CSC 347 - Concepts of Programming Languages

Algebraic Data Types

Instructor: Stefan Mitsch

Pattern Matching with Complex Datatypes

- Recursively traverse a data structure

```
1 abstract class Node
2 class Val(val x: Int) extends Node
3 class Op(val l: Node, val r: Node, val f: (Int, Int)=>Int) extends Node
4
5 def eval(node: Node) : Int = node match
6   case n: Val => n.x
7   case n: Op =>
8     val l = eval(n.l)
9     val r = eval(n.r)
10    f(l, r)
```

- Why do we not have support to decompose objects into their components?

```
1 def eval(node: Node) : Int = node match
2   case Val(x) => x
3   case Op(l, r) => f(eval(l), eval(r))
```

Learning Objectives

❓ How do we represent complex user-defined data types that support pattern matching?

- Understand algebraic data types
- Understand Scala enum classes

Algebraic Data Types

- Product types: *combine multiple data elements* into one unit (tuples and classes)
- Sum types: *distinguish between multiple alternatives* (discriminated union, variants, subclasses)
- *Algebraic data types*: sum of products
- 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`

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

- `new` unnecessary for constructing instances

```
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

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 a value and a binary operator
- Scala `enum` similar to Java interface

```
1 enum Node:
2   case Val(val x: Int)
3   case Op(val l: Node, r: Node, val f: (Int, Int)=>Int)
```

- Definition can be recursive

- Create instances

```
1 val three = Val(3)
2 val op = Op(three, Val(5), _ + _)
```

Disjoint Union: Scala Enum

- Pattern match to decompose

```
1 def eval(node: Node) : Int = node match
2   case Val(x)    => x
3   case Op(l, r) => f(eval(l), eval(r))
4
5 // (3+5)*2
6 eval(Op(Op(Val(3), Val(5), _ + _), Val(2), _ * _))
```

Disjoint Union: C

- Union types in C

```c
struct s_val_t;
struct s_op_t;

union u_node_t {
  struct s_val_t* u_val;
  struct s_op_t* u_op;
};

struct s_val_t { int x; };

struct s_op_t {
  struct u_node_t l;
  struct u_node_t r;
  int (*op)(int,int);
};
```

⚡ What is stored in `u_node_t`?

- … must be tagged manually

```c
enum e_node_t {
  e_val,
  e_op,
};

struct node_t {
  enum e_node_t tag;
  union u_node_t content;
};

struct s_op_t {
  struct node_t l;
  struct node_t r;
  int (*op)(int,int);
};
```

Disjoint Union: C

- Create instances: tag / union selector must match!

```c
int add(int x, int y) { return x+y; }

struct s_val_t three = { .x=3 };
struct s_val_t five  = { .x=5 };
struct s_op_t op = {
  .l = {
    .tag = e_val,
    .content = { .u_val = &three }
  },
  .r = {
    .tag = e_val,
    .content = { .u_val = &five }
  },
  .op = &add
};

struct node_t node = {
  .tag = e_op,
  .content = { .u_op = &op }
};
```

Disjoint Union: C

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

```c
int eval (struct node_t* node) {
  switch (node->tag) {
    case e_val:
      return node->content.u_val->x;
    case e_op: {
      struct s_op_t* o = node->content.u_op;
      return o->op(eval(&o->l), eval(&o->r));
    }
    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`

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

- Define functions between `PeanoNat` and `Int`

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