CSC 347 - Concepts of Programming Languages

Algebraic Data Types

Instructor: Stefan Mitsch

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

```
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

```
enum Expr:
  // Arithmetic expressions
  case Number(n:Int)
  case Plus(l:Expr, r:Expr)
  case Minus(l:Expr, r:Expr)
  case Times(l:Expr, r:Expr)

  // Identifiers a,...,x,y,z
  case Ident(c:Char)

  // Comparisons l>=r
  case GEq(l:Expr, r:Expr)

  // Parenthesized expressions (e)
  case Par(e:Expr)
```

```
  // Programs

  // Assignment x := v
  case Assign(x:Ident, v:Expr)

  // Conditional if p then t else e
  case If(p:Expr, t:Expr, e:Expr)

  // Loop while p do b
  case While(p:Expr, b:Expr)

  // Sequential composition p ; r
  case Seq(p:Expr, r:Expr)
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: $\langle \mathbf{e}, \xi \rangle \Downarrow \langle v, \xi' \rangle$)

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

Implement a Language in Scala

- Reduction rules for integer expressions
- Numbers

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

- Addition etc.

$$\frac{\langle \mathtt{e}_1, \xi_0 \rangle \Downarrow \langle v_1, \xi_1 \rangle \quad \langle \mathtt{e}_2, \xi_1 \rangle \Downarrow \langle v_2, \xi_2 \rangle}{\langle \mathtt{e}_1 \mathtt{+} \mathtt{e}_2, \xi_0 \rangle \Downarrow \langle v_1 + v_2, \xi_2 \rangle}\ (\text{Add}), \ldots$$

- Parentheses

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

```scala
def eval(e: Expr, s: Store): (Int, Store) = e match
  case Number(n)      => (n, s)

  case Plus(e1, e2)   =>
    val (v1, s1) = eval(e1, s)
    val (v2, s2) = eval(e2, s1)
    (v1+v2, s2)

  ...

  case Par(e)         => eval(e, s)
```

Implement a Language in Scala

- Reduction rules for assignment
- Variable lookup

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

- Assignment

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

- Sequential expression composition

$$\frac{\langle \mathtt{e}_1,\xi_0\rangle \Downarrow \langle v_1,\xi_1\rangle \qquad \langle \mathtt{e}_2,\xi_1\rangle \Downarrow \langle v_2,\xi_2\rangle}{\langle \mathtt{e}_1\,;\mathtt{e}_2,\xi_0\rangle \Downarrow \langle v_2,\xi_2\rangle}\ (;)$$

```scala
def eval(e: Expr, s: Store): (Int, Store) = e match
  ...

  case Ident(c)       => (s(c), s)

  case Assign(Ident(x), e)   =>
    val (v,s1) = eval(e, s)
    (v, s1.updated(x,v))

  case Seq(e1, e2)    =>
    val (v1, s1) = eval(e1, s)
    val (v2, s2) = eval(e2, s1)
    (v2, s2)
```

Implement a Language in Scala

Reduction rules for conditionals and loops

- Comparison

$$\frac{\langle \mathtt{e}_1,\xi_0\rangle \Downarrow \langle v_1,\xi_1\rangle \qquad \langle \mathtt{e}_2,\xi_1\rangle \Downarrow \langle v_2,\xi_2\rangle}{\langle \mathtt{e}_1\mathtt{>=}\mathtt{e}_2,\xi_0\rangle \Downarrow \langle \max(0,v_1-v_2+1),\xi_2\rangle}\ (\geq)$$

- Conditional

$$\frac{\langle \mathtt{e}_1,\xi_0\rangle \Downarrow \langle v_1,\xi_1\rangle \qquad v_1\neq 0 \qquad \langle \mathtt{e}_2,\xi_1\rangle \Downarrow \langle v_2,\xi_2\rangle}{\langle \mathtt{if\ e}_1\ \mathtt{then\ e}_2\ \mathtt{else\ e}_3\ \mathtt{fi},\xi_0\rangle \Downarrow \langle v_2,\xi_2\rangle}\ (\text{iftrue})$$

$$\frac{\langle \mathtt{e}_1,\xi_0\rangle \Downarrow \langle v_1,\xi_1\rangle \qquad v_1= 0 \qquad \langle \mathtt{e}_3,\xi_1\rangle \Downarrow \langle v_2,\xi_2\rangle}{\langle \mathtt{if\ e}_1\ \mathtt{then\ e}_2\ \mathtt{else\ e}_3\ \mathtt{fi},\xi_0\rangle \Downarrow \langle v_2,\xi_2\rangle}\ (\text{iffalse})$$

- Loop

$$\frac{\langle \mathtt{e}_1,\xi_0\rangle \Downarrow \langle v_1,\xi_1\rangle \qquad v_1= 0}{\langle \mathtt{while\ e}_1\ \mathtt{do\ e}_2\ \mathtt{od},\xi_0\rangle \Downarrow \langle 0,\xi_1\rangle}\ (\text{whileend})$$

$$\frac{\langle \mathtt{e}_1,\xi_0\rangle \Downarrow \langle v_1,\xi_1\rangle \qquad v_1\neq 0 \qquad \langle \mathtt{e}_2\mathtt{;while\ e}_1\ \mathtt{do\ e}_2\ \mathtt{od},\xi_1\rangle \Downarrow \langle v_2,\xi_2\rangle}{\langle \mathtt{while\ e}_1\ \mathtt{do\ e}_2\ \mathtt{od},\xi_0\rangle \Downarrow \langle v_2,\xi_2\rangle}\ (\text{whilerec})$$

```scala
def eval(e: Expr, s: Store): (Int, Store) = e match
  ...

  case GEq(e1, e2)    =>
    val (v1, s1) = eval(e1, s)
    val (v2, s2) = eval(e2, s1)
    (math.max(0,v1-v2+1), s2)

  case If(e1, e2, e3) =>
    val (v1, s1) = eval(e1, s)
    if v1!=0
      then eval(e2, s1)
      else eval(e3, s1)

  case w@While(e1, e2)  =>
    val (v1, s1) = eval(e1, s)
    if v1==0
      then (0, s1)
      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

```scala
enum MyList:
  case Empty
  case Cons (head:Int, tail:MyList)
object MyList:
  def create(elems: Int*) =
    elems.foldRight(Empty)((e, l) => Cons(e, l))
```

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

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