A 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.

A.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 47.

```
object IntPlaceholder {
  @shift def unary_! : String = ???
  def cpsApply[Domain](f: String => Domain): Int => Domain = { i: Int =>
    f(i.toString)
  }
}

object StringPlaceholder {
  @shift def unary_! : String = ???
  def cpsApply[Domain](f: String => Domain): String => Domain = f
}

def f1 = "Hello␣World!"
def f2 = "Hello␣" + !StringPlaceholder + "!"
def f3 = "The␣value␣of␣" + !StringPlaceholder + "␣is␣" + !IntPlaceholder + "."

println(f1) // Output: Hello World!
println(f2("World")) // Output: Hello World!
println(f3("x")(3)) // Output: The value of x is 3.
```

Listing 47. 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 48.

```
// The type of f1 is inferred as `String`
def f1 = "Hello␣World!"

// The type of f2 is inferred as `String => String`
def f2 = StringPlaceholder.cpsApply { tmp =>
  "Hello␣" + tmp + "!"
}

// The type of f3 is inferred as `String => Int => String`
def f3 = StringPlaceholder.cpsApply { tmp0 =>
  IntPlaceholder.cpsApply { tmp1 =>
    "The␣value␣of␣" + tmp0 + "␣is␣" + tmp1 + "."
  }
}
```

Listing 48. 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.

A.2 The prefix problem

The `prefix` problem introduced in [Asai and Kameyama 2007] is a problem that requires polymorphic delimited continuations, which is a CPS function whose answer type can be modified, i.e. `(A => B) => C` where `B` and `C` differ. We provide a `PolymorphicShift` LDK to perform `shift` control operator for polymorphic delimited continuations, as defined in listing 49. Note that `PolymorphicShift` is not interpreted by `Dsl`, hence it is not ad-hoc polymorphic.

```
final case class PolymorphicShift[A, B, C](cpsApply: (A => B) => C) {
  @shift def unary_! : A = ???
}

implicit def implicitPolymorphicShift[A, B, C](cpsApply: (A => B) => C) =
    PolymorphicShift(cpsApply)
```

Listing 49. The definition of PolymorphicShift

The solution to `prefix` problem uses "underscore trick" along with `PolymorphicShift`, as shown in listing 50.

```
def visit[A](lst: List[A]): (List[A] => List[A] @reset) => List[List[A]] = _ {
  lst match {
    case Nil =>
      !{ (h: List[A] => List[A]) =>
        Nil
      }
    case a :: rest =>
      a :: !{ (k: List[A] => List[A] @reset) =>
        k(Nil) :: k(!visit(rest))
      }
  }
}

def prefix[A](lst: List[A]) = !visit(lst)

// Output: List(List(1), List(1, 2), List(1, 2, 3))
println(prefix(List(1, 2, 3)))
```

Listing 50. 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.

A.3 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 [21]. Unlike state monads, our LDK-based approach is more straightforward, and supports multiple mutable states without manually lifting.

A.3.1 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 51 and 52, respectively.

```
case class Get[S]() extends Keyword[Get[S], S]
```

Listing 51. The definition of `Get` LDK

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

Listing 52. The definition of `Put` LDK

Listing 53 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: O
println(upperCasedLastCharacter("foo"))
```

Listing 53. Using `Get` and `Put` in a unary function

The `Dsl` instances for `Get` and `Put` used in `upperCasedLastCharacter` are shown in listings 54 and 55. 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)
    }
  }
```

[21] 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 54. 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 55. The Dsl instance for Put LDK

Traditionally, the data type of state monad is an opaque type alias of `S => (S, A)`, which is more complicated than our domain type `S => 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.

A.3.2 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 56, 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 56. 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, which is defined in listing 34.

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.

A.4 Asynchronous programming

With the help of Dsl derivation, a complex DSL can be composed of simple features. In this section we will present a sophisticated implementation of asynchronous task, which is composed of three independent features: (1) asynchronous result handling (2) exception handling (3) stack safety, as defined in listing 57. 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 57. 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 !! ... !! L_0 !! Throwable !! R_0 !! ... !! R_i), and some derived Dsls to optimize tail calls as trampolines, which support domains whose types match the pattern of (TailRec[...] !! R_0 !! ... !! R_i), where $L_0 \dots L_i$ and $R_0 \dots R_i$ are arbitrary number of types [22]. Therefore, Dsl instances for Task are composed from these orthogonal features.

In appendix A.4.1, we will present how to create an asynchronous HTTP client from Task; in appendix A.4.2, 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.

A.4.1 An asynchronous HTTP client. Listing 58 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.")
  }
}
```

[22] Those Dsls are implemented in the Dsl derivation technique described in section 5. Check the artifact for the complete implementation

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

def asynchronousHttpClient(url: URL): Task[String] = _ {
  val socket = AsynchronousSocketChannel.open()
  try {
    val port = if (url.getPort == -1) 80 else url.getPort
    val address = new InetSocketAddress(url.getHost, port)
    !Connect(socket, address)
    val request = ByteBuffer.wrap(s"GET␣${url.getPath}␣HTTP/1.1\r\nHost:${url.
        getHost}\r\nConnection:Close\r\n\r\n".getBytes)
    !writeAll(socket, request)
    val MaxBufferSize = 100000
    val response = ByteBuffer.allocate(MaxBufferSize)
    !readAll(socket, response)
    response.flip()
    io.Codec.UTF8.decoder.decode(response).toString
  } finally {
    socket.close()
  }
}
```

Listing 58. 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 !! ...)` [23].

`!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 59.

```
val httpResponse = asynchronousHttpClient(new URL("http://example.com/")).
    blockingAwait
httpResponse should startWith("HTTP/1.1␣200␣OK")
```

Listing 59. Using the example HTTP client

A.4.2 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 60.

[23] Check the artifact for complete implementation.

```
val Urls = Seq(
  new URL("http://example.com/"),
  new URL("http://example.org/")
)
def parallelTask: Task[Seq[String]] = {
  val url: URL = !Fork(Urls)
  val content: String = !httpClient(url)
  !Return(content)
}

val Seq(fileContent0, fileContent1) = parallelTask.blockingAwait
assert(fileContent0.startsWith("HTTP/1.1 200 OK"))
assert(fileContent1.startsWith("HTTP/1.1 200 OK"))
```

Listing 60. 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.

A.4.3 Modularity and performance. Our approach achieved both better modularity and better performance than previous implementation.

Most of previous asynchronous programing libraries, including Scala Future [Haller et al. 2012], Monix [Nedelcu et al. 2017], and Cats effects [Typelevel 2017], are built from a solid implementation, along with some callback scheduler for custom behaviors. In contrast, our approach separate atomic features into independent `Dsl` type classes.

Scalaz Concurrent [Yoshida et al. 2017] or other monad transformer [Liang et al. 1995] based approach can separate asynchronous programing features into monad of asynchronous handling and monad transformer of exception handling. Even though, trampolines are not able to implemented as monad transformers, as a result, intrusive code for trampolines must be present in each monad instance and monad transformer instance, or the call stack may overflow. In contrast, our approach allows non-intrusive `Dsl` derivation for `TailRec[Unit]`, then the ability of stack safety will be added on previously stack unsafe `Dsl`s.

According to our benchmarks in appendix B, on HotSpot JVM, our `Task` implementation is much faster than monad transformer based implementations, and has similar performance in comparison to solid implementations. On GraalVM, our `Task` implementation is faster than all other implementations.

A.5 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.

A.5.1 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 61.

```
def compositeNumbersBelow(n: Int) = (for {
```

```
  i <- 2 until math.ceil(math.sqrt(n)).toInt
  j <- 2 * i until n by i
} yield j).to[Set]
```

Listing 61. 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 61 can be rewrite to listing 62 with the help of the `Each` LDK, or listing 63 after removing the temporary variable `j`.

```
def compositeNumbersBelow(n: Int): Set[Int] = Set {
  val i = !Each(2 until math.ceil(math.sqrt(n)).toInt)
  val j = !Each(2 * i until n by i)
  j
}

// Output: Set(10, 14, 6, 9, 12, 8, 4)
println(compositeNumbersBelow(15))
```

Listing 62. Calculating all composite numbers below n with `Each` LDK

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

Listing 63. 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 64.

```
def heterogeneous = List { !Each(Array("foo", "bar", "baz")) + !Each("LDK") }

// Output: List(fooL, fooD, fooK, barL, barD, barK, bazL, bazD, bazK)
println(heterogeneous)
```

Listing 64. LDK-based heterogeneous collection comprehension based on `Array` and `String`

A.5.2 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 65.

```
def primeNumbersBelow(maxNumber: Int) = List {
  val compositeNumbers = compositeNumbersBelow(maxNumber)
  val i = !Each(2 until maxNumber)
  if (compositeNumbers(i)) !Continue
  i
```

```
}

// Output: List(2, 3, 5, 7, 11, 13)
println(primeNumbersBelow(15))
```

Listing 65. 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.

A.5.3 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 60 by Each, those URLs will be fetched sequentially, as shown in listing 66.

```
def sequentialTask: Task[Seq[String]] = {
  val url: URL = !Each(Urls)
  val content: String = !httpClient(url)
  !Return(content)
}
```

Listing 66. Using HTTP client in parallel

A.5.4 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 67.

```
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 67. Build a command-line by using generator and collection comprehension together

B 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 appendix A.4, 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 *Dsl.scala*
Scala Future [Haller et al. 2012]	Scala Async [Haller and Zaugg 2013]
Scala Continuation library [Rompf et al. 2009]	Scala Continuation compiler plug-in
Monix tasks [Nedelcu et al. 2017]	`for`-comprehension
Cats effects [Typelevel 2017]	`for`-comprehension
Scalaz Concurrent [Yoshida et al. 2017]	`for`-comprehension

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

B.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.

B.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 68.

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 69 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.

```
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 68. The most efficient implementation of sum based on ordinary CPS function

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

Listing 69. The most efficient implementation of sum based on Scala Futures

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

B.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 70 and 71). For other effect systems, we refactored the previous sum benchmarks to use Scala

Async, Scala Continuation's `@cps` annotations, and **`for`**-comprehension, respectively (listings 72 to 77).

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

Listing 70. Left-associated sum based on LDKs of *Dsl.scala*

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

Listing 71. Right-associated sum based on LDKs of *Dsl.scala*

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

Listing 72. Left-associated sum based on Scala Async

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:

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`.

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

Listing 73. Right-associated sum based on Scala Async

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

Listing 74. Left-associated sum based on Scala Continuation plug-in

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

Listing 75. Right-associated sum based on Scala Continuation plug-in

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

Listing 76. 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 77. Right-associated sum based on **for**-comprehension

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

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

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

B.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.

B.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 78 shows the benchmark code for Scala Future.

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

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

Listing 78. Cartesian product for Scala Future, based on Scalaz's `traverseM`

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 79:

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

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

Listing 79. Cartesian product for ordinary CPS functions, based on *Dsl.scala*

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.

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

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

B.2.2 The performance of collection manipulation in direct style DSLs. We then refactored the benchmarks to direct style DSLs. Listing 80 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.

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

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

We also merged `cellTask` and `listTask` in the *Dsl.scala* version of benchmark as shown in listing 81.

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

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`.

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

Listing 81. Cartesian product for ordinary CPS functions, in one function

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

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 Dyvbig, Simon Peyton Jones, and Amr Sabry. 2007. A monadic framework for delimited continuations. *Journal of Functional Programming* 17, 6 (2007), 687–730.

Andrzej Filinski. 1994. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 446–457.

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

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

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

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

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

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

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

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/