

Concepts of Programming Languages

Lecture Notes: 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:

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

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:

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

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`.

```

1 // Scala 3
2 enum Color:
3   case Blue
4   case White
5
6 // Scala 2
7 final trait Color
8 case object Blue extends Color
9 case object White extends Color

```

More useful are enum constructs that define case classes:

```

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

```

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

```

1 // create instances
2 val three = Number(3)
3 val e = Plus(three, Number(5))
4
5 // decompose with pattern matching
6 def eval(e: Expr) : Int = e match
7   case Number(n) => n
8   case Plus(l, r) => eval(l) + eval(r)
9   // ...

```

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

```

1 struct s_absolute_t {
2   int year;
3   int mon;
4   int day;
5 };
6
7 struct s_relative_t {
8   int days_offset;
9 };
10
11 union u_ds_t {
12   struct s_absolute_t u_absolute;
13   struct s_relative_t u_relative;
14 };
15
16 // create instances, must supply tags!
17 struct ds_t ds[2];
18 ds[0].tag = e_absolute;
19 ds[0].content.u_absolute.year = 2030;
20 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.
30                 year,
31                 dsp->content.u_absolute.
32                     mon,
33                     dsp->content.u_absolute.
34                         day);
35             break;
36         case e_relative:
37             printf ("relative (%d)\n", dsp->content.u_relative.
38                 days_offset);
39             break;
40         default:
41             fprintf (stderr, "Unknown tag\n");
42             exit (1);
43     }
44 }

```

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 peanozint(p: PeanoNat) : Int = p match
6     case PeanoNat.Zero => 0
7     case PeanoNat.Succ(n) => 1 + peanozint (n)
8
9 import PeanoNat.*
10 val q = Succ(Succ(Succ(Zero))) // val q: Peano = ...
11 peanozint(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)

```