Direct Style Effect Systems

The Print[A] Example

A Comprehension Aid

=>

?=>

Part 1

based on

Direct-style Effects Explained

Noel Welsh
@noelwelsh

Context is King

Adam Warski
@adamwarski

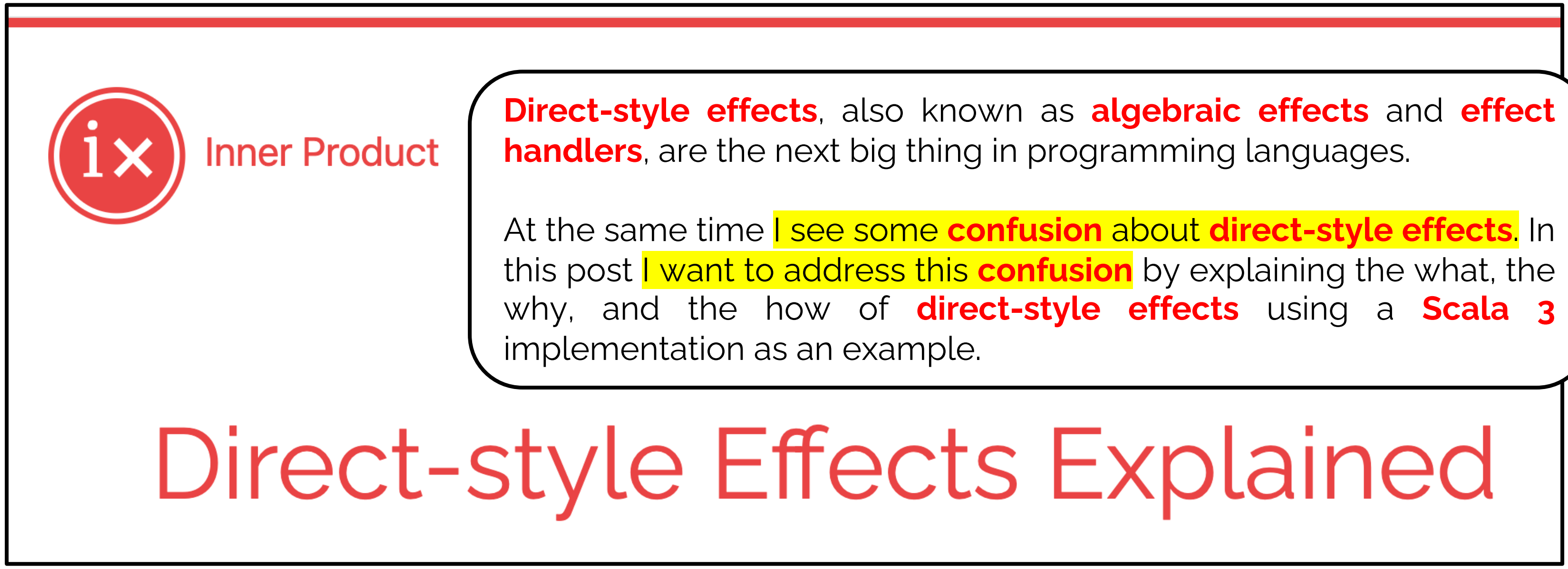
The type of a singleton object

Daniel Westheide
@kaffeecoder

slides by  @philip_schwarz FP Illuminated https://fpilluminated.com/

Have you read this blog post by **Noel Welsh**?

https://www.inner-product.com/posts/direct-style-effects/



Inner Product

Direct-style Effects Explained

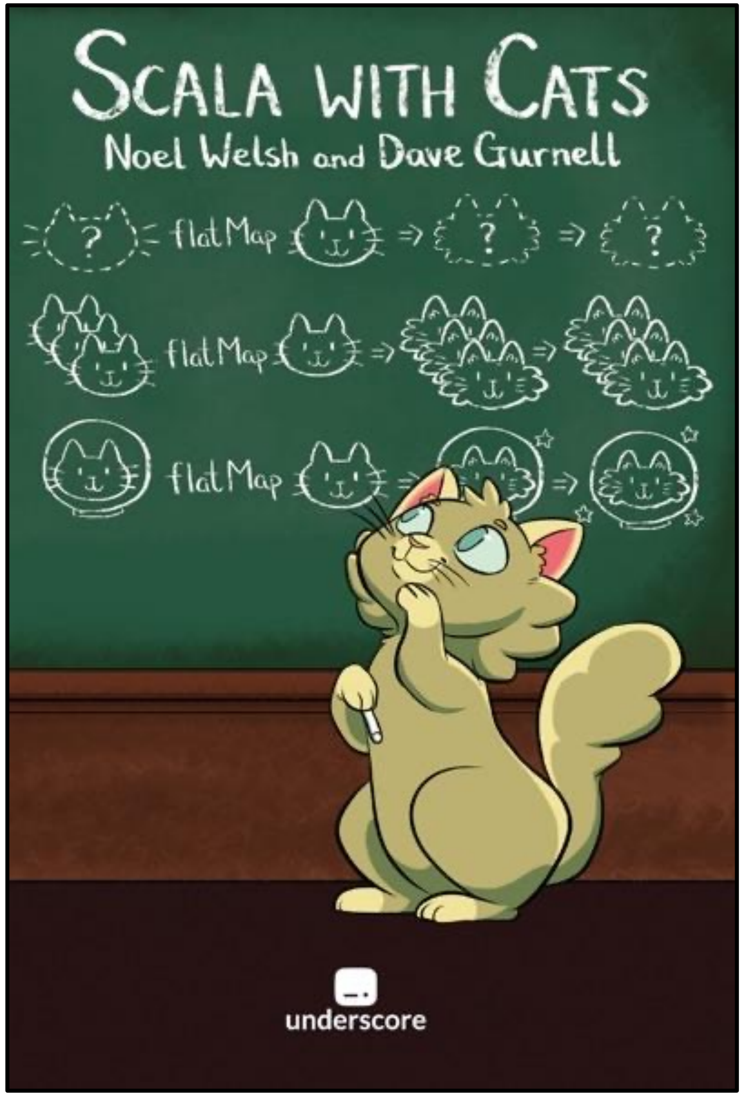
Direct-style effects, also known as **algebraic effects** and **effect handlers**, are the next big thing in programming languages.

At the same time I see some **confusion** about **direct-style effects**. In this post I want to address this **confusion** by explaining the what, the why, and the how of **direct-style effects** using a **Scala 3** implementation as an example.

Noel Welsh

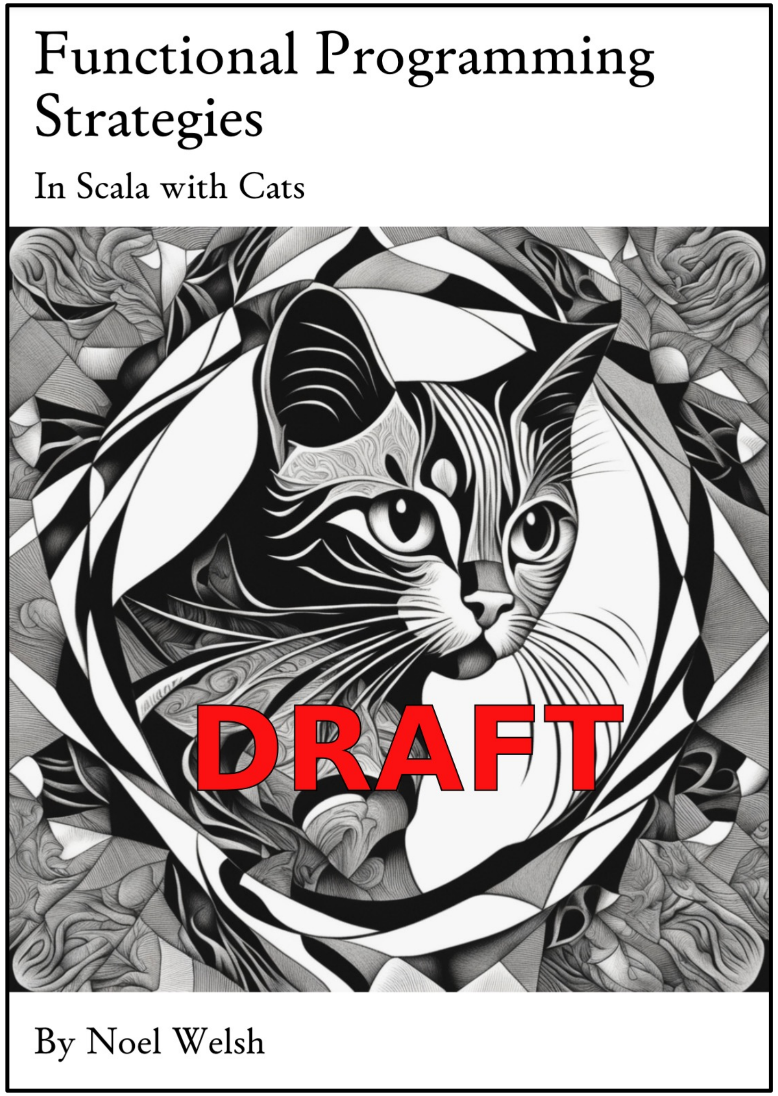
@noelwelsh

Noel Welsh is, of course, the co-author of **Scala with Cats**, and author of the upcoming **Functional Programming Strategies**



Noel Welsh

@noelwelsh



The subject of this deck is the small `Print[A]` program seen in the blog post section highlighted below.

In order to understand the program, I had to fill two lacunae. The first one was minor, but the other one was not, and filling it was long overdue. In the passage below, **Noel** kind of alludes to it when he speaks of machinery that is new in **Scala 3** and is probably unfamiliar to many readers.

The intention behind this deck is to share the following:

1. The two bits of knowledge that I used to fill the above lacunae. I acquired the first one from a book written by **Daniel Westheide** , and the second one from a blog post by **Adam Warski**.
2. The process that I followed to reach an understanding of the program, in case someone else finds bits of it useful.

Direct-style Effect Systems in Scala 3

Let's now implement a **direct-style effect system** in **Scala 3**. This requires some machinery that is new in **Scala 3**. Since that's probably unfamiliar to many readers we're going to start with an example, explain the programming techniques behind it, and then explain the concepts it embodies.

Our example is a simple **effect system** for printing to the console. The implementation is below. You can save this in a file (called, say, **Print.scala**) and run it with **scala-cli** with the command **scala-cli Print.scala**.

Noel Welsh

@noelwelsh

Before we get started, we need as context the blog post section highlighted below. See next slide for the contents of that section.

- What We Care About
  - Direct and Monadic Style
  - Description and Action
- Reasoning and Composing with Effects
- The Design Space of Effect Systems
- Direct-style Effect Systems in Scala 3
  - Composition of Direct-Style Effects
  - Effects That Change Control Flow
- Capturing, Types, and Effects
- Conclusions and Further Reading

Noel Welsh

@noelwelsh

Description and Action

Any **effect system** must have a **separation** between **describing** the **effects** that should occur, and actually **carrying out** those **effects**. This is a requirement of **composition**. Consider perhaps the **simplest effect** in any programming language: **printing to the console**. In **Scala** we can accomplish this as a **side effect** with **println**:

```scala
println("OMG, it's an effect")
```

Imagine we want to **compose** the **effect** of **printing to the console** with the **effect** that **changes the color of the text** on the console. With the **println side effect** we cannot do this. Once we call **println** the output is already printed; there is no opportunity to change the color.

Let me be clear that the goal is ***composition***. We can certainly use two **side effects** that happen to occur in the correct order to get the output with the color we want.

```scala
println("\u001b[91m") // Color code for bright red text
println("OMG, it's an effect")
```

However this is not the same thing as **composing** an **effect** that **combines** these two **effects**. For example, the example above doesn't reset the foreground color so all subsequent output will be bright red. This is the classic problem of side effects: they have **"action at a distance"** meaning one part of the program can change the meaning of another part of the program. This in turns means we **cannot reason locally**, nor can we build programs in a **compositional** way.

What we really want is to write code like

```scala
Effect.println("OMG, it's an effect").foregroundBrightRed
```

which limits the foreground colour to just the given text. We can only do that if we have a **separation** between **describing** the **effect**, as we have done above, and actually **running** it.

Next, let’s take a quick glance at **Noel**’s program, just to get an idea of its size and constituent parts.

```scala
@main def go(): Unit = {
  // Declare some `Prints`
  val message: Print[Unit] =
    Print.println("Hello from direct-style land!")

  // Composition
  val red: Print[Unit] =
    Print.println("Amazing!").prefix(Print.print("> ").red)

  // Make some output
  Print.run(message)
  Print.run(red)
}
```

```scala
type Print[A] = Console ?=> A
```

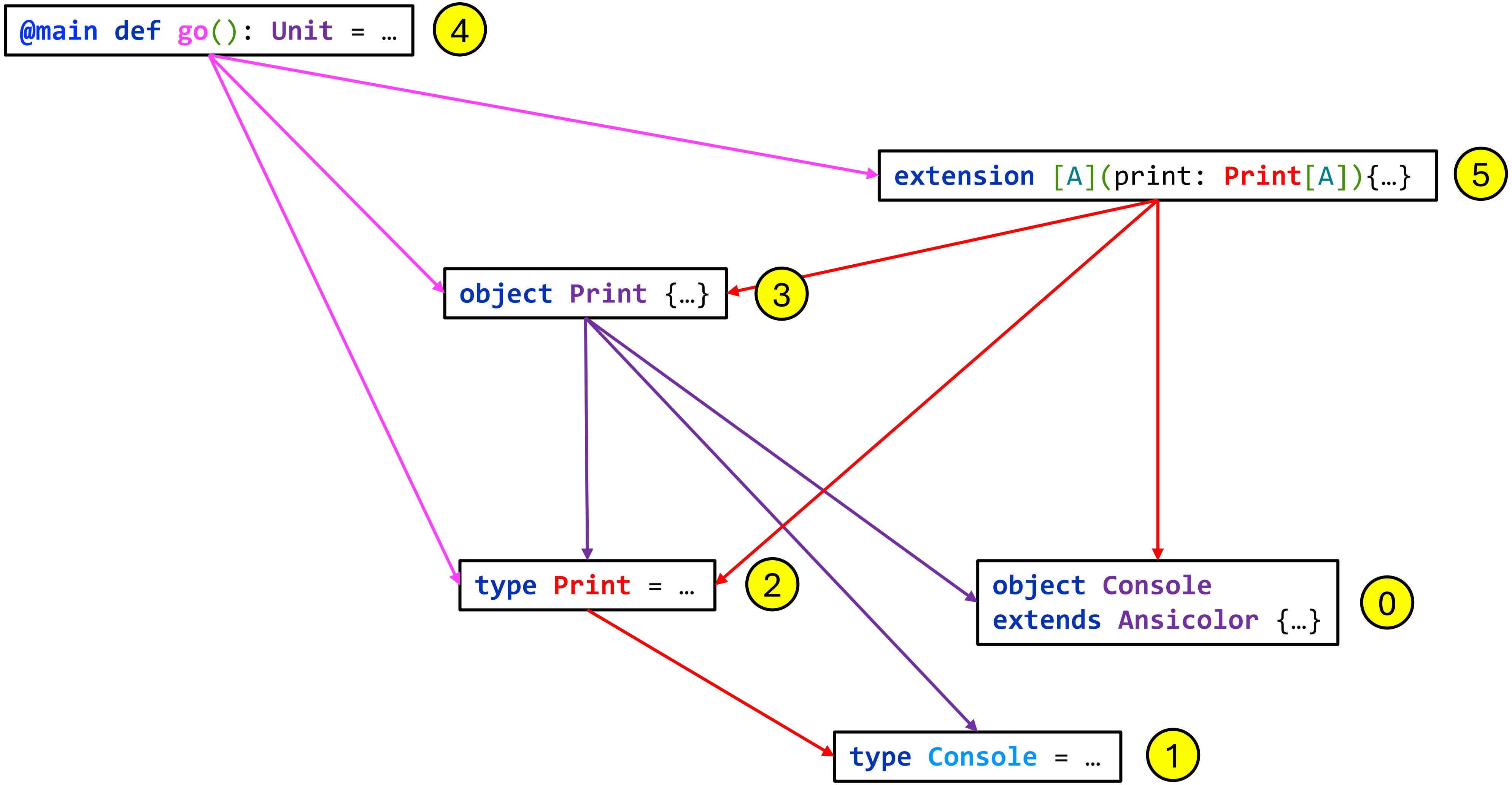
```scala
// For convenience, so we don't have
// to write Console.type everywhere.
type Console = Console.type
```

Because this program is small, it might be tempting to assume that it is trivial to understand. In that respect, I found its size to be deceptive. See the next slide for the dependencies that exist between its modules.

```scala
object Print {
  def print(msg: Any)(using c: Console): Unit =
    c.print(msg)

  def println(msg: Any)(using c: Console): Unit =
    c.println(msg)

  def run[A](print: Print[A]): A = {
    given c: Console = Console
    print
  }

  /** Constructor for `Print` values */
  inline def apply[A](inline body: Console ?=> A): Print[A] =
    body
}
```

```scala
extension [A](print: Print[A]) {

  /** Insert a prefix before `print` */
  def prefix(first: Print[Unit]): Print[A] =
    Print {
      first
      print
    }

  /** Use red foreground color when printing */
  def red: Print[A] =
    Print {
      Print.print(Console.RED)
      val result = print
      Print.print(Console.RESET)
      result
    }
}
```

To understand how the program works, let us consider each component in turn, in the order imposed by the numeric labels shown next to the components.



`@main def go(): Unit = …` 4

`extension [A](print: Print[A]){…}` 5

`object Print {…}` 3

`type Print = …` 2

`object Console extends Ansicolor {…}` 0

`type Console = …` 1

The first component to consider is **Console**. All we need to say about it, is that it is a **singleton object** providing two methods that are used by the program to write to the console.

```
object Console extends Ansicolor {…}
```

⓪

```
/** Prints an object to `out` using its `toString` method.
 *
 *  @param obj the object to print; may be null.
 *  @group console-output
 */
def print(obj: Any): Unit = …
```

```
/** Prints out an object to the default output, followed
 * by a newline character.
 *
 *  @param x the object to print.
 *  @group console-output
 */
def println(x: Any): Unit = …
```

The next component to consider is the following **type alias** definition.

```
// For convenience, so we don't have
// to write Console.type everywhere.
type Console = Console.type
```

①

What is **Console.type**, and why is **Console** needed?

Console.type is the **singleton type** of **singleton object Console**.

If you already knew that, you can skip the next two slides, which provide a bit more detail on that topic.

As to the reason for defining **Console**, it is convenience.

Since the program contains references to the type of **Console**, it is less verbose and less distracting to reference the type with **Console** than with **Console.type**.

Evaluation semantics

Scala's singleton objects are instantiated lazily. To illustrate that, let's print something to the standard output in the body of our `Colors` object:

```
object Colors {
  println("initialising common colors")
  val White: Color = new Color(255, 255, 255)
  val Black: Color = new Color(0, 0, 0)
  val Red:   Color = new Color(255, 0, 0)
  val Green: Color = new Color(0, 255, 0)
  val Blue:  Color = new Color(0, 0, 255)
}
```

If you re-start the REPL, you won't see anything printed to the standard output just yet.

As soon as you access the `Colors` object for the first time, though, it will be initialised and you will see something printed to the standard output.

Accessing the object after that will not print anything again, because the object has already been initialised. Let's try this out in the REPL:

```
scala> val colors = Colors
initialising common colors
val colors: Colors.type = Colors$@4e060c41

scala> val blue = colors.Blue
val blue: Color = Color@61da01e6
```

This lazy initialization is pretty similar to how the singleton pattern is often implemented in Java.

Daniel Westheide
@kaffeecoder

The type of a singleton object

As we have seen in the REPL output above, the type of the expression `Colors` is not `Colors`, but `Colors.type`, and the toString value in that REPL session was `Colors`$@4e060c41.

That last part is the object id and will be a different one every time you start a new **Scala** REPL.

What can we learn from this? For one, `Colors` itself is not a type, or a class, it is an instance of a type.

Also, while there can be an arbitrary number of values of type **Meeple**, there can only be exactly one value of type `Colors.type`, and that value is bound to the name `Colors`.

Because of this, `Colors.type` is called a **singleton type**.

Since `Colors` itself is not a type, you cannot define a function with the following signature:

```
def pickColor(colors: Colors): Color
```

You can define a function with this signature, though:

```
def pickColor(colors: Colors.type): Color
```

Daniel Westheide
@kaffeecoder

The next component to look at is the following **type alias** definition:

```
type Print[A] = Console ?=> A
```

②

On the left hand side of the definition we have `Print[A]`, which is a description of an **effect**.

But what does the right hand side of the definition mean?

It is the type of a **Context Function**.

Noel Welsh

@noelwelsh

A `Print[A]` is a **description**: a program that when run may print to the **console** and also compute a value of type `A`. It is implemented as a **context function**.

You can think of a **context function** as a normal function with **given** (**implicit**) parameters.

In our case a `Print[A]` is a **context function** with a **Console given** parameter (**Console** is a type in the **Scala** standard library.)

Adam Warski wrote a great blog post on **Context Functions** called **Context is King**.

To further understand what a **Context Function** is, let's look at some excerpts from that post.

What is a context function?

Before we dive into usage examples and consider why you would be at all interested in using **context functions**, let's see what they are and how to use them.

A **regular function** can be written in **Scala** in the following way:

```scala
val f: Int => String = (x: Int) => s"Got: $x"
```

A **context function** looks similar, however, the crucial difference is that the parameters are **implicit**. That is, when using the function, the parameters need to be in the **implicit scope**, and provided earlier to the compiler e.g. using **given**; by default, they are not passed **explicitly**.

The **type** of a **context function** is written down using `?=>` instead of `=>`, and in the implementation, we can refer to the **implicit parameters** that are in scope, as defined by the type. In **Scala 3**, this is done using **`summon[T]`**, which in **Scala 2** has been known as **`implicitly[T]`**. Here, in the body of the function, we can access the given `Int` value:

```scala
val g: Int ?=> String = s"Got: ${summon[Int]}"
```

Just as **f** has a type **`Function1`**, **g** is an instance of **`ContextFunction1`**:

```scala
val ff: Function1[Int, String] = f
val gg: ContextFunction1[Int, String] = g
```

Context is King

Adam Warski

@adamwarski

Context functions are regular values that can be passed around as parameters or stored in collections.

https://blog.softwaremill.com/context-is-king-20f533474cb3

```
val g: Int ?=> String = s"Got: ${summon[Int]}"
```

We can invoke a **context function** by **explicitly** providing the **implicit parameters**:

```
println(g(using 16))
```

Or we can provide the value in the **implicit scope**. The compiler will figure out the rest:

```
println {
  given Int = 42
  g
}
```

Context is King

Adam Warski

@adamwarski

***Sidenote**: you should never use "common" types such as **Int** or **String** for **given/implicit** values. Instead, anything that ends up in the **implicit scope** should have a narrow, custom type, to avoid accidental **implicit scope** contamination.*

In the next excerpt, **Adam** looks at an example usage of **context functions**.

Sane ExecutionContexts

Let's start looking at some usages! If you've been doing any programming using **Scala 2** and **Akka**, you've probably encountered the **ExecutionContext**. Almost any method that was dealing with **Futures** probably had the additional **implicit ec**: **ExecutionContext** parameter list.

For example, here's what a simplified fragment of a business logic function that saves a new user to the database, if a user with the given email does not yet exist, might look like in **Scala 2**:

```scala
case class User(email: String)

def newUser(u: User)(implicit ec: ExecutionContext): Future[Boolean] = {
  lookupUser(u.email).flatMap {
    case Some(_) => Future.successful(false)
    case None    => saveUser(u).map(_ => true)
  }
}

def lookupUser(email: String)(implicit ec: ExecutionContext): Future[Option[User]] = ???
def saveUser(u: User)(implicit ec: ExecutionContext): Future[Unit] = ???
```

Context is King

Adam Warski

@adamwarski

We assume that the **lookupUser** and **saveUser** methods interact with the database in some **asynchronous** or **synchronous** way.

Note how the **ExecutionContext** needs to be **threaded** through all of the invocations. It's not a deal-breaker, but still an annoyance and one more piece of **boilerplate**.

It would be great if we could capture the fact that we require the **ExecutionContext** in some abstract way …

Turns out, with **Scala 3** we can! That's what **context functions** are for. Let's define a **type alias**:

```
type Executable[T] = ExecutionContext ?=> Future[T]
```

Any method where the result type is an **Executable**[T], will require a **given** (**implicit**) **execution context** to obtain the result (the **Future**).

Here's what our code might look like after refactoring:

```
case class User(email: String)

def newUser(u: User): Executable[Boolean] = {
  lookupUser(u.email).flatMap {
    case Some(_) => Future.successful(false)
    case None    => saveUser(u).map(_ => true)
  }
}

def lookupUser(email: String): Executable[Option[User]] = ???
def saveUser(u: User): Executable[Unit] = ???
```

Context is King

Adam Warski

@adamwarski

The type signatures are shorter — that's one gain. The code is otherwise unchanged — that's another gain.

For example, the **lookupUser** method requires an **ExecutionContext**. It is automatically provided by the compiler since it is in scope — as specified by the top-level **context function** method signature.

This slide and the next one are just a recap of the changes that **Adam** made when he introduced **context functions**.

```
case class User(email: String)

def newUser(u: User)(implicit ec: ExecutionContext): Future[Boolean] = {
 lookupUser(u.email).flatMap {
  case Some(_) => Future.successful(false)
  case None    => saveUser(u).map(_ => true)
 }
}

def lookupUser(email: String)(implicit ec: ExecutionContext): Future[Option[User]] = ???
def saveUser(u: User)(implicit ec: ExecutionContext): Future[Unit] = ???
```

```
type Executable[T] = ExecutionContext ?=> Future[T]

case class User(email: String)

def newUser(u: User): Executable[Boolean] = {
 lookupUser(u.email).flatMap {
  case Some(_) => Future.successful(false)
  case None    => saveUser(u).map(_ => true)
 }
}

def lookupUser(email: String): Executable[Option[User]] = ???
def saveUser(u: User): Executable[Unit] = ???
```

```
                                                                                            -+  type Executable[T] = ExecutionContext ?=> Future[T]
                                                                                            =
case class User(email: String)                                                                  case class User(email: String)

def newUser(u: User)(implicit ec: ExecutionContext): Future[Boolean] = {                    <>  def newUser(u: User): Executable[Boolean] = {
  lookupUser(u.email).flatMap {                                                             =     lookupUser(u.email).flatMap {
    case Some(_) => Future.successful(false)                                                        case Some(_) => Future.successful(false)
    case None    => saveUser(u).map(_ => true)                                                      case None    => saveUser(u).map(_ => true)
  }                                                                                               }
}                                                                                               }

def lookupUser(email: String)(implicit ec: ExecutionContext): Future[Option[User]] = ???    <>  def lookupUser(email: String): Executable[Option[User]] = ???
def saveUser(u: User)(implicit ec: ExecutionContext): Future[Unit] = ???                        def saveUser(u: User): Executable[Unit] = ???
```

The next component to look at is **singleton object** `Print`.

③

```
object Print {
  def print(msg: Any)(using c: Console): Unit =
    c.print(msg)

  def println(msg: Any)(using c: Console): Unit =
    c.println(msg)

  def run[A](print: Print[A]): A = {
    given c: Console = Console
    print
  }

  /** Constructor for `Print` values */
  …<we’ll come back to this later>…
}
```

```
type Print[A] = Console ?=> A
```

The **print** and **println** functions are straightforward: given a message and an **implicit console**, they display the message by passing it to the corresponding **console** functions.

The **run** function is also straightforward: given a `Print[A]`, i.e. a **value** (**context function**) **describing** a **printing effect** (a **functional effect**), it carries out (**runs**) the **effect**, which causes the **printing** (the **side effect**) to **take place** (to get **done**). The way it does this is by making available an **implicit Console**, and returning `Print[A]`, which causes the latter (i.e. a **context function**) to be invoked.

The next component we need to look at is the **main** function:

```
@main def go(): Unit = {
  // Declare some `Prints`
  val message: Print[Unit] =
    Print.println("Hello from direct-style land!")

  // Composition
  //…<we’ll come back to this later>…

  // Make some output
  Print.run(message)
  //…<we’ll come back to this later>…
}
```

④

Hold on a second! We saw on the previous slide that `Print.println` returns `Unit`, so how can the type of **message** be `Print[Unit]`?

```
val message: Print[Unit] =
  Print.println("Hello from direct-style land!")
```

Noel’s explanation can be found on the next slide.

Noel Welsh

@noelwelsh

Context function types have a **special rule** that makes constructing them easier: a normal expression will be converted to an expression that produces a **context function** if the type of the expression is a **context function**.

Let's unpack that by seeing how it works in practice. In the example above we have the line

```
val message: Print[Unit] =
  Print.println("Hello from direct-style land!")
```

`Print.println` is an expression with type `Unit`, not a **context function type**.

However `Print[Unit]` is a **context function type**. This type annotation causes `Print.println` to be **converted** to a **context function type**.

You can check this yourself by removing the type annotation:

```
val message =
  Print.println("Hello from direct-style land!")
```

This will not compile.

FWIW, looking back at `Print`, I couldn't help trying out the following successful modification, which changes the **print** and **println** functions from **side-effecting**, i.e. invoking them causes **side effects**, to **effectful**, i.e. they return a **functional effect**, a **description** of a computation which, when executed, will cause **side effects**.

```scala
object Print {
  def print(msg: Any)(using c: Console): Unit =
    c.print(msg)

  def println(msg: Any)(using c: Console): Unit =
    c.println(msg)

  def run[A](print: Print[A]): A = {
    given c: Console = Console
    print
  }

  /** Constructor for `Print` values */
  …<we'll come back to this later>…
}
```

```scala
object Print {
  def print(msg: Any): Print[Unit] =
    summon[Console].print(msg)

  def println(msg: Any): Print[Unit] =
    summon[Console].println(msg)

  def run[A](print: Print[A]): A = {
    given c: Console = Console
    print
  }

  /** Constructor for `Print` values */
  …<we'll come back to this later>…
}
```

Note that in the following, the above modification does not change the need for **message** to be annotated with type `Print[Unit]`

```scala
val message: Print[Unit] =
  Print.println("Hello from direct-style land!")
```

My explanation is that with or without the modification, expression `Print.println("Hello from direct-style land!")` cannot be evaluated without a console being available, but none are available, so **message** cannot be of type **Unit**. Without the modification, the expression is automatically converted to a **context function**, whereas with the modification, the value of the expression is already a **context function**. In both cases, the **context function** cannot be invoked (due to no console being available), so **message** *has* to be assigned the **context function**.

Let’s run the code that we have seen so far:

```
 $ sbt run
  [info] welcome to sbt 1.9.9 (Eclipse Adoptium Java 17.0.7)
  …
  [info] running go
  Hello from direct-style land!
  [success] Total time: 1 s, completed 5 May 2024, 17:15:32
```

It works!

```
@main def go(): Unit = {
  // Declare some `Prints`
  val message: Print[Unit] =
    Print.println("Hello from direct-style land!")

  // Composition
  //…<we’ll come back to this later>…

  // Make some output
  Print.run(message)
  //…<we’ll come back to this later>…

}
```

```
type Print[A] = Console ?=> A
```

```
// For convenience, so we don't have
// to write Console.type everywhere.
type Console = Console.type
```

```
object Print {
  def print(msg: Any)(using c: Console): Unit =
    c.print(msg)

  def println(msg: Any)(using c: Console): Unit =
    c.println(msg)

  def run[A](print: Print[A]): A = {
    given c: Console = Console
    print
  }

  /** Constructor for `Print` values */
  …<we’ll come back to this later>…
}
```

Remember this?

…
Imagine we want to **compose** the **effect** of **printing to the console** with the **effect** that **changes the color of the text** on the console. With the **println side effect** we cannot do this. Once we call **println** the output is already printed; there is no opportunity to change the color.
…
What we really want is to write code like

```
Effect.println("OMG, it's an effect").foregroundBrightRed
```

which limits the foreground colour to just the given text. We can only do that if we have a **separation** between **describing** the **effect**, as we have done above, and actually **running** it.

Noel Welsh
@noelwelsh

The final component that needs looking at (see next slide) supports exactly the above approach to **composing effects**.

Here on the right (highlighted in yellow) is an example of its usage (viewing this code has been postponed until now).

The red **effect composes** the **effect** of **printing "> "** with the **effect** of changing the string's color to red, and then **composes** the resulting **effect** with the **effect** of **printing "Amazing!"**, which is done by **prefixing** the string printed in red by the former **effect**, with the string printed by the latter **effect**.

```
@main def go(): Unit = {
  // Declare some `Prints`
  val message: Print[Unit] =
    Print.println("Hello from direct-style land!")

  // Composition
  val red: Print[Unit] =
    Print.println("Amazing!").prefix(Print.print("> ").red)

  // Make some output
  Print.run(message)
  Print.run(red)
}
```

```scala
extension [A](print: Print[A]) {

  /** Insert a prefix before `print` */
  def prefix(first: Print[Unit]): Print[A] =
    Print {
      first
      print
    }

  /** Use red foreground color when printing */
  def red: Print[A] =
    Print {
      Print.print(Console.RED)
      val result = print
      Print.print(Console.RESET)
      result
    }
}
```

```scala
@main def go(): Unit = {
  // Declare some `Prints`
  val message: Print[Unit] =
    Print.println("Hello from direct-style land!")

  // Composition
  val red: Print[Unit] =
    Print.println("Amazing!").prefix(Print.print("> ").red)

  // Make some output
  Print.run(message)
  Print.run(red)
}
```

Here (on the **left**) you can see how functions **prefix** and **red** (used on the **right**) are implemented

```scala
object Print {
  def print(msg: Any)(using c: Console): Unit =
    c.print(msg)

  def println(msg: Any)(using c: Console): Unit =
    c.println(msg)

  def run[A](print: Print[A]): A = {
    given c: Console = Console
    print
  }

  /** Constructor for `Print` values */
  inline def apply[A](inline body: Console ?=> A): Print[A] =
    body
}
```

Context function types have a **special rule** that makes constructing them easier: a normal expression will be converted to an expression that produces a **context function** if the type of the expression is a **context function**.

Running a `Print[A]` uses another bit of **special sauce**: if there is a **given** value of the correct type in scope of a **context function**, that **given** value will be automatically applied to the function. This is also what makes **direct-style composition**, an example of which is shown above, work. The calls to `Print.print` are in a **context** where a `Console` is available, and so will be evaluated once the surrounding **context function** is run.

Noel Welsh

@noelwelsh

We use the same trick (see green box) with `Print.apply`, which is a general purpose constructor. You can call **apply** with any expression and it will be converted to a **context function**. (As far as I know it is not essential to use **inline**, but all the examples I learned from do this so I do it as well. I assume it is an optimization.)

To help understand the extension functions for **composing effects**, here is their code again but with IntelliJ IDEA’s **X-Ray Mode** switched on.

`ev$0` stands for evidence, and its type is **Console**.

```
extension [A](print: Print[A]) {

  /** Insert a prefix before `print` */
  def prefix(first: Print[Unit]): Print[A] =
    Print {
      first(ev$0)
      print(ev$0)
    }(ev$0)

  /** Use red foreground color when printing */
  def red: Print[A] =
    Print {
      Print.print(Console.RED)(ev$0)
      val result: A = print(ev$0)
      Print.print(Console.RESET)(ev$0)
      result
    }(ev$0)
}
```

Remember how **Adam** modified his code to use **context functions**? See the next slide for an important point that he made in his blog post after reflecting on those changes.

```
                                                                                          -+  type Executable[T] = ExecutionContext ?=> Future[T]
                                                                                          =
case class User(email: String)                                                                case class User(email: String)

def newUser(u: User)(implicit ec: ExecutionContext): Future[Boolean] = {                  <>  def newUser(u: User): Executable[Boolean] = {
  lookupUser(u.email).flatMap {                                                           =     lookupUser(u.email).flatMap {
    case Some(_) => Future.successful(false)                                                      case Some(_) => Future.successful(false)
    case None    => saveUser(u).map(_ => true)                                                    case None    => saveUser(u).map(_ => true)
  }                                                                                             }
}                                                                                             }

def lookupUser(email: String)(implicit ec: ExecutionContext): Future[Option[User]] = ???  <>  def lookupUser(email: String): Executable[Option[User]] = ???
def saveUser(u: User)(implicit ec: ExecutionContext): Future[Unit] = ???                      def saveUser(u: User): Executable[Unit] = ???
```

```
type Executable[T] = ExecutionContext ?=> Future[T]
```

Executable as an abstraction

However, the **purely syntactic change** we've seen above — giving us cleaner type signatures — isn't the only difference. Since we now have an **abstraction** for "a computation requiring an execution context", we can build **combinators** that operate on them. For example:

```
// retries the given computation up to `n` times, and returns the
// successful result, if any
def retry[T](n: Int, f: Executable[T]): Executable[T]

// runs all of the given computations, with at most `n` running in
// parallel at any time
def runParN[T](n: Int, fs: List[Executable[T]]): Executable[List[T]]
```

This is possible because of a seemingly innocent syntactic, but huge **semantical difference**. The result of a method:

```
def newUser(u: User)(implicit ec: ExecutionContext): Future[Boolean]
```

is a **running computation**, which will eventually return a boolean. On the other hand:

```
def newUser(u: User): Executable[Boolean]
```

returns a **lazy computation**, which will only be run when an **ExecutionContext** is provided (either through the **implicit** scope or **explicitly**). This makes it possible to implement operators as described above, which can govern when and how the computations are run.

If you've encountered the **IO**, **ZIO** or **Task** datatypes before, this might look familiar. The basic idea behind those datatypes is similar: capture asynchronous computations as lazily evaluated values, and provide a rich set of combinators, forming a concurrency toolkit. Take a look at cats-effect, Monix, or ZIO for more details!

Context is King

Adam Warski

@adamwarski

I am seeing the following similarity with the `Print[A]` program

```scala
type Executable[T] = ExecutionContext ?=> Future[T]
```

```scala
// retries the given computation up to `n` times, and returns the
// successful result, if any
def retry[T](n: Int, f: Executable[T]): Executable[T] = …

// runs all of the given computations, with at most `n` running in
// parallel at any time
def runParN[T](n: Int, fs: List[Executable[T]]): Executable[List[T]] = …
```

This is possible because of a seemingly innocent syntactic, but huge **semantical difference**. The result of a method:

```scala
def newUser(u: User)(implicit ec: ExecutionContext): Future[Boolean]
```

is a **running computation**, which will eventually return a boolean. On the other hand:

```scala
def newUser(u: User): Executable[Boolean]
```

returns a **lazy computation**, which will only be run when an **ExecutionContext** is provided (either through the **implicit** scope or **explicitly**). This makes it possible to implement operators as described above, which can govern when and how the computations are run.

```scala
type Print[A] = Console ?=> A
```

```scala
extension [A](print: Print[A]) {
  /** Insert a prefix before `print` */
  def prefix(first: Print[Unit]): Print[A] = … ‡

  /** Use red foreground color when printing */
  def red: Print[A] = …
}
```

This is possible because of a seemingly innocent syntactic, but huge **semantical difference**. The result of a method:

```scala
def print(msg: Any)(using c: Console): Unit
```

is a **printing side effect**. On the other hand:

```scala
def print(msg: Any): Print[Unit]
```

returns a **lazy computation**, which will only be run when a **Console** is provided (either through the **given** scope or **explicitly**). This makes it possible to implement operators as described above, which can govern how the computations are composed.

‡ While it is quite possible to increase the similarity by changing `Print[Unit]` to `Print[A]` or `Print[B]`, it makes sense not to do so because the **prefix** function discards the value of its parameter.

Let’s uncomment the last few bits of code and run the program again:

```
$ sbt run
 [info] welcome to sbt 1.9.9 (Eclipse Adoptium Java 17.0.7)
 …
 [info] running go
 Hello from direct-style land!
 > Amazing!
 [success] Total time: 1 s, completed 5 May 2024, 17:15:32
```

And yes, we now see two messages rather than one, the second one being printed by the **composite effect**.

As a recap, in the next slide we see the whole program.

```
@main def go(): Unit = {
  // Declare some `Prints`
  val message: Print[Unit] =
    Print.println("Hello from direct-style land!")

  // Composition
  val red: Print[Unit] =
    Print.println("Amazing!").prefix(Print.print("> ").red)

  // Make some output
  Print.run(message)
  Print.run(red)
}
```

```
type Print[A] = Console ?=> A
```

```
// For convenience, so we don't have
// to write Console.type everywhere.
type Console = Console.type
```

Noel Welsh

@noelwelsh

```
object Print {
  def print(msg: Any)(using c: Console): Unit =
    c.print(msg)

  def println(msg: Any)(using c: Console): Unit =
    c.println(msg)

  def run[A](print: Print[A]): A = {
    given c: Console = Console
    print
  }

  /** Constructor for `Print` values */
  inline def apply[A](inline body: Console ?=> A): Print[A] =
    body
}
```

```
extension [A](print: Print[A]) {

  /** Insert a prefix before `print` */
  def prefix(first: Print[Unit]): Print[A] =
    Print {
      first
      print
    }

  /** Use red foreground color when printing */
  def red: Print[A] =
    Print {
      Print.print(Console.RED)
      val result = print
      Print.print(Console.RESET)
      result
    }
}
```

To conclude Part 1, in the next and slide, **Noel** describes the concepts behind his program.

Noel Welsh

@noelwelsh

That’s the mechanics of how **direct-style effect systems** work in **Scala**: it all comes down to **context functions**.

Notice what we have in these examples: we write code in the **natural direct style**, but we still have an **informative type**, `Print[A]`, that helps us **reason about effects** and we can **compose together** values of type `Print[A]`.

I’m going to deal with **composition** of different **effects** and more in just a bit. First though, I want describe the concepts behind what we’ve done.

Notice in **direct-style effects** we split **effects** into **two parts**: **context functions** that define the **effects** we need, and the **actual implementation** of those **effects**. In the literature these are called **algebraic effects** and **effect handlers** respectively. This is an important difference from **IO**, where the same type indicates the need for **effects** and provides the **implementation** of those **effects**.

Also notice that we use the **argument type** of **context functions** to indicate the **effects** we **need**, rather the **result type** as in **monadic effects**. This difference avoids the “colored function” problem with **monads**. We can think of the **arguments** as specifying **requirements** on the **environment** or **context** in which the **context functions** [operate?], hence the name.

Now let’s look at **composition** of **effects**, and **effects** that modify **control flow**.

That’s it for Part 1. If you liked it, consider checking out https://fpilluminated.com/ for more content like this.

FP Iλλuminated