ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Foundations of Scala

Foundations of Software

Martin Odersky

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   =   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?

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 $x = \nu(x \colon T) \ldots$.

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

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.

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

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

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

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 $G \sim s$ 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 a normal form, or $s \mid t \longrightarrow s' \mid t'$, for some store s', term t'.