `BangNotation` plug-in itself allows domain changing when the `cpsApply` method is implemented without `Dsl` type class. Thus, the `printf` problem can be trivially resolved by our compiler plug-ins as described in section 4.1.

**Built-in library-defined keywords** are shipped with *Dsl.scala*, to provide many language features that are not available natively in Scala, including:

- The `Await` LDK for asynchronous programming with Scala Future, similar to the `await` and `async` keywords in C#, Python and JavaScript.
- The `Shift` LDK for asynchronous programming with delimited continuations, similar to the `shift` operator in Scala Continuations.
- The `AsynchronousIo` LDKs for perform I/O on an asynchronous channel.
- 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 `Continue` LDK to skip an element in a LDK-based collection comprehension, similar to `continue` keyword in many languages.
- 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 2013].

**Asynchronous task utilities** contains a `Task` type and related utility functions, for stack-safe asynchronous programming with the ability of exception handling and auto-closeable resource management. `Task` is a type alias of delimited continuation whose answer type is composed of `TailRec` and `Throwable` in the approach described in section 4.3.

According to the result of the benchmarks shown in section 5, the computational performance of `Task` in *Dsl.scala* is comparable to state-of-the-art Scala asynchronous programming libraries when running in HotSpot Server VM, and it achieves significant higher performance than state-of-the-art libraries when running in GraalVM.

## 4 USE CASES

We will present some use cases of name-based CPS transformation and LDK in this section, to illustrate the simplicity of our approach, in comparison to previous solutions.

<sup>11</sup>Note that the `for` expression is not converted as it is not a first-class control flow but a group of nested method calls.

#### 4.1 Resolve the printf problem, trivially

The type-safe printf problem [Danvy 1998] is often used to demonstrate the ability of modifying the answer type of a typed delimited continuation. The problem can be also resolved by *Dsl.scala*'s CPS-transformation plug-ins as shown in listing 26.

---

```

643 object IntPlaceholder {
644   @shift def unary_! : String = ???
645   def cpsApply[Domain](f: String => Domain): Int => Domain = { i: Int =>
646     f(i.toString)
647   }
648 }
649
650 object StringPlaceholder {
651   @shift def unary_! : String = ???
652   def cpsApply[Domain](f: String => Domain): String => Domain = f
653 }
654
655 def f1 = "Hello_World!"
656 def f2 = "Hello_" + !StringPlaceholder + "!"
657 def f3 = "The_value_of_" + !StringPlaceholder + "_is_" + !IntPlaceholder + "."
658
659 println(f1) // Output: Hello World!
660 println(f2("World")) // Output: Hello World!
661 println(f3("x")(3)) // Output: The value of x is 3.

```

---

Listing 26. A solution of the type-safe printf problem in *Dsl.scala*

This solution works because our plug-ins performs CPS-transformation for `f1`, `f2` and `f3`, as shown in listing 27.

---

```

667 // The type of f1 is inferred as `String`
668 def f1 = "Hello_World!"
669
670 // The type of f2 is inferred as `String => String`
671 def f2 = StringPlaceholder.cpsApply { tmp =>
672   "Hello_" + tmp + "!"
673 }
674
675 // The type of f3 is inferred as `String => Int => String`
676 def f3 = StringPlaceholder.cpsApply { tmp0 =>
677   IntPlaceholder.cpsApply { tmp1 =>
678     "The_value_of_" + tmp0 + "_is_" + tmp1 + "."
679   }
680 }

```

---

Listing 27. The translated source code of *Dsl.scala*-base solution of printf problem

Our solution is more concise than the solution with Scala Continuations [Rompf et al. 2009], because: (1) No explicit reset is required, as reset is automatically added by the `ResetEverywhere` plug-in. (2) No explicit `@cps` type annotation is required, since the `BangNotation` plug-in is

name-based. The type of `f1`, `f2` and `f3` can be inferred automatically, according to Scala's type inference algorithm for closures.

## 4.2 The underscore trick

As described in section 3, our compiler plug-ins automatically perform reset control operation for every function. However, a complex continuation is usually executed across multiple functions, which requires an approach to prevent the automatically performed reset control operation.

We will propose two approaches to resolve the problem. The first solution is called “underscore trick”, which will be discussed in this section. Another solution is automatically derived Return LDK, which will be described in section 4.3.

For example, in addition to **yield**, Python generators also allow the **return** and **yield from** keywords. A generator that contains both **yield** and **return** keywords can be invoked by **yield from** from another generator. The elements **yielded** in the former generator will be added into the latter generator, and the return value of the former generator can be used in the latter generator, too. An example of **return** and **yield from** is shown in listing 28.

---

```
def returnable_generator() -> Generator[str, None, int]:
    yield 'inside_returnable_generator'
    return 1

def generator_test() -> Iterator[str]:
    yield 'before_returnable_generator'
    v = yield from returnable_generator()
    yield 'after_returnable_generator'
    yield f'the_return_value_of_returnable_generator_is_{v}'

# Output:
# before returnable_generator
# inside returnable_generator
# after returnable_generator
# the return value of returnable_generator is 1
print(*generator_test(), sep='\n')
```

---

Listing 28. Use **yield from** and **return** in Python generators

Unlike generators introduced in section 2.1, `returnable_generator` has the additional ability of returning values, thus its return type becomes to `Generator[str, None, int]`, where `str` is the iterator element type and `int` is the type to return<sup>12</sup>.

When porting **return** and **yield from** to Scala, the return type should indicate both the element type and the type to return, thus `Stream` is not applicable for return type any more. We can instead use the return type `Continuation[Stream[String], Int]`, as shown in listing 29. It accepts a callback function `k`, which can handle the `Int` value being returned and resume the rest program in `generatorTest`. Note that the underscore character is a Scala parameter placeholder for the callback function of the created `Continuation` closure.

---

```
def returnableGenerator(): Continuation[Stream[String], Int] = _ {
```

---

<sup>12</sup>Note that the declared return type and the type to return are different in Python generators. In other words, the **return** keyword in Python “lifts” the plain value to a `Generator`.



```

736     !Yield("inside_returnableGenerator")
737     1
738 }
739
740 def generatorTest(): Stream[String] = {
741     !Yield("before_returnableGenerator")
742     val v = !Shift(returnableGenerator())
743     !Yield("after_returnableGenerator")
744     !Yield(s"the_return_value_of_returnableGenerator_is_$v")
745     Stream.empty
746 }
747
748 generatorTest.foreach(println)

```

---

Listing 29. Returning an additional value in LDK-based generators

We also create `Shift`, an additional ad-hoc polymorphic LDK used in `generatorTest`, to perform the continuation<sup>13</sup>. It can be considered as the LDK-based replacement of Python’s **yield from** keyword, which is defined as listing 30.

```

754 case class Shift[Domain, Value](continuation: Continuation[Domain, Value])
755 extends Keyword[Shift[Domain, Value], Value]

```

---

Listing 30. The definition of `Shift` LDK

As described in section 2.5, we had split the LDK declaration (i.e. a subtype of `Keyword`) from its implementation (i.e. a `Dsl` type class instance). A `Dsl` type class instance of `Dsl[Shift[Stream[String], Int], Stream[String], Int]` is required to perform `!Shift` in the domain of `Stream[String]`. The implementation should forward `cpsApply` call to the underlying continuation of the `Shift` LDK, as shown in listing 31.

```

764 implicit def shiftDsl[Domain, Value] =
765     new Dsl[Shift[Domain, Value], Domain, Value] {
766         def cpsApply(keyword: Shift[Domain, Value], handler: Value => Domain) =
767             keyword.continuation(handler)
768     }

```

---

Listing 31. The `Dsl` instance of `Shift` LDK, to forward `cpsApply` to the underlying continuation

The `Shift` LDK can be considered as a simple wrapper of `Continuation` that forward `cpsApply` calls to the underlying continuation.

Semantically, the automatically performed reset control operator is prevented by the prepending underscore character. We call this usage of the underscore character the “underscore trick”.

**4.2.1 Answer type modification.** “Underscore trick” can be used along with any CPS functions, not only monomorphic `Continuation` but also polymorphic delimited continuations [Asai and Kameyama 2007]. Polymorphic delimited continuations is a CPS function whose answer type can be modified, i.e.  $(A \Rightarrow B) \Rightarrow C$  where  $B$  and  $C$  differ. We provide a `PolymorphicShift` LDK to perform shift control operator for polymorphic delimited continuations, as defined in listing 32. Note that `PolymorphicShift` is not interpreted by `Dsl`, hence it is not ad-hoc polymorphic.

---

<sup>13</sup>There is an implicit conversion from `Continuation` to `Shift` LDK in *Dsl.scala*, thus the explicit `Shift()` call can be omitted. We keep the explicit instantiation of `Shift` in this section for clarity.

---

```

785 final case class PolymorphicShift[A, B, C](cpsApply: (A => B) => C) {
786   @shift def unary_! : A = ???
787 }
788
789
790 implicit def implicitPolymorphicShift[A, B, C](cpsApply: (A => B) => C) =
791   PolymorphicShift(cpsApply)

```

---

Listing 32. The definition of PolymorphicShift

The prefix problem introduced in [Asai and Kameyama 2007] can be resolved by the “underscore trick” along with PolymorphicShift, as shown in listing 33.

---

```

797 def visit[A](lst: List[A]): (List[A] => List[A] @reset) => List[List[A]] = _ {
798   lst match {
799     case Nil =>
800       !{ (h: List[A] => List[A]) =>
801         Nil
802       }
803     case a :: rest =>
804       a :: !{ (k: List[A] => List[A] @reset) =>
805         k(Nil) :: k(!visit(rest))
806       }
807   }
808 }
809
810 def prefix[A](lst: List[A]) = !visit(lst)
811
812 // Output: List(List(1), List(1, 2), List(1, 2, 3))
813 println(prefix(List(1, 2, 3)))

```

---

Listing 33. The solution to prefix problem by the “underscore trick”

Traditional polymorphic delimited continuations performs CPS transformation across functions. In contrast, with the help of the “underscore trick”, we achieve the same ability of answer type modification as polymorphic delimited continuations, by performing function-local CPS-translation.

### 4.3 Dsl Derivation

Another solution to allow continuations to cross multiple functions is Dsl Derivation.

In section 2.5, we have present how to create an LDK for different domains, interpreted by different implementations of Dsl type class instances. In this section, we will discuss derived Dsl type class instances for an LDK, available in derived domains.

A derived domain means a domain whose type signature contains another domain, and a derived Dsl means a Dsl whose implementation internally invokes another Dsl. For example, the domain Continuation[Stream[String], Int], which we used in section 4.2, can be considered as a derived domain of Stream[String]. The derived Dsl should be able to “lift” the domain Stream[String] to Continuation[Stream[String], Int], without manually created closures in the “underscore trick”. In addition, Dsl derivation approach supports early return, which is impossible in “underscore trick”.

For example, the native keyword **return** in Python can early return from a function, as shown in listing 34.

---

```

def early_generator(early_return: bool) -> Generator[str, None, int]:
    yield 'inside_early_generator'
    if early_return:
        yield 'early_return'
        return 1
    yield 'normal_return'
    return 0

def early_generator_test() -> Iterator[str]:
    yield 'before_early_generator'
    v = yield from early_generator(True)
    yield 'after_early_generator'
    yield f'the_return_value_of_early_generator_is_{v}'

# Output:
# before early_generator
# inside early_generator
# early return
# after early_generator
# the return value of early_generator is 1
print(*early_generator_test(), sep='\n')

```

---

Listing 34. Use **yield from** and **return** in Python generators

The ability of early return is impossible with Scala native keyword **return**, because **return 0** does not compile in a function whose type is not a `Int`. Instead we defined a new `Return` LDK to port Python **return** to Scala, as shown in listings 35 and 36.

---

```

case class Return[A](returnValue: A) extends Keyword[Return[A], Nothing]

```

---

Listing 35. The definition of `Return` LDK

---

```

def earlyGenerator(earlyReturn: Boolean): Continuation[Stream[String], Int] = {
    !Yield("inside_earlyGenerator")
    if (earlyReturn) {
        !Yield("early_return")
        !Return(1)
    }
    !Yield("normal_return")
    !Return(0)
}

def earlyGeneratorTest(): Stream[String] = {
    !Yield("before_earlyGenerator")
    val v = !Shift(earlyGenerator(true))
    !Yield("after_earlyGenerator")
    !Yield(s"the_return_value_of_earlyGenerator_is_$v")
}

```

---

```

883     Stream.empty
884 }

```

```

886 earlyGeneratorTest.foreach(println)

```

---

Listing 36. Use Shift and Return in LDK-based generators

Since we don't manually create the closure of the Continuation, the return type of `earlyGenerator` becomes `Continuation[Stream[String], Int]`, which requires some Dsl instances that we have not defined, which are listed below:

- (1) `Dsl[Yield[String], Stream[String], Unit]`  
(required by `!Yield` in `earlyGeneratorTest`)
- (2) `Dsl[Shift[Stream[String], Int], Stream[String], Int]`  
(required by `!Shift` in `earlyGeneratorTest`)
- (3) `Dsl[Yield[String], Continuation[Stream[String], Int], Unit]`  
(required by `!Yield` in `earlyGenerator`)
- (4) `Dsl[Return[Int], Continuation[Stream[String], Int], Nothing]`  
(required by `!Return` in `earlyGenerator`)

As discussed in section 4.2, items 1 and 2 can be resolved by `yieldDsl` and `shiftDsl`, respectively.

Items 3 and 4 are new instances, which can be derived from original Dsls:

Item 3 should register a callback handler and then return a new continuation, whose answer type is `Stream[String]`. Thus, the `Yield[String]` keyword can be performed inside the newly created continuation, by the existing Dsl instance `yieldDsl[String, String]`. The extracted value `v` and the final handler `k` is then passed to handler to continue the execution of rest program, as shown in listing 37.

---

```

908 implicit def yieldContinuationDsl = {
909     new Dsl[Yield[String], Continuation[Stream[String], Int], Unit] {
910         def cpsApply(keyword: Yield[String], handler: Unit => Continuation[Stream[
911             String], Int]): Continuation[Stream[String], Int] = { k =>
912             val v = !keyword
913             handler(v)(k)
914         }
915     }
916 }

```

---

Listing 37. The derived Dsl instance for Yield LDK, which can be used in a Continuation

As described in section 3, `!keyword` will be desugared to `keyword.cpsApply { v => ... }`, which is equivalent to `yieldDsl[String, String].cpsApply(keyword, { v => ... })` after inlining. Therefore, `yieldContinuationDsl` can be considered as a derived Dsl instance of implicitly resolved `yieldDsl`.

The implementation of `yieldContinuationDsl` can be generalized to any other LDKs, since `yieldContinuationDsl` does not depend on internal details of `Yield` and `yieldDsl`. Any instances of `Dsl[Keyword, Continuation[LeftDomain, RightDomain], Value]` can be derived from `Dsl[Keyword, LeftDomain, Value]` as shown in listing 38.

---

```

928 implicit def derivedContinuationDsl[Keyword, LeftDomain, RightDomain, Value](
929     implicit restDsl: Dsl[Keyword, LeftDomain, Value]
930 ) = {

```

```

932 new Dsl[Keyword, Continuation[LeftDomain, RightDomain], Value] {
933   def cpsApply(keyword: Keyword, handler: Value => Continuation[LeftDomain,
934     RightDomain]): Continuation[LeftDomain, RightDomain] = { k =>
935     restDsl.cpsApply(keyword, handler(_)(k))
936   }
937 }
938 }

```

---

Listing 38. The generic derived Dsl instance for any LDK, which can be used in a Continuation

Now the required Dsl instance of item 3 can be resolved from either `yieldContinuationDsl`, or, more generically, `derivedContinuationDsl(yieldDsl)`.

Similarly, since a `!Return` LDK immediately returns from the current function, the implementation of Dsl instance for `!Return` should skip the rest part of the function, which is captured as a callback function passed to `cpsApply`, as shown in listing 39. Then, item 4 can be resolved as `derivedContinuationDsl(returnDsl)`.

---

```

948 implicit def returnDsl[A] =
949   new Dsl[Return[A], A, Nothing] {
950     def cpsApply(keyword: Return[A], handler: Nothing => A) =
951       keyword.returnValue
952   }
953

```

---

Listing 39. The Dsl instance of Return LDK, to skip the registered callback function

Dsl derivation enables heterogeneous LDKs to be present in one function, whose return type is a derived domain composed from the required domain of LDKs in use. We will present another example of creating extensible effects with the help of Dsl derivation in section 4.4.2.

#### 4.4 Mutable states

Purely functional programming languages usually do not support first-class mutable variables. In those languages, mutable states can be implemented in state monads. In this section, we will present an alternative approach based on LDK to simulate mutable variable in a pure language<sup>14</sup>. Unlike state monads, our LDK-based approach is more straightforward, and supports multiple mutable states without manually lifting.

**4.4.1 Single mutable state.** We use unary function as the domain of mutable state. The parameter of the unary function can be read from Get LDK, and changed by Put LDK, which are defined in listings 40 and 41, respectively.

---

```

970 case class Get[S]() extends Keyword[Get[S], S]
971

```

---

Listing 40. The definition of Get LDK

---

```

974 case class Put[S](value: S) extends Keyword[Put[S], Unit]
975

```

---

Listing 41. The definition of Put LDK

---

<sup>14</sup>Scala is an impure language, but we don't use Scala's native `var` or other impure features when simulating mutable states, therefore, our approach can be ported to Haskell or other pure languages as described in section 7.

Listing 42 is an example of a unary function that accepts a string parameter and returns the upper-cased last character of the parameter. The initial value is read from Get LDK, then it is changed to upper-case by Put LDK. At last, another Get LDK is performed to read the changed value, whose last character is then returned.

---

```

def upperCasedLastCharacter: String => Char = {
  val initialValue = !Get[String]()
  !Put(initialValue.toUpperCase)

  val upperCased = !Get[String]()
  Function.const(upperCased.last)
}

// Output: 0
println(upperCasedLastCharacter("foo"))

```

---

Listing 42. Using Get and Put in a unary function

The Dsl instances for Get and Put used in upperCasedLastCharacter are shown in listings 43 and 44. The Dsl instance for Get LDK passes the currentValue to the handler of current LDK, and then continues the enclosing unary function; the Dsl instance for Put LDK ignores previousValue and continues the enclosing unary function with the new value in Put.

---

```

implicit def getDsl[S0, S <: S0, A] =
  new Dsl[Get[S0], S => A, S0] {
    def cpsApply(keyword: Get[S0], handler: S0 => S => A): S => A = {
      currentValue =>
        handler(currentValue)(currentValue)
    }
  }

```

---

Listing 43. The Dsl instance for Get LDK

---

```

implicit def putDsl[S0, S >: S0, A] =
  new Dsl[Put[S0], S => A, Unit] {
    def cpsApply(keyword: Put[S0], handler: Unit => S => A): S => A = {
      previousValue =>
        handler(())(keyword.value)
    }
  }

```

---

Listing 44. The Dsl instance for Put LDK

Traditionally, the data type of state monad is an opaque type alias of  $S \Rightarrow (S, A)$ , which is more complicated than our domain type  $S \Rightarrow A$ , indicating state monads are potentially less efficient than LDK-based implementation. We will discuss the reason why monad-based DSL are more complicated and less efficient than LDK-based DSL in section 6.7.

4.4.2 *Multiple mutable states.* Get and Put LDKs can be performed on multiple mutable states as well. The domain types are curried functions in those use cases.

In listing 45, we present an example to create a formatter that performs Put on a Vector[Any] to store parts of the string content. At last, a Return LDK is performed at last to concatenate those parts. The formatter internally performs Get LDKs of different types to retrieve different parameters.

---

```
def formatter: Double => Int => Vector[Any] => String = {
  !Put(!Get[Vector[Any]] :+ "x=")
  !Put(!Get[Vector[Any]] :+ !Get[Double])
  !Put(!Get[Vector[Any]] :+ ",y=")
  !Put(!Get[Vector[Any]] :+ !Get[Int])

  !Return((!Get[Vector[Any]]).mkString)
}

// Output: x=0.5,y=42
println(formatter(0.5)(42)(Vector.empty))
```

---

Listing 45. Using Get and Put in a curried function

Since we had introduced Dsl instance for Get and Put LDKs in unary functions, now we only need a derived Dsl instance to port these LDKs in curried functions, as shown in listing 46.

---

```
implicit def derivedFunction1Dsl[Keyword, State, Domain, Value](
  implicit restDsl: Dsl[Keyword, Domain, Value]
): Dsl[Keyword, State => Domain, Value] =
  new Dsl[Keyword, State => Domain, Value] {
    def cpsApply(keyword: Keyword, handler: Value => State => Domain): State =>
      Domain = { state: State =>
        restDsl.cpsApply(keyword, handler(_)(state))
      }
  }
}
```

---

Listing 46. Derived Dsl instance in a curried function

By combining getDsl and derivedFunction1Dsl together, the Scala compiler automatically searches matched type in the curried function when resolving the implicit Dsl instance for a Get LDK. For example, !Get[Vector[Any]]() reads the third parameter of the formatter. It will be translated to Get[Vector[Any]]().cpsApply { \_ => ... }, where the cpsApply call requires an instance of type Dsl[Get[Vector[Any]], Double => Int => Vector[Any] => String, Vector[Any]], which will be resolved as derivedFunction1Dsl(derivedFunction1Dsl(getDsl)). Similarly, the Dsl instance for reading the first parameter and the second parameter can be resolved as getDsl and derivedFunction1Dsl(getDsl), respectively.

Derived Dsl instance for Put and Return can be resolved similarly. Since all the !Put LDK in formatter write the third parameter, their Dsl instances are derivedFunction1Dsl(derivedFunction1Dsl(putDsl)); the Dsl instance for !Return are derivedFunction1Dsl(derivedFunction1Dsl(derivedFunction1Dsl(returnDsl))).

Now we had demonstrated a simple and straightforward solution for the feature of multiple mutable states, with the help of nested Dsl derivation.



## 4.5 Asynchronous programming

With the help of Dsl derivation, a complex DSL can be composed of simple features. For example, we provided a type alias Task for asynchronous programming, as a higher performance replacement of `scala.concurrent.Future`, as shown in listing 47. A new infix type alias `!!` is used instead of Continuation, as a shorter notation for nested Continuation types.

---

```
type !![Domain, Value] = Continuation[Domain, Value]
type Task[A] = TailRec[Unit] !! Throwable !! A
```

---

Listing 47. The definition of asynchronous Task

Task supports the features of tail call optimization and exception handling. Each feature corresponds to a part the type signature. `scala.util.control.TailCalls.TailRec` is used for tail call optimization, and `scala.Throwable` is used to represent the internal exceptional state.

We create some derived Dsls to handle exceptions, which support domains whose types match the pattern of  $(L_i \text{ !! } \dots \text{ !! } L_0 \text{ !! } \text{Throwable} \text{ !! } R_0 \text{ !! } \dots \text{ !! } R_i)$ , and some derived Dsls to optimize tail calls as trampolines, which support domains whose types match the pattern of  $(\text{TailRec}[\dots] \text{ !! } R_0 \text{ !! } \dots \text{ !! } R_i)$ , where  $L_0 \dots L_i$  and  $R_0 \dots R_i$  are arbitrary number of types<sup>15</sup>. Therefore, Dsl instances for Task are composed from these orthogonal features.

In section 4.5.1, we will present how to create an asynchronous HTTP client from Task; in section 4.5.2, we will introduce the usage of `Task[Seq[A]]`, which collects the results of multiple tasks into a Seq, either executed in parallel or sequentially.

*4.5.1 An asynchronous HTTP client.* Listing 48 is an example of an HTTP client built from low-level Java NIO.2 asynchronous IO operations. Note that the “underscore trick” is used to allow Task to be executed across functions.

---

```
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 asynchronousHttpClient(url: URL): Task[String] = _ {
```

---

<sup>15</sup>Those Dsls are implemented in the Dsl derivation technique described in section 4.3. Check the artifact for the complete implementation

```

1128 val socket = AsynchronousSocketChannel.open()
1129 try {
1130     val port = if (url.getPort == -1) 80 else url.getPort
1131     val address = new InetSocketAddress(url.getHost, port)
1132     !Connect(socket, address)
1133     val request = ByteBuffer.wrap(s"GET_${url.getPath}_HTTP/1.1\r\nHost:${url.
1134         getHost}\r\nConnection:Close\r\n\r\n".getBytes)
1135     !writeAll(socket, request)
1136     val MaxBufferSize = 100000
1137     val response = ByteBuffer.allocate(MaxBufferSize)
1138     !readAll(socket, response)
1139     response.flip()
1140     io.Codec.UTF8.decoder.decode(response).toString
1141 } finally {
1142     socket.close()
1143 }
1144 }

```

---

Listing 48. An asynchronous HTTP client

We defined Connect, Read and Write LDKs to register handlers to Java NIO.2 asynchronous IO operators. In addition to Task domain, those LDKs also support any domains that match types of (... !! Unit !! Throwable !! ...) or (... !! TailRec[Unit] !! Throwable !! ...) <sup>16</sup>.

!readAll(...) and !writeAll(...) are equivalent to !Shift(readAll(...)) and !Shift(writeAll(...)). The explicit Shift calls are omitted because we provided an implicit conversion from any Continuations(including Tasks) to Shift LDKs.

We also provided a blockingAwait method in an implicit class, to blocking await the result of the asynchronous task, therefore, asynchronousHttpClient can be used synchronously, as shown in listing 49.

```

1156 val httpResponse = asynchronousHttpClient(new URL("http://example.com/")).
1157     blockingAwait
1158 httpResponse should startWith("HTTP/1.1_200_OK")
1159

```

---

Listing 49. Using the example HTTP client

**4.5.2 Parallel execution.** 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, as shown in listing 50.

```

1165 val Urls = Seq(
1166     new URL("http://example.com/"),
1167     new URL("http://example.org/")
1168 )
1169 def parallelTask: Task[Seq[String]] = {
1170     val url: URL = !Fork(Urls)
1171     val content: String = !httpClient(url)
1172     !Return(content)
1173 }
1174

```

---

<sup>16</sup>Check the artifact for complete implementation.

```

1177
1178 val Seq(fileContent0, fileContent1) = parallelTask.blockingAwait
1179 assert(fileContent0.startsWith("HTTP/1.1_200_OK"))
1180 assert(fileContent1.startsWith("HTTP/1.1_200_OK"))
1181 %

```

---

Listing 50. Using HTTP client in parallel

Since the execution of `parallelTask` is forked, the two URLs will be downloaded in parallel. The results are then collected into a Task of Seq at the !Return LDK.

The Task implemented in *Dsl.scala* is light-weight and faster. See section 5 for the performance benchmark between `dsl.task.Task`, `scala.concurrent.Future`, `scalaz.concurrent.Task` and `monix.eval.Task`.

## 4.6 Collection comprehensions

List comprehension or array comprehension is a feature to create a collection based on some other collections, which has been implemented as first class feature in many programming languages including Scala. In this section, we will present the Each LDK, which allows collection comprehensions for arbitrary collection types. Unlike other first class comprehension, our LDK-based collection comprehension collaborates with other LDKs, thus allowing creating complex code of effects or actions along with collection comprehensions.

**4.6.1 Heterogeneous comprehensions.** Suppose we want to calculate all composite numbers below  $n$ , the program can be written in Scala's native **for**-comprehension as shown in listing 51.

```

1200 def compositeNumbersBelow(n: Int) = (for {
1201   i <- 2 until math.ceil(math.sqrt(n)).toInt
1202   j <- 2 * i until n by i
1203 } yield j).to[Set]

```

---

Listing 51. Calculating all composite numbers below  $n$  with **for**-comprehension

The `compositeNumbersBelow` can be ported to LDK-based collection comprehension with the following steps:

- (1) Replacing the **for** keyword and the trailing `.to[CollectionType]` by the heading *CollectionType*.
- (2) Replacing every `p <- e` by `val p = !Each(e)`.
- (3) Moving the value to **yield** to the last expression position of the comprehension block.

Therefore, listing 51 can be rewrite to listing 52 with the help of the Each LDK, or listing 53 after removing the temporary variable `j`.

```

1214 def compositeNumbersBelow(n: Int): Set[Int] = Set {
1215   val i = !Each(2 until math.ceil(math.sqrt(n)).toInt)
1216   val j = !Each(2 * i until n by i)
1217   j
1218 }
1219
1220 // Output: Set(10, 14, 6, 9, 12, 8, 4)
1221 println(compositeNumbersBelow(15))
1222

```

---

Listing 52. Calculating all composite numbers below  $n$  with Each LDK

---

```

1226
1227 def compositeNumbersBelow(n: Int): Set[Int] = Set {
1228     val i = !Each(2 until math.ceil(math.sqrt(n)).toInt)
1229     !Each(2 * i until n by i)
1230 }

```

---

Listing 53. Calculating all composite numbers below  $n$  with Each LDK, the simplicied version

Note that `compositeNumbersBelow` creates a `Set`, which is different from the type of source collection. Our LDK-base collection comprehension allows heterogeneous source collection types. Even other collection-like types, including `Array` and `String`, are supported, as shown in listing 54.

```

1237 def heterogeneous = List { !Each(Array("foo", "bar", "baz")) + !Each("LDK") }
1238
1239 // Output: List(fooL, fooD, fooK, barL, barD, barK, bazL, bazD, bazK)
1240 println(heterogeneous)

```

---

Listing 54. LDK-based heterogeneous collection comprehension based on `Array` and `String`

4.6.2 *Filters*. We also provides the `Continue` LDK to skip an element from the source collections. It provides the similar feature to the `if` clause in Scala's native `for`-comprehension. An example of using `Continue` LDK to calculate prime numbers is shown in listing 55.

```

1247 def primeNumbersBelow(maxNumber: Int) = List {
1248     val compositeNumbers = compositeNumbersBelow(maxNumber)
1249     val i = !Each(2 until maxNumber)
1250     if (compositeNumbers(i)) !Continue
1251     i
1252 }
1253
1254 // Output: List(2, 3, 5, 7, 11, 13)
1255 println(primeNumbersBelow(15))

```

---

Listing 55. Calculating all prime numbers below  $n$  with Each and Continue LDK

The implementation of `Continue` LDK is similar to `Return`, except is pass an empty collection to the handler instead of the given value.

4.6.3 *Asynchronous comprehensions*. The `Each` LDK can be used in `Task` of collections as well, with the help of `Dsl` derivation. The usage of `Each` is very similar to the `Fork` keyword. The only difference is that `Each` sequentially executes tasks while `Fork` executes tasks in parallel. For example, if we replace the `Fork` LDK in listing 50 by `Each`, those URLs will be fetched sequentially, as shown in listing 56.

```

1266 def sequentialTask: Task[Seq[String]] = {
1267     val url: URL = !Each(Urls)
1268     val content: String = !httpClient(url)
1269     !Return(content)
1270 }
1271

```

---

Listing 56. Using HTTP client in parallel

4.6.4 *Generator comprehensions*. Since the Each LDK works in any function that returns a collection, it can be also used in Stream functions, which support the Yield LDK as well. As a result, generator and collection comprehension can be used together.

Suppose we are creating a function to prepare flags for invoking the gcc command line tool. Given a source file and a list of include paths, it should return a Stream of the command line. It can be implemented from the Yield, Each and Continue as shown in listing 57.

```
def gccFlagBuilder(sourceFile: String, includes: String*): Stream[String] = {
  !Yield("gcc")
  !Yield("-c")
  !Yield(sourceFile)
  val include = !Each(includes)
  !Yield("-I")
  !Yield(include)
  !Continue
}

// Output: List(gcc, -c, main.c, -I, lib1/include, -I, lib2/include)
println(gccFlagBuilder("main.c", "lib1/include", "lib2/include").toList)
```

Listing 57. Build a command-line by using generator and collection comprehension together

## 5 BENCHMARKS

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

Our benchmarks measured the performance of LDKs in the Task domain mentioned in section 4.5, along with other combination of effect system with direct style DSL, listed in table 1:

Effect System	direct style DSL
The Task LDK	name-based CPS transformation provided by <i>Dsl.scala</i>
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

### 5.1 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.

5.1.1 *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, the “bind” operation is `flatMap`; for *Dsl.scala*, the “bind” operation is Shift LDK, which may or may not be equivalent to `flatMap` according to the type of the current domain. In Continuation domain, the *Dsl* instance for Shift LDK is resolved as `derivedContinuationDsl(shiftDsl)`, whose `cpsApply` method flat maps a Continuation to another Continuation; when using “underscore trick”, the *Dsl* instance for Shift LDK is resolved as `shiftDsl`, which just forwards `cpsApply` to the underlying CPS function as a plain function call.

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`, as shown in listing 58.

---

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

---

Listing 58. The most efficient implementation of sum based on ordinary 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 59 shows the benchmark code for Scala Futures. The code for all the other effect systems are similar to it.

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.

**5.1.2 The performance impact of direct style DSLs.** In this section, we will present the performance impact when different syntax notations are introduced. For ordinary CPS functions, we added one more `!`-notation to avoid manually passing the callback in the previous benchmark (listings 60 and 61). For other effect systems, we refactored the previous sum benchmarks to use Scala Async, Scala Continuation’s `@cps` annotations, and `for`-comprehension, respectively (listings 62 to 67).

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 tables 3 and 4:

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

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

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



---

```

1422 def loop(tasks: List[Future[Int]], accumulator: Int = 0): Future[Int] = {
1423   tasks match {
1424     case head :: tail =>
1425       head.flatMap { i =>
1426         loop(tail, i + accumulator)
1427       }
1428     case Nil =>
1429       Future.successful(accumulator)
1430   }
1431 }

```

---

Listing 59. The most efficient implementation of sum based on Scala Futures

---

```

1434 def loop(tasks: List[Task[Int]]): Task[Int] = _ {
1435   tasks match {
1436     case head :: tail =>
1437       !head + !loop(tail)
1438     case Nil =>
1439       0
1440   }
1441 }

```

---

Listing 60. Left-associated sum based on LDKs of *Dsl.scala*


---

```

1445 def loop(tasks: List[Task[Int]], accumulator: Int = 0): Task[Int] = _ {
1446   tasks match {
1447     case head :: tail =>
1448       !loop(tail, !head + accumulator)
1449     case Nil =>
1450       accumulator
1451   }
1452 }

```

---

Listing 61. Right-associated sum based on LDKs of *Dsl.scala*


---

```

1456 def loop(tasks: List[Future[Int]]): Future[Int] = async {
1457   tasks match {
1458     case head :: tail =>
1459       await(head) + await(loop(tail))
1460     case Nil =>
1461       0
1462   }
1463 }

```

---

Listing 62. Left-associated sum based on Scala Async

---

```

1471 def loop(tasks: List[Future[Int]], accumulator: Int = 0): Future[Int] = async {
1472   tasks match {
1473     case head :: tail =>
1474       await(loop(tail, await(head) + accumulator))
1475     case Nil =>
1476       accumulator
1477   }
1478 }

```

---

Listing 63. Right-associated sum based on Scala Async

---

```

1481 def loop(tasks: List[() => Int @suspendable]): Int @suspendable = {
1482   tasks match {
1483     case head :: tail =>
1484       head() + loop(tail)
1485     case Nil =>
1486       0
1487   }
1488 }
1489 }

```

---

Listing 64. Left-associated sum based on Scala Continuation plug-in

---

```

1492 def loop(tasks: List[() => Int @suspendable], accumulator: Int = 0): Int @
1493   suspendable = {
1494   tasks match {
1495     case head :: tail =>
1496       loop(tail, head() + accumulator)
1497     case Nil =>
1498       accumulator
1499   }
1500 }

```

---

Listing 65. Right-associated sum based on Scala Continuation plug-in

---

```

1504 def loop(tasks: List[Task[Int]]): Task[Int] = {
1505   tasks match {
1506     case head :: tail =>
1507       for {
1508         i <- head
1509         accumulator <- loop(tail)
1510       } yield i + accumulator
1511     case Nil =>
1512       Task(0)
1513   }
1514 }

```

---

Listing 66. 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 67. Right-associated sum based on **for**-comprehension

The result demonstrates that the name-based CPS transformation 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 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 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.

## 5.2 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.

**5.2.1 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 68 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 as shown in listing 69:

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 ordinary CPS functions created from *Dsl.scala* are still the top 3 scored effect systems.

```

1569 def cellTask(taskX: Future[Int], taskY: Future[Int]): Future[List[Int]] = async
1570   {
1571     List(await(taskX), await(taskY))
1572   }
1573
1574 def listTask(rows: List[Future[Int]], columns: List[Future[Int]]): Future[List[
1575   Int]] = {
1576   rows.traverseM { taskX =>
1577     columns.traverseM { taskY =>
1578       cellTask(taskX, taskY)
1579     }
1580   }
1581 }

```

Listing 68. Cartesian product for Scala Future, based on Scalaz's traverseM

```

1585 def cellTask(taskX: Task[Int], taskY: Task[Int]): Task[List[Int]] = _ {
1586   List(!taskX, !taskY)
1587 }
1588
1589 def listTask(rows: List[Task[Int]], columns: List[Task[Int]]): Task[List[Int]]
1590   = {
1591   cellTask(!Each(rows), !Each(columns))
1592 }

```

Listing 69. Cartesian product for ordinary 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

**5.2.2 The performance of collection manipulation in direct style DSLs.** We then refactored the benchmarks to direct style DSLs. Listing 70 is the code for Scala Future, written in ListT monad transformer provided by Scalaz. The benchmarks for Monix tasks, Scalaz Concurrent are also rewritten in the similar style.

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.

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

We also merged cellTask and listTask in the *Dsl.scala* version of benchmark as shown in listing 71.

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

Listing 71. Cartesian product for ordinary 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`.

6 RELATED WORKS

Previous works related to *Dsl.scala* can be divided into two categories:

**Generic protocols of control flow operators** are motivated by the goal similar to our `Dsl` type class. Operators of specific purposes can be implemented in single protocol, therefore, users of those operators can use a common interface for different domains. Monads and CPS functions are notable examples of such protocols.

**Direct style notations** provide similar syntaxes to our name-based CPS transformation. Those notations allow users to write sequential imperative style code that will be translated to CPS or monadic style that consist of nested closures. **yield**, **async** / **await**, reset / shift, **for**-comprehension, **do**-notation and **!**-notation are notable examples of such notations.

## 6.1 Generators

A generator is a special procedure to lazily produce values, which can be consumed as an iterator. Early implementation of generators are shipped in Alghard [Shaw et al. 1977] and CLU [Liskov et al. 1977], and the feature is now available in Python, ECMAScript, C#, and many other programming languages.

The execution of a generator will be paused at the **yield** statement, and can be resumed when the consumer side of the generator asks for the next value. The **yield** statement can be considered as a direct style notation for producer / consumer pattern.

Generators can be used for creating eDSLs in the following approach:

- The producer side **yields** command objects of the DSL.
- The consumer side interprets these produced command objects to actually perform operations.

However, the type of the command object is a part of the protocol between the consumer and the producer, and must be determined in advance. Therefore, the number of available commands in a generator is fixed. A generator DSL is not composable with other generator DSLs. In addition, generators are traditionally implemented as a first class feature by the compiler, thus they do not collaborate with other direct style notation unless changing the compiler.

In contrast, our LDK-based generators can be used along with other LDKs, including but not limited to Shift (section 4.2), Return (section 4.3), Await (section 2.4), Each (section 4.6.4), without modifying the compiler.

## 6.2 async and await

**async** and **await** are compiler-defined keywords in Python, ECMAScript, or C#, to compose multiple asynchronous tasks into one task. Like generators, **async** and **await** provide a special purpose direct style notation, which does not collaborate with other direct style notations, unless modifying the implementation of the compiler like [Selivanov 2016] did.

Alternatively, we provide Await, Shift LDK in *Dsl.scala* for asynchronous programming with Scala Futures and Continuation, respectively. Those asynchronous LDKs collaborate with other LDK as demonstrated in sections 2.4 and 4.5 and section 4.6.3.

## 6.3 Delimited continuations

Delimited continuations operators of shift and reset [Danvy and Filinski 1990] are direct style notations for performing CPS-transformation. The underlying data structures are either monomorphic [Danvy and Filinski 1989] or polymorphic [Asai and Kameyama 2007], which can be considered as a generic protocol of control flow operators.

Our BangNotation compiler plug-ins described in section 3 can be considered as a simplified version of delimited continuations operators, disallowing delimited continuations across multiple functions. Fortunately, this limitation can be overcome by underscore trick as described in section 4.2.

An ordinary delimited continuation is a CPS function whose implementation is predetermined. In contrast, we introduced the *Dsl* type class as a ad-hoc polymorphic CPS function, adaptive to the enclosing domain, as described in section 2.5.

## 6.4 for-comprehension

**for**-comprehension is a Scala language feature, originally used to produce collections. It is a general form of list comprehension. The Scala compiler internally translates **for**-comprehension expressions into method calls to `map`, `flatMap` and `withFilter`. The translation is also name-based, therefore, **for**-comprehension can be used not only for collection generation, but also as a general direct style notation for asynchronous programming<sup>17</sup>, resource management<sup>18</sup>, or creating monadic expression [Twitter, Inc. 2016; Typelevel 2017; Yoshida et al. 2017].

However, complex imperative procedures that contain native Scala control flow statements are not supported in **for**-comprehensions. In addition, **for**-comprehensions always end with a `map`, preventing tail call optimization when composing multiple **for**-comprehensions, consuming more memory and resulting worse computational performance than manually written `flatMap` calls, according to the benchmarks in section 5.1.2.

## 6.5 do-notation

**do**-notation was originally introduced in Haskell [Jones et al. 1998] as a direct style notation for creating monadic expressions in an imperative style. **do**-notation in Idris or `RebindableSyntax` in Haskell are name-based, as they can be used with type classes other than monads.

However, programs written in **do**-notation can be unnecessarily verbose, since nested `>>=` operators are not supported, instead, each `>>=` call must be present in an individual statement.

## 6.6 !-notation

**!**-notation is a direct style notation in Idris [Brady 2013], to make up nested expressions in **do**-notation. Our `BangNotation` and `ResetEverywhere` compiler plug-ins are re-implementation of Idris's **!**-notation in Scala, with some minor differences. Our compiler plug-ins support more native control flow expressions, including **do/while** and **try/catch/finally/throw**.

Since Idris's **!**-notation is name-based, `Ds1` type class can be port to Idris and work with **!**-notation as well.

## 6.7 Monads

A monad is a generic protocol of control flow operators used in Haskell and many other functional programming languages. A monad defines two primary operators for creating monadic expressions of a certain type. (1) The return operator<sup>19</sup> lifts a plain value to a monadic value. (2) The `>>=` operator<sup>20</sup> composes two steps monadic expressions into one monadic value, where the second step is a handler to map the value of the first step into a new monadic value.

Since a monad is specified to a certain monadic data type, the capacity of a monadic data type is predetermined, unless introducing an additional abstract layer of interpreters. For example, the `List` monad in Haskell<sup>21</sup> can be used to create a `List` based on `Lists`, but it cannot create a `List` based on other collection types, nor creating a `List` from a generator.

In our LDK approach, we remove the limitation of monads by separating the concept of monadic value into two distinguishing concepts: domain and library-defined keyword (LDK). A domain is the return type of the enclosing function, and an LDK is an operation in the domain. Therefore, a collection can be created from other collections or generators with different LDKs as discussed in sections 4.6.1 and 4.6.4.

<sup>17</sup>SIP-14 - Futures and Promises

<sup>18</sup>Scala ARM

<sup>19</sup>Also called `point` or `pure`.

<sup>20</sup>Also called `flatMap` or `bind`.

<sup>21</sup>A Haskell `List` is lazy by default, equivalent to a Scala `Stream`.



In addition, separating domains and LDKs can lead simpler implementation. Our `Get` and `Put` can be used in the domain of ordinary unary functions, while the monadic data type for a state monad is more complicated, as discussed in section 4.4.1. Also, a typical implementation of `Cont` monad [Dybvig et al. 2007], as defined in eqs. (1) and (2), is more complicated, as it creates  $\lambda\kappa.t_1(\lambda v.t_2v\kappa)$  and  $\lambda v.t_2v\kappa$ , two more additional closures, for each `>>=` operator. In contrast, the `Shift` LDK for delimited continuation runs as a simple forwarder, which creates no closure, as defined in listing 31.

$$\text{return}_k = \lambda t.\lambda\kappa.kt \quad (1)$$

$$>>_k = \lambda t_1.\lambda t_2.\lambda\kappa.t_1(\lambda v.t_2v\kappa) \quad (2)$$

In fact, the `>>=` operator of a monad is equivalent to a special case of `Dsl` when the domain type and the LDK type are the same, and the `return` operator of a monad can be considered another special case of `Dsl` when the LDK is a `Return`, which holds a plain value of the domain type.

There are other workarounds to overcome the limitation of monads, which will be discussed in sections 6.7.1 and 6.7.2.

**6.7.1 Monad transformers.** Monad transformers [Liang et al. 1995] are monads derived from other monads. “Monad transformer” is to “monad” as “`Dsl` derivation” is to “`Dsl`”. The monadic data type can be composed at type level as a chain of monad transformers, so that various operations can be lifted to the same nested transformed monadic data type, which can be then used in a single monadic code block. The process to perform an operation in a monad transformer is shown in fig. 3.

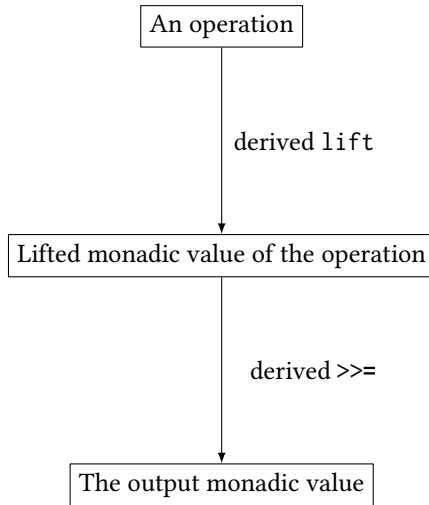


Fig. 3. Performing an operation in a monad transformer block

However, as [Kiselyov et al. 2013] pointed out, lifting a atomic type to a deeply nested transformed monadic data type is inefficient. The overhead of lifting a single operation increases as long as the number of nested monad transformers increases.

The inefficient lifting can be avoided in our `Dsl` derivation approach, since an LDK already represents an operation for any compatible domains. Only the `cpsApply` method in the `Dsl` type class is derived, as shown in fig. 4. The performance improvement is also observed in the benchmark at section 5.2.2.

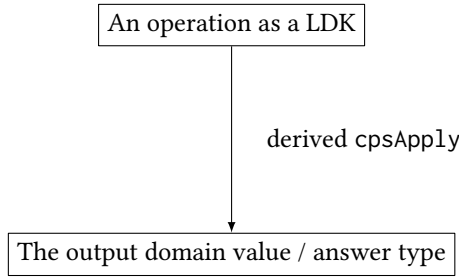


Fig. 4. Performing an operation in a Dsl block

**6.7.2 Effect handlers.** Effect handler [Kiselyov et al. 2013] is an alternative approach to monad transformers. A eDSL code block is considered as a script of Eff, which is composed of effects. A generic Eff monad type class instance composes individual effects into a larger script Eff, which is then interpreted by a stack of Handlers for each type of effect.

The effect handler approach is more efficient than monad transformer because Eff is a light-weight script instead of the underlying data structure. Lifting an atomic effect to an Eff is faster than lifting real data structures. However, this approach lacks of straightforwardness and keyword-wise extensibility in comparison to our LDK approach.

**Straightforwardness** determines whether an eDSL behaves in the native conventions of the hosting language. Effect handlers are not straightforward because Eff is an additional intermediate script, which is unable to directly collaborate with the hosting language, instead, every eDSL written Eff requires two steps of type classes, Monad and Handler, in order to produce data understandable by the hosting language, as shown in fig. 1. What is worse is, two Effs cannot invoke each other if they contain different effect stacks.

Our LDK approach is more straightforward, as only one type class Dsl is used to interpret the eDSL, as shown in fig. 2. Instead of producing indirect script Effs, LDKs directly produces the underlying data structures, which can be easily used in the hosting language. In addition, an eDSL block can be used in another eDSL block of a different domain as long as the underlying data types are compatible. For example, in LDK approach, the generatorTest function in listing 29 can internally call the returnableGenerator function even when the two functions return values of different types.

**Keyword-wise Extensibility** determines whether a new keyword or operator can be introduced into an eDSL without change the return domain type. Unfortunately, the effect handler approach is not extensible in keyword-wise because an Eff consists of a stack of effects, and each effect is specially designed for only a fix number of supported operators. An Eff is only extensible in the domain-wise by appending a new effect, which will change the type of Eff. In contrast, our LDK approach is extensible in both domain-wise and keyword-wise. (1) Domain-wise extensibility is achieved by Dsl derivation as described in section 4.3; (2) Keyword-wise extensibility is achieved by defining new LDKs that can be used in an existing domain. For example, we introduced Yield LDK in LDK-based collection comprehension in listing 57 without changing the return type nor the implementation of existing Each LDK.

## 7 HASKELL IMPLEMENTATION

We ported *Dsl.scala* to Haskell as the package *control-dsl*. The Dsl type class in Haskell is defined in listing 72. The type of cpsApply is slightly different from Scala version Dsl defined in listing 1,

as `k` is a arity-2 type parameter while `Keyword` is a first-order type parameter<sup>22</sup>. The additional type parameters for `k` improve the ability of type inference in **do** notation.

---

```
class Dsl k r a where
  cpsApply :: k r a -> (a -> r) -> r
```

---

Listing 72. Dsl type class in *control-dsl*

We also provided some helper functions for Dsl based **do**-notation, as shown in listing 73. `RebindableSyntax` language extension is required to enable those functions for **do**-notation.

---

```
(>>=) k = cpsApply k
k >> a = k >>= const a

data Return r' r a where Return :: r' -> Return r' r Void

return r = Return r >>= absurd
fail r = return (userError r)
```

---

Listing 73. Helpers for Dsl based **do**-notation

Unfortunately, the additional `r` type parameter prevents Dsl derivation, since the an LDK whose type is `k r a` can only be present in **do** blocks of type `r`. As a result, we are not able to port `derivedFunction1Dsl` to Haskell as instance `Dsl k r a => Dsl k (s -> r) a`.

To allow derived keywords, we introduced a new type class `PolyCont`, which looses the restriction in Dsl. Instead of deriving Dsl, an LDK author creates derived `PolyCont`, and finally resolves Dsl from derived `PolyCont`, as shown in listing 74.

---

```
class PolyCont k r a where
  runPolyCont :: k r' a -> (a -> r) -> r

instance PolyCont k r a => PolyCont k (s -> r) a where
  runPolyCont k f s = runPolyCont k (\a -> f a s)

instance PolyCont k r a => Dsl k r a where
  cpsApply = runPolyCont
```

---

Listing 74. `PolyCont` derivation

`Yield` and `Return` LDKs are ported to Haskell with the help of `PolyCont`, as shown in listing 75. We created another Dsl instance for monomorphic delimited continuation `Cont`, which is used to created control flow operators with nested **do**-notation, as shown in listings 76 and 77.

<sup>22</sup> We use shorter identifiers in *control-dsl* to confirms Haskell naming conventions, as shown below:

Identifiers in <i>Dsl.scala</i>	Identifiers in <i>control-dsl</i>
Keyword	k
Domain	r
Value	a
Continuation	Cont
handler	f

```
data Get r a where Get :: forall s r. Get r s

instance PolyCont Get (s -> r) s where
  runPolyCont Get f s = f s s

data Yield x r a where Yield :: x -> Yield x r ()

instance PolyCont (Yield x) [x] () where
  runPolyCont (Yield x) f = x : f ()

instance PolyCont (Return r) r Void where
  runPolyCont (Return r) _ = r
```

Listing 75. PolyCont instances for Get, Yield and Return

```
newtype Cont r a = Cont { runCont :: (a -> r) -> r }

instance Dsl Cont r a where
  cpsApply = runCont
```

Listing 76. Dsl instance for Cont

```
when :: Bool -> Cont r () -> Cont r ()
when True k = k
when False _ = Cont ($) ()
```

Listing 77. Control flow operator when

With the help of the above control flow operators, we are able to create direct style DSL in **do**-notation, as shown in listing 78

```
f = do
  Yield "foo"
  config <- Get @Bool
  when config $ do
    Yield "bar"
    return ()
  return "baz"
```

Listing 78. Nested Dsl do blocks

f is a **do** block that contains LDKs Yield, Get and Return (invoked by **return** internally). With the help of built-in PolyCont instances for those keywords, f can be used as a function that accepts a boolean parameter, as shown in listing 79

In fact, f can be any type as long as PolyCont instances for involved keywords are provided. The type can be inferred by GHC, as shown in listing 80

For example, f can be interpreted as an impure IO () (listing 82), providing the instances defined in listing 81.

---

```

1961 > f False :: [String]
1962 ["foo", "baz"]
1963
1964 > f True :: [String]
1965 ["foo", "bar", "baz"]

```

---

Listing 79. Running `f` purely in REPL

---

```

1969 > :type f
1970 f :: (PolyCont (Yield [Char]) r (),
1971       PolyCont (Return [Char]) r Void, PolyCont Get r Bool) =>
1972       r

```

---

Listing 80. The inferred type of a `do` block

---

```

1976 instance PolyCont (Yield String) (IO ()) () where
1977   runPolyCont (Yield a) = (Prelude.>=) (putStrLn $ "Yield_" ++ a)
1978 instance PolyCont Get (IO ()) Bool where
1979   runPolyCont Get f = putStrLn "Get" Prelude.>> f False
1980 instance PolyCont (Return String) (IO ()) Void where
1981   runPolyCont (Return r) _ = putStrLn $ "Return_" ++ r

```

---

Listing 81. Custom effectful instances for built-in LDKs

---

```

1985 > f :: IO ()
1986 Yield foo
1987 Get
1988 Return baz

```

---

Listing 82. Running `f` effectfully in REPL

In brief, the Haskell implementation *control-dsl* can infer type better than *Dsl.scala*, while the `do`-notation is more verbose than `!`-notation in *Dsl.scala*.

## 8 CONCLUSION

We have presented a novel approach to create direct style embedded domain specific languages that are more extensible, more straightforward and more efficient than existing monad based and continuation based solutions. The main highlights of our approaches are:

- (1) ability to define LDKs that work with existing native types, as if they are first-class features;
- (2) extensibility in both keyword-wise and domain-wise;
- (3) *Dsl* derivation, allowing an LDK to be adaptive to various domains.

The capacity of LDKs is the superset of both monads and ordinary delimited continuations, thus LDKs can be used in various domains as 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.

8.1 Future work

Two types of polymorphism are involved in this paper. We implemented a `BangNotation` Scala compiler plug-in to perform name-based CPS-transformation, which support answer type modification, or **polymorphic delimited continuation**; we introduced `Dsl` type class, which allows run an LDK as a CPS function adaptive to the answer type, or **ad-hoc polymorphic delimited continuation**. In the future, we will investigate how to represent a delimited continuation that is both polymorphic and ad-hoc polymorphic.

## 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 and Yukiyoshi Kameyama. 2007. Polymorphic delimited continuations. In *Asian Symposium on Programming Languages and Systems*. Springer, 239–254.
- Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593.
- 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. 1998. Functional unparsing. *Journal of functional programming* 8, 6 (1998), 621–625.
- O. Danvy and A. Filinski. 1989. *A Functional Abstraction of Typed Contexts*. Technical Report 89/12. DIKU, University of Copenhagen, Copenhagen, Denmark.
- Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *LISP and Functional Programming*.
- R Kent Dybvig, Simon Peyton Jones, and Amr Sabry. 2007. A monadic framework for delimited continuations. *Journal of Functional Programming* 17, 6 (2007), 687–730.
- Martin Fowler. 2010. *Domain-specific languages*. Pearson Education.
- 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.
- Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 333–343.
- Lightbend, Inc. 2017. *Akka FSM*. Lightbend, Inc. <https://doc.akka.io/docs/akka/2.5.10/fsm.html>
- Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. 1977. Abstraction mechanisms in CLU. *Commun. ACM* 20, 8 (1977), 564–576.
- 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](https://www.tensorflow.org/api_guides/python/control_flow_ops)
- Alexandru Nedelcu, Sorin Chiprian, Mihai Soloi, Andrei Oprea, Jisoo Park, Dawid Dworak, Omar Mainegra, Piotr Gawryś, A. Alonso Dominguez, Leandro Bolivar, Ryo Fukumuro, Ian McIntosh, Denys Zadorozhnyi, and Oleg Pyzhov. 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, Stéphane 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 Rumpf, 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/>
- Mary Shaw, William A Wulf, and Ralph L London. 1977. Abstraction and verification in Alphard: Defining and specifying iteration and generators. *Commun. ACM* 20, 8 (1977), 553–564.
- 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/>