Scala 3: What Is “Direct Style”?

Dean Wampler · Follow

Published in Scala 3

7 min read · Jun 4, 2023

Scala uses *monads* extensively for many operations with non-trivial *effects*, such as those involving asynchronous computation, including I/O. The idea of **direct style** is to allow writing code with non-trivial *effects* more simply without the boilerplate of monads.

Vista Tower, Chicago, © 2023, Dean Wampler

> *I recommend viewing this Martin Odersky talk about the on-going work on direct style. The following discussion is based on it.*

Motivation

Consider the example of `Future`s. We currently use them like this:

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

The `for` comprehension invokes `map` and `flatMap` to wait on the futures and extract the values.

A new **direct style** implementation of futures would be used liked this:

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

In both cases, the two futures are executing in parallel, which might take a while. The sum of the returned values is returned in a new future. The `value` method returns the result of a future once it is available or it throws an exception if the future returns a `Failure`. However, an implementation based on the `boundary` and `break` mechanism discussed below, this exception would be caught by the `Future` library, used to cancel the other running `Future`, if one is still running, and then return a `Failure` future for `sum`.

So, direct style simplifies code, making it is easier to write and understand, plus it enables cleaner separation of concerns, such as handling timeouts and failures in futures, and it cleanly supports composability, which monads don't provide unless you use cumbersome machinary such as *monad transformers*.

Implementing Direct Style in Scala

In Martin's talk, he discusses four aspects of building support for direct style in Scala:

1. `boundary` and `break` — now available in Scala 3.3.0.
2. Error handling — enabled, but not yet used in the library, which is still the Scala 2.13 library.
3. Suspensions — work in progress.
4. Concurrent library design built on the above — work in progress.

Let's discuss these topics.

`boundary` and `break`

This mechanism is defined with a new addition to the API, `scala.util.boundary$`. It provides a cleaner alternative to non-local returns.

- `boundary` defines a context for a computation.
- `break` returns a value from within the enclosing boundary.

Here is an example from Martin's talk, which is also discussed in the API. I have added this example and some tests in the *Programming Scala* code examples here and here.

```
import scala.util.boundary, boundary.break

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

This is one way to return the index for an element found in a collection or -1 if not found. The `boundary` defines the scope where a non-local return may be invoked. If the desired element `elem` is found, then we call `break` to return the index. This

breaks out of the `for` comprehension, too, since there is no point in continuing. If `elem` isn't found, `-1` is returned through the normal return path.

Here is a slightly simplified implementation. (The full 3.3.0 source code is here):

```
package scala.util

object boundary:
  final class Label[-T]

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

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

  final class Break[T] (val label: Label[T], val value: T) extends RuntimeExc

end boundary
```

In the `firstIndex` example above, the `boundary:` line is short for `boundary.apply:` with the indented code below it passed as the body.

Well actually, the `block` passed to `boundary.apply` is a *context function* that is called within `boundary.apply` to return the block of code shown in the example. Note the `final class Label[T]` declaration in `boundary`. Users don't define `Label` instances themselves. Instead, this is done inside the implementation (not shown) of `boundary.apply` to provide the *capability* of doing a non-local return. Using a `Label` in this way prevents the user from trying to call `break` without an enclosing `boundary`.

Rephrasing all that, we don't want users to call `break` without an enclosing `boundary`. That's why `break` requires an in-scope given instance of `Label`, which the implementation of `boundary.apply` creates before it calls the code block you provide. If your code block calls `break`, a given `Label` will be in-scope.

You don't have to do anything to create the context function passed to `boundary.apply`. It is synthesized from your block of code automatically when

`boundary.apply` is called.

Look at `firstIndex` again. If we do find an element that is equal to `elem`, then we call break to return the index `i` from the `boundary`. If we don't find the element, then a "normal" return is used to return `-1`. We never reach the `-1` expression if `break` is called.

The `boundary` and `break` mechanism is a better alternative to `scala.util.control.NonLocalReturns` and `scala.util.control.Breaks` which are deprecated as of Scala 3.3.0. The new mechanism is easier for developers to use and it adds the following additional benefits:

- The implementation uses a new `scala.util.boundary$.Break` class that derives from `RuntimeException`. Therefore, non-local `break`s are logically implemented as non-fatal exceptions and the implementation is optimized to suppress unnecessary stack trace generation. Stack traces are unnecessary because we are handling these exceptions, not barfing them on the user!
- Better performance is provided when a `break` occurs to the enclosing scope inside the same method (i.e., the same stack frame), where it can be rewritten to a jump call.

Error Handling

Next Martin discussed new ways of handling errors that leverage `boundary` and `break`, and are partly inspired by the way Rust handles errors.

He used a simpler example of trying a computation that will hopefully return a result wrapped in a `Some`, but if it can't succeed, then it will return `None`. Hence, an "optional" result, if you will. (This implementation is also now in the book's code examples.)

```
import scala.util.boundary, boundary.break, boundary.Label

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)
```

We lose all information about *why* it failed. A proper error handling feature should retain this information.

`optional.apply` defines a boundary by calling `boundary.apply`. The reason the `Label` is typed with `None.type`, is because if and when we call `break`, it is because we are returning `None` to the boundary. We don't break if we successfully computed something to return in a `Some`, which will be returned normally.

Inspired by a similar-looking construct in Rust, if you have an `Option` instance, calling the new extension method `?` on it will either return the enclosed object or call `break` to return `None` out of the enclosing boundary. So, while the `?` method looks more generally useful for deconstructing an `Option`, it really requires us to decide what to do when it is a `None`. Here, we are inside a boundary and we use a break. Let's look at an example to understand how it used.

Suppose you have a sequence (rows) of sequences (columns). You want to return a new sequence with just the first column, i.e., a sequence with the first element in each row. Because a row might be empty, we must wrap the returned sequence in an `Option`. If any row is empty, we'll return `None` for the whole thing:

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

The concision is very nice, but it requires some unpacking to understand what's happening. `optional` defines the boundary, which will hopefully see a `Some[Seq[T]]` returned or, worst case, a `None`.

We map over the input sequence and for each nested sequence, we get the `headOption`, but then immediately extract the element using the new extension method `?`. The `map` iteration will continue as long as those calls to `?` return an

instance of `T`, meaning the nested sequences are not empty. However, the first time an empty sequence is found, the implementation of `?` will call `break`, immediately terminating the `map` iteration and returning a `None` out of the `optional` block.

Here's what to expect:

```
val xssSome = List(List(0), List(1,0), List(2,1,0), List(3,2,1,0))
val xssNone = List(List(0), Nil, List(2,1,0), List(3,2,1,0))
assert(firstColumn(xssSome) == Some(List(0,1,2,3)))
assert(firstColumn(xssNone) == None)
```

To really appreciate this example, try writing the same method without the `boundary` and `break` mechanism. It's certainly doable, but more tedious!

Suspensions and a New Concurrency Library

Back to the futures :) , `boundary` and `break` can be used for adding new concurrency abstractions to Scala following a direct style, like the `Futures` example above, as well *continuations,* which Martin is calling *suspensions*. I won't all discuss the details he covered here, but I will mention a few topics.

The implementation of new concurrency features is not trivial for Scala, because:

1. Scala now runs on three platforms: JVM, JavaScript, and native.
2. Even on the JVM, using the new lightweight fibers coming in Project Loom would only be available to users on the most recent JVMs (19 and later).

Besides writing custom implementations for each of these scenarios, other possible implementations might use source or bytecode rewriting.

Work has started in the lampepfl/async repo, a "strawman" for ideas, both for conceptual abstractions for concurrency (like a new `Future` type), as well as implementations. The plan is for this repo to evolve into a new concurrency library for Scala.

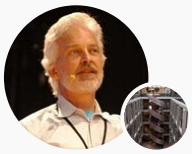
Final Thoughts

I've always liked Scala for the principled and deeply-considered approaches Martin and his team have taken to fundamental language and feature design. All programming language communities have ad hoc and "good enough" implementations for effects, like concurrency. Scala has very thoughtful systems for these purposes. However, we need to continue making it easier and more robust for people to write such code, which is commonplace and can't be reserved for the few elite programmers.

For these reasons, Scala's emerging **direct style** is a welcome trend.

> For a concise summary of the more "mainstream" notable changes in Scala 3, see my Scala 3 Highlights page.
>
> See Programming Scala, Third Edition for a comprehensive introduction to Scala 3, including details on how to migrate from Scala 2.

Scala 3 Programming Scala



Written by Dean Wampler

851 Followers · Editor for Scala 3

The person who is wrong on the Internet. ML/AI and FP enthusiast. Engineering Director, watsonx.ai at IBM Research. Speaker, author, pretend photographer.

More from Dean Wampler and Scala 3

Dean Wampler in Distributed Computing with Ray

Ray for the Curious

Dean Wampler, December 19, 2019

10 min read · Dec 19, 2019

321

Dean Wampler in Scala 3

Scala 3: Infix Operator Notation

For a long time, Scala has supported a useful “trick” called infix operator notation. If a method takes a single argument, you can call it...

3 min read · Mar 28, 2021

43

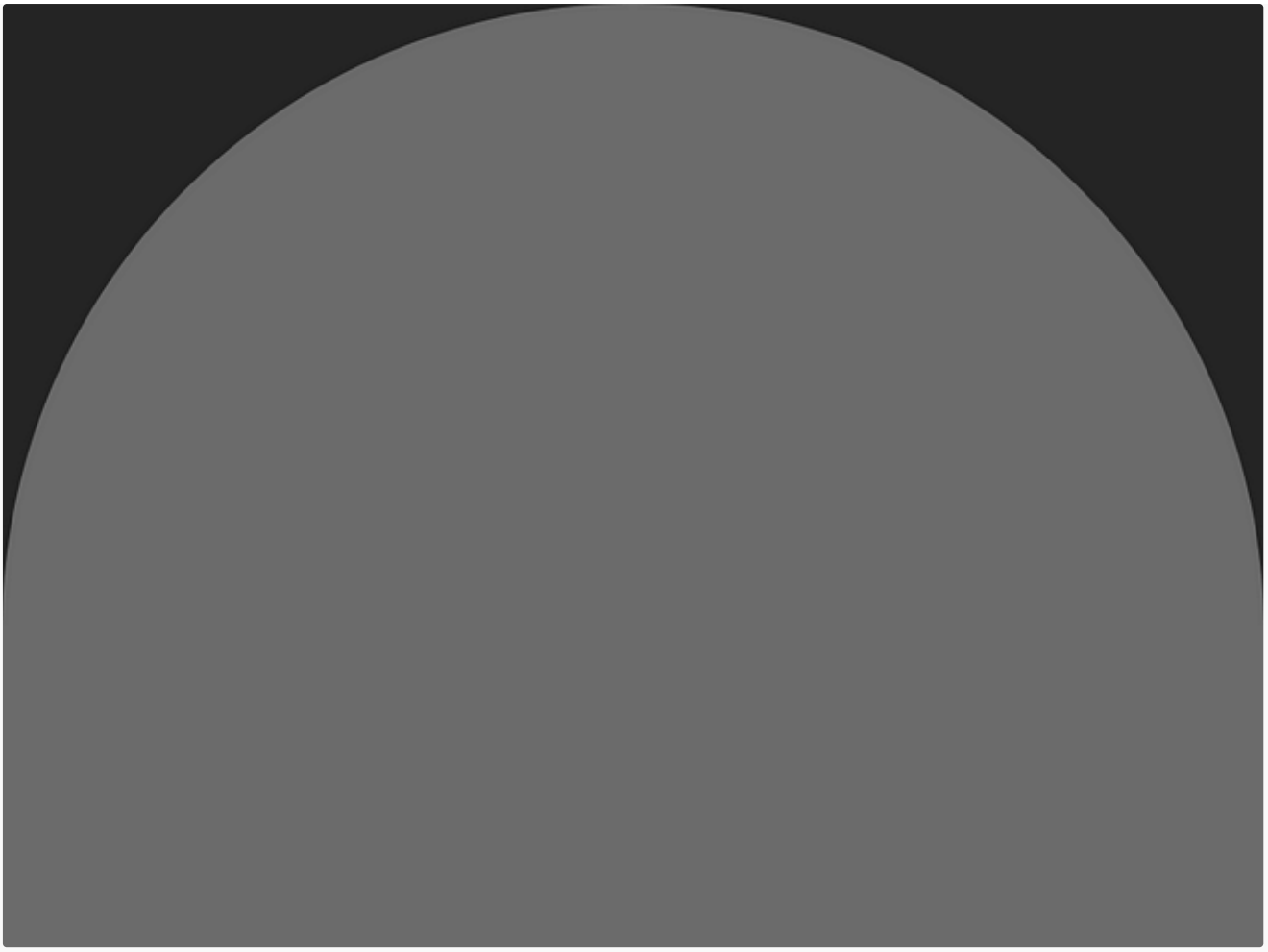
Dean Wampler in Scala 3

Scala 3: Dependent Types, Part I

Scala 3 expands on the type-level computing you can do at compile time. This post starts a discussion of dependent types.

5 min read · Jan 3, 2021

136 3



Dean Wampler in Distributed Computing with Ray

Ray Tips and Tricks, Part I—ray.wait

This series of posts provides an expanded update for a RISELab post last year on tips and tricks for using Ray effectively.

5 min read · Jan 28, 2020

97

Recommended from Medium

Tim Evdokimov

Running Cats Effect on Virtual Threads of JDK21

Cats Effect is an amazing piece of high-end machinery, enabling clear separation of effects and logics for complex concurrent asynchronous...

4 min read · Dec 31, 2023

Dotan Nahum

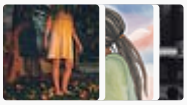
Introduction to Loco: the “Rust on Rails”

Loco is a Web or API framework for Rust: a “Rust on Rails”. Strongly inspired by Rails, it contains everything you need to go from side...

13 min read · 6 days ago

151 2

Lists



Staff Picks

591 stories · 780 saves

Stories to Help You Level-Up at Work

19 stories · 497 saves

Self-Improvement 101

20 stories · 1408 saves

Productivity 101

20 stories · 1290 saves

C. L. Beard in OpenSourceScribes

How Discord Moved from Go to Rust

And Why Rust was a Better Choice—A Lesson in Ownership Rules

· 5 min read · Sep 21, 2023

2

Rebecca Jean T.

Betelgeuse: The Great Dimming and When it Might Supernova

· 7 min read · Feb 20, 2024

loudsilence

Getting Started with the thiserror Crate for Rust

Rust is a powerful and expressive language that offers many features for writing reliable and efficient code. One of these features is its...

3 min read · 5 days ago

1

mohamed mahmoud habib

The Future of Social Psychology in the Era of Artificial Intelligence

Social psychology is among the crucial branches of psychology that focuses on studying the impact of social factors on human behavior and...

4 min read · Dec 28, 2023