

CSC 447 - Concepts of Programming Languages

Algebraic Data Types

Instructor: Eric J. Fredericks



Learning Objectives

- ❓ How do we represent complex user-defined data types that support pattern matching?
 - 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
- Discriminated union, variants, subclasses

```
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
 - $X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\}$

- 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`)

```
1 case class C (x:Int, y:String)
2 val c:C = C (5, "hello")
3 val a:Int = c.x
4 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))
```



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
```




Implement a Language in Scala

- Implement **Grammar** to represent abstract syntax tree as an algebraic data type

```
1 enum Expr:
2   // Arithmetic expressions
3   case Number(n:Int)
4   case Plus(l:Expr, r:Expr)
5   case Minus(l:Expr, r:Expr)
6   case Times(l:Expr, r:Expr)
7
8   // Identifiers a,...,x,y,z
9   case Ident(c:Char)
10
11   // Comparisons l>=r
12   case GEq(l:Expr, r:Expr)
13
14   // Parenthesized expressions (e)
15   case Par(e:Expr)
```

```
1   // Programs
2
3   // Assignment x := v
4   case Assign(x:Ident, v:Expr)
5
6   // Conditional if p then t else e
7   case If(p:Expr, t:Expr, e:Expr)
8
9   // Loop while p do b
10  case While(p:Expr, b:Expr)
11
12  // Sequential composition p ; r
13  case Seq(p:Expr, r:Expr)
14 end Expr
```



Implement a Language in Scala

- Express a program as an abstract syntax tree: absolute value of an expression

```
1 val abs = (n: Expr) =>
2   Seq(
3     Assign(Ident('i'), n),           // i := <n>
4     If(                               // if
5       GEq(Ident('i'), Number(0)),    //   i >= 0
6       Assign(Ident('i'), Ident('i')), //   then i := i
7       Assign(Ident('i'), Minus(Number(0), Ident('i')))) //   else i := 0-i
8   )                                   // fi
9 )
```



Implement a Language in Scala

- Implement an interpreter following the [Formal Semantics](#)
- Define a store for the values of variables

```
1 type Store = Map[Char, Int]
```

- Define the interpreter signature (follows from shape of judgments:

$$\langle e, \xi \rangle \Downarrow \langle v, \xi' \rangle$$

```
1 def eval(e: Expr, s: Store): (Int, Store)
```



Implement a Language in Scala

- Reduction rules for integer expressions

- Numbers

$$\frac{}{\langle n, \xi \rangle \Downarrow \langle n, \xi \rangle} \text{ (Num)}$$

- Addition etc.

$$\frac{\langle e_1, \xi_0 \rangle \Downarrow \langle v_1, \xi_1 \rangle \quad \langle e_2, \xi_1 \rangle \Downarrow \langle v_2, \xi_2 \rangle}{\langle e_1 + e_2, \xi_0 \rangle \Downarrow \langle v_1 + v_2, \xi_2 \rangle} \text{ (Add), } \dots$$

- Parentheses

$$\frac{\langle e, \xi \rangle \Downarrow \langle v, \xi_1 \rangle}{\langle (e), \xi \rangle \Downarrow \langle v, \xi_1 \rangle} \text{ (Par)}$$

```
1 def eval(e: Expr, s: Store): (Int, Store) = e match
2   case Number(n)      => (n, s)
3
4   case Plus(e1, e2)   =>
5     val (v1, s1) = eval(e1, s)
6     val (v2, s2) = eval(e2, s1)
7     (v1+v2, s2)
8
9   ...
10
11  case Par(e)          => eval(e, s)
```



Implement a Language in Scala

- Reduction rules for assignment

- Variable lookup

$$\frac{}{\langle x, \xi \rangle \Downarrow \langle \xi(x), \xi \rangle} \text{ (Var)}$$

- Assignment

$$\frac{\langle e, \xi_0 \rangle \Downarrow \langle v, \xi_1 \rangle}{\langle x := e, \xi_0 \rangle \Downarrow \langle v, \xi_1 \{x \mapsto v\} \rangle} \text{ (:=)}$$

- Sequential expression composition

$$\frac{\langle e_1, \xi_0 \rangle \Downarrow \langle v_1, \xi_1 \rangle \quad \langle e_2, \xi_1 \rangle \Downarrow \langle v_2, \xi_2 \rangle}{\langle e_1 ; e_2, \xi_0 \rangle \Downarrow \langle v_2, \xi_2 \rangle} \text{ (;)}$$

```
1 def eval(e: Expr, s: Store): (Int, Store) = e match
2   ...
3
4   case Ident(c)          => (s(c), s)
5
6   case Assign(Ident(x), e) =>
7     val (v, s1) = eval(e, s)
8     (v, s1.updated(x, v))
9
10  case Seq(e1, e2)      =>
11    val (v1, s1) = eval(e1, s)
12    val (v2, s2) = eval(e2, s1)
13    (v2, s2)
```



Implement a Language in Scala

Reduction rules for conditionals and loops

- Comparison

$$\frac{\langle e_1, \xi_0 \rangle \Downarrow \langle v_1, \xi_1 \rangle \quad \langle e_2, \xi_1 \rangle \Downarrow \langle v_2, \xi_2 \rangle}{\langle e_1 \geq e_2, \xi_0 \rangle \Downarrow \langle \max(0, v_1 - v_2 + 1), \xi_2 \rangle} (\geq)$$

- Conditional

$$\frac{\langle e_1, \xi_0 \rangle \Downarrow \langle v_1, \xi_1 \rangle \quad v_1 \neq 0 \quad \langle e_2, \xi_1 \rangle \Downarrow \langle v_2, \xi_2 \rangle}{\langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi}, \xi_0 \rangle \Downarrow \langle v_2, \xi_2 \rangle} (\text{iftrue})$$
$$\frac{\langle e_1, \xi_0 \rangle \Downarrow \langle v_1, \xi_1 \rangle \quad v_1 = 0 \quad \langle e_3, \xi_1 \rangle \Downarrow \langle v_2, \xi_2 \rangle}{\langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi}, \xi_0 \rangle \Downarrow \langle v_2, \xi_2 \rangle} (\text{iffalse})$$

- Loop

$$\frac{\langle e_1, \xi_0 \rangle \Downarrow \langle v_1, \xi_1 \rangle \quad v_1 = 0}{\langle \text{while } e_1 \text{ do } e_2 \text{ od}, \xi_0 \rangle \Downarrow \langle 0, \xi_1 \rangle} (\text{whileend})$$
$$\frac{\langle e_1, \xi_0 \rangle \Downarrow \langle v_1, \xi_1 \rangle \quad v_1 \neq 0 \quad \langle e_2; \text{while } e_1 \text{ do } e_2 \text{ od}, \xi_1 \rangle \Downarrow \langle v_2, \xi_2 \rangle}{\langle \text{while } e_1 \text{ do } e_2 \text{ od}, \xi_0 \rangle \Downarrow \langle v_2, \xi_2 \rangle} (\text{whilerec})$$

```
1 def eval(e: Expr, s: Store): (Int, Store) = e match
2   ...
3
4   case GEq(e1, e2) =>
5     val (v1, s1) = eval(e1, s)
6     val (v2, s2) = eval(e2, s1)
7     (math.max(0, v1 - v2 + 1), s2)
8
9   case If(e1, e2, e3) =>
10    val (v1, s1) = eval(e1, s)
11    if v1 != 0
12      then eval(e2, s1)
13      else eval(e3, s1)
14
15   case w@While(e1, e2) =>
16     val (v1, s1) = eval(e1, s)
17     if v1 == 0
18       then (0, s1)
19       else eval(Seq(e2, w), s1)
```



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
```

- 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`

```
1 enum PeanoNat:  
2   case Zero  
3   case Succ(n: PeanoNat)
```

- Evaluate `PeanoNat` to `Int`

```
1 import PeanoNat.*  
2 def peano2int (p: PeanoNat, result: Int = 0): Int = p match  
3   case Zero      => result  
4   case Succ(n)   => peano2int (n, result+1) // tail-recursive  
5  
6 val q = Succ(Succ(Succ(Zero))) // val q: Peano = ...  
7 peano2int(q) // : Int = 3
```




Exercise: Linked List

? 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)
```



Exercise: Linked List

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

```
1 enum MyList:  
2   case Empty  
3   case Cons (head:Int, tail:MyList)  
4 object MyList:  
5   def create(elems: Int*) =  
6     elems.foldRight(Empty)((e, l) => Cons(e, l))
```

- ? Create an instance of a list?
- Use method `create`

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



Exercise: Linked List

? Generalize to any element type?

```
1 enum MyList[+X]:  
2   case Empty  
3   case Cons (head:X, tail:MyList[X])
```

? Compute the length of such a list?

- Recursive with pattern matching

```
1 def length [X] (xs:MyList[X]): Int = xs match  
2   case MyList.Empty      => 0  
3   case MyList.Cons(a,as) => 1 + length(as)
```

- Tail-recursive with pattern matching

```
1 def length [X] (xs:MyList[X], result:Int = 0): Int = xs match  
2   case MyList.Empty      => result  
3   case MyList.Cons(a,as) => length(as, result+1)
```



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))
```