Scala for TAPL’ers
2

Adriaan Moors
<adriaan cs kuleuven be>

24/10/2007

Variable Binding
- Theory (quick rehash)
- Scala implementation

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)

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
- *Looking* at λx. x has an effect! (quantum mechanics ;-))
  - Generates a **fresh** x' and proceeds 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

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

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

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

Nominal Abstract Syntax

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

x not in λx. x, as λx. x = λx'. x'
(λx. x ~ 1 \\ 1)
1 not in 1 \\ 1

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

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

$$\frac{\begin{array}{c} n \not=^{\alpha} n' \\ n' \notin^{\alpha} t \end{array}}{n' \notin^{\alpha} n \mathbin{\backslash\backslash} t} \quad \text{F\_UNBOUND}$$

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

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

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

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

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

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

The new 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
}
```

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

```

```scala
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
              X-A                           X-B                  X-NAME

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

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

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))  X-BIND

  def fresh(a: Name)
    = if(a.alphaEq(binder)) true // implicitly alpha-convert binder F-A-CONVERT
      else body.fresh(a)
                                F-UNBOUND
  def alphaEq(o: AlsoNominal[\\[T]]): Boolean   Q-BIND & Q-BINDX
    = (binder.alphaEq(o.binder) && body.alphaEq(o.body)) ||
      (!binder.alphaEq(o.binder) &&
        o.body.fresh(binder) && o.body.swap(o.binder, binder).alphaEq(body))
}
```

$$\frac{n \not\equiv^{\alpha} n_1 \quad n \notin^{\alpha} t_1 \quad t_1\,[n \leftrightarrow n_1] = t_2 \quad t_2 \equiv^{\alpha} t}{n \setminus\!\!\setminus t \equiv^{\alpha} n_1 \setminus\!\!\setminus t_1}\ \text{Q\_BINDX}$$

$$\frac{t \equiv^{\alpha} t'}{n \setminus\!\!\setminus t \equiv^{\alpha} n' \setminus\!\!\setminus t'}\ \text{Q\_BIND}$$

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](private val binder: Name,
                 private val body: AlsoNominal[T]) extends Nominal[\\[T]]{
...
  def unabs: (Name, AlsoNominal[T]) = {
    val freshBinder = new 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:

- encapsulate unabs (next time)
  - is unabs really necessary in eval?
  - use extractors to make pattern matching use unabs: http://lambda-the-ultimate.org/node/1960
- implement transitive closure of this relation (DIY!)

Homework

- Complete the implementation
- 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