Concepts of Programming Languages
Lecture Notes: Week 5—Algebraic Data Types

Stefan Mitsch

School of Computing, DePaul University
smitsch@depaul.edu

Algebraic Data Types (80min)

Algebraic data types combine *product types* and *sum types* into sums of products. Let's now take a closer look at these two concepts.

In previous lectures, we have discussed tuples as an example of *product types*. Product types combine a fixed number of heterogeneous elements and draw their name from the Cartesian product of sets:

$$X * Y = \{(x, y) \mid x \in X \land y \in Y\} \ .$$

Another example for product types are Scala case classes, for example a product of a number and a string is below:

```
1  case class C (x: Int, y: String)
2  val c = C(5, "hello") // "new" optional when creating
       instances of case classes
3  val n = c match
4    case C(a, _) => a
```

Case classes enjoy special compiler treatment in Scala. The constructor arguments are turned into visible immutable fields, we get a sensible `toString`, `==`, and `hashCode` implementation. We also get a *companion object* with factory methods for constructing instances, and as we have seen in the example above, convenient pattern matching support through generated `unapply` extractor methods. Pairs and tuples are builtin syntactic sugar for case classes in Scala.

Sum types, in contrast, combine alternatives, such as in a discriminated or tagged union or variants. An example of a sum type is the Scala `Either` type. Mathematically, sum types correspond to the union of sets:

$$X \cup Y = \{z \mid z \in X \lor z \in Y\} \ .$$

When we want to keep track of the source of an element in the resulting union, we can also think of sum types as a coproduct or disjoint union of sets:

$$X \oplus Y = \{(0, x) \mid x \in X\} \cup \{(1, y) \mid y \in Y\} \ .$$

In a coproduct, the elements are tagged to indicate their source. In Scala, we can create sum types by creating a type hierarchy; Scala 2 explicitly asks for a type hierarchy, which might be familiar from Java, Scala 3 provides nice syntactic support in the form of `enum`.

```
// Scala 3
enum Color:
  case Blue
  case White

// Scala 2
final trait Color
case object Blue extends Color
case object White extends Color
```

More useful are enum constructs that define case classes:

```
enum Expr:
  case Number(x: Int)
  case Plus(l: Expr, r: Expr)
  // ...
```

Again, we use pattern matching to decompose a sum type into its elements.

```
// create instances
val three = Number(3)
val e = Plus(three, Number(5))

// decompose with pattern matching
def eval(e: Expr) : Int = e match
  case Number(n)  => n
  case Plus(l, r) => eval(l) + eval(r)
  // ...
```

Other programming languages also support sum types, but often in a less convenient way. For example, in C union types must be tagged manually.

```
struct s_absolute_t {
  int year;
  int mon;
  int day;
};

struct s_relative_t {
  int days_offset;
};

union u_ds_t {
  struct s_absolute_t u_absolute;
  struct s_relative_t u_relative;
};

// create instances, must supply tags!
struct ds_t ds[2];
ds[0].tag = e_absolute;
ds[0].content.u_absolute.year = 2030;
ds[0].content.u_absolute.mon  = 0;
```

```
21 ds[0].content.u_absolute.day   = 1;
22 ds[1].tag = e_relative;
23 ds[1].content.u_relative.days_offset = -5;
24
25 // examine tag to decompose
26 void print_ds (struct ds_t *dsp) {
27   switch (dsp->tag) {
28   case e_absolute:
29     printf ("absolute (%d, %d, %d)\n", dsp->content.u_absolute.
         year,
30                                        dsp->content.u_absolute.
                                             mon,
31                                        dsp->content.u_absolute.
                                             day);
32     break;
33   case e_relative:
34     printf ("relative (%d)\n", dsp->content.u_relative.
         days_offset);
35     break;
36   default:
37     fprintf (stderr, "Unknown tag\n");
38     exit (1);
39   }
40 }
```

Algebraic data types can be recursive. For example, an implementation of Peano arithmetic (natural numbers that are defined as successors of 0) is given below.

```
1  enum PeanoNat:
2    case Zero
3    case Succ(n: PeanoNat)
4
5  def peano2int(p: PeanoNat) : Int = p match
6    case PeanoNat.Zero     => 0
7    case PeanoNat.Succ(n) => 1 + peano2int (n)
8
9  import PeanoNat.*
10 val q = Succ(Succ(Succ(Zero))) // val q: Peano = ...
11 peano2int(q) // : Int = 3
```

Algebraic Data Type for Lists In a similar fashion, we can reconstruct the Scheme way of building lists with empty lists and cons cells.

```
1  enum MyList[+X]:
2    case Empty
3    case Cons (head:X, tail:MyList[X])
4
5  def length [X] (xs:MyList[X]): Int = xs match
6    case MyList.Empty => 0
7    case MyList.Cons(a,as) => 1 + length(as)
```

```
8
9  import MyList.*
10 val xs = Cons(11, Cons(21, Cons(31, Empty))) // : MyList[Int] =
       ...
11 length(xs)     // : Int = 3
12 val ys = Empty // : MyList[Nothing] = ...
```

Algebraic Data Type for Trees With algebraic data types, we also can define trees. Below is an example of a tree that uses leafs as delimiters but stores all data in inner nodes.

```
1 enum Tree[+X]:
2   case Leaf
3   case Node(l:Tree[X], c:X, r:Tree[X])
```

Now let's extend that tree to a red-black tree and implement the rotate-left function.

```
1 enum Color { case Red, Black }
2 enum RBTree[+K,+V]:
3   case Leaf
4   case Node(k:K, v:V, c:Color, l:RBTree[K,V], r:RBTree[K,V])
5
6 import RBTree.*;
7 def rotateLeft[K,V](t:Node[K,V]) : Node[K,V] =
8   t match
9     case Node(k1, v1, c, l, Node(k2, v2, Color.Red, m, r)) =>
10      Node(k2, v2, c, Node(k1, v1, Color.Red, l, m), r)
```