ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Foundations of Scala

Foundations of Software

Martin Odersky

Where are we when modelling Scala?

Simple (?) example: `List` type:

```
trait List[T] {
  def isEmpty: Boolean; def head: T; def tail: List[T]
}
def Cons[T](hd: T, tl: List[T]) = new List[T] {
  def isEmpty = false; def head = hd; def tail = tl
}
def Nil[T] = new List[T] {
  def isEmpty = true; def head = ???; def tail = ???
}
```

New Problems

- List is *parameterized*.
- List is *recursive*.
- List can be *invariant* or *covariant*.

Covariant List type

```
trait List[+T] {
  def isEmpty = false; def head = hd; def tail = tl
}
```

`Cons`, `Nil` as before.

Modelling Parameterized Types

Traditionally: Higher-kinded types.

- Besides plain types, have functions from types to types, and functions over these and so on.
- Needs a kinding system:

```
*                // Kind of normal types
* -> *           // Kind of unary type constructors
* -> * -> *
(* -> *) -> *
...
```

- Needs some way to express type functions, such as a λ for types.

Modelling Recursive Types

Traditionally: Have a constructor for recursive types $\mu t. T(t)$.

Example:

```
mu ListInt. { head: Int, tail: ListInt }
```

Tricky interactions with equality and subtyping.

Consider:

```
type T = mu t. Int -> Int -> t
```

How do `T` and `Int -> T` relate?

Modelling Variance

Traditionally: Express definition site variance

```
trait List[+T] ...
trait Function1[-T, +U] ...

List[C], Function1[D, E]
```

as use-site variance (aka Java wildcards):

```
trait List[T] ...
trait Function1[T, U]

List[_ <: C]
Function1[_ >: D, _ <: E]
```

Meaning of Wildcards

A type like `Function1[_ >: D, _ <: E]` means:

The type of functions where the argument is some (unknown) supertype of `D` *and the result is some (unknown) subtype of* `E`.

This can be modelled as an *existential type*:

```
Function1[X, Y] forSome { type X >: D; type Y <: E }  // Scala
ex X >: D, Y <: E. Function1[X, Y]  // More traditional notation
```

Combining Several of These Features

… is possible, but gets messy rather quickly

Idea: Use Path Dependent Types as a Common Basis

Here is a re-formulation of `List`.

```
trait List { self =>
  type T
  def isEmpty: Boolean
  def head: T
  def tail: List { type T = self.T }
}
def Cons[X](hd: T, tl: List { type T = X }) = new List {
  type T = X
  def isEmpty = false
  def head = hd
  def tail = tl
}
```

Analogous for `Nil`.

Handling Variance

```
trait List { self =>
  type T
  def isEmpty: Boolean
  def head: T
  def tail: List { type T <: self.T }
}
def Cons[X](hd: T, tl: List { type T <: X }) = new List {
  type T = X
  def isEmpty = false
  def head = hd
  def tail = tl
}
```

Elements needed:

- Variables, functions
- Abstract types `{ type T <: B }`
- Refinements `C { ... }`
- Path-dependent types `self.T`.

Abstract Types

- An abstract type is a type without a concrete implementation
- Instead only (upper and/or lower) bounds are given.

Example

```
trait KeyGen {
  type Key
  def key(s: String): this.Key
}
```

Implementations of Abstract Types

- Abstract types can be refined in subclasses or implemented as *type aliases*.

Example

```
object HashKeyGen {
  type Key = Int
  def key(s: String) = s.hashCode
}
```

Generic Functions over Abstract Types

We can write functions that work for all implementations of an abstract type like this:

```
def mapKeys(k: KeyGen, ss: List[String]): List[k.Key] =
  xs.map(x => k.key(x))
```

- `k.Key` is a *path-dependent* type.
- The type depends on the value of `k`, which is a term.
- The type of `mapKeys` is a *dependent function type*.
  mapKeys: (k: KeyGen, ss: List[String]) -> List[k.Key]
- Note that the occurrence of `k` in the type is essential; without it we could not express the result type!.

Dependent Functions in Scala

Scala allows to define a dependent method whose result type depends on the parameters. We have seen an example in `mapKeys`:

```
mapKeys(k: KeyGen, ss: List[String]): List[k.Key] = ...
```

But we cannot express the type of `mapKeys` directly. The best we can do is use the same encoding we use for simple function types:

```
trait KeyFun {
  def apply(k: KeyGen, ss: List[String]): List[k.Key]
}

mapKeys = new KeyFun {
  def apply(k: KeyGen, ss: List[String]): List[k.Key] = ...
}
```

This means that we need to define need one type per dependent function,

Formalization

We now formalize these ideas in a calculus.

DOT standands for (path)-Dependent Object Types.

Program:

- Syntax, Typing rules (this week)
- An approach to the meta theory (next week).

Syntax

x, y, z	**Variable**
a, b, c	**Term member**
A, B, C	**Type member**
$S, T, U ::=$	**Type**
$\top$	top type
$\bot$	bot type
$\{a : T\}$	field declaration
$\{A : S..T\}$	type declaration
$x.A$	type projection
$S \wedge T$	intersection
$\mu(x{:}\, T)$	recursive type
$\forall(x{:}\, S)\, T$	dependent function

$v ::=$	**Value**
$\nu(x{:}\, T)\, d$	object
$\lambda(x{:}\, T)\, t$	lambda
$s, t, u ::=$	**Term**
x	variable
v	value
$x.a$	selection
$x\ y$	application
$\textbf{let}\ x = t\ \textbf{in}\ u$	let
$d ::=$	**Definition**
$\{a = t\}$	field def.
$\{A = T\}$	type def.
$d_1 \wedge d_2$	aggregate def.

DOT Types

DOT	Scala	
$\top$	`Any`	Top type
$\bot$	`Nothing`	Bottom type
$\{a : T\}$	`{ val a: T }`	Record field
$\{A : S..T\}$	`{ type A >: S <: T}`	Abstract type
$T \wedge U$	`T & U`	Intersection
		(Together these can form records)
$x.A$	`x.A`	Type projection
$\mu(x: T)$	`{x => ...}`	Recursive type
		(Scala allows only recursive records)
$\forall(x: S)\, T$	`S => T`	Dependent function type
		(Scala has only simple function types)

DOT Definitions

Definitions make concrete record values.

DOT	Scala	
$\{a = t\}$	`{ val a = t }`	Field definition
$\{A = T\}$	`{ type A = T }`	Type definition
$d_1 \wedge d_2$	-	Record formation
		(Scala uses $\{d_1; \dots; d_n\}$ directly)

Definitions are grouped together in an object

DOT	Scala	
$\nu(x\colon T)d$	`new { x: T => d }`	Instance creation

DOT Terms

DOT values are objects and lambdas.

DOT terms have member selection and application work on *variables*, not values or full terms.

```
x.a        instead of     t.a
x y        instead of     t u
```

This is not a reduction of expressiveness. With `let`, we can apply the following *desugarings*, where `x` and `y` are fresh variables:

```
t.a    --->    let x = t in x.a
t u    --->    let x = t in let y = u in x y
```

This way of writing programs is also called *administrative normal form* (ANF).

Programmer-Friendlier Notation

In the following we use the following ASCII versions of DOT constructs.

`(x: T) => U`	for	$\lambda(x{:}\,T)\,U$
`(x: T) -> U`	for	$\forall(x{:}\,T)\,U$
`new(x: T)d`	or	
`new { x: T => d }`	for	$\nu(x{:}\,T)\,d$
`rec(x: T)`	or	
`{ x => T }`	for	$\mu(x{:}\,T)$
`T & U`	for	$T \wedge U$
`Any`	for	$\top$
`Nothing`	for	$\bot$

Encoding of Generics

For generic *types*: Encode type parameters as type members

For generic *functions*: Encode type parameters as value parameters which carry a type field. Hence polymorphic (universal) types become dependent function types.

Example: The polymorphic type of the `twice` method

$$\forall X.(X \to X) \to X \to X$$

is represented as

```
(cX: {A: Nothing..Any}) -> (f: (cX.A) -> cX.A) -> (x:cX.A) -> cX.A
```

`cX` is a menmonic for "cell containing a type variance X".

Example: Church Booleans

Let

```
type IFT  =  { if: (x: {A: Nothing..Any}) -> (t: x.A) -> (f: x.A) -> x.A }
```

Then define:

```
    let boolimpl =
      let boolImpl =
        new(b: { Boolean: IFT..IFT } &
               { true: IFT } &
               { false: IFT })
          { Boolean = IFT } &
          { true = (x: {A: Nothing..Any}) => (t: x.A) => (f: x.A) => t } &
          { false = (x: {A: Nothing..Any}) => (t: x.A) => (f: x.A) => f }
    in ...
```

Church Booleans API

To hide the implementation details of `boolImpl`, we can use a wrapper:

```
let bool =
  let boolWrapper =
    (x: rec(b: {Boolean: Nothing..Any} &
              {true: b.Boolean} &
              {false: b.Boolean})) => x
  in boolWrapper boolImpl
```

Abbreviations and Syntactic Sugar

We use the following Scala-oriented syntax for type members.

```
type A              for     {A: Nothing..Any}
type A = T          for     {A: T..T}
type A >: S         for     {A: S..Any}
type A <: U         for     {A: Nothing..U}
type A >: S <: U    for     {A: S..U}
```

Abbreviations (2)

We group multiple, intersected definitions or declarations in one pair of braces, replacing `&` with `;` or a newline. E.g, the definition

```
{ type A = T; a = t  }
```

expands to

```
{ A = T} & { a = t }
```

and the type

```
{ type A <: T; a: T }
```

expands to

```
{ A: S..T } & { a: T }
```

Abbreviations (3)

We expand type ascriptions to applications:

```
t: T
```

expands to

```
((x: T) => x) t
```

(which expands in turn to)

```
let y = (x: T) => x in let z = t in x z
```

Abbreviations (4)

We abbreviate

```
new (x: T)d
```

to

```
new { x => d }
```

if the type of definitions `d` is given explicitly, and to

```
new { d }
```

if `d` does not refer to the `this` reference `x`.

Church Booleans, Abbreviated

```
let bool =
  new { b =>
    type Boolean = if: (x: { type A }) -> (t: x.A) -> (f: x.A) -> x.A
    true  = (x: { type A }) => (t: x.A) => (f: x.A) => t
    false = (x: { type A }) => (t: x.A) => (f: x.A) => f
  }: { b => type Boolean; true: b.Boolean; false: b.Boolean }
```

Example: Covariant Lists

We now model the following Scala definitions in DOT:

```scala
package scala.collection.immutable
trait List[+A] {
  def isEmpty: Boolean; def head: A; def tail: List[A]
}
object List {
  def nil: List[Nothing] = new List[Nothing] {
    def isEmpty = true; def head = head; def tail = tail // infinite loops
  }
  def cons[A](hd: A, tl: List[A]) = new List[A] {
    def isEmpty = false; def head = hd; def tail = tl
  }
}
```

Encoding of Lists

```
let scala_collection_immutable_impl = new { sci =>
  type List = { thisList =>
    type A
    isEmpty: bool.Boolean
    head: thisList.A
    tail: sci.List & {type A <: thisList.A }
  }
  cons = (x: {type A}) => (hd: x.A) =>
    (tl: sci.List & { type A <: thisList.A }) =>
      let l = new {
        type A = x.A
        isEmpty = bool.false
        head = hd
        tail = tl }
      in l
```

Encoding of Lists (ctd)

```
 nil = (x: {type A}) => (hd: x.A) =>
   (tl: sci.List & { type A <: thisList.A }) =>
     let l = new { l =>
       type A = x.A
       isEmpty = bool.true
       head = l.head
       tail = l.tail }
     in l
}            // end implementation new { sci => ...
```

List API

We wrap `scala_collection_immutable_impl` to hide its implementation types.

```
let scala_collection_immutable = scala_collection.immutable_impl: { sci =>
  type List <: { thisList =>
    type A
    isEmpty: bool.Boolean
    head: thisList.A
    tail: sci.List & {type A <: thisList.A }
  }
  nil: sci.List & { type A = Nothing }
  cons: (x: {type A}) -> (hd: x.A) ->
    (tl: sci.List & { type A <: thisList.A }) ->
      sci.List & { type A = x.A }
```

Nominal Types

The encodings give an explanation what nominality means.

A nominaltype such as `List` is simply an abstract type, whose implementation is hidden.

Still To Do

The rest of the calculus is given by three definitions:

An evaluation relation $t \longrightarrow t'$.

Type assignment rules $\Gamma \vdash x : T$

Subtyping rules $\Gamma \vdash T <: U$.

Evaluation $t \longrightarrow t'$

Evaluation is particular since it works on variables not values.

This is needed to keep reduced terms in ANF form.

$$
\begin{array}{rcll}
e[t] & \longrightarrow & e[t'] & \textbf{if } t \longrightarrow t' \\
\textbf{let } x = v \textbf{ in } e[x\ y] & \longrightarrow & \textbf{let } x = v \textbf{ in } e[[z := y]t] & \textbf{if } v = \lambda(z\colon T)t \\
\textbf{let } x = v \textbf{ in } e[x.a] & \longrightarrow & \textbf{let } x = v \textbf{ in } e[t] & \textbf{if } v = \nu(x\colon T)\ldots\{a = t\}\ldots \\
\textbf{let } x = y \textbf{ in } t & \longrightarrow & [x := y]t & \\
\textbf{let } x = \textbf{let } y = s \textbf{ in } t \textbf{ in } u & \longrightarrow & \textbf{let } y = s \textbf{ in let } x = t \textbf{ in } u &
\end{array}
$$

where the *evaluation context* e is defined as follows:

$$
e ::= [\,] \mid \textbf{let } x = [\,] \textbf{ in } t \mid \textbf{let } x = v \textbf{ in } e
$$

Note that evaluation uses only *variable renaming*, not full substitution.

Type Assignment $\Gamma \vdash t : T$

$$\frac{x \in \Gamma}{\Gamma \vdash x : T} \tag{\textsc{Var}}$$

$$\frac{\Gamma,\ x : T \vdash t : U}{\Gamma \vdash \lambda(x{:}\, T)t : \forall(x{:}\, T)U} \tag{\textsc{All-I}}$$

$$\frac{\Gamma \vdash x : \forall(z{:}\, S)T \quad \Gamma \vdash y : S}{\Gamma \vdash x\, y : [z := y]T} \tag{\textsc{All-E}}$$

$$\frac{\Gamma,\ x : T \vdash d : T}{\Gamma \vdash \nu(x{:}\, T)d : \mu(x{:}\, T)} \tag{\{\}-I}$$

$$\frac{\Gamma \vdash x : \{a : T\}}{\Gamma \vdash x.a : T} \tag{\{\}-E}$$

Type Assignment (2)

$$\frac{\Gamma \vdash t : T \quad \Gamma, x : T \vdash u : U \quad x \notin fv(U)}{\Gamma \vdash \textbf{let } x = t \textbf{ in } u : U} \qquad (\textsc{Let})$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash x : \mu(x : T)} \qquad (\textsc{Rec-I})$$

$$\frac{\Gamma \vdash x : \mu(x : T)}{\Gamma \vdash x : T} \qquad (\textsc{Rec-E})$$

$$\frac{\Gamma \vdash x : T \quad \Gamma \vdash x : U}{\Gamma \vdash x : T \wedge U} \qquad (\textsc{And-I})$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T <: U}{\Gamma \vdash t : U} \qquad (\textsc{Sub})$$

Type Assignment

Note that there are now 4 rules which are not syntax-directed: (Sub), (And-I), (Rec-I), and (Rec-E).

It turns out that the meta-throey becomes simpler if (And-I), (Rec-I), and (Rec-E) are not rolled into subtyping.

Definition Type Assignment $\Gamma \vdash d : T$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \{a = t\} : \{a : T\}} \qquad (\textsc{Fld-I})$$

$$\Gamma \vdash \{A = T\} : \{A : T..T\} \qquad (\textsc{Typ-I})$$

$$\frac{\Gamma \vdash d_1 : T_1 \quad \Gamma \vdash d_2 : T_2 \\ dom(d_1), dom(d_2) \text{ disjoint}}{\Gamma \vdash d_1 \wedge d_2 : T_1 \wedge T_2} \qquad (\textsc{AndDef-I})$$

Note that there is no subsumption rule for definition type assignment.

Subtyping $\Gamma \vdash T <: U$

$$\Gamma \vdash T <: \top \tag{Top}$$

$$\Gamma \vdash \bot <: T \tag{Bot}$$

$$\Gamma \vdash T <: T \tag{Refl}$$

$$\frac{\Gamma \vdash S <: T \quad \Gamma \vdash T <: U}{\Gamma \vdash S <: U} \tag{Trans}$$

$$\Gamma \vdash T \wedge U <: T \tag{And$_1$-<:}$$

$$\Gamma \vdash T \wedge U <: T \tag{And$_2$-<:}$$

$$\frac{\Gamma \vdash S <: T \quad \Gamma \vdash S <: U}{\Gamma \vdash S <: T \wedge U} \tag{<:-And}$$

Subtyping (2)

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash x.A <: T} \qquad (\textsc{Sel-<:})$$

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash S <: x.A} \qquad (\textsc{<:-Sel})$$

$$\frac{\begin{array}{c}\Gamma \vdash S_2 <: S_1 \\ \Gamma,\ x : S_2 \vdash T_1 <: T_2\end{array}}{\Gamma \vdash \forall(x : S_1)\, T_1 <: \forall(x : S_2)\, T_2} \qquad (\textsc{All-<:-All})$$

$$\frac{\Gamma \vdash T <: U}{\Gamma \vdash \{a : T\} <: \{a : U\}} \qquad (\textsc{Fld-<:-Fld})$$

$$\frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash \{A : S_1..T_1\} <: \{A : S_2..T_2\}} \qquad (\textsc{Typ-<:-Typ})$$

Conclusion

DOT is a fairly small calculus that can express “classical” Scala programs.

Even though the calculus is small, its meta theory turned out to be surprisingly hard.

More on this next week.