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
val b = a * "3-4*i".toComplex
b.conj
```

```
Conj(
  Mul(
    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 $\text{Ex}^F \equiv \forall A.\left(F^A \Rightarrow \text{Err} + A\right)$
- 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] ⇒ b match { case Bind(p, f) ⇒
        run(extract)(p).flatMap(f andThen run(extract))
    }     // Here, the .flatMap is from Either.
    case Literal(a) ⇒ Right(a) // pure: A ⇒ Err + A
    case Ops(f) ⇒ 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]⇒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] ⇒ 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: $\text{DSL}^{F+G+H,A}$

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 ⇒ 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 ⇒ Literal(a)) andThen run(ex))
  == run(ex)(prg).flatMap(a ⇒ run(ex)(Literal(a))
  == run(ex)(prg).flatMap(a ⇒ 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 ц (tsè) and the Chinese word 水 (shuǐ)
- We want to *multiply* ц by 水. Multiply how?
- Say, we want an associative (but noncommutative) product of them
  - So we want to define a *semigroup* that *contains* ц and 水 as elements
    - while we still know nothing about ц and 水
- Consider the set of all *unevaluated expressions* such as ц·水·水·ц·水
  - Here ц·水 is different from 水·ц but $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
- All these expressions form a **free semigroup** generated by ц and 水
  - This is the most unrestricted semigroup that contains ц and 水
- Example calculation: (水·水)·(ц·水)·ц = 水·水·ц·水·ц

How to represent this as a data type:

- **Tree encoding**: the full expression tree: (((水,水),(ц,水)),ц)
  - Implement the operation $a \cdot b$ as pair constructor (easy)
- **Reduced encoding**, as a "smart" structure: List(水,水,ц,水,ц)
  - 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):

$$\begin{aligned}
\mathbf{u}_1 \otimes \mathbf{v}_1 + (\mathbf{u}_2 \otimes \mathbf{v}_2 + \mathbf{u}_3 \otimes \mathbf{v}_3) &= (\mathbf{u}_1 \otimes \mathbf{v}_1 + \mathbf{u}_2 \otimes \mathbf{v}_2) + \mathbf{u}_3 \otimes \mathbf{v}_3 \\
\mathbf{u} \otimes (a_1 \mathbf{v}_1 + a_2 \mathbf{v}_2) &= a_1 (\mathbf{u} \otimes \mathbf{v}_1) + a_2 (\mathbf{u} \otimes \mathbf{v}_2) \\
(a_1 \mathbf{v}_1 + a_2 \mathbf{v}_2) \otimes \mathbf{u} &= a_1 (\mathbf{v}_1 \otimes \mathbf{u}) + a_2 (\mathbf{v}_2 \otimes \mathbf{u})
\end{aligned}$$

  - We have such a product for 3-dimensional vectors only; ignore that
- Consider *unevaluated expressions* of the form $\mathbf{u}_1 \otimes \mathbf{v}_1 + \mathbf{u}_2 \otimes \mathbf{v}_2 + \ldots$
  - 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: $\text{FSIS} \equiv \text{Int} + \text{String} + \text{FSIS} \times \text{FSIS}$
- For a semigroup S and given $\text{Int} \Rightarrow S$ and $\text{String} \Rightarrow S$, map $\text{FSIS} \Rightarrow S$
- Simplify and generalize this construction by setting $Z = \text{Int} + \text{String}$
  - The tree encoding is $\text{FS}^Z \equiv Z + \text{FS}^Z \times \text{FS}^Z$

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

- The reduced encoding is $\text{FSR}^Z \equiv Z \times \text{List}^Z$ (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: $\text{FM}^Z \equiv 1 + Z + \text{FM}^Z \times \text{FM}^Z$
- For a monoid M and given $Z \Rightarrow M$, map $\text{FM}^Z \Rightarrow M$

```
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 $\text{FMR}^Z \equiv \text{List}^Z$ (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 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 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 FS^X, so `fmap` is composable as usual
  - The "interpreter" commutes with `fmap` as well (naturality law):

$$\begin{tikzcd}
 & \text{FS}^Y \arrow[dr, "\text{run}^S g^{:Y\Rightarrow S}"] & \\
\text{FS}^X \arrow[ur, "\text{fmap}\, f^{:X\Rightarrow Y}"] \arrow[rr, "\text{run}^S (f\circ g)^{:X\Rightarrow S}"'] & & S
\end{tikzcd}$$

- Combine two free semigroups: 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 ex_i into evidence of a typeclass constraint on S_i

```
// Typeclass ExZ[S] has the single method ‘extract: Z ⇒ 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)
}
```

- Refactor `run` using a helper function `wrap`

```
def wrap[S: ExZ](z: Z): S = implicitly[ExZ[S]].extract(z)
```

- Refactor the rest of `run` into functions with constraint `[S: ExZ]`,

```
def x[S: ExZ : Semigroup]: S = wrap(1) |+| wrap(2)
```

- The type of `x` is $\forall S.\, (Z \Rightarrow S) \times (S \times S \Rightarrow S) \Rightarrow S$; equivalently:

$$\forall S.\, (Z \Rightarrow S) \times (S \times S \Rightarrow S) \Rightarrow S \;\cong\; \forall S.\, ((Z + S \times S) \Rightarrow S) \Rightarrow S$$

- This is known as the "**Church encoding**" (for the free semigroup)
- Church encoding works for any type: $A \cong \forall X.\, (A \Rightarrow X) \Rightarrow X$
  - which *resembles* the type of the continuation monad, $(A \Rightarrow R) \Rightarrow R$
  - but $\forall X$ makes the function fully generic and 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 Ch^A

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

$$\begin{tikzcd} \text{id} : (A \Rightarrow A) \arrow[r, "\text{ch.run}^A"] \arrow[d, "\text{fmap}_{\textbf{Reader}_A}(f)"] & A \arrow[d, "f"] \\ f : (A \Rightarrow X) \arrow[r, "\text{ch.run}^X"'] & X \end{tikzcd}$$

- 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 `ch = a2c(c2a(ch))`, apply both sides to an `f: A⇒X` and get `ch.run(f) = a2c(c2a(ch)).run(f) = f(c2a(ch)) = f(ch.run(a⇒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 `c2a(a2c(a)) = identity(a) = a`
- Church encoding satisfies laws: it is built up from parts of `run` method

Church encoding IV. Recursive types and type constructors

- Consider the recursive type $P \equiv Z + P \times P$ (tree with Z-typed leaves)
  - The Church encoding is $\forall X. ((Z + X \times X) \Rightarrow X) \Rightarrow X$
  - This is *non-recursive*: the recursive use of P is replaced by X
- Generalize to recursive type $P \equiv S^P$ where S is a "structure functor":
  - The Church encoding is $\forall X. \left(S^X \Rightarrow X\right) \Rightarrow X$
    - ★ Church encoding of recursive types is non-recursive
- Church encoding for a type constructor $P^\bullet$:
  - Notation: $P^\bullet$ is a type function; Scala syntax is `P[_]`
  - The Church encoding is $\text{Ch}^{P^\bullet, A} = \forall F^\bullet. \left(\forall X. P^X \Rightarrow F^X\right) \Rightarrow F^A$
  - Note: $\forall X. P^X \Rightarrow F^X$ or $P^\bullet \leadsto F^\bullet$ resembles a natural transformation
    - ★ Except that $P^\bullet$ and $F^\bullet$ are not necessarily functors, so no naturality law
- Church encoding for a recursively defined type constructor $P^\bullet$:
  - Definition: $P^A \equiv S^{P^\bullet, A}$ where $S^{\bullet^\bullet, A}$ is the "structure transformation"
  - Notation: $S^{\bullet^\bullet, A}$ 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 $\text{Ch}^{P^\bullet, A} = \forall F^\bullet. \left(S^{F^\bullet} \leadsto F^\bullet\right) \Rightarrow F^A$
- Works the same if $P^\bullet$ belongs to a typeclass – use $\forall F^\bullet : C. (...)$
  - Laws of the typeclass are satisfied automatically

Properties of free type constructions

Generalizing from our examples so far:

- We "enriched" Z to a monoid FM^Z and F^A to a monad $\text{DSL}^{F,A}$
  - The "enrichment" adds case classes representing the needed operations
  - Very similar recipes for a type Z and for a type constructor F^A
- Obtain a **free type construction**, which performs no computations
  - wrap Z in just enough stuff to make it look like a monoid
- A value of a free construction can be "run" to yield concrete 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 $F^\bullet \leadsto \text{FreeC}^{F,\bullet}$
  - For all $M^\bullet : C$, given $F^\bullet \leadsto M^\bullet$, must have `run` : $\text{FreeC}^{F,\bullet} \leadsto M^\bullet$
  - The laws of typeclass C must hold after interpreting into an $M^\bullet : C$
  - Given any `t`: $F^\bullet \leadsto G^\bullet$, must have `fmap(t)`: $\text{FreeC}^{F,\bullet} \leadsto \text{FreeC}^{G,\bullet}$

Worked example III: free functor

- Only one operation: $F^Z \times (Z \Rightarrow A) \Rightarrow F^A$
- Tree encoding: $\text{FreeF}^{F^\bullet,A} \equiv F^A + \exists Z.\text{FreeF}^{F^\bullet,Z} \times (Z \Rightarrow A)$
  - The existential quantifier is represented by an extra type parameter
  - It is not $\forall Z$ because a specific Z is used when building up a value
- 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):
  $\text{FreeF}^{F^\bullet,A} \equiv \forall P^\bullet.\left(\forall C.\left(F^C + \exists Z.P^Z \times (Z \Rightarrow C)\right) \leadsto P^C\right) \Rightarrow P^A$

Encoding with a hidden type: how it works

- We have

$$\exists Z.Z \times (Z \Rightarrow A) \cong A$$

$$\exists Z.F^{Z} \times (Z \Rightarrow A) \cong F^{A} \quad \text{if } F^{\bullet} \text{ is a functor}$$

  - How can we construct a value of type $\exists Z.Z \times (Z \Rightarrow A)$?
  - To construct a value of type

Worked examples IV: free contrafunctor

- Operation 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
```

Worked examples V: free pointed functor

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

- Pointed functor operations: $A \Rightarrow \mathrm{P}^A$ and $P^A \times (A \Rightarrow B) \Rightarrow P^B$
- 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 functor over a functor $F^\bullet$ is $\text{FreeF}^{F^\bullet,A} \equiv A + F^A$

Worked example: free filterable functor

- Operations:

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

  Can recover `map` from `mapOpt`, so let us 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: $\text{FreeFi}^{F^\bullet,A} = F^{1+A}$

Worked example: free monad

- Operations:

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

- Free monad when starting from a functor F

Worked example: free applicative

- Generalize

Free constructions as “universal” DSL programs

- Generalize

Type classes not available for free constructions

- Generalize

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.yes