CSC 347 - Concepts of Programming Languages

Algebraic Data Types

Instructor: Stefan Mitsch

Learning Objectives

❓ How do we represent complex user-defined data types?

- Understand algebraic data types
- Understand Scala enum classes

Algebraic Data Types

- Product types: tuples
- Sum types: discriminated/tagged union, variants
- *Algebraic data types*: sum of products
- Examples seen so far
  - `Option` type
  - `List` type
- Decompose values of algebraic data types with pattern matching

Algebraic Data Types

- Algebraic Data Types (wikipedia) in PLs
  - D
  - F#
  - Haskell
  - Julia
  - Hope (1970s, first PL with this feature)
  - ML
  - OCaml
  - Rust
  - Scala
  - Swift

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`

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

- `new` unnecessary for constructing instances

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

- Extract elements with pattern matching

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

Scala Case Classes

- Compiler treatment for `case` classes
- Constructor arguments are visible and immutable

```
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
```

- Generate sensible `toString` implementation
- Generate companion object with `apply` method
  - used to construct instances
- Generate pattern matching support
  - see `unapply` method / extractors in textbook

Tuples are Case Classes

- Pairs / tuples are *syntactic sugar* for case classes
- See `Tuple3.scala` source

```
case class Tuple3[+T1, +T2, +T3](_1: T1, _2: T2, _3: T3)
    extends Product3[T1, T2, T3]:
  override def toString() = "(" + _1 + "," + _2 + "," + _3 + ")"
```

- Examine runtime type without syntactic sugar

```
scala> (5, "hello", true).getClass
res0: Class[_ <: (Int, String, Boolean)] = class scala.Tuple3
```

Set Union

- Cartesian product of sets
  $X \times Y = \{(x, y) \mid x \in X \land y \in Y\}$
- *Union* of sets
  $X \cup Y = \{z \mid z \in X \lor z \in Y\}$
- *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

Disjoint Union: Scala Enum

- Disjoint union of 3 ints and 1 int
  - Scala `enum` similar to Java interface

```
enum DateSpecifier:
  case Absolute (year:Int,mon:Int,day:Int)
  case Relative (daysOffset:Int)
```

- Create instances

```
val ds = new Array[DateSpecifier] (2)
ds (0) = DateSpecifier.Absolute (2030, 0, 1) // Months are 0.11
ds (1) = DateSpecifier.Relative (-5)
```

Disjoint Union: Scala Enum

- Pattern match to decompose

```
import java.util.{Calendar,Date}

def resolveDate (d:DateSpecifier) : Calendar =
  val b = Calendar.Builder()
  d match
    case DateSpecifier.Absolute (y, m, d) => b.setDate (y, m, d).build
    case DateSpecifier.Relative (o)       =>
      val c = b.build                     // Jan 1 1970, Gregorian calendar
      c.setTime (Date())                  // Today, Gregorian calendar
      c.add (Calendar.DAY_OF_YEAR, o)     // Add days offset
      c                                   // Return updated calendar object
```

Disjoint Union: C

- Union types in C

```c
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;
};
```

- … must be tagged manually

```c
enum e_ds_t {
  e_absolute,
  e_relative,
};

struct ds_t {
  enum e_ds_t tag;
  union u_ds_t content;
};
```

Disjoint Union: C

- Create instances: tag / union selector must match!

```
struct ds_t ds[2];
ds[0].tag = e_absolute;
ds[0].content.u_absolute.year = 2030;
ds[0].content.u_absolute.mon  = 0;
ds[0].content.u_absolute.day  = 1;
ds[1].tag = e_relative;
ds[1].content.u_relative.days_offset = -5;
```

Disjoint Union: C

- Examine tag to decompose: only access union selector matching tag!

```
void print_ds (struct ds_t *dsp) {
  switch (dsp->tag) {
  case e_absolute:
    printf ("absolute (%d, %d, %d)\n", dsp->content.u_absolute.year,
                                       dsp->content.u_absolute.mon,
                                       dsp->content.u_absolute.day);
    break;
  case e_relative:
    printf ("relative (%d)\n", dsp->content.u_relative.days_offset);
    break;
  default:
    fprintf (stderr, "Unknown tag\n");
    exit (1);
  }
}
```

Recursive Types

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

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

- Define functions between `PeanoNat` and `Int`

```
def peano2int (p:PeanoNat): Int = p match
  case PeanoNat.Zero    => 0
  case PeanoNat.Succ(n) => 1 + peano2int (n)

import PeanoNat.*
val q = Succ (Succ (Succ (Zero))) // : Peano = ...
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

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

Exercise: Linked List

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

❓ Create an empty list?

- Simply use `Empty`

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

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

```
import MyList.*
val xs = Cons (11, Cons(21, Cons(31, Empty)))
// xs : MyList[Int] = ...
```

Exercise: Linked List

❓ Generalize to any element type?

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

❓ Compute the length of such a list?

- Recursive with pattern matching

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

Exercise: Binary Tree

- Data stored at leaves
- Operations stored at internal nodes
- Internal nodes have left and right subtrees
-
```
enum Tree[X]:
  case Leaf (data:X)
  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

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

Summary

- Algebraic data types: Sums of products
- In Scala: `enum` and `case` classes