Scala for TAPL'ers
I

Adriaan Moors
<adriaan cs kuleuven be>
17/10/2007

OO & FP
Pattern Matching
Variable Binding
  Substitution
  Evaluation (λ)

Scala

What Java should have been

- Fusion of object-oriented and functional programming
- Designed by Martin Odersky et al, EPFL
- Come see his talk in December! (13 or 14)

http://scala-lang.org

(Recommended reading: Scala by example,
Scala reference: have a look at Ch. 1--6, 8)

Java vs. Scala

```
class Test {
  public static void main(String[] args) {
    System.out.println("finally");
  }
}
```

```
$ scala
Welcome to Scala version 2.6.0-RC3.
Type in expressions to have them evaluated.
Type :help for more information.

scala> println("finally")
finally
```

Confession

- I cheated!
- ```
  class Test {
    public static void main(String[] args) {
      System.out.println("finally");
    }
  }
  ```
- Full Scala program:

```
object Test extends Application {
  println("finally")
}
```

- Honest!
```

Functional Programming

- Primitives: functions
  - Lambda abstraction
  - Application

```
scala> val inc = (x : Int) => x + 1
inc: (Int) => Int = <function>

scala> inc(41)
res0: Int = 42
```

OO

- Primitives: objects and methods
  - Function = object with `apply` method
  - Application = method-call

```
scala> object inc extends Function1[Int, Int] {
         def apply(x: Int) = x + 1 }
defined module inc // : Int => Int

scala> inc(41) // really: inc.apply(41)
res1: Int = 42
```

Scala Syntax cheat sheet

- Definitions start with a keyword
  - **class** / **trait** / **object** *name*[*params*](*args*) **extends** *T* { *members* }
  - **val** / **var** *name* : *type* = ...
  - **def** *name*[*params*](*args*) : *type* = ...
- Arbitrary nesting allowed
- Modifiers
  - **abstract** (only with **class**)
  - **override** (required when overriding)
- Types come after names (and can often be omitted)

[OO] modelling

- Let's model something

$$
\begin{array}{lll}
t ::= & x & \textit{variable} \\
 & \lambda x.\, t & \textit{abstraction} \\
 & t\ t & \textit{application}
\end{array}
$$

In Java

```
interface Term {}

class Var implements Term {
  public Var(String n){_name = n;}
  private String _name;
  public void setName(String n) {
    _name = n;
  }
  public String getName() {
    return _name;
  }
}

// And so on... ARGH!
// Don't have time to write down the rest...
// We shall mention J* no more
```

In Scala

```
trait Syntax {
  trait Term
  case class Var(name: Name) extends Term
  case class Abs(x: Name, body: Term) extends Term
  case class App(fun: Term, arg: Term) extends Term
}
```

[for now, assume Name = String]

- Consider `case class Var(name: Name) extends Term`
  - defines class `Var`, subclass of `Term`
  - has one member, `val name: Name`
    - can be overridden with getter/setter:
      - `def` name: Name
      - `def` name_= (n: Name): Unit
  - construction = like calling `def Var(name: Name): Name`
  - “deconstruction” = pattern matching

Case Classes

- Convenient way of defining a class with
  - its public fields, and
  - default constructor
- Instantiation may omit the `new` keyword
- Encapsulation = ok
  - fields can be overridden
- Pattern matching

Exercise: mental parsing

```scala
case class Var(name: Name) extends Term
case class Abs(x: Name, body: Term) extends Term
case class App(fun: Term, arg: Term) extends Term
```

[for now, assume Name = String]

- Encode the following Lambda terms:
  - λx. x
  - (λx. x x) (λx. x x)
  - λy. λx. y

Pattern Matching

```
scala> case class Person(age: Int, name: String)
defined class Person

scala> val jeff = Person(36, "Jeffrey")
jeff: Person = Person(36,Jeffrey)

scala> jeff match {
     | case Person(_, name) => "Oh, "+ name +"!"
     | }
res0: String = Oh, Jeffrey!
```

[Shorter]

```
scala> val Person(_, name) = jeff
name: String = Jeffrey
```

Pattern Matching

See 7.1 and 7.2 of Scala by Example

- pattern:
  - case class constructor w/ patterns as args
  - pattern variable (lower case!)
    - may only occur once in pattern!
  - don't care (_)
  - literals (`1`, "`abc`")
  - constant identifiers (upper case)
- guard: arbitrary expression

Exercise

(from Scala by Example)

Exercise 7.2.2 Consider the following definitions representing trees of integers:

```
trait IntTree
case object EmptyTree extends IntTree
case class Node(elem: Int, smaller: IntTree, greater: IntTree)
            extends IntTree
def contains(t: IntTree, v: Int): Boolean = t match {
   case EmptyTree                  => false
   case Node(x, _ , _ ) if v == x  => true
   case Node(x, l, _)   if v < x   => contains(l, v)
   case Node(x, _, r)              => contains(r, v)
}
def insert(t: IntTree, v: Int): IntTree  = t match {
   case EmptyTree                  => Node(v, EmptyTree, EmptyTree)
   case Node(x, _ , _ ) if v == x  => t
   case Node(x, l, r)   if v < x   => Node(x, insert(l, v), r)
   case Node(x, l, r)              => Node(v, insert(l, x), r)
}
```

Evaluation

using pattern matching

- Implement
  - $(\lambda x.\ t)\ v \rightarrow [x \mapsto v]\ t$
  - $t \rightarrow t' \quad \Rightarrow v\ t \rightarrow v\ t'$
  - $t \rightarrow t' \quad \Rightarrow t\ t_a \rightarrow t'\ t_a$

```
def eval(tm: Term): Term = tm match {
  case App(Abs(x, t), v) if v.isValue => subst(t, x, v)
  case App(v, t) if v.isValue => App(v, eval(t))
  case App(t1, t2) => App(eval(t1), t2)
}
```

Take that, Visitor pattern!

Variable Binding

Design Space

1. Invent fresh names when necessary.
2. Distinguish bound and unbound variables, must all be distinct.
3. Canonical representation: no need to rename.
   (Pierce uses de Bruijn)
4. Introduce explicit substitution.
5. Avoid variables.

http://www.cl.cam.ac.uk/~amp12/papers/frepbm/frepbm.pdf

Nominal Abstract Syntax

- Names are opaque atoms (a natural, or an object reference)
- Looking "under a binder" (directly at its scope) automatically freshens that binder in the scope
- Ex.: Looking at λx. x has an effect! (quantum mechanics ;-))
  - Generate a **fresh** x' and proceed as if we looked at λx'. x'

Nominal Abstract Syntax

t	$::=$		term
	$\mid$	n	name
	$\mid$	$n \ \backslash\backslash \ t$	n is bound in t
	$\mid$	$C\,(t_1\,,\,..\,,\,t_k)$	composite data structure

$\boxed{n \notin^{\alpha} t}$ n does not occur in t (modulo α-conversion)

$$\frac{n' \not\equiv^{\alpha} n}{n' \notin^{\alpha} n} \quad \text{F\_NAME}$$

$$\frac{}{n \notin^{\alpha} n \ \backslash\backslash \ t} \quad \text{F\_A\_CONVERT}$$

$$\frac{n \not\equiv^{\alpha} n' \qquad n' \notin^{\alpha} t}{n' \notin^{\alpha} n \ \backslash\backslash \ t} \quad \text{F\_UNBOUND}$$

$$\frac{\forall i \in 1..k.\ n \notin^{\alpha} t_i}{n \notin^{\alpha} C\,(t_1\,,\,\ldots\,,\,t_k)} \quad \text{F\_COMPOSITE}$$

x not in λx. x, as λx. x = λx’. x’
(λx. x ~ I \\ I)
I not in I \\ I

Swapping

$\boxed{t\,[a \leftrightarrow b]\; = t'}$ swapping a and b in t yields t'

$$\frac{}{a\,[a \leftrightarrow b]\; = b} \quad \text{X\_A}$$

$$\frac{}{b\,[a \leftrightarrow b]\; = a} \quad \text{X\_B}$$

$$\frac{\begin{array}{c} n \not\equiv^{\alpha} a \\ n \not\equiv^{\alpha} b \end{array}}{n\,[a \leftrightarrow b]\; = n} \quad \text{X\_NAME}$$

$$\frac{\begin{array}{c} n\,[a \leftrightarrow b]\; = n' \\ t\,[a \leftrightarrow b]\; = t' \end{array}}{(n \;\backslash\backslash\; t)\,[a \leftrightarrow b]\; = n' \;\backslash\backslash\; t'} \quad \text{X\_BIND}$$

$$\frac{\forall i \in 1..k.\; t_i\,[a \leftrightarrow b]\; = t_i'}{C\,(t_1\,,\,\ldots\,,\,t_k)\,[a \leftrightarrow b]\; = C\,(t_1'\,,\,\ldots\,,\,t_k')} \quad \text{X\_COMPOSITE}$$

Equality

$\boxed{t \ \equiv^{\alpha} \ t'}$ t is syntactically equal to t' modulo alpha-conversion

λx. λy. x = λx'. λy'. x'
1\\2\\1 = 4\\3\\4

λx. λy. x != λx'. λy'. x
1\\2\\1 != 4\\3\\1

$$\frac{\begin{array}{l} n \ \not\equiv^{\alpha} \ n_1 \\ n \not\in^{\alpha} t_1 \\ t_1 \, [n \leftrightarrow n_1] \ = t_2 \\ t_2 \ \equiv^{\alpha} \ t \end{array}}{n \ \backslash\backslash \ t \ \equiv^{\alpha} \ n_1 \ \backslash\backslash \ t_1} \quad \text{Q\_BINDX}$$

$$\frac{}{a \ \equiv^{\alpha} \ a} \quad \text{Q\_NAME}$$

$$\frac{\begin{array}{l} n \ \equiv^{\alpha} \ n' \\ t \ \equiv^{\alpha} \ t' \end{array}}{n \ \backslash\backslash \ t \ \equiv^{\alpha} \ n' \ \backslash\backslash \ t'} \quad \text{Q\_BIND}$$

$$\frac{\forall i \in 1..k. \ t_i \ \equiv^{\alpha} \ t'_i}{C \ (t_1 \ , \ \ldots \ , \ t_k) \ \equiv^{\alpha} \ C \ (t'_1 \ , \ \ldots \ , \ t'_k)} \quad \text{Q\_COMPOSITE}$$

Exercise

- Derive (or not):
  - 1 \\ 1 ≡ 2 \\ 2
  - 1 \\ 1 ≡ 2 \\ 1
  - 1 \\ 2 ≡ 2 \\ 1

Names in Scala

```scala
type AlsoNominal[T] = T with Nominal[T] // avoid F-bounds
trait Nominal[Self] { // something that contains names
  // return this entity after swapping a and b
  def swap(a: Name, b: Name): AlsoNominal[Self]

  // a is in the set of free variables of this entity
  def fresh(a: Name): Boolean

  def alphaEq(other: AlsoNominal[Self]): Boolean
}
class Name(val name: String) extends Nominal[Name] {
  def swap(a: Name, b: Name) : AlsoNominal[Name]
    = if(this.alphaEq(a)) b else if(this.alphaEq(b)) a else this

  def fresh(a: Name) = ! this.alphaEq(a)

  def alphaEq(other: AlsoNominal[Name]) = other eq this // identity
}
```

Variable Binding

```
case class \\[T](binder: Name, body: AlsoNominal[T]) extends Nominal[\\[T]]{
  def swap(a: Name, b: Name)
    = new \\(binder swap(a, b), body swap(a, b))

  def fresh(a: Name)
    = if(a.alphaEq(binder)) true // implicitly alpha-convert binder
      else body.fresh(a)

  def alphaEq(o: AlsoNominal[\\[T]]): Boolean
    = (binder.alphaEq(o.binder) && body.alphaEq(o.body)) ||
      (o.body.fresh(binder) && o.body.swap(o.binder, binder).alphaEq(body))
}
```

Embedding in AST

```
trait Syntax {
  trait Term
  case class Var(name: Name) extends Term
  case class Abs(abs: \\[Term]) extends Term
  case class App(fun: Term, arg: Term) extends Term
}
```

Substitution

```
def subst(self: Term, n: Name, to: Term): Term = self match {
  case Var(name) => if(name == n) to else self
  case App(fun, arg) => App(subst(fun, n, to),
                            subst(arg, n, to))
  case Abs(a) =>
    val (x, body) = a.unabs
    Abs(new \\(x, subst(body, n, to)))
}
case class \\[T](binder: Name, body: AlsoNominal[T]) extends Nominal[\\[T]]{
...
  def unabs: (Name, AlsoNominal[T]) = {
    val freshBinder = Name(binder.name) // fresh name --> avoid capture
    (freshBinder, body.swap(binder, freshBinder))
  }
}
```

Evaluation

```
trait Evaluation { self: Syntax with Substitution with Binding with Evaluation =>
  def eval(tm: Term): Term = tm match {
    case App(Abs(a), v) if v.isValue =>
      val (x, t) = a.unabs
      subst(t, x, v)
    case App(v, t) if v.isValue => App(v, eval(t))
    case App(t1, t2) => App(eval(t1), t2)
  }
}
```

TODO:

- implement closure of this relation (DIY!)
- encapsulate unabs (next time)
  - `class \\[T](private val binder: Name, private val body: AlsoNominal[T])`
  - define extractor so that pattern matching always calls unabs

Homework

- Complete the implementation yourself!
  - (Code soon available from my homepage)

    http://www.cs.kuleuven.be/~adriaan/?q=fst_scala1

- Find beta-reductions that trigger tricky substitutions and evaluate them
  - Example: $[x \mapsto y](\lambda y.\ x)$
- Play around with it! Will build on this next time!
- Problems/questions/bug-reports: Toledo (?)

Next Up

- Improving our library for variable binding
- Advanced Scala hacking
  - implicit conversions
  - syntactic tricks (DSL as a library)
- Parsing
  - Parser combinators
- Type checking