ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Recap: Functions and Pattern Matching

Principles of Reactive Programming

Martin Odersky

Recap: Case Classes

Case classes are Scala’s preferred way to define complex data.

Example: Representing JSON (Java Script Object Notation)

```
{ ”firstName” : ”John”,
  ”lastName” : ”Smith”,
  ”address”: {
     ”streetAddress”: ”21 2nd Street”,
     ”state”: ”NY”,
     ”postalCode”: 10021
  },
  ”phoneNumbers”: [
    { ”type”: ”home”, ”number”: ”212 555-1234” },
    { ”type”: ”fax”, ”number”: ”646 555-4567” }
  ]
}
```

Representation of JSON in Scala

```
abstract class JSON
case class JSeq (elems: List[JSON])                extends JSON
case class JObj (bindings: Map[String, JSON]) extends JSON
case class JNum (num: Double)                      extends JSON
case class JStr (str: String)                      extends JSON
case class JBool(b: Boolean)                       extends JSON
case object JNull                                  extends JSON
```

Example

```
val data = JObj(Map(
  ”firstName” -> JStr(”John”),
  ”lastName” -> JStr(”Smith”),
  ”address” -> JObj(Map(
    ”streetAddress” -> JStr(”21 2nd Street”),
    ”state” -> JStr(”NY”),
    ”postalCode” -> JNum(10021)
  )),
  ”phoneNumbers” -> JSeq(List(
    JObj(Map(
      ”type” -> JStr(”home”), ”number” -> JStr(”212 555-1234”)
    )),
    JObj(Map(
      ”type” -> JStr(”fax”), ”number” -> JStr(”646 555-4567”)
    )) )) ))
```

Pattern Matching

Here’s a method that returns the string representation JSON data:

```
def show(json: JSON): String = json match {
  case JSeq(elems) =>
    ”[” + (elems map show mkString ”, ”) + ”]”
  case JObj(bindings) =>
    val assocs = bindings map {
      case (key, value) => ”\”” + key + ”\”: ” + show(value)
    }
    ”{” + (assocs mkString ”, ”) + ”}”
  case JNum(num) => num.toString
  case JStr(str) => ’\”’ + str + ’\”’
  case JBool(b)  => b.toString
  case JNull     => ”null”
}
```

Case Blocks

Question: What’s the type of:

```
{ case (key, value) => key + ”: ” + value }
```

Case Blocks

Question: What's the type of:

```
{ case (key, value) => key + ”: ” + value }
```

Taken by itself, the expression is not typable.

We need to prescribe an expected type.

The type expected by `map` on the last slide is

```
JBinding => String,
```

the type of functions from pairs of strings and `JSON` data to `String`. where `JBinding` is

```
type JBinding = (String, JSON)
```

Functions Are Objects

In Scala, every concrete type is the type of some class or trait.

The function type is no exception. A type like

```
JBinding => String
```

is just a shorthand for

```
scala.Function1[JBinding, String]
```

where `scala.Function1` is a trait and `JBinding` and `String` are its type arguments.

The Function1 Trait

Here’s an outline of trait `Function1`:

```
trait Function1[-A, +R] {
  def apply(x: A): R
}
```

The pattern matching block

```
{ case (key, value) => key + ”: ” + value }
```

expands to the `Function1` instance

```
new Function1[JBinding, String] {
  def apply(x: JBinding) = x match {
    case (key, value) => key + ”: ” + show(value)
  }
}
```

Subclassing Functions

One nice aspect of functions being traits is that we can subclass the function type.

For instance, maps are functions from keys to values:

```
trait Map[Key, Value] extends (Key => Value) ...
```

Subclassing Functions

One nice aspect of functions being traits is that we can subclass the function type.

For instance, maps are functions from keys to values:

```
trait Map[Key, Value] extends (Key => Value) ...
```

Sequences are functions from `Int` indices to values:

```
trait Seq[Elem] extends Int => Elem
```

That's why we can write

```
elems(i)
```

for sequence (and array) indexing.

Partial Matches

We have seen that a pattern matching block like

```
{ case ”ping” => ”pong” }
```

can be given type `String => String`.

```
val f: String => String = { case ”ping” => ”pong” }
```

But the function is not defined on all its domain!

```
f(”pong”)       // gives a MatchError
```

Is there a way to find out whether the function can be applied to a given argument before running it?

Partial Functions

Indeed there is:

```
val f: PartialFunction[String, String] =  { case ”ping” => ”pong” }
f.isDefinedAt(”ping”)         // true
f.isDefinedAt(”pong”)         // false
```

The partial function trait is defined as follows:

```
trait PartialFunction[-A, +R] extends Function1[-A, +R] {
  def apply(x: A): R
  def isDefinedAt(x: A): Boolean
}
```

Partial Function Objects

If the expected type is a `PartialFunction`, the Scala compiler will expand

```
{ case ”ping” => ”pong” }
```

as follows:

```
new PartialFunction[String, String] {
  def apply(x: String) = x match {
    case ”ping” => ”pong”
  }
  def isDefinedAt(x: String) = x match {
    case ”ping” => true
    case _ => false
  }
}
```

Exercise

Given the function

```
val f: PartialFunction[List[Int], String] = {
  case Nil => ”one”
  case x :: y :: rest => ”two”
}
```

What do you expect is the result of

`f.isDefinedAt(List(1, 2, 3))` ?

○ `true`

○ `false`

Exercise(2)

How about the following variation:

```
val g: PartialFunction[List[Int], String] = {
  case Nil => ”one”
  case x :: rest =>
    rest match {
      case Nil => ”two”
    }
}
```

`g.isDefinedAt(List(1, 2, 3))` gives:

○ `true`

○ `false`