CSC 347 - Concepts of Programming Languages

Algebraic Data Types

Instructor: James Riely

Learning Objectives

❓ Complex data in functional programming?

- Identify sums and products in algebraic data types
- Use Scala `enum` classes to express algebraic data types

Algebraic Data Types

- Algebraic data types: sum of products

Product types

- Combine multiple data elements into one unit
- Tuples and classes

```
1 val product =
2   (1, "hello", List(1, 2, 3))
```

Sum types

- Distinguish multiple alternatives
- Union type `Int | String`
- Discriminated union

```
1 class A
2 class B extends A
3 class C extends A
```

- Algebraic Data Types (wikipedia) in PLs

Product Types

- Named for *Cartesian product* of sets
  - $X \times Y = \{(x, y) \mid x \in X \land y \in Y\}$
- Case class definition for product of `Int` and `String`

```
1 case class C(x:Int, y:String)
```

- Construct instances (`new` unnecessary)

```
1 val c:C = C(5, "hello")
```

- Extract elements with pattern matching

```
1 val n:Int = c match
2   case C(a, b) => a
```

Scala Case Classes

- Compiler treatment for `case` classes
- Class parameters are visible and immutable (`val`)

```scala
case class C (x:Int, y:String)
val c:C = C (5, "hello")
val a:Int = c.x
c.x = 6 // error: reassignment to val
```

- Sensible `toString` implementation
- Companion object with `apply` method
  - used to construct instances
- Pattern matching support
  - see `unapply` method / extractors in textbook

Sum Types

- Named for sums (union) of sets
  $X \cup Y = \{z \mid z \in X \vee z \in Y\}$
- *Coproduct* or *disjoint union* of sets
  $X \oplus Y = X \uplus Y = \{(0, x) \mid x \in X\} \cup \{(1, y) \mid y \in Y\}$
- Elements are tagged to indicate their source (the class of the element)

Disjoint Union: Scala Enum

- Scala `enum` lists disjoint alternatives

```
1 enum Color:
2   case Blue
3   case White
```

- Disjoint union of value and operator

```
1 enum Expr:
2   case Number(x: Int)
3   case Plus(l: Expr, r: Expr)
4   ...
```

- Definition can be recursive

- Create instances

```
1 val b = Color.Blue
```

```
1 import Expr.*
2 val three = Number(3)
3 val plus = Plus(three, Number(5))
```

Coding Exercise

- Object-oriented vs. functional calculator

Disjoint Union: Scala Enum

- Pattern match to decompose

```
1 def eval(e: Expr) : Int = e match
2   case Number(x)   => x
3   case Plus(l, r)  => eval(l) + eval(r)
4   ...
5 end eval
```

- Create instance and pass to `eval`

```
1 // (3+5)*2
2 val v = Times(Plus(Number(3), Number(5)), Number(2))
3 val i: Int = eval(v) // i==16
```

Exercise: Linked List

❓ Which "states" does a list have → which case classes and case objects?

- An `Empty` list and a `Cons` cell of at least one element

```
1 enum MyList:
2   case Empty
3   case Cons (head:Int, tail:MyList)
4 end MyList
```

Exercise: Linked List

```
1 enum MyList:
2   case Empty
3   case Cons (head:Int, tail:MyList)
4 end MyList
```

❓ Create an empty list?

- Simply use `Empty`

```
1 import MyList.*
2 val xs = Empty
```

- ❓ Create an instance of a list?
- Nest `Cons` and terminate with `Empty`

```
1 import MyList.*
2 val xs = Cons (1, Cons(2, Cons(3, Empty)))
```

Exercise: Linked List

```scala
enum MyList:
  case Empty
  case Cons (head:Int, tail:MyList)
end MyList

object MyList:
  def apply(elems: Int*) : MyList =
    if elems.isEmpty then Empty
    else Cons(elems.head, apply(elems.tail*))
end MyList
```

- ❓ Create an instance of a list?
- Use method `apply` (name can be omitted)

```scala
import MyList.*
// val xs = Cons (1, Cons(2, Cons(3, Empty)))
val xs = MyList(1, 2, 3) // MyList.apply(1, 2, 3)
```

Summary

Algebraic data types: sum of products

Product types

- Combine multiple data elements
- Tuples and classes

```
1 val product = (1, "hello", List(1, 2, 3))
```

Sum types

- Distinguish multiple alternatives
- Subclasses

```
1 class A; class B extends A; class C extends A
```

- Union types: `B | C`

- In Scala: `enum` and `case` classes

```
1 enum Sum:
2   case Product1(i: Int, s: String)
3   case Product2(i: Int, j: Int, k: Int)
4   ...
```

Exercise: Peano Natural Numbers

- Peano natural numbers: either 0 or a transitive successor of it
- Algebraic data type `PeanoNat`

```scala
enum PeanoNat:
  case Zero
  case Succ(n: PeanoNat)
```

- Evaluate `PeanoNat` to `Int`

```scala
import PeanoNat.*
def peano2int (p: PeanoNat, result: Int = 0): Int = p match
  case Zero    => result
  case Succ(n) => peano2int (n, result+1) // tail-recursive

val q = Succ(Succ(Succ(Zero))) // val q: Peano = ...
peano2int(q) // : Int = 3
```

Exercise: Binary Operations Tree

- Data stored at leaves
- Operations stored at internal nodes
- Internal nodes have left and right subtrees
- Algebraic data type

```
1 enum Tree[X]:
2   case Leaf (data:X)
3   case Node (l:Tree[X], f:(X,X)=>X, r:Tree[X])
```

- ❓ Fold tree into result by applying all the intermediate operations
- Recursive with pattern matching

```
1 def fold [X] (t: Tree[X]) : X = t match
2   case Leaf(x) => x
3   case Node(l, f, r) => f(fold(l), fold(r))
```