CS 2340 Objects and Design - Scala

Case Classes

Christopher Simpkins
chris.simpkins@gatech.edu

Arithmetic Expressions

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
                 left: Expr, right: Expr) extends Expr
```

- Each clase class extends either an abstract class or trait
- Bodies of classes are empty, typical of case classes
- Class parameters are implicitly defined as `val` fields (no `val` annotation required - save some typing)

Syntactic Conveniences 1/2

Factory Methods:

```
scala> val v = Var("x")
v: Var = Var(x)

scala> val op = BinOp("+", Number(1), v)
op: BinOp = BinOp(+,Number(1.0),Var(x))
```

Implicit field definition (`val`s):

```
scala> v.name
res0: String = x

scala> op.left
res1: Expr = Number(1.0)
```

Syntactic Conveniences 2/2

"Naturally" defined `toString` and `==` in terms of `equals`:

```
scala> println(op)
BinOp(+,Number(1.0),Var(x))

scala> op.right == Var("x")
res3: Boolean = true
```

`copy` method:

```
scala> op.copy(operator = "-")
res4: BinOp = BinOp(-,Number(1.0),Var(x))
```

Simplifying Arithmetic Expressions

Say we want to simplify arithmetic expressions.

```
UnOp("-", UnOp("-", e)) => e  // Double negation
BinOp("+", e, Number(0)) => e // Adding zero
BinOp("*", e, Number(1)) => e // Multiplying by one
```

In other words, given an expression of one of the forms on the left, we want to convert it to an expresiosn of the corresponding right-hand side. This is easy with pattern matching.

A simplifyTop Function

Given the following function, which is essentially the conversion rules we defined earlier couched in some minimal syntax:

```
def simplifyTop(expr: Expr): Expr = expr match {
  case UnOp("-", UnOp("-", e)) => e // Double negation
  case BinOp("+", e, Number(0)) => e // Adding zero
  case BinOp("*", e, Number(1)) => e // Multiplying by one
  case _ => expr
}
```

We can simplify arithmetic expressions like this:

```
scala> simplifyTop(UnOp("-", UnOp("-", Var("x"))))
res4: Expr = Var(x)
```

`match` Expressions

- General form: *selector* `match` { *alternatives* }
- Alternatives: *pattern* => *expression*
- Selector is matched against each pattern sequentially until a match is found.
- Expression corresponding to matched pattern is evaluated and returned as value of the `match` expression
- No fall through to subsequent alternatives
- _ is used as a default if no other patterns match

```
def simplifyTop(expr: Expr): Expr = expr match {
  case UnOp("-", UnOp("-", e)) => e // Double negation
  case BinOp("+", e, Number(0)) => e // Adding zero
  case BinOp("*", e, Number(1)) => e // Multiplying by one
  case _ => expr
}
```

Wildcard Patterns

Wildcard pattern matches any object. Can be used for defaults:

```
expr match {
  case BinOp(op, left, right) =>
    println(expr +" is a binary operation")
  case _ =>
}
```

... or to ignore parts of patterns:

```
expr match {
  case BinOp(_, _, _) => println(expr +" is a binary operation")
  case _ => println("It's something else")
}
```

Constant Patterns

Constant patterns match their values:

```
def describe(x: Any) = x match {
  case 5 => "five"
  case true => "truth"
  case "hello" => "hi!"
  case Nil => "the empty list"
  case _ => "something else"
}
```

```
scala> describe(5)
res6: java.lang.String = five

scala> describe(true)
res7: java.lang.String = truth

scala> describe(Nil)
res9: java.lang.String = the empty list

scala> describe(List(1,2,3))
res10: java.lang.String = something else
```

Variable Patterns

Variable patterns match any object, like a widlcard, but bind the variable name to the object:

```
expr match {
  case 0 => "zero"
  case somethingElse => "not zero: "+ somethingElse
}
```

Note that some constants look like variables:

```
scala> import math.{E, Pi}
import math.{E, Pi}

scala> E match {
  case Pi => "strange math? Pi = "+ Pi
  case _ => "OK"
}
res11: java.lang.String = OK
```

`E` didn't match the constant `Pi`

Gotcha: Variable-Constant Disambiguation

Simple names starting with lowercase letters treated as variable patterns.

Here `pi` is a variable pattern, not a constant:

```
scala> val pi = math.Pi
pi: Double = 3.141592653589793
scala> E match {
  case pi => "strange math? Pi = "+ pi
}
res12: java.lang.String = strange math? Pi = 2.718281828459045
```

In fact, with a variable pattern like this you can't even add a default alternative because the variable pattern is exhaustive:

```
scala> E match {
  case pi => "strange math? Pi = "+ pi
  case _ => "OK"
}
<console>:9: error: unreachable code
         case _ => "OK"
                   ^
```

Constructor Patterns

- A constructor pattern consists of a name and patterns within parentheses
- Name should be the name of a case class, the names in parentheses can be any kind of pattern (including other case classes!)
- Nesting permits powerful *deep matches*

```
expr match {
  case BinOp("+", e, Number(0)) => e // a deep match
  case _ => expr
}
```

Sequence Patterns

Match a list of length three with 0 as first element and return second element as the value of the match expression:

```
expr match {
  case List(0, e, _) => e
  case _ => null
}
```

Match a list of any length greater than 1 with 0 as first element and return second element as the value of the match expression:

```
expr match {
  case List(0, e, _*) => e
  case _ => null
}
```

Tuple Patterns

A pattern like `(a, b, c)` matches an arbitrary 3-tuple. Given:

```
def tupleDemo(expr: Any) = expr match {
  case (a, b, c) => println("matched "+ a + b + c)
  case _ =>
}
```

You can pick apart a 3-tuple:

```
scala> tupleDemo(("a ", 3, "-tuple"))
matched a 3-tuple

scala> tupleDemo(("foo", "bar", "baz"))
matched foobarbaz
```

Typed Patterns

Typed patterns are convenient replacement for type tests and type casts. Given:

```
def generalSize(x: Any) = x match {
  case s: String => s.length
  case m: Map[_, _] => m.size
  case _ => -1
}
```

You can get the size of objects of various types:

```
scala> generalSize("abc")
res16: Int = 3

scala> generalSize(Map(1 -> 'a', 2 -> 'b'))
res17: Int = 2

scala> generalSize(math.Pi)
res18: Int = -1
```

Gotcha: Type Erasure of Generics

Run this with `scala -unchecked` to get details of unchecked warnings

```
scala> def isIntIntMap(x: Any) = x match {
  case m: Map[Int, Int] => true
  case _ => false
}
<console>:5: warning: non variable type-argument Int in
type pattern is unchecked since it is eliminated by erasure
         case m: Map[Int, Int] => true
                 ^
```

That warning means you'll get suprising behavior due to type parameter erasure:

```
scala> isIntIntMap(Map(1 -> 1))
res19: Boolean = true

scala> isIntIntMap(Map("abc" -> "abc"))
res20: Boolean = true
```

This limitation is inherited from the JVM. Note that you can match the

Nested Pattern Variable Binding

In addition to simple variable binding, you can bind a variable to a matched nested pattern using *variable* `@` before the pattern:

```
expr match {
  case UnOp("abs", e @ UnOp("abs", _)) => e
  case _ =>
}
```

The code above matches double applications of the `abs` operator and simplifies them by returning an equivalent single aplication (which is just the inner pattern).

Pattern Guards

What if we wanted to convert an addition of a number to itself to a multiplication of the number by two? Can't do it with only syntactic pattern matching:

```
scala> def simplifyAdd(e: Expr) = e match {
  case BinOp("+", x, x) => BinOp("*", x, Number(2))
  case _ => e
}
<console>:11: error: x is already defined as value x
         case BinOp("+", x, x) => BinOp("*", x, Number(2))
                            ^
```

Pattern guards allow us to add simple semantic checks to patterns:

```
scala> def simplifyAdd(e: Expr) = e match {
  case BinOp("+", x, y) if x == y => BinOp("*", x, Number(2))
  case _ => e
}
simplifyAdd: (e: Expr)Expr
```

Sealed Classes

Seal a class by adding the `sealed` keyword to root class definition:

```
sealed abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr) extends
    Expr
```

Compiler will then ensure that all pattern matches are exhaustive:

```
def describe(e: Expr): String = e match {
  case Number(_) => "a number"
  case Var(_) => "a variable"
}
```

Above code will generate compiler warnings:

```
warning: match is not exhaustive!
missing combination UnOp
missing combination BinOp
```

The `Option` Type

Takes the form `Option[T]` and has two values:

- `Some(x)` where `x` is a value of type `T`, or
- `None`, an object which represents a missing value.

Typically used with pattern matching. The `get` method on `Map` returns an `Option[T]`:

```
scala> val capitals = Map("France" -> "Paris", "Japan" -> "Tokyo")
...
scala> def show(x: Option[String]) = x match {
  case Some(s) => s
  case None => "?"
}
...
scala> show(capitals get "Japan")
res25: String = Tokyo

scala> show(capitals get "North Pole")
res27: String = ?
```

Better than returning `null` because in, for example, Java's collections, you have to remember which collections may return `null`'s, where in Scala this is made explicit and checked by the compiler.

Patterns in Variable Definitions

You can use pattern patching to

- take apart tuples for multiple assignment, or
- deconstruct a case class instance.

```
scala> val myTuple = (123, "abc")
myTuple: (Int, java.lang.String) = (123,abc)

scala> val (number, string) = myTuple
number: Int = 123
string: java.lang.String = abc

scala> val exp = new BinOp("*", Number(5), Number(1))
exp: BinOp = BinOp(*,Number(5.0),Number(1.0))

scala> val BinOp(op, left, right) = exp
op: String = *
left: Expr = Number(5.0)
right: Expr = Number(1.0)
```

Case Sequences as Partial Functions

A sequence of cases can be used anywhere a function literal can be used because a case sequence is a special kind function literal.

- Each case is an entry point with its own list of parameters specified by the pattern.
- The body of each entry point is the right-hand side of the case.

```
val withDefault: Option[Int] => Int = {
  case Some(x) => x
  case None => 0
}
```

`withDefault` is a `val` of type `Option[Int] => Int` – a function type – and its value is a sequence of cases.

```
scala> withDefault(Some(10))
res28: Int = 10

scala> withDefault(None)
res29: Int = 0
```

Case Sequences in the Scala Library

Case sequences in Actors:

```
react {
  case (name: String, actor: Actor) => {
    actor ! getip(name)
    act()
  }
  case msg => {
    println("Unhandled message: "+ msg)
    act()
  }
}
```

Remember the `reactions` variable in `scala.swing.MainFrame`?

```
var nClicks = 0
reactions += {
  case ButtonClicked(b) =>
    nClicks += 1
    label.text = "Number of button clicks: "+ nClicks
}
```

`Reactions` extends `PartialFunction[Event, Unit]`