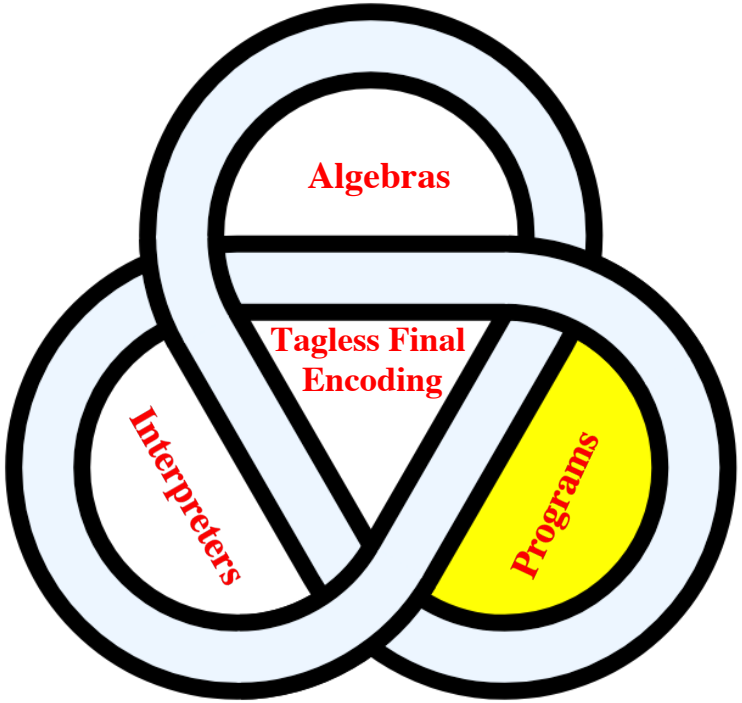
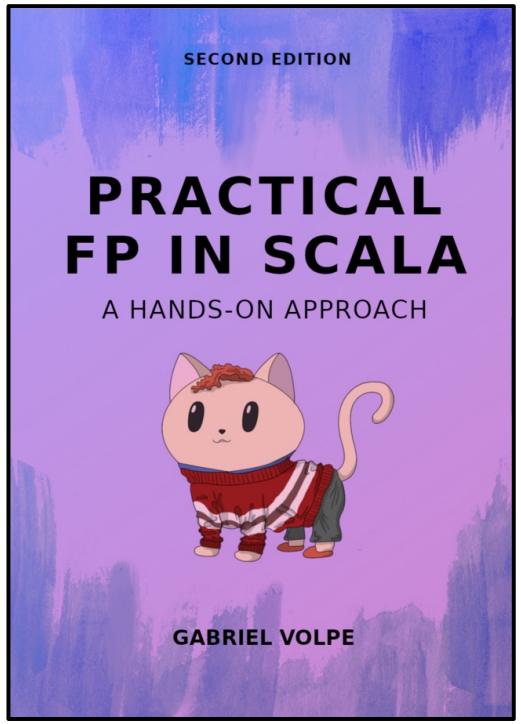
Tagless Final Encoding

Algebras and Interpreters and also Programs

an introduction, through the work of Gabriel Volpe



Gabrieλ Volpe
@volpegabriel87



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

Tagless final is a great **technique** used to structure **purely functional** applications.

Gabrielλ Volpe
@volpegabriel87

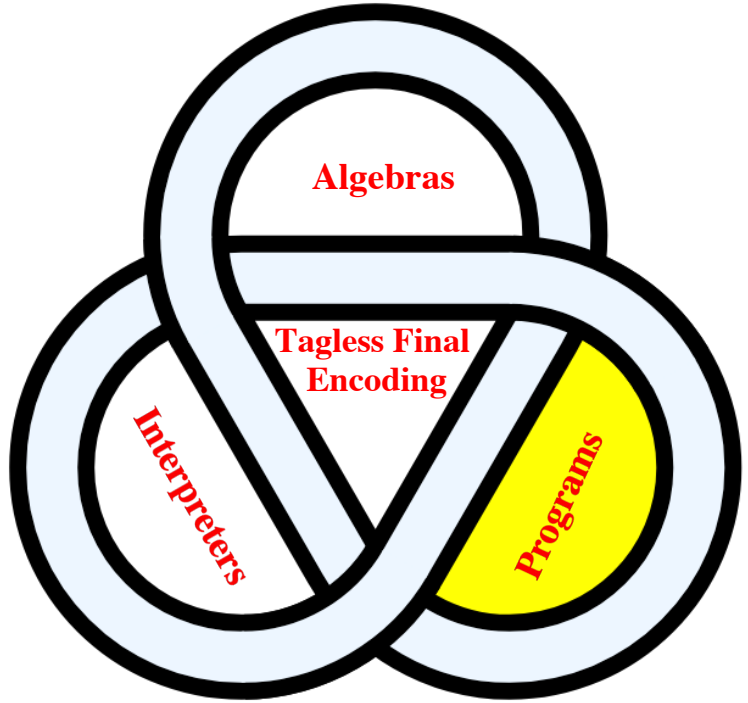
This slide deck is a quick, introductory look, at **Tagless Final**, as explained (in rather more depth than is possible or appropriate here) by **Gabriel** in his great book: **Practical FP in Scala, a Hands-on Approach**.

In the first six slides, we are going to see **Gabriel** introduce the key elements of the technique.

Tagless final is all about **algebras** and **interpreters**. Yet, something is missing when it comes to writing applications…

@philip_schwarz

Yes, as you'll see, to the two official pillars of **Tagless Final**, i.e. **algebras** and **interpreters**, **Gabriel** adds a third one: **programs**.



GabrieλVolpe
@volpegabriel87

Algebras

An **algebra** describes a new language (**DSL**) within a host language, in this case, **Scala**.

```
trait Counter[F[_]] {
  def increment: F[Unit]
  def get: F[Int]
}
```

This is a **tagless final encoded algebra**; **tagless algebra**, or **algebra** for short: a **simple interface** that **abstracts** over the **effect type** using a **type constructor** `F[_]`.

Do not confuse **algebras** with **typeclasses**, which in **Scala**, happen to share the same **encoding**.

The difference is that **typeclasses** should have **coherent instances**, whereas **tagless algebras** could have many **implementations**, or more commonly called **interpreters**.

...
Overall, **tagless algebras** seem a **perfect fit** for **encoding** **business concepts**. For example, an **algebra** responsible for managing items could be **encoded** as follows.

```
trait Items[F[_]] {
  def getAll: F[List[Item]]
  def add(item: Item): F[Unit]
}
```

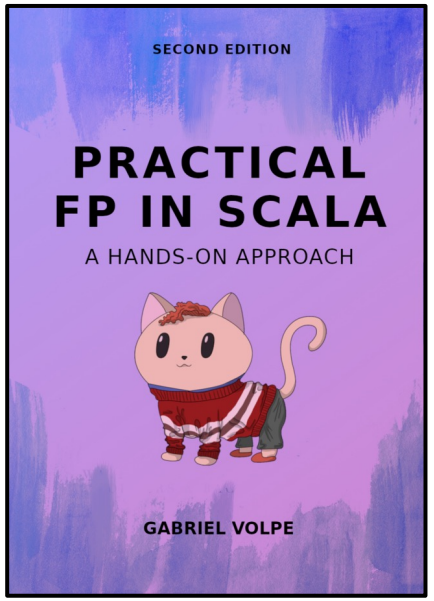
Nothing new, right? This **tagless final encoded algebra** is merely an **interface** that **abstracts** over the **effect type**. Notice that neither the **algebra** nor its functions have any **typeclass constraint**.

Tips

`Tagless algebras should not have typeclass constraints`

If you find yourself needing to add a **typeclass constraint**, such as **Monad**, to your **algebra**, what you probably need is a **program**.

The reason being that **typeclass constraints** define **capabilities**, which belong in **programs** and **interpreters**. **Algebras** should remain completely **abstract**.



Gabrielλ Volpe
@volpegabriel87

Interpreters

We would normally have two **interpreters** per **algebra**: one for **testing** and one for doing real things. For instance, we could have two different **implementations** of our **Counter**.

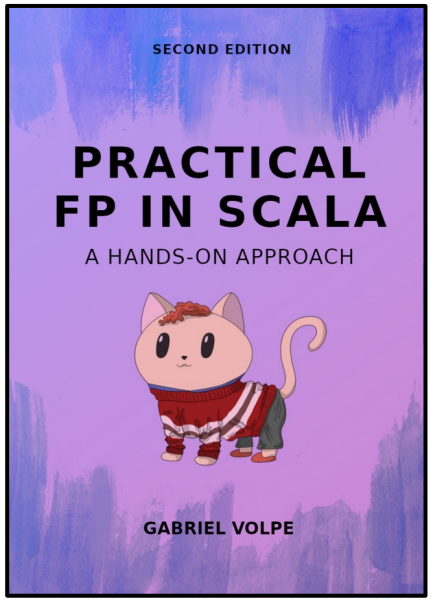
A default interpreter using **Redis**.

```
object Counter {

  @newtype case class RedisKey(value: String)

  def make[F[_]: Functor](
    key: RedisKey,
    cmd: RedisCommands[F, String, Int]
  ): Counter[F] =
    new Counter[F] {
      def increment: F[Unit] =
        cmd.increment(key.value).void
      def get: F[Int] =
        cmd.get(key.value).map(_.getOrElse(0))
  }
}
```

And a **test interpreter** using an in-memory data structure.

```
def testCounter[F[_]](
    ref: Ref[F, Int]
): Counter[F] = new Counter[F] {
  def increment: F[Unit] = ref.update(_ + 1)
  def get: F[Int] = ref.get
}
```

Interpreters help us **encapsulate state** and allow **separation of concerns**: the **interface** knows nothing about the **implementation details**. Moreover, **interpreters** can be written either using a **concrete datatype** such as `IO` or going **polymorphic** all the way, as we did in this case.



We are currently working our way through slides containing excerpts from **Chapter 2: Tagless Final Encoding**.

The next slide is an exception in that it contains an excerpt from **Chapter1: Best Practices**, which has already introduced (in a less formal way), the concept of an **interpreter** for the **Counter trait** (without yet referring to the latter as an **algebra**).

GabrieλVolpe
@volpegabriel87

In-memory counter

Let's say we need an in-memory **counter** that needs to be accessed and modified by other components. Here is what our **interface** could look like.

```
trait Counter[F[_]] {
  def increment: F[Unit]
  def get: F[Int]
}
```

It has a **higher-kinded type** `F[_]`, representing an **abstract effect**, which most of the time ends up being `IO`, but it could really be any other **concrete type** that fits the shape.

Next, we need to define an **interpreter** in the companion object of our **interface**, in this case using a `Ref`. We will talk more about it in the next section.

…

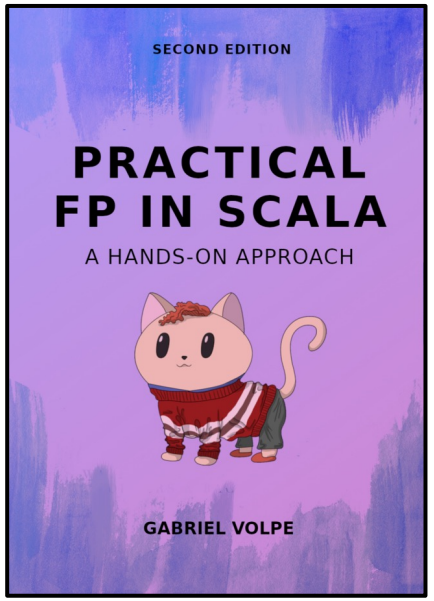
```
object Counter {
  def make[F[_]: Functor: Ref.Make]: F[Counter[F]] =
    Ref.of[F, Int](0).map { ref  =>
      new Counter[F] {
        def increment: F[Unit] = ref.update(_ + 1)
        def get: F[Int] = ref.get
    }
  }
}
```

```
import cats.Functor
import cats.effect.kernel.Ref
import cats.syntax.functor._
```

Moving on, it's worth highlighting that other **programs** will interact with this counter solely via its **interface**. E.g.

```
// prints out 0,1,6 when executed
def program(c: Counter[IO]): IO[Unit] =
  for {
    _ <- c.get.flatMap(IO.println)
    _ <- c.increment
    _ <- c.get.flatMap(IO.println)
    _ <- c.increment.replicateA(5).void
    _ <- c.get.flatMap(IO.println)
  } yield ()
```

In the next chapter, we will discuss whether it is best to pass the dependency implicitly or explicitly.



Gabrielλ Volpe
@volpegabriel87

Programs

Tagless final is all about **algebras** and **interpreters**. Yet, something is missing when it comes to writing applications: we need to use these **algebras** to describe **business logic**, and this **logic** belongs in what I like to call **programs**.

Notes

`Programs can make use of algebras and other programs`

Although it is not an official name – and it is not mentioned in the original **tagless final** paper – it is how we will be referring to such **interfaces** in this book.

Say we need to increase a counter every time there is a new item added. We could **encode** it as follows.

```
class ItemsCounter[F[_]: Apply](
    counter: Counter[F],
    items: Items[F]
){
  def addItem(item: Item): F[Unit] =
    items.add(item) *>
      counter.increment
}
```

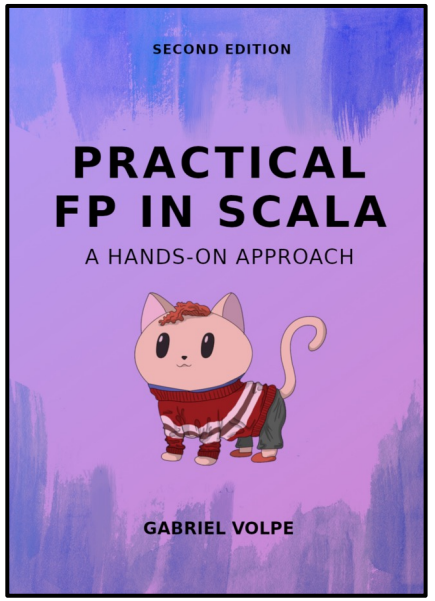
Observe the characteristics of this **program**. It is **pure business logic**, and it holds **no state** at all, which in any case, must be **encapsulated** in the **interpreters**. Notice the **typeclass constraints** as well; it is a **good practice** to have them in **programs** instead of **tagless algebras**.

…
Moreover, we can discuss **typeclass constraints**. In this case, we only need Apply to use *> (alias for productR). However, it would also work with Applicative or Monad. The rule of thumb is to **limit ourselves** to adopt the **least powerful** **typeclass** that gets the job done.

It is worth mentioning that Apply itself doesn't specify the **semantics** of **composition** solely with this **constraint**, *> might **combine** its arguments **sequentially** or **parallelly**, depending on the underlying **typeclass instance**. To ensure our **composition** is **sequential**, we could use FlatMap instead of Apply.

Tips

`When adding a typeclass constraint, remember about the principle of least power`



Gabrielλ Volpe
@volpegabriel87

Other kinds of **programs** might be directly **encoded** as functions.

```
def program[F[_]: Console: Monad]: F[Unit] =
  for {
    _ <- Console[F].println("Enter your name: ")
    n <- Console[F].readLine
    _ <- Console[F].println(s"Hello $n!")
} yield ()
```

Furthermore, we could have **programs composed** of other **programs**.

```
class MasterMind[F[_]: Console: Monad](
    itemsCounter: ItemsCounter[F],
    counter: Counter[F]
){
  def logic(item: Item): F[Unit] =
    for {
      _ <- itemsCounter.addItem(item)
      c <- counter.get
      _ <- Console[F].println(s"Number of items: $c")
    } yield ()
}
```

Whether we **encode programs** in one way or another, they should describe **pure business logic** and nothing else.
The question is: what is **pure business logic**? We could try and define a set of rules to abide by. It is allowed to:

- **Combine pure computations** in terms of **tagless algebras** and **programs**.
  - Only doing what our **effect constraints** allows us to do.
- Perform **logging** (or console stuff) only via a **tagless algebra**.
  - In Chapter 8, we will see how to ignore **logging** or **console stuff** in **tests**, which are most of the time irrelevant in such context.

You can use this as a reference. However, the answer should come up as a collective agreement within your team.

Let’s refactor a little bit the first **program** that we came across.

```scala
def program(c: Counter[IO]): IO[Unit] =
  for {
    _ <- c.get.flatMap(IO.println)
    _ <- c.increment
    _ <- c.get.flatMap(IO.println)
    _ <- c.increment.replicateA(5).void
    _ <- c.get.flatMap(IO.println)
  } yield ()
```

REFACTOR

```scala
def program(counter: Counter[IO]): IO[Unit] =
  for {
    _ <- display(counter)
    _ <- counter.increment
    _ <- display(counter)
    _ <- repeat(5){ counter.increment }
    _ <- display(counter)
  } yield ()

def display(counter: Counter[IO]): IO[Unit] =
  counter.get.flatMap(IO.println)

def repeat(n: Int)(action: IO[Unit]): IO[Unit] =
  action.replicateA(n).void
```

@philip_schwarz

On the next slide, we are going to take that **program** and wrap it in a tiny **application** that can be executed.

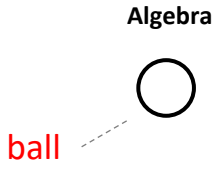
We are also going to use a little bit of **UML**, to show how the **program** uses an **algebra** (i.e. an interface) implemented by an **interpreter**.

To do this, we are going to slightly abuse UML, along the lines described by Martin Fowler in https://martinfowler.com/bliki/BallAndSocket.html.

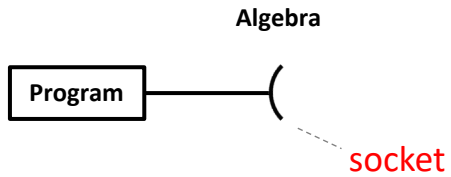
UNIFIED MODELING LANGUAGE™

valid UML notation

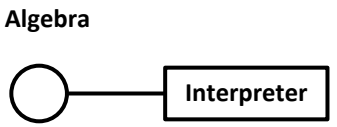
invalid UML notation



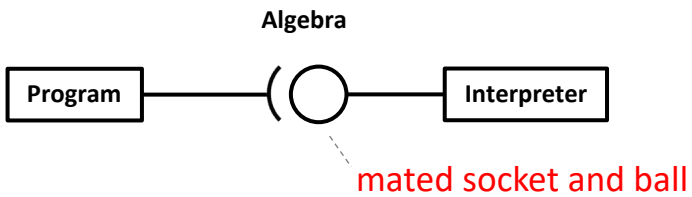
Algebra interface.



Interface **Algebra** is used (required) by **Program**.



Interface **Algebra** is realized (implemented) by **Interpreter**.



Interface **Algebra**, which is realized (implemented) by **Interpreter**, is used (required) by **Program**.

In a further twist, I'll also be putting **Algebra** inside the ball.

```scala
import cats.effect.IO
import cats.effect.IOApp.Simple

object Application extends Simple {

  // prints out 0, 1, 6 when executed
  override def run: IO[Unit] =
    Counter.make[IO].flatMap(program(_))

  private def program(counter: Counter[IO]): IO[Unit] =
    for {
      _ <- display(counter)
      _ <- counter.increment
      _ <- display(counter)
      _ <- repeat(5){ counter.increment }
      _ <- display(counter)
    } yield ()

  private def display(counter: Counter[IO]): IO[Unit] =
    counter.get.flatMap(IO.println)

  private def repeat(n: Int)(action: IO[Unit]): IO[Unit] =
    action.replicateA(n).void

}
```

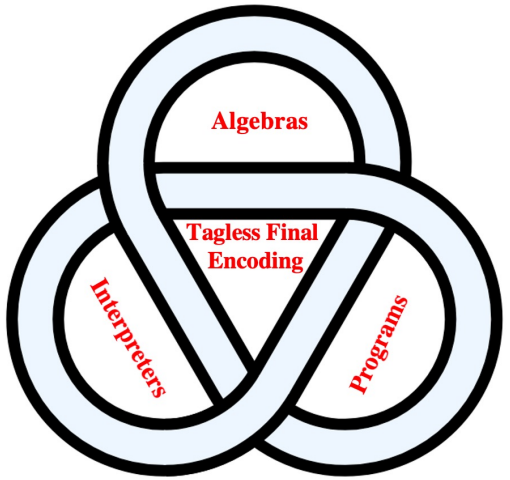
Program

Algebra

```scala
trait Counter[F[_]] {
  def increment: F[Unit]
  def get: F[Int]
}
```

Interpreter

```scala
object Counter {
  def make[F[_]: Functor: Ref.Make]: F[Counter[F]] =
    Ref.of[F, Int](0).map { ref  =>
      new Counter[F] {
        def increment: F[Unit] =
          ref.update(_ + 1)
        def get: F[Int] =
          ref.get
      }
    }
}
```



Next, we take the mastermind **program** that we saw earlier, and use it in a tiny application that tests it a bit.

Because we are going to need it in the next slide, here is a much pared down version of the Item referenced by the **program**.

```
import io.estatico.newtype.macros.newtype

import java.util.UUID

object item {

  @newtype case class ItemId(value: UUID)

  @newtype case class ItemName(value: String)

  @newtype case class ItemDescription(value: String)

  case class Item(uuid: ItemId, name: ItemName, description: ItemDescription)
}
```

```scala
import cats.effect.IO
import cats.effect.IOApp.Simple
import item.{Item, ItemDescription, ItemId, ItemName}

import java.util.UUID

object TestMasterMindProgram extends Simple {

  val item = Item(
    ItemId(UUID.fromString("0c69d914-6ff6-11ee-b962-0242ac120002")),
    ItemName("Practical FP in Scala"),
    ItemDescription("A great book")
  )

  override def run: IO[Unit] = {
    for {
      counter <- TestCounter.make[IO]
      items <- TestItems.make[IO]
      itemsCounter = new ItemsCounter[IO](counter, items)
      masterMind = new MasterMind(itemsCounter, counter)

      itemsBefore <- items.getAll
      countBefore <- counter.get
      _ = assert(itemsBefore.isEmpty)
      _ = assert(countBefore == 0)

      _ <- masterMind.logic(item)

      itemsAfter <- items.getAll
      countAfter <- counter.get
      _ = assert(itemsAfter.sameElements(List(item)))
      _ = assert(countAfter == 1)
    } yield ()
  }
}
```

Program

```scala
class MasterMind[F[_]: Console: Monad](
    itemsCounter: ItemsCounter[F],
    counter: Counter[F]
){
  def logic(item: Item): F[Unit] =
    for {
      _ <- itemsCounter.addItem(item)
      c <- counter.get
      _ <- Console[F].println(s"Number of items: $c")
    } yield ()
}
```

```scala
class ItemsCounter[F[_]: Apply](
    counter: Counter[F],
    items: Items[F]
){
  def addItem(item: Item): F[Unit] =
    items.add(item) *>
      counter.increment
}
```

Program

Program

Algebra

```scala
trait Counter[F[_]] {
  def increment: F[Unit]
  def get: F[Int]
}
```

Algebra

```scala
trait Items[F[_]] {
  def getAll: F[List[Item]]
  def add(item: Item): F[Unit]
}
```

Interpreter

Interpreter

```scala
object TestCounter {
  def make[F[_]: Functor: Ref.Make]: F[Counter[F]] =
    Ref.of[F, Int](0).map { ref =>
      new Counter[F] {
        def increment: F[Unit] =
          ref.update(_ + 1)
        def get: F[Int] =
          ref.get
      }
    }
```

```scala
object TestItems {
  def make[F[_]: Functor : Ref.Make]: F[Items[F]] =
    Ref.of[F, Map[ItemId, Item]](Map.empty).map { ref =>
      new Items[F] {
        def getAll: F[List[Item]] =
          ref.get.map(_.values.toList)
        def add(item: Item): F[Unit] =
          ref.update(_ + (item.uuid -> item))
      }
    }
}
```

Of course this was just a quick introduction to the **Tagless Final** technique.

See the book for much more depth, and many other important aspects of using the technique.

The next slide contains just a taster.

Some might question the decision to invest in this **technique** for a business application, claiming it entails **great complexity**.

This is a fair concern but let's ask ourselves, what's the alternative? Using **IO** directly in the entire application? By all means, this could work, but at what cost? At the very least, we would be giving up on **parametricity**† and the **principle of least power**.

Gabriel Volpe
@volpegabriel87

`def concrete(c: Counter[IO]): IO[Int] =`	<>	`def constrained[F[_]: Log: Monad: Time]( c: Counter[F]): F[Int] =`
`  for {` `    x <- c.get`	=	`  for {` `    x <- c.get`

`    _ <- IO.println(s"Current count: $x")`
`    t <- IO(Instant.now().atZone(ZoneOffset.UTC).getHour())`
`    _ <- IO.println(s"Current hour: $t")`
`    _ <- c.incr.replicateA(10).void.whenA(t  >= 12)`
	<>	`    _ <- Log[F].info(s"Current count: $x")`
		`    t <- Time[F].getHour`
		`    _ <- Log[F].info(s"Current hour: $t")`
		`    _ <- c.incr.replicateA(10).void.whenA(t.int  >= 12)`

`    y <- c.get` `} yield y`	=	`    y <- c.get` `} yield y`

There is a huge **mix of concerns**. How can we possibly reason about this function? How can we even test it? We got ourselves into a very **uncomfortable situation**.

Now let's compare it against the **abstract equivalent** of it. Instead of performing **side-effects**, we have now **typeclass constraints** and **capabilities**.

Teams making use of this **technique** will immediately understand that all we can do in the body of the **constrained function** is to **compose Counter**, **Log**, and **Time actions sequentially** as well as to use any property made available by the **Monad constraint**. It is true, however, that the **Scala compiler** does not **enforce** it so this is up to the **discipline** of the team.

Since **Scala** is a **hybrid language**, the only thing stopping us from running wild **side-effects** in this function is **self-discipline** and peer reviews. However, **good practices** are required in any team for multiple purposes, so I would argue it is not necessarily a bad thing, as we can do the same thing in **programs encoded directly** in **IO**.

† https://en.wikipedia.org/wiki/Parametricity

That’s all. I hope you found it useful.