Algebraic Data Types

Principles of Functional Programming

Algebraic Data Types

We have seen how heterogenous data is described by a hierarchy of case classes and objects.

Defining new data types is quite common in statically typed functional languages, so there exists a shorthand for it.

The shorthand generalizes the `enum` syntax we have seen earlier.

Example

```
enum Student {
  case Listening
  case Asking(question: String)
}
```

- This definition introduces a `Student` type consisting of two cases, `Listening` and `Asking`.
- The `Asking` case is parameterized with a value parameter `question`.
- Since `Listening` is not parameterized, it is treated as a normal enum value.

Matching on Constructors

Since `Asking` is not a constant, we match on it using a *constructor pattern*

```
student match
  case Student.Listening => "Student is listening"
  case Student.Asking(q) => s"Student is asking: $q"
```

A constructor pattern allows us to *extract* the value of the `question` parameter, in case the `student` value is indeed of type `Asking`.

Here, the `q` identifier is bound to the `question` parameter of the `student` object.

Relationship With Case Classes

Algebraic Data types expand to class hierarchies with case classes.

For instance, the `Student` type expands to something like:

```
abstract class Student
object Student {
  val Listening extends Student
  case class Asking(question: String) extends Student
}
```

Comparison With Object Oriented Decomposition

OO decomposition and algebraic data types are two ways of defining types and operations.

When should you use one or the other?

Use an algebraic data type to model

- a type with a fixed number of alternatives, where
- all alternatives are pure data types that do not contain methods.

Use a hierarchy with classes, if

- The set of alternatives is open, i.e. new alternatives can be added after the fact, or
- Alternatives are complex, consisting of methods as well as parameters.

Enumerations Can Be Recursive

A list of integer values can be modeled as either an empty list, or a node containing both a number and a reference to the remainder of the list.

```
enum List {
  case Empty
  case Node(value: Int, next: List)
}
```

A list with values `1`, `2`, and `3` can be constructed as follows:

```
List.Node(1, List.Node(2, List.Node(3, List.Empty)))
```

Examples of Lists

```
List(1, 2, 3)
```

```
List(List(true, false), List(3))
```

Exercise: Arithmetic Expressions

Define an algebraic datatype modeling arithmetic expressions.

An expression can be:

- a number (e.g. 42),
- a sum of two expressions,
- or, the product of two expressions.

Expr Type Definition

```
enum Expr {
  case Number(n: Int)
  case Sum(lhs: Expr, rhs: Expr)
  case Prod(lhs: Expr, rhs: Expr)
}
```

Expr Type Definition

```
enum Expr {
  case Number(n: Int)
  case Sum(lhs: Expr, rhs: Expr)
  case Prod(lhs: Expr, rhs: Expr)
}
```

```
val one = Expr.Number(1)   > one: Expr = Number(1)
val two = Expr.Number(2)   > two: Expr = Number(2)
Expr.Sum(one, two)         > Sum(Number(1), Number(2))
```

Exercise

Is the following match expression valid?

```
expr match {
  case Expr.Sum(x, x) => Expr.Prod(2, x)
  case e              => e
}
```

- 0 Yes
- 0 No

Exercise

Implement an `eval` operation that takes an `Expr` as parameter and evaluates its value:

```
def eval(e: Expr): Int
```

Examples of use:

```
eval(Expr.Number(42))                                          > 42
eval(Expr.Prod(2, Expr.Sum(Expr.Number(8), Expr.Number(13))))  > 42
```

Exercise

Implement a `show` operation that takes an `Expr` as parameter and returns a `String` representation of the operation.

Be careful to introduce parenthesis when necessary!

```
def show(e: Expr): String
```

Examples of use:

```
show(Expr.Number(42))                                          > "42"
show(Expr.Prod(2, Expr.Sum(Expr.Number(8), Expr.Number(13))))  > "2 * (8 + 13)"
```

Summary

In this lecture, we have seen:

- how to define data types accepting alternative values using `enum` definitions,
- enumeration cases can be discriminated using pattern matching,
- enumeration cases can take parameters or be simple values,
- pattern matching allows us to extract information carried by an object.