Theory of Types and Programming Languages Spring 2022

Week 13

Foundations of Scala

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: Boolean; def head: T; def tail: List[T]
}
```

`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
```

Also written as a shorthand:

```
type 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: 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 standands for (path)-<u>D</u>ependent <u>O</u>bject <u>T</u>ypes.

Program:

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

Syntax

x, y, z	*Variable*
a, b, c	*Termmember*
A, B, C	*Typemember*
$S, T, U ::=$	*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 de

DOT Types

DOT	Scala	
$\top$	`Any`	Top type
$\bot$	`Nothing`	Bottom type
$\{a : T\}$	`{ def 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)
$\forall(x : S)\ T$	`(x: S) => T`	Dependent function type

DOT Definitions

Definitions make concrete record values.

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

Definitions are grouped together in an object

DOT	Scala	
$\nu(x : T)\, d$	`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}) -> (cX.A -> cX.A) -> 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}) -> 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 }

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..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.

```
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: Nothing..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 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:

```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 = ...

    nil = ...

    // definitions in next slide...
```

Encoding of Lists (ctd)

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

    nil = new { l: { head: Nothing, tail: Nothing } =>
        type A = Nothing
        isEmpty = bool.true
        head = l.head
        tail = l.tail
      }

  } // end of "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 }
    }

    cons: (x: {type A}) -> (hd: x.A) ->
      (tl: sci.List & { type A <: x.A }) ->
        sci.List & { type A = x.A }

    nil: sci.List & { type A = Nothing }

  }
```

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.

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

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 $\Gamma \vdash t : T$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad (\textsc{Var})$$

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

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

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

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

Type Assignment (2)

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

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

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

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

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

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

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

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

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

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

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

Subtyping $\Gamma \vdash T <: U$

$$\Gamma \vdash T <: \top \qquad (\textsc{Top})$$

$$\Gamma \vdash \bot <: T \qquad (\textsc{Bot})$$

$$\Gamma \vdash T <: T \qquad (\textsc{Refl})$$

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

$$\Gamma \vdash T \wedge U <: T \qquad (\textsc{And}_1\text{-}{<:})$$

$$\Gamma \vdash T \wedge U <: U \qquad (\textsc{And}_2\text{-}{<:})$$

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

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.

Foundations of Scala – Examples

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.

```scala
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
```

Foundations of Scala – Meta Theory

Foundations of Scala – 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   =   let x = (y: Bool) => y 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  |  v  |  let x = v 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?

3.1 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 $x = \nu(x : T) \ldots$.

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.

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

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

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

Dealing With It: A False Start

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

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

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.

$$\Gamma \vdash \{A = T\} : \{A : T..T\} \tag{\textsc{Typ-I}}$$

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.

$$s | t \longrightarrow s' | t'$$

Evaluation $s|t \longrightarrow s'|t'$

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

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 $s|t \longrightarrow s'|t'$, 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 $s|t \longrightarrow s'|t'$, for some store s', term t'.

Objects with Pattern Matching

An Example

"Typeful" encoding of expressions in a simple language

```
sealed class Expr[A]

case class IntLit(value: Int) extends Expr[Int]

case class First [A, B](pair: Expr[Pair[A, B]]) extends Expr[A]

case class Second[A, B](pair: Expr[Pair[A, B]]) extends Expr[B]

case class MkPair[A, B](lhs: Expr[A], rhs: Expr[B])
                                                extends Expr[Pair[A,B]]
```

An Example

Pattern matching on instances:

```
def eval[T](expr: Expr[T]): T =
  expr match {
    case e: IntLit => e.value
    case e: MkPair[_, _] => Pair(eval(e.lhs), eval(e.rhs))
    case e: First[T, _] => eval(e.pair).first
    case e: Second[_, T] => eval(e.pair).second
  }
```

An Example

Pattern matching on instances:

```
def eval[T](expr: Expr[T]): T =
  expr match {
    case e: IntLit => e.value
    case e: MkPair[_, _] => Pair(eval(e.lhs), eval(e.rhs))
    case e: First[T, _] => eval(e.pair).first
    case e: Second[_, T] => eval(e.pair).second
  }
```

The `IntLit` branch of this example should compile even though `expr.value` returns an `Int` where a value of type `T` is expected

in the `IntLit` branch, we discover `expr` is an `Expr[T]` *and* an `Expr[Int]`

and since `Expr` is invariant, this can only hold if `T` and `Int` are the same type.

An Example

Pattern matching on instances:

```
def eval[T](expr: Expr[T]): T =
  expr match {
    case e: IntLit => e.value
    case e: MkPair[_, _] => Pair(eval(e.lhs), eval(e.rhs))
    case e: First[T, _] => eval(e.pair).first
    case e: Second[_, T] => eval(e.pair).second
  }
```

The `IntLit` branch of this example should compile even though `expr.value` returns an `Int` where a value of type `T` is expected

in the `IntLit` branch, we discover `expr` is an `Expr[T]` *and* an `Expr[Int]`

and since `Expr` is invariant, this can only hold if `T` and `Int` are the same type.

In general, this form of reasoning is non-trivial.

Questions

How to reason about these things?

We need a form of *local* subtyping reasoning. . .

Questions

How to reason about these things?

We need a form of *local* subtyping reasoning...

Should this still work when `Expr` is made *covariant*?

```
sealed class Expr[+A]
```

How to justify it? (Soundness?)

Answers in DOT

Type parameters are “just” type members. . .

```
val g: {

  type Expr <: { type A }

  type IntLit <: Expr & { type A = Int; val value: Int }
  def newIntLit(i: Int): IntLit

} = new {

  type Expr = { type A }

  type IntLit = Expr & { type A = Int; val value: Int }
  def newIntLit(i: Int): IntLit =
    new { type A = Int; val value = i }

}
```

Answers in DOT

Pattern matching *uncovers* subtyping information by refining types

```
def eval(tp: { type T }, e: g.Expr & { type A = tp.T }): tp.T =
  e match {
    case e1: g.IntLit =>
      // Here, e1.A = e.A = tp.T and e1.A = Int so tp.T = Int
      e1.value
    ...
  }
```

Answers in DOT

Pattern matching *uncovers* subtyping information by refining types

```
def eval(tp: { type T }, e: g.Expr & { type A = tp.T }): tp.T =
  e match {
    case e1: g.IntLit =>
      // Here, e1.A = e.A = tp.T and e1.A = Int so tp.T = Int
      e1.value
    ...
  }
```

Covariant case:

```
def eval(tp: { type T }, e: g.Expr & { type A <: tp.T }): tp.T =
  e match {
    case e1: g.IntLit =>
      // Here, e1.A = e.A <: tp.T and e1.A = Int so tp.T :> Int
      e1.value
    ...
  }
```

Subtyping Reconstruction

We call this reasoning **subtyping reconstruction** (SR).

SR subsumes another problem found in functional programming languages related to generalized algebraic data types (GADTs)

Subtyping Reconstruction

We call this reasoning **subtyping reconstruction** (SR).

SR subsumes another problem found in functional programming languages related to generalized algebraic data types (GADTs)

SR is sound thanks to DOT$^{(*)}$ being sound!

Subtyping Reconstruction

We call this reasoning **subtyping reconstruction** (SR).

SR subsumes another problem found in functional programming languages related to generalized algebraic data types (GADTs)

SR is sound thanks to DOT[(*)] being sound!

[(*)] Actually an *extension* of the DOT presented here, with:

- Types rooted in *paths* (as in `x.a.b.c.A`), not just variables (as in `x.A`)
- Singleton types `x.type`, used for things like `scrut.type & IntLit`
- A representation of runtime class instance tests (basic `case` construct)

Subtyping Reconstruction

Paper currently in submission!

1

A case for DOT: Theoretical Foundations for Objects With Pattern Matching and GADT-style Reasoning

ANONYMOUS AUTHOR(S)

Many programming languages in the OO tradition have support for forms of pattern matching. Examples include Ceylon and Scala with the recent additions of Java, Kotlin as well as TypeScript and Flow. As we demonstrate, combining pattern matching with generic classes can result in puzzling type errors, for instance with certain approaches to representing ASTs such as database queries. We show that in order to correctly accept such examples, a compiler needs to support *subtyping reconstruction*: pattern matching on classes should recover subtyping information which was originally necessary to construct the class instance. We demonstrate cDOT, a calculus intended to serve as formal foundations for our approach. As cDOT is based on pDOT, itself a formal foundation for Scala, it remains applicable in the presence of advanced object-oriented features such as generic inheritance, type constructor variance, and first-class recursive modules. Subtyping reconstruction subsumes GADTs from ML-like languages. To demonstrate this, we show a variant of $\lambda_{2,G\mu}$, a constraint-based GADT calculus, and we encode it into cDOT.