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 interpret(p: Prg): (Double, Double) = ...
```

- Benefits: programs are data, can compose & transform before running
- Shortcomings: this DSL works only with simple expressions
  - Hard to represent variable binding and conditional use
  - 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 use

- Example: imperative API for reading and writing files
- Need to bind a *non-DSL variable* to a value computed by DSL
- The rest of the DSL program must be a function of that variable

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

```
Read(Path(Literal("/file")))
.bind { str ⇒ // read value ‘str‘
  if (str.nonEmpty)
    Read(Path(str))
  else Literal("Error: empty path")
}
```

- Unevaluated operations are implemented via case classes:

```
sealed trait Prg { def bind: ??? }
case class Path(s: Prg) extends Prg
case class Read(p: Prg) extends Prg
case class Literal(s: String) extends Prg
```

- The required type signature of `bind` is

```
def bind(f: String ⇒ Prg): Prg
```

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

The interpreter pattern III. Type safety

- 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] { def bind: ??? }
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]
```

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

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

- The required type signature of `bind` seems to be

```
def bind(f: String ⇒ Prg[String]): Prg[String]
```

The interpreter pattern IV. Cleaning up the DSL

```
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`
- Cannot actually implement `.bind()` without running the program!

To fix these problems:

- Promote `bind` to a new unevaluated operation `Bind()`
- 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]
```

The interpreter pattern V. Define a `Monad` instance

- We can now 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[B])
}
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: String ⇒ Prg[String] = for {
  path ← Path("/file")
  str  ← Read(path)
} yield str

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

- Interpreter: `def run[A](p: 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[A]
case class Literal[A](a: A) extends Prg[A]
case class Ops[A](f: F[A]) extends Prg[A] // custom operations here
```

- Interpreter: `def run[F[_], A](p: DSL[F, A]): A = ...`

Free

- Consider

Free

- Consider

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
- Example calculation: (水·水)·(ц·水)·ц = 水·水·ц·水·ц

How to represent this as a data type:

- Redundant encoding, as the full expression tree: ((水,水),(ц,水)),ц)
  - Implement the operation $a \cdot b$ as pair constructor (easy and cheap)
- Reduced encoding, as a "smart" structure: List(水,水,ц,水,ц)
  - Implement the opration $a \cdot b$ 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 $\mathbf{u}_1 \otimes \mathbf{v}_1 + \mathbf{u}_2 \otimes \mathbf{v}_2 + ...$
  - 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

Exercises

1. Show that