

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 `Literal`, `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] { def bind(f: String⇒Prg[String]): Prg[String] }
case class Path(s: Prg[String]) extends Prg[nio.file.Path]
case class Read(p: Prg[nio.file.Path]) extends Prg[String]
case class Literal(s: String) 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:

- Promote `Literal` to a fully parameterized operation
- Replace `Prg[A]` by `A` in case class arguments

Resulting DSL:

```
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]
```

- Note the `flatMap`-like type signature of `Bind`

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” $ex : \forall A. F^A \Rightarrow A$
`def run[F[_], A](ex: Ex[F])(p: DSL[F, A]): A = ...`

- The constructor `DSL[F[_], A]` is called a **free monad** over `F`

- Consider

- Consider

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
- 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:

- Redundant encoding, as the full expression tree: $((\text{水}, \text{水}), (\mathfrak{u}, \text{水})), \mathfrak{u})$
 - ▶ Implement the operation $a \cdot b$ as pair constructor (easy and cheap)
- Reduced encoding, as a “smart” structure: $\text{List}(\text{水}, \text{水}, \mathfrak{u}, \text{水}, \mathfrak{u})$
 - ▶ Implement the operation $a \cdot b$ 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**
- Redundant encoding: unevaluated expression tree
 - ▶ A list of any number of vector pairs $\sum_i \mathbf{u}_i \otimes \mathbf{v}_i$
- Reduced encoding: a matrix
 - ▶ Reduced encoding requires proof and more complex operations

- 1 Show that