Building Interpreters in Scala

Scala Exchange 2015

Noel Welsh, @noelwelsh

underscore

Introduction

1

What are interpreters?

2

Why do we care?

What are we going to cover?

1

What are Interpreters?

Separate representation from computation

Example: a Scala program

1 + 1 `Example.scala`

2

Evaluation

`0x00000002`

(in RAM)

1 + 1

↓

2

Representation

1 + 1 Representation

Interpretation

2

$1 + 1$	Representation
↓	Interpretation
2	Result

Note: We’re not making a distinction between interpreters and compilers

Representation sometimes called a abstract syntax tree (AST)

Example: a simple calculator

Abstract syntax tree is an algebraic data type

```scala
sealed trait Expr
final case class Plus(left: Expr, right: Expr)
 extends Expr
final case class Minus(left: Expr, right: Expr)
 extends Expr
final case class Multiply(left: Expr, right: Expr)
 extends Expr
final case class Divide(left: Expr, right: Expr)
 extends Expr
final case class Value(get: Double) extends Expr
```

An expression is a value built using the AST

```
Add(Value(1), Value(1))
```

An interpreter is structural recursion

```scala
sealed trait Expression {
  def eval: Double =
    this match {
      case Plus(l, r)     => l.eval + r.eval
      case Minus(l, r)    => l.eval - r.eval
      case Multiply(l, r) => l.eval * r.eval
      case Divide(l, r)   => l.eval / r.eval
      case Value(v)       => v
    }
}
```

All our interpreters will have the same general structure

2

Why Interpreters?

Interpreters give us total control of a program's semantics

This can make hard things easy

Example: Feature Gating

Problem: you want to gradually roll out features to selected cohorts of users

Solution: pepper the code with conditionals

No!

~~Solution: pepper the code with conditionals~~

Solution: define a language of feature gates

Can change in real-time
(no deployment required)

Can statically check before going live

Can optimise logic

Business users can read and understand

Flexible Feature Control at Instagram

https://engineering.instagram.com/posts/496049610561948/flexible-feature-control-at-instagram/

Example: Big Data

Problem: data is so

big!

Describe computation
as directed acyclic graph

Compile to run across a cluster and/or GPU

Spark, Tensor Flow, etc.

Example: Service Orchestration

Problem: Lots of network traffic in service oriented architecture.

Describe service calls in embedded DSL

Automatically batch
and cache calls

Haxl (Facebook), Stitch (Twitter)

What are we going to cover?

Implementation techniques for interpreters

Object language is the one we’re implementing

Host language (Scala) is what we’re writing it in

Goal: Reuse host language as far as possible

Topics

I

Untyped object language

I

Algebraic data types and structural recursion

II

Add functions and bindings

II

Higher-order abstract syntax

Add types

III

Generalised algebraic data types

Add composition of interpreters

IV

Free monads

Untyped Interpreters

Implement an AST and interpreter for simple arithmetic expressions

Yes, this is the example we have already covered

See `Arithmetic.scala` in the `untyped` project

Let’s introduce strings in addition to numbers

We need to make some changes

Operations must check the tags of values

Evaluation can result in an error

Complete the interpreter in `NumbersAndStrings.scala` in the `untyped` project

II

Functions and bindings

Let’s add functions

This requires **bindings**

A binding is an association between a name and a value

```
val foo = 1
((x) => …)(2)
```

And references—where we refer to a name

```
foo + 2
(x) => x + 1
```

The only sane implementation is lexical scoping

Implementing this by hand is involved

Alternatively, reuse Scala bindings

Alternatively, reuse Scala bindings

This idea is called
higher-order abstract syntax

```
val one = number(1.0)
Plus(one, one)
```

Represent functions as … functions

and function application is function application

Complete the interpreter in `Hoas.scala` in the `untyped` project

Using HOAS makes our interpreter simpler

by reusing host language's implementation

but means we must
accept the host language's
binding semantics

III

Type Checking

Let’s add types

We could implement a type checking algorithm but let’s reuse Scala’s type checking

We’ll implement simple types, meaning no generics

```scala
// A is the type of the value
// the expression evaluates
// to
sealed trait Expr[A]
```

No `Value` type needed. Values represented directly

Our interpreter can work with any Scala data type

Literals evaluate to themselves

```
final case class Literal[A](get: A)
extends Expr[A]
```

Function application

```
final case class Apply[A,B](f: Expr[A
=> B], arg: Expr[A]) extends Expr[B]
```

Conditionals require both branches have same type

```
final case class If[A](cond:
Expr[Boolean], t: Expr[A], f:
Expr[A]) extends Expr[A]
```

Functions

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

This is called a generalised algebraic data type (GADT)

A GADT occurs when we declare an algebraic data type with a type variable

```scala
sealed trait Expr[A]
```

And cases in the ADT instantiate the variable with a “complex” type

```
case class Function[A,B](…)
extends Expr[A => B]
```

Scala’s support for GADTs is … amusing? … infuriating?

You decide!

Complete the interpreter in `Hoas.scala` in the `gadt` project

IV

Interpreter Composition

Our current interpreters are not compositional

The AST is fixed

The interpreter is fixed

We’ll now look at one way to overcome this problem

The free monad

Bonus: the free applicative

Also reuse Scala for most of our language

What The Free?

A free functor is left adjoint to a forgetful functor

:D

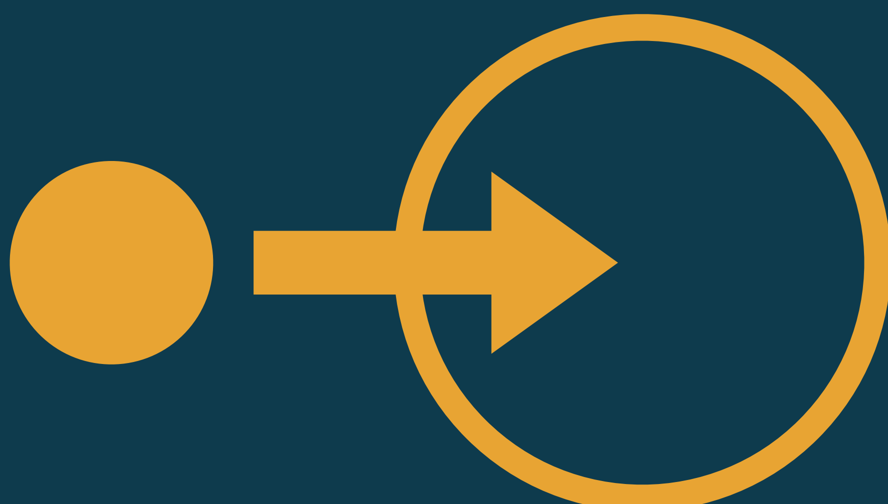
Let’s start again

Monads

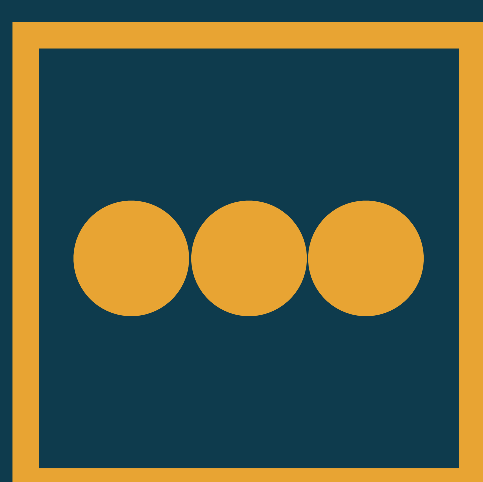
Option[A], List[A], Future[A], ...

A value in a context

Option[A]

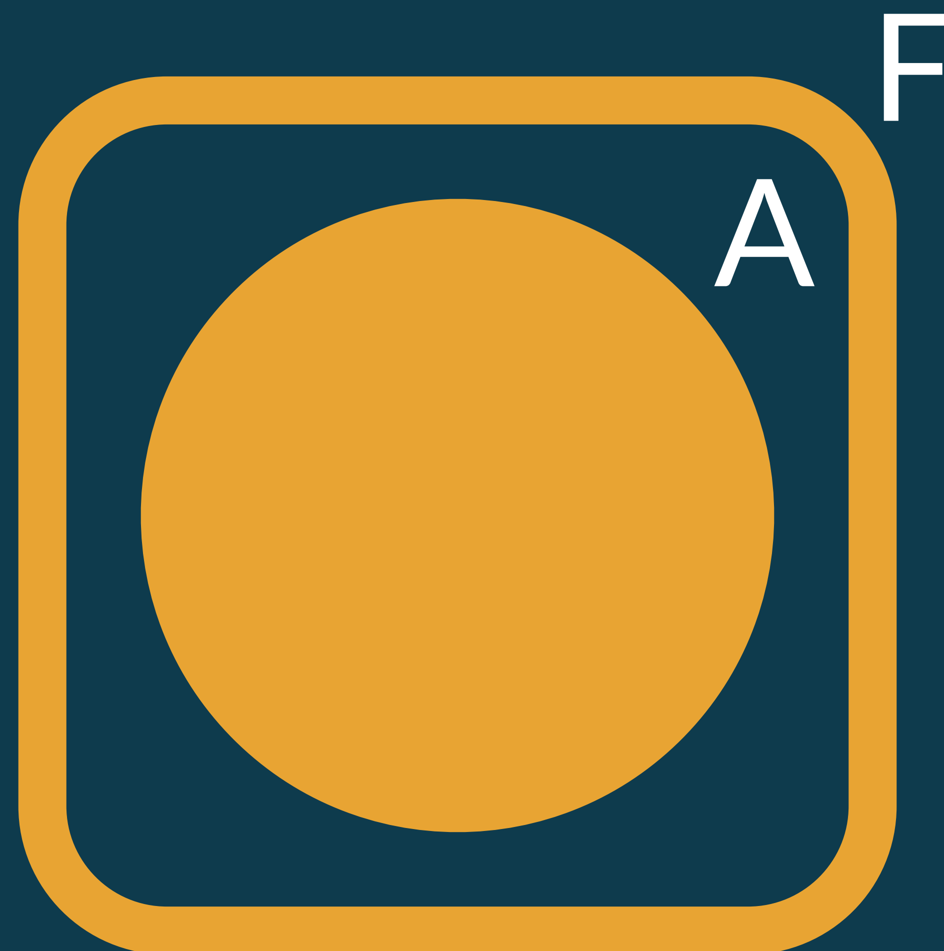


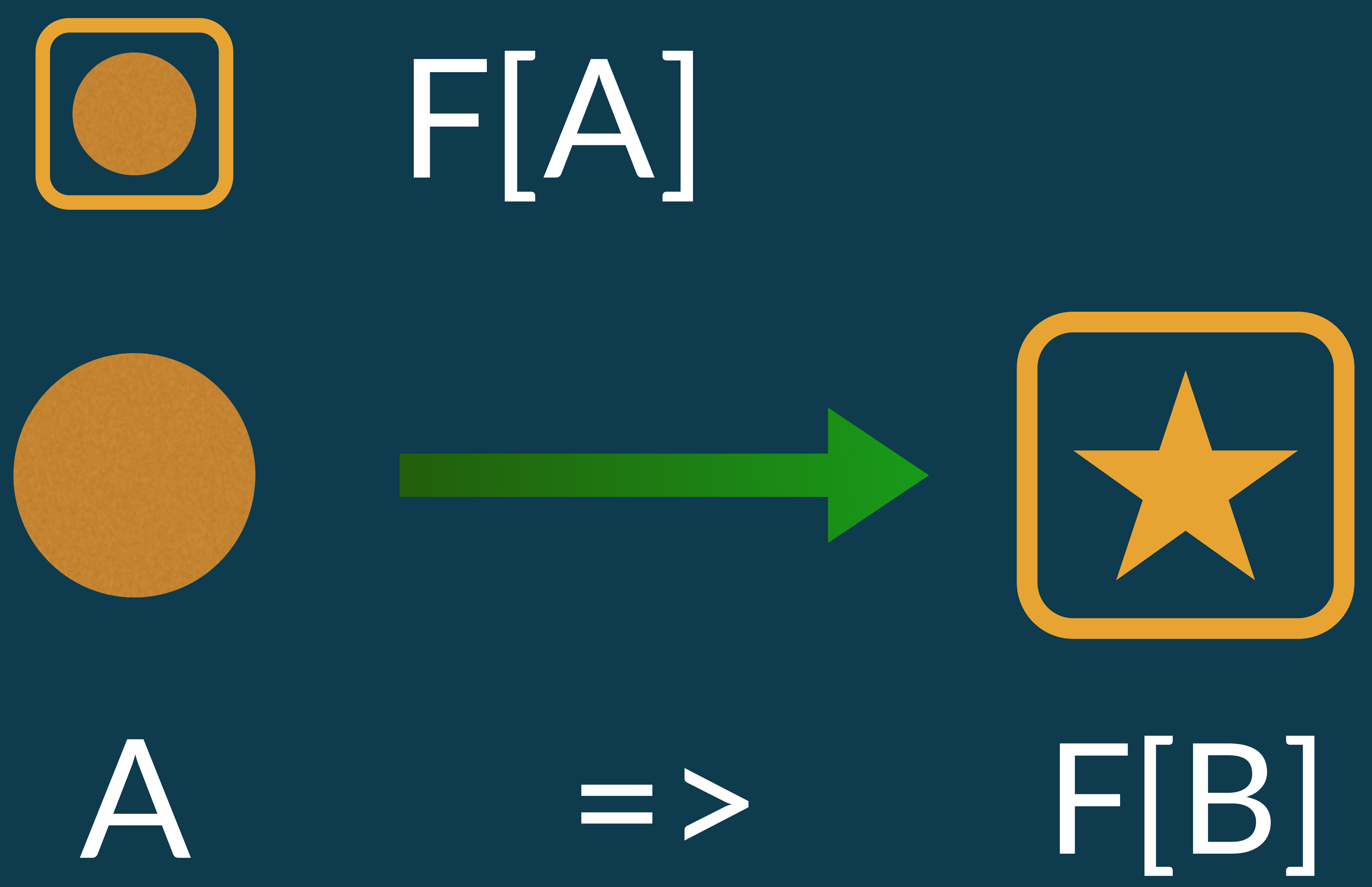
Future[A]

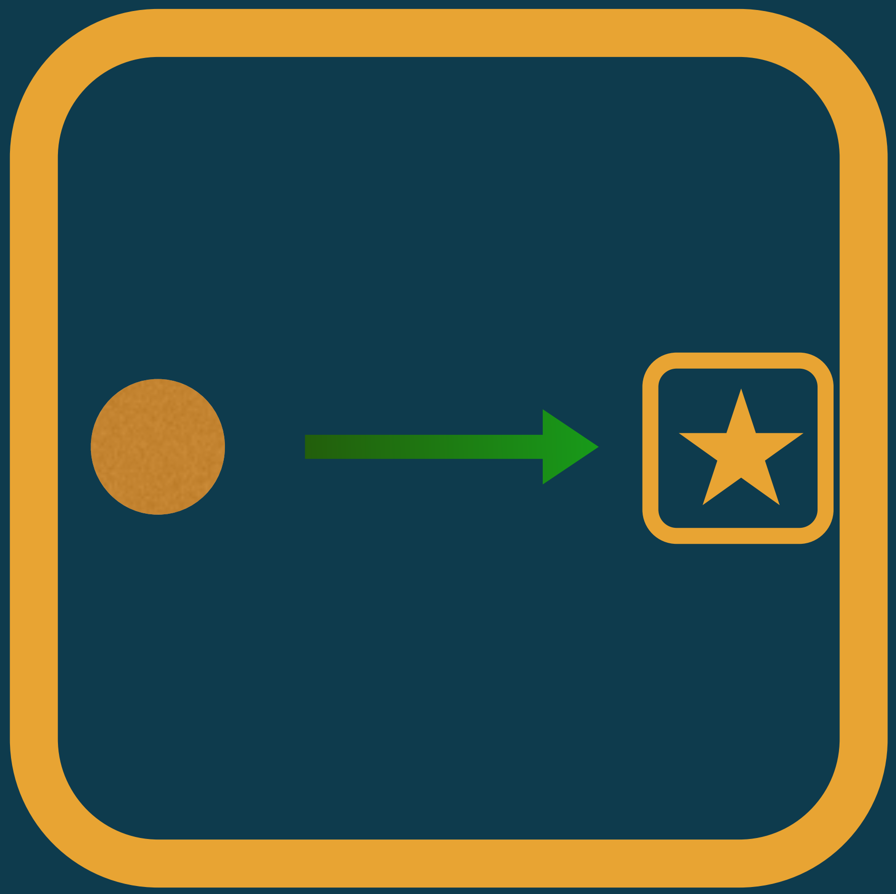


List[A]

F[A]

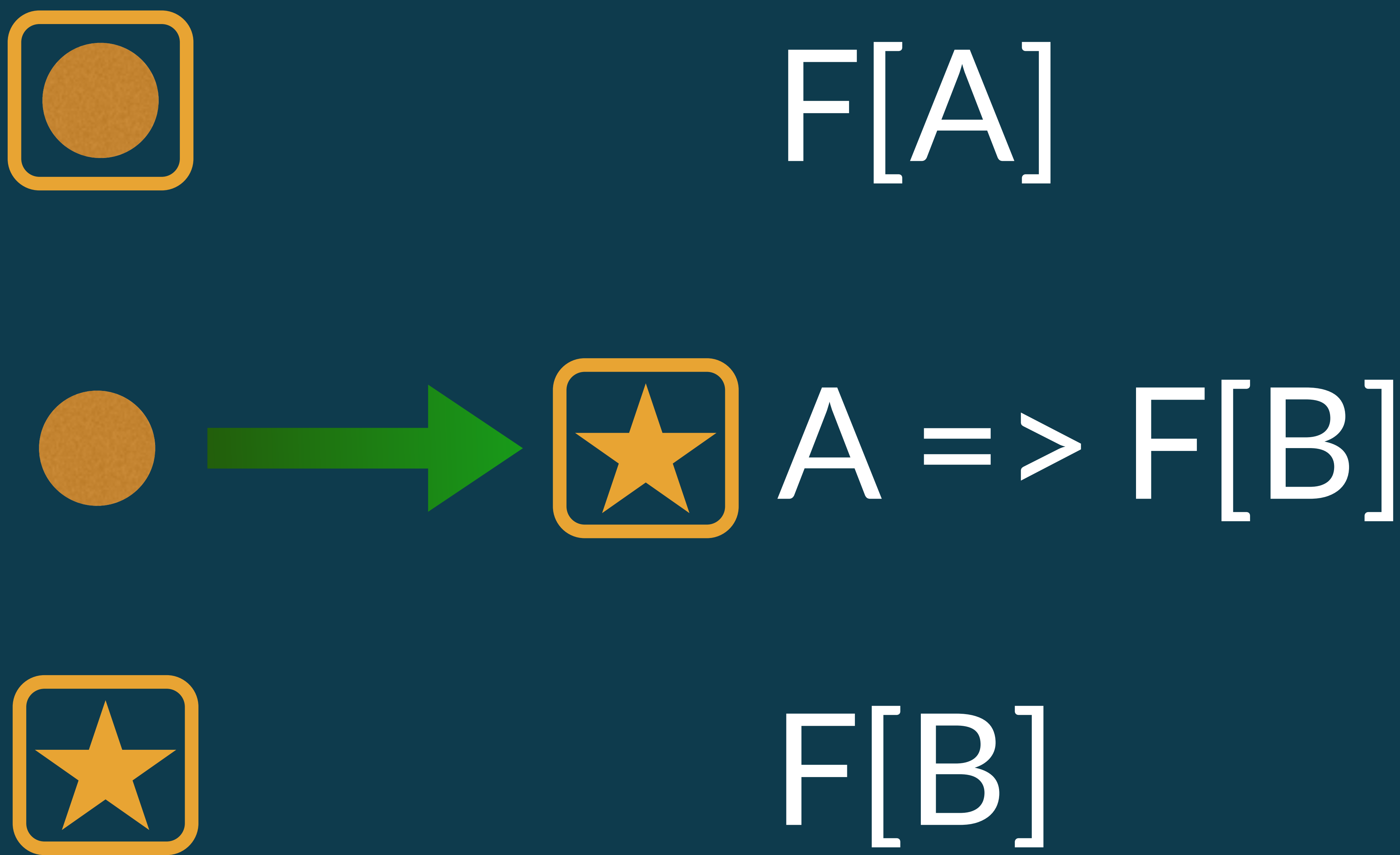

A
F


F[A]
A
=>
F[B]



F

B

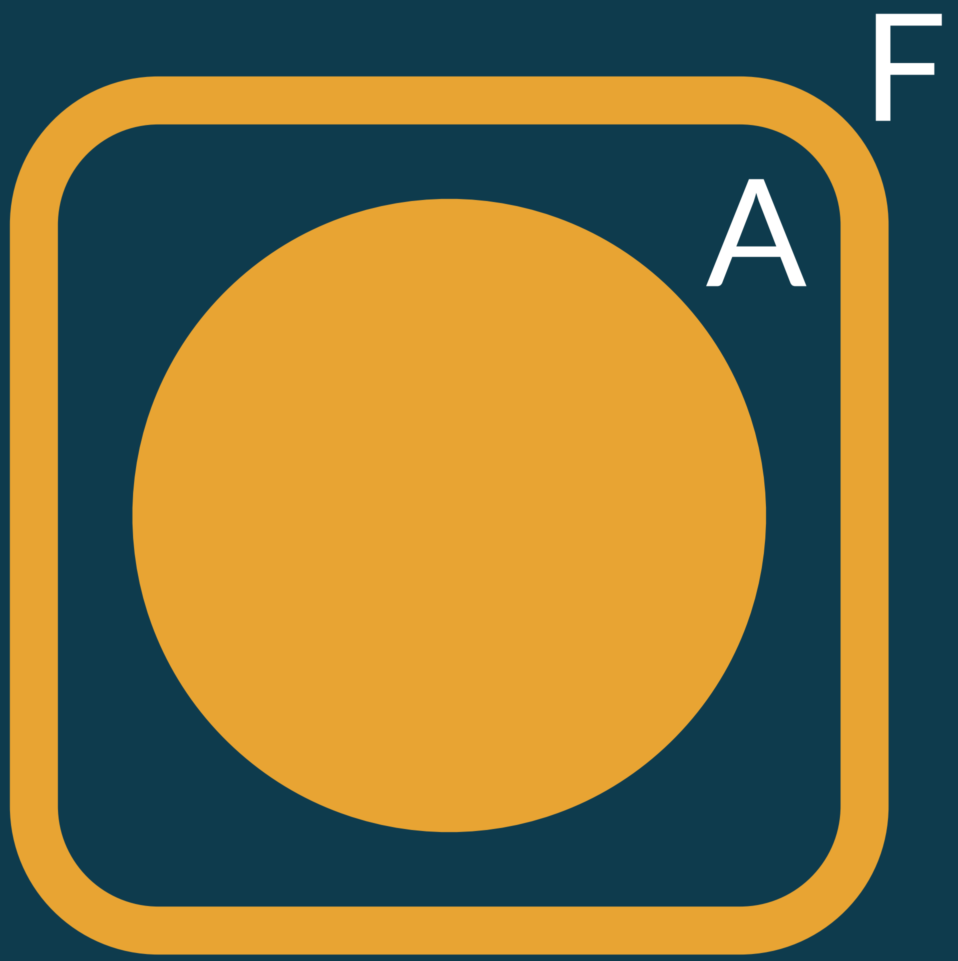

F[A]
A => F[B]
F[B]

FlatMap

Monads allow us to transform a value and its context

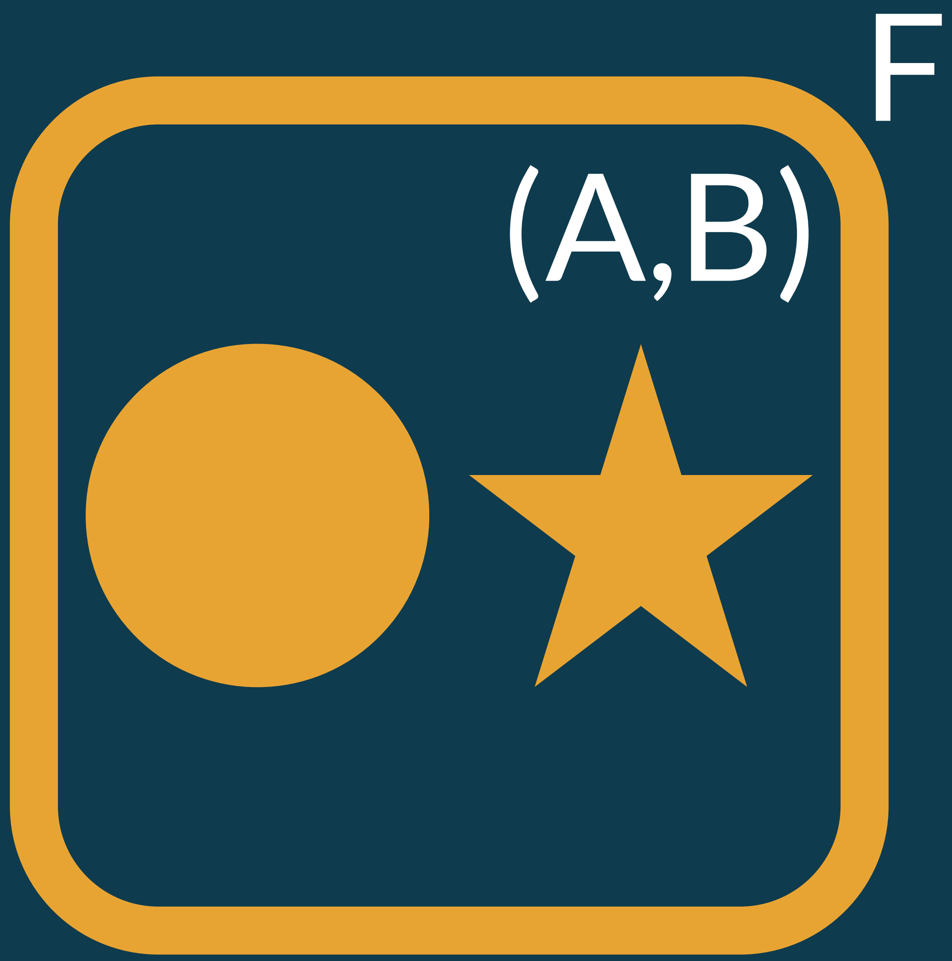
Can express any control flow with a monad

Applicatives


F
A

F

B


F
(A,B)

$F[A]$

$F[B]$

$F[(A,B)]$

Join

`|@|` collapses nested tuples

```
F[(A,B)] |@| F[C] =
F[(A,B,C)]
```

A bit of indirection (the `ApplyBuilder`) makes this work

Free Structures

Service orchestration example

Applicative is parallel composition of requests

Monad is sequential composition

Separate the structure of the computation from the process that gives it meaning

Separate the structure of the computation from the process that gives it meaning

Represent monadic / applicative operations as AST

```scala
sealed trait Monad[A]
case class FlatMap[A,B](f: A => Monad[B])
 extends Monad[B]
case class Point[A](a: A)
 extends Monad[A]
```

This is almost the free monad

```
Free[F[_],A]
```

F is a type constructor (like Expr)

`A` is the type held by the type constructor

Free

F

A

The free applicative is the same idea

Complete the interpreter in `Orchestration.scala` in the `free` project

A note about code samples: to fit the slides I made a few changes to my normal style. Make your case classes final. Favour full names over abbreviations.