Chapter 10: Free type constructions

Sergei Winitzki

Academy by the Bay

2018-11-22

The interpreter pattern I. Expression trees as programs

Represent a program as a data structure, run later

- Example: a simple DSL for complex numbers

```
val a = "1+2*i".cplx
val b = a * "3-4*i".cplx
b.conj
```

```
Conj(
  Mul(
   Cplx("1+2*i"), Cplx("3-4*i")
))
```

- `Cplx`, `Mul`, `Conj` etc. are defined as case classes:

```
sealed trait Expr
final case class Cplx(s: String) extends Expr
final case class Mul(e1: Expr, e2: Expr) extends Expr
final case class Conj(e: Expr) extends Expr
```

- An interpreter will "run" the program and return a complex number

```
def interpret(e: Expr): (Double, Double) = ...
```

- This technique creates a DSL that works only with simple expressions
  - Hard to represent variable binding and variable reuse
  - Cannot mix in any non-DSL code (e.g. a numerical algorithms library)

The interpreter pattern II. Variable binding

Represent an imperative program in the form of a data structure

- Example: complex numbers

```
read(buffer1, "/file1")
create("/file2")
write("/file2", buffer1)
read(buffer2, "/file3")
delete("/file3")
return buffer2
```

```
List( Create("/file"),
Write("/file", "hello"),
Read(buffer, "/file"),
Delete("/file")
```

- a

Free

- Consider

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