

· Foundations of Software, Week 14

Dependent Object Types

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*:

```
exists X >: D, Y <: E. Function1[X, Y]
```


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: X, 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: X, 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 extends KeyGen {  
  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] =  
  ss.map(s => k.key(s))
```

- `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!

Formalization

We now formalize these ideas in a calculus.

DOT stands for (path)-Dependent Object Types.

Program:

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

Syntax

x, y, z

a, b, c

A, B, C

$S, T, U ::=$

\top

\perp

$\{a : T\}$

$\{A : S..T\}$

$x.A$

$S \wedge T$

$\mu(x : T)$

$\forall(x : S) T$

Variable

Term member

Type member

Type

top type

bot type

field declaration

type declaration

type projection

intersection

recursive type

dependent function

$v ::=$

$\nu(x : T) d$

$\lambda(x : T) t$

$s, t, u ::=$

x

ν

$x.a$

$x y$

let $x = t$ **in** u

$d ::=$

$\{a = t\}$

$\{A = T\}$

$d_1 \wedge d_2$

Value

object

lambda

Term

variable

value

selection

application

let

Definition

field def.

type def.

aggregate def.

DOT Types

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

DOT Definitions

Definitions make concrete record values.

DOT	Scala	
$\{a = t\}$	$\{ \text{def } a = t \}$	Field definition
$\{A = T\}$	$\{ \text{type } A = T \}$	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	
$\nu(x: T)d$	$\text{new } \{ x: T \Rightarrow d \}$	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:

$t.a$	--->	<code>let x = t in x.a</code>
$t\ u$	--->	<code>let x = t in let y = u in x y</code>

This way of writing programs is also called *monadic normal form* (MNF).

Programmer-Friendlier Notation

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

$(x: T) \Rightarrow U$	for	$\lambda(x: T) U$
$(x: T) \rightarrow U$	for	$\forall(x: T) U$
$\text{new}(x: T)d$	or	
$\text{new } \{ x: T \Rightarrow d \}$	for	$\nu(x: T)d$
$\text{rec}(x: T)$	or	
$\{ x \Rightarrow T \}$	for	$\mu(x: T)$
$T \ \& \ U$	for	$T \wedge U$
Any	for	\top
Nothing	for	\perp

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

$$\forall X. (X \rightarrow X) \rightarrow X \rightarrow X$$

is represented as

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

cX is a mnemonic for “cell containing a type variance X”.

Example: Church Booleans

Let

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

Then define:

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

Church Booleans API

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

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


Abbreviations and Syntactic Sugar

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

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

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: Nothing..T } & { a: T }
```

Abbreviations (3)

We expand type ascriptions to applications:

$t : T$

expands to

$((x : T) \Rightarrow x) \ t$

(which expands in turn to)

$\text{let } y = (x : T) \Rightarrow x \text{ in let } z = t \text{ in } y \ z$

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  = {if: (x: { type A }) => (t: x.A) => (f: x.A) => t}  
    false = {if: (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:

```
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 <: x.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}) =>
  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 <: x.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.

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

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

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

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

Type Assignment $\Gamma \vdash t : T$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (\text{VAR})$$

$$\frac{\Gamma, x : T \vdash t : U}{\Gamma \vdash \lambda(x : T) t : \forall(x : T) U} \quad (\text{ALL-I})$$

$$\frac{\Gamma \vdash x : \forall(z : S) T \quad \Gamma \vdash y : S}{\Gamma \vdash x y : [z := y] T} \quad (\text{ALL-E})$$

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

$$\frac{\Gamma \vdash x : \{a : T\}}{\Gamma \vdash x.a : T} \quad (\{\}-\text{E})$$

Type Assignment (2)

$$\frac{\Gamma \vdash t : T \quad \Gamma, x : T \vdash u : U \quad x \notin \text{fv}(U)}{\Gamma \vdash \mathbf{let } x = t \mathbf{ in } u : U} \quad (\text{LET})$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash x : \mu(x : T)} \quad (\text{REC-I})$$

$$\frac{\Gamma \vdash x : \mu(x : T)}{\Gamma \vdash x : T} \quad (\text{REC-E})$$

$$\frac{\Gamma \vdash x : T \quad \Gamma \vdash x : U}{\Gamma \vdash x : T \wedge U} \quad (\text{AND-I})$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T <: U}{\Gamma \vdash t : U} \quad (\text{SUB})$$

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-theory becomes simpler if (And-I), (Rec-I), and (Rec-E) are not rolled into subtyping.

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

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \{a = t\} : \{a : T\}} \quad (\text{FLD-I})$$

$$\Gamma \vdash \{A = T\} : \{A : T..T\} \quad (\text{TYP-I})$$

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

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

Subtyping $\Gamma \vdash T <: U$

$$\Gamma \vdash T <: \top \quad (\text{TOP})$$

$$\Gamma \vdash \perp <: T \quad (\text{BOT})$$

$$\Gamma \vdash T <: T \quad (\text{REFL})$$

$$\frac{\Gamma \vdash S <: T \quad \Gamma \vdash T <: U}{\Gamma \vdash S <: U} \quad (\text{TRANS})$$

$$\Gamma \vdash T \wedge U <: T \quad (\text{AND}_1\text{-<:})$$

$$\Gamma \vdash T \wedge U <: U \quad (\text{AND}_2\text{-<:})$$

$$\frac{\Gamma \vdash S <: T \quad \Gamma \vdash S <: U}{\Gamma \vdash S <: T \wedge U} \quad (<:\text{-AND})$$

Subtyping (2)

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash x.A <: T} \quad (\text{SEL-}<:)$$

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash S <: x.A} \quad (<:-\text{SEL})$$

$$\frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma, x : S_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(x:S_1) T_1 <: \forall(x:S_2) T_2} \quad (\text{ALL-}<:-\text{ALL})$$

$$\frac{\Gamma \vdash T <: U}{\Gamma \vdash \{a : T\} <: \{a : U\}} \quad (\text{FLD-}<:-\text{FLD})$$

$$\frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash \{A : S_1..T_1\} <: \{A : S_2..T_2\}} \quad (\text{TYP-}<:-\text{TYP})$$

Example: Uses of Abstract Types

- 1 To encode type parameters (as in `List`)
- 2 To hide information (as in `KeyGen`)
- 3 To resolve variance puzzlers

Resolving Variance Puzzlers with Abstract Types

A standard example to justify unsound covariance is this:

Let's model animals which eat food items.

Both `Animal` and `Food` are the root of a type hierarchy.

```
trait Animal
trait Cow extends Animal with Food
trait Lion extends Animal

trait Food
trait Grass extends Food
```

Adding eat

```
trait Animal {  
  def eat(food: Food): Unit  
}  
trait Cow extends Animal {  
  def eat(food: Grass): Unit  
}  
trait Lion extends Animal {  
  def eat(food: Cow): Unit  
}
```

Problem: eat in Cow or Lion does not override correctly the eat in Animal, because of the contravariance rule for function subtyping.

Refining the Model

We can get the right behavior with an abstract type.

```
trait Animal {  
  type Diet <: Food  
  def eat(food: Diet): Unit  
}  
  
trait Cow extends Animal {  
  type Diet <: Grass  
  def eat(food: this.Diet): Unit  
}  
  
object Milka extends Cow {  
  type Diet = AlpineGrass  
  def eat(food: AlpineGrass): Unit  
}
```

Translating to DOT

```
type Animal = { this => {Diet: Nothing..Food} & {eat: this.Diet -> Unit}}  
type Cow    = { this => {Diet: Nothing..Grass} & {eat: this.Diet -> Unit}}
```

Do we have Cow <: Animal?

Translating to DOT

```
type Animal = { this => {Diet: Nothing..Food} & {eat: this.Diet -> Unit}}  
type Cow    = { this => {Diet: Nothing..Grass} & {eat: this.Diet -> Unit}}
```

Is Cow <: Animal?

No. There is no subtyping rule for recursive types.

Translating to DOT

But we *do* have:

```
x: Cow
==>    // expand the definition
x: { this => {Diet: Nothing..Grass} & {eat: this.Diet -> Unit}}
==>    // by (Rec-E)
x: {Diet: Nothing..Grass} & {eat: x.Diet -> Unit}}
==>    // by (Sub)
x: {Diet: Nothing..Food} & {eat: x.Diet -> Unit}}
==>    // by (Rec-I)
x: { this => {Diet: Nothing..Food} & {eat: this.Diet -> Unit}}
==>    // Collapse the definition
x: Animal
```

The Meta Theory

As usual, need to prove progress and preservation theorems.

Theorem (Preservation) If $\Gamma \vdash t : T$ and $t \longrightarrow u$ then $\Gamma \vdash u : T$.

Theorem (Progress) If $\vdash t : T$ then t is a value or there is a term u such that $t \longrightarrow u$.

(?)

The Meta Theory

As usual, need to prove progress and preservation theorems.

Theorem (Preservation) If $\Gamma \vdash t : T$ and $t \longrightarrow u$ then $\Gamma \vdash u : T$.

Theorem (Progress) If $\vdash t : T$ then t is a value or there is a term u such that $t \longrightarrow u$.

(?)

In fact this is wrong. Counter example:

$t = \text{let } x = (y: \text{Bool}) \Rightarrow y \text{ in } x$

Fixing Progress

Theorem (Progress) If $\vdash t : T$ then t is an *answer* or there is a term u such that $t \longrightarrow u$.

Answers n are defined by the production

$$n ::= x \mid v \mid \text{let } x = v \text{ in } n$$

Why It's Difficult

We always need some form of inversion.

E.g.:

- If $\Gamma \vdash x : \forall(x : S) T$
then x is bound to some lambda value $\lambda(x : S') t$,
where $S <: S'$ and $\Gamma \vdash t : T$.

This looks straightforward to show.

But it isn't.

User-Definable Theories

In DOT, the subtyping relation is given in part by user-definable definitions

```
type T >: S <: U
```

This makes T a supertype of S and a subtype of U.

By transitivity, $S <: U$.

So the type definition above proves a subtype relationship which was potentially not provable before.

Bad Bounds

What if the bounds are non-sensical?

Example

```
type T >: Any <: Nothing
```

By the same argument as before, this implies that

```
Any <: Nothing
```

Once we have that, again by transitivity we get $S <: T$ for arbitrary S and T .

That is the subtyping relations collapses to a point.

Bad Bounds and Inversion

A collapsed subtyping relation means that inversion fails.

Example: Say we have a binding $x = \nu(x: T)...$

So in the corresponding environment Γ we would expect a binding $x: \mu(x: T)$.

But if every type is a subtype of every other type, we also get with subsumption that $\Gamma \vdash x: \forall(x: S)U!$.

Hence, we cannot draw any conclusions from the type of x . Even if it is a function type, the actual value may still be a record.

Can We Exclude Bad Bounds Statically?

Unfortunately, no.

Consider:

```
type S = { type A; type B >: A <: Bot }  
type T = { type A >: Top <: B; type B }
```

Individually, both types have good bounds. But their intersection does not:

```
type S & T == { type A >: Top <: Bot; type B >: Top <: Bot }
```

So, bad bounds can arise from intersecting types with good bounds.

But maybe we can verify all intersections in the program?

Bad Bounds Can Arise at Run-Time

The problem is that types can get more specific at run time.

Recall again preservation: If $\Gamma \vdash t : T$ and $t \longrightarrow u$ then $\Gamma \vdash u : T$.

Because of subsumption u might also have a type S which is a true subtype of T .

That S could have bad bounds (say, arising from an intersection).

Dealing With It: A False Start

Bad bounds make problems by combining the selection subtyping rules with transitivity.

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash x.A <: T} \quad (\text{SEL-}<:)$$

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash S <: x.A} \quad (<:-\text{SEL})$$

Can we “tame” these rules so that bad bounds cannot be exploited? E.g.

Dealing With It: A False Start

$$\frac{\Gamma \vdash x : \{A : S..T\} \quad \Gamma \vdash S <: T}{\Gamma \vdash x.A <: T} \quad (\text{SEL-}<:)$$

$$\frac{\Gamma \vdash x : \{A : S..T\} \quad \Gamma \vdash S <: T}{\Gamma \vdash S <: x.A} \quad (<:-\text{SEL})$$

Problem: we lose monotonicity. Tighter assumptions may yield worse results.

Dealing With It: Another False Start

Can we get rid of transitivity instead?

I.e. only use algorithmic version of subtyping rules?

We tried (for a long time), but got nowhere.

Transitivity seems to be essential for inversion lemmas and many other aspects of the proof.

Dealing With It: The Solution

Observation: To prove preservation, we need to reason at the top-level only about environments that arise from an actual computation. I.e. in

- If $\Gamma \vdash t : T$ and $t \longrightarrow u$ then $\Gamma \vdash u : T$.

The environment Γ corresponds to an evaluated `let` prefix, which binds variables to values.

And values have guaranteed good bounds because all type members are aliases.

$$\Gamma \vdash \{A = T\} : \{A : T..T\} \quad (\text{TYP-I})$$

Introducing Explicit Stores

We have seen that the `let` prefix of a term acts like a store.

For the proofs of progress and preservation it turns out to be easier to model the store explicitly.

A store is a set of bindings $x = v$ or variables to values.

The evaluation relation now relates terms and stores.

$$s \mid t \longrightarrow s' \mid t'$$

Evaluation $s \mid t \longrightarrow s' \mid t'$

$$\begin{array}{llll} s \mid x.a & \longrightarrow & s \mid t & \text{if } s(x) = \nu(x: T) \dots \{a = t\} \dots \\ s \mid xy & \longrightarrow & s \mid [z := y]t & \text{if } s(x) = \lambda(z: T) t \\ s \mid \text{let } x = y \text{ in } t & \longrightarrow & s \mid [x := y]t & \\ s \mid \text{let } x = v \text{ in } t & \longrightarrow & s, x = v \mid t & \\ s \mid \text{let } x = t \text{ in } u & \longrightarrow & s' \mid \text{let } x = t' \text{ in } u & \text{if } s \mid t \longrightarrow s' \mid t' \end{array}$$

Relationship between Stores and Environments

For the theorems and proofs of progress and preservation, we need to relate environment and store.

Definition: An environment Γ *corresponds* to a store s , written $\Gamma \sim s$, if for every binding $x = v$ in s there is an entry $\Gamma \vdash x : T$ where $\Gamma \vdash_! v : T$.

$\Gamma \vdash_! v : T$ is an exact typing relation.

We define $\Gamma \vdash_! x : T$ iff $\Gamma \vdash x : T$ by a typing derivation which ends in a (All-I) or ($\{\}$ -I) rule

(i.e. no subsumption or substructural rules are allowed at the toplevel).

Progress and Preservation, 2nd Take

Theorem (Preservation)

If $\Gamma \vdash t : T$ and $\Gamma \sim s$ and $s \mid t \longrightarrow s' \mid t'$, then there exists an environment $\Gamma' \supset \Gamma$ such that, one has $\Gamma' \vdash t' : T$ and $\Gamma' \sim s'$.

Theorem (Progress)

If $\Gamma \vdash t : T$ and $\Gamma \sim s$ then either t is an answer, or $s \mid t \longrightarrow s' \mid t'$, for some store s' , term t' .

Conclusion

DOT is a fairly small calculus that can express “classical” Scala programs and Module systems with abstract types.

It gives a principled foundation for nominal type systems.

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