Interpreters and you

Dave Gurnell, @davegurnell

underscore

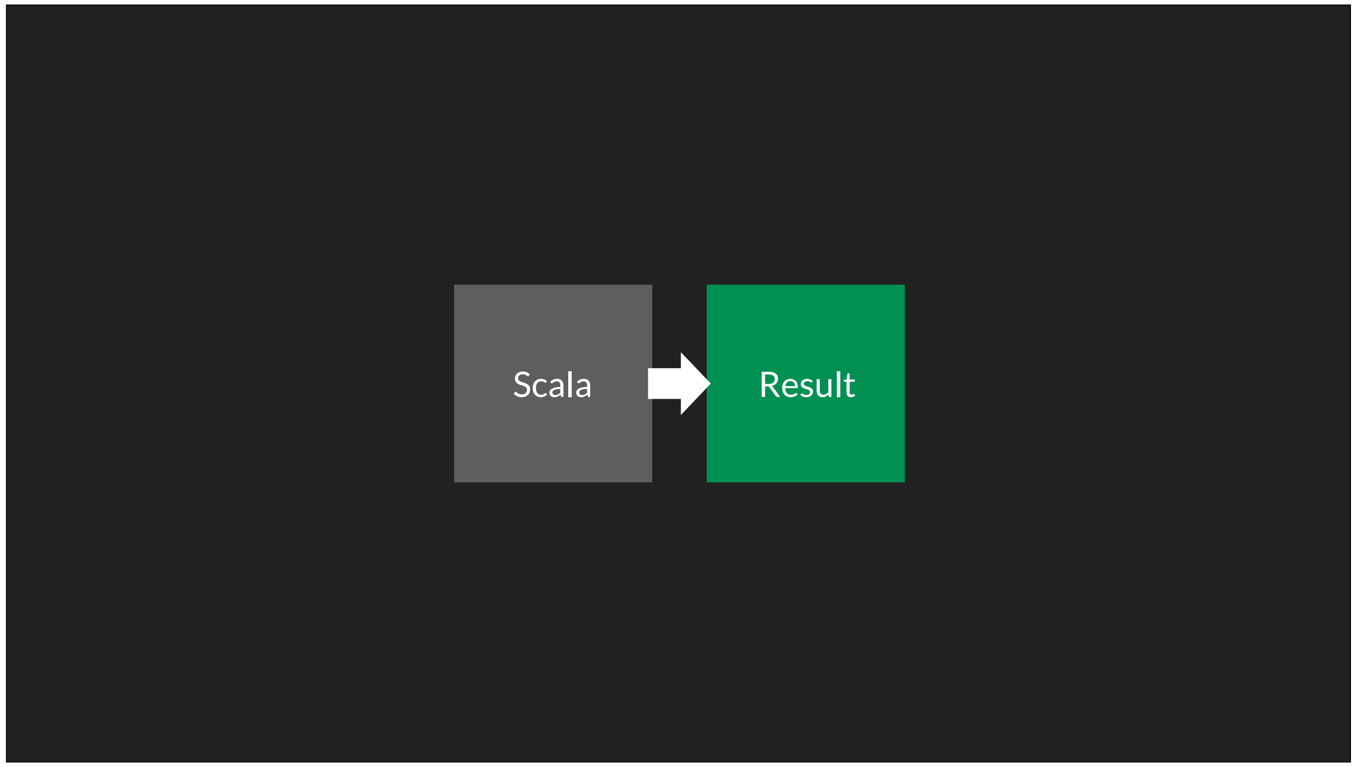
This talk is about “the interpreter pattern”, which is a really useful pattern that you can use to make your Scala programs more modular and easier to maintain.

In the talk I’ll describe that this pattern is and why it’s useful, and I’ll walk you through a number of ways of implementing it and benefitting from it in your code.

In particular, if you’ve heard the terms “free monad” and “finally tagless”, you’ll see that these patterns are just particular instances of the interpreter pattern. We’ll see what both of these things are by deriving them from simpler principles.

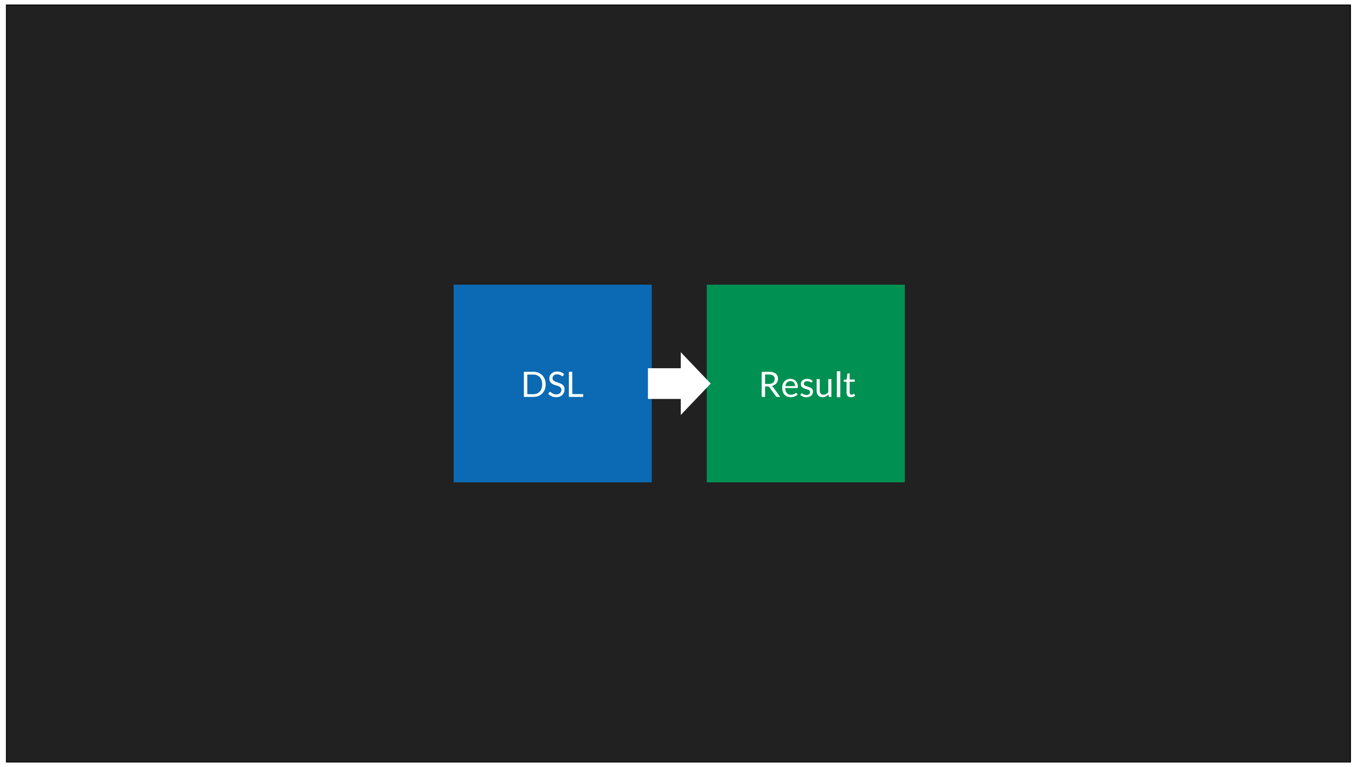
Scala is a *general purpose programming language*.

We use it every day to write *programs*. I define “program” with a specific meaning here.



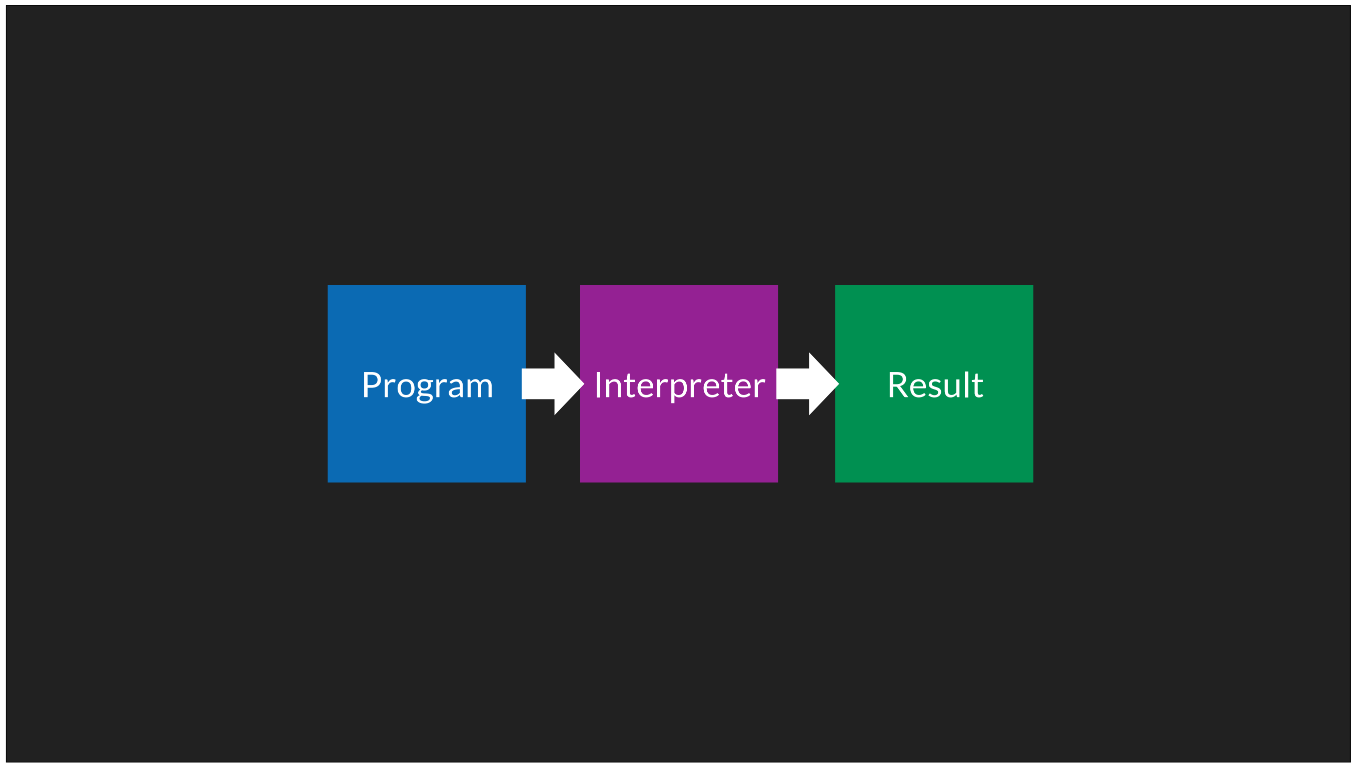
By “program”, I mean a block of code that computes a *result*. This could be a command line application, an endpoint in a web service, or a component in a larger application.

General purpose programs are jacks of all trades but masters of none. In order to write programs, we often need to take care of implementation details that make our code harder to read.



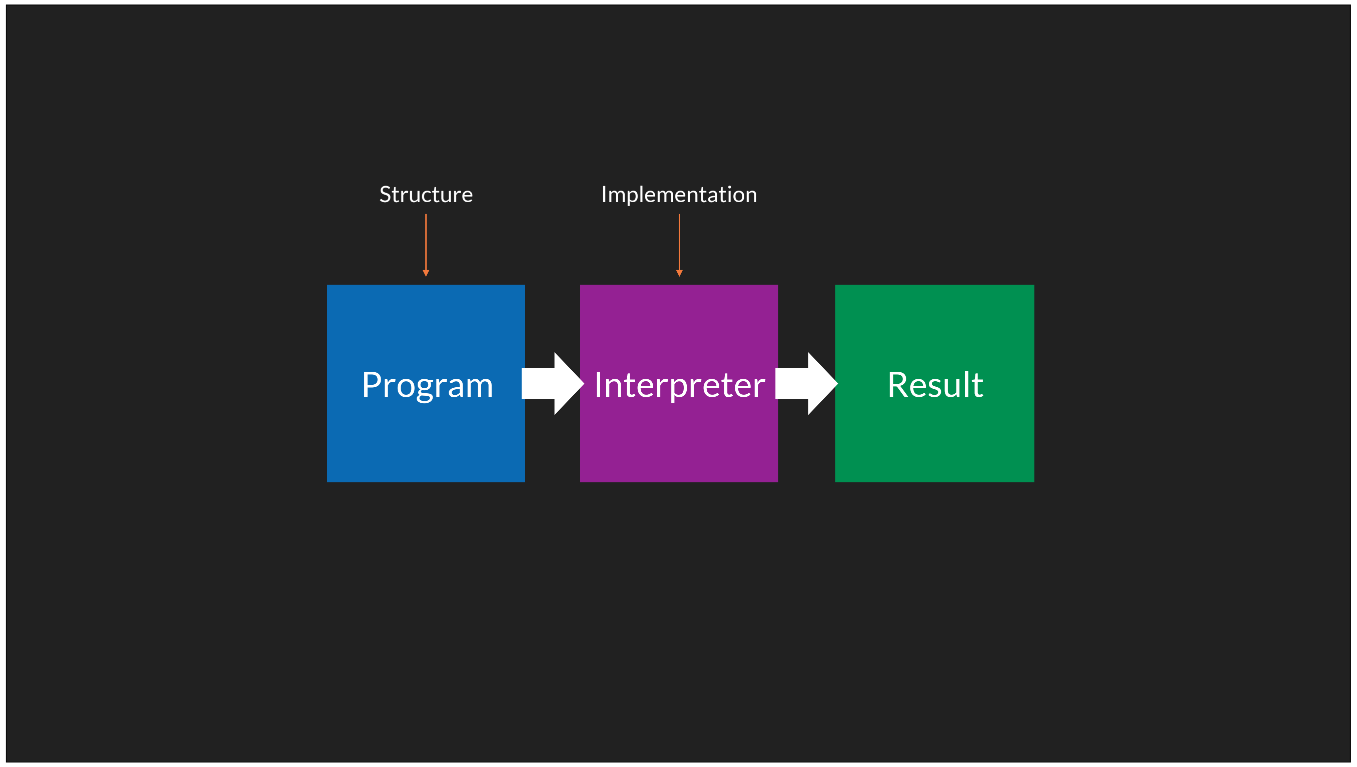
One way of getting rid of these details is to create a higher-level *domain specific language* that is tailored to a particular type of problem. When we invent a DSL, we can choose the set of atomic steps to hide implementation details and make code easier to read.

However, we can't simply *run* a program a program written in a DSL. There's no compiler and runtime like there is with Scala.



So we have to split our codebase into two parts:

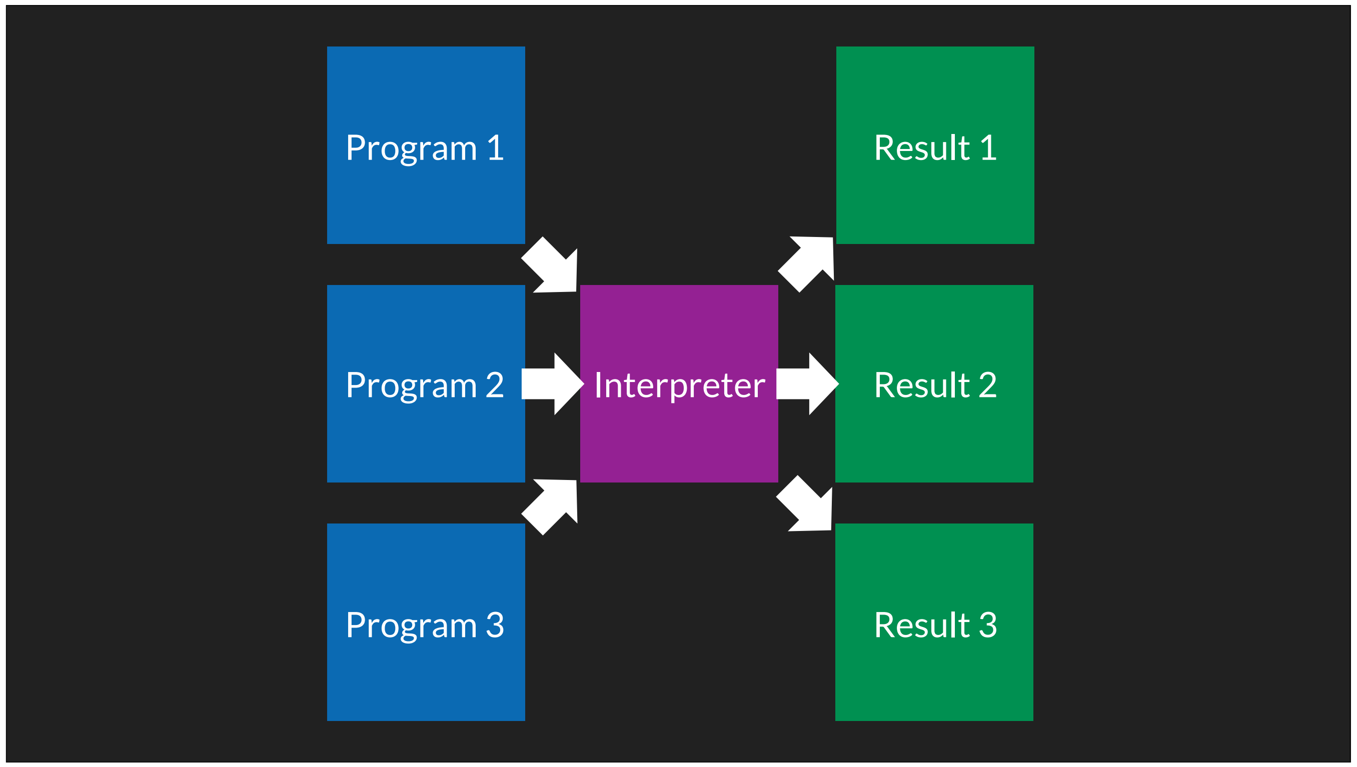
- a *program* that represents a set of steps that we want to run and…
- an *interpreter* that takes a program as input, does some work, and outputs a result.



Programs are inert — they represent the *structure* of a computation but they don’t do anything on their own.

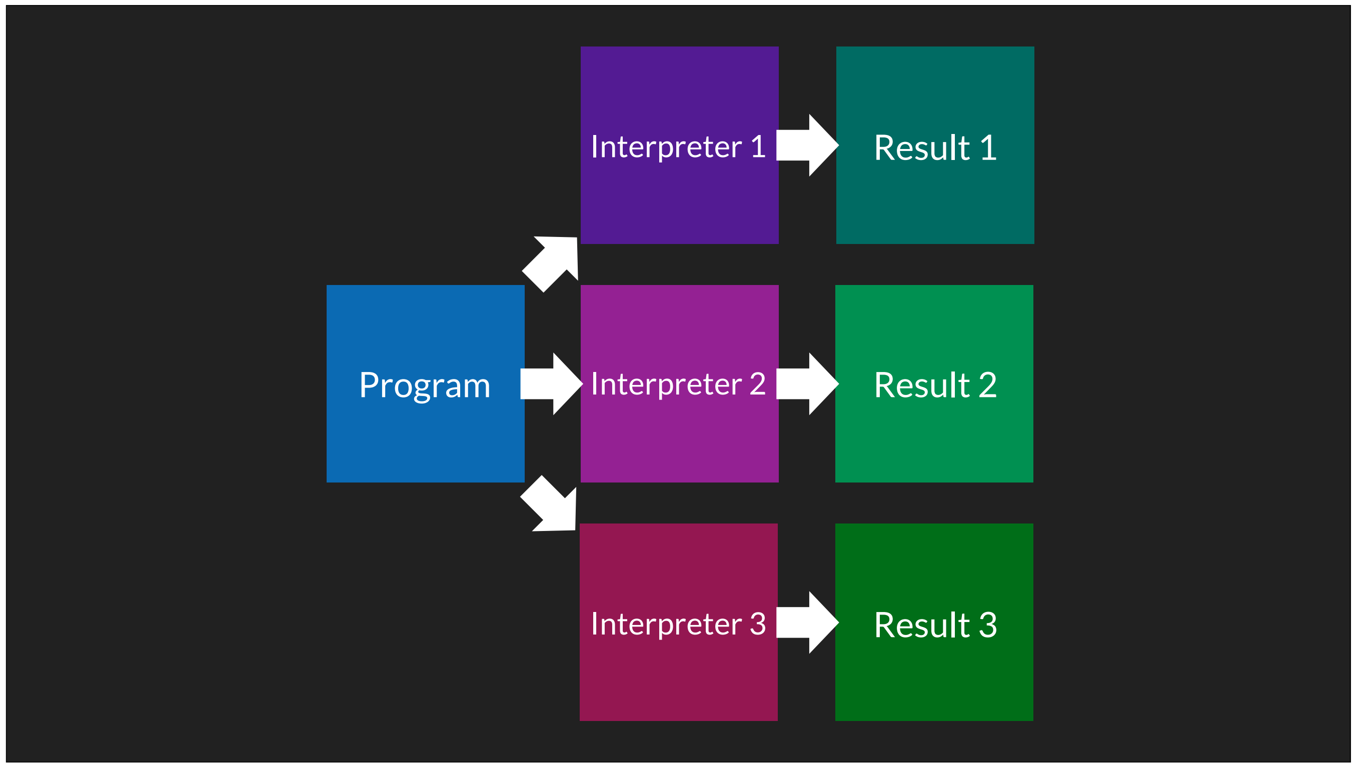
Interpreters provide the implementation details to run programs in the host programming language.

Splitting our codebase into two like this has numerous advantages in terms of modularity and reuse.



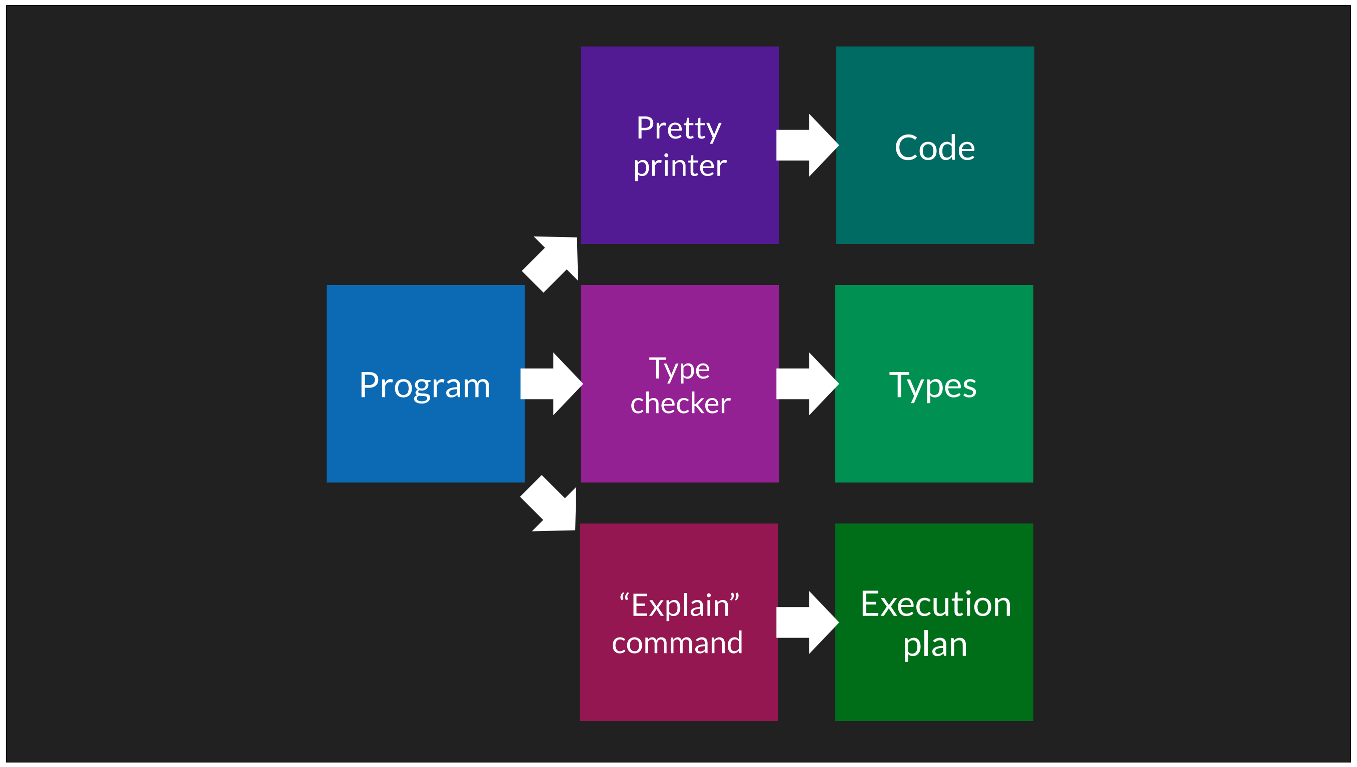
From the interpreter's point of view…

We can write an interpreter once and re-use it in lots of programs. This makes our code easier to understand because we take care of all the implementation details in one place.



From the program’s point of view…

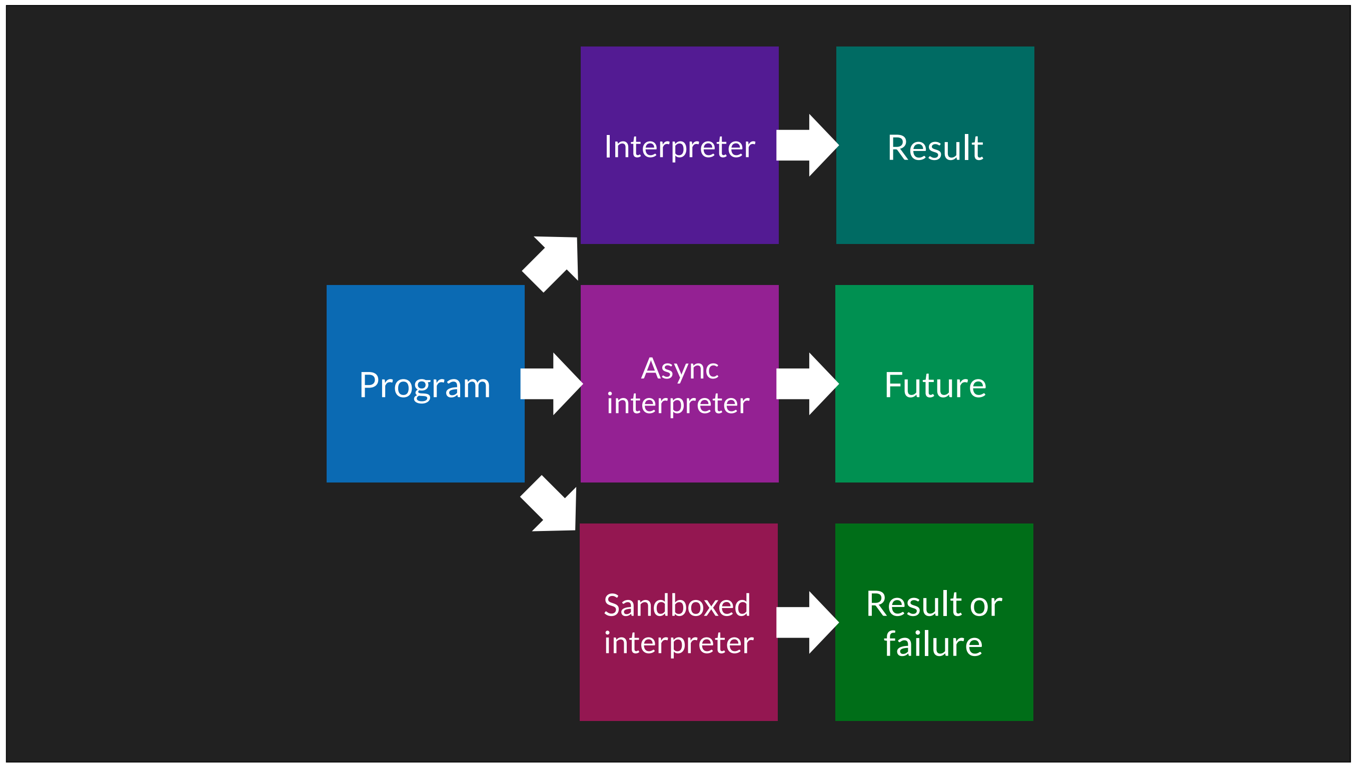
We can create a set of different interpreters that interpret the program in different ways.



We can create interpreters that output different kinds of results…

For example, we can type check or pretty print input expressions instead of running them.

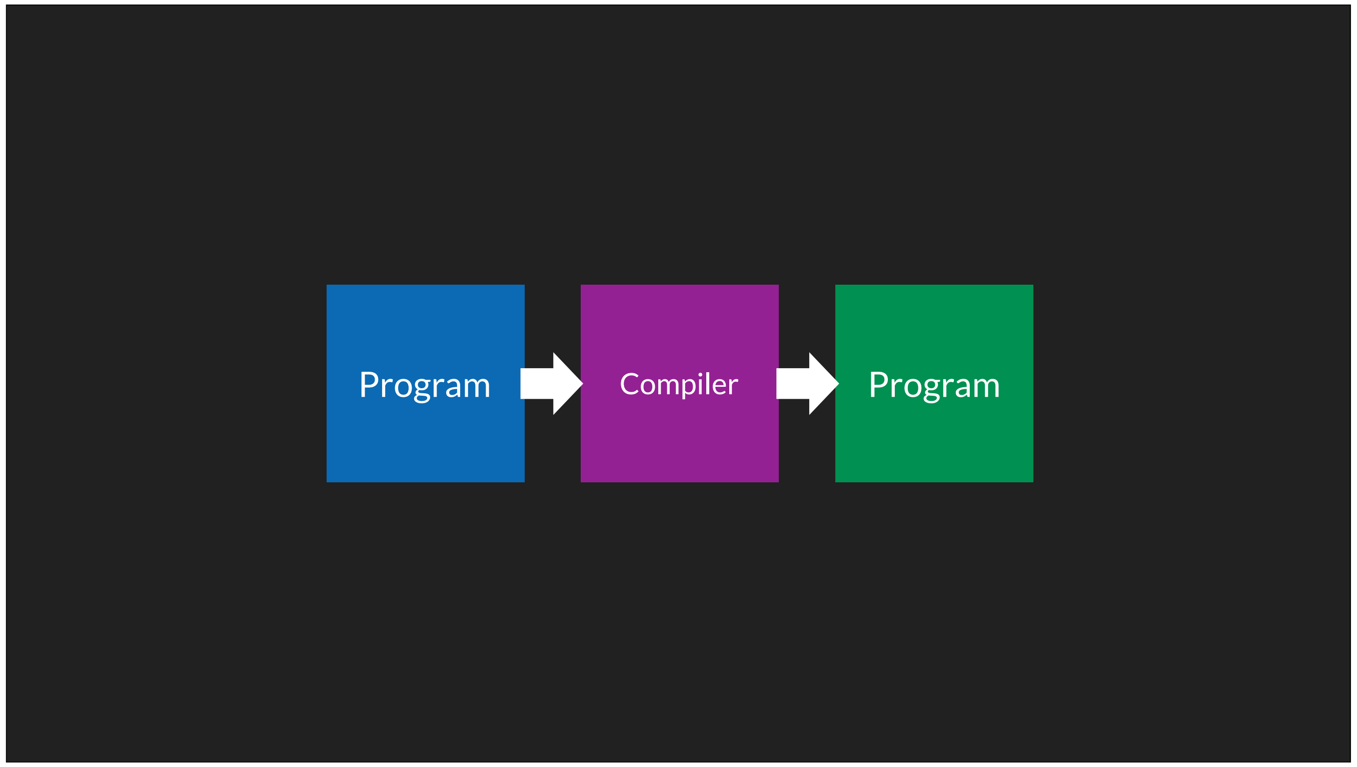
We can also *explain* results as we compute them, making it easier for humans to understand the behaviour of our applications. See Doug Clinton and Martin Carolan's talk at flatMap(Oslo) 2018 about *Droidspeak* for a great real-world example of this.



Another reason to implement different interpreters is abstracting over *effects*.

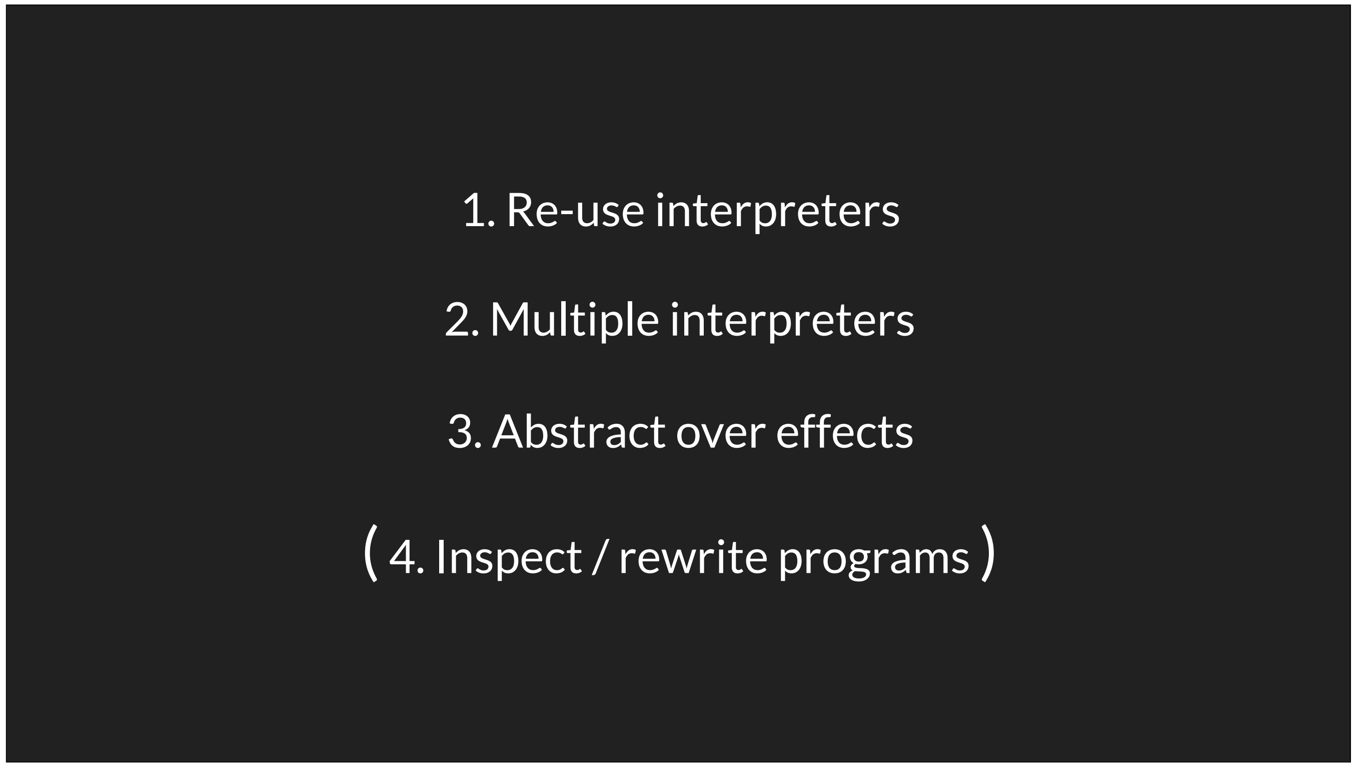
For example, we can write asynchronous interpreters that run branches in parallel.

We can also abstract over monads: *Future*, *Task*, Cats Effect's *IO*, Slick's *DBIO*, Doobie's *ConnectionIO*, and so on. This makes it much easier to port our code between libraries.



Finally, we can write interpreters that consume one program and output another, either in the same DSL or a different one. We call these interpreters *compilers*.

An example of this is *Haxl* from Facebook (TODO: it is from Facebook, right?) or *Stitch* (TODO: check name) from Twitter, which are DSLs that automatically optimise programs to minimise the number of microservice calls required during their execution.



1. Re-use interpreters

2. Multiple interpreters

3. Abstract over effects

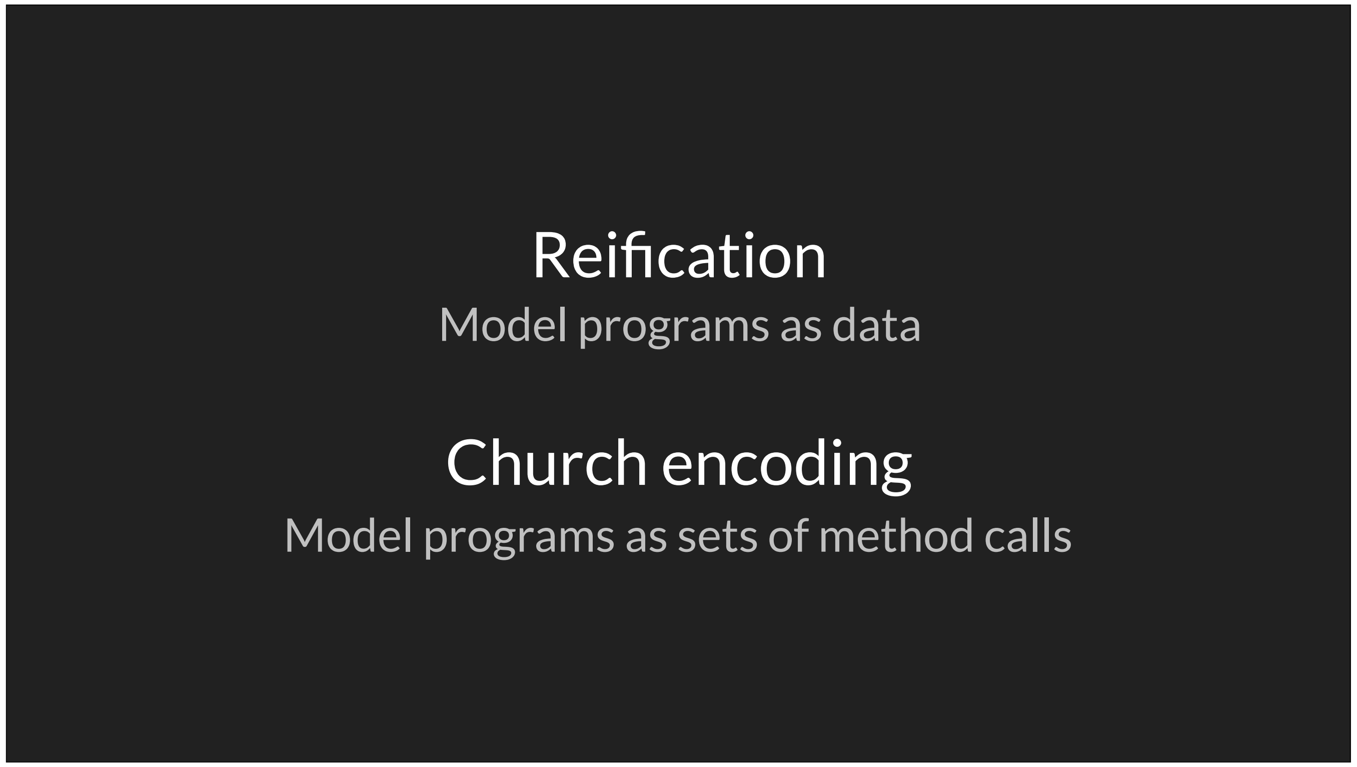
(4. Inspect / rewrite programs)

So the interpreter pattern has numerous benefits for modularity and re-use.

I’ve put inspecting, rewriting, and compiling programs in parentheses because it tends to be something that we have to target separately. Every approach in this talk will provide the top three benefits, but we have to specifically set out to achieve the fourth one. We’ll see more examples of this later.

Big approaches

There are a couple of big approaches to designing DSLs,
and we’ll go through each of them in this talk.

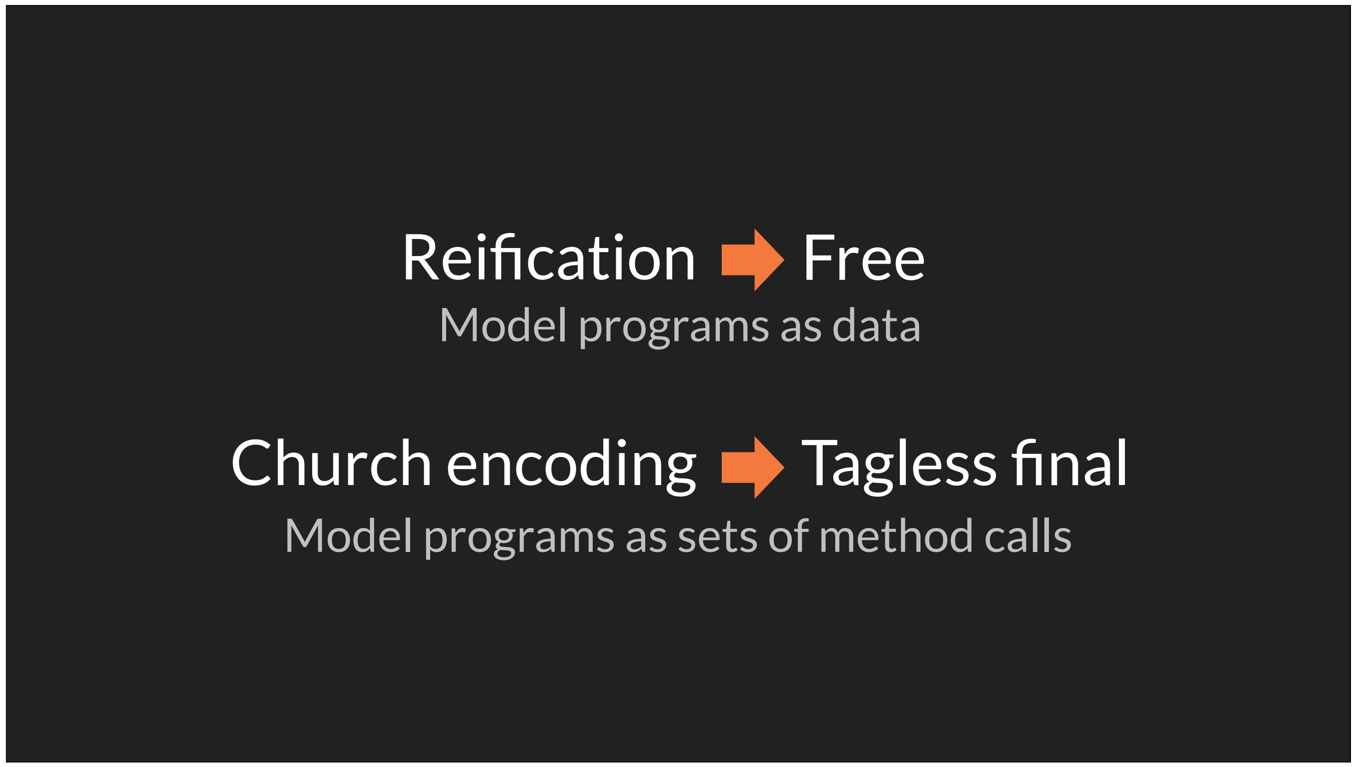

Reification
Model programs as data

Church encoding
Model programs as sets of method calls

The main approaches are:

- Reification, which means representing programs as data structures. In this approach, we represent a program as an AST or expression tree. An interpreter is a function that accepts a program as an input.

- Church encoding is an alternative to reification where we represent programs as sequences of abstract method calls. We’ll look at this approach towards the end of the talk.


Reification ➡ Free

Model programs as data

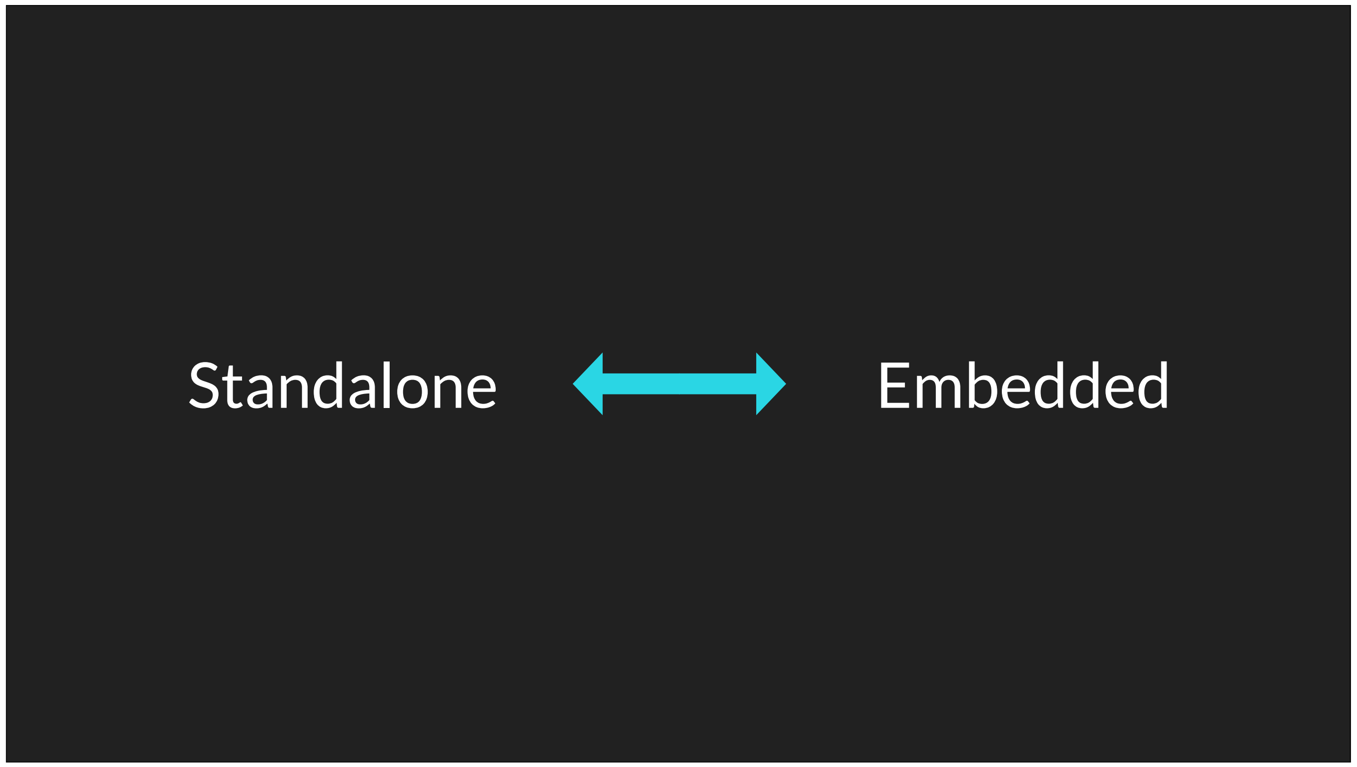
Church encoding ➡ Tagless final

Model programs as sets of method calls

Interestingly, if we look at various ways of creating reified DSLs, we eventually end up looking at the “free monad”.

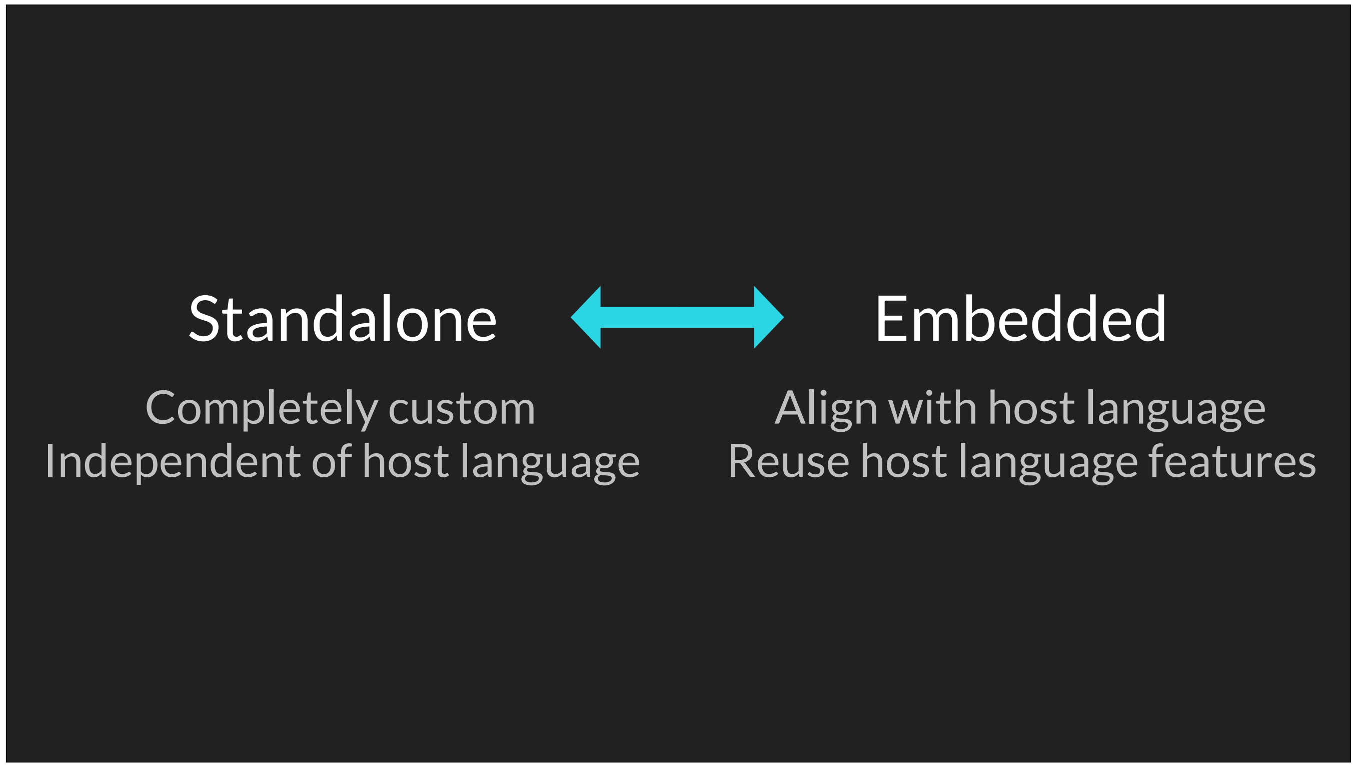
And if we consider various ways of church encoding DSLs, we eventually end up with “tagless final” encoding.

We’ll motivate and introduce both of these encodings towards the end of this talk.


Standalone ⟷ Embedded

As we go along, I also want you to consider another axis of classifying DSLs.

When we design and implement a DSL, we can choose how much we want it to lean on features from the host language (in our case Scala).



Standalone

Completely custom
Independent of host language

Embedded

Align with host language
Reuse host language features

At one end of the spectrum we can build a completely standalone language with its own parser, tooling, type system, and execution semantics. This gives us complete flexibility but is a huge amount of work.

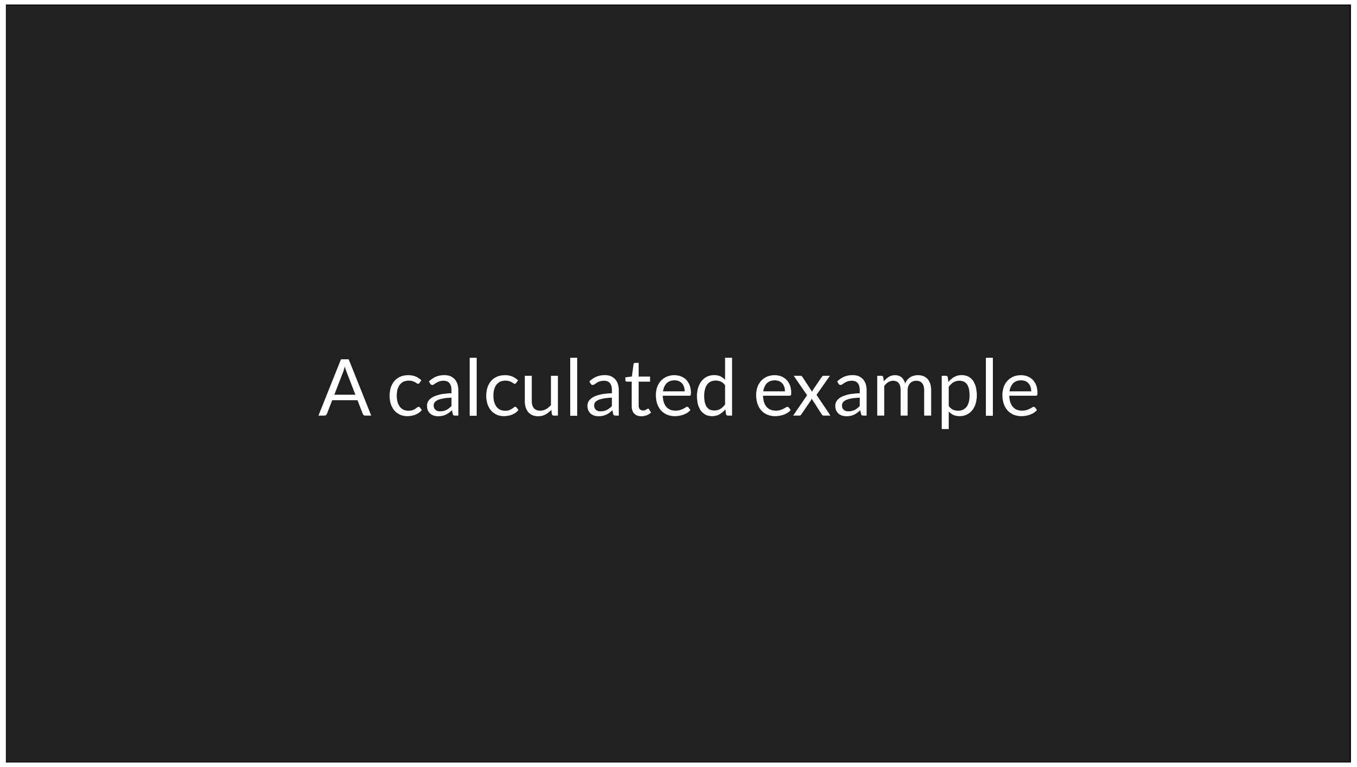
At the other end of the spectrum we can build various kinds of “embedded DSLs”. These are more like libraries that re-use features of Scala such as variable definitions, operator precedence, scoping rules, types, and implicits. The more features they re-use, the “deeper” the embedding.

We’ll focus on embedded languages in this talk because they are quicker to build and experiment with than standalone ones.

Reified encodings

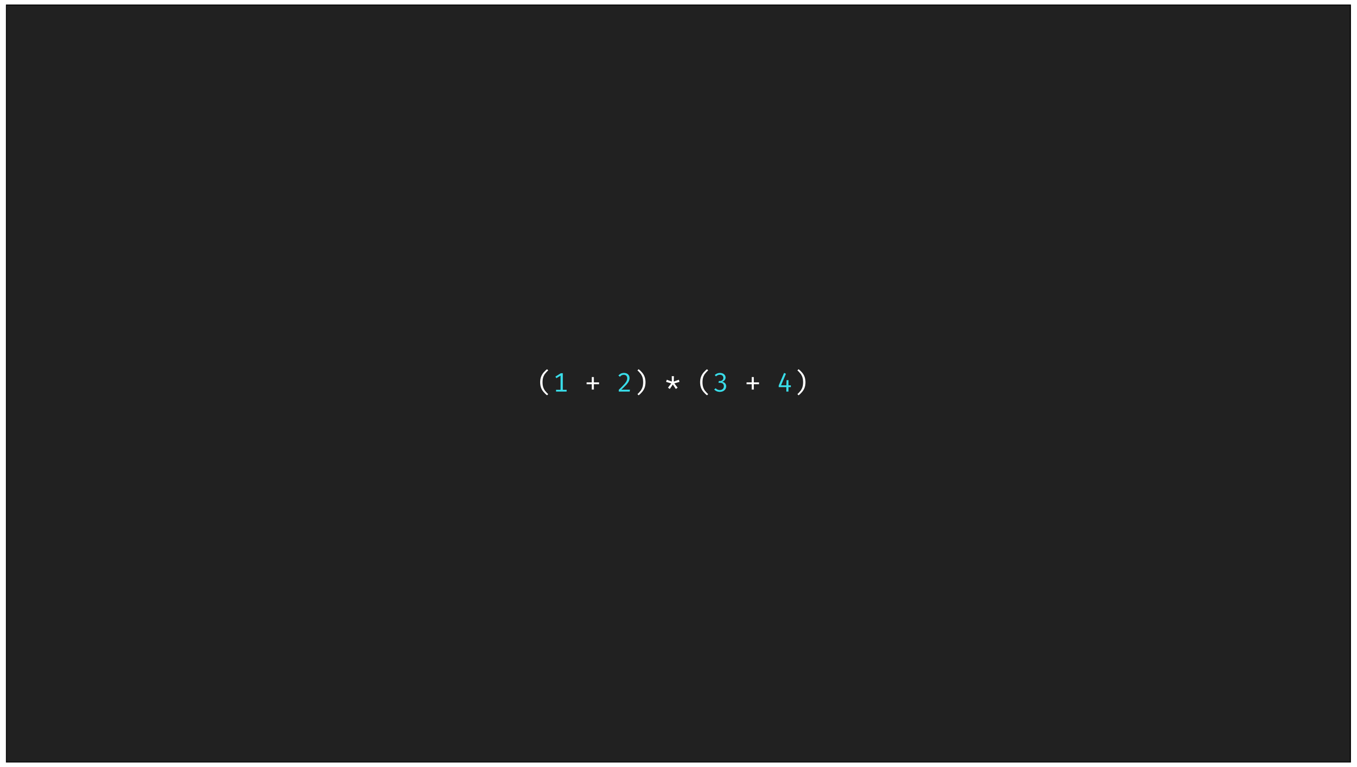
That’s the intro out of the way. We’ll spend the rest of the talk looking at various types of DSL encodings and seeing how they compare.

Let’s dive in to some reified encodings.



A calculated example

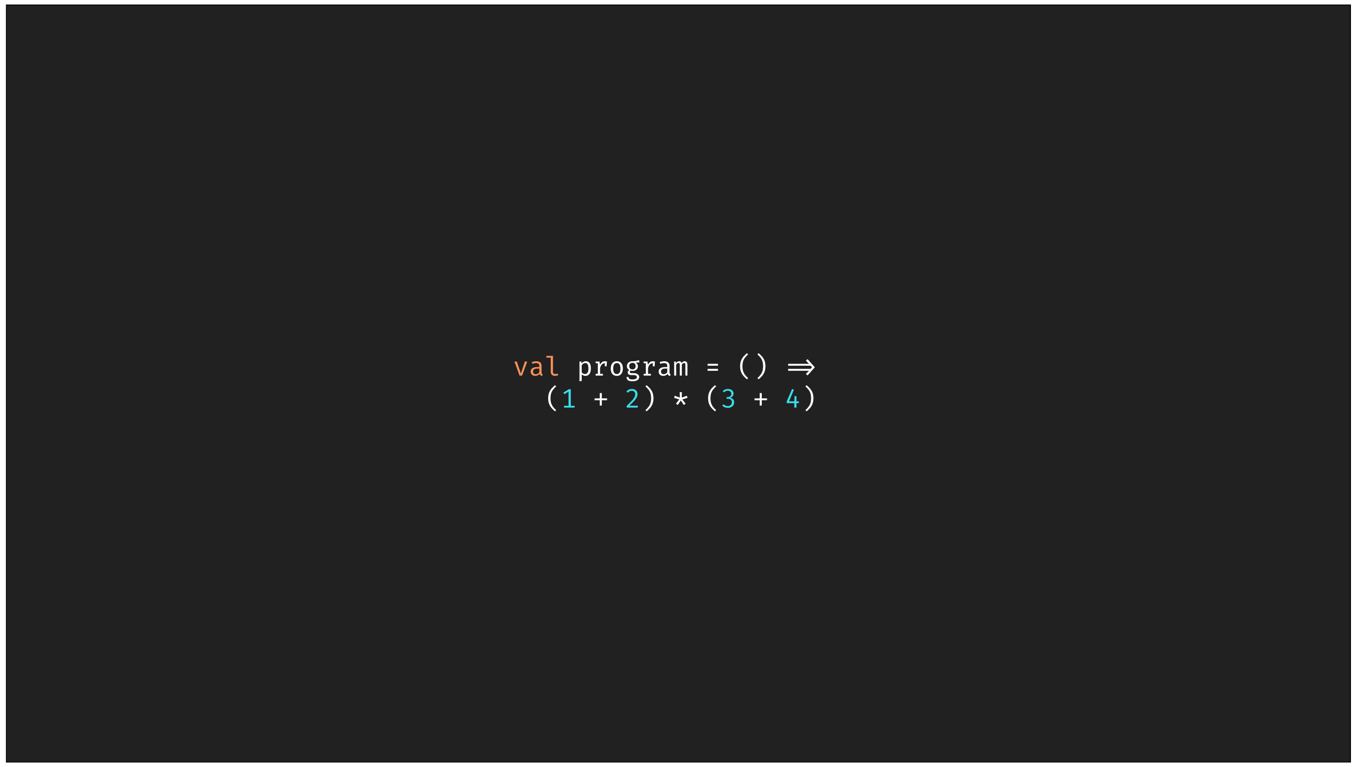
We’ll start by looking at “hello world” of DSLs: a calculator.



```
(1 + 2) * (3 + 4)
```

We’re going to build a DSL that allows us to add and multiply integers.

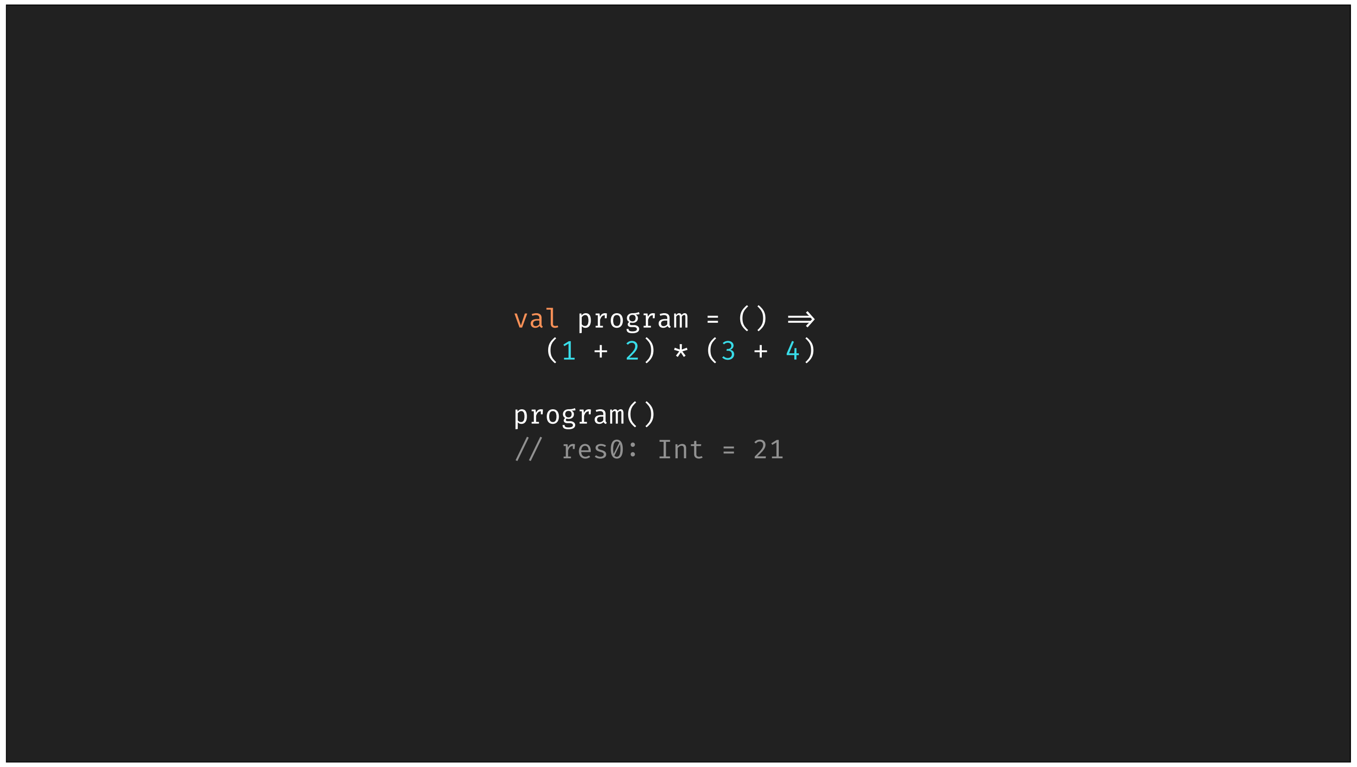
This is a good starting point because it neatly avoids discussion of type errors and the “correctness” of programs. We can always add and multiply numbers, and we always produce new numbers that we can add and multiply further.



```scala
val program = () ⇒
  (1 + 2) * (3 + 4)
```

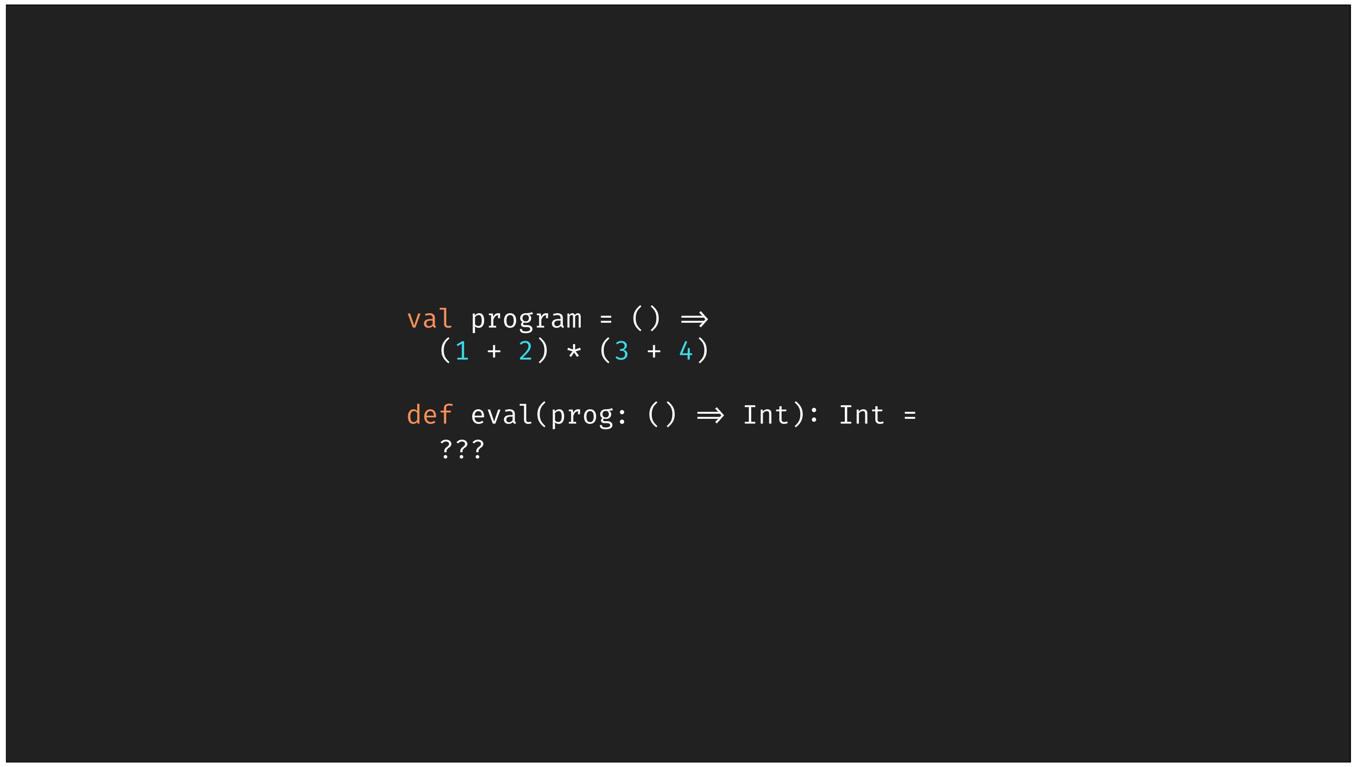
Let’s think about how we can represent “programs” of this type in Scala. The simplest representation is to wrap an expression in a function.

Functions tick all the boxes to identify as programs: they are inert values that represent a sequence of steps to execute, but do nothing on their own.



```
val program = () ⇒
  (1 + 2) * (3 + 4)

program()
// res0: Int = 21
```

In order to execute a function I have to call it. This runs the code inside and computes a result.



```
val program = () ⇒
  (1 + 2) * (3 + 4)

def eval(prog: () ⇒ Int): Int =
  ???
```

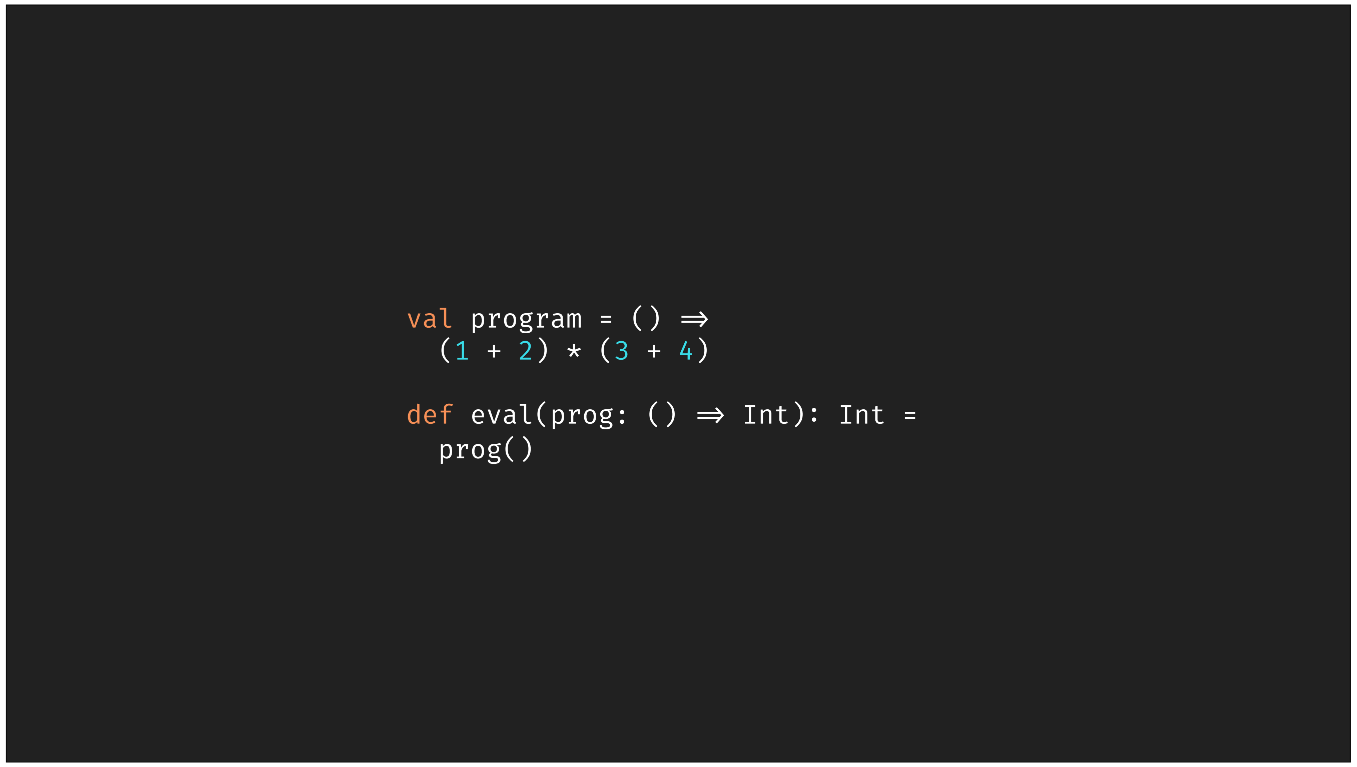
But rather than call the functions directly, let's create an "interpreter" to run them for us.

Here's an interpreter. It's a function called "eval". It accepts a program as a parameter and returns the result of running the program.

Let's run through a mental exercise here. I want you to consider two things:

1. How much do we know about `prog` within `eval()`? …and related…
2. How many different ways are there of running `prog` within `eval()`?

You should quickly realise that we actually know nothing about the program we're running. We can't even tell if it's adding and multiplying integers. It could be spawning threads and returning a thread count.



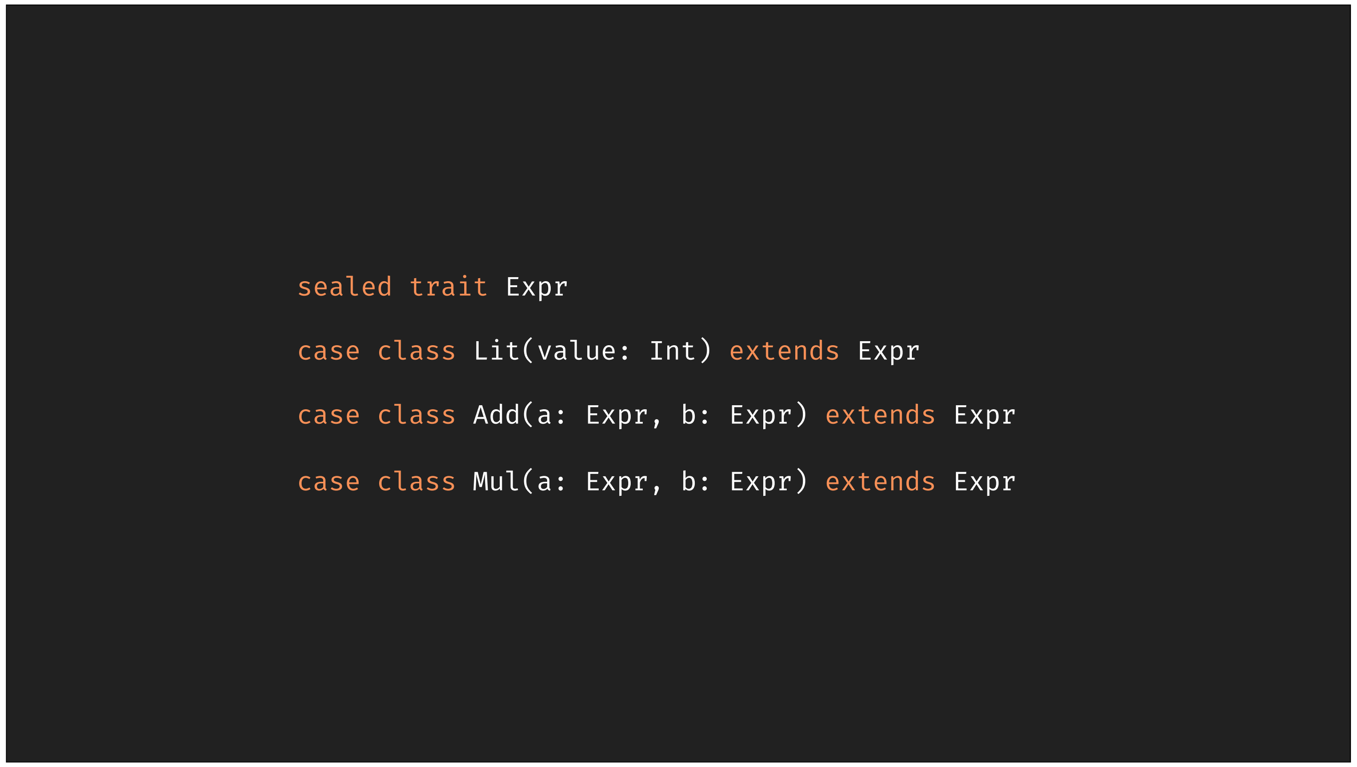
```scala
val program = () ⇒
  (1 + 2) * (3 + 4)

def eval(prog: () ⇒ Int): Int =
  prog()
```

Because we know nothing about `prog`, there's only one way we can implement `eval()`. We can call the function.

There are two problems here:

1. Scala functions are opaque. We can't see inside them to understand what they do.
2. Even if we could see inside them, Scala is such a general language, it would be very difficult to reason about the steps represented by `prog`.

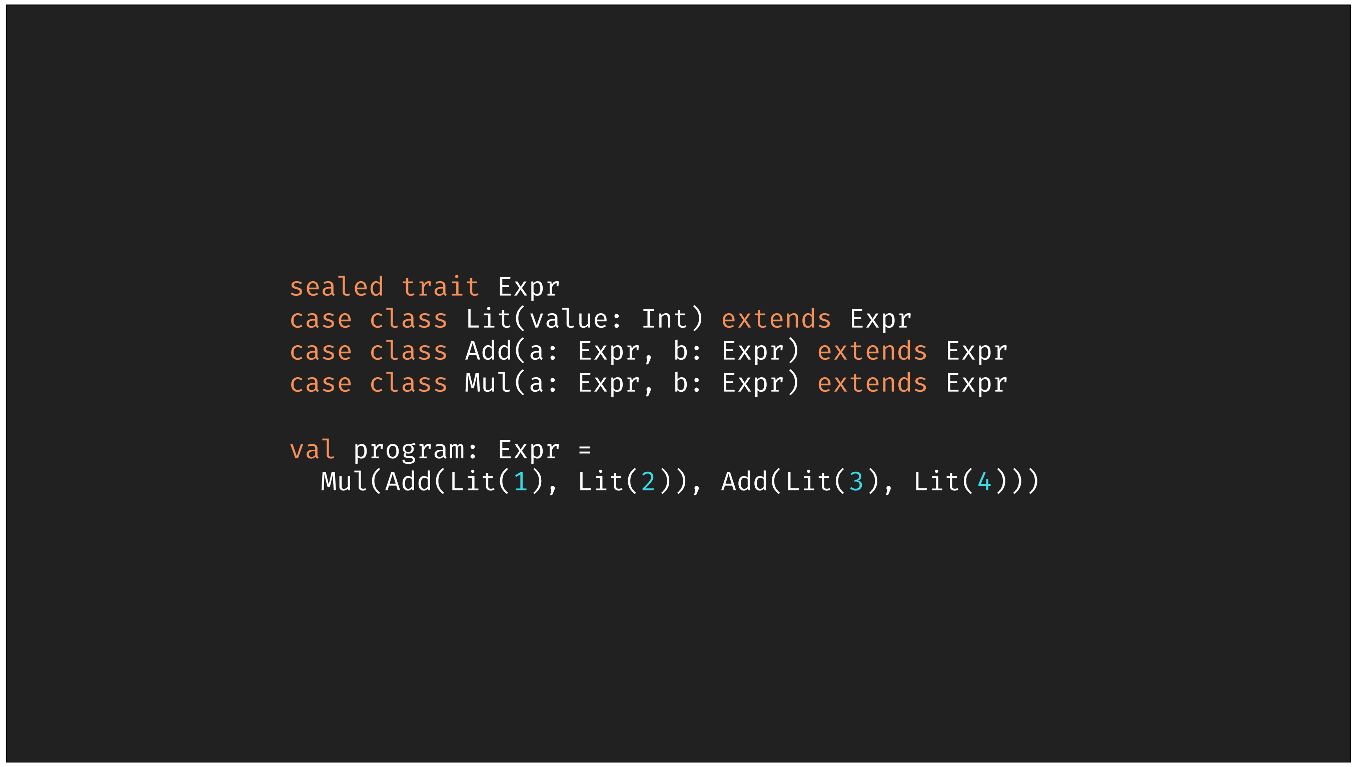
To counter these problems, let's start again but model our programs using a DSL.



```scala
sealed trait Expr

case class Lit(value: Int) extends Expr

case class Add(a: Expr, b: Expr) extends Expr

case class Mul(a: Expr, b: Expr) extends Expr
```

Here is one candidate encoding. A program is an “expression”, which is one of three concrete things:

- a “literal” that wraps up an integer;
- an “addition” that adds the results of two other expressions;
- a “multiplication” that multiplies the results of two other expressions.



```scala
sealed trait Expr
case class Lit(value: Int) extends Expr
case class Add(a: Expr, b: Expr) extends Expr
case class Mul(a: Expr, b: Expr) extends Expr

val program: Expr =
  Mul(Add(Lit(1), Lit(2)), Add(Lit(3), Lit(4)))
```

With this DSL we can represent the same sequence of steps we saw before.

The syntax is more verbose, but this is something we can easily fix using syntactic tricks such as symbolic method names and extension methods.

```scala
sealed trait Expr
case class Lit(value: Int) extends Expr
case class Add(a: Expr, b: Expr) extends Expr
case class Mul(a: Expr, b: Expr) extends Expr

val program: Expr =
  Mul(Add(Lit(1), Lit(2)), Add(Lit(3), Lit(4)))

def eval(expr: Expr): Int =
  expr match {
    case Lit(value) => value
    case Add(a, b)  => eval(a) + eval(b)
    case Mul(a, b)  => eval(a) * eval(b)
  }

eval(program)
// res0: Int = 21
```

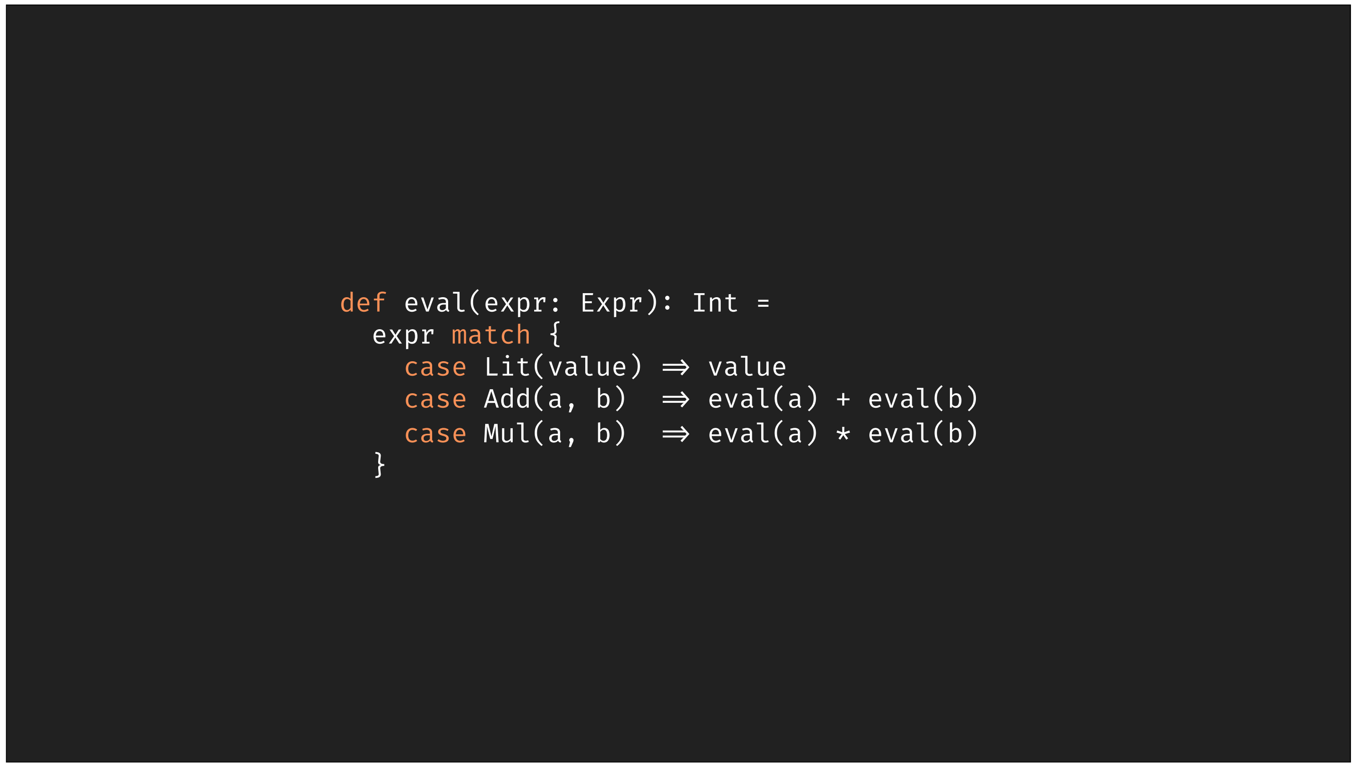
`program` represents the same calculation as the Scala function we had before. However, unlike it's functional counterpart we can no longer run it directly. To run it, we need to introduce an interpreter.

Because `Expr` is an algebraic data type with a fixed set of subtypes, it is easy to implement `eval` using pattern matching:

- if we see a literal, we return the value inside;
- if we see an addition or multiplication, we evaluate the subexpressions and return the result.

The end result is a complete traversal of the expression tree, yielding the correct output.

This code is much more verbose than our Scala function. What do we get from it?



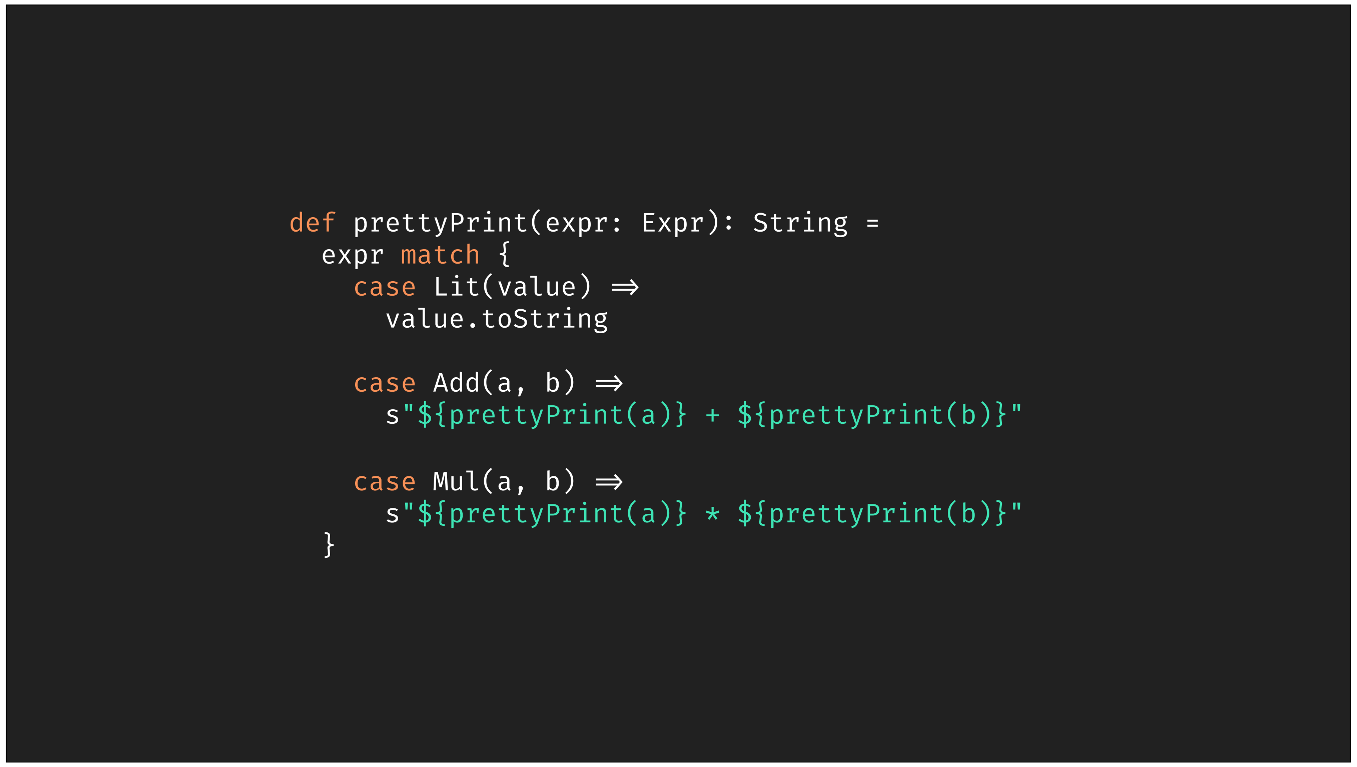
```scala
def eval(expr: Expr): Int =
  expr match {
    case Lit(value) ⇒ value
    case Add(a, b)  ⇒ eval(a) + eval(b)
    case Mul(a, b)  ⇒ eval(a) * eval(b)
  }
```

We get lots of benefits in terms of modularity.

The `eval()` method we've written is just one way of interpreting a program.

```scala
def evalAsync(expr: Expr): Future[Int] =
  expr match {
    case Lit(value) ⇒
      Future.successful(value)

    case Add(a, b) ⇒
      val x = evalAsync(a)
      val y = evalAsync(b)
      x.flatMap(a ⇒ y.map(b ⇒ a + b))

    case Mul(a, b) ⇒
      val x = evalAsync(a)
      val y = evalAsync(b)
      x.flatMap(a ⇒ y.map(b ⇒ a * b))
  }
```

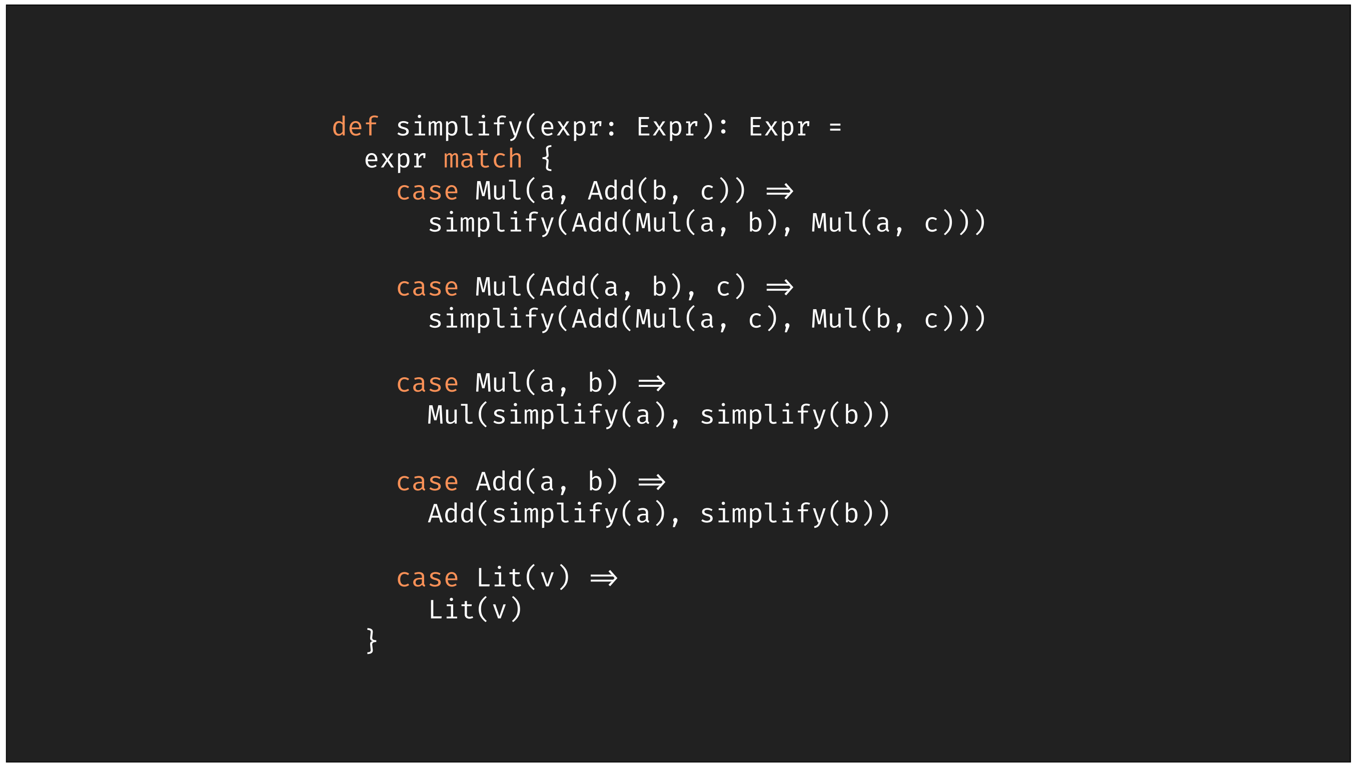
We can write an asynchronous interpreter that evaluates the branches in our DSL in parallel.



```scala
def prettyPrint(expr: Expr): String =
  expr match {
    case Lit(value) ⇒
      value.toString

    case Add(a, b) ⇒
      s"${prettyPrint(a)} + ${prettyPrint(b)}"

    case Mul(a, b) ⇒
      s"${prettyPrint(a)} * ${prettyPrint(b)}"
  }
```

We can write a pretty printer that traverses the expression and prints it in string form.

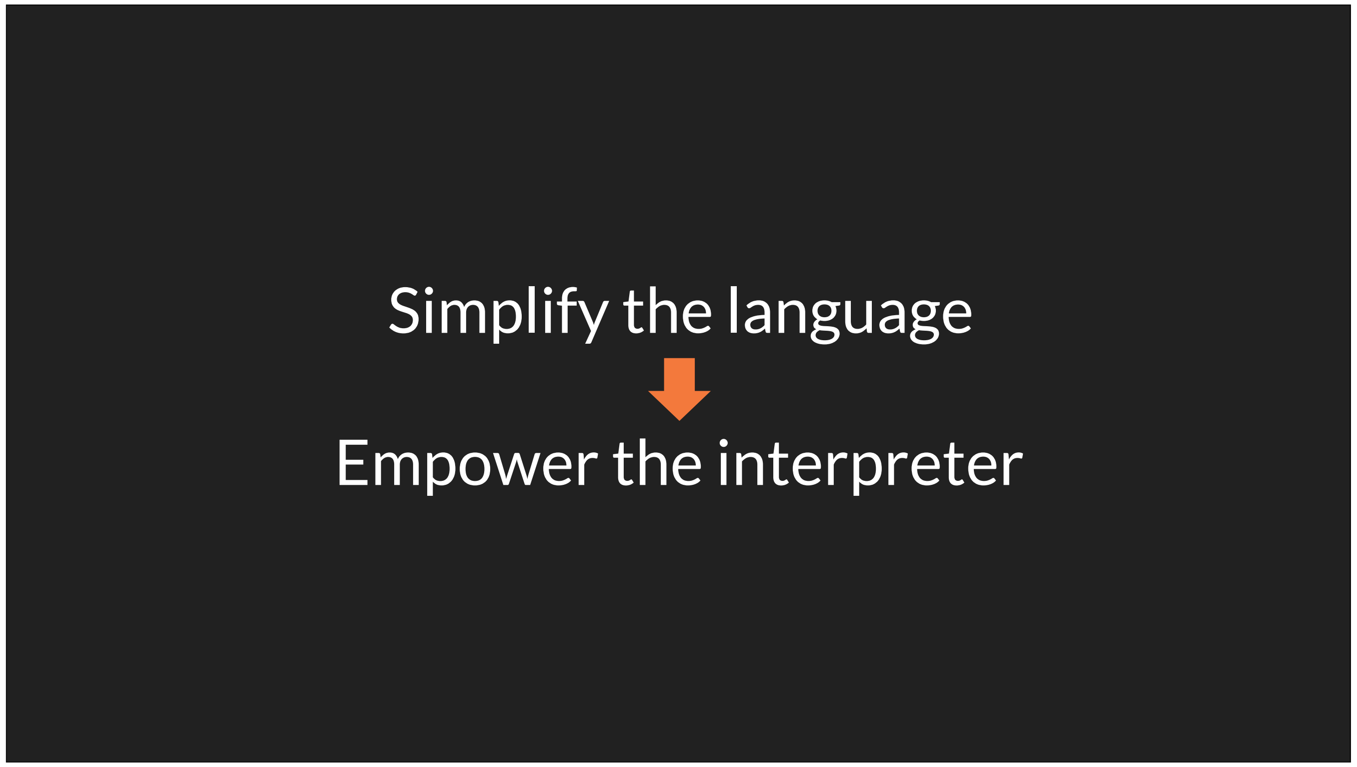
(Note: This implementation doesn't deal with operator precedence correctly. We'll leave that as an exercise to the viewer.)



```scala
def simplify(expr: Expr): Expr =
  expr match {
    case Mul(a, Add(b, c)) ⇒
      simplify(Add(Mul(a, b), Mul(a, c)))

    case Mul(Add(a, b), c) ⇒
      simplify(Add(Mul(a, c), Mul(b, c)))

    case Mul(a, b) ⇒
      Mul(simplify(a), simplify(b))

    case Add(a, b) ⇒
      Add(simplify(a), simplify(b))

    case Lit(v) ⇒
      Lit(v)
  }
```

We can even write a “compiler” that alters the structure of the program.

This `simplify()` method inspects and reorders expressions to “multiply out the brackets”. If we start with a multiplication of two additions, we end up with an addition of several multiplications instead.



Simplify the language

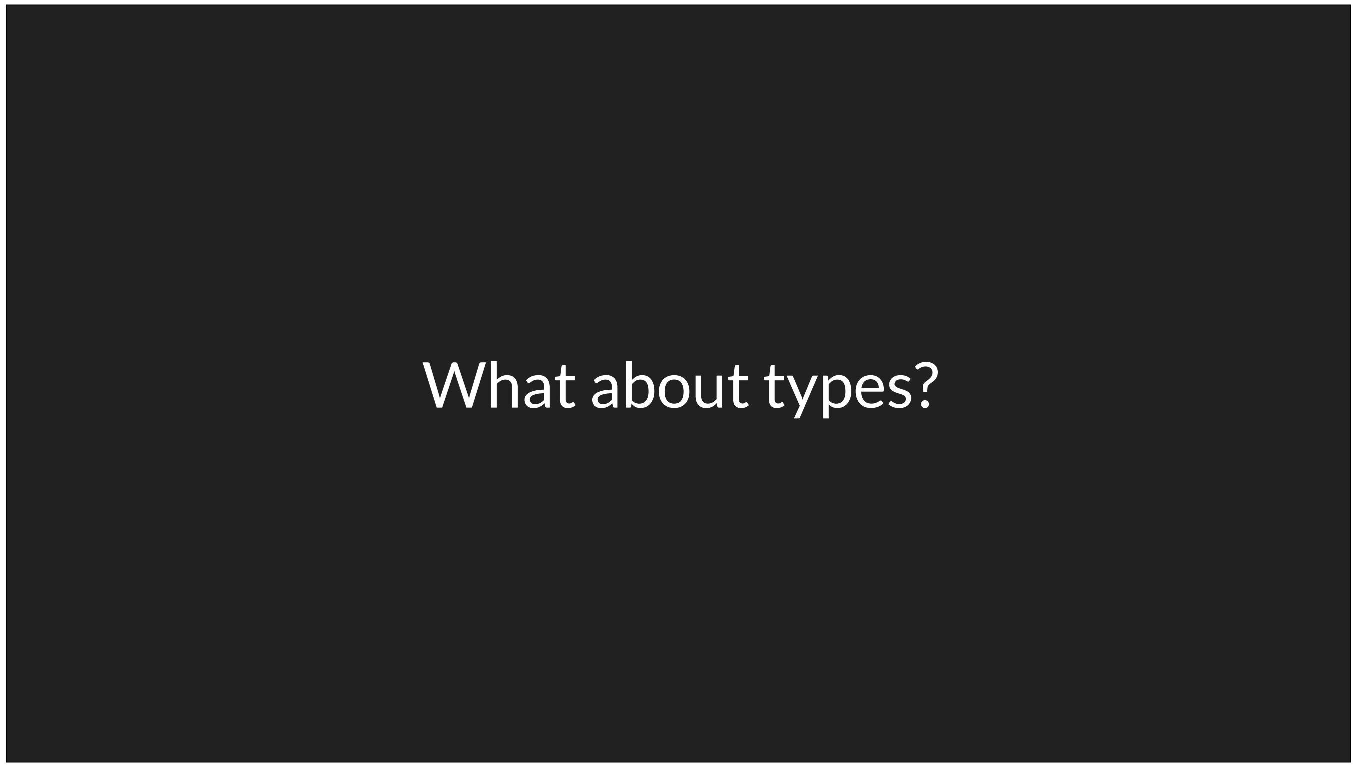
⬇

Empower the interpreter

We get all of this flexibility precisely because we have simplified the input language. By constraining the input to our interpreter, we open up more options for its implementation.

As a wise man once said, “constraints liberate, liberty constrains”.

(This wise man is Runar Bjarnasson, talking in a keynote at Scala World a couple of years back. I’ve included a link at the end of the talk.)



What about types?

That example provided a simple verification of the tenets we set out at the start of the talk.

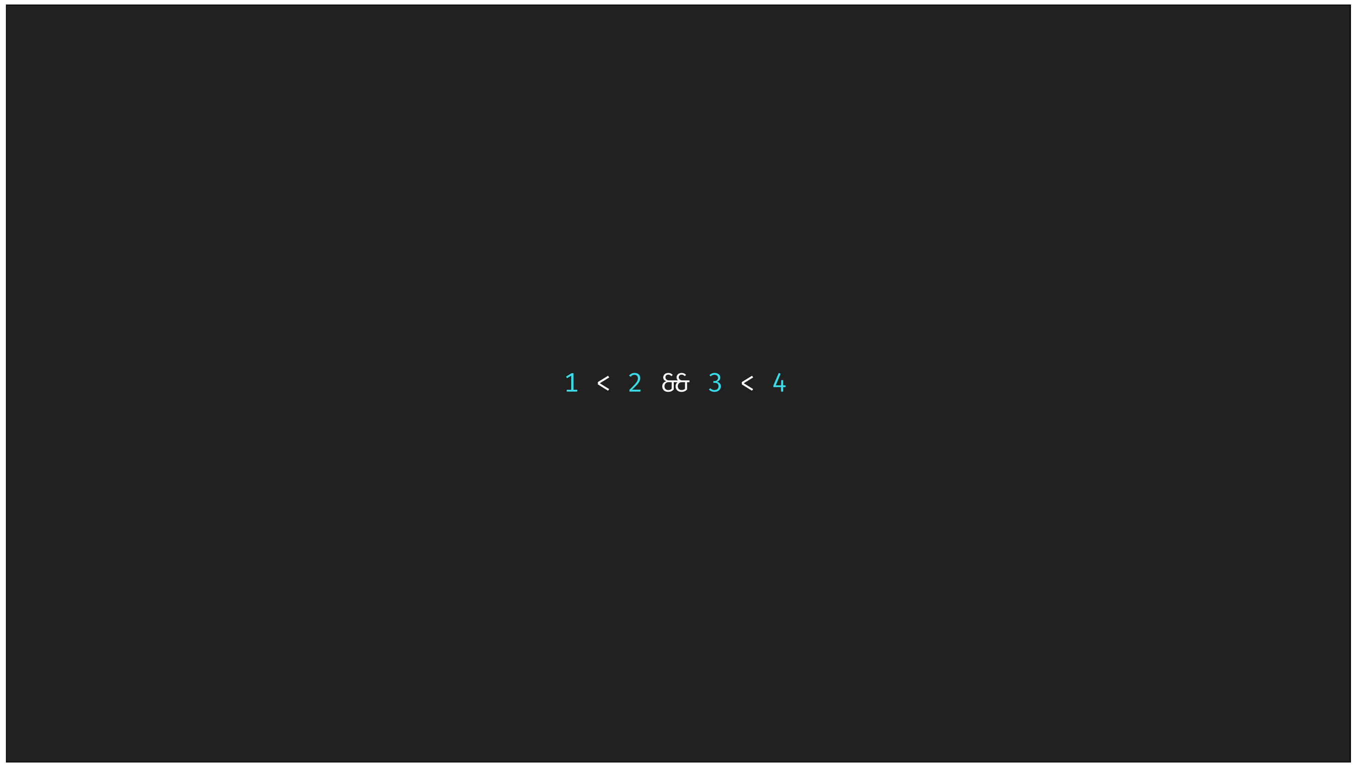
Now let’s see what happens when we make things more complicated. Let’s introduce the notion of types and type errors.

There are a few ways of handling this…

Untyped DSLs

The first route is to make our DSL “untyped”.

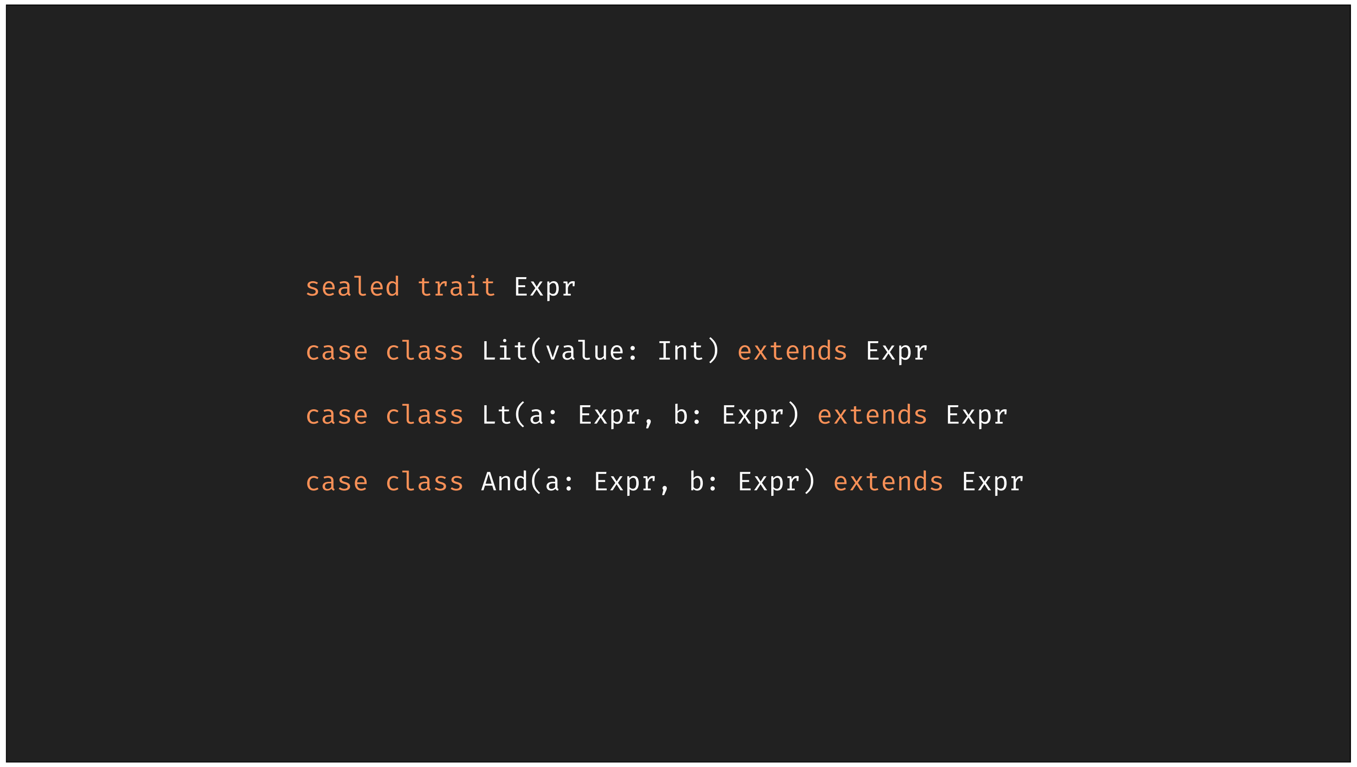
This means we ignore potential type errors in our programs and handle them in the interpreter.



```
1 < 2 && 3 < 4
```

To discuss this we'll modify our problem domain a little by introducing comparisons and boolean expressions.

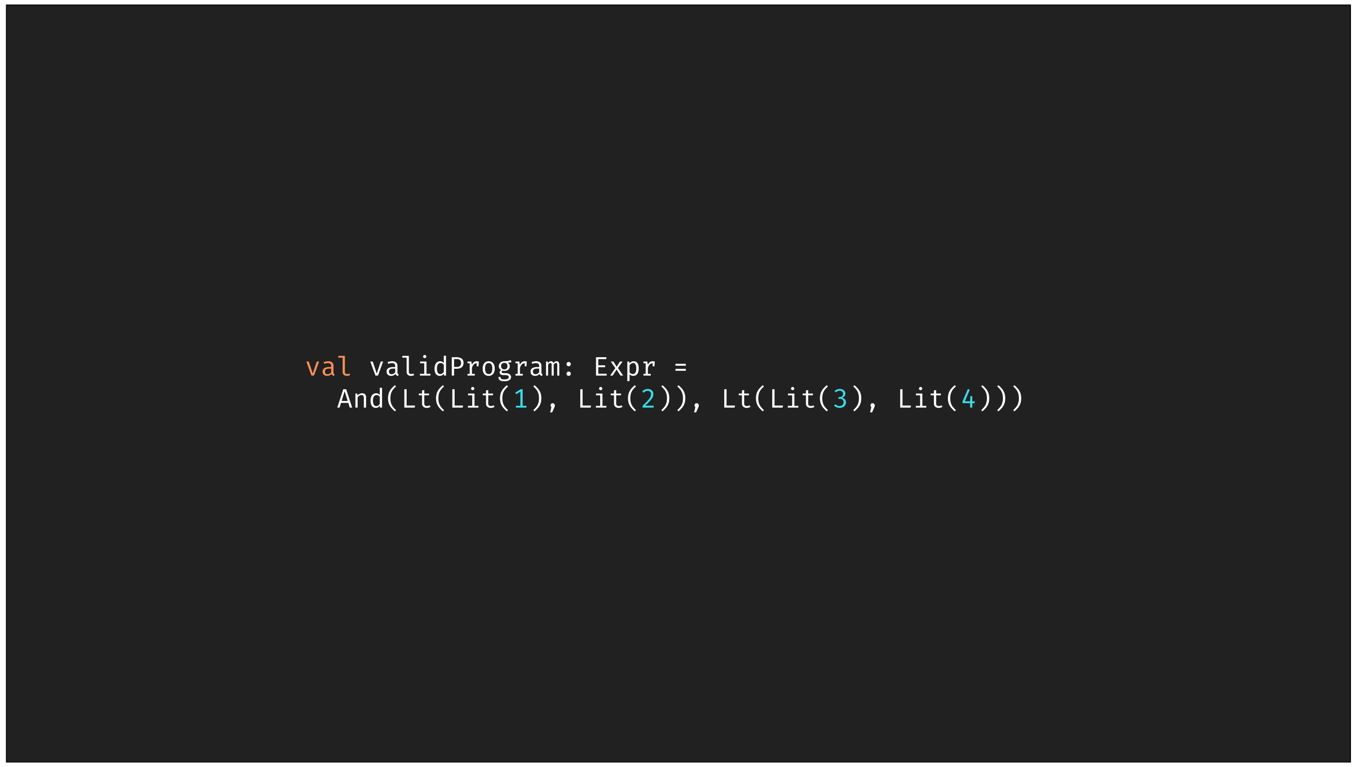
Now we're dealing with two types of data: integers and booleans.



```scala
sealed trait Expr

case class Lit(value: Int) extends Expr

case class Lt(a: Expr, b: Expr) extends Expr

case class And(a: Expr, b: Expr) extends Expr
```

Here’s a modified `Expr` type that is capable of representing these programs.

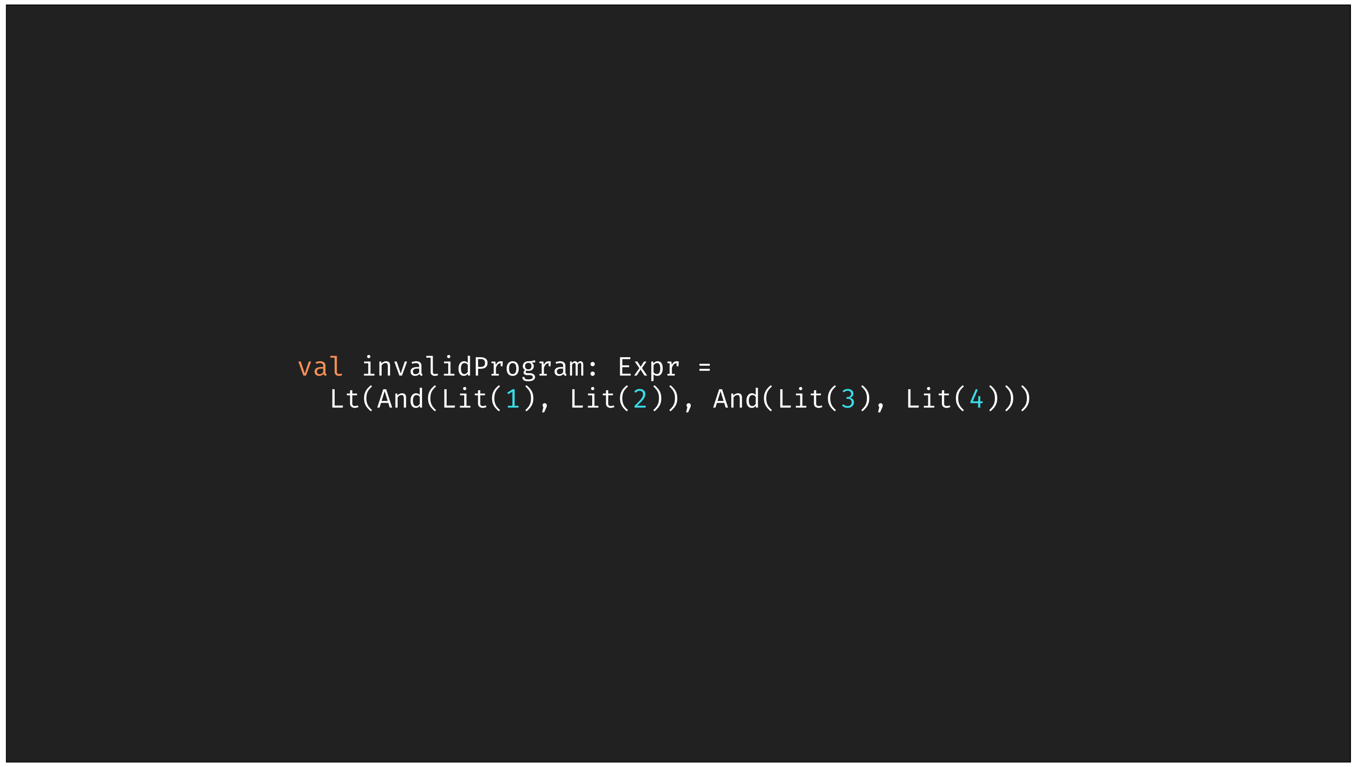
All we’ve done is swap the addition operator for a “less than”, and the multiplication operator for a Boolean “and”.



```
val validProgram: Expr =
  And(Lt(Lit(1), Lit(2)), Lt(Lit(3), Lit(4)))
```

We can represent valid programs in this language exactly as before.

However…

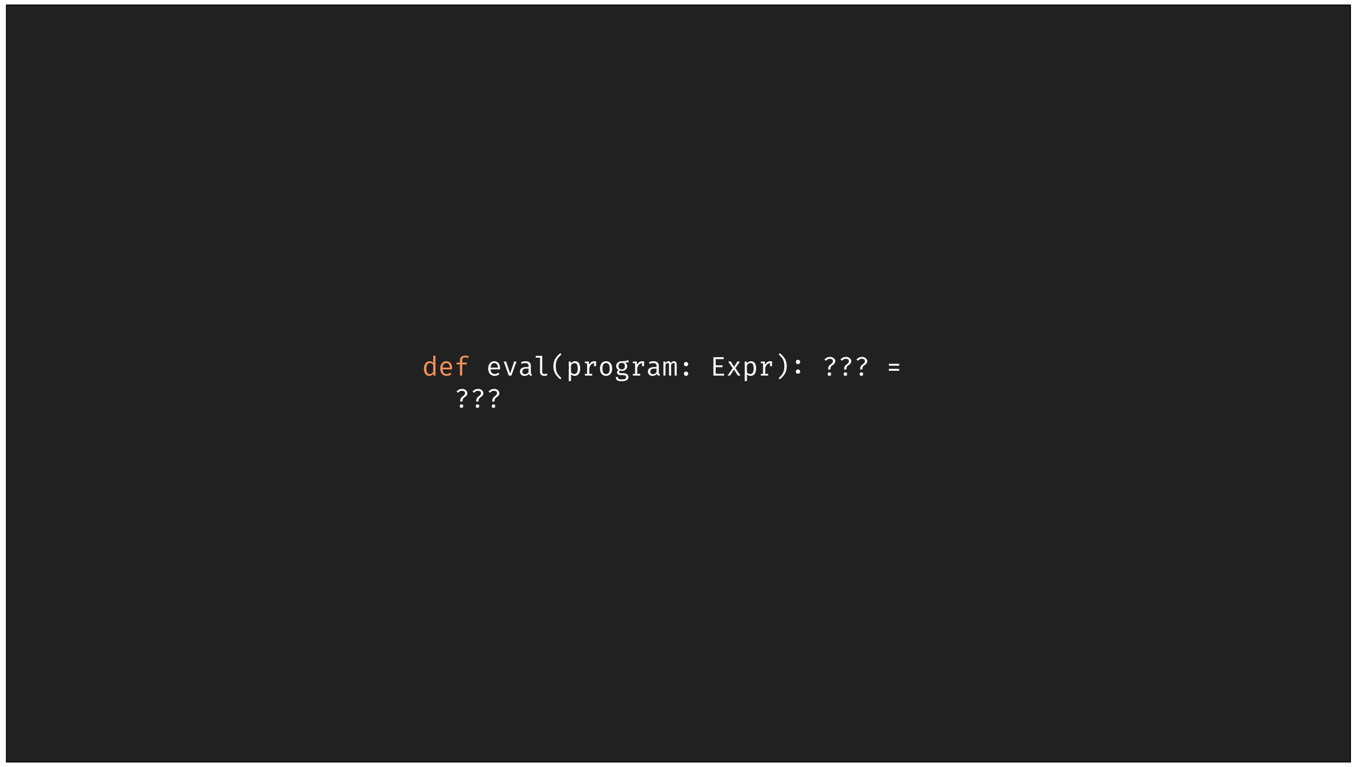


```scala
val invalidProgram: Expr =
  Lt(And(Lit(1), Lit(2)), And(Lit(3), Lit(4)))
```

We can also represent invalid expressions.

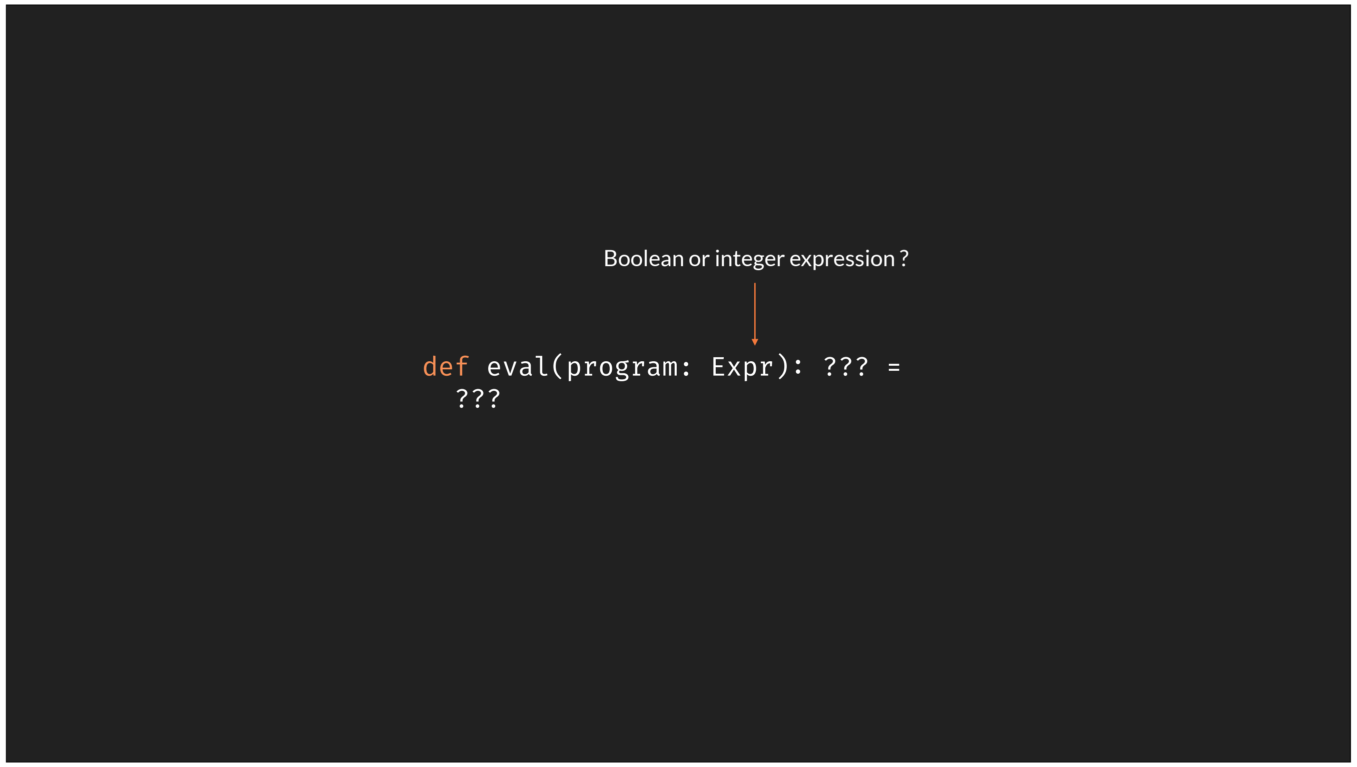
Here I'm trying to apply the “and” operator to two pairs of integers, and compare the results with“less than”.

This is a completely valid program according to the types in our DSL. However, the semantics of our DSL have diverged somewhat from the semantics of Scala where such operations are illegal.



```
def eval(program: Expr): ??? =
  ???
```

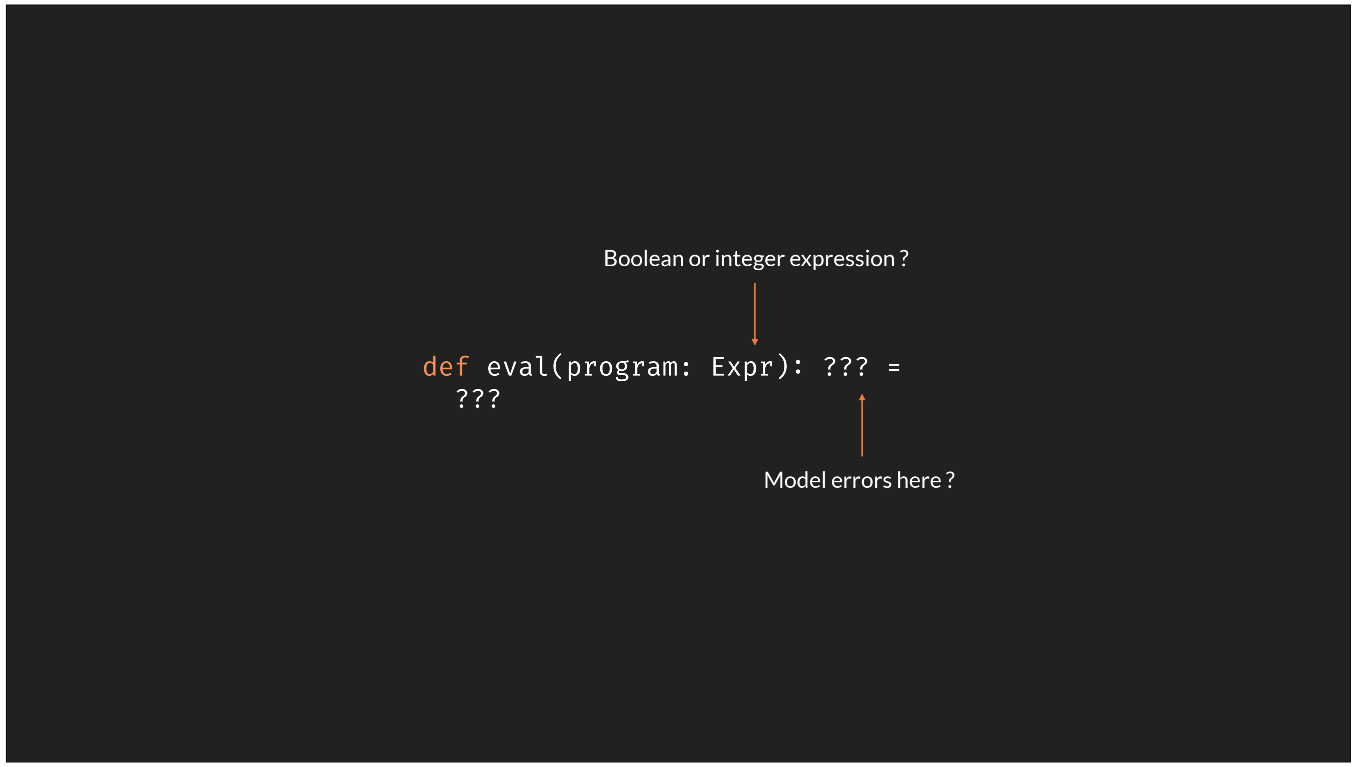
The differences between our DSL and Scala cause complications when we're implementing our interpreter. There are at least three things to handle…



Boolean or integer expression ?

```
def eval(program: Expr): ??? =
  ???
```

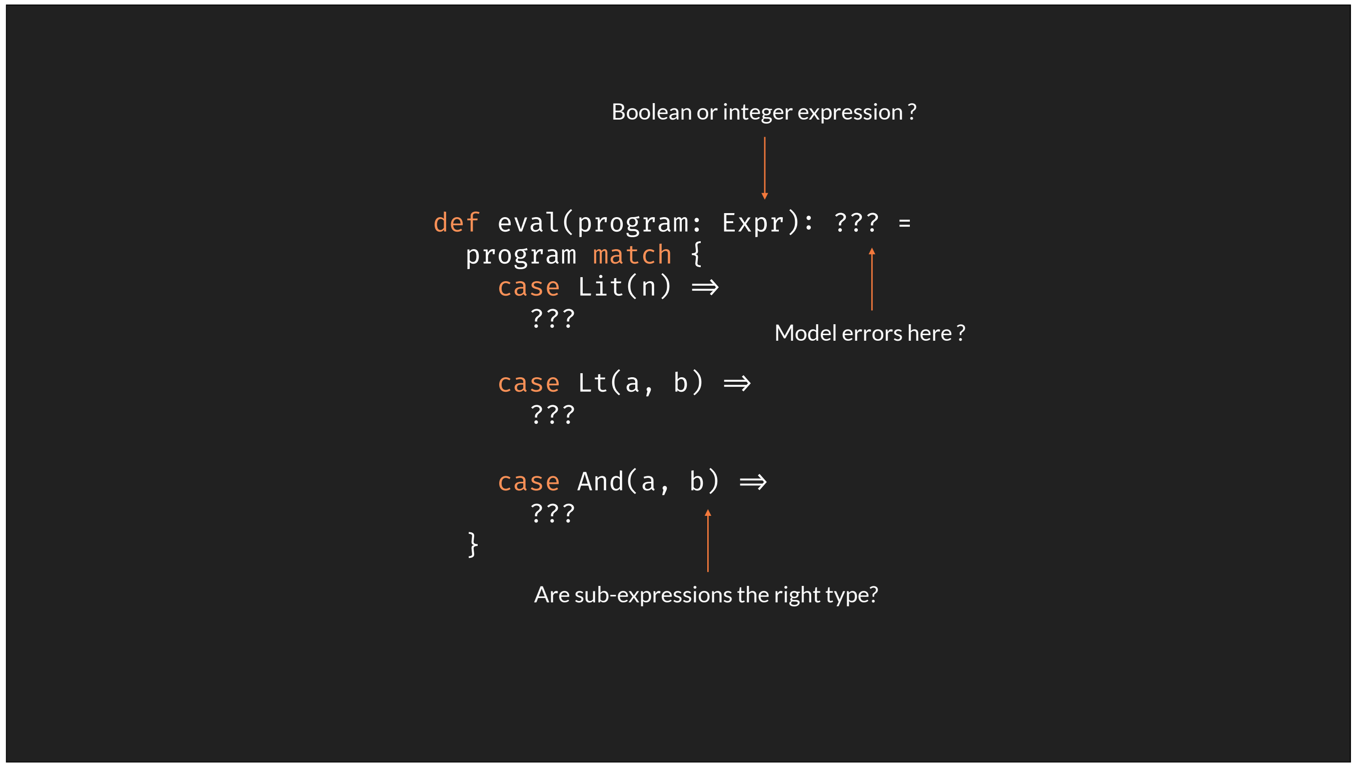
First, the input expression might yield one of two result types. We need to deal with this in the type signature for `eval()`.



```
def eval(program: Expr): ??? =
  ???
```

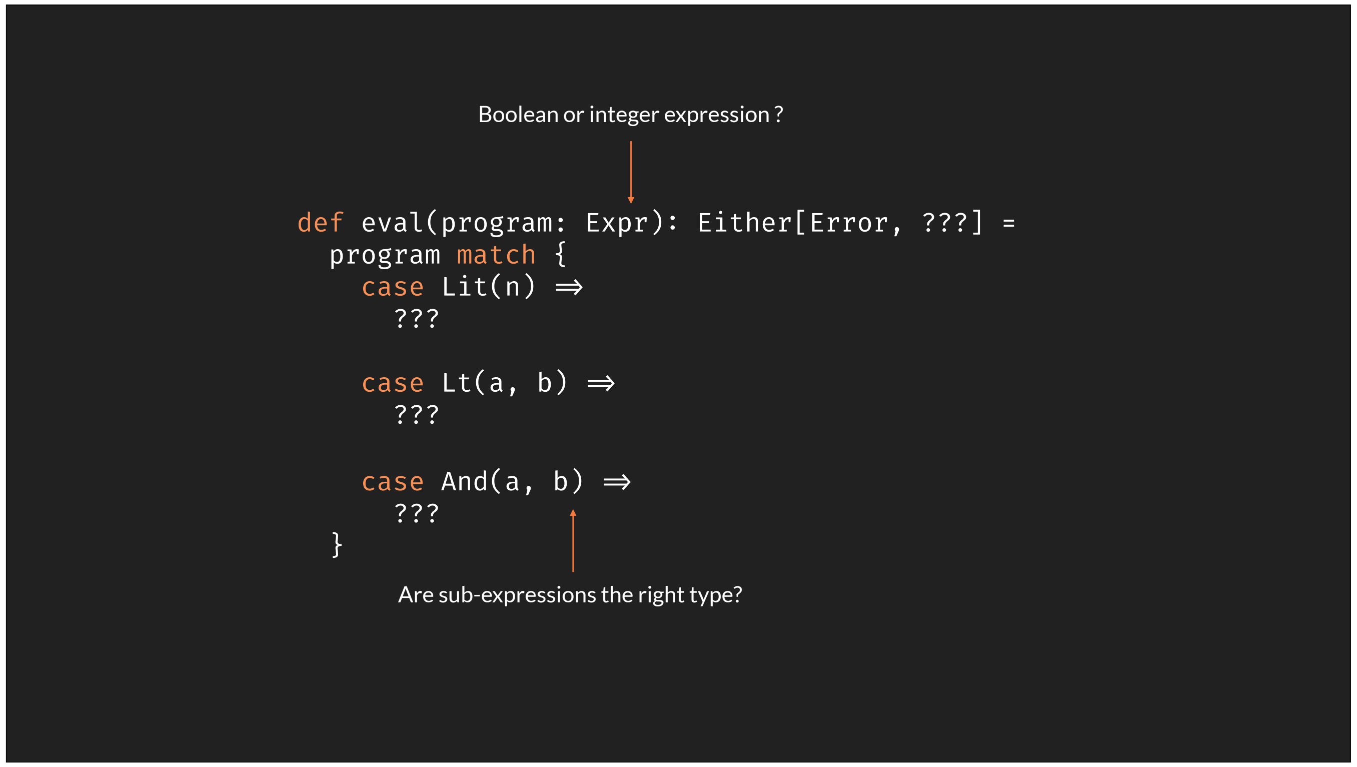
Second, we need a way of handling possible type errors.

We *could* do what Javascript does, which is to define all infix operators for all parameter types. However, if you've probably seen some of the weird semantics Javascript operators have, you may decide that it's easier to have some notion of runtime errors in our interpreter.


Boolean or integer expression ?
Model errors here ?
Are sub-expressions the right type?

```
def eval(program: Expr): ??? =
  program match {
    case Lit(n) ⇒
      ???

    case Lt(a, b) ⇒
      ???

    case And(a, b) ⇒
      ???
  }
```

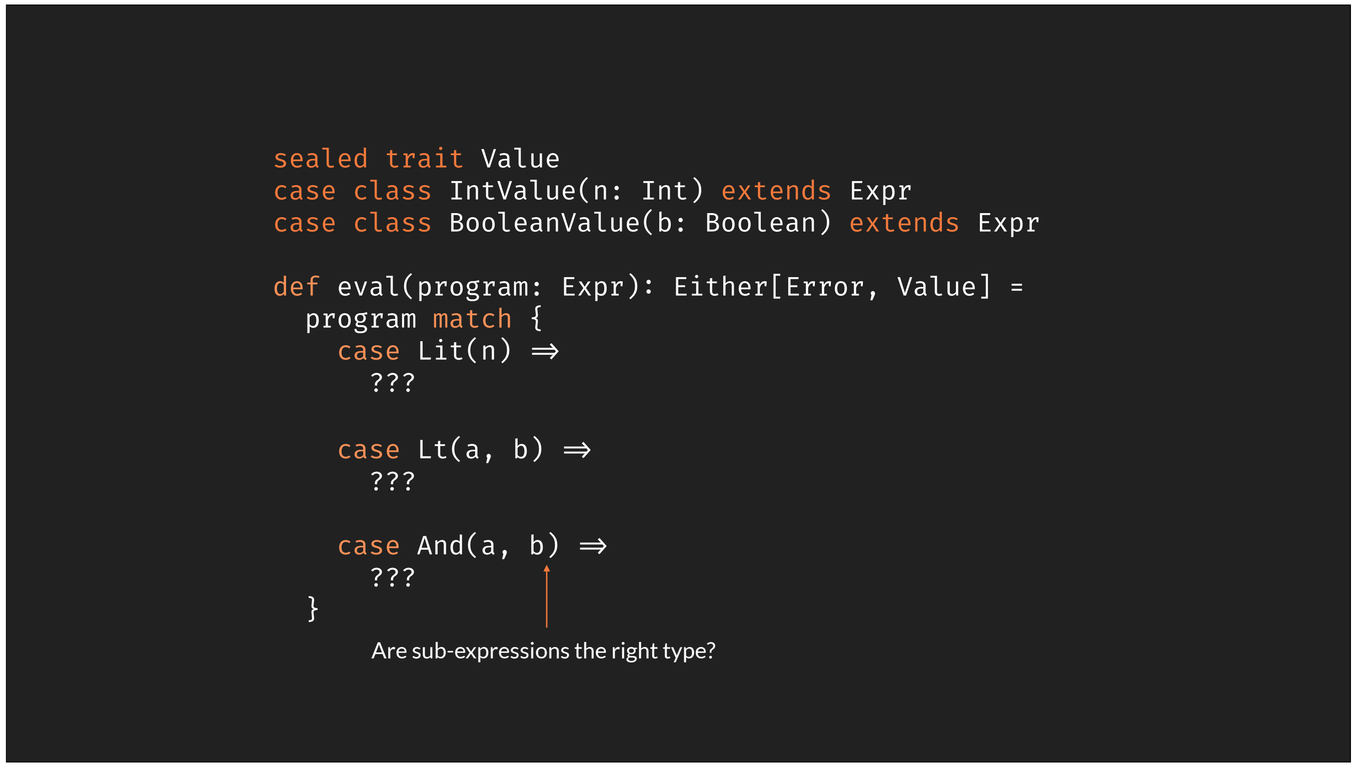
Finally, if we expand the code out a bit, when we're interpreting "less than" and "and" operators, we have to verify that the operands yield values of the right type.


Boolean or integer expression ?
Are sub-expressions the right type?

```
def eval(program: Expr): Either[Error, ???] =
  program match {
    case Lit(n) ⇒
      ???

    case Lt(a, b) ⇒
      ???

    case And(a, b) ⇒
      ???
  }
```

We can represent possible runtime errors by making our interpreter yield an `Either`.

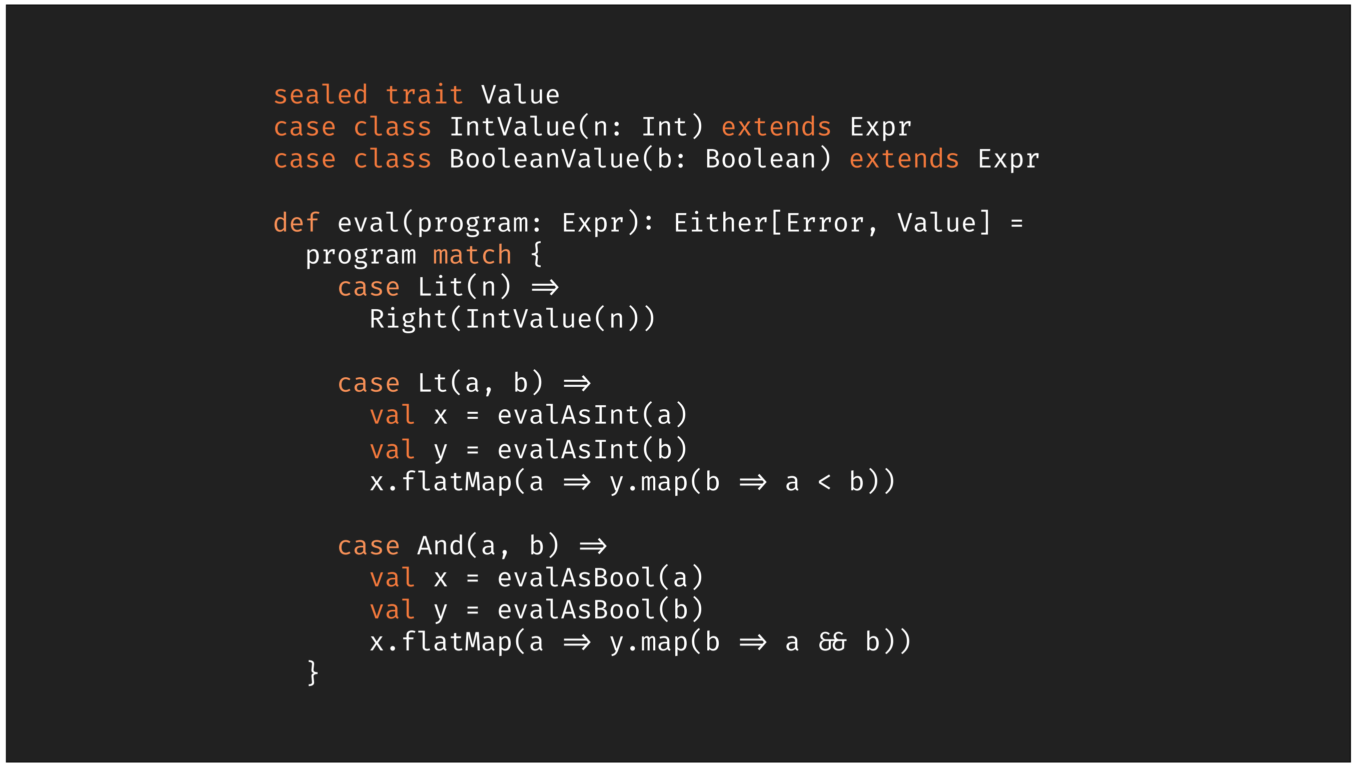
We’ll invent some error type here. It could either be a type alias for `String` or something with more structure to it.



```
sealed trait Value
case class IntValue(n: Int) extends Expr
case class BooleanValue(b: Boolean) extends Expr

def eval(program: Expr): Either[Error, Value] =
  program match {
    case Lit(n) ⇒
      ???

    case Lt(a, b) ⇒
      ???

    case And(a, b) ⇒
      ???
  }
```

Are sub-expressions the right type?

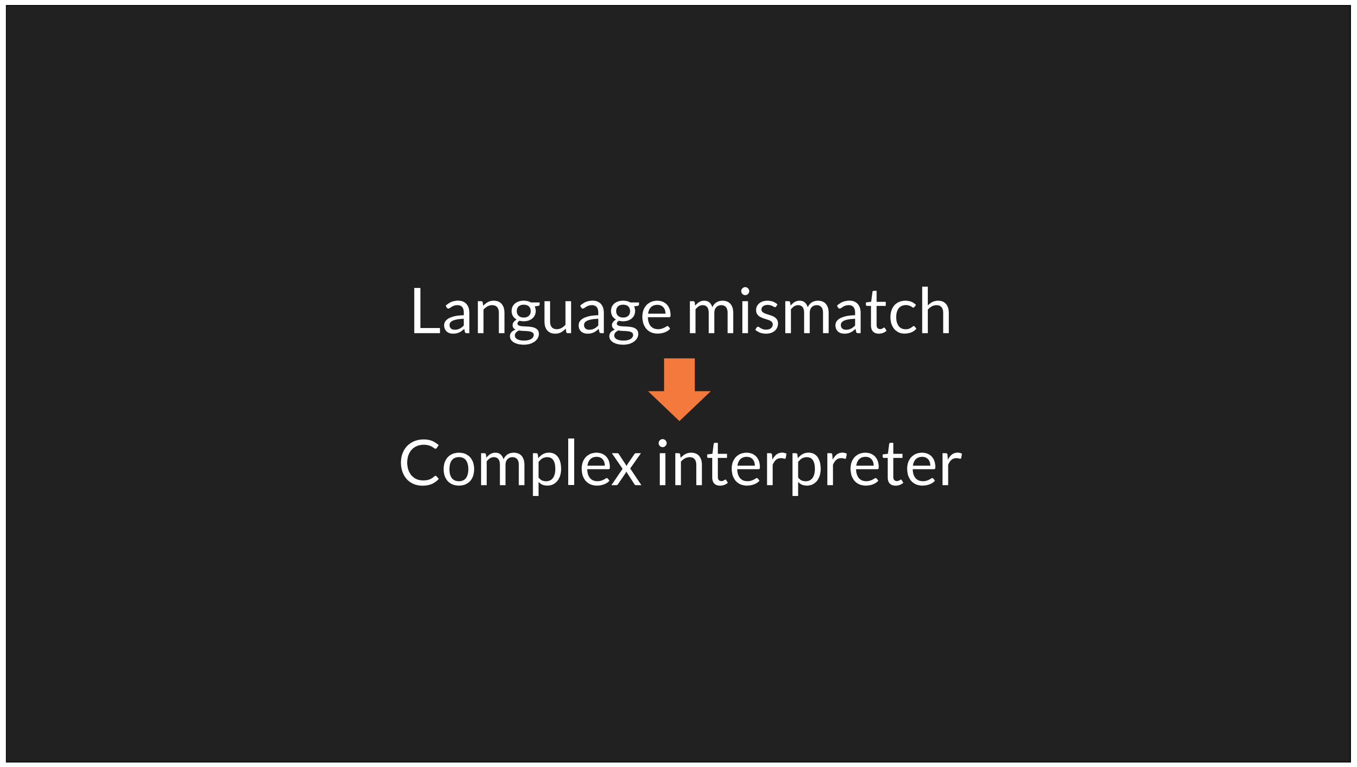
We can handle the disjunction of result types by introducing an algebraic data type to wrap up an integer *or* a boolean.



```scala
sealed trait Value
case class IntValue(n: Int) extends Expr
case class BooleanValue(b: Boolean) extends Expr

def eval(program: Expr): Either[Error, Value] =
  program match {
    case Lit(n) ⇒
      Right(IntValue(n))

    case Lt(a, b) ⇒
      val x = evalAsInt(a)
      val y = evalAsInt(b)
      x.flatMap(a ⇒ y.map(b ⇒ a < b))

    case And(a, b) ⇒
      val x = evalAsBool(a)
      val y = evalAsBool(b)
      x.flatMap(a ⇒ y.map(b ⇒ a && b))
  }
```

Finally, when interpreting “and” or “less than”, we can use some helper methods to evaluate the operands and pattern match on the resulting value to pull out a value has the correct type.

We’ll leave the implementation of `evalAsInt()` and `evalAsBool()` as an exercise for the viewer. There are also complete implementations in an SBT project accompanying these slides on Github.



Language mismatch
⬇
Complex interpreter

The key take-home point here is that there's an impedance mismatch between our DSL (which is untyped) and Scala (which is most definitely typed).

The mismatch shakes out as extra lines of code in our interpreter.

Can this be simplified?

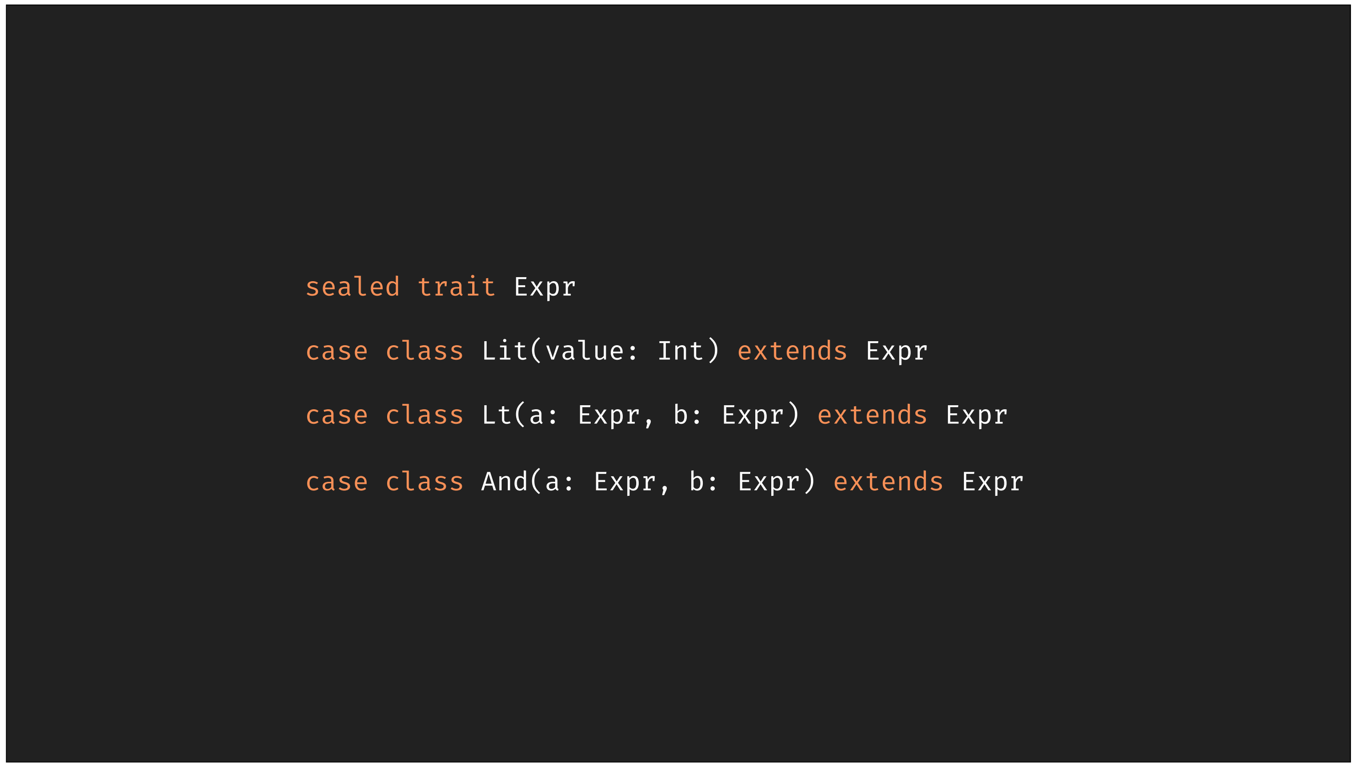
A sensible question is: “Can we make the interpreter simpler?”

The answer is: “Yes we can, by reducing the distance between our DSL and Scala”.

Typed DSLs

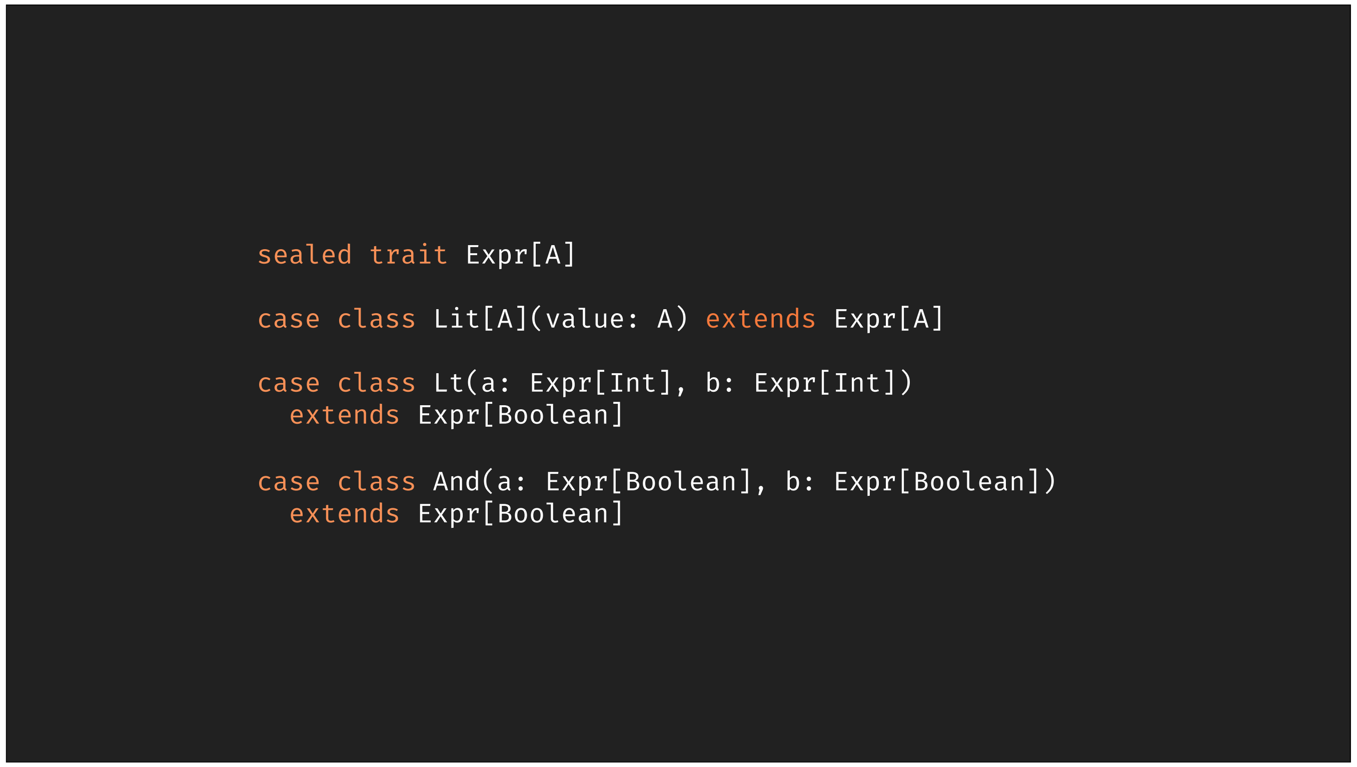
Let’s make our DSL typed.

We’ll use a subset of Scala’s type system, to keep the DSL simple and provide a clean mapping onto our host language in the interpreter.



```scala
sealed trait Expr

case class Lit(value: Int) extends Expr

case class Lt(a: Expr, b: Expr) extends Expr

case class And(a: Expr, b: Expr) extends Expr
```

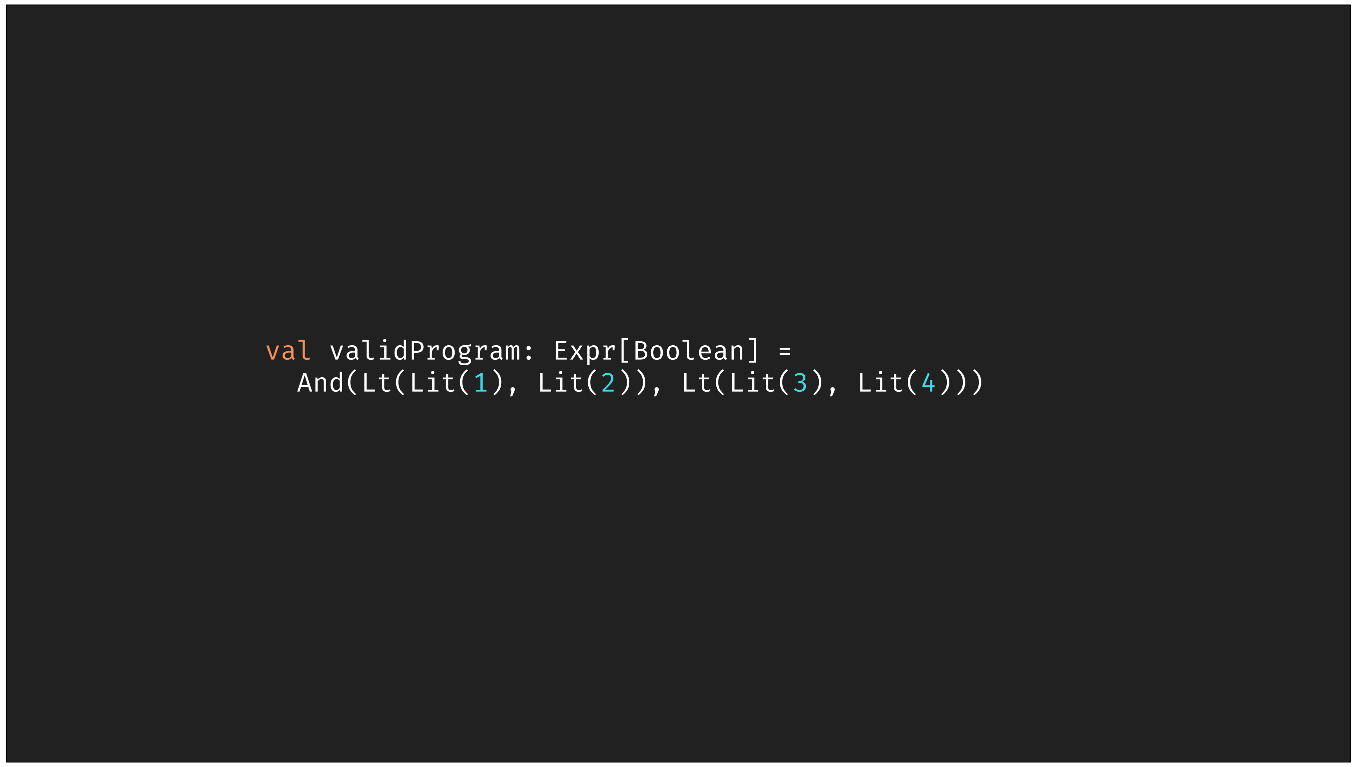
Here’s our untyped DSL. To type it we simply introduce a type parameter.



```scala
sealed trait Expr[A]

case class Lit[A](value: A) extends Expr[A]

case class Lt(a: Expr[Int], b: Expr[Int])
  extends Expr[Boolean]

case class And(a: Expr[Boolean], b: Expr[Boolean])
  extends Expr[Boolean]
```

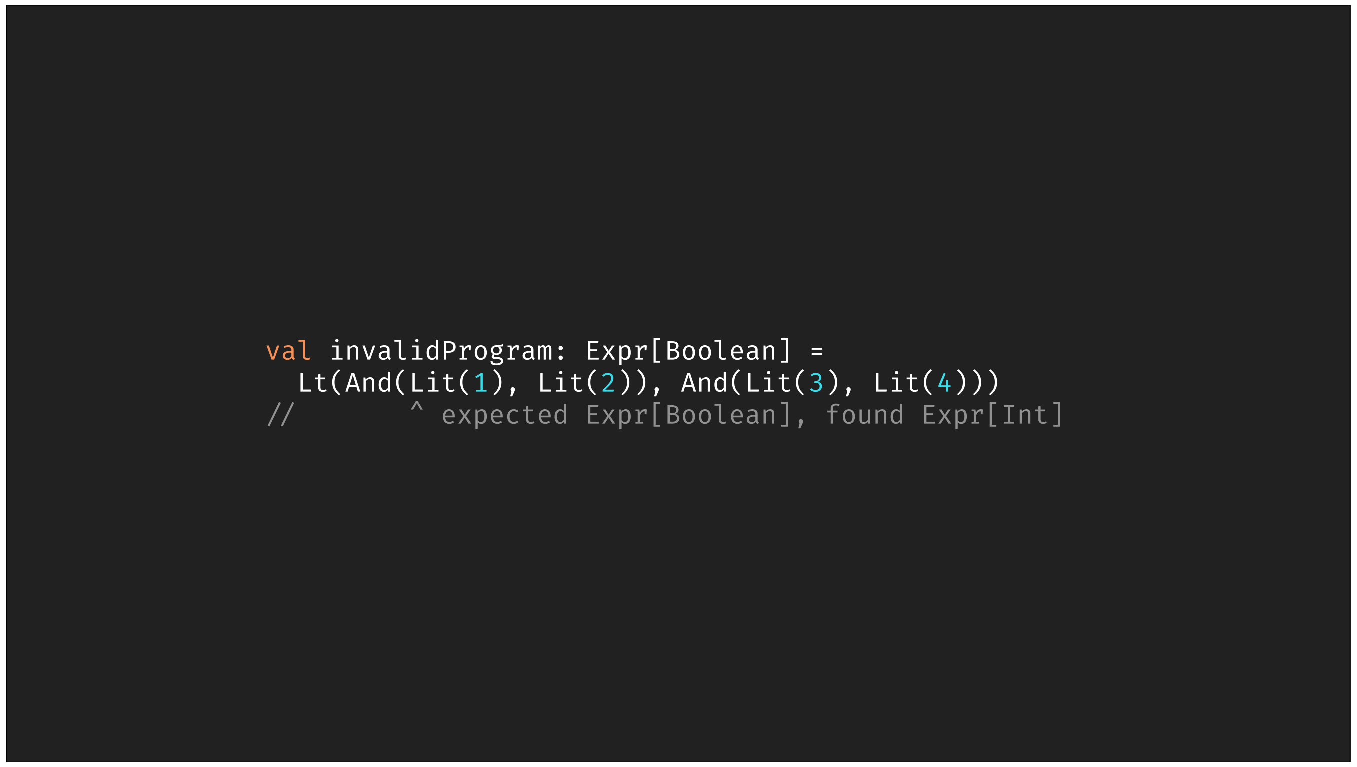
The `A` parameter here represents the type of value that an expression will compute:

- a literal can return any type;
- a less-than returns a boolean and expects operands that return integers;
- an and returns a boolean and expects operands that return booleans.



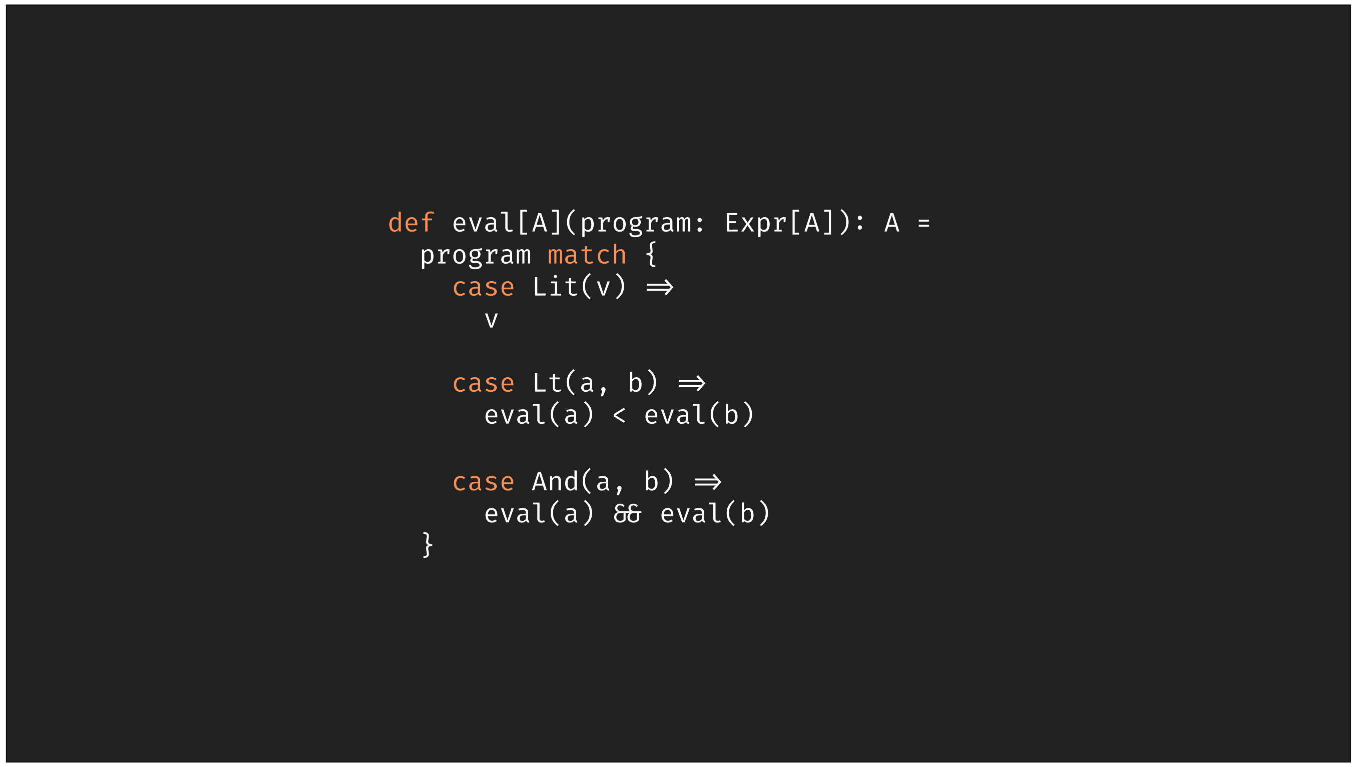
```scala
val validProgram: Expr[Boolean] =
  And(Lt(Lit(1), Lit(2)), Lt(Lit(3), Lit(4)))
```

The typed DSL allows us to represent valid programs with exactly the same syntax as the untyped DSL.



```scala
val invalidProgram: Expr[Boolean] =
  Lt(And(Lit(1), Lit(2)), And(Lit(3), Lit(4)))
//       ^ expected Expr[Boolean], found Expr[Int]
```

However, the Scala compiler is checking the result types whenever we next expressions. This makes it impossible to represent programs with type errors in them.



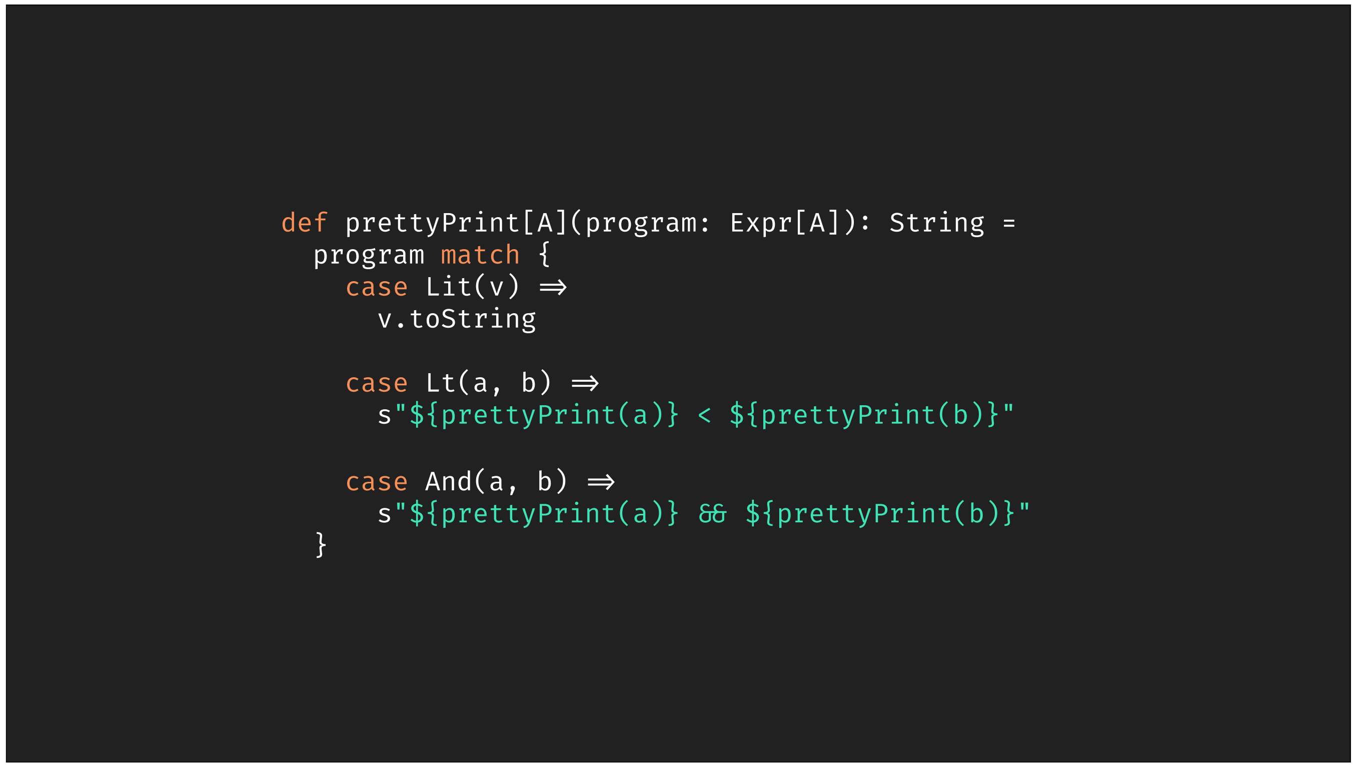
```
def eval[A](program: Expr[A]): A =
  program match {
    case Lit(v) ⇒
      v

    case Lt(a, b) ⇒
      eval(a) < eval(b)

    case And(a, b) ⇒
      eval(a) && eval(b)
  }
```

As an added bonus, because the types in our DSL map directly onto types in Scala, the implementation of our interpreter becomes trivial.

And… we still have all of the flexibility we had before…

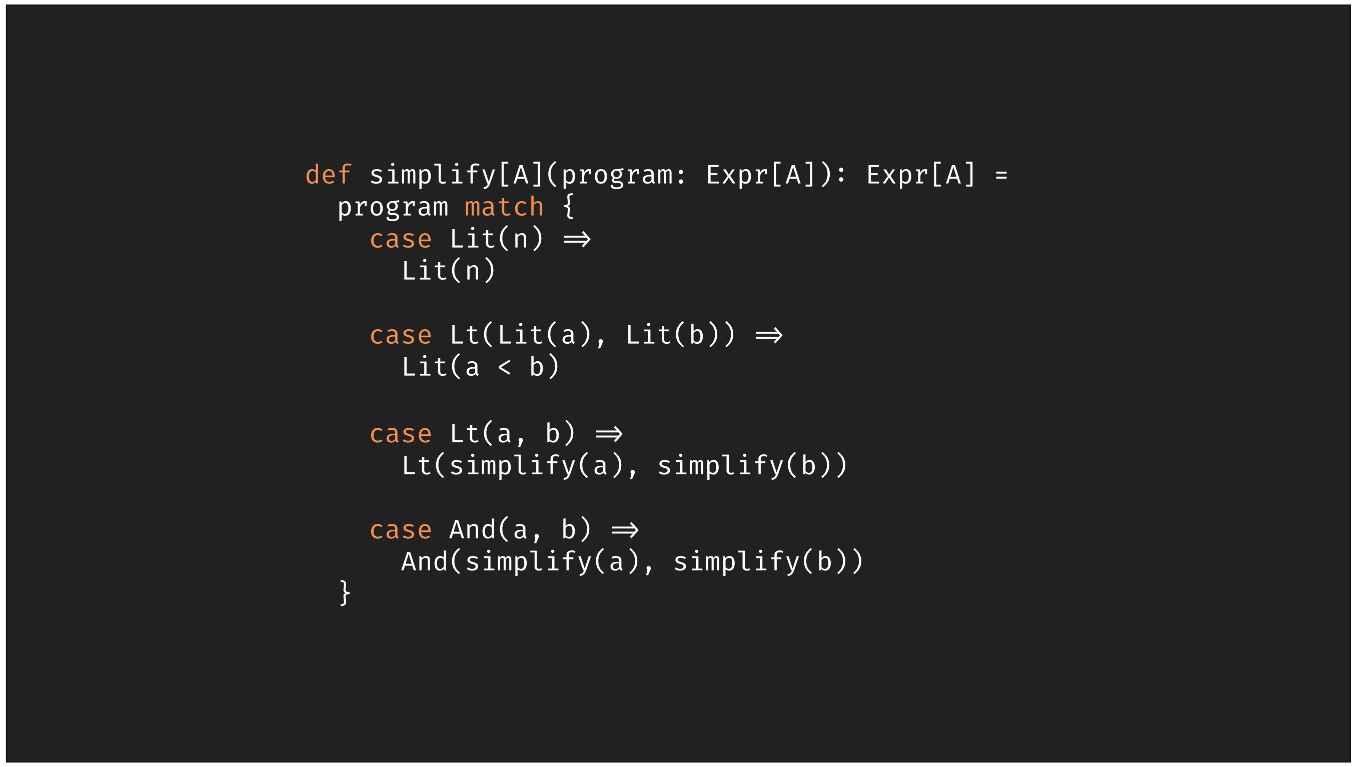
```scala
def evalAsync[A](program: Expr[A]): Future[A] =
  program match {
    case Lit(v) ⇒
      v

    case Lt(a, b) ⇒
      val x = evalAsync(a)
      val y = evalAsync(b)
      x.flatMap(a ⇒ y.map(b ⇒ a < b))

    case And(a, b) ⇒
      val x = evalAsync(a)
      val y = evalAsync(b)
      x.flatMap(a ⇒ y.map(b ⇒ a && b))
  }
```

We can easily write an asynchronous version of the interpreter.



```scala
def prettyPrint[A](program: Expr[A]): String =
  program match {
    case Lit(v) =>
      v.toString

    case Lt(a, b) =>
      s"${prettyPrint(a)} < ${prettyPrint(b)}"

    case And(a, b) =>
      s"${prettyPrint(a)} && ${prettyPrint(b)}"
  }
```

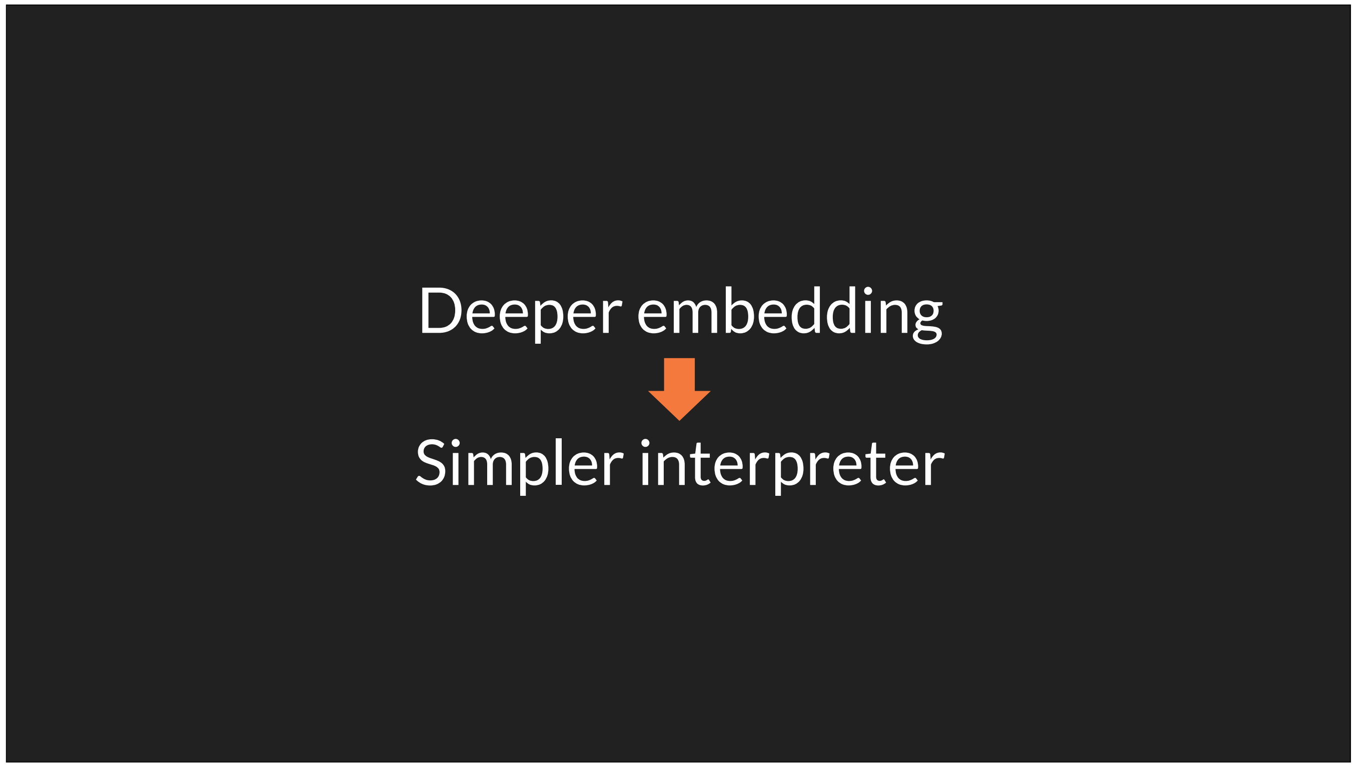
We can pretty print expressions.



```scala
def simplify[A](program: Expr[A]): Expr[A] =
  program match {
    case Lit(n) ⇒
      Lit(n)

    case Lt(Lit(a), Lit(b)) ⇒
      Lit(a < b)

    case Lt(a, b) ⇒
      Lt(simplify(a), simplify(b))

    case And(a, b) ⇒
      And(simplify(a), simplify(b))
  }
```

And we can inspect and rewrite expressions.

This interpreter pre-computes the results of “less than” expressions producing a program containing only boolean literals and logical operators.


Deeper embedding

⬇

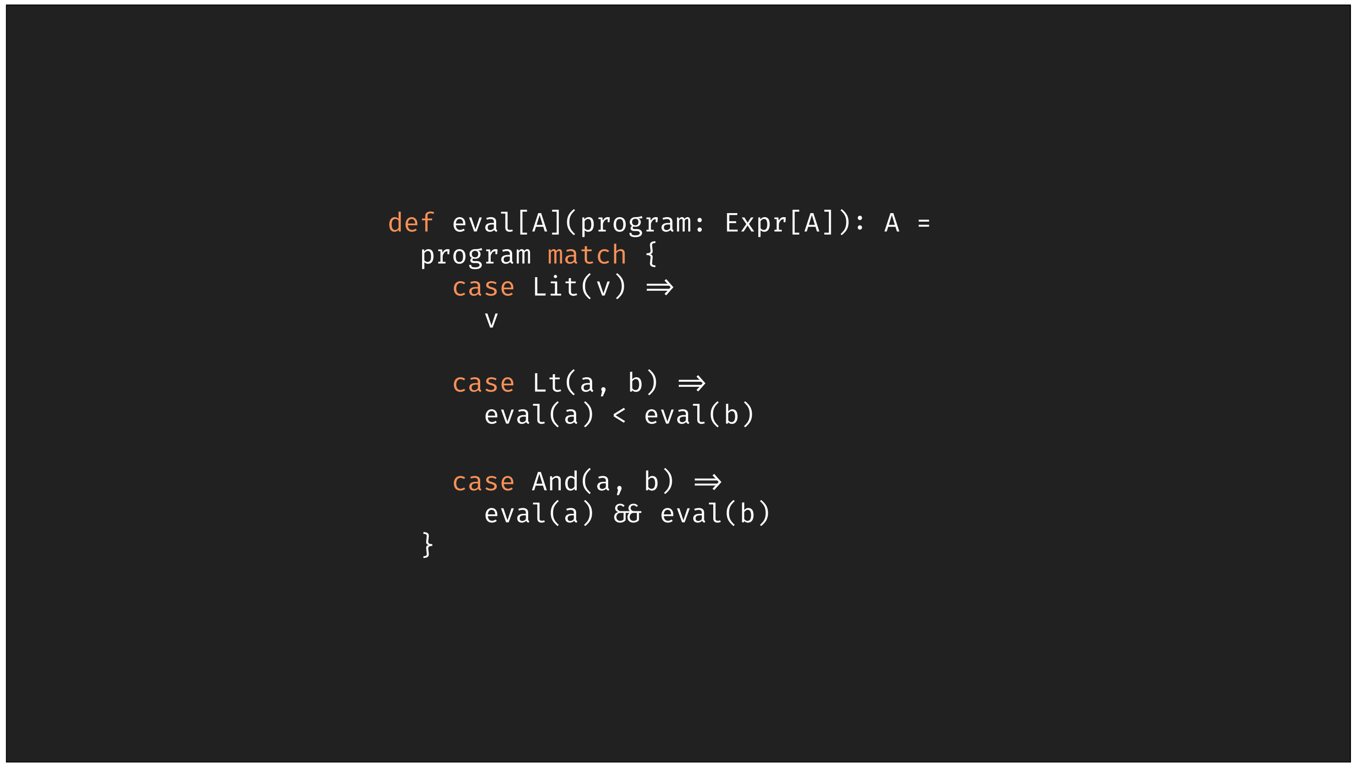
Simpler interpreter

By introducing types and aligning them with Scala's type system, we are embedding our DSL more deeply in Scala. This reduces the implementation work we have to do.

Ordering

I want to take a little detour here to talk about ordering.

You may have noticed that there’s an implicit ordering in our language and our interpreter.



```
def eval[A](program: Expr[A]): A =
  program match {
    case Lit(v) ⇒
      v

    case Lt(a, b) ⇒
      eval(a) < eval(b)

    case And(a, b) ⇒
      eval(a) && eval(b)
  }
```

Whenever we interpret a “less than” or an “and”, we implicitly evaluate the left hand side before the right.

This is a trivial detail in our demo DSL because it only contains pure expressions. However, we’re normally creating DSLs to handle some kind of state or side-effect.

Can we have explicit ordering?

In more complex problem domains, ordering matters. It makes sense to allow our users to specify ordering rather than embedding it directly into the DSL ourselves.

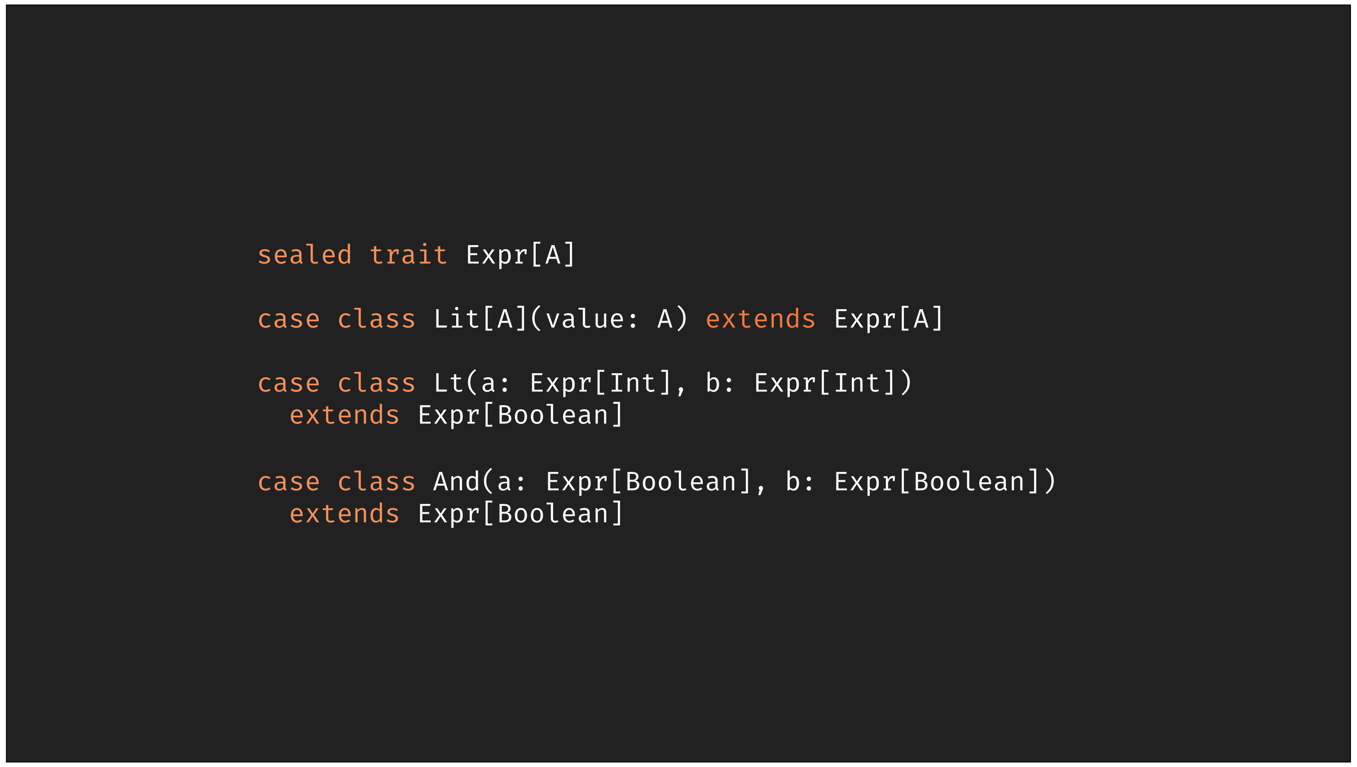
What tool do we use to represent sequencing and ordering in functional programming?

Yes, with monads!

That’s right, monads!

Monadic DSLs

Let’s make our DSL monadic.



```scala
sealed trait Expr[A]

case class Lit[A](value: A) extends Expr[A]

case class Lt(a: Expr[Int], b: Expr[Int])
  extends Expr[Boolean]

case class And(a: Expr[Boolean], b: Expr[Boolean])
  extends Expr[Boolean]
```

Here’s our typed DSL from the last example.

We’re going to add a new type of expression to take care of ordering.

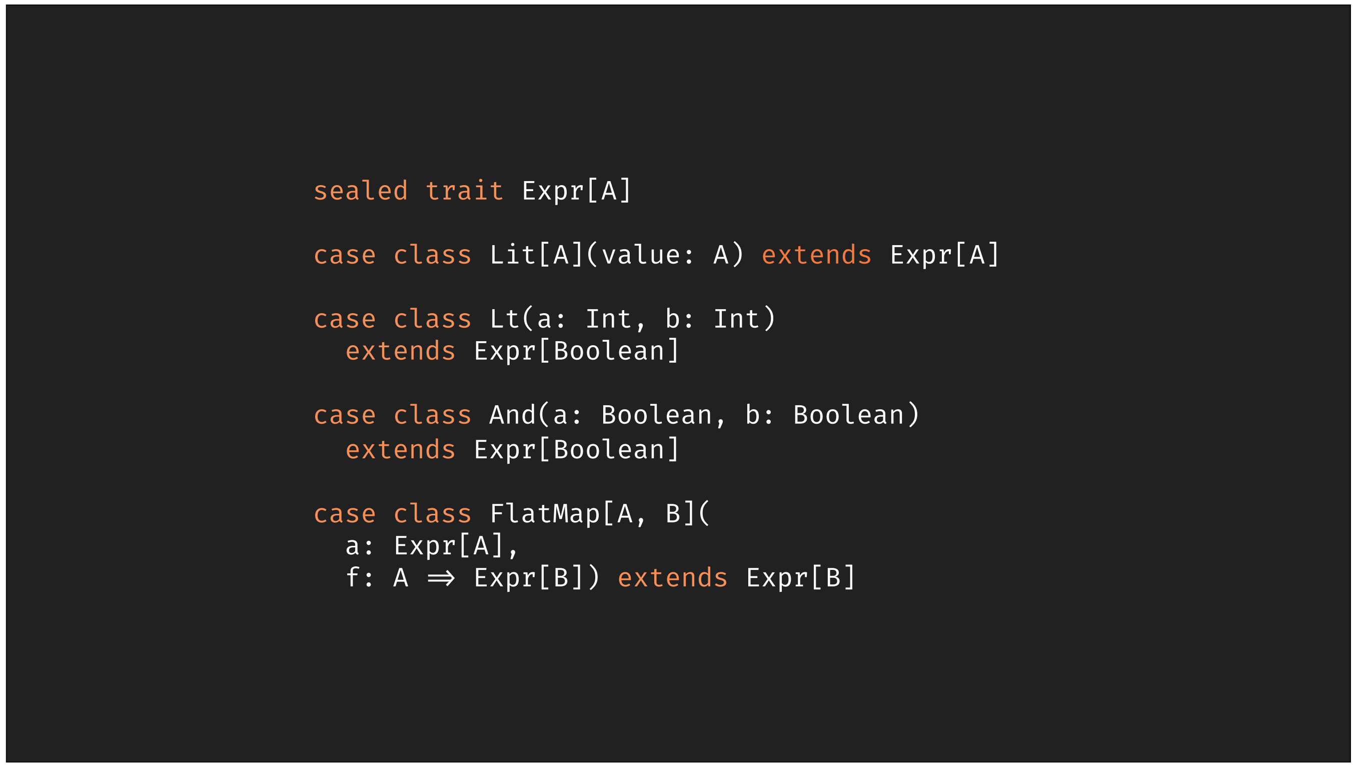
```scala
sealed trait Expr[A]

case class Lit[A](value: A) extends Expr[A]

case class Lt(a: Expr[Int], b: Expr[Int])
  extends Expr[Boolean]

case class And(a: Expr[Boolean], b: Expr[Boolean])
  extends Expr[Boolean]

case class FlatMap[A, B](
  a: Expr[A],
  f: A => Expr[B]) extends Expr[B]
```

Here it is. We've called it `FlatMap` and given it similar semantics to the `flatMap()` method on Option, List, Future, and so on:

- it contains an `Expr[A]` representing some initial step in our program;
- it contains a function `f` that calculates a second step given the output of the first step.

Interpreting a `FlatMap` involves interpreting the first step, feeding its result to the function to obtain the second step, then interpreting the second step to get our overall result of type `B`.

We can use nested `FlatMaps` to construct arbitrarily long sequences of steps of the other types of `Expr`.

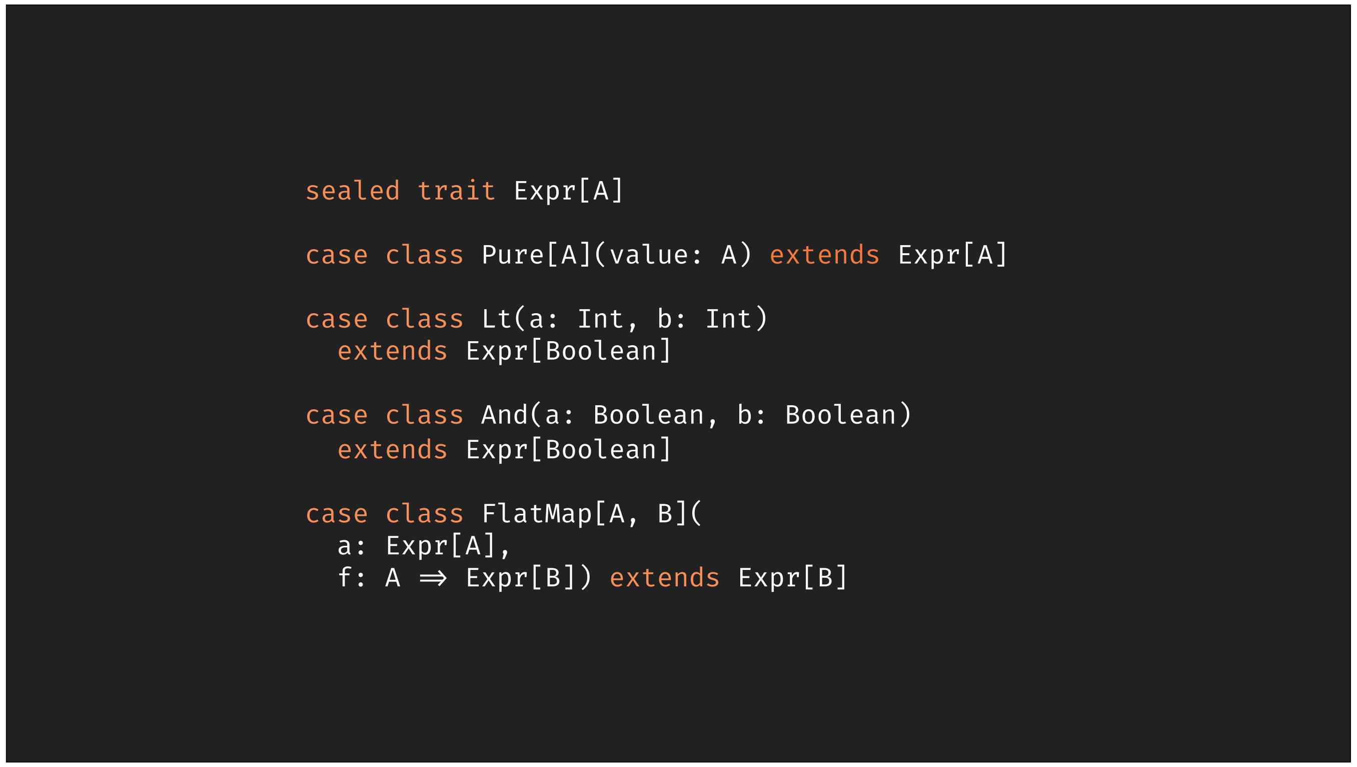


```scala
sealed trait Expr[A]

case class Lit[A](value: A) extends Expr[A]

case class Lt(a: Int, b: Int)
  extends Expr[Boolean]

case class And(a: Boolean, b: Boolean)
  extends Expr[Boolean]

case class FlatMap[A, B](
  a: Expr[A],
  f: A ⇒ Expr[B]) extends Expr[B]
```

Interestingly, now we have `FlatMap` to do our sequencing, we no longer need the arguments to `Lt` and `And` to be instances of `Expr`.

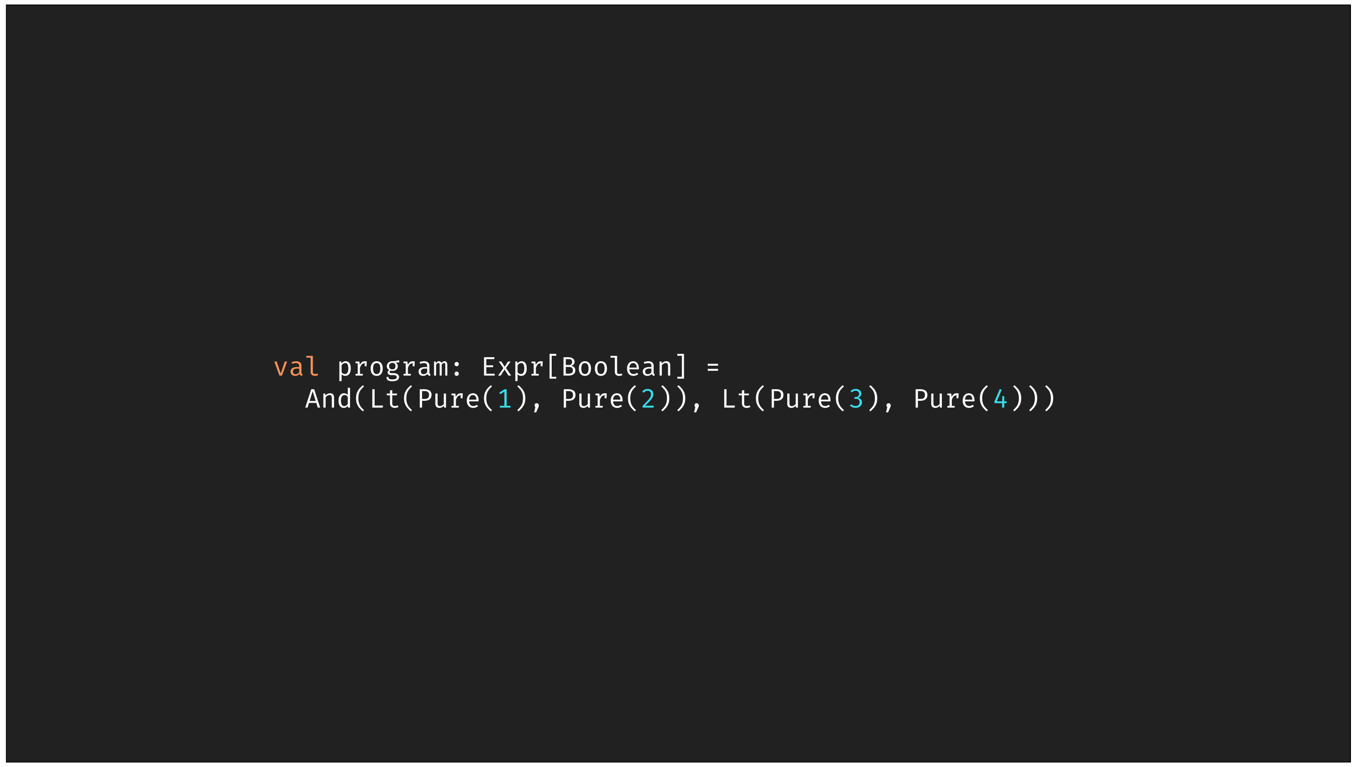
We previously represented these operands as expressions to allow us to compose programs. We can now use `FlatMap` to do the composition instead.

So now, `Lit`, `Lt`, and `And` represent atomic steps in our programs, and `FlatMap` sequences these steps.



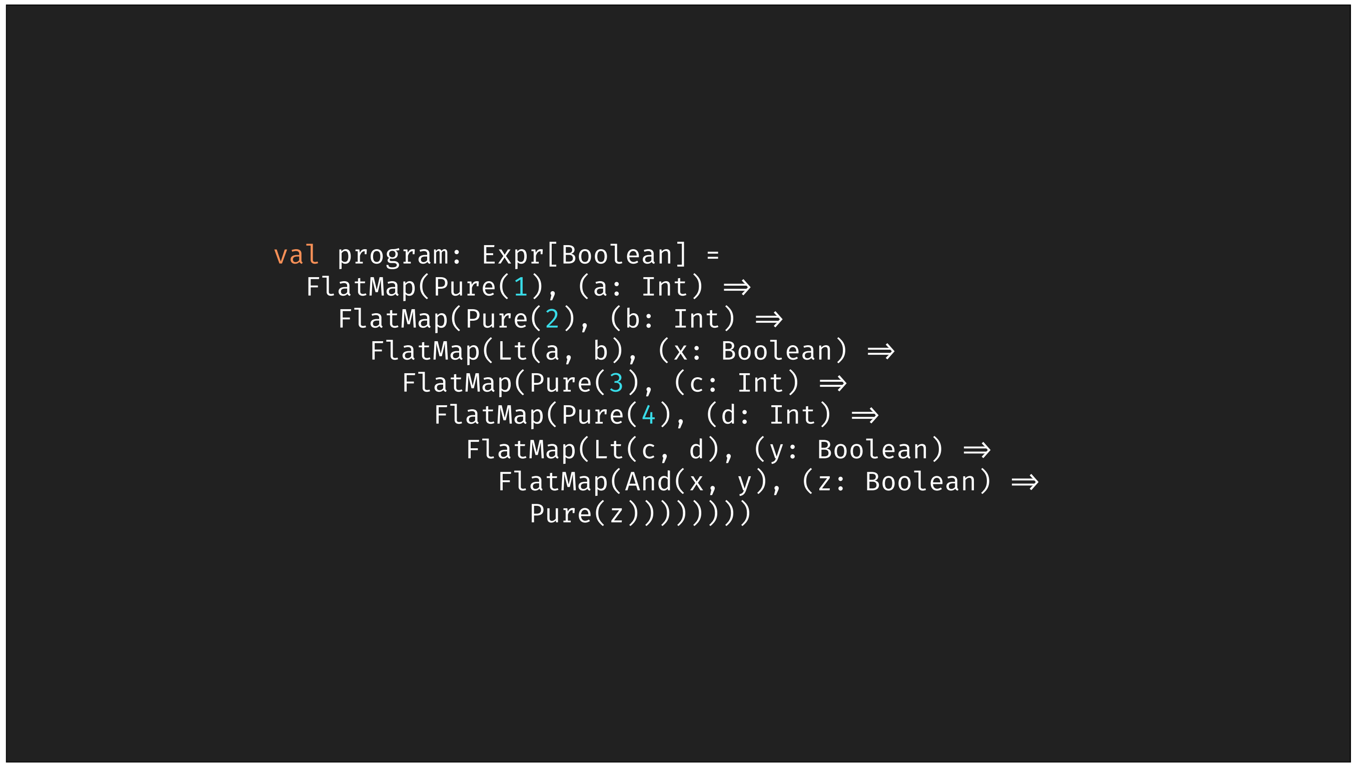
```scala
sealed trait Expr[A]

case class Pure[A](value: A) extends Expr[A]

case class Lt(a: Int, b: Int)
  extends Expr[Boolean]

case class And(a: Boolean, b: Boolean)
  extends Expr[Boolean]

case class FlatMap[A, B](
  a: Expr[A],
  f: A ⇒ Expr[B]) extends Expr[B]
```

We'll make one other tweak here: our "literal" expression just wraps a value in an expression. This is exactly equivalent to the "pure" operation in Cats (or "point" operation in Scalaz), so we'll rename it to "Pure" here for consistency.


```scala
val program: Expr[Boolean] =
  And(Lt(Pure(1), Pure(2)), Lt(Pure(3), Pure(4)))
```

Now we've changed the structure of our DSL, we can no longer directly nest subexpressions. We have to do it with `FlatMap` instead. So we can't write programs like this. Instead we have to…

…wait for it…

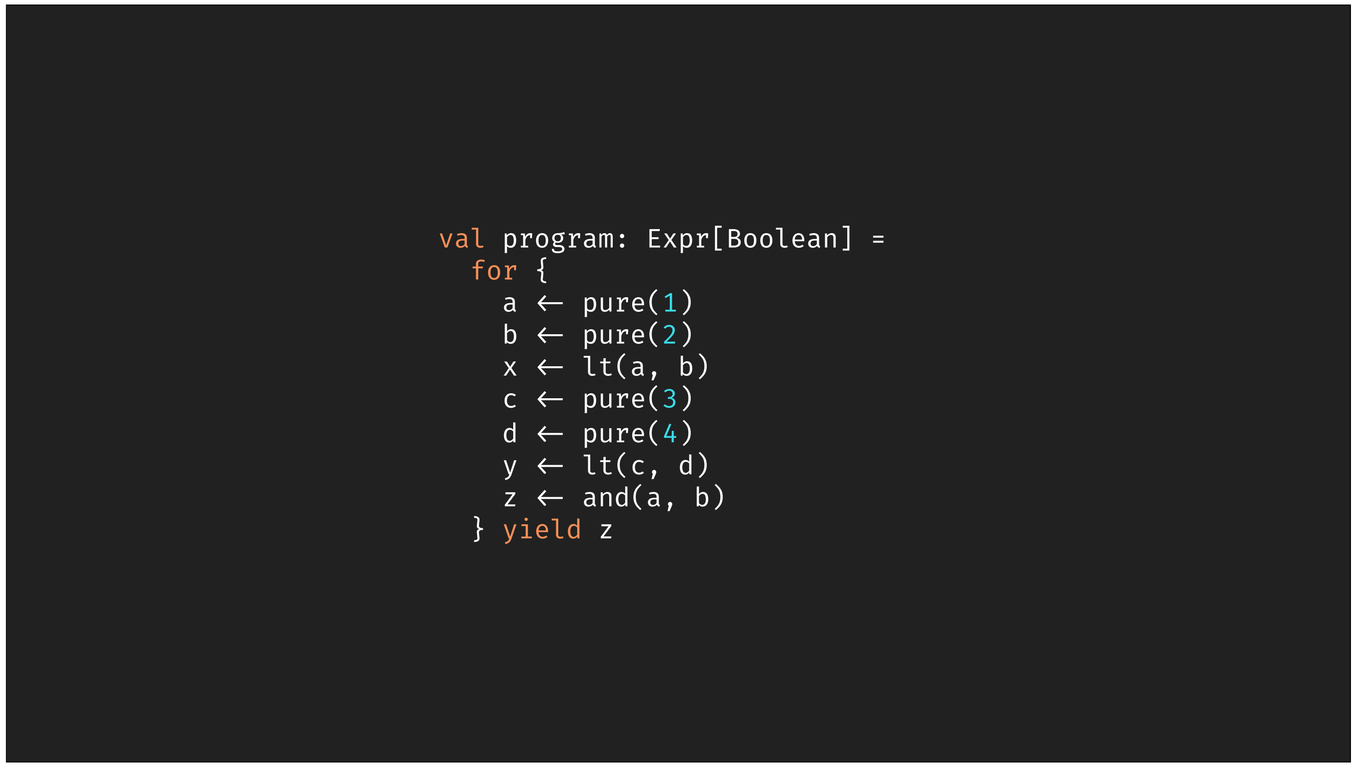


```
val program: Expr[Boolean] =
  FlatMap(Pure(1), (a: Int) ⇒
    FlatMap(Pure(2), (b: Int) ⇒
      FlatMap(Lt(a, b), (x: Boolean) ⇒
        FlatMap(Pure(3), (c: Int) ⇒
          FlatMap(Pure(4), (d: Int) ⇒
            FlatMap(Lt(c, d), (y: Boolean) ⇒
              FlatMap(And(x, y), (z: Boolean) ⇒
                Pure(z))))))))
```

…write them like this.

But look at the structure — it’s still the same structure we had before:

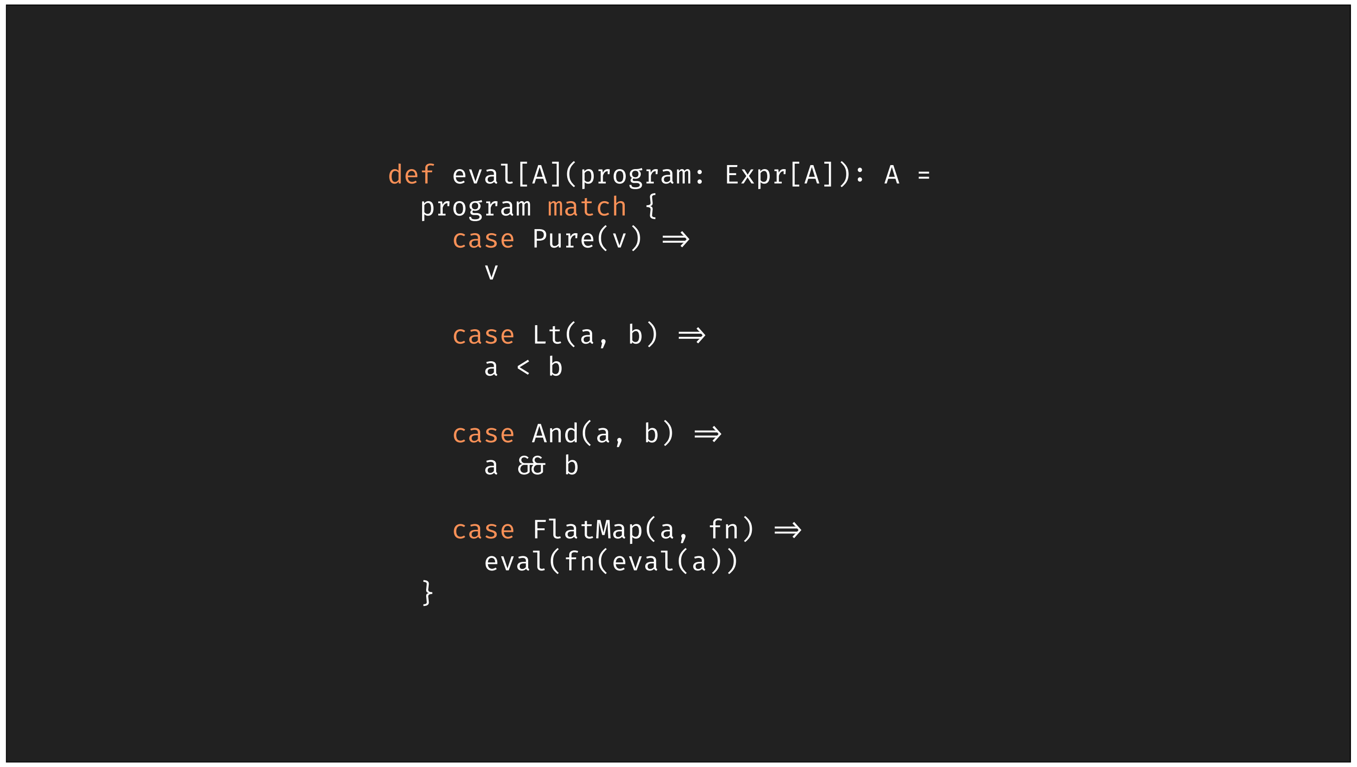
- step 1 calculates the literal 1 and passes it along as the variable `a`;
- step 2 calculates the literal 2 and passes it along as the variable `b`;
- step 3 compares the values `a` and `b` and passes the result as `x`;
- and so on…



```scala
val program: Expr[Boolean] =
  for {
    a ← pure(1)
    b ← pure(2)
    x ← lt(a, b)
    c ← pure(3)
    d ← pure(4)
    y ← lt(c, d)
    z ← and(a, b)
  } yield z
```

If we sprinkle a few methods here and there (including adding `flatMap()` and `map()` methods to `Expr`), we can represent this sequence with a regular `for`-comprehension.

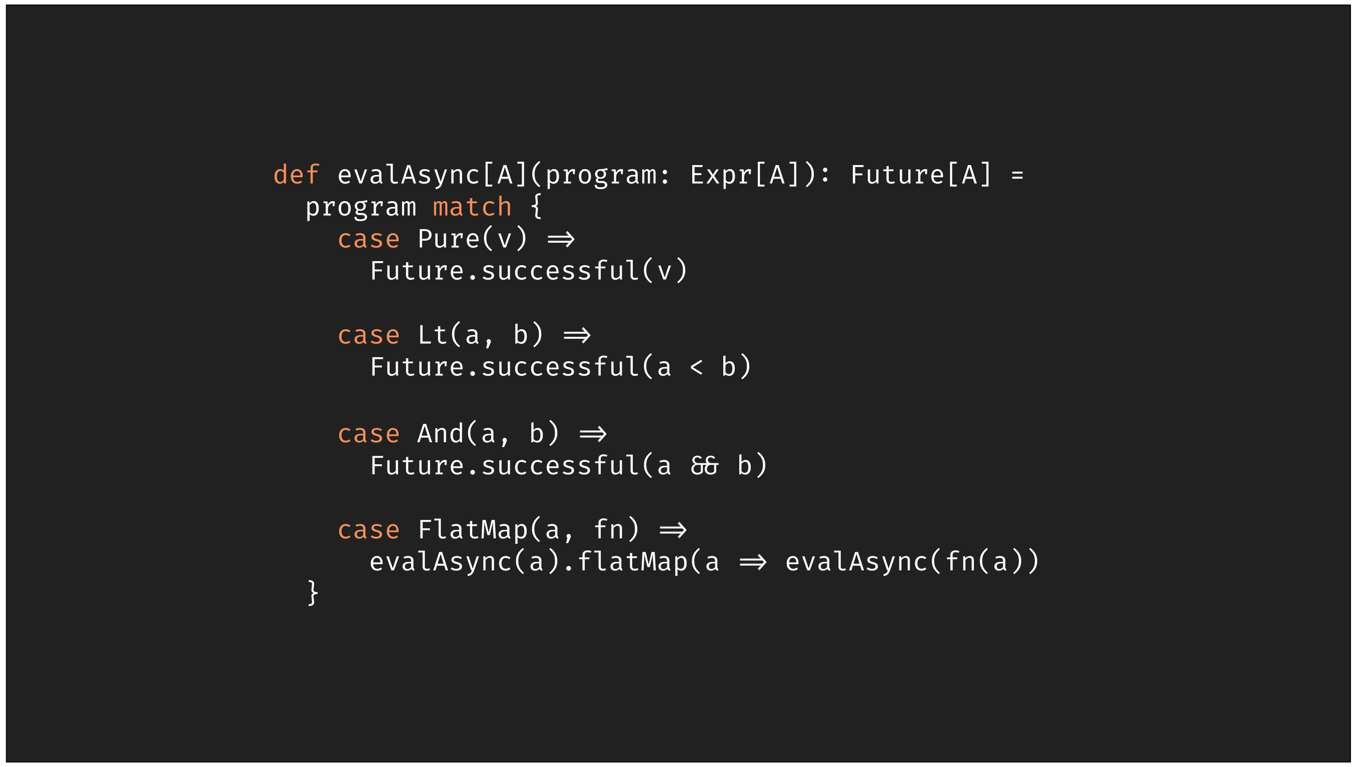
This gives us precise control over the ordering of statements. We can shuffle them arbitrarily within the sequence as long as we never use the value of a variable before we introduce it.



```scala
def eval[A](program: Expr[A]): A =
  program match {
    case Pure(v) ⇒
      v

    case Lt(a, b) ⇒
      a < b

    case And(a, b) ⇒
      a && b

    case FlatMap(a, fn) ⇒
      eval(fn(eval(a)))
  }
```

The interpreter is simple too. The implementations of `Pure`, `Lt`, and `And` are trivial, and the implementation of `FlatMap` does what we expect:

- it interprets the first step `a`;
- it passes the result to `fn` to calculate the second step;
- it interprets the second step and returns the result.

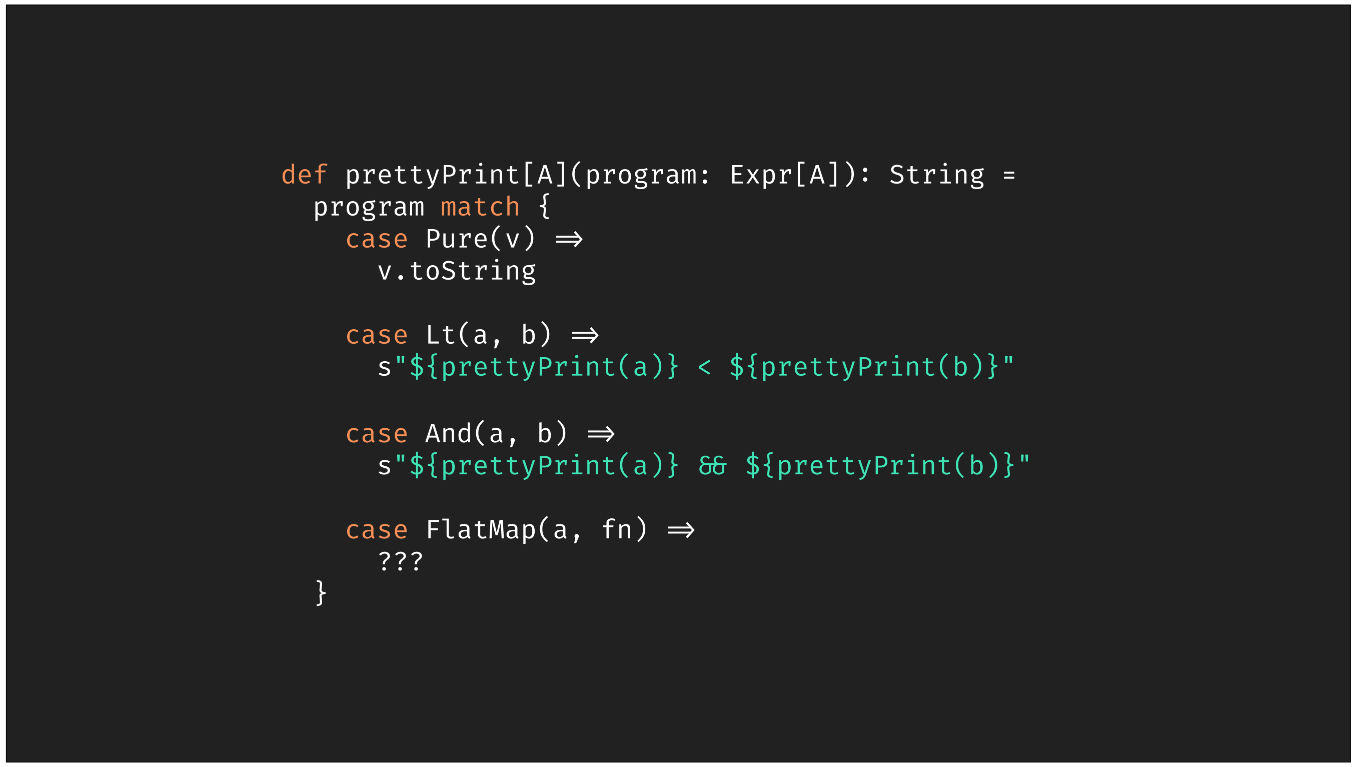


```
def evalAsync[A](program: Expr[A]): Future[A] =
  program match {
    case Pure(v) ⇒
      Future.successful(v)

    case Lt(a, b) ⇒
      Future.successful(a < b)

    case And(a, b) ⇒
      Future.successful(a && b)

    case FlatMap(a, fn) ⇒
      evalAsync(a).flatMap(a ⇒ evalAsync(fn(a))
  }
```

It is still trivial to write interpreters to handle different effects.

As an side, note that we can no longer execute branches in parallel because we've only encoded sequencing using `FlatMap`.

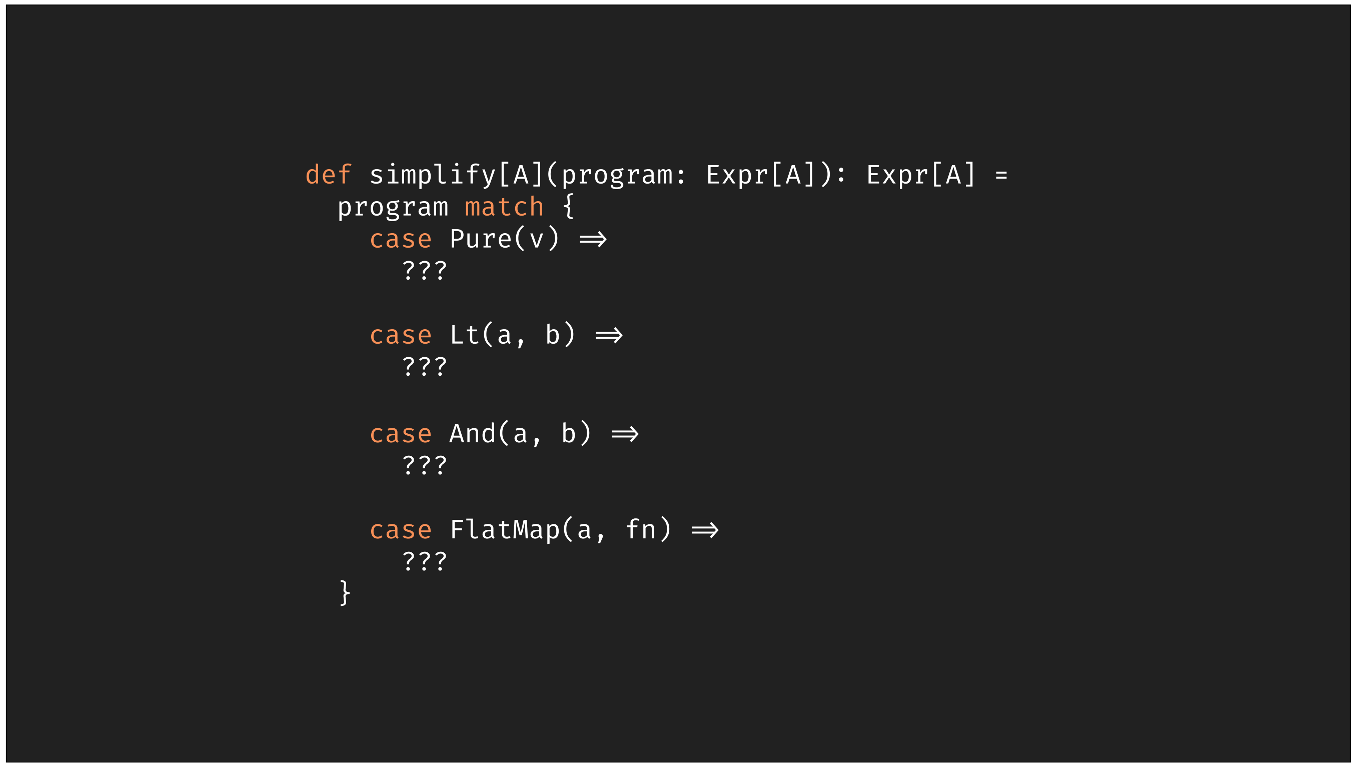
This is a simple language design problem: if introduce an explicit `Parallel` step or applicative functor equivalent, we'll be able to execute in parallel once again.



```
def prettyPrint[A](program: Expr[A]): String =
  program match {
    case Pure(v) ⇒
      v.toString

    case Lt(a, b) ⇒
      s"${prettyPrint(a)} < ${prettyPrint(b)}"

    case And(a, b) ⇒
      s"${prettyPrint(a)} && ${prettyPrint(b)}"

    case FlatMap(a, fn) ⇒
      ???
  }
```

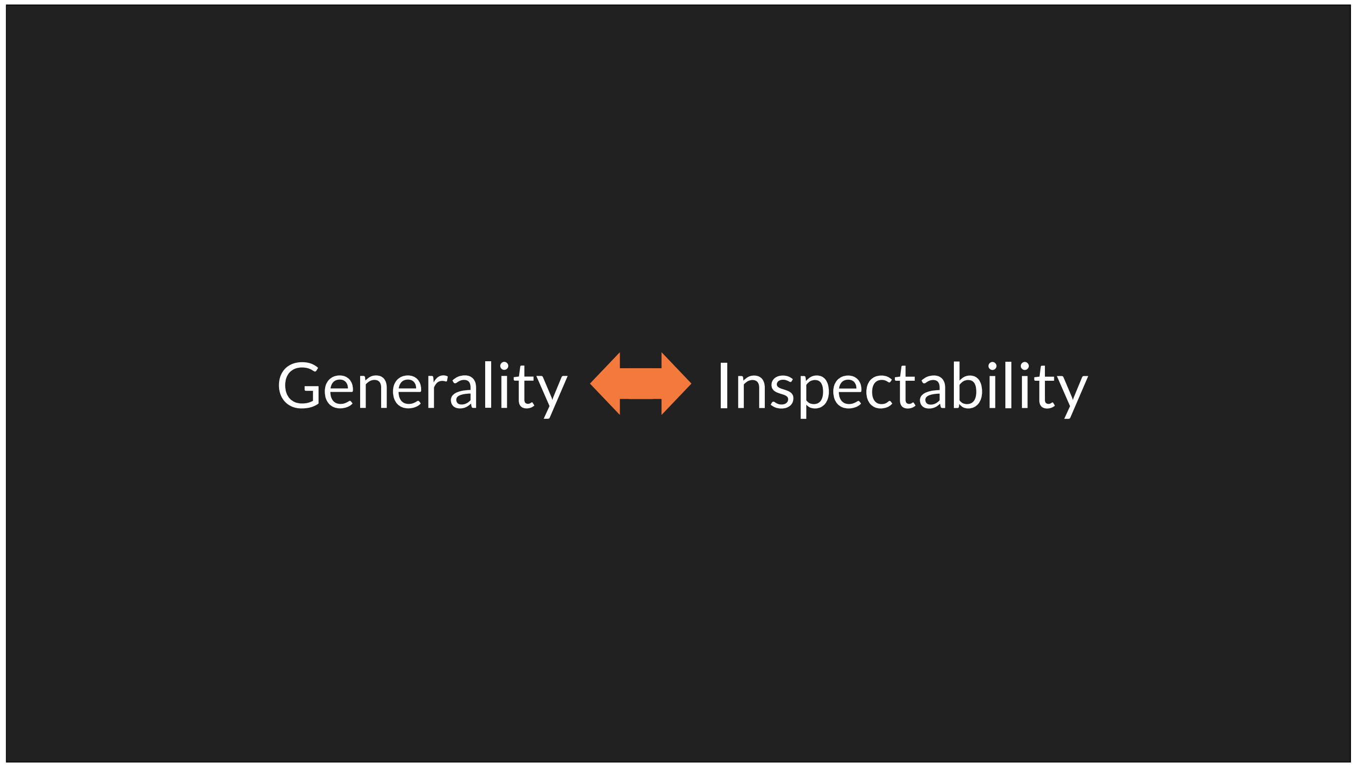
One big negative consequence of `FlatMap` is that we lose the ability to perform certain types of inspection.

For example, it's no longer possible to pretty print our code. The reason is we can't reason about the behaviour of the code inside `fn`. The function could introduce any kind of side-effect, so we can't guarantee that the sequence of steps will always be the same on every run through the program.



```scala
def simplify[A](program: Expr[A]): Expr[A] =
  program match {
    case Pure(v) ⇒
      ???

    case Lt(a, b) ⇒
      ???

    case And(a, b) ⇒
      ???

    case FlatMap(a, fn) ⇒
      ???
  }
```

Compilers like `simplify()` are similarly difficult to implement. The generality of Scala functions impedes our ability to reliably rewrite programs in different orders.


Generality ⟷ Inspectability

The big take-home here is that introducing a monadic step has two effects:

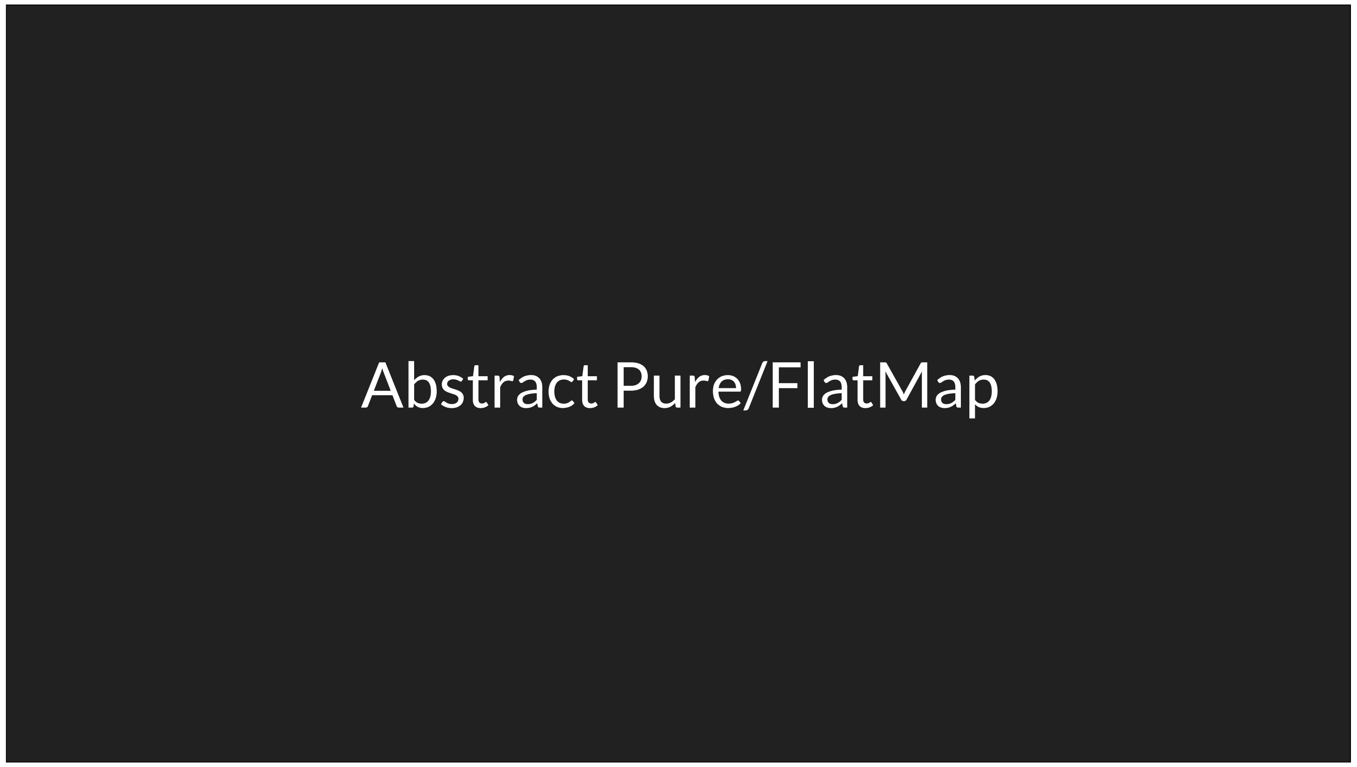
- it makes our DSL much more “powerful” or “general”… i.e. the DSL can represent many more kinds of program;
- with increased generality comes restricted inspectability… we lose the ability to reason about programs in certain kinds of ways.

Do we have to
write FlatMap ourselves?

Now I want to shift tacks again. We’ve introduced this new `FlatMap` step, which seems like a pretty general functional programming tool. We might sensibly ask whether we need to implement this ourselves, or whether it already exists in our functional programming library of choice.

Free monadic DSLs

And, of course, it turns out that it does. This leads us directly on to the “free monad”.

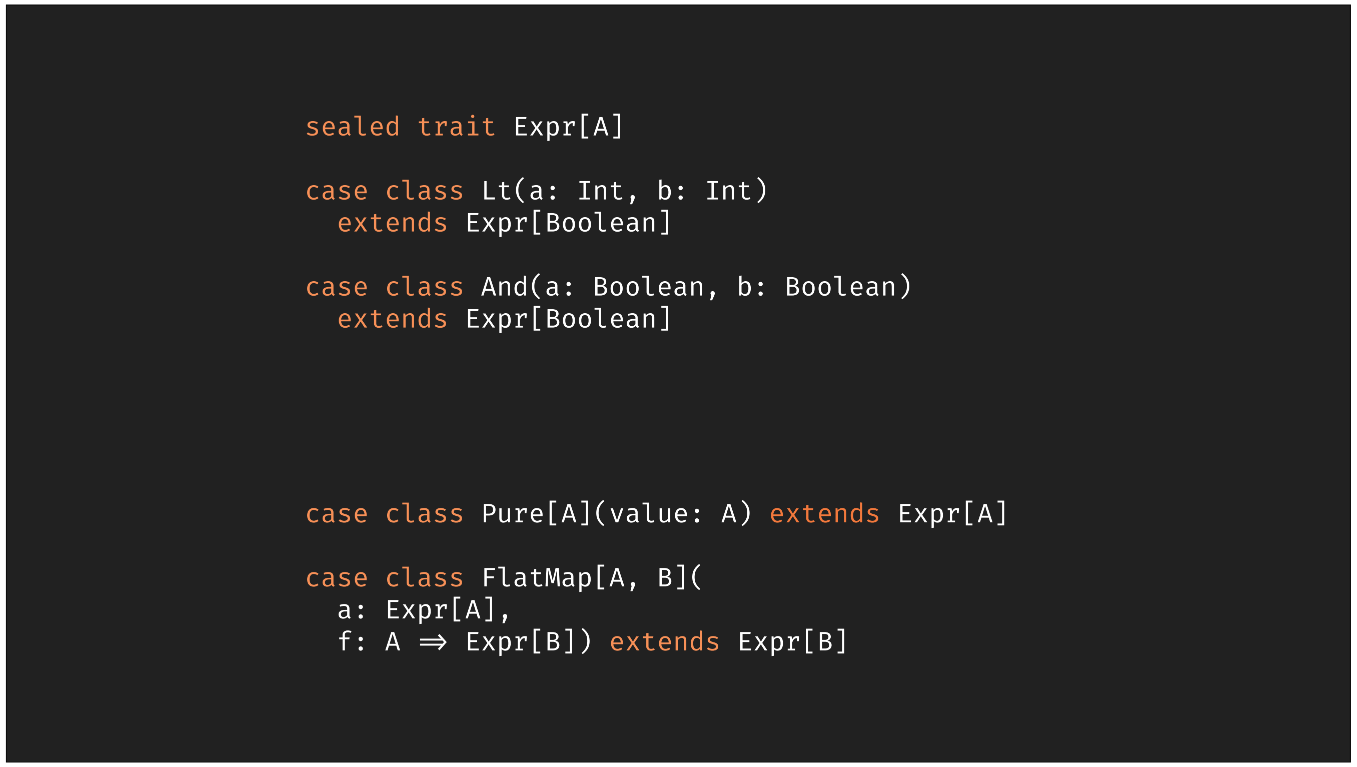


Abstract Pure/FlatMap

The idea of `Free` is that it provides generalised FlatMap and Pure constructs. We can inject our own steps into these constructs, making it easier to implement and compose DSLs.

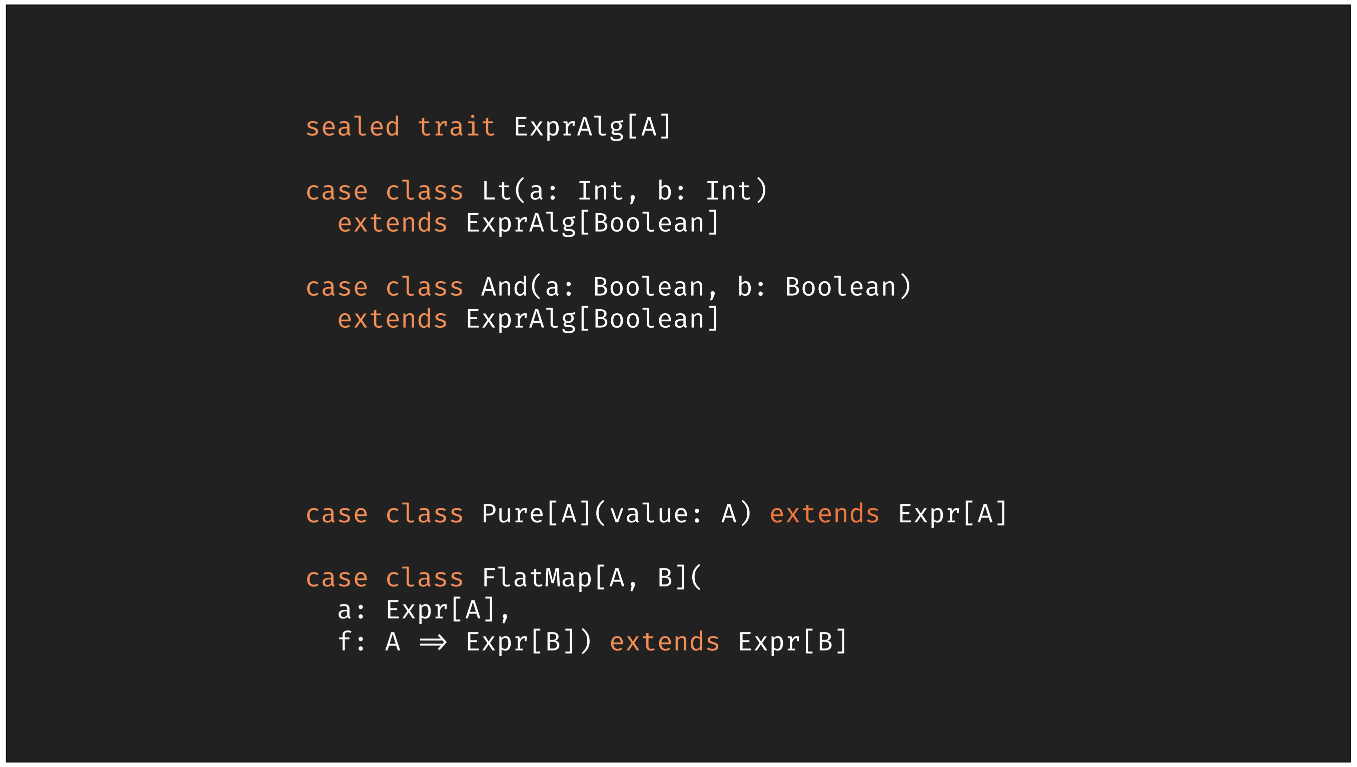
```scala
sealed trait Expr[A]

case class Pure[A](value: A) extends Expr[A]

case class Lt(a: Int, b: Int)
  extends Expr[Boolean]

case class And(a: Boolean, b: Boolean)
  extends Expr[Boolean]

case class FlatMap[A, B](
  a: Expr[A],
  f: A => Expr[B]) extends Expr[B]
```

Let’s rewrite our code to derive the free monad.



```scala
sealed trait Expr[A]

case class Lt(a: Int, b: Int)
  extends Expr[Boolean]

case class And(a: Boolean, b: Boolean)
  extends Expr[Boolean]




case class Pure[A](value: A) extends Expr[A]

case class FlatMap[A, B](
  a: Expr[A],
  f: A => Expr[B]) extends Expr[B]
```

We'll start by separating our DSL into two: the individual steps, and the `Pure` and `FlatMap` that will become part of the monad.



```scala
sealed trait ExprAlg[A]

case class Lt(a: Int, b: Int)
  extends ExprAlg[Boolean]

case class And(a: Boolean, b: Boolean)
  extends ExprAlg[Boolean]




case class Pure[A](value: A) extends Expr[A]

case class FlatMap[A, B](
  a: Expr[A],
  f: A ⇒ Expr[B]) extends Expr[B]
```

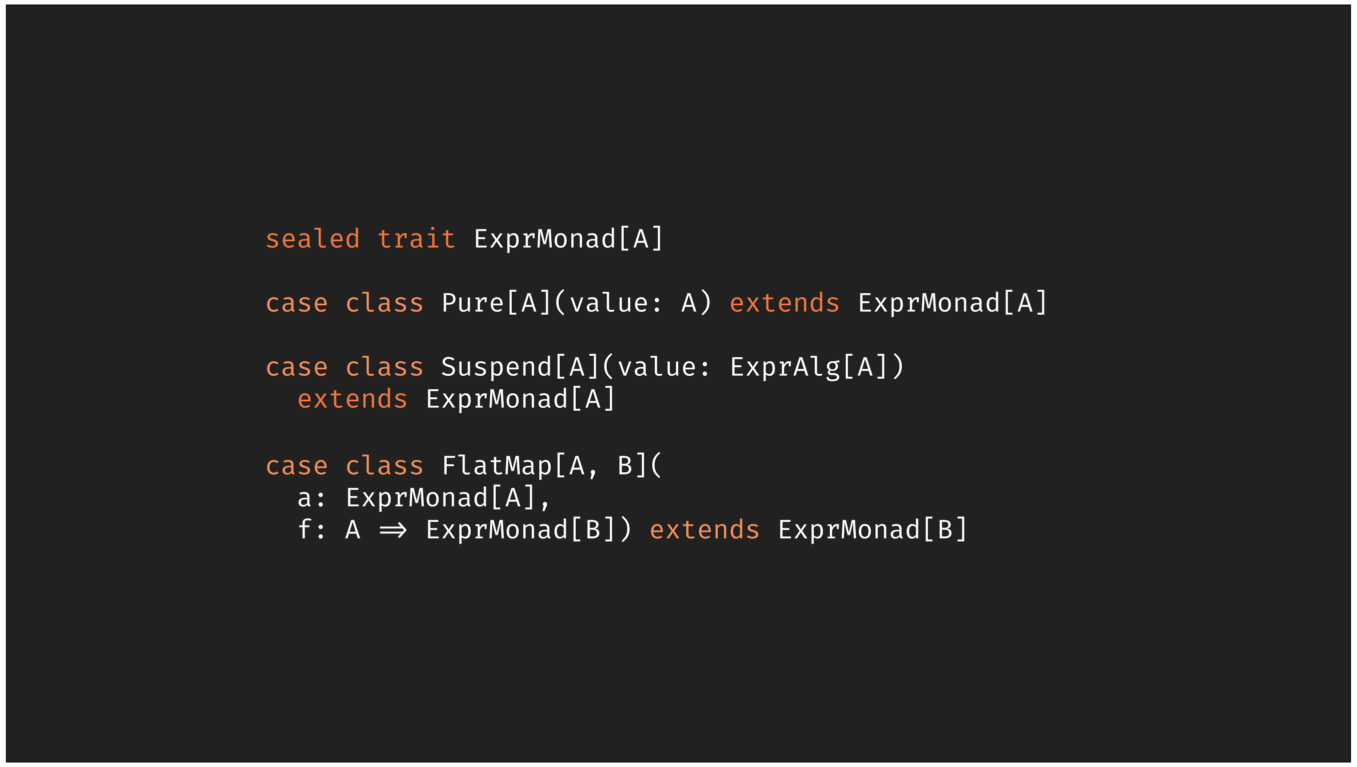
We'll rename the steps `ExprAlg`. The common terminology for working with the Free monad refers to the steps as an "algebra". `Lt` and `And` are parts of this algebra.

```scala
sealed trait ExprAlg[A]

case class Lt(a: Int, b: Int)
  extends ExprAlg[Boolean]

case class And(a: Boolean, b: Boolean)
  extends ExprAlg[Boolean]


sealed trait ExprMonad[A]

case class Pure[A](value: A) extends ExprMonad[A]

case class FlatMap[A, B](
  a: ExprMonad[A],
  f: A ⇒ ExprMonad[B]) extends ExprMonad[B]
```

We’ll name the monadic part `ExprMonad`.

```scala
sealed trait ExprAlg[A]

case class Lt(a: Int, b: Int)
  extends ExprAlg[Boolean]

case class And(a: Boolean, b: Boolean)
  extends ExprAlg[Boolean]



sealed trait ExprMonad[A]

case class Pure[A](value: A) extends ExprMonad[A]

case class Suspend[A](value: ExprAlg[A])
  extends ExprMonad[A]

case class FlatMap[A, B](
  a: ExprMonad[A],
  f: A ⇒ ExprMonad[B]) extends ExprMonad[B]
```

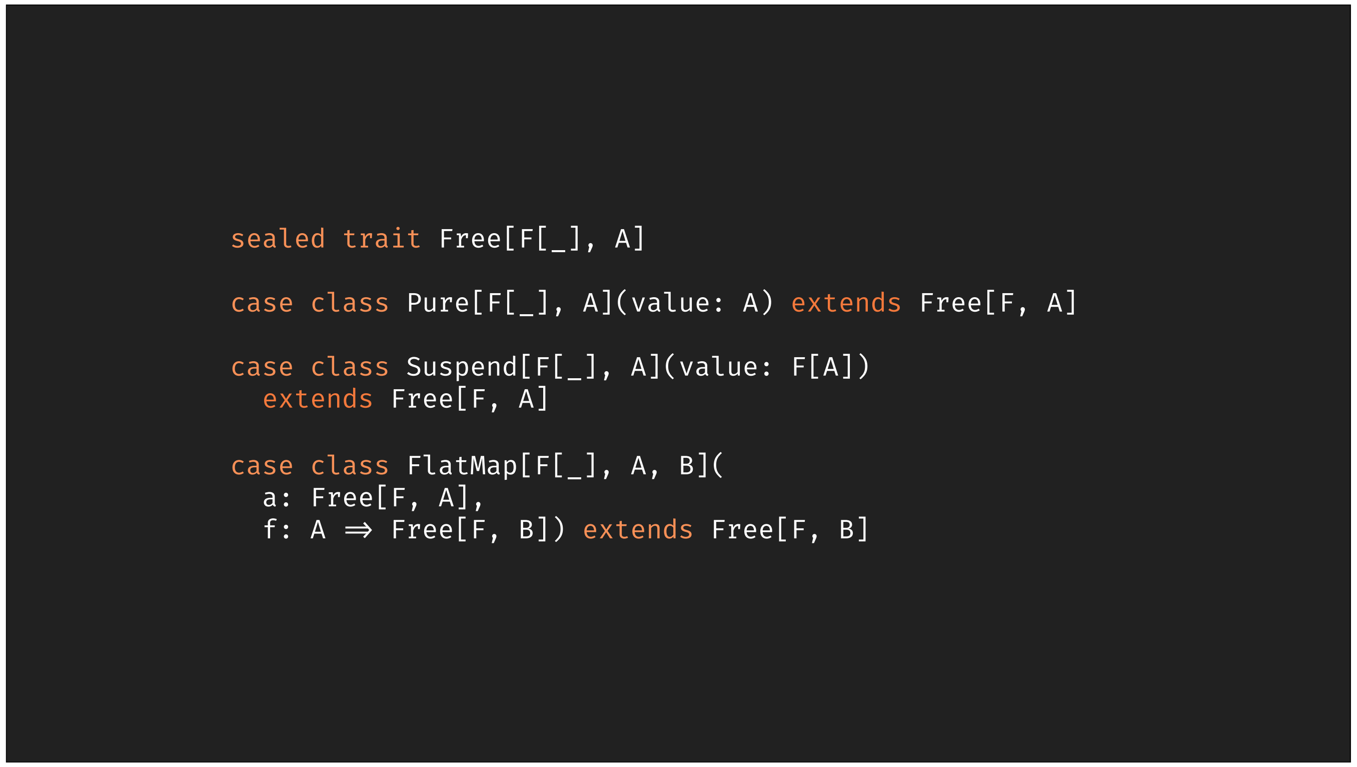
We need a new type of `ExprMonad` to wrap up an `ExprAlg`. In Cats this is called `Suspend`, so we'll introduce this too.



```scala
sealed trait ExprMonad[A]

case class Pure[A](value: A) extends ExprMonad[A]

case class Suspend[A](value: ExprAlg[A])
  extends ExprMonad[A]

case class FlatMap[A, B](
  a: ExprMonad[A],
  f: A ⇒ ExprMonad[B]) extends ExprMonad[B]
```

In this encoding, programs are instances of `ExprMonad` that wrap up sequences of values and `ExprAlg` steps via `Pure` and `Suspend`.

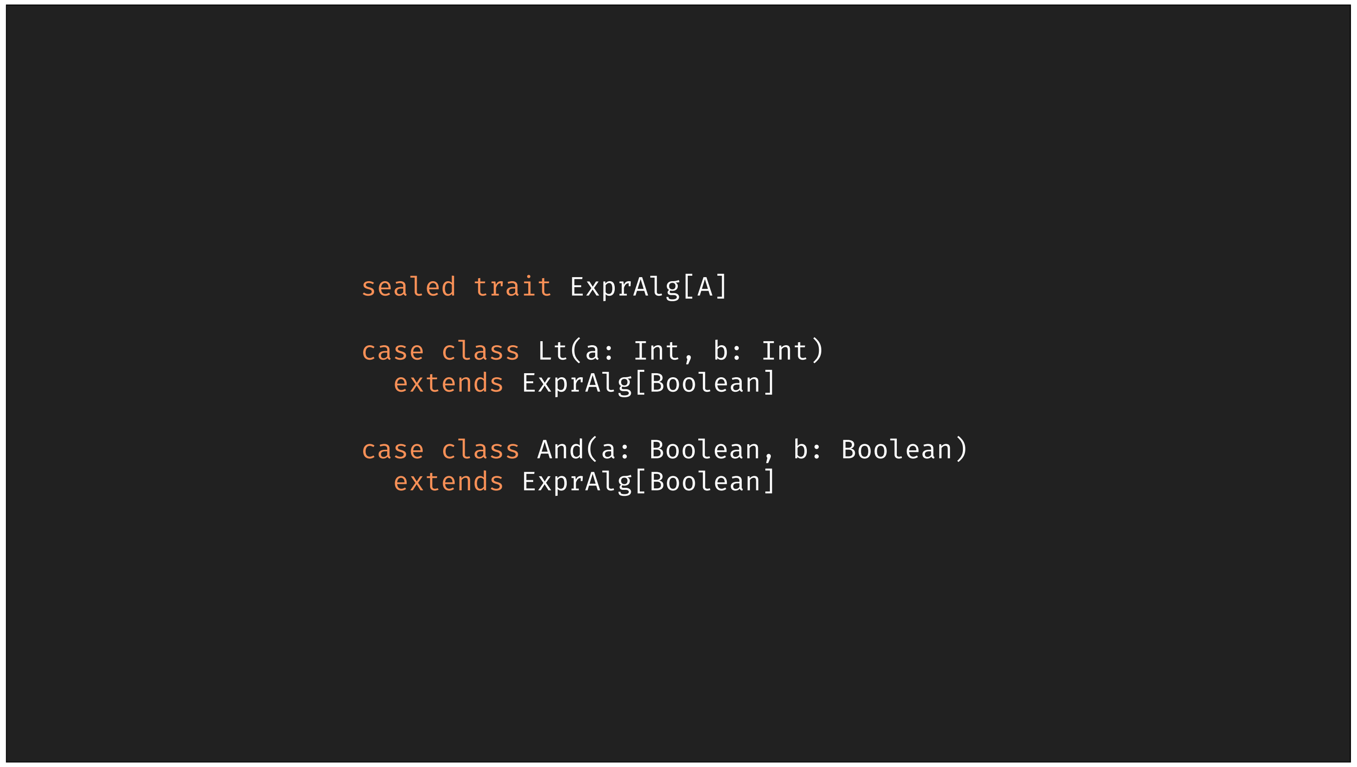
This is hard-coded to `ExprAlg` but it's pretty simple to generalise it to any algebra with any set of steps in it.



```scala
sealed trait Free[F[_], A]

case class Pure[F[_], A](value: A) extends Free[F, A]

case class Suspend[F[_], A](value: F[A])
  extends Free[F, A]

case class FlatMap[F[_], A, B](
  a: Free[F, A],
  f: A ⇒ Free[F, B]) extends Free[F, B]
```

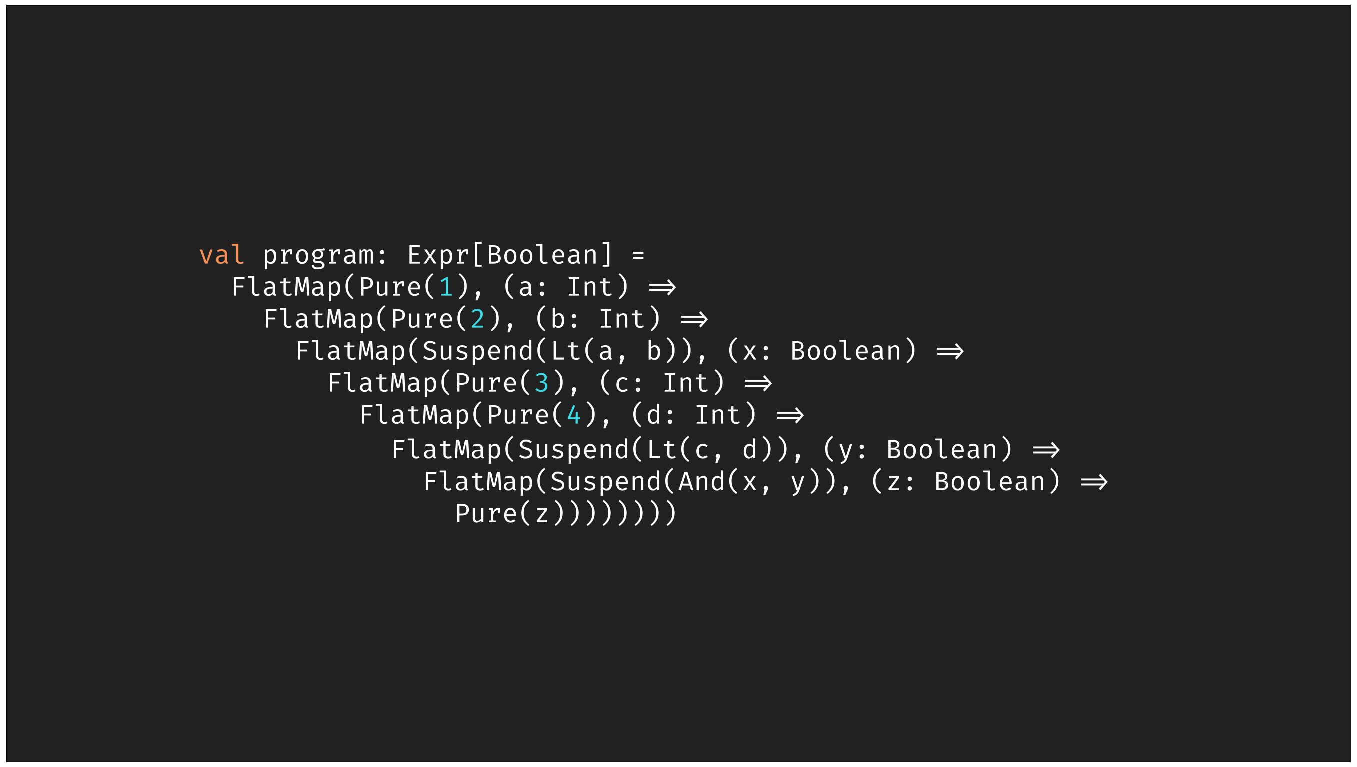
We introduce a second type parameter, `F`, that has a single type argument like `ExprAlg` does:

- `Suspect` now wraps up an `F` (an instance of the algebra);
- `Pure` and `FlatMap` work exactly as they did before.
-



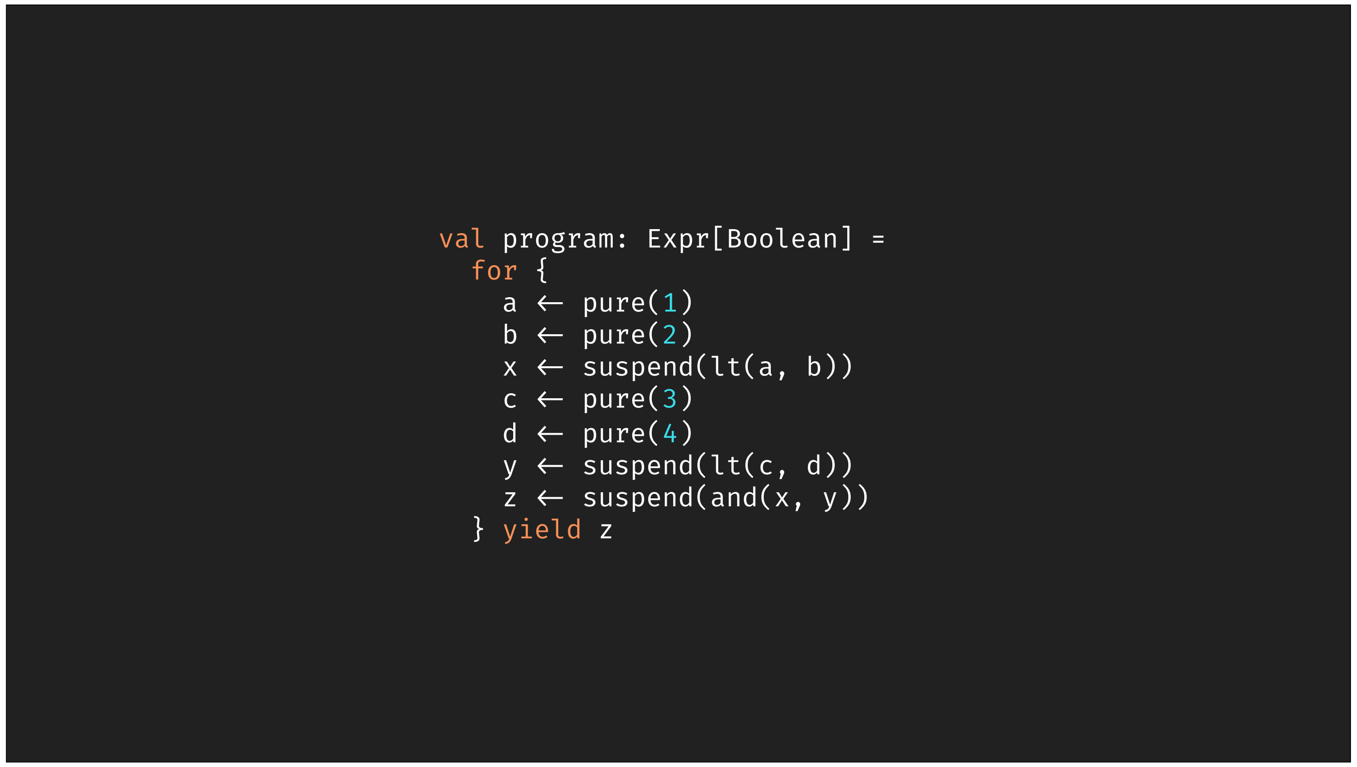
```scala
sealed trait ExprAlg[A]

case class Lt(a: Int, b: Int)
  extends ExprAlg[Boolean]

case class And(a: Boolean, b: Boolean)
  extends ExprAlg[Boolean]
```

When we combine this with `ExprAlg`, we end up with a DSL that can represent programs like this…



```scala
val program: Expr[Boolean] =
  FlatMap(Pure(1), (a: Int) ⇒
    FlatMap(Pure(2), (b: Int) ⇒
      FlatMap(Suspend(Lt(a, b)), (x: Boolean) ⇒
        FlatMap(Pure(3), (c: Int) ⇒
          FlatMap(Pure(4), (d: Int) ⇒
            FlatMap(Suspend(Lt(c, d)), (y: Boolean) ⇒
              FlatMap(Suspend(And(x, y)), (z: Boolean) ⇒
                Pure(z))))))))
```

This is almost the same as the monadic example we gave before, except that any instance of `Lt` or `And` is wrapped in `Suspend`. It's a simple indirection that's a small price to pay for this extra generality.



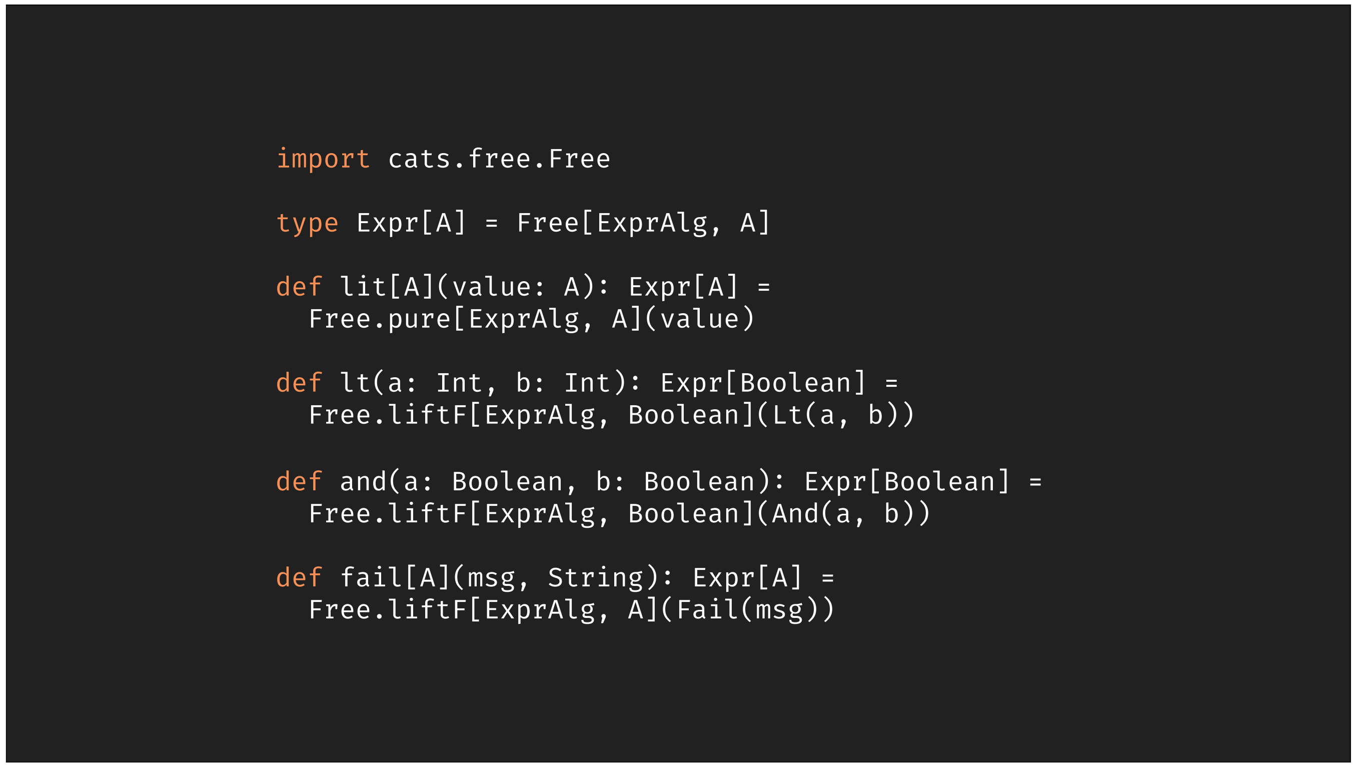
```
val program: Expr[Boolean] =
  for {
    a ← pure(1)
    b ← pure(2)
    x ← suspend(lt(a, b))
    c ← pure(3)
    d ← pure(4)
    y ← suspend(lt(c, d))
    z ← suspend(and(x, y))
  } yield z
```

And with a bit of syntax we can produce a `for` comprehension that makes the structure of the program very clear.

However, this isn’t quite how we’d normally use `Free` in practice.

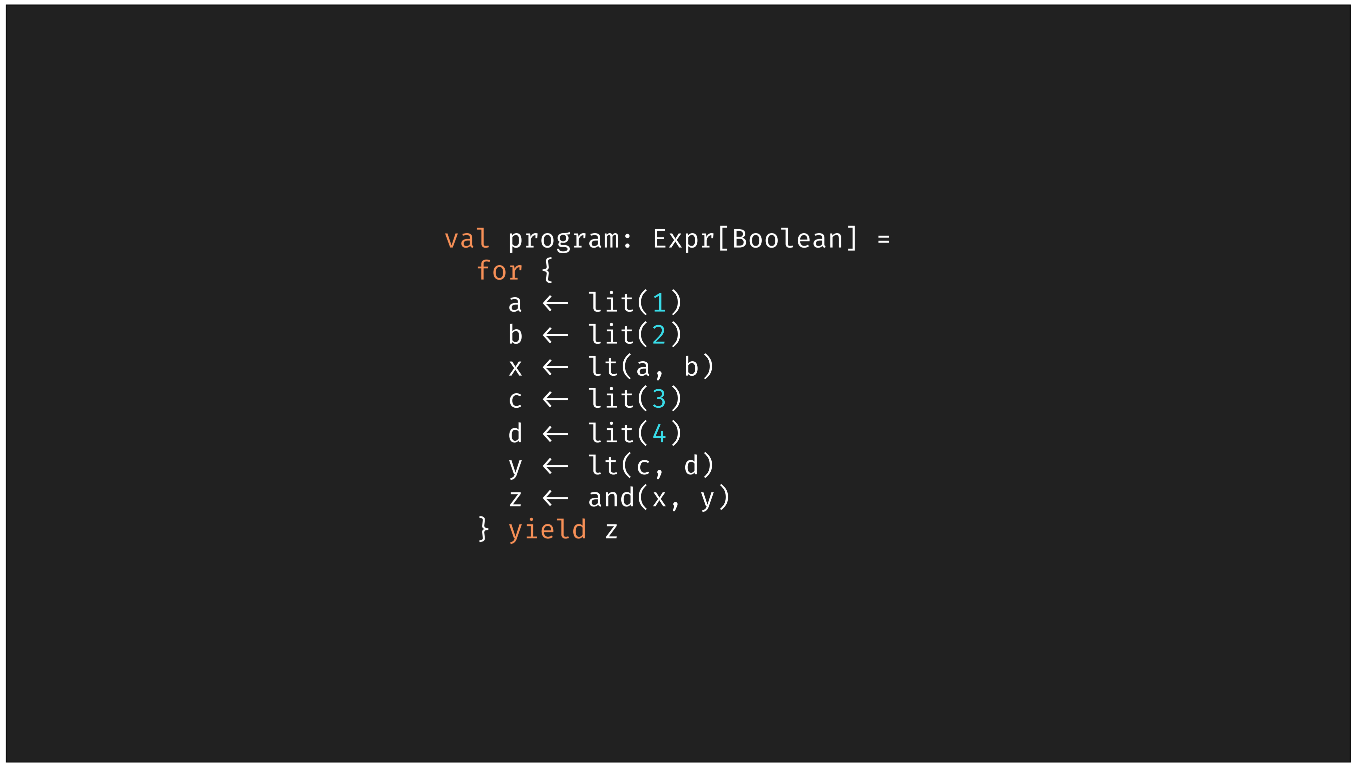
```scala
import cats.free.Free

type Expr[A] = Free[ExprAlg, A]
```

Typically we'd define our algebra as before, import `Free` from our favourite functional programming library, and create a type alias that specialises `Free` to our algebra.



```scala
import cats.free.Free

type Expr[A] = Free[ExprAlg, A]

def lit[A](value: A): Expr[A] =
  Free.pure[ExprAlg, A](value)

def lt(a: Int, b: Int): Expr[Boolean] =
  Free.liftF[ExprAlg, Boolean](Lt(a, b))

def and(a: Boolean, b: Boolean): Expr[Boolean] =
  Free.liftF[ExprAlg, Boolean](And(a, b))

def fail[A](msg, String): Expr[A] =
  Free.liftF[ExprAlg, A](Fail(msg))
```

We’d then write a set of helper methods called “smart constructors” that produce an instance of any given term in our algebra pre-wrapped in a `Suspend`.



```
val program: Expr[Boolean] =
  for {
    a ← lit(1)
    b ← lit(2)
    x ← lt(a, b)
    c ← lit(3)
    d ← lit(4)
    y ← lt(c, d)
    z ← and(x, y)
  } yield z
```

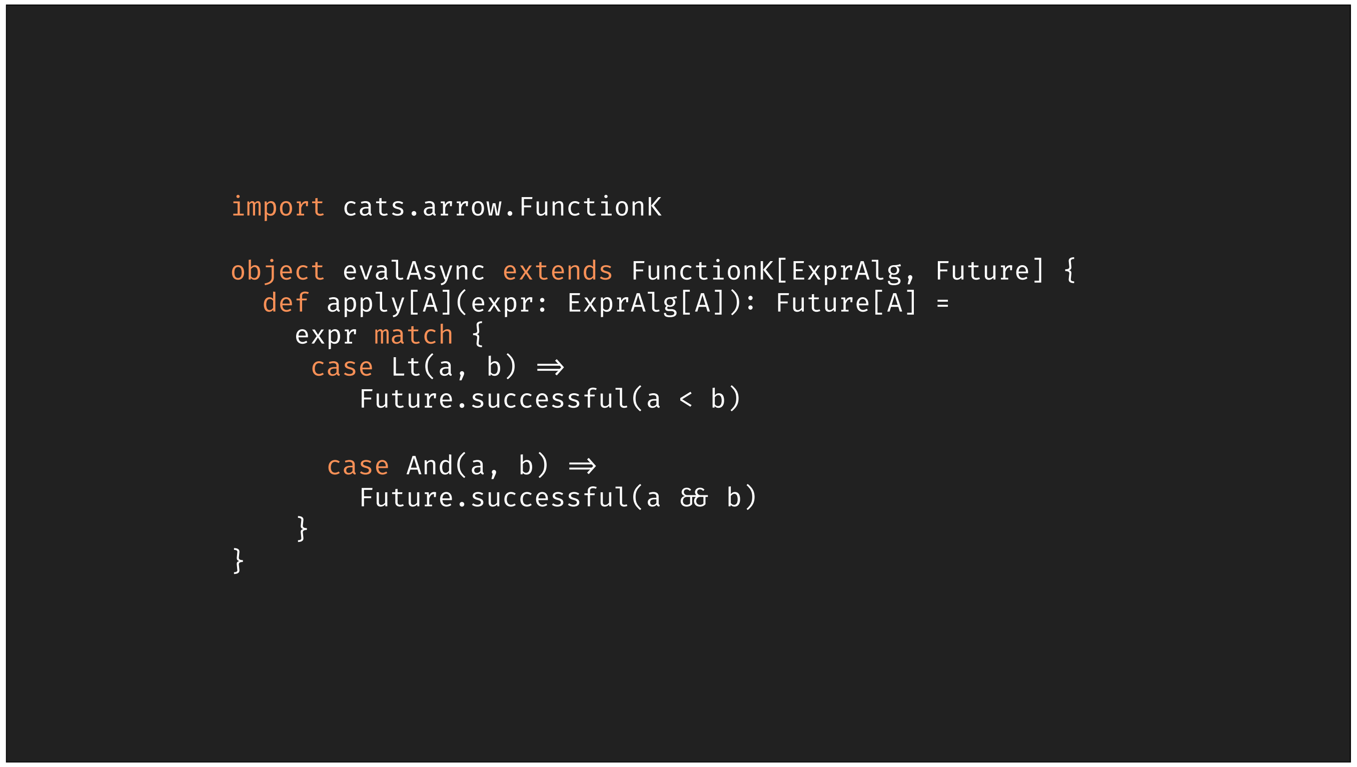
With these helpers, programs look a little simpler.

But what about the interpreter?

The interpreter is split into two sets of code:

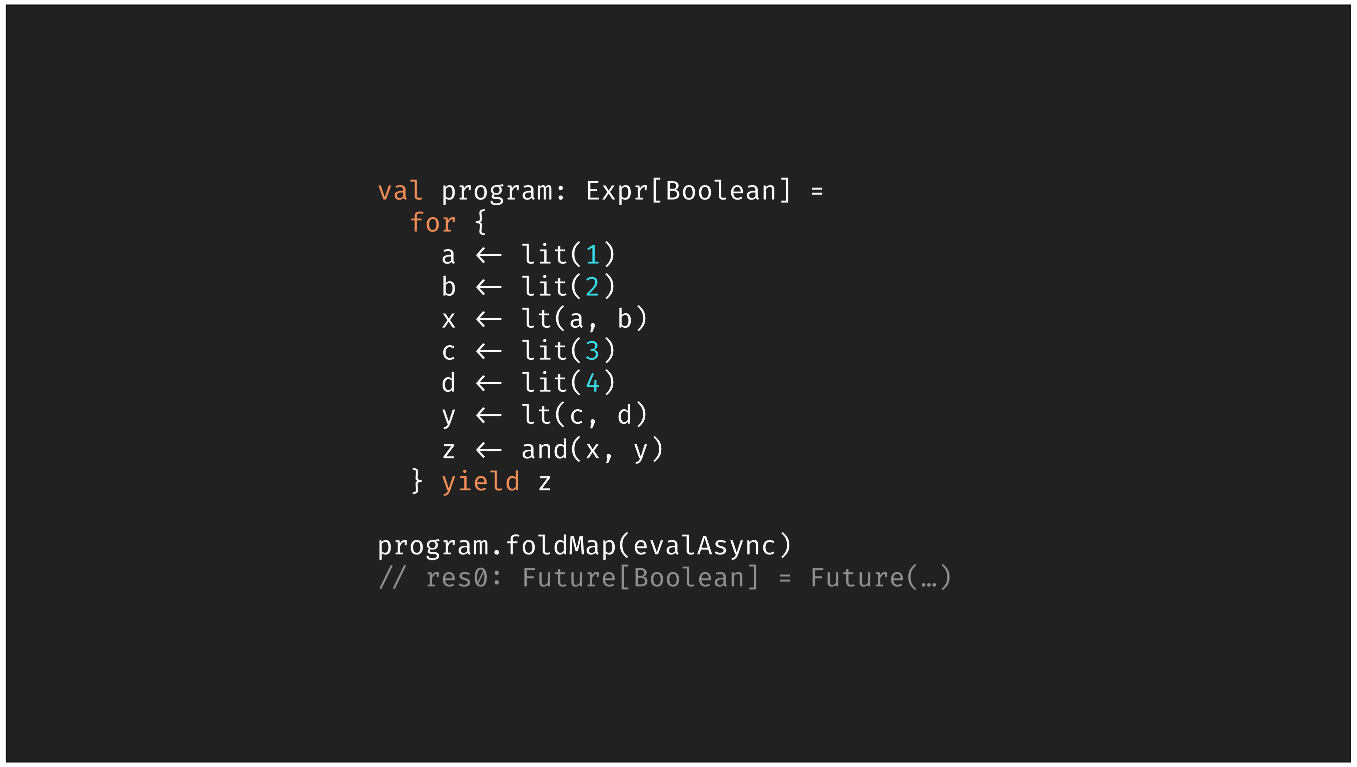
- code to interpret `Pure`, `Suspend`, and `FlatMap`;
- code to interpret instances of the algebra.

The first of these is provided for free (pun intended). We only have to specify the second.



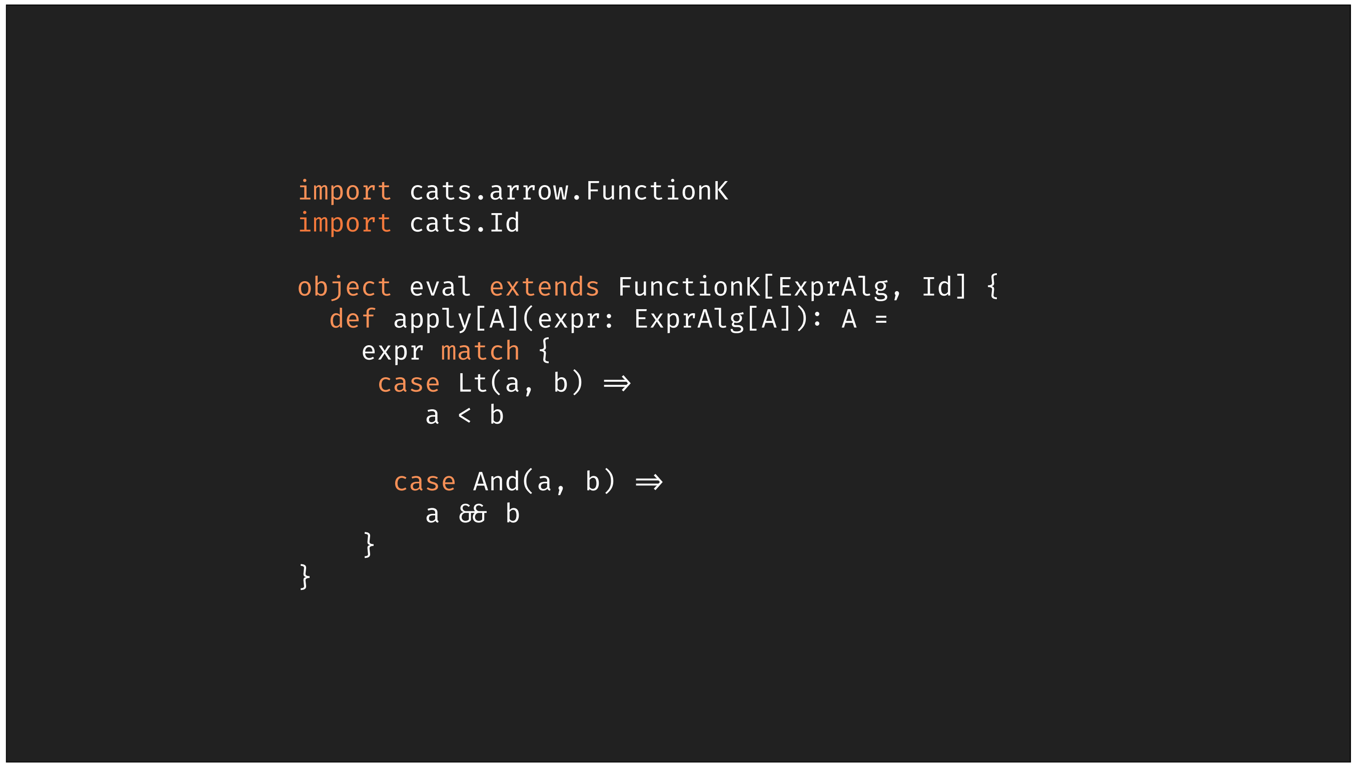
```scala
import cats.arrow.FunctionK

object evalAsync extends FunctionK[ExprAlg, Future] {
  def apply[A](expr: ExprAlg[A]): Future[A] =
    expr match {
     case Lt(a, b) =>
        Future.successful(a < b)

      case And(a, b) =>
        Future.successful(a && b)
    }
}
```

Here’s how we do it. The technical name for this is a “natural transformation”, which is a little beyond what we need to cover right now. The important part is the `apply()` method in the middle, which transforms an instance of `ExprAlg` into an instance of some other monad (in this case `Future`).



```
val program: Expr[Boolean] =
  for {
    a ← lit(1)
    b ← lit(2)
    x ← lt(a, b)
    c ← lit(3)
    d ← lit(4)
    y ← lt(c, d)
    z ← and(x, y)
  } yield z

program.foldMap(evalAsync)
// res0: Future[Boolean] = Future(…)
```

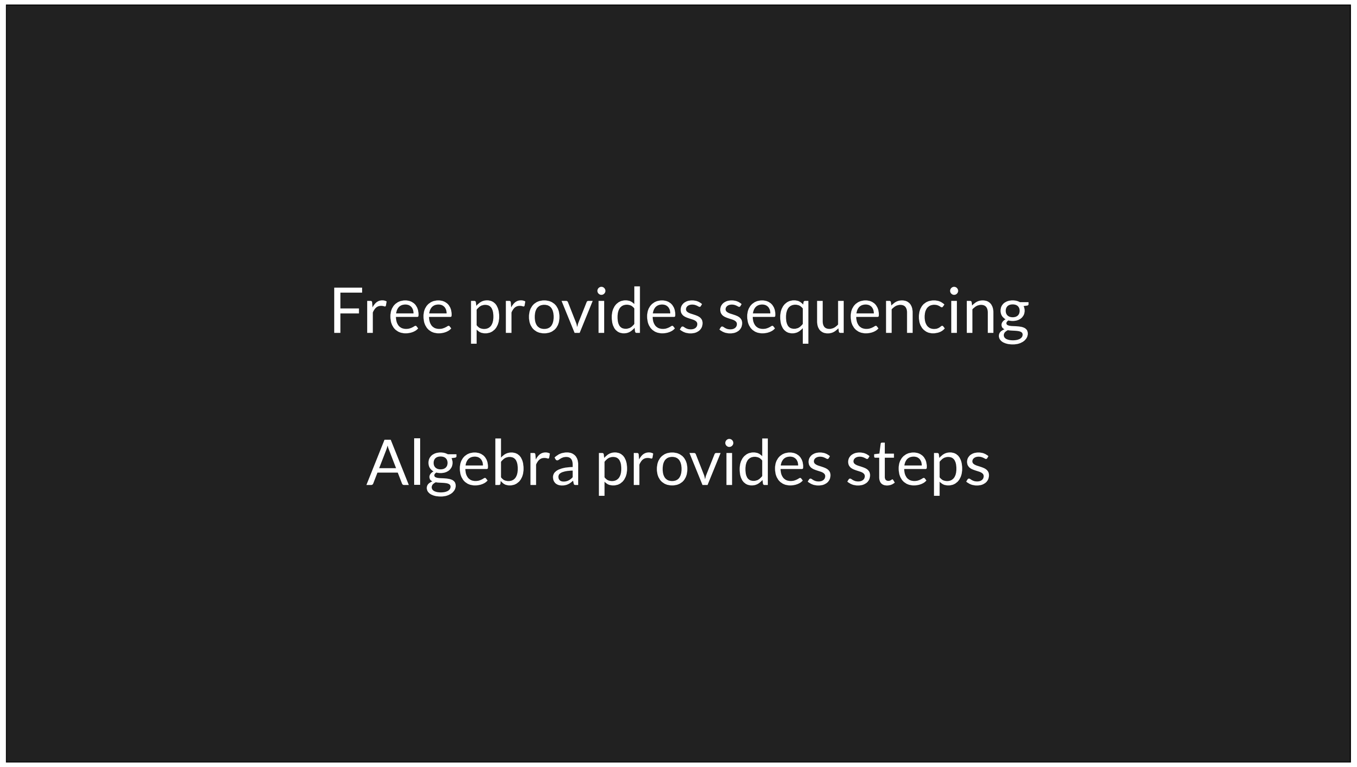
Once we've defined our natural transformation, we can pass it to a method called `foldMap()` on `Free` that handles the interpretation of `Pure`, `Suspend`, and `FlatMap`.



```scala
import cats.arrow.FunctionK
import cats.Id

object eval extends FunctionK[ExprAlg, Id] {
  def apply[A](expr: ExprAlg[A]): A =
    expr match {
     case Lt(a, b) ⇒
        a < b

      case And(a, b) ⇒
        a && b
    }
}
```

If we want to interpret to a plain value, rather than a value wrapped in a monad, we can use this thing called the “identity” monad that takes plain values and makes them look like monads.

If you don’t know how this works then don’t worry about it. Check out Chapter 4 of “Scala with Cats”, which I’ll link to at the end of the talk.



Free provides sequencing

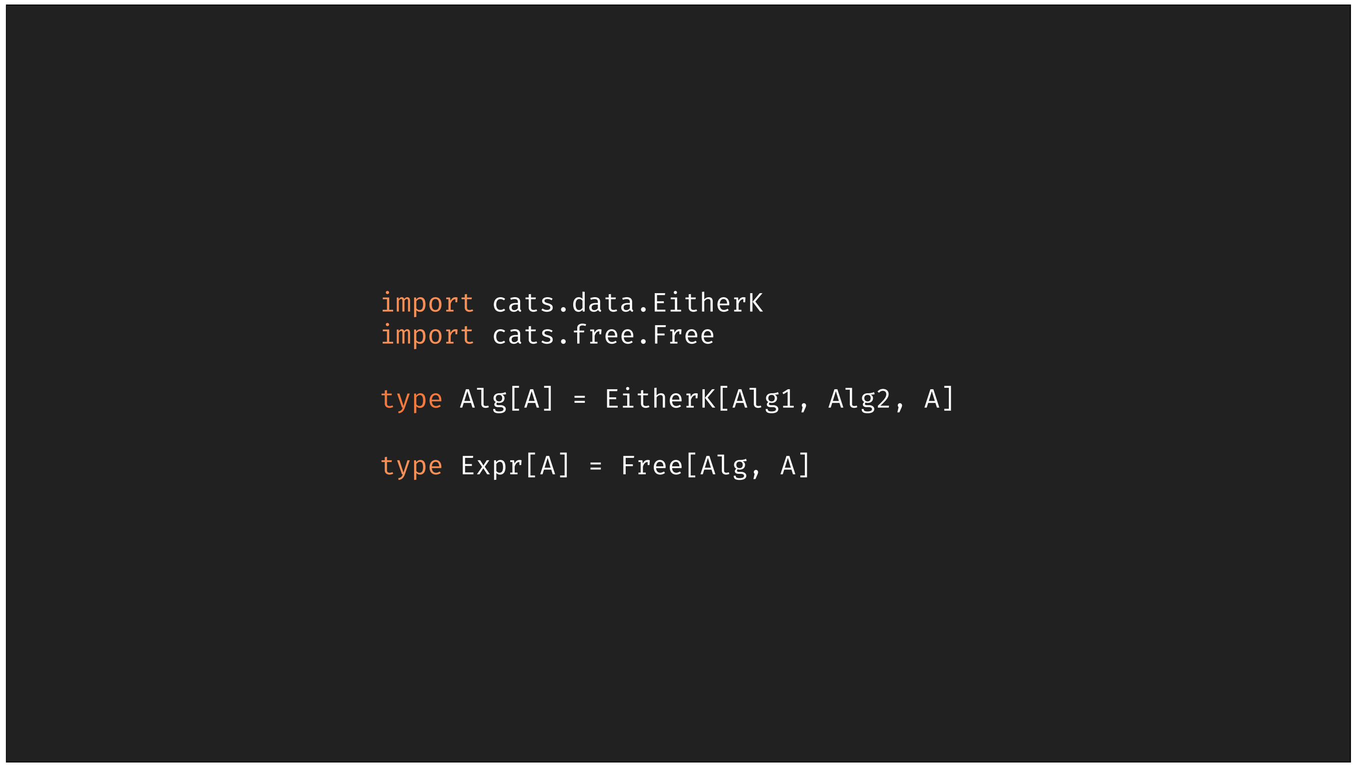
Algebra provides steps

The key take-home point about `Free` is that it factors out the sequencing aspect of our programs, allowing us to plug in implementations of the individual steps.

Combine DSLs

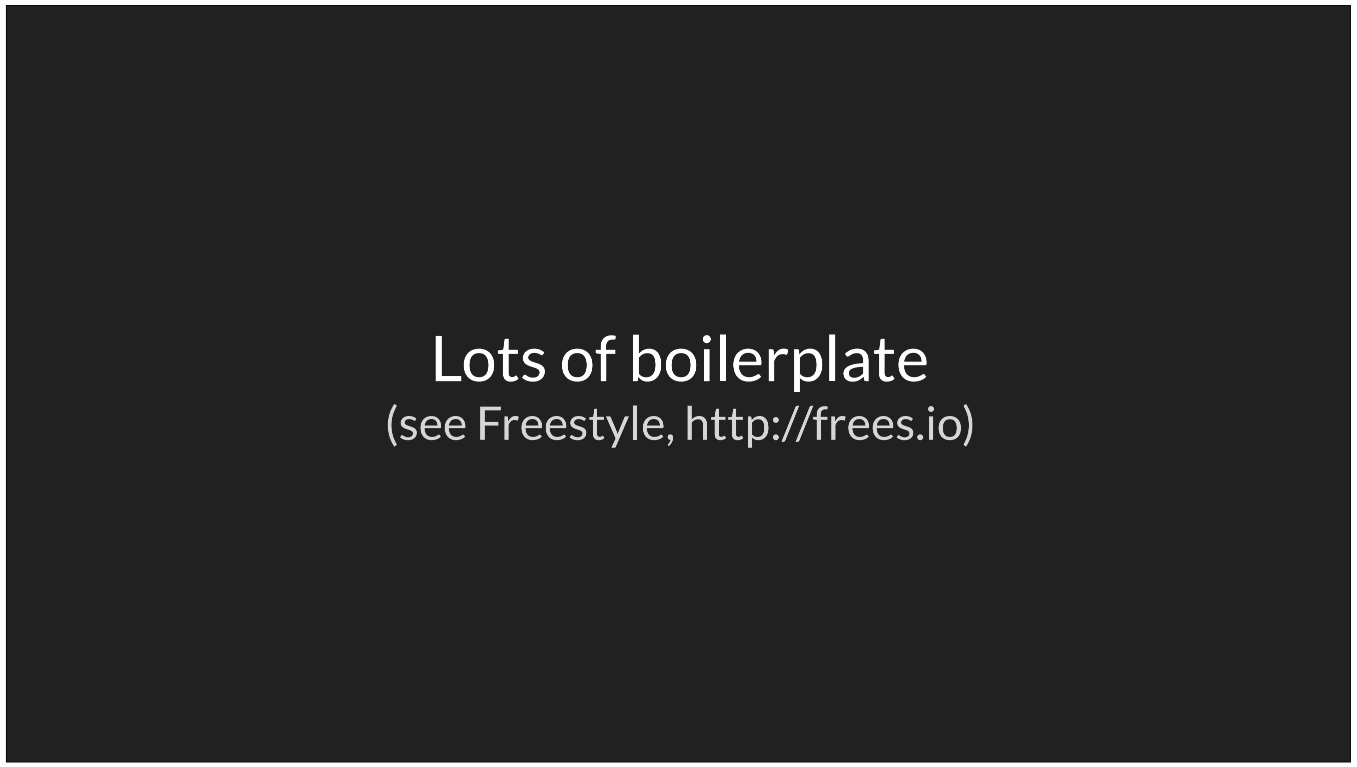
This has a huge benefit in that it allows us to combine multiple algebras within the same DSL.

Imagine we had two DSLs. If each had its own `FlatMap` step, it would be very hard to combine the two: we wouldn't know which `FlatMap` to use at each stage of the program.



```scala
import cats.data.EitherK
import cats.free.Free

type Alg[A] = EitherK[Alg1, Alg2, A]

type Expr[A] = Free[Alg, A]
```

With `Free`, though, we have one implementation of `FlatMap` and two algebras. There are a number of predefined data types, including this `EitherK` type from Cats, that allow us to combine algebras to produce one big algebra that is a disjunction of the steps in its constituent parts. If we do this, we can mix DSLs quite easily.


Lots of boilerplate
(see Freestyle, http://frees.io)

When we’re using Free, however, there is a lot of boilerplate. This is true of single-algebra implementations and doubly true of multi-algebra implementations. There are a few libraries out there that eliminate this boilerplate with clever Scala tricks.

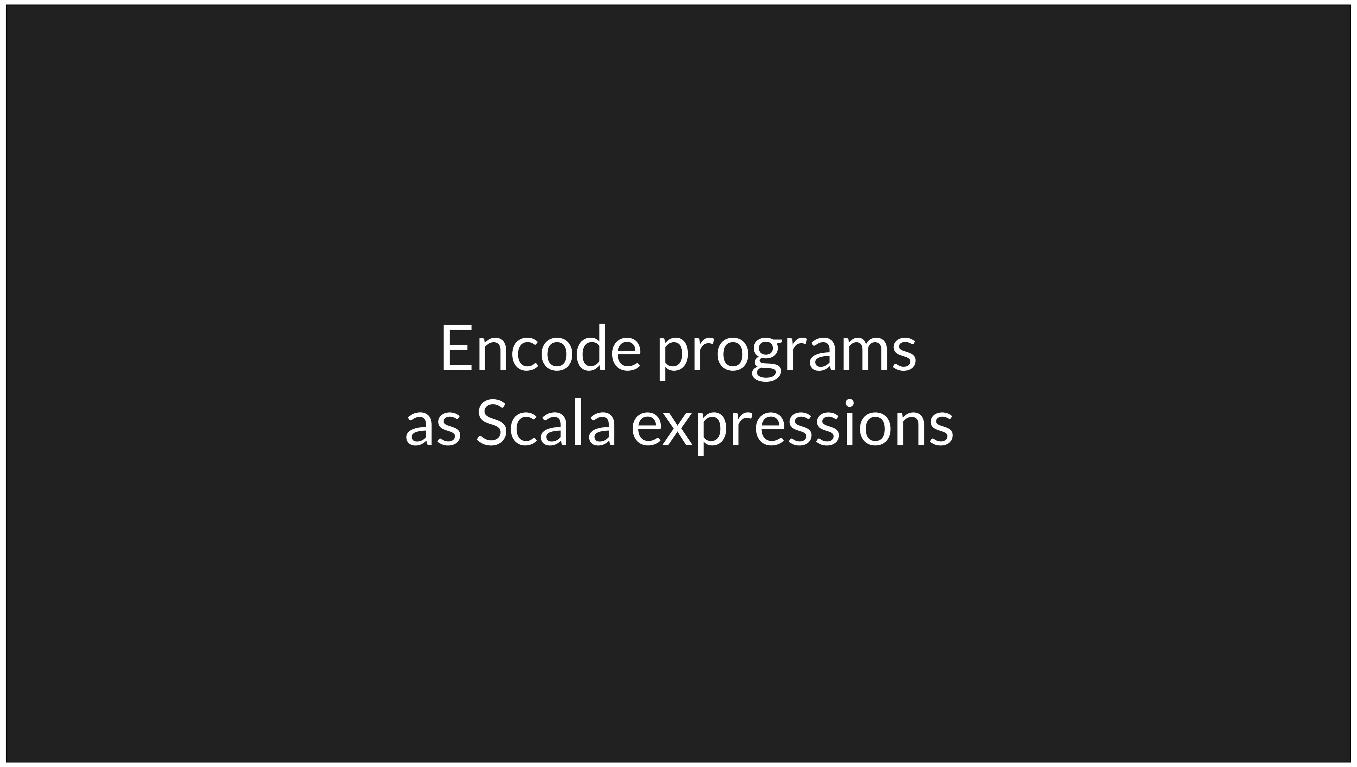
“Freek” is one example. “Freestyle” is another. I recommend looking at Freestyle for more information. It is incredibly opinionated, but it allows you to write quite complex DSLs with relatively little boilerplate.

Church encodings

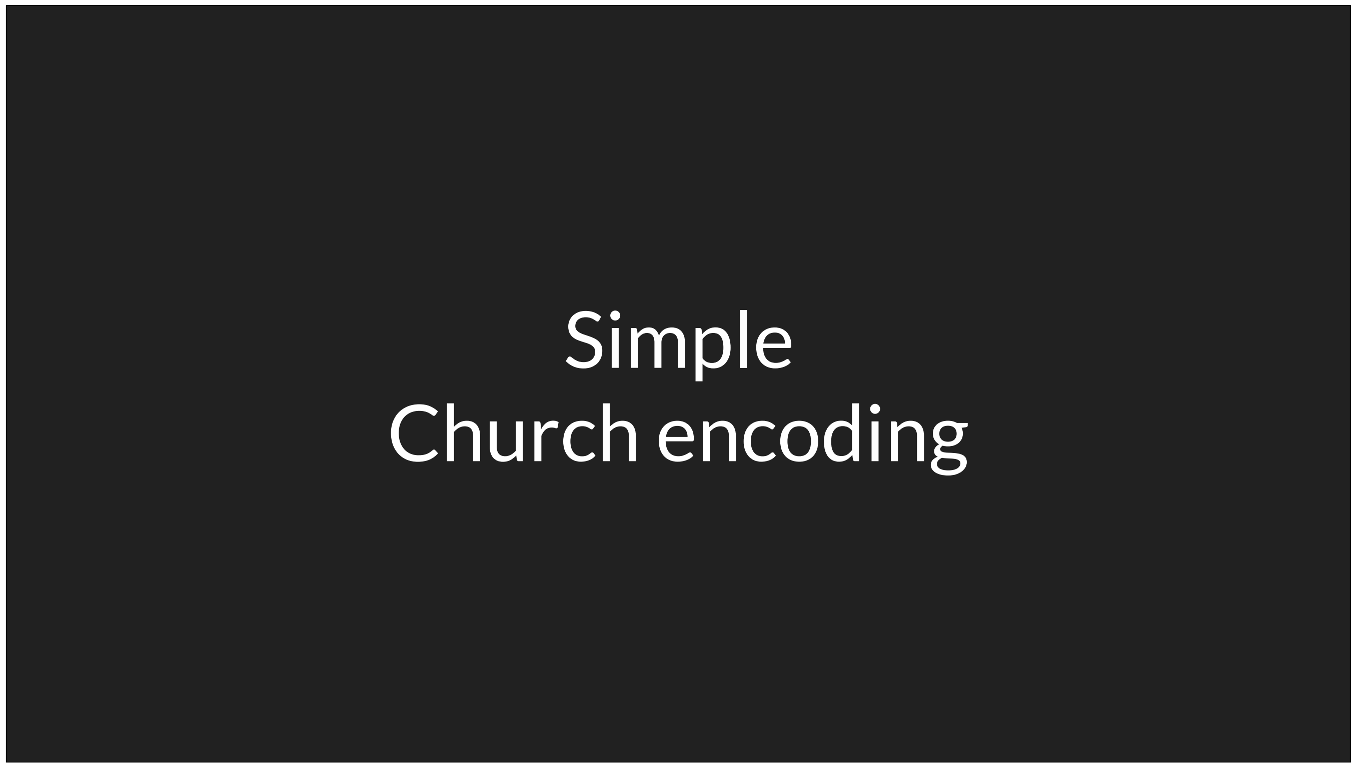
That's all I wanted to look at for reified encodings:

- we introduced the methodology: representing expressions as data;
- we investigated untyped and typed DSLs;
- we introduced a `FlatMap` step to allow the user to control ordering in a natural way;
- we motivated the `Free` monad, which allows us to combine DSLs and interpreters.

Now let's look at another approach to implementing DSLs: Church encoding.


Encode programs as Scala expressions

Church encoding is strictly equivalent to reification. We can convert back and forth between the two representations. However, in Church encoding we represent programs as sequences of abstract method calls rather than values.

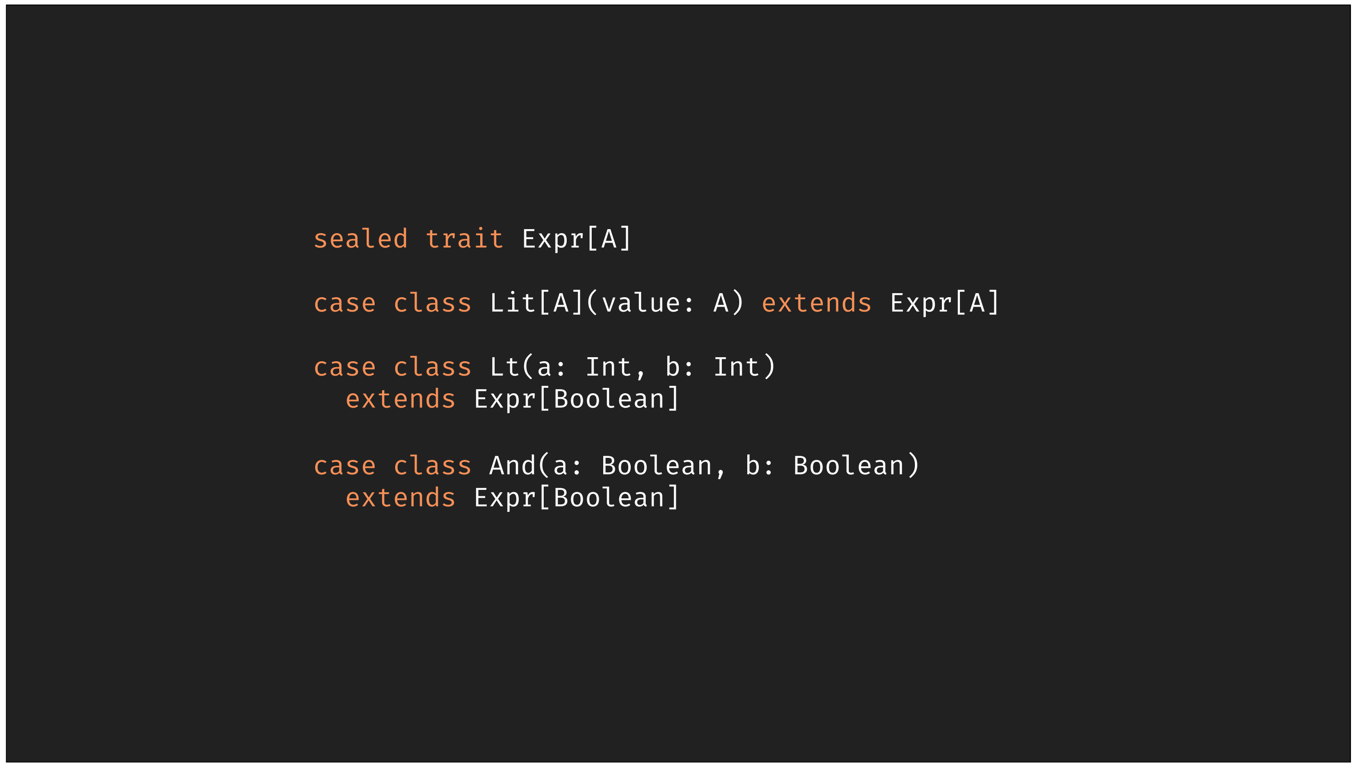


Simple
Church encoding

We’ll look at two examples of Church encoding.

We’ll start with a simple example that shows the correlation to reification.

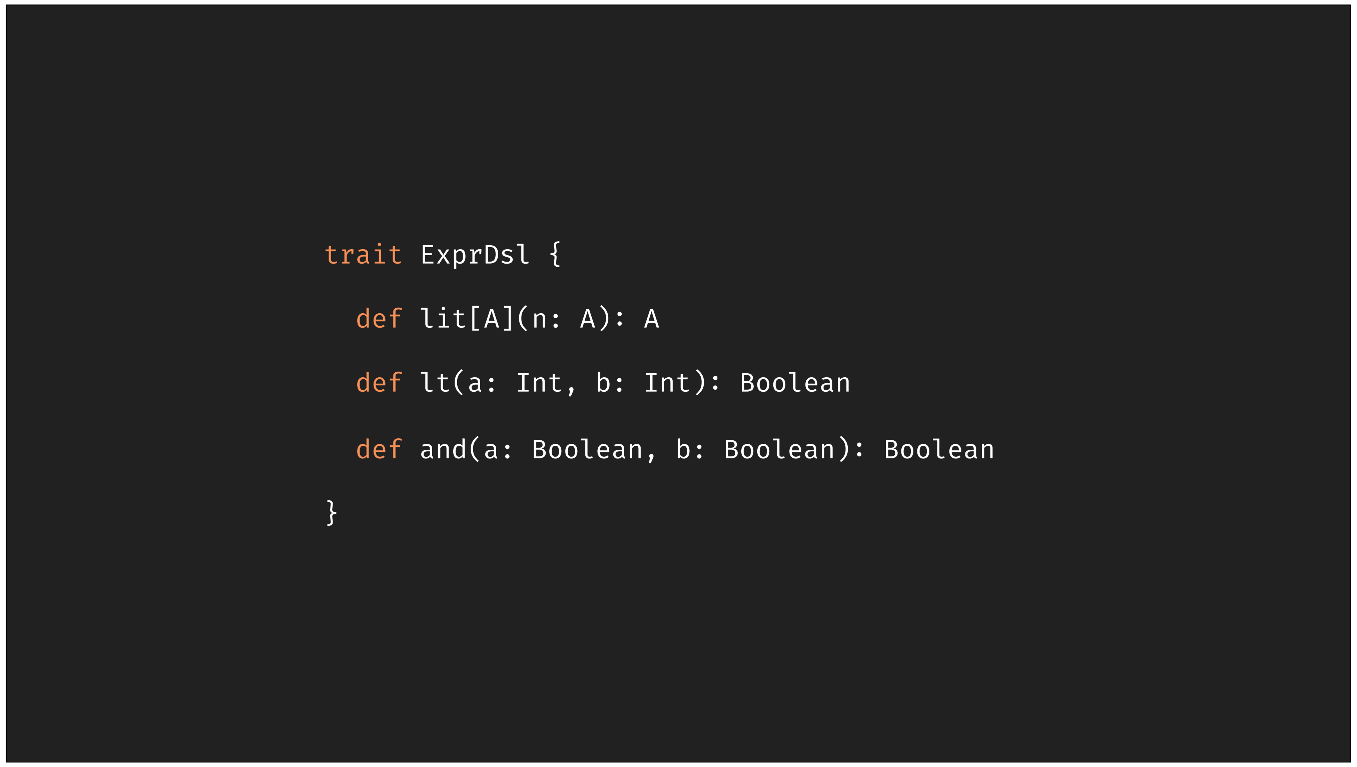
Then we’ll move on to look at tagless final encoding.



```scala
sealed trait Expr[A]

case class Lit[A](value: A) extends Expr[A]

case class Lt(a: Int, b: Int)
  extends Expr[Boolean]

case class And(a: Boolean, b: Boolean)
  extends Expr[Boolean]
```

We can church encode any sum type by converting it to an object with a method for each subtype.

Compare the two versions of this DSL to see the equivalence…

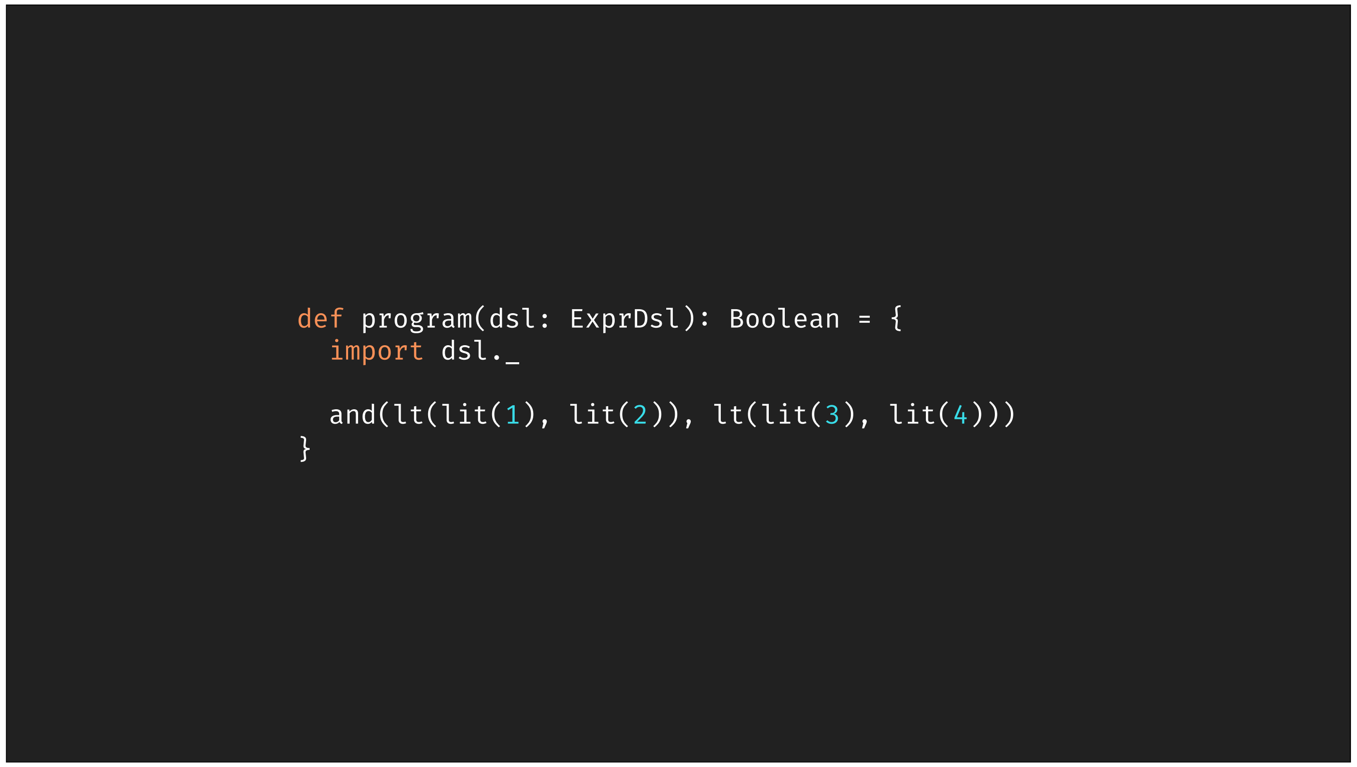


```scala
trait ExprDsl {
  def lit[A](n: A): A
  def lt(a: Int, b: Int): Boolean
  def and(a: Boolean, b: Boolean): Boolean
}
```

The transformation is mechanical:

- our sealed trait becomes an object. I’ve renamed the object slightly here to avoid confusion;
- each case class becomes a method;
- each field becomes a parameter;
- the result type of the case class becomes the return type of the method.

We can represent programs as Scala expressions involving calls to these methods.

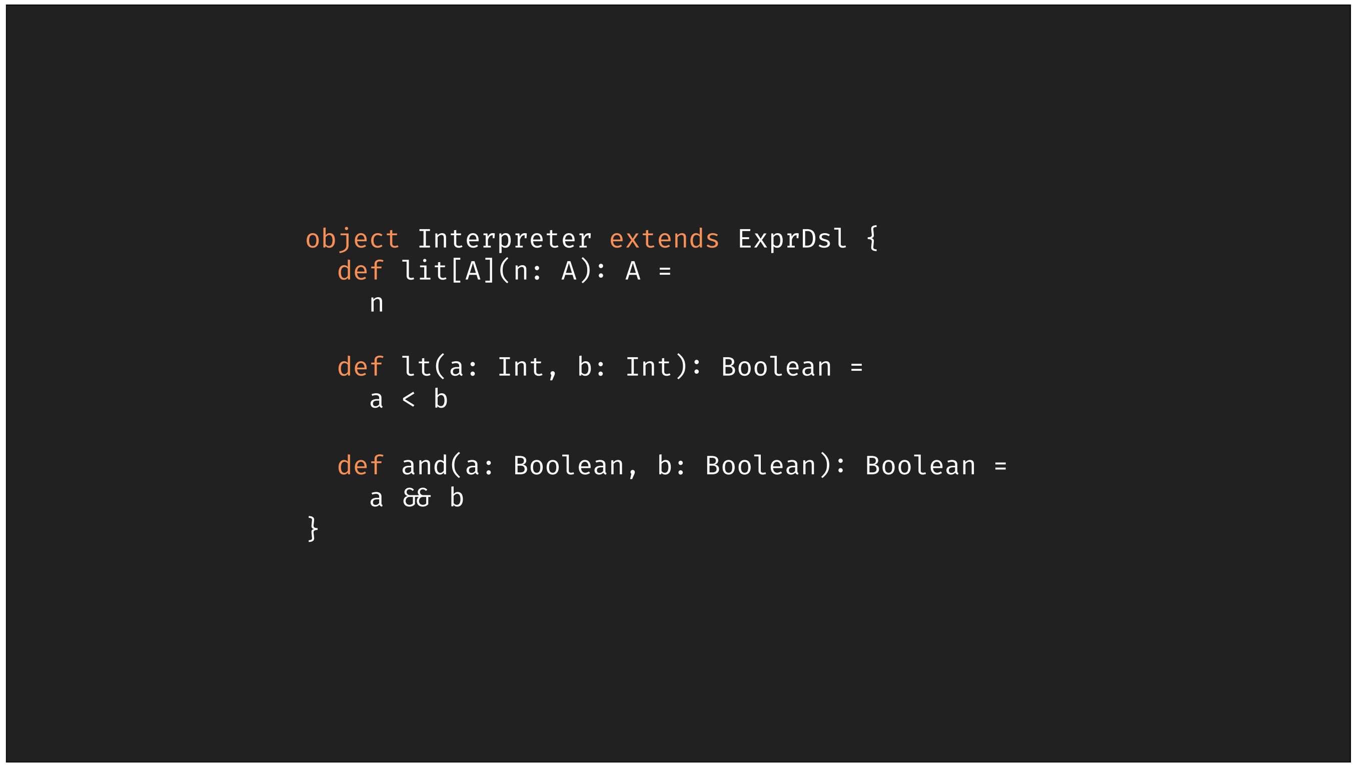


```scala
def program(dsl: ExprDsl): Boolean = {
  import dsl._

  and(lt(lit(1), lit(2)), lt(lit(3), lit(4)))
}
```

Here we're building a program by:

- taking an instance of our DSL as a parameter;
- running various methods from the DSL in a program-specific order.

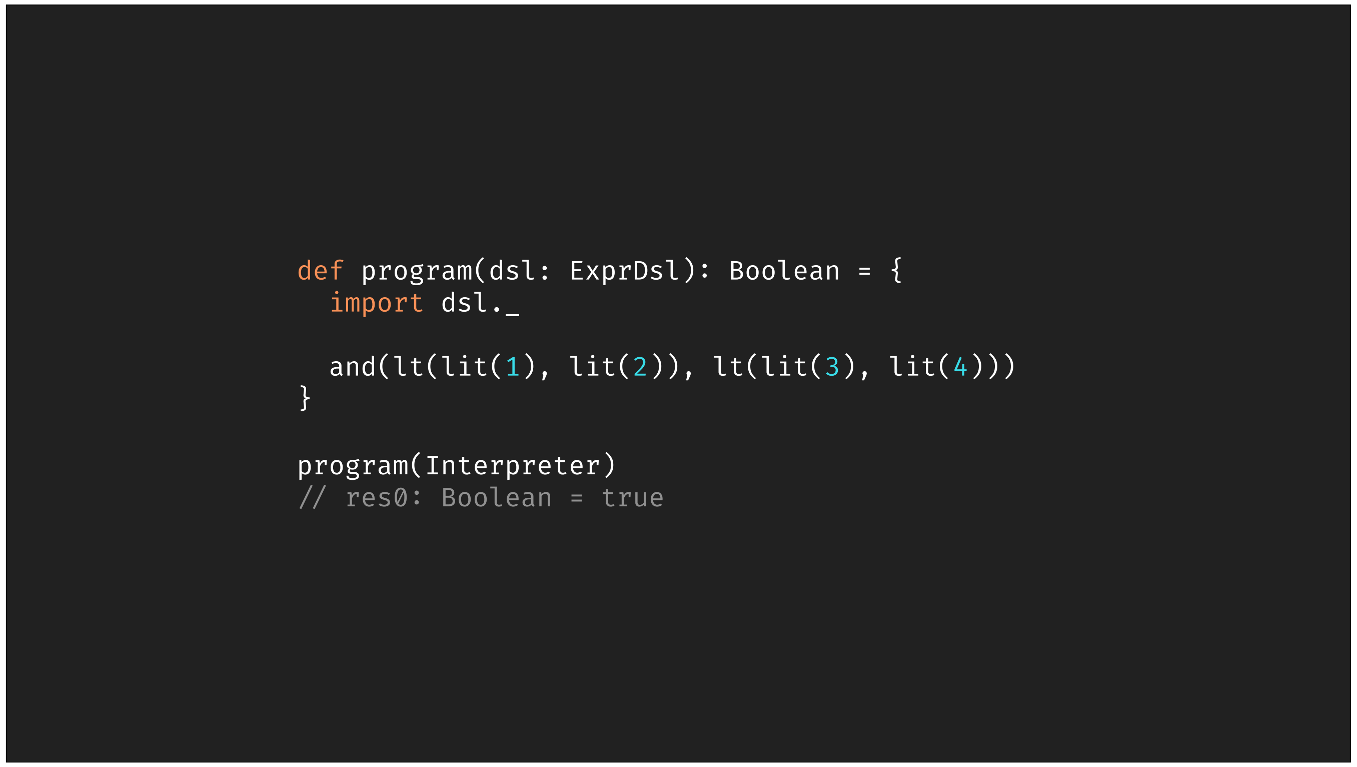
Because `ExprDsl` is a trait, there's no semantics here. The program is still doing what it did before: specifying a sequence of abstract steps.



```scala
object Interpreter extends ExprDsl {
  def lit[A](n: A): A =
    n

  def lt(a: Int, b: Int): Boolean =
    a < b

  def and(a: Boolean, b: Boolean): Boolean =
    a && b
}
```

We give the steps meaning by creating a concrete instance of `ExprDsl`.

In this case I’m mapping each method onto the simplest corresponding Scala expression.



```
def program(dsl: ExprDsl): Boolean = {
  import dsl._

  and(lt(lit(1), lit(2)), lt(lit(3), lit(4)))
}

program(Interpreter)
// res0: Boolean = true
```

We can run a program by calling its method, passing in a concrete instance of the DSL.

This may seem slightly odd at first, but it really is equivalent to what we were doing before. In the reified encoding, we pass the program to the interpreter as a parameter. In this encoding, we pass the interpreter to the program to enable it to run.

Can’t abstract over effects

Note that we can’t really implement many different versions of the interpreter here. The method signatures are too specific. We also can’t abstract over effects. To do this we need to introduce a level of indirection.

Tagless final encoding

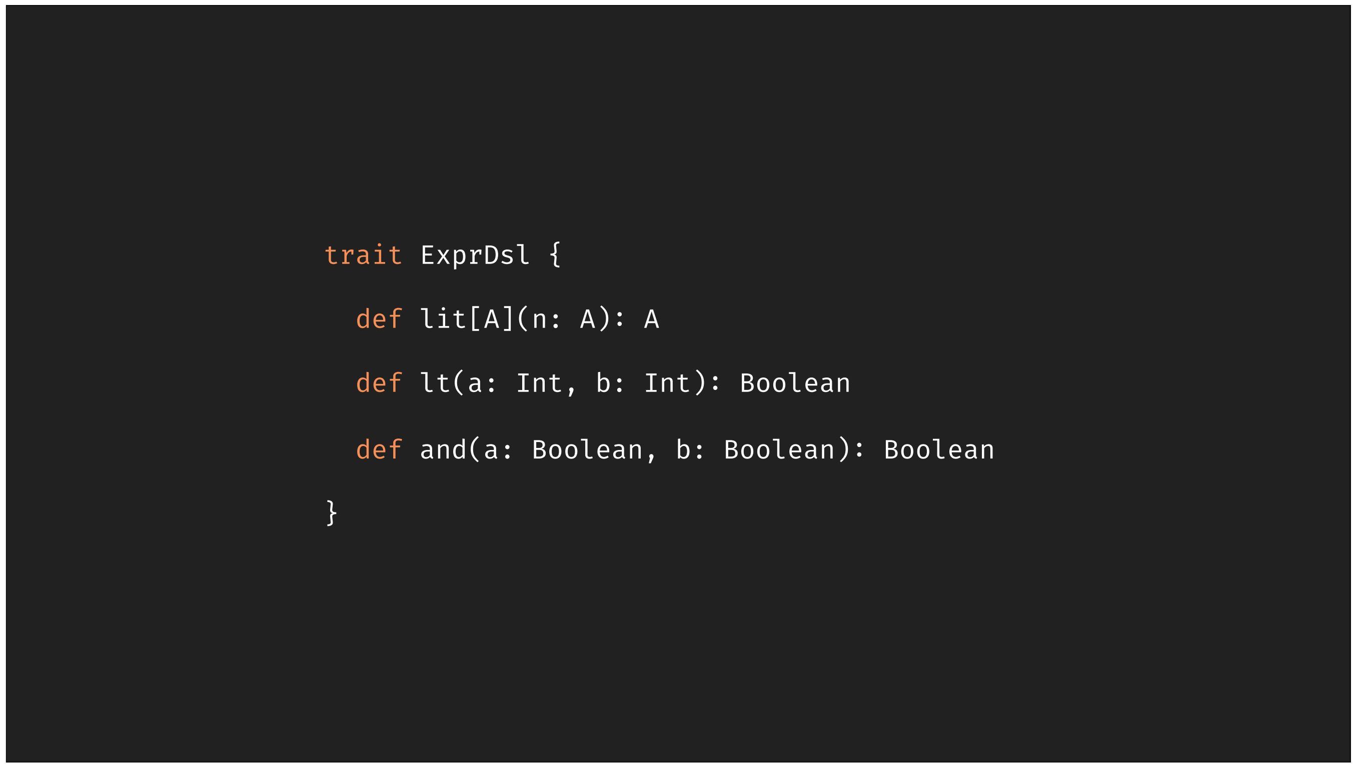
This brings us to tagless final encoding…

Tagless final encoding
(Finally tagless encoding?)

…or is it finally tagless encoding?

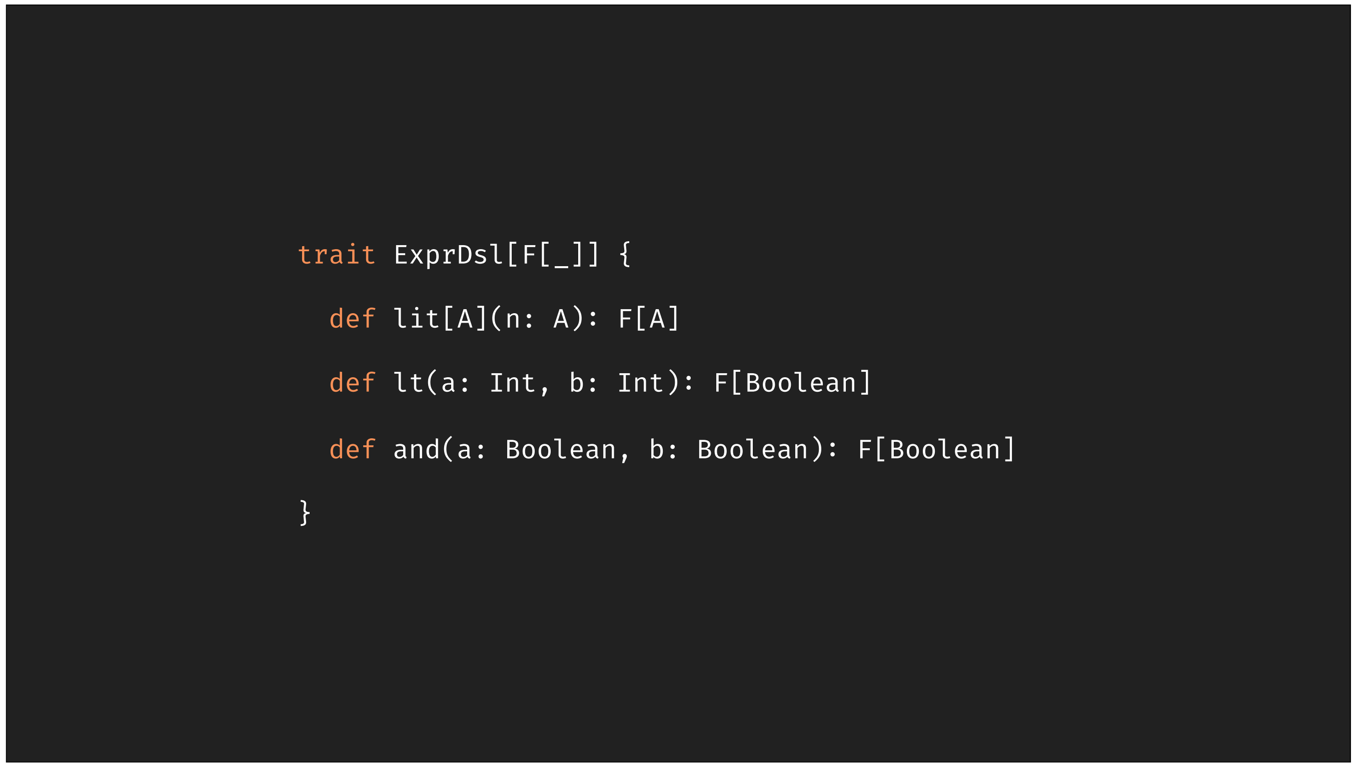
Who knows.

Regardless, this is a subtle modification to our Church encoded model that allows us to abstract over effects.



```
trait ExprDsl {
  def lit[A](n: A): A
  def lt(a: Int, b: Int): Boolean
  def and(a: Boolean, b: Boolean): Boolean
}
```

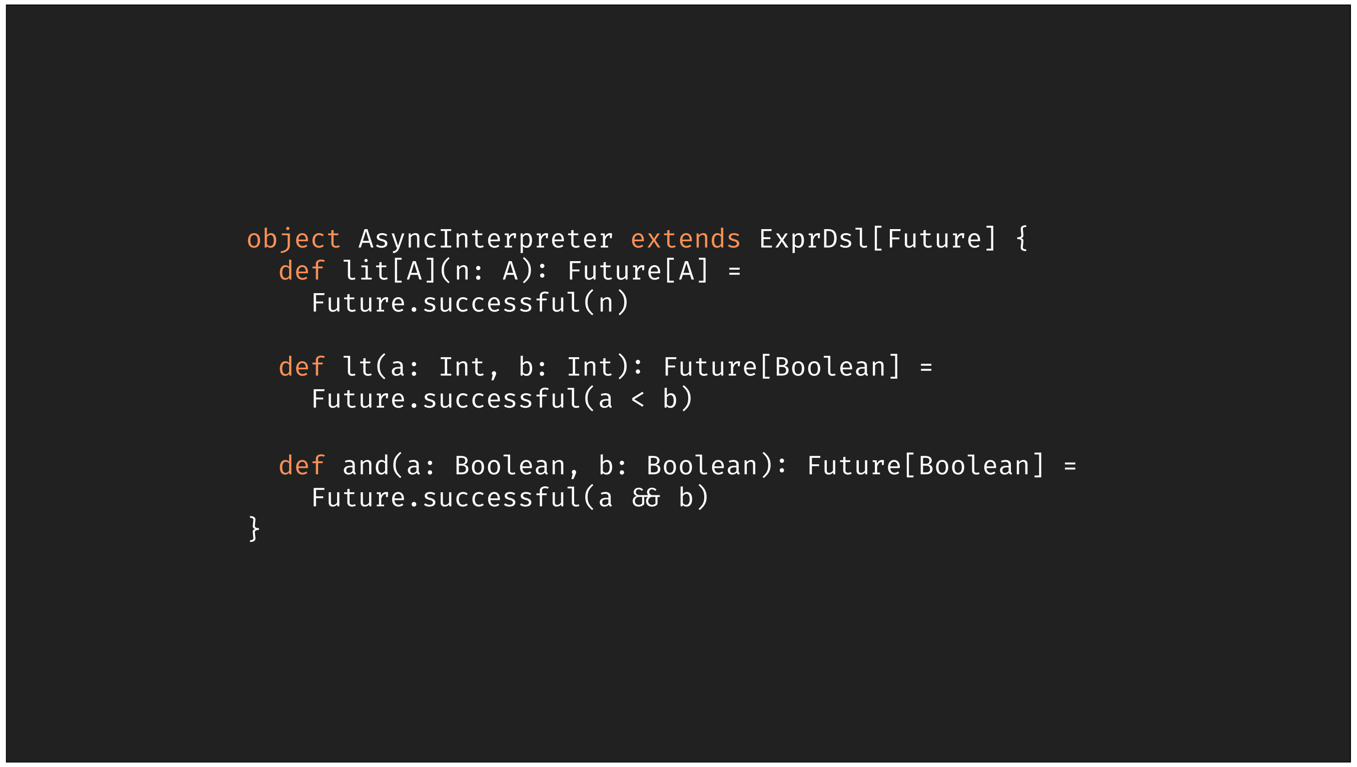
Here’s our existing DSL. All we have to do to abstract over effects is to introduce a type constructor around the return types.



```scala
trait ExprDsl[F[_]] {
  def lit[A](n: A): F[A]
  def lt(a: Int, b: Int): F[Boolean]
  def and(a: Boolean, b: Boolean): F[Boolean]
}
```

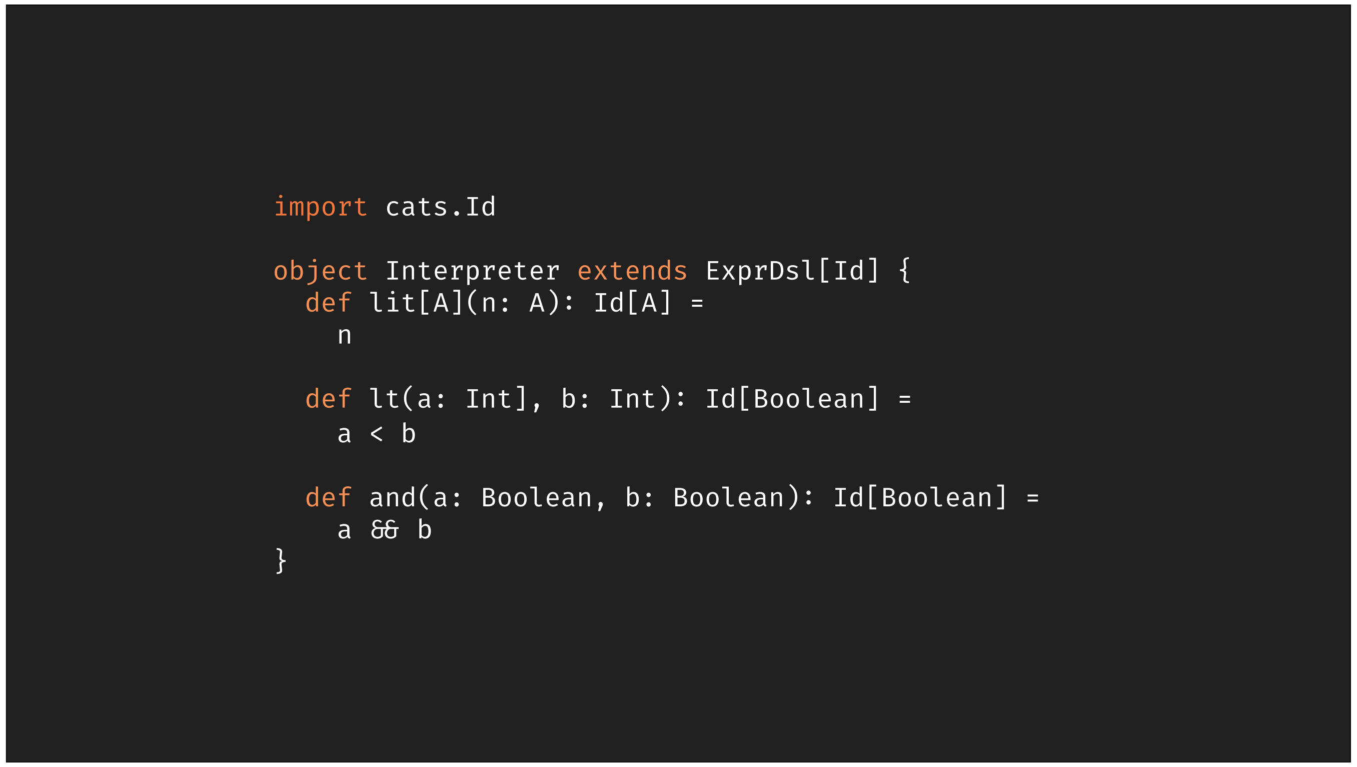
We introduce some new type `F` as a parameter to `ExprDsl`. Every method returns an `F`.

We can bind `F` to different concrete types in different interpreters.



```scala
object AsyncInterpreter extends ExprDsl[Future] {
  def lit[A](n: A): Future[A] =
    Future.successful(n)

  def lt(a: Int, b: Int): Future[Boolean] =
    Future.successful(a < b)

  def and(a: Boolean, b: Boolean): Future[Boolean] =
    Future.successful(a && b)
}
```

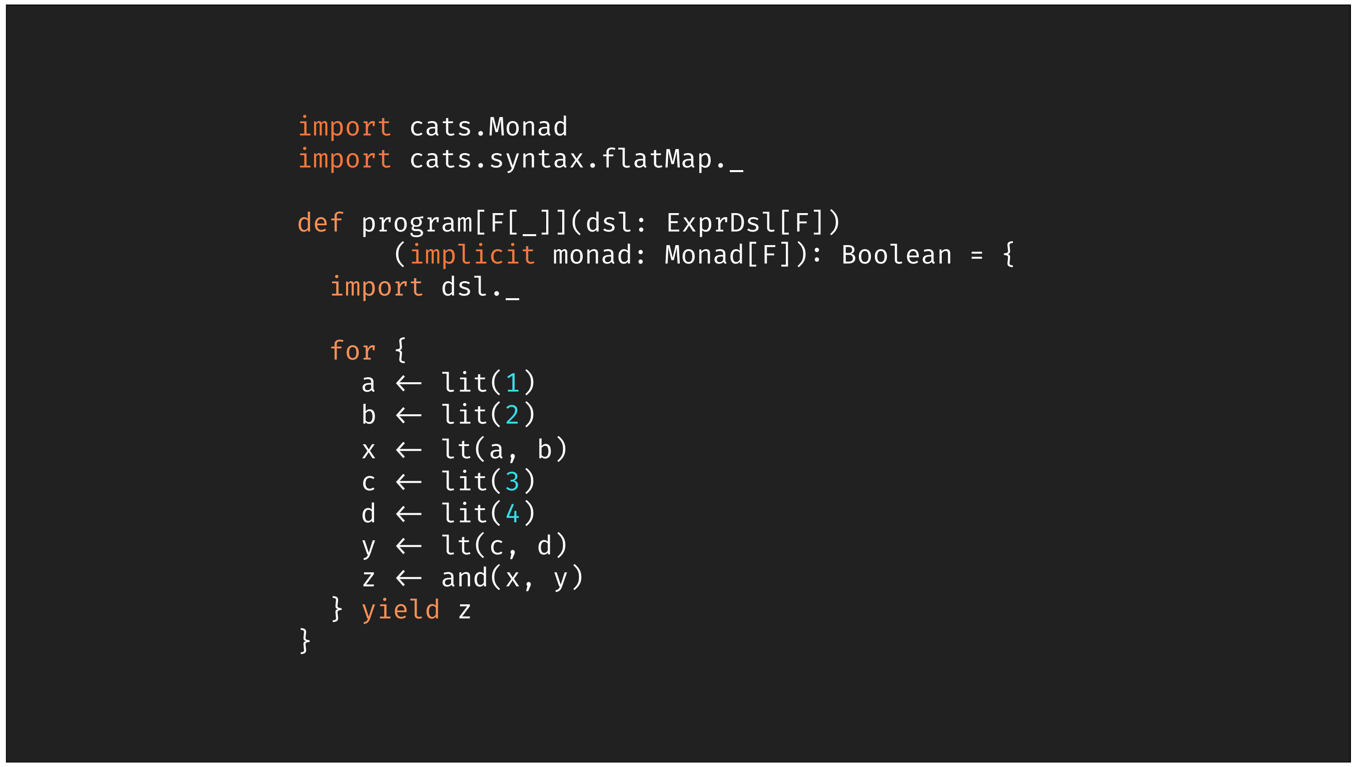
Here’s an interpreter that binds `F` to `Future`, creating an asynchronous version of our DSL.



```scala
import cats.Id

object Interpreter extends ExprDsl[Id] {
  def lit[A](n: A): Id[A] =
    n

  def lt(a: Int], b: Int): Id[Boolean] =
    a < b

  def and(a: Boolean, b: Boolean): Id[Boolean] =
    a && b
}
```

Here’s an interpreter that binds `F` to `Id`, creating a vanilla synchronous DSL.

And we can go on.



```scala
import cats.Monad
import cats.syntax.flatMap._

def program[F[_]](dsl: ExprDsl[F])
      (implicit monad: Monad[F]): Boolean = {
  import dsl._

  for {
    a ← lit(1)
    b ← lit(2)
    x ← lt(a, b)
    c ← lit(3)
    d ← lit(4)
    y ← lt(c, d)
    z ← and(x, y)
  } yield z
}
```

When we create a program, we still pass an interpreter as a parameter.

However, we also pass other constraints to give us more functionality in `F`. Here we’re passing a `Monad[F]`, which allows us to `map()` and `flatMap()` in our program as well as call the methods of the DSL.

```scala
import cats.Monad
import cats.syntax.flatMap._

def program[F[_]](dsl: ExprDsl[F])
      (implicit monad: Monad[F]): Boolean = {
  import dsl._

  for {
    a ← lit(1)
    b ← lit(2)
    x ← lt(a, b)
    c ← lit(3)
    d ← lit(4)
    y ← lt(c, d)
    z ← and(x, y)
  } yield z
}

program(AsyncInterpreter)
// res0: Future[Boolean] = Future(…)
```

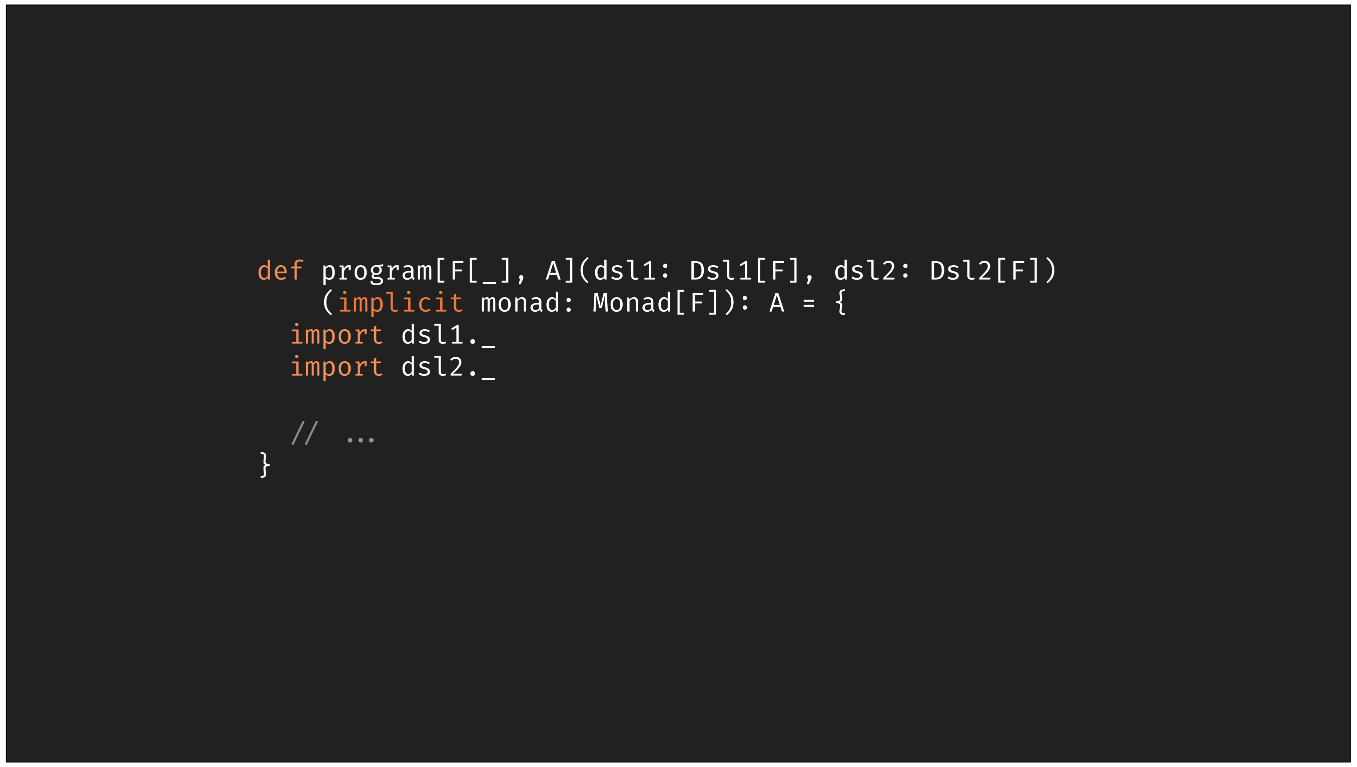
When we call `program()`, we bind `F` to a particular type. As long as there is an interpreter and a `Monad` instance for that type, our program will run.

```scala
import cats.Monad
import cats.syntax.flatMap._

def program[F[_]](dsl: ExprDsl[F])
      (implicit monad: Monad[F]): Boolean = {
  import dsl._

  for {
    a ← lit(1)
    b ← lit(2)
    x ← lt(a, b)
    c ← lit(3)
    d ← lit(4)
    y ← lt(c, d)
    z ← and(x, y)
  } yield z
}

program(Interpreter)
// res0: Boolean = true
```

Passing a different interpreter gives us the same result wrapped in a different effect, as expected.

Combine DSLs

Given the amount of hassle we had combining DSLs with Free, it is interesting to note that combining DSLs it trivial in this encoding.



```scala
def program[F[_], A](dsl1: Dsl1[F], dsl2: Dsl2[F])
    (implicit monad: Monad[F]): A = {
  import dsl1._
  import dsl2._

  // ...
}
```

We simply pass two DSLs as parameters to our program instead of one. If we restrict them to the same `F` type, we get the same sharing of `flatMap()` and `map()` that `Free` gave us.

Difficult to inspect

Like `Free`, our programs are somewhat opaque and difficult to inspect. However, in many cases we don’t need this..

Less boilerplate than Free

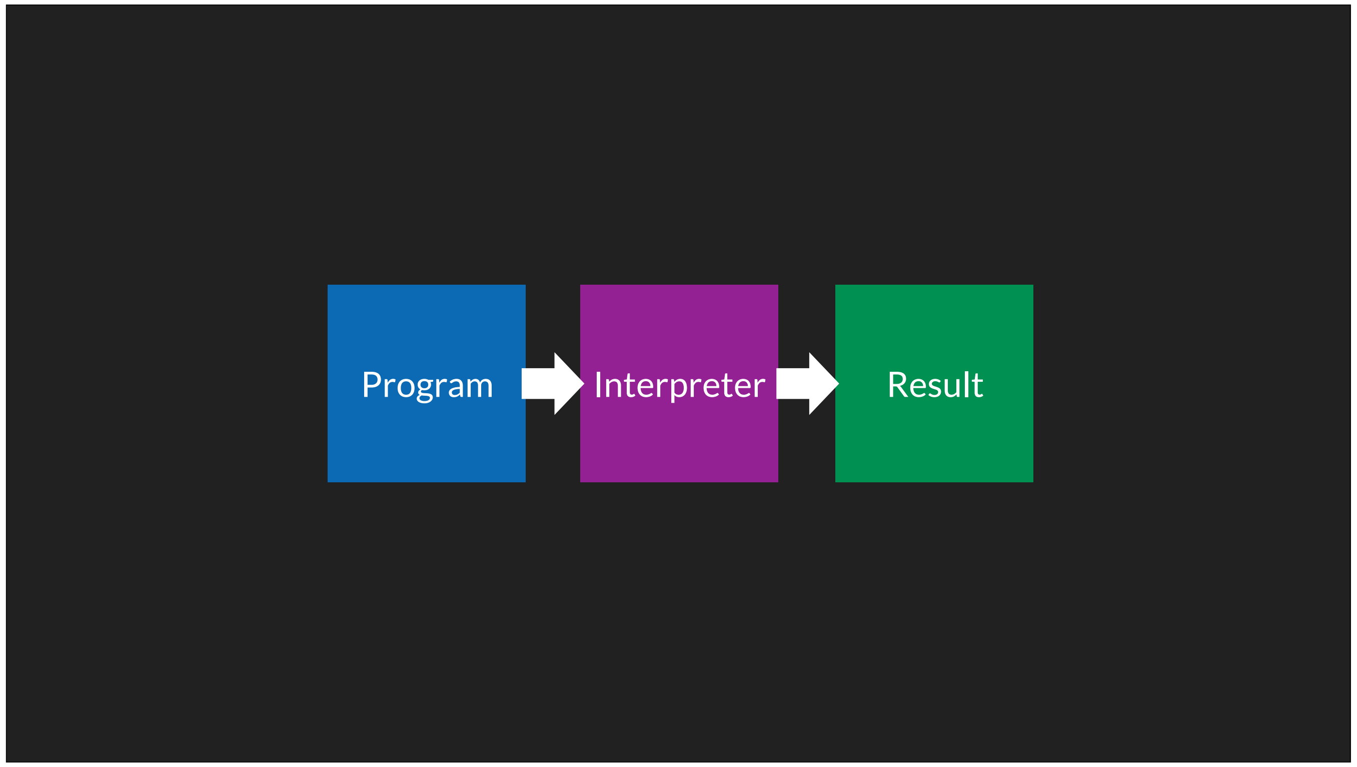
Tagless final provides an extremely elegant encoding for combining DSLs and abstracting over effects.

It’s similar in many ways to the Free monad, and uses significantly less boilerplate.

Summary

And that completes our coverage of Church encoding, which completes the main content for this talk.

To finish off let’s recap…

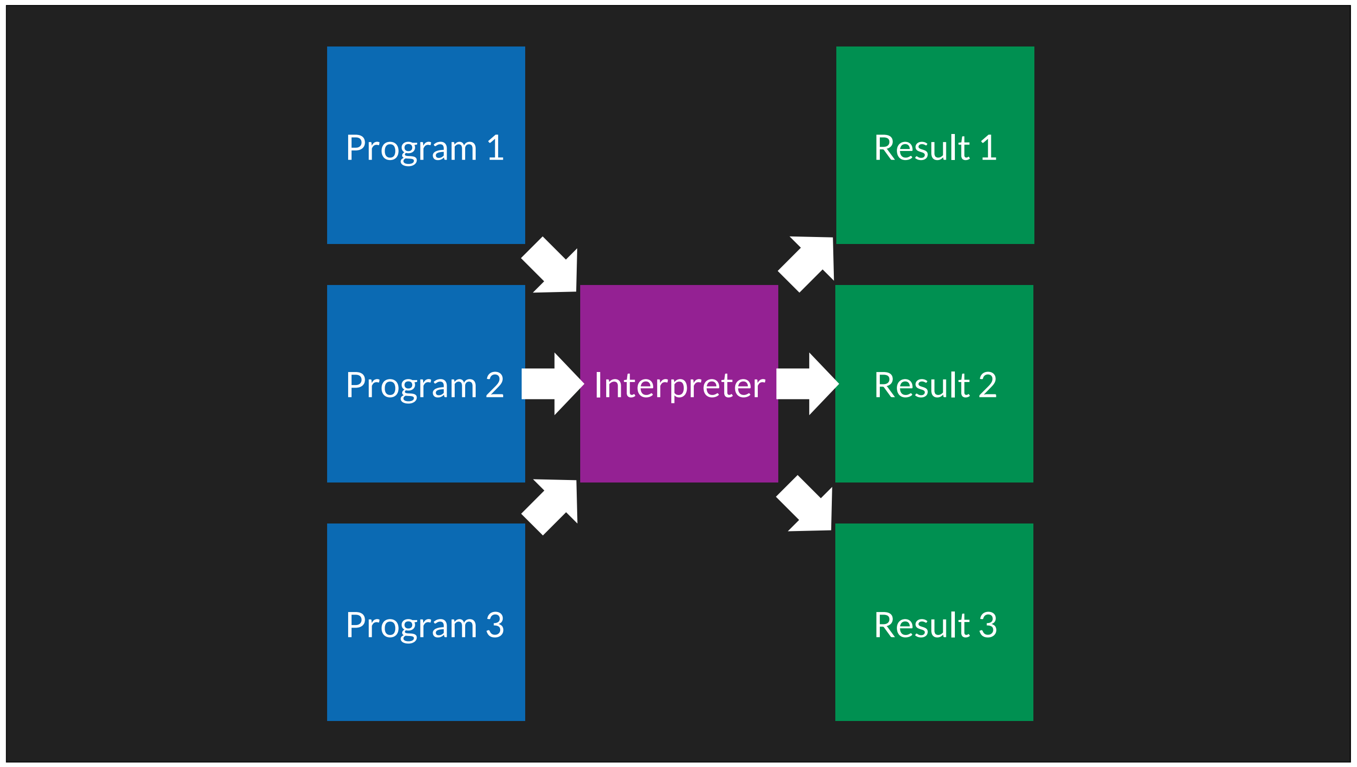


We’ve been discussing various ways of implementing the “interpreter pattern” in Scala.

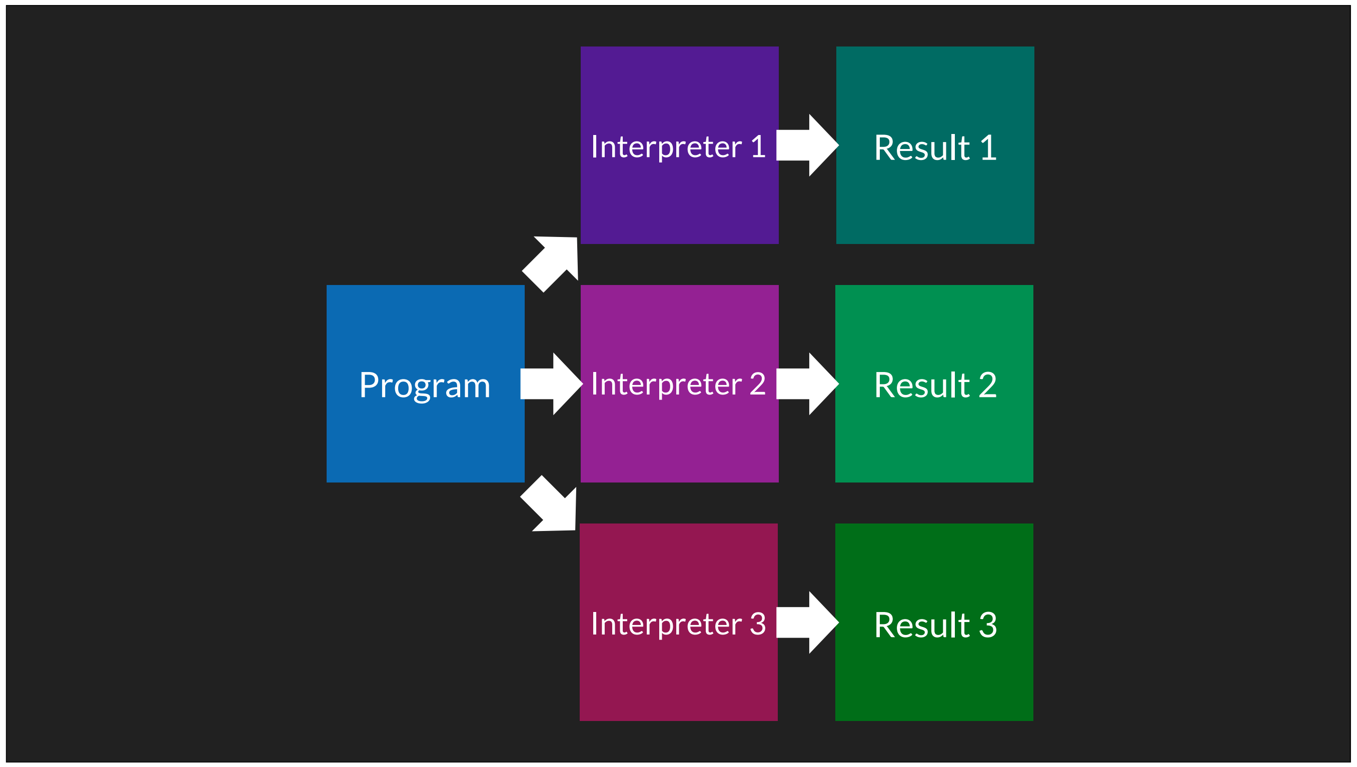
This pattern allows us to separate our code into two halves:

- “programs”, written in some high-level DSL, specify the structure of a computation;
- “interpreters” provide the implementation details required to run programs and compute a result.

The main benefits of the interpreter pattern are modularity and reuse…

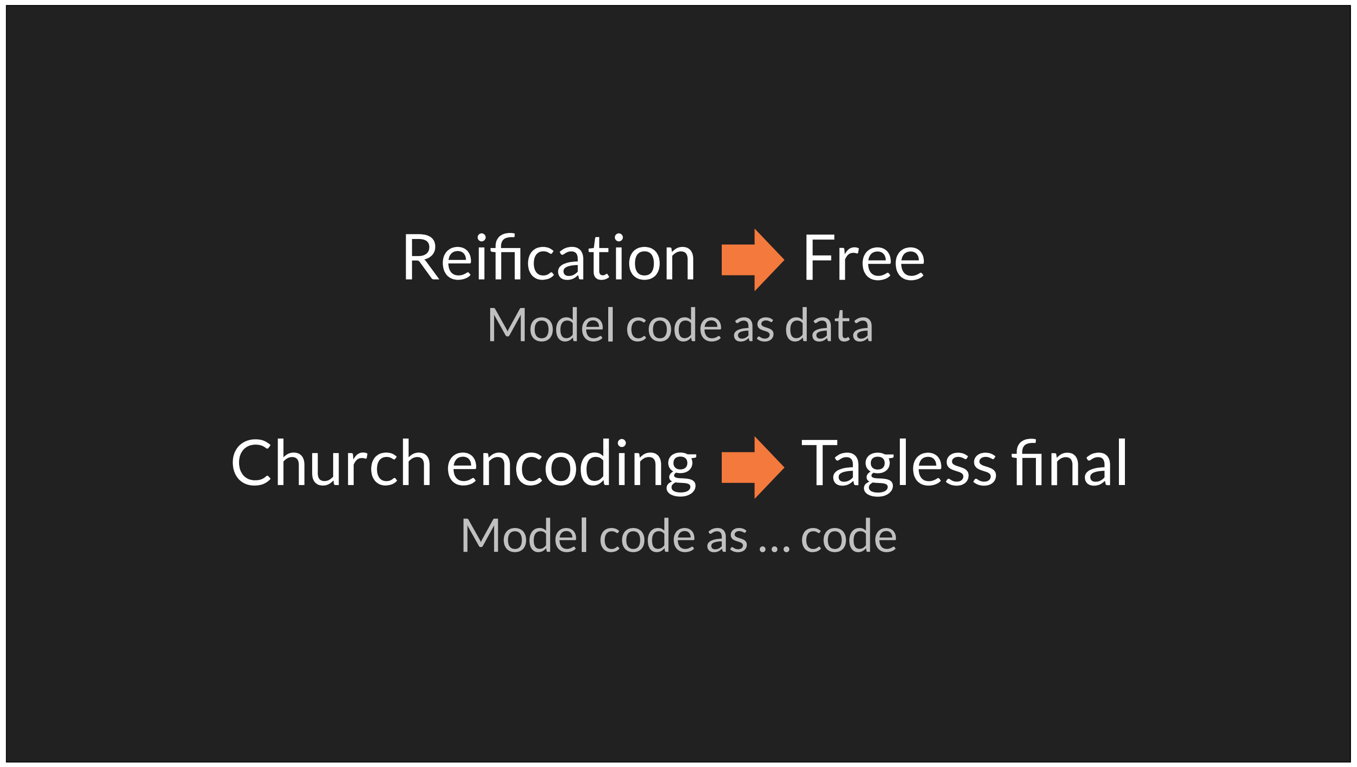


…reusing the same interpreter to run different programs…



…and implementing different interpreters to run programs in different ways:

- either to produce different results from the same code;
- or to abstract over different effects.



Reification ➡ Free

Model code as data

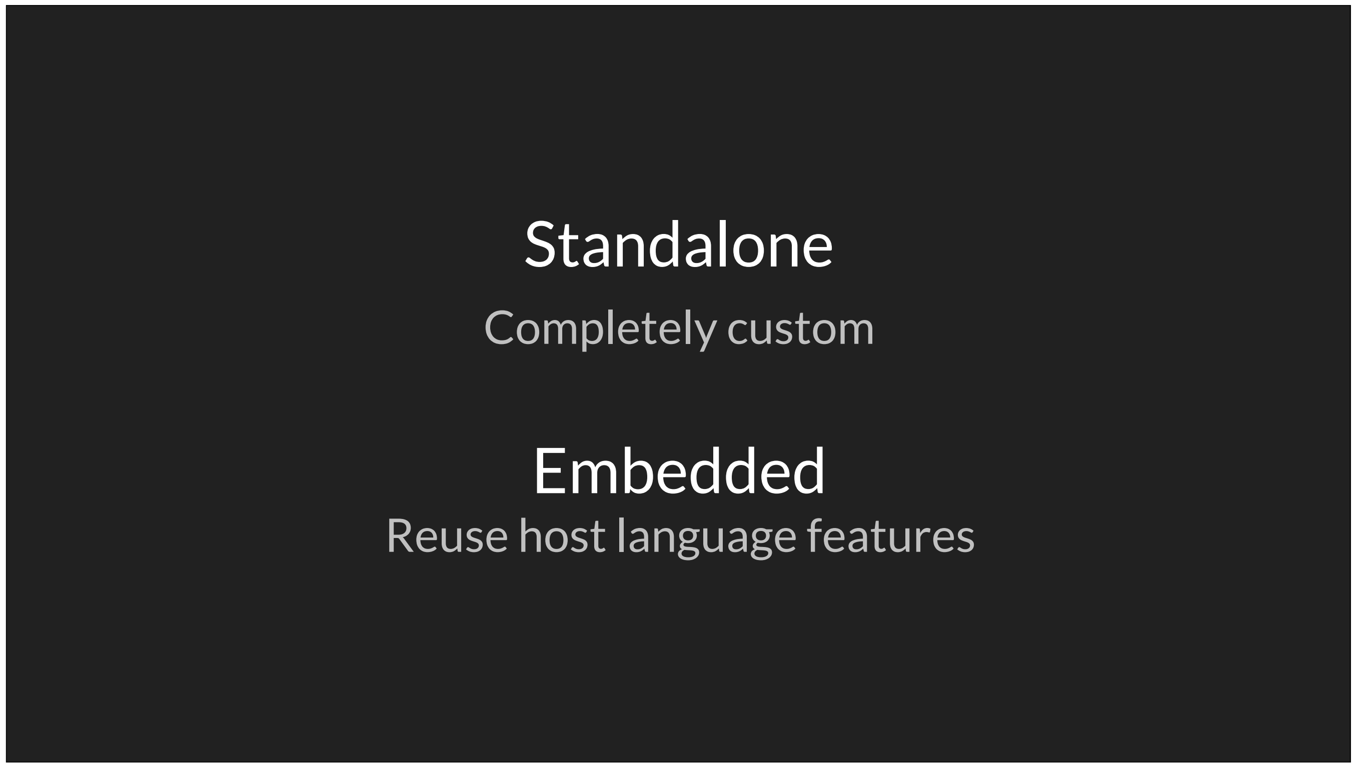
Church encoding ➡ Tagless final

Model code as … code

There are two main approaches to implementing the interpreter pattern:

- “reification”, where we represent programs as values and interpreters as functions, and;
- “Church encoding”, where we represent DSLs as sets of abstract methods, programs as expressions written in terms of those methods, and interpreters as concrete implementations of those methods.

We discussed various approaches to each encoding including the Free monad, which is a form of reification, and Tagless final, which is a form of church encoding.



Standalone

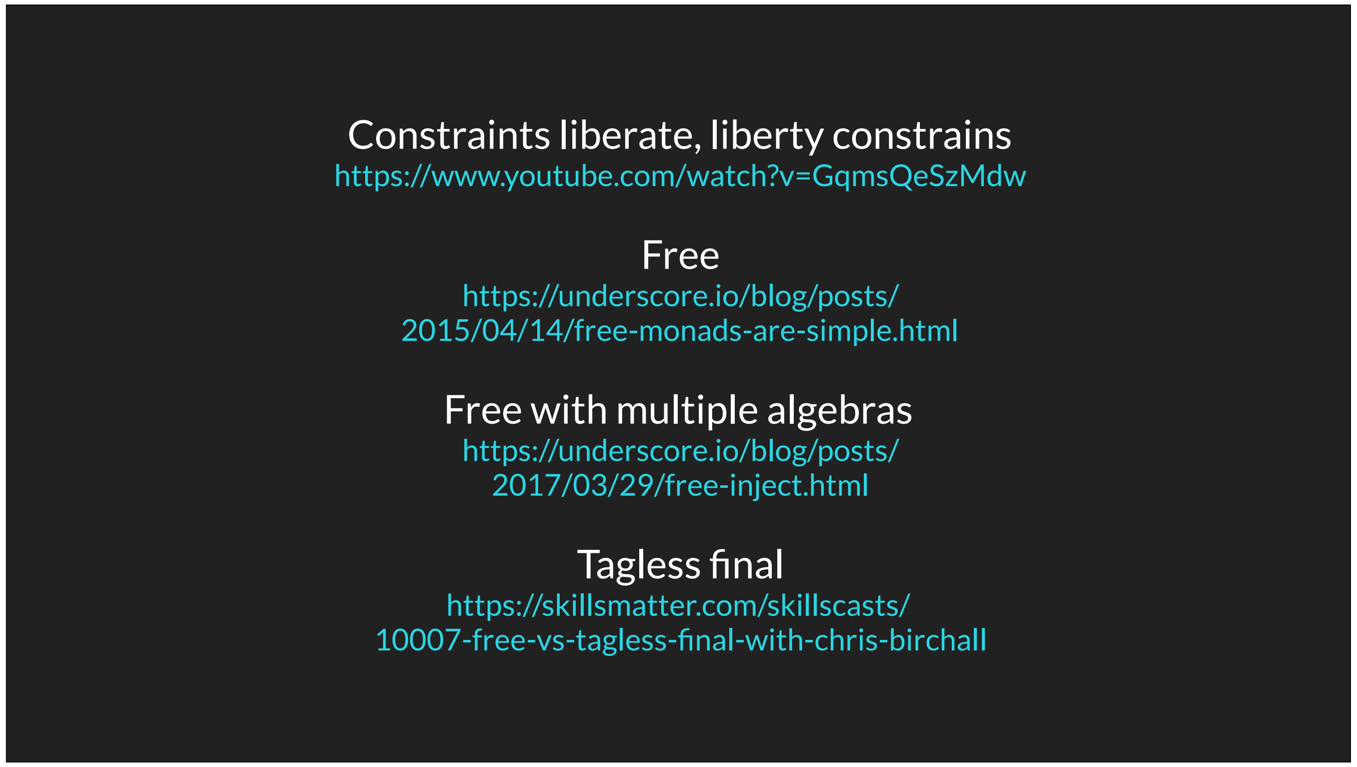
Completely custom

Embedded

Reuse host language features

Along the way we discussed levels of embedding of the DSL in the host language.

Broadly speaking, the more we re-use the host language (so called “deeper embedding”, the quicker it is to create a powerful DSL). However, the more general our DSL becomes, the less power we have to inspect and modify programs.



Constraints liberate, liberty constrains
https://www.youtube.com/watch?v=GqmsQeSzMdw

Free
https://underscore.io/blog/posts/2015/04/14/free-monads-are-simple.html

Free with multiple algebras
https://underscore.io/blog/posts/2017/03/29/free-inject.html

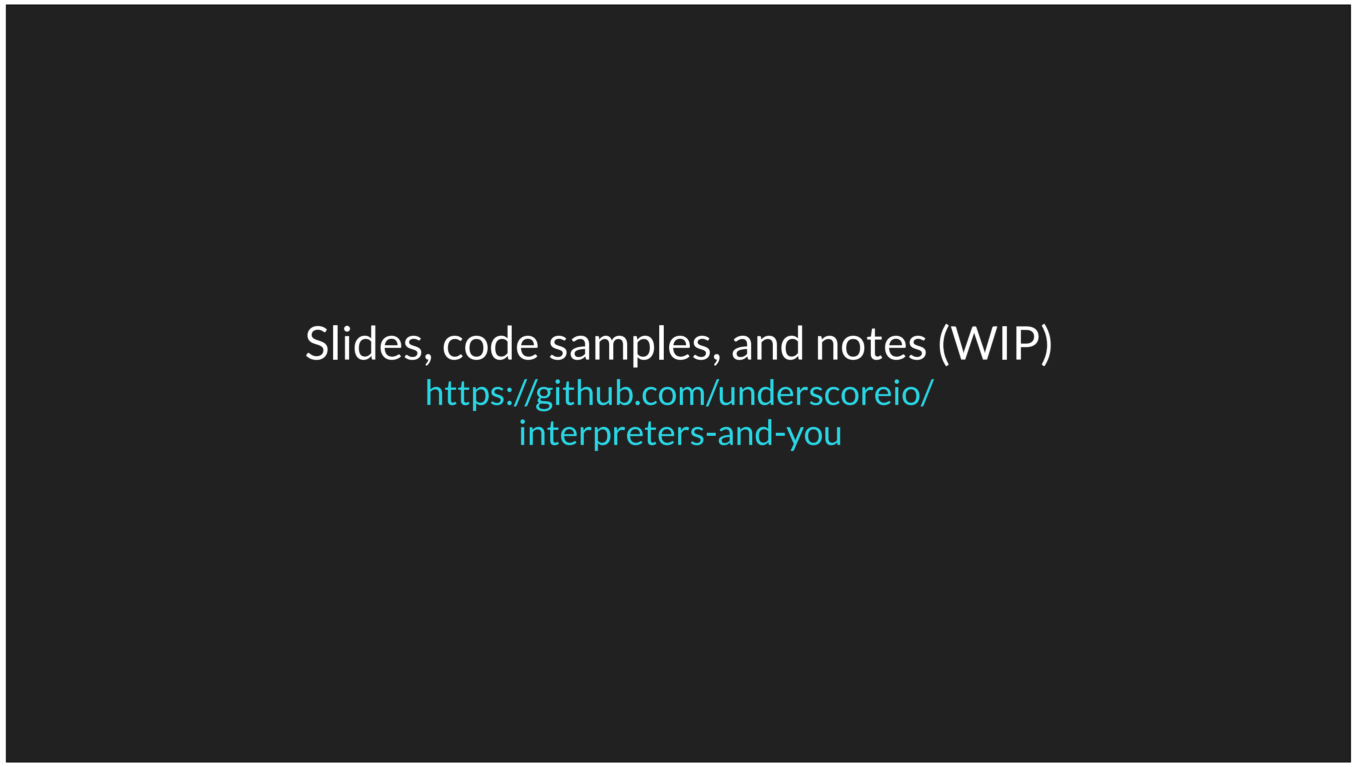
Tagless final
https://skillsmatter.com/skillscasts/10007-free-vs-tagless-final-with-chris-birchall

Here are a few bits of further reading/viewing:

- The top link is a Scala World keynote by Runar, in which he discusses this notion of constraints improving inspecability.

- The next links are Underscore blog posts about Free, with a single and multiple algebras… each uses the implementation of Free from Cats.

- The final link is a talk by Chris Birchall on tagless final encoding. In the talk he motivates tagless final and then live-codes a non-trivial web app using the technique. It's a really great watch.


Slides, code samples, and notes (WIP)

https://github.com/underscoreio/interpreters-and-you

The slides for this talk are available on Github in the Underscore organization.

There is also an SBT project containing complete implementations of each approach, and the beginnings of a write-up that may become a set of blog posts or an essay (no promises though ;)

Thank you

https://github.com/underscoreio/interpreters-and-you

Dave Gurnell, @davegurnell

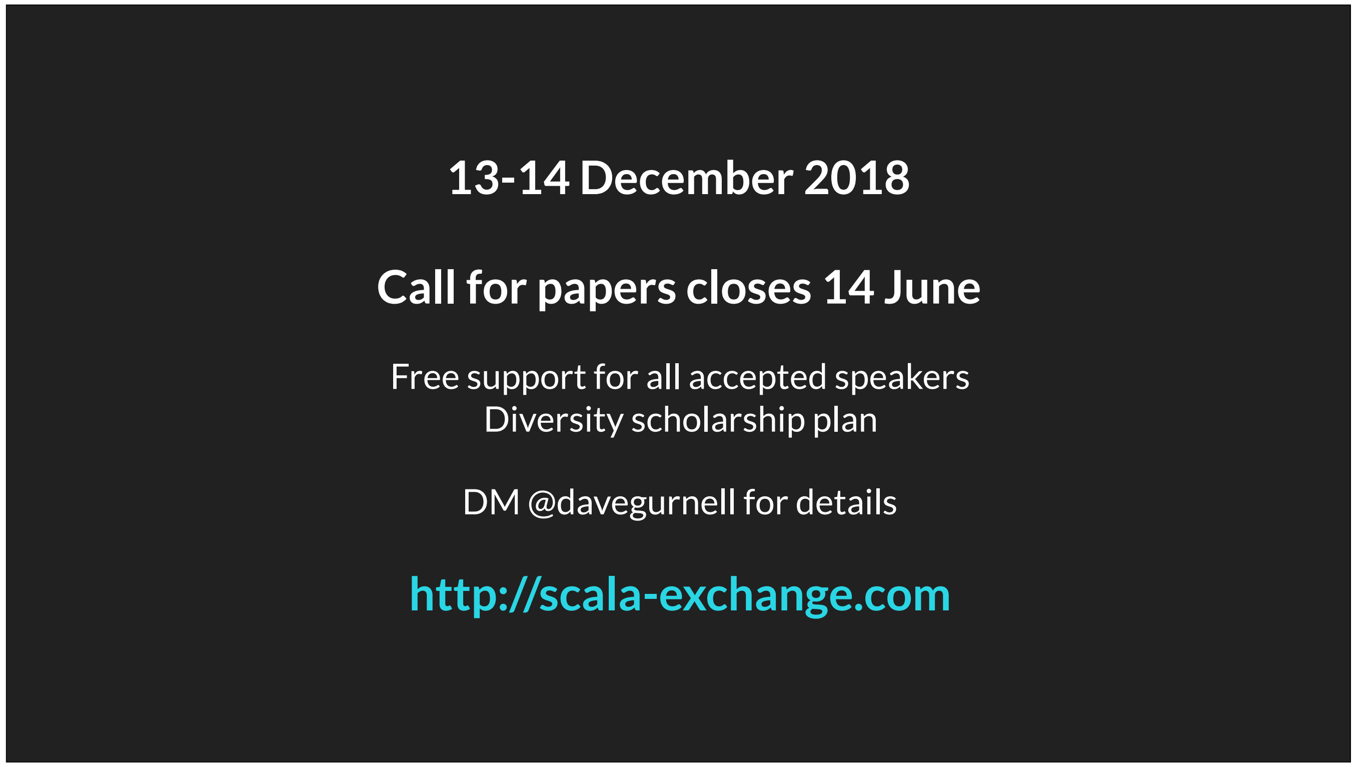
Quick promo…

While we’re here, I’d like to take a second to do a quick bit of promotion.

Scala Exchange is on 13th/14th December this year.

The CfP is currently open (it closes mid June).

We'd love to hear talk ideas from you, regardless of your level of experience with tech speaking.



13-14 December 2018

Call for papers closes 14 June

Free support for all accepted speakers
Diversity scholarship plan

DM @davegurnell for details

http://scala-exchange.com

ScalaX is a great conference that provides lots of support for new speakers:

- shepherding of abstracts to help you write up your ideas;
- speaker training for anyone accepted via the CfP;
- diversity scholarships and bursaries for speakers.

If you’d like to know more, please get in touch! DM @davegurnell on the Tweeters.

Thanks!