

# Chapter 10: Free type constructions

Sergei Winitzki

Academy by the Bay

2018-11-22

# The interpreter pattern I. Expression trees

Main idea: Represent a program as a data structure, run it later

- Example: a simple DSL for complex numbers

```
val a = "1+2*i".toComplex      Conj(  
val b = a * "3-4*i".toComplex  Mul(  
b.conj                        Str("1+2*i"), Str("3-4*i"))  
                                ))
```

- *Unevaluated* operations `Str`, `Mul`, `Conj` are defined as case classes:

```
sealed trait Prg  
case class Str(s: String) extends Prg  
case class Mul(p1: Prg, p2: Prg) extends Prg  
case class Conj(p: Prg) extends Prg
```

- An *interpreter* will “run” the program and return a complex number

```
def run(prg: Prg): (Double, Double) = ...
```

- Benefits: programs are data, can compose & transform before running
- Shortcomings: this DSL works only with simple expressions
  - ▶ Cannot represent variable binding and conditional computations
  - ▶ Cannot use any non-DSL code (e.g. a numerical algorithms library)

# The interpreter pattern II. Variable binding

## A DSL with variable binding and conditional computations

- Example: imperative API for reading and writing files
  - ▶ Need to bind a *non-DSL variable* to a value computed by DSL
  - ▶ Later, need to use that non-DSL variable in DSL expressions
  - ▶ The rest of the DSL program is a (Scala) function of that variable

```
val p = path("/file")
val str: String = read(p)
if (str.nonEmpty)
  read(path(str))
else "Error: empty path"
```

```
Bind(
  Read(Path(Literal("/file"))),
  { str => // read value 'str'
    if (str.nonEmpty)
      Read(Path(Literal(str)))
    else Literal("Error: empty path")
  })
```

- Unevaluated operations are implemented via case classes:

```
sealed trait Prg
case class Bind(p: Prg, f: String => Prg) extends Prg
case class Literal(s: String) extends Prg
case class Path(s: Prg) extends Prg
case class Read(p: Prg) extends Prg
```

- Interpreter: `def run(prg: Prg): String = ...`

# The interpreter pattern III. Type safety

- So far, the DSL has no type safety: every value is a `Prg`
  - ▶ We want to avoid errors, e.g. `Read(Read(...))` should not compile
- Let `Prg[A]` denote a DSL program returning value of type `A` *when run*:

```
sealed trait Prg[A]
case class Bind(p: Prg[String], f: String ⇒ Prg[String])
  extends Prg[String]
case class Literal(s: String) extends Prg[String]
case class Path(s: Prg[String]) extends Prg[nio.file.Path]
case class Read(p: Prg[nio.file.Path]) extends Prg[String]
```

- Interpreter: `def run(prg: Prg[String]): String = ...`
- Our example DSL program is type-safe now:

```
val prg: Prg[String] = Bind(
  Read(Path(Literal("/file"))),
  { str: String ⇒
    if (str.nonEmpty)
      Read(Path(Literal(str)))
    else Literal("Error: empty path")
  })
```

# The interpreter pattern IV. Cleaning up the DSL

Our DSL so far:

```
sealed trait Prg[A]
case class Bind(p: Prg[String], f: String ⇒ Prg[String])
  extends Prg[String]
case class Literal(s: String) extends Prg[String]
case class Path(s: Prg[String]) extends Prg[nio.file.Path]
case class Read(p: Prg[nio.file.Path]) extends Prg[String]
```

Problems with this DSL:

- Cannot use `Read(p: nio.file.Path)`, only `Read(p: Prg[nio.file.Path])`
- Cannot bind variables or return values other than `String`

To fix these problems, make `Literal` a fully parameterized operation and replace `Prg[A]` by `A` in case class arguments

```
sealed trait Prg[A]
case class Bind[A, B](p: Prg[A], f: A ⇒ Prg[B]) extends Prg[B]
case class Literal[A](a: A) extends Prg[A]
case class Path(s: String) extends Prg[nio.file.Path]
case class Read(p: nio.file.Path) extends Prg[String]
```

- The type signatures of `Bind` and `Literal` are like `flatMap` and `pure`

# The interpreter pattern V. Define Monad-like methods

- We can actually define the methods `map`, `flatMap`, `pure`:

```
sealed trait Prg[A] {  
  def flatMap[B](f: A ⇒ Prg[B]): Prg[B] = Bind(this, f)  
  def map[B](f: A ⇒ B): Prg[B] = flatMap(this, f andThen Prg.pure)  
}  
object Prg { def pure[A](a: A): Prg[A] = Literal(a) }
```

- These methods don't run anything, only create unevaluated structures
- DSL programs can now be written as functor blocks and composed:

```
def readPath(p: String): Prg[String] = for {  
  path ← Path(p)  
  str  ← Read(path)  
} yield str
```

```
val prg: Prg[String] = for {  
  str ← readPath("/file")  
  result ← if (str.nonEmpty)  
    readPath(str)  
    else Prg.pure("Error: empty path")  
} yield result
```

- Interpreter: `def run[A](prg: Prg[A]): A = ...`

# The interpreter pattern VI. Refactoring to an abstract DSL

- Write a DSL for complex numbers in a similar way:

```
sealed trait Prg[A] { def flatMap ... } // no code changes
case class Bind[A, B](p: Prg[A], f: A⇒Prg[B]) extends Prg[B]
case class Literal[A](a: A) extends Prg[A]
type Complex = (Double, Double) // custom code starts here
case class Str(s: String) extends Prg[Complex]
case class Mul(c1: Complex, C2: Complex) extends Prg[Complex]
case class Conj(c: Complex) extends Prg[Complex]
```

- Refactor this DSL to separate common code from custom code:

```
sealed trait DSL[F[_], A] { def flatMap ... } // no code changes
type Prg[A] = DSL[F, A] // just for convenience
case class Bind[A, B](p: Prg[A], f: A⇒Prg[B]) extends Prg[B]
case class Literal[A](a: A) extends Prg[A]
case class Ops[A](f: F[A]) extends Prg[A] // custom operations here
```

- Interpreter is parameterized by a “value extractor”

$$\text{Ex}^F \equiv \forall A. (F^A \Rightarrow A)$$

```
def run[F[_], A](ex: Ex[F])(prg: DSL[F, A]): A = ...
```

## The interpreter pattern VII. Handling errors

- To handle errors, we want to evaluate `DSL[F[_], A]` to `Either[Err, A]`
- Suppose we have a value extractor of type  $\text{Ex}^F \equiv \forall A. (F^A \Rightarrow \text{Err} + A)$
- The code of the interpreter is almost unchanged:

```
def run[F[_], A](extract: Ex[F])(prg: DSL[F, A]): Either[Err, A] =  
  prg match {  
    case b: Bind[F, _, A]  $\Rightarrow$  b match { case Bind(p, f)  $\Rightarrow$   
      run(extract)(p).flatMap(f andThen run(extract))  
    }    // Here, the .flatMap is from Either.  
    case Literal(a)  $\Rightarrow$  Right(a) // pure: A  $\Rightarrow$  Err + A  
    case Ops(f)  $\Rightarrow$  extract(f)  
  }
```

- The code of `run` only uses `flatMap` and `pure` from `Either`
- We can generalize to any other monad  $M^A$  instead of `Either[Err, A]`

The resulting construction:

- Start with an “operations type constructor”  $F^A$  (often not a functor)
- Use  $\text{DSL}^{F,A}$  and interpreter  $\text{run}^{M,A} : (\forall X. F^X \Rightarrow M^X) \Rightarrow \text{DSL}^{F,A} \Rightarrow M^A$
- Create a DSL program  $\text{prg} : \text{DSL}^{F,A}$  and an extractor  $\text{ex}^X : F^X \Rightarrow M^X$
- Run the program with the extractor: `run(ex)(prg)`; get a value  $M^A$



## The interpreter pattern VIII. Monadic DSLs: summary

- Begin with a number of operations, which are typically functions of fixed known types such as  $A_1 \Rightarrow B_1$ ,  $A_2 \Rightarrow B_2$  etc.
- Define a type constructor (typically not a functor) encapsulating all the operations as case classes, with or without type parameters

```
sealed trait F[A]  
case class Op1(a1: A1) extends F[B1]  
case class Op2(a1: A2) extends F[B2]
```

- Use `DSL[F,A]` with this `F` to write monadic DSL programs `prg: DSL[F,A]`
- Choose a target monad `M[A]` and implement an extractor `ex: F[A]  $\Rightarrow$  M[A]`
- Run the program with the extractor, `val res: M[A] = run(ex)(prg)`

Further directions (out of scope for this chapter):

- May choose another monad `N[A]` and use interpreter `M[A]  $\Rightarrow$  N[A]`
  - ▶ E.g. transform into another monadic DSL to optimize, test, etc.
- Since `DSL[F,A]` has a monad API, we can use monad transformers on it
- Can combine two or more DSLs in a disjunction: `DSLF+G+H,A`

# Monad laws for DSL programs

Monad laws hold for DSL programs only after evaluating them

- Consider the law  $\text{flm}(\text{pure}) = \text{id}$ ; both functions  $\text{DSL}^{F,A} \Rightarrow \text{DSL}^{F,A}$
- Apply both sides to some  $\text{prg} : \text{DSL}^{F,A}$  and get the new value

```
prg.flatMap(pure) == Bind(prg, a  $\Rightarrow$  Literal(a))
```

- This new value is *not equal* to `prg`, so this monad law fails!
  - ▶ Other laws fail as well because operations never reduce anything
- After interpreting this program into a target monad  $M^A$ , the law holds:

```
run(ex)(prg).flatMap((a  $\Rightarrow$  Literal(a)) andThen run(ex))  
  == run(ex)(prg).flatMap(a  $\Rightarrow$  run(ex)(Literal(a))  
  == run(ex)(prg).flatMap(a  $\Rightarrow$  pure(a))  
  == run(ex)(prg)
```

- ▶ Here we have assumed that the laws hold for  $M^A$
- ▶ All other laws also hold after interpreting into a lawful monad  $M^A$

The monad law violations are “not observable”

# Free constructions in mathematics: Example I

- Consider the Russian letter  $\mathfrak{u}$  (tsè) and the Chinese word 水 (shuǐ)
- We want to *multiply*  $\mathfrak{u}$  by 水. Multiply how?
- Say, we want an associative (but noncommutative) product of them
  - ▶ So we want to define a *semigroup* that *contains*  $\mathfrak{u}$  and 水 as elements
    - ★ while we still know nothing about  $\mathfrak{u}$  and 水
- Consider the set of all *unevaluated expressions* such as  $\mathfrak{u} \cdot \text{水} \cdot \text{水} \cdot \mathfrak{u} \cdot \text{水}$ 
  - ▶ Here  $\mathfrak{u} \cdot \text{水}$  is different from  $\text{水} \cdot \mathfrak{u}$  but  $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
- All these expressions form a **free semigroup** generated by  $\mathfrak{u}$  and 水
  - ▶ This is the most unrestricted semigroup that contains  $\mathfrak{u}$  and 水
- Example calculation:  $(\text{水} \cdot \text{水}) \cdot (\mathfrak{u} \cdot \text{水}) \cdot \mathfrak{u} = \text{水} \cdot \text{水} \cdot \mathfrak{u} \cdot \text{水} \cdot \mathfrak{u}$

How to represent this as a data type:

- **Tree encoding**: the full expression tree:  $(((\text{水}, \text{水}), (\mathfrak{u}, \text{水})), \mathfrak{u})$ 
  - ▶ Implement the operation  $a \cdot b$  as pair constructor (easy)
- **Reduced encoding**, as a “smart” structure:  $\text{List}(\text{水}, \text{水}, \mathfrak{u}, \text{水}, \mathfrak{u})$ 
  - ▶ Implement  $a \cdot b$  by concatenating the lists (more expensive)

## Free constructions in mathematics: Example II

- Want to define a product operation for  $n$ -dimensional vectors:  $\mathbf{v}_1 \otimes \mathbf{v}_2$
- The  $\otimes$  must be linear and distributive (but not commutative):

$$\mathbf{u}_1 \otimes \mathbf{v}_1 + (\mathbf{u}_2 \otimes \mathbf{v}_2 + \mathbf{u}_3 \otimes \mathbf{v}_3) = (\mathbf{u}_1 \otimes \mathbf{v}_1 + \mathbf{u}_2 \otimes \mathbf{v}_2) + \mathbf{u}_3 \otimes \mathbf{v}_3$$

$$\mathbf{u} \otimes (a_1 \mathbf{v}_1 + a_2 \mathbf{v}_2) = a_1 (\mathbf{u} \otimes \mathbf{v}_1) + a_2 (\mathbf{u} \otimes \mathbf{v}_2)$$

$$(a_1 \mathbf{v}_1 + a_2 \mathbf{v}_2) \otimes \mathbf{u} = a_1 (\mathbf{v}_1 \otimes \mathbf{u}) + a_2 (\mathbf{v}_2 \otimes \mathbf{u})$$

- ▶ We have such a product for 3-dimensional vectors only; ignore that
- Consider *unevaluated expressions* of the form  $\mathbf{u}_1 \otimes \mathbf{v}_1 + \mathbf{u}_2 \otimes \mathbf{v}_2 + \dots$ 
  - ▶ A free vector space generated by pairs of vectors
- Impose the equivalence relationships shown above
  - ▶ The result is known as the **tensor product**
- Tree encoding: full unevaluated expression tree
  - ▶ A list of any number of vector pairs  $\sum_i \mathbf{u}_i \otimes \mathbf{v}_i$
- Reduced encoding: an  $n \times n$  matrix
  - ▶ Reduced encoding requires proofs and more complex operations

# Worked example I: Free semigroup

Implement a free semigroup `FSIS` generated by two types `Int` and `String`

- A value of `FSIS` can be an `Int`; it can also be a `String`
- If `x, y` are of type `FSIS` then so is `x |+| y`

```
sealed trait FSIS // tree encoding: full expression tree
case class Wrap1(x: Int) extends FSIS
case class Wrap2(x: String) extends FSIS
case class Comb(x: FSIS, y: FSIS) extends FSIS
```

- Short type notation:  $FSIS \equiv Int + String + FSIS \times FSIS$
- For a semigroup  $S$  and given  $Int \Rightarrow S$  and  $String \Rightarrow S$ , map  $FSIS \Rightarrow S$
- Simplify and generalize this construction by setting  $Z = Int + String$ 
  - ▶ The tree encoding is  $FS^Z \equiv Z + FS^Z \times FS^Z$

```
def |+|(x: FS[Z], y: FS[Z]): FS[Z] = Comb(x, y)
def run[S: Semigroup, Z](extract: Z  $\Rightarrow$  S): FS[Z]  $\Rightarrow$  S = {
  case Wrap(z)  $\Rightarrow$  extract(z)
  case Comb(x, y)  $\Rightarrow$  run(extract)(x) |+| run(extract)(y)
} // Semigroup laws will hold after applying run().
```

- The reduced encoding is  $FSR^Z \equiv Z \times List^Z$  (non-empty list of  $Z$ 's)
  - ▶ `x |+| y` requires concatenating the lists, but `run()` is faster

## Worked example II: Free monoid

Implement a free monoid `FM[Z]` generated by type `Z`

- A value of `FM[Z]` can be the empty value; it can also be a `Z`
- If `x, y` are of type `FM[Z]` then so is `x |+| y`

```
sealed trait FM[Z] // tree encoding
case class Empty[Z]() extends FM[Z]
case class Wrap[Z](z: Z) extends FM[Z]
case class Comb[Z](x: FM[Z], y: FM[Z]) extends FM[Z]
```

- Short type notation:  $FM^Z \equiv 1 + Z + FM^Z \times FM^Z$
- For a monoid  $M$  and given  $Z \Rightarrow M$ , map  $FM^Z \Rightarrow M$

```
def |+|(x: FM[Z], y: FM[Z]): FM[Z] = Comb(x, y)
def run[M: Monoid, Z](extract: Z => M): FM[Z] => M = {
  case Empty() => Monoid[M].empty
  case Wrap(z) => extract(z)
  case Comb(x, y) => run(extract)(x) |+| run(extract)(y)
} // Monoid laws will hold after applying run().
```

- The reduced encoding is  $FMR^Z \equiv List^Z$  (list of `Z`'s)
  - ▶ Implementing `|+|` requires concatenating the lists
- Reduced encoding and tree encoding give identical results after `run()`

# Mapping a free semigroup to different targets

What if we interpret  $\text{FS}^X$  into *another* free semigroup?

- Given  $Y \Rightarrow Z$ , can we map  $\text{FS}^Y \Rightarrow \text{FS}^Z$ ?
  - Need to map  $\text{FS}^Y \equiv Y + \text{FS}^Y \times \text{FS}^Y \Rightarrow Z + \text{FS}^Z \times \text{FS}^Z$
  - This is straightforward since  $\text{FS}^X$  is a functor in  $X$ :

```
def fmap[Y, Z](f: Y => Z): FS[Y] => FS[Z] = {  
  case Wrap(y) => Wrap(f(y))  
  case Comb(a, b) => Comb(fmap(f)(a), fmap(f)(b))  
}
```

- Now we can use `run` to interpret  $\text{FS}^X \Rightarrow \text{FS}^Y \Rightarrow \text{FS}^Z \Rightarrow S$ , etc.
  - Functor laws hold for  $\text{FS}^X$ , so `fmap` is composable as usual
  - The “interpreter” commutes with `fmap` as well (naturality law):

$$\begin{array}{ccc} & \text{FS}^Y & \\ \text{fmap } f: X \Rightarrow Y \nearrow & & \searrow \text{run}^S g: Y \Rightarrow S \\ \text{FS}^X & \xrightarrow{\text{run}^S (f \circ g): X \Rightarrow S} & S \end{array}$$

- Combine two free semigroups:  $\text{FS}^{X+Y}$ ; inject parts:  $\text{FS}^X \Rightarrow \text{FS}^{X+Y}$

# Church encoding I: Motivation

- Multiple target semigroups  $S_i$  require many “extractors”  $\text{ex}_i : Z \Rightarrow S_i$
- Refactor extractors  $\text{ex}_i$  into evidence of a typeclass constraint on  $S_i$

// Typeclass `ExZ[S]` has a single method, `extract`:  $Z \Rightarrow S$ .

```
implicit val exZ: ExZ[MySemigroup] = { z => ... }  
def run[S: ExZ : Semigroup](fm: FM[Z]): S = fm match {  
  case Wrap(z) => implicitly[ExZ[S]].extract(z)  
  case Comb(x, y) => run(x) |+| run(y)  
}
```

- `run()` replaces case classes by fixed functions parameterized by `S: ExZ`; instead we can represent `FM[Z]` directly by such functions, for example:

```
def wrap[S: ExZ](z: Z): S = implicitly[ExZ[S]].extract(z)  
def x[S: ExZ : Semigroup]: S = wrap(1) |+| wrap(2)
```

- The type of `x` is  $\forall S. (Z \Rightarrow S) \times (S \times S \Rightarrow S) \Rightarrow S$ ; an equivalent type is

$$\forall S. ((Z + S \times S) \Rightarrow S) \Rightarrow S$$

- This is the “**Church encoding**” (of the free semigroup over  $Z$ )
- The Church encoding is based on the theorem  $A \cong \forall X. (A \Rightarrow X) \Rightarrow X$ 
  - ▶ this *resembles* the type of the continuation monad,  $(A \Rightarrow R) \Rightarrow R$
  - ▶ but  $\forall X$  makes the function fully generic, like a natural transformation



## Church encoding II: Disjunction types

- Consider the Church encoding for the disjunction type  $P + Q$ 
    - The encoding is  $\forall X. (P + Q \Rightarrow X) \Rightarrow X \cong \forall X. (P \Rightarrow X) \Rightarrow (Q \Rightarrow X) \Rightarrow X$
- ```
trait Disj[P, Q] { def run[X](cp: P => X)(cq: Q => X): X }
```

- Define some values of this type:

```
def left[P, Q](p: P) = new Disj[P, Q] {  
  def run[X](cp: P => X)(cq: Q => X): X = cp(p)  
}
```

- Now we can implement the analog of the `case` expression simply as
- ```
val result = disj.run {p => ...} {q => ...}
```

- This works in programming languages that have no disjunction types

General recipe for implementing the Church encoding:

```
trait Blah { def run[X](cont: ... => X): X }
```

- For convenience, define a type class `Ex` describing the inner function:

```
trait Ex[X] { def cp: P => X; def cq: Q => X }
```

- Different methods of this class return `X`; convenient with disjunctions

- Church-encoded types have to be “run” for pattern-matching

# Church encoding III: How it works

Why is the type  $\text{Ch}^A \equiv \forall X. (A \Rightarrow X) \Rightarrow X$  equivalent to the type  $A$ ?

```
trait Ch[A] { def run[X](cont: A => X): X }
```

- If we have a value of  $A$ , we can get a  $\text{Ch}^A$

```
def a2c[A](a: A): Ch[A] = new Ch[A] {  
  def run[X](cont: A => X): X = cont(a)  
}
```

- If we have a  $\text{ch} : \text{Ch}^A$ , we can get an  $a : A$

```
def c2a[A](ch: Ch[A]): A = ch.run[A](a => a)
```

- The functions `a2c` and `c2a` are inverses of each other

- ▶ To implement a value  $\text{ch} : \text{Ch}^A$ , we must compute an  $x : X$  given  $f : A \Rightarrow X$ , for *any*  $X$ , which *requires* having a value  $a : A$  available

- To show that  $\text{ch} = \text{a2c}(\text{c2a}(\text{ch}))$ , apply both sides to an  $f : A \Rightarrow X$  and get  $\text{ch.run}(f) = \text{a2c}(\text{c2a}(\text{ch})).\text{run}(f) = f(\text{c2a}(\text{ch})) = f(\text{ch.run}(a \Rightarrow a))$

- ▶ This is naturality of `ch.run` as a transformation between `Reader` and `Id`
  - ★ Naturality of `ch.run` follows from parametricity of its code
- ▶ It is straightforward to compute  $\text{c2a}(\text{a2c}(a)) = \text{identity}(a) = a$

- Church encoding satisfies laws: it is built up from parts of `run` method

$$\begin{array}{ccc} \text{id} : (A \Rightarrow A) & \xrightarrow{\text{ch.run}^A} & A \\ \downarrow \text{fmap}_{\text{Reader}_A}(f) & & \downarrow f \\ f : (A \Rightarrow X) & \xrightarrow{\text{ch.run}^X} & X \end{array}$$

## Worked example III: Free functor I

- The `Functor` type class has one method, `fmap`:  $(Z \Rightarrow A) \Rightarrow F^Z \Rightarrow F^A$
- The tree encoding of a free functor over  $F^\bullet$  needs two case classes:

```
sealed trait FF[F[_], A]
case class Wrap[F[_], A](fa: F[A]) extends FF[F, A]
case class Fmap[F[_], A, Z](f: Z  $\Rightarrow$  A)(ffz: FF[F, Z]) extends FF[F, A]
```

- The constructor `Fmap` has an extra type parameter  $Z$ , which is “hidden”

Consider a simple example of this:

```
sealed trait Q[A]; case class QZ[A, Z](a: A, z: Z) extends Q[A]
```

- Need to use specific type  $Z$  when constructing a value of `Q[A]`, e.g.,

```
val q: Q[Int] = QZ[Int, String](123, "abc")
```

- ▶ The type  $Z$  is hidden inside  $q : Q^{\text{Int}}$ ; all we know is that  $Z$  “exists”
- Type notation for this:  $Q^A \equiv \exists Z. A \times Z$ 
  - ▶ The existential quantifier applies to the “hidden” type parameter
  - ▶ The constructor `QZ` has type  $\exists Z. (A \times Z \Rightarrow Q^A)$
  - ▶ It is not  $\forall Z$  because a specific  $Z$  is used when building up a value
  - ▶ The code does not show  $\exists Z$  explicitly! We need to keep track of that

# Encoding with an existential type: How it works

Show that  $P^A \equiv \exists Z. Z \times (Z \Rightarrow A) \cong A$

```
sealed trait P[A]; case class PZ[A, Z](z: Z, f: Z ⇒ A) extends P[A]
```

- How to construct a value of type  $P^A$  for a given  $A$ ?
  - ▶ Have a function  $Z \Rightarrow A$  and a  $Z$ , construct  $Z \times (Z \Rightarrow A)$
  - ▶ Particular case:  $Z \equiv A$ , have  $a : A$  and build  $a \times \text{id}^{A \Rightarrow A}$

```
def a2p[A](a: A): P[A] = PZ[A, A](a, identity)
```

- Cannot extract  $Z$  out of  $P^A$  – the type  $Z$  is hidden
- Can extract  $A$  out of  $P^A$  – do not need to know  $Z$

```
def p2a[A]: P[A] ⇒ A = { case PZ(z, f) ⇒ f(z) }
```

- Cannot transform  $P^A$  into anything else other than  $A$
- A value of type  $P^A$  is observable only via `p2a`
  - ▶ Therefore the functions `a2p` and `p2a` are “observational” inverses (i.e. we need to use `p2a` in order to compare values of type  $P^A$ )

If  $F^\bullet$  is a functor then  $Q^A \equiv \exists Z. F^Z \times (Z \Rightarrow A) \cong F^A$

- A value of  $Q^A$  can be observed only by extracting an  $F^A$  from it
- Can define `f2q` and `q2f` and show that they are observational inverses

## Worked example III: Free functor II

- Tree encoding of **FF** has type  $\text{FF}^{F^\bullet, A} \equiv F^A + \exists Z. \text{FF}^{F^\bullet, Z} \times (Z \Rightarrow A)$
- Derivation of the reduced encoding:

- ▶ A value of type  $\text{FF}^{F^\bullet, A}$  must be of the form

$$\exists Z_1. \exists Z_2 \dots \exists Z_n F^{Z_n} \times (Z_n \Rightarrow Z_{n-1}) \times \dots \times (Z_2 \Rightarrow Z_1) \times (Z_1 \Rightarrow A)$$

- ▶ The functions  $Z_1 \Rightarrow A$ ,  $Z_2 \Rightarrow Z_1$ , etc., must be composed associatively
- ▶ The equivalent type is  $\exists Z_n. F^{Z_n} \times (Z_n \Rightarrow A)$
- Reduced encoding:  $\text{FreeF}^{F^\bullet, A} \equiv \exists Z. F^Z \times (Z \Rightarrow A)$ 
  - ▶ Substituted  $F^Z$  instead of  $\text{FreeF}^{F^\bullet, Z}$  and eliminated the case  $F^A$
  - ▶ The reduced encoding is non-recursive
  - ▶ Requires a proof that this encoding is equivalent to the tree encoding
  - ▶ If  $F^\bullet$  is already a functor, can show  $F^A \cong \exists Z. F^Z \times (Z \Rightarrow A)$

- Church encoding (starting from the tree encoding):

$$\text{FreeF}^{F^\bullet, A} \equiv \forall P^\bullet. (\forall C. (F^C + \exists Z. P^Z \times (Z \Rightarrow C)) \rightsquigarrow P^C) \Rightarrow P^A$$

- ▶ The structure of the type expression:  $\forall P^\bullet. (\forall C. (\dots)^C \rightsquigarrow P^C) \Rightarrow P^A$ 
  - ★ Cannot move  $\forall C$  or  $\exists Z$  to the outside of the type expression!

# Church encoding IV: Recursive types and type constructors

- Consider the recursive type  $P \equiv Z + P \times P$  (tree with  $Z$ -valued leaves)
  - ▶ The Church encoding is  $\forall X. ((Z + X \times X) \Rightarrow X) \Rightarrow X$
  - ▶ This is *non-recursive*: the inductive use of  $P$  is replaced by  $X$
- Generalize to recursive type  $P \equiv S^P$  where  $S^\bullet$  is a “induction functor”:
  - ▶ The Church encoding of  $P$  is  $\forall X. (S^X \Rightarrow X) \Rightarrow X$ 
    - ★ Church encoding of recursive types is non-recursive
    - ★ Example: Church encoding of `List[Int]`
- Church encoding of a type constructor  $P^\bullet$ :
  - ▶ Notation:  $P^\bullet$  is a type function; Scala syntax is `P[_]`
  - ▶ The Church encoding is  $\text{Ch}^{P^\bullet, A} = \forall F^\bullet. (\forall X. P^X \Rightarrow F^X) \Rightarrow F^A$
  - ▶ Note:  $\forall X. P^X \Rightarrow F^X$  or  $P^\bullet \rightsquigarrow F^\bullet$  resembles a natural transformation
    - ★ Except that  $P^\bullet$  and  $F^\bullet$  are not necessarily functors, so no naturality law
  - ▶ Example: Church encoding of `Option[_]`
- Church encoding of a *recursively* defined type constructor  $P^\bullet$ :
  - ▶ Definition:  $P^A \equiv S^{P^\bullet, A}$  where  $S^{P^\bullet, A}$  describes the “induction principle”
  - ▶ Notation:  $S^{\bullet, A}$  is a higher-order type function; Scala syntax: `S[_][_, A]`
    - ★ Example:  $\text{List}^A \equiv 1 + A \times \text{List}^A \equiv S^{\text{List}^\bullet, A}$  where  $S^{P^\bullet, A} \equiv 1 + A \times P^A$
  - ▶ The Church encoding of  $P^A$  is  $\text{Ch}^{P^\bullet, A} = \forall F^\bullet. (S^{F^\bullet} \rightsquigarrow F^\bullet) \Rightarrow F^A$ 
    - ★ The Church encoding of `List[_]` is non-recursive

# Church encoding V: Type classes

- Look at the Church encoding of the free semigroup:

$$\text{ChFS}^Z \equiv \forall X. (Z \Rightarrow X) \times (X \times X \Rightarrow X) \Rightarrow X$$

- If  $X$  is constrained to the `Semigroup` typeclass, we will already have a value  $X \times X \Rightarrow X$ , so we can omit it:  $\text{ChFS}^Z = \forall X^{\text{Semigroup}}. (Z \Rightarrow X) \Rightarrow X$ 
  - The “induction functor” for “semigroup over  $Z$ ” is  $\text{SemiG}^X \equiv Z + X \times X$
  - So the Church encoding is  $\forall X. (\text{SemiG}^X \Rightarrow X) \Rightarrow X$

Generalize to arbitrary type classes:

- Type class  $C$  is defined by its operations  $C^X \Rightarrow X$  (with a suitable  $C^\bullet$ )
  - call  $C^\bullet$  the **method functor** of the inductive typeclass  $C$
- Tree encoding of “free  $C$  over  $Z$ ” is recursive,  $\text{FreeC}^Z \equiv Z + C^{\text{FreeC}^Z}$
- Church encoding is  $\text{FreeC}^Z \equiv \forall X. (Z + C^X \Rightarrow X) \Rightarrow X$ 
  - Equivalently,  $\text{FreeC}^Z \equiv \forall X^{C^\bullet}. (Z \Rightarrow X) \Rightarrow X$
- Laws of the typeclass are satisfied automatically after “running”
- Works similarly for type constructors: operations  $C^{P^\bullet, A} \Rightarrow P^A$
- Free typeclass  $C$  over  $F^\bullet$  is  $\text{FreeC}^{F^\bullet, A} \equiv \forall P^{C^\bullet}. (F^\bullet \leadsto P^\bullet) \Rightarrow P^A$

# Properties of free type constructions

Generalizing from our examples so far:

- We “enriched”  $Z$  to a monoid  $\text{FM}^Z$ , and  $F^A$  to a monad  $\text{DSL}^{F,A}$ 
  - ▶ The “enrichment” adds case classes representing the needed operations
  - ▶ Works for a generating type  $Z$  and for a generating type constructor  $F^A$
- Obtain a **free type construction**, which performs no computations
  - ▶  $\text{FM}^Z$  wraps  $Z$  in “just enough” stuff to make it look like a monoid
  - ▶  $\text{FreeF}^{F^\bullet,A}$  wraps  $F^A$  in “just enough” stuff to make it look like a functor
- A value of a free construction can be “run” to yield non-free values

Questions:

- Can we construct a free typeclass  $C$  over any type constructor  $F^A$ ?
  - ▶ Yes, with typeclasses: (contra)functor, filterable, monad, applicative
- Which of the possible encodings to use?
  - ▶ Tree encoding, reduced encodings, Church encoding
- What are the laws for the  $\text{FreeC}^{F,A}$  – “free instance of  $C$  over  $F$ ”?
  - ▶ For all  $F^\bullet$ , must have `wrap[A]` :  $F^A \Rightarrow \text{FreeC}^{F,A}$  or  $F^\bullet \rightsquigarrow \text{FreeC}^{F,\bullet}$
  - ▶ For all  $M^\bullet : C$ , must have `run` :  $(F^\bullet \rightsquigarrow M^\bullet) \Rightarrow \text{FreeC}^{F,\bullet} \rightsquigarrow M^\bullet$
  - ▶ The laws of typeclass  $C$  must hold after interpreting into an  $M^\bullet : C$
  - ▶ Given any `t`:  $F^\bullet \rightsquigarrow G^\bullet$ , must have `fmap(t)`:  $\text{FreeC}^{F,\bullet} \rightsquigarrow \text{FreeC}^{G,\bullet}$



# Recipes for encoding free typeclass instances

- Build a free instance of typeclass  $C$  over  $F^\bullet$ , as a type constructor  $P^\bullet$ 
  - ▶ The typeclass  $C$  can be functor, contrafunctor, monad, etc.
- Assume that  $C$  has methods  $m_1, m_2, \dots$ , with type signatures  $m_1 : Q_1^{P^\bullet, A} \Rightarrow P^A, m_2 : Q_2^{P^\bullet, A} \Rightarrow P^A$ , etc., where  $Q_i^{P^\bullet, A}$  are covariant in  $P^\bullet$ 
  - ▶ **Inductive typeclass** is defined via a methods functor,  $S^{P^\bullet} \rightsquigarrow P^\bullet$
- The tree encoded  $FC^A$  is a disjunction defined recursively by

$$FC^A \equiv F^A + Q_1^{FC^\bullet, A} + Q_2^{FC^\bullet, A} + \dots$$

```
sealed trait FC[A]; case class Wrap[A](fa: F[A]) extends FC[A]
case class Q1[A](...) extends FC[A]
case class Q2[A](...) extends FC[A]; ...
```

- ▶ Any type parameters within  $Q_i$  are then existentially quantified
  - ▶ `run()` maps  $F^\bullet \rightsquigarrow M^\bullet$  in the disjunction and recursively for other parts
- Derive a reduced encoding via reasoning about possible values of  $FC^A$  and by taking into account the laws of the typeclass  $C$
- A Church encoding can use the tree encoding or the reduced encoding
  - ▶ Church encoding is “automatically reduced”, but performance may differ

# Properties of inductive typeclasses

If a typeclass  $C$  is inductive with methods  $C^X \Rightarrow X$  then:

- A free instance of  $C$  over  $Z$  can be tree-encoded as  $\text{FreeC}^Z \equiv Z + C^{\text{FreeC}^Z}$ 
  - ▶ All inductive typeclasses have free instances,  $\text{FreeC}^Z$
- If  $P:C$  and  $Q:C$  then  $P \times Q$  and  $Z \Rightarrow P$  also belong to typeclass  $C$ 
  - ▶ but not necessarily  $P + Q$  or  $Z \times P$
  - ▶ Proof: can implement  $(C^P \Rightarrow P) \times (C^Q \Rightarrow Q) \Rightarrow C^{P \times Q} \Rightarrow P \times Q$  and  $(C^P \Rightarrow P) \Rightarrow C^{Z \Rightarrow P} \Rightarrow Z \Rightarrow P$ , but cannot implement  $(...) \Rightarrow P + Q$
- Analogous properties hold for type constructor typeclasses
  - ▶ Methods described as  $C^{F^\bullet, A} \Rightarrow F^A$  with type constructor parameter  $F^\bullet$

What typeclasses *cannot* be tree-encoded (or have no “free” instances)?

- Any typeclass with a method *not ultimately returning* a value of  $P^A$ 
  - ▶ Example: a typeclass with methods  $\text{pt} : A \Rightarrow P^A$  and  $\text{ex} : P^A \Rightarrow A$
- Such typeclasses are not inductive
  - ▶ Typeclasses with methods of the form  $P^A \Rightarrow \dots$  are **co-inductive**

## Worked example IV: Free contrafunctor

- Method contramap :  $C^A \times (B \Rightarrow A) \Rightarrow C^B$
- Tree encoding:  $\text{FreeCF}^{F^\bullet, B} \equiv F^B + \exists A. \text{FreeCF}^{F^\bullet, A} \times (B \Rightarrow A)$
- Reduced encoding:  $\text{FreeCF}^{F^\bullet, B} \equiv \exists A. F^A \times (B \Rightarrow A)$ 
  - ▶ The reduced encoding is non-recursive
  - ▶ Example:  $F^A \equiv A$ , “interpret” into the contrafunctor  $C^A \equiv A \Rightarrow \text{String}$

```
def prefixLog[A](p: A): A ⇒ String = a ⇒ p.toString + a.toString
```

- If  $F^\bullet$  is already a contrafunctor then  $\text{FreeCF}^{F^\bullet, A} \cong F^A$

## Worked example V: Free pointed functor

Over an arbitrary type constructor  $F^\bullet$ :

- Pointed functor methods  $\text{pt} : A \Rightarrow P^A$  and  $\text{map} : P^A \times (A \Rightarrow B) \Rightarrow P^B$
- Tree encoding:  $\text{FreeP}^{F^\bullet, A} \equiv A + F^A + \exists Z. \text{FreeP}^{F^\bullet, Z} \times (Z \Rightarrow A)$
- Reduced encoding:  $\text{FreeP}^{F^\bullet, A} \equiv A + \exists Z. F^Z \times (Z \Rightarrow A)$
- This reuses the free functor as  $\text{FreeP}^{F^\bullet, A} = A + \text{FreeF}^{F^\bullet, A}$

If the type constructor  $F^\bullet$  is *already* a functor,  $\text{FreeF}^{F^\bullet, A} \cong F^A$  and so:

- Free pointed functor over a functor  $F^\bullet$  is simplified:  $A + F^A$
- If  $F^\bullet$  is already a pointed functor, need not use the free construction
  - ▶ If we do, we will have  $\text{FreeP}^{F^\bullet, A} \not\cong F^A$
  - ▶ only functors and contrafunctors do not change under “free”

## Worked example VI: Free filterable functor

- Methods:

$$\text{map} : F^A \Rightarrow (A \Rightarrow B) \Rightarrow F^B$$

$$\text{mapOpt} : F^A \Rightarrow (A \Rightarrow 1 + B) \Rightarrow F^B$$

- We can recover `map` from `mapOpt`, so we keep only `mapOpt`
- Tree encoding:  $\text{FreeFi}^{F^\bullet, A} \equiv F^A + \exists Z. \text{FreeFi}^{F^\bullet, Z} \times (Z \Rightarrow 1 + A)$
- Reduced encoding:  $\text{FreeFi}^{F^\bullet, A} \equiv \exists Z. F^Z \times (Z \Rightarrow 1 + A)$ , non-recursive
- If  $F^\bullet$  is already a functor, can simplify:  $\text{FreeFi}^{F^\bullet, A} = F^{1+A}$ 
  - ▶ Free filterable over a filterable functor  $F^\bullet$  is  $F^{1+A} \not\equiv F^A$
- Free filterable contrafunctor is constructed in a similar way

## Worked example VII: Free monad

- Methods:

$$\text{pure} : A \Rightarrow F^A$$

$$\text{flatMap} : F^A \Rightarrow (A \Rightarrow F^B) \Rightarrow F^B$$

- Can recover `map` from `flatMap` and `pure`, so we keep only `flatMap`
- Tree encoding:  $\text{FreeM}^{F^\bullet, A} \equiv F^A + A + \exists Z. \text{FreeM}^{F^\bullet, Z} \times (Z \Rightarrow \text{FreeM}^{F^\bullet, A})$
- Derive a reduced encoding:
  - ▶ can simplify  $A \times (A \Rightarrow \text{FreeM}^{F^\bullet, B}) \cong \text{FreeM}^{F^\bullet, B}$
  - ▶ use associativity to replace  $\text{FreeM}^A \times (A \Rightarrow \text{FreeM}^B) \times (B \Rightarrow \text{FreeM}^C)$  by  $\text{FreeM}^A \times (A \Rightarrow \text{FreeM}^B \times (B \Rightarrow \text{FreeM}^C))$
  - ▶ therefore we can replace  $\exists Z. \text{FreeM}^{F^\bullet, Z} \times \dots$  by  $\exists Z. F^Z \times \dots$
- Reduced encoding:  $\text{FreeM}^{F^\bullet, A} \equiv A + \exists Z. F^Z \times (Z \Rightarrow \text{FreeM}^{F^\bullet, A})$
- Free monad over a functor  $F^\bullet$  is  $\text{FreeM}^{F^\bullet, A} \equiv A + F^{\text{FreeM}^{F^\bullet, A}}$ 
  - ▶ Free monad  $\text{FreeM}^{M^\bullet, \bullet}$  over a monad  $M^\bullet$  is not equivalent to  $M^\bullet$
- Free monad over a pointed functor  $F^\bullet$  is  $\text{FreeM}^{F^\bullet, A} \equiv F^A + F^{\text{FreeM}^{F^\bullet, A}}$ 
  - ▶ start from half-reduced encoding  $F^A + \exists Z. F^Z \times (Z \Rightarrow \text{FreeM}^{F^\bullet, A})$
  - ▶ replace the existential type by an equivalent type  $F^{\text{FreeM}^{F^\bullet, A}}$

## Worked example VIII: Free applicative functor

- Methods:

$$\text{pure} : A \Rightarrow F^A$$

$$\text{ap} : F^A \Rightarrow F^{A \Rightarrow B} \Rightarrow F^B$$

- We can recover **map** from **ap** and **pure**, so we keep only **ap**
- Tree encoding:  $\text{FreeAp}^{F^\bullet, A} \equiv F^A + A + \exists Z. \text{FreeAp}^{F^\bullet, Z} \times \text{FreeAp}^{F^\bullet, Z \Rightarrow A}$
- Reduced encoding:  $\text{FreeAp}^{F^\bullet, A} \equiv A + \exists Z. F^Z \times \text{FreeAp}^{F^\bullet, Z \Rightarrow A}$ 
  - ▶ Requires derivation
- Free applicative over a functor  $F^\bullet$ :

$$\text{FreeAp}^{F^\bullet, A} \equiv A + \text{FreeZ}^{F^\bullet, A}$$

$$\text{FreeZ}^{F^\bullet, A} \equiv F^A + \exists Z. F^Z \times \text{FreeZ}^{F^\bullet, Z \Rightarrow A}$$

- ▶  $\text{FreeZ}^{F^\bullet, \bullet}$  is the reduced encoding of “free zippable” (no **pure**)
- $\text{FreeAp}^{F^\bullet, \bullet}$  over an applicative functor  $F^\bullet$  is not equivalent to  $F^\bullet$

# Laws for free typeclass constructions

Consider an inductive typeclass  $C$  with methods  $C^A \Rightarrow A$

Define a free instance of  $C$  over  $Z$  recursively,  $\text{FreeC}^Z \equiv Z + C^{\text{FreeC}^Z}$

- $\text{FreeC}^Z$  has an instance of  $C$ , i.e. we can implement  $C^{\text{FreeC}^Z} \Rightarrow \text{FreeC}^Z$
- For any  $P:C$  we can implement the functions

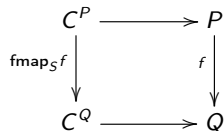
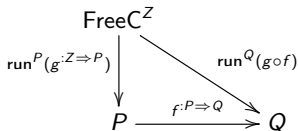
$$\text{run}^P : (Z \Rightarrow P) \Rightarrow \text{FreeC}^Z \Rightarrow P$$

$$\text{wrap} : Z \Rightarrow \text{FreeC}^Z$$

such that  $\text{run}(\text{wrap}) = \text{id}$  and, for all  $g:Z \Rightarrow P$ ,  $\text{wrap} \circ \text{run}(g) = g$

- For any  $P:C$ ,  $Q:C$ ,  $g:Z \Rightarrow P$ , and a typeclass-preserving  $f:P \Rightarrow Q$ , we have

$$\text{run}^P(g) \circ f = \text{run}^Q(g \circ f) \quad - \text{“universal property” of run}$$



- $f:P \Rightarrow Q$  **preserves typeclass  $C$**  if the diagram on the right commutes
- $\text{FreeC}^Z$  is always a functor in  $Z$ ;  $\text{fmap} : (Y \Rightarrow Z) \Rightarrow \text{FreeC}^Y \Rightarrow \text{FreeC}^Z$



# Combining the generating constructors in a free typeclass

- Consider  $\text{FreeC}^Z$  for an inductive typeclass  $C$  with methods  $C^X \Rightarrow X$
- We would like to combine generating constructors  $Z_1, Z_2$ , etc.
  - ▶ In a monadic DSL – combine different operations defined separately
  - ▶ Note that monads do not compose in general
- To combine generators, use  $\text{FreeC}^{Z_1+Z_2}$ ; an “instance over  $Z_1$  and  $Z_2$ ”
  - ▶ but need to inject parts into disjunction, which is cumbersome
- Church encoding makes this easier to manage:
  - ▶  $\text{FreeC}^Z \equiv \forall X. (Z \Rightarrow X) \times (C^X \Rightarrow X) \Rightarrow X$  and then

$$\text{FreeC}^{Z_1+Z_2} \equiv \forall X. (Z_1 \Rightarrow X) \times (Z_2 \Rightarrow X) \times (C^X \Rightarrow X) \Rightarrow X$$

- ▶ Encode the functions  $Z_i \Rightarrow X$  via typeclasses [ExZ1](#), [ExZ2](#), etc., where typeclass [ExZ1](#) has method  $Z_1 \Rightarrow X$ , etc.
- ▶ Then

$$\text{FreeC}^{Z_1+Z_2} = \forall X^{E_{Z_1}:E_{Z_2}}. (C^X \Rightarrow X) \Rightarrow X$$

so we can postpone choosing  $X$  until we run the DSL program

- ▶ Easier to reuse code

# Combining different free typeclasses

To combine free instances of different typeclasses  $C_1$  and  $C_2$ :

- Option 1: use functor composition,  $\text{Free}C_{12}^Z \equiv \text{Free}C_1^{\text{Free}C_2^Z}$ 
  - ▶ Order of composition matters!
  - ▶ Operations of  $C_2$  need to be lifted into  $C_1$
  - ▶ Works only for positive inductive typeclasses
- Option 2: use disjunction of method functors,  $C^X \equiv C_1^X + C_2^X$ , and build the free typeclass instance using  $C^X$ 
  - ▶ Church encoding:  $\text{Free}C_{12}^Z \equiv \forall X. (Z \Rightarrow X) \times (C_1^X + C_2^X \Rightarrow X) \Rightarrow X$
- Example 1:  $C_1$  is functor,  $C_2$  is contrafunctor
  - ▶ Interpret a free functor/contrafunctor into a profunctor
- Example 2:  $C_1$  is monad,  $C_2$  is applicative functor
  - ▶ Interpret into a monad that has an optimized `zip` implementation
  - ▶ Use `Future` but translate `zip` into parallel execution

# Exercises

- 1 Implement a free semigroup generated by a type  $Z$  in the tree encoding and in the reduced encoding. Show that the semigroup laws hold for the reduced encoding but not for the tree encoding before interpreting into a lawful semigroup  $S$ .
- 2 Consider a free monoid generated by a type  $Z$  when  $Z$  is already a monoid. Show that the resulting type is not equivalent to  $Z$ .
- 3 Implement a monadic DSL with operations `put: A ⇒ 1` and `get: A`; run examples.
- 4 Implement the Church encoding of the type constructor  $P^A \equiv \text{Int} + A \times A$ . For the resulting type constructor, implement a `Functor` instance.
- 5 Describe the monoid type class via a method functor  $C^\bullet$  (such that the monoid's operations are combined into the type  $S^M \Rightarrow M$ ). Using  $S^\bullet$ , implement the free monoid over a type  $Z$  in the Church encoding.
- 6 Assuming that  $F^\bullet$  is a functor, define  $Q^A \equiv \exists Z. F^Z \times (Z \Rightarrow A)$  and implement `f2q: F^A ⇒ Q^A` and `q2f: Q^A ⇒ F^A`. Show that these functions are natural transformations, and that they are inverses of each other “observationally”, i.e. after applying `q2f` in order to compare values of  $Q^A$ .
- 7 Using  $\exists Z. Z \times (Z \Rightarrow A) \cong A$ , show that  $\exists Z. Z \cong 1$  and that  $\exists Z. Z \times A \cong A$ .
- 8 Derive a reduced encoding for a free applicative functor over  $F^\bullet$ , where the type constructor  $F^\bullet$  is already a pointed functor.
- 9 Implement a “free pointed filterable” typeclass (combining pointed and filterable) over a type constructor  $F^\bullet$ . Start from the tree encoding and derive a reduced encoding. Find a simplified encoding for the case when  $F^\bullet$  is already a functor.