Direct Style Scala

Martin Odersky
EPFL

Scalar Conference
March 24, 2023

Shifting Foundations

Trends

- Widespread support for async/await
- Runtimes get better support for fibers or continuations.

Examples

- Goroutines,
- Project Loom in Java,
- Kotlin coroutines,
- OCaml or Haskell delimited continuations,
- Research languages such as Effekt, Koka

Thesis of this talk

- This will deeply influence libraries and frameworks
- It's very attractive now to go back to direct style.

Shifting Foundations

Trends

- Widespread support for async/await
- Runtimes get better support for fibers or continuations.

Examples

- Goroutines,
- Project Loom in Java,
- Kotlin coroutines,
- OCaml or Haskell delimited continuations,
- Research languages such as Effekt, Koka

Thesis of this talk

- This will deeply influence libraries and frameworks
- It's very attractive now to go back to direct style.

How will this influence Scala in the future?

1. There will likely be native foundations for direct-style reactive programming
   - Delimited continuations on Scala Native
   - Fibers on latest Java
   - Source or bytecode rewriting for older Java, JS
2. This will enable new techniques for designing and composing software
3. There will be a move away from monads as the primary way of code composition.

Building a Direct-Style Stack

- First step: Boundary/break
- Error handling
- Suspensions
- Concurrency library design built on that

Building a Direct-Style Stack

- First step: Boundary/break

  `(shipped)`
- Error handling

  `(enabled)`
- Suspensions

  `(wip)`
- Concurrency library design built on that

  `(wip)`

Warmup: Boundary/break

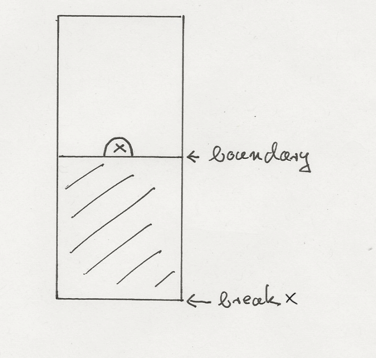
A cleaner alternative to non-local returns (which will go away)

```
def firstIndex[T](xs: List[T], elem: T): Int =
  boundary:
    for (x, i) <- xs.zipWithIndex do
      if x == elem then break(i)
    -1
```

`boundary` establishes a boundary

`break` returns with a value from it.

Stack View



API

```scala
package scala.util

object boundary:
  final class Label[-T]

  def break[T](value: T)(using label: Label[T]): Nothing =
    throw Break(label, value)

  inline def apply[T](inline body: Label[T] ?=> T): T = ...
end boundary
```

To `break`, you need a `label` that represents the boundary.

In a sense, `label` is a *capability* that *enables* to `break`.

(This is a common pattern)

Implementation

The implementation of `break` produces efficient code.

- If `break` appears in the same stackframe as its `boundary`, use a jump.
- Otherwise use a fast exception that does not capture a stack trace.

A stack trace is not needed since we know the exception will be handled (*)

(*) To be 100% sure, this needs capture checking.

Implementation

The implementation of `break` produces efficient code.

- If `break` appears in the same stackframe as its `boundary`, use a jump.
- Otherwise use a fast exception that does not capture a stack trace.

A stack trace is not needed since we know the exception will be handled (*)

(*) To be 100% sure, this needs capture checking.

Stage 2: Error handling

`boundary/break` can be used as the basis for flexible error handling. For instance:

```
def firstColumn[T](xss: List[List[T]]): Option[List[T]] =
  optional:
    xss.map(_.headOption.?)
```

Optionally, returns the first column of the matrix `xss`.

Returns `None` if there is an empty row.

Error handling implementation

`optional` and `?` on options can be implemented quite easily on top of `boundary/break`:

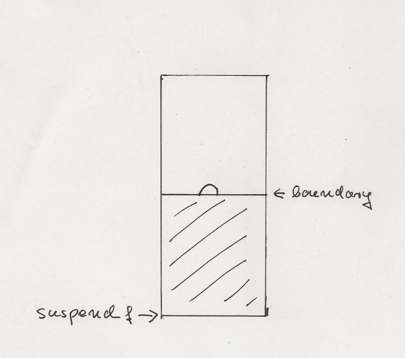
```
object optional:
  inline def apply[T](inline body: Label[None.type] ?=> T)
    : Option[T] = boundary(Some(body))

  extension [T](r: Option[T])
    inline def ? (using label: Label[None.type]): T = r match
      case Some(x) => x
      case None => break(None)
```

Analogous implementations are possible for other result types such as `Either` or a Rust-like `Result`.

My ideal way of error handling would be based on `Result` + `?`.

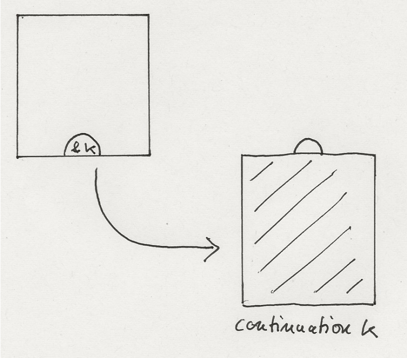
Stage 3: Suspensions

Question: What if we could store the stack segment between a `break` and its `boundary` and re-use it at some later time?



Suspensions

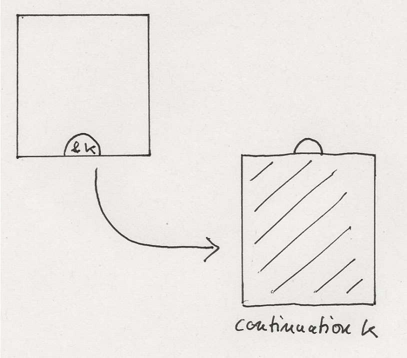
Question: What if we could store the stack segment between a `break` and its `boundary` and re-use it at some later time?



This is the idea of *delimited continuations*.

Suspensions

Question: What if we could store the stack segment between a `break` and its `boundary` and re-use it at some later time?



This is the idea of *delimited continuations*.

Suspension API

```
class Suspension[-T, +R]:
  def resume(arg: T): R = ???

def suspend[T, R](body: Suspension[T, R] => R)(using Label[R]): T
```

Suspensions are quite powerful.

They can express at the same time *algebraic effects* and *monads*.

Generators

Python-style generators are a simple example of algebraic effects.

```
def example = Generator:
  produce(”We’ll give you all the numbers divisible by 3 or 2”)
  for i <- 1 to 1000 do
    if i % 3 == 0 then
      produce(s”$i is divisible by 3”)
    else if i % 2 == 0 then
      produce(s”$i is even”)
```

Here, `Generator` is essentially a simplified `Iterator`

```
trait Generator[T]:
  def nextOption: Option[T]
```

Algebraic Effects

Task: Build a `generate` implementation of `Generator`, so that one can compute the `leafs` of a `Tree` like this:

```
enum Tree[T]:
  case Leaf(x: T)
  case Inner(xs: List[Tree[T]])

def leafs[T](t: Tree[T]): Generator[T] =
  generate:                                         // effect scope
    def recur(t: Tree[T]): Unit = t match
      case Tree.Leaf(x)    => produce(x)            // effect
      case Tree.Inner(xs) => xs.foreach(recur)
    recur(t)
```

Generator Implementation

```scala
trait Produce[-T]:
  def produce(x: T): Unit

def generate[T](body: Produce[T] ?=> Unit) = new Generator[T]:
  def nextOption: Option[T] = step()

  var step: () => Option[T] =
```

The Step Function

```scala
trait Produce[-T]:                       // effect type
  def produce(x: T): Unit

def generate[T](body: Produce[T] ?=> Unit) = new Generator[T]:
  def nextOption: Option[T] = step()

  var step: () => Option[T] = () =>
    boundary:
      given Produce[T] with              // handler
        def produce(x: T): Unit =
          suspend[Unit, Option[T]]: k =>
            step = () => k.resume(())
            Some(x)
      body
      None
```

Summary: Algebraic Effects

Effects are *methods* of effect traits

Handlers are *implementations* of effect traits

- They are passed as *implicit parameters*.
- They can *abort* part of a computation via break
- They can also *suspend* part of a computation as a continuation and resume it later.

Implementing Suspensions

There are several possibilities:

- Directly in the runtime, as shown in the designs
- On top of fibers (requires some compromises)
- By bytecode rewriting (e.g. Quasar, `javactrl`)
- By source rewriting

Suspensions and Monads:

- Wadler (1993): Continuations can be expressed as a monad.

  "*Haskell is the essence of ML*"

- Filinski (1994): Every monad can be expressed in direct style using just delimited continuations.

  "*ML is the essence of Haskell*"

My take: designs based on continuations are simpler to compose than monads.

Suspensions and Monads:

- Wadler (1993): Continuations can be expressed as a monad.

  "*Haskell is the essence of ML*"

- Filinski (1994): Every monad can be expressed in direct style using just delimited continuations.

  "*ML is the essence of Haskell*"

My take: designs based on continuations are simpler to compose than monads.

Direct-Style Futures

With `suspend`(*), we can implement lightweight and universal `await` construct that can be called anywhere.

This can express simple, direct-style futures.

```
val sum = Future:
  val f1 = Future(c1.read)
  val f2 = Future(c2.read)
  f1.value + f2.value
```

Structured concurrency: Local futures `f1` and `f2` complete before `sum` completes. This might mean that one of them is cancelled if the other returns with a failure.

(*) Loom-like fibers would work as well.

Compare with Status Quo

```
val sum =
  val f1 = Future(c1.read)
  val f2 = Future(c2.read)
  for
    x <- f1
    y <- f2
  yield x + y
```

Composition of futures is monadic
but creation isn't, which is a bit awkward.

A Strawman

`lampepfl/async` is an early stage prototype of a modern, low-level concurrency library in direct style.

Main elements

- **Futures:** the primary *active* elements.
- **Channels:** the primary *passive* elements.
- **Async Sources** Futures and Channels both implement a new fundamental abstraction: an *asynchronous source*.
- **Async Contexts** An async context is a *capability* that allows a computation to suspend while waiting for the result of an async source.

Link: `github.com/lampepfl/async`

Futures

The `Future` trait is defined as follows:

```
trait Future[+T] extends Async.Source[Try[T]], Cancellable:
  def result(using Async): Try[T]
  def value(using Async): T = result.get
```

The `result` method can be defined like this:

```
  def result(using async: Async): T = async.await(this)
```

`async` is a *capability* that allows to suspend in an `await` method.

Futures

The `Future` trait is defined as follows:

```
trait Future[+T] extends Async.Source[Try[T]], Cancellable:
  def result(using Async): Try[T]
  def value(using Async): T = result.get
```

The `result` method can be defined like this:

```
  def result(using async: Async): T = async.await(this)
```

`async` is a *capability* that allows to suspend in an `await` method.

Async

The `Async` trait is defined as follows:

```
trait Async:
  def await[T](src: Async.Source[T]): T

  def scheduler: ExecutionContext
  def group: CancellationGroup
  def withGroup(group: CancellationGroup): Async
```

`await` gets the (first) element of an `Async.Source`.

It suspends if necessary.

Async.Source

- Futures are a particular kind of an async source. (Other implementations come from channels).
- Async sources are the primary means of communication between asynchronous computations,
- They can be composed in interesting ways.

For instance, `map` and `filter` are provided:

```
extension [T](s: Source[T])
  def map[U](f: T => U): Source[U]
  def filter(p: T => Boolean): Source[T]
```

Async.Source

- Futures are a particular kind of an async source. (Other implementations come from channels).
- Async sources are the primary means of communication between asynchronous computations,
- They can be composed in interesting ways.

For instance, `map` and `filter` are provided:

```
extension [T](s: Source[T])
  def map[U](f: T => U): Source[U]
  def filter(p: T => Boolean): Source[T]
```

Races

A race passes on the first of several sources:

```
def race[T](sources: Source[T]*): Source[T]
```

Higher-level operation:

```
def either[T1, T2](src1: Source[T1], src2: Source[T2])
    : Source[Either[T, U]] =
  race(
    src1.map(Left(_)),
    src2.map(Right(_))
  )
```

Structured Concurrency

It's now easy to implement `zip` and `alt` on futures:

```
extension [T](f1: Future[T])
  def zip[U](f2: Future[U])(using Async): Future[(T, U)] =
    Future:
      await(either(f1, f2)) match
        case Left(Success(x1))    => (x1, f2.value)
        case Right(Success(x2))   => (f1.value, x2)
        case Left(Failure(ex))    => throw ex
        case Right(Failure(ex))   => throw ex
```

Structured Concurrency

It's now easy to implement `zip` and `alt` on futures:

```
extension [T](f1: Future[T])
  def alt(f2: Future[T])(using Async): Future[T] =
    Future:
      await(either(f1, f2)) match
        case Left(Success(x1))     => x1
        case Right(Success(x2))    => x2
        case Left(_: Failure[?])  => f2.value
        case Right(_: Failure[?]) => f1.value
```

Why Futures & Channels?

`Futures`: The simplest way to get parallelism

- Define a computation
- Run it in parallel
- Await the result when needed

`Channels`: The canonical way of communication between computations.

Both are instances of asynchronous sources

Why not Coroutines?

Often, coroutines (in the sense of CSP or goroutines) are used instead of futures to work with channels.

But:

- We need to be able to wait for a coroutine's termination.
- We need to handle any exceptions in the coroutine on the outside.

Both are achieved by using a `Future[Unit]`.

So no different abstractions are needed.

Why `Try` as Result Type ?

Natural solution if the language supports exceptions.

But there is a common complaint for current futures:

- If the error type is fixed to be `Exception` it becomes awkward to handle other errors.

For instance, how would you implement this function?

```
def acrobatics(xs: List[Future[Result[T, E]]])
    : Future[Result[List[T], E]] =
```

Why `Try` as Result Type ?

Natural solution if the language supports exceptions.

But there is a common complaint for current futures:

- If the error type is fixed to be `Exception` it becomes awkward to handle other errors.

For instance, how would you implement this function?

```
def acrobatics(xs: List[Future[Result[T, E]]])
    : Future[Result[List[T], E]] =
```

Why an ErrorType Fixed to Try?

Natural solution if the language supports exceptions.

But there is a common complaint for current futures:

- If the error type is fixed to be `Exception` it becomes awkward to handle other errors.

For instance, how would you implement this function?

```
def acrobatics(xs: List[Future[Result[T, E]]])
    : Future[Result[List[T], E]] =
```

Why an ErrorType Fixed to Try?

Natural solution if the language supports exceptions.

But there is a common complaint for current futures:

- If the error type is fixed to be `Exception` it becomes awkward to handle other errors.

New direct style abstractions don't have that problem anymore!

```
def acrobatics(xs: List[Future[Result[T, E]]])
    : Future[Result[List[T], E]] =
  Future:
    Result:
      xs.map(_.value.?)
```

Simple compositions, no traverse or lift is needed.

Conclusion

Direct style has lots to offer

Suspensions can express every monad, but, provide more flexible composition.

This gives completely new possibilities to express practical foundations for concurrency and async I/O.

The future will be interesting...

Thank You

Conclusion

Direct style has lots to offer

Suspensions can express every monad, but, provide more flexible composition.

This gives completely new possibilities to express practical foundations for concurrency and async I/O.

The future will be interesting...

Thank You