Safe type-level abstraction in Scala

Adriaan Moors, Frank Piessens, Martin Odersky

FOOL 2008

(take 2)

Special thanks to Philipp Haller & his laptop, no thanks to my crashed mac

Scalina, briefly

- OO calculus that models Scala
  - based on nuObj
- Scala tightly integrates FP
  - encode parameterisation: abstract member
  - encode application: refinement
- Cannot faithfully encode higher-order (type-level) functions in nuObj
  - Scalina

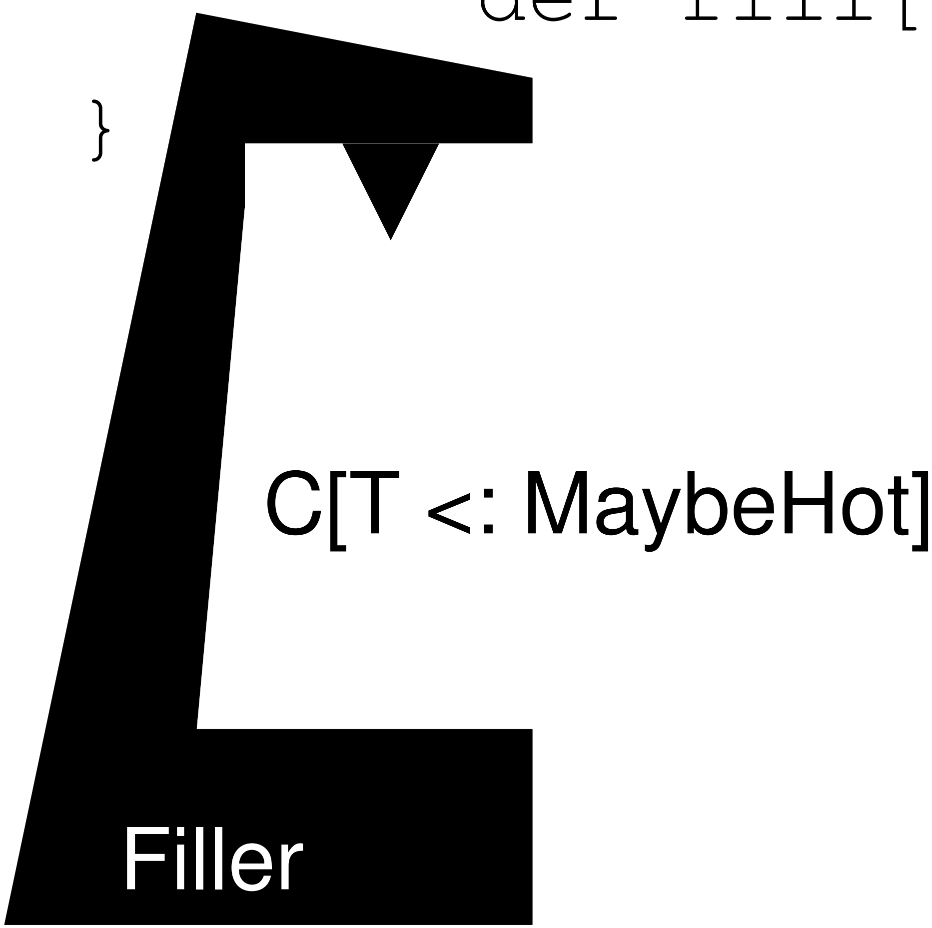
Generic Cup

(caution! contents may be hot)

```
class Cup[T <: MaybeHot]


class MaybeHot

class Tepid extends MaybeHot


// statically avoid lawsuits

class PaperCup[T <: Tepid]
```

A generic cup filler

```
class Filler[C[T <: MaybeHot]]{
          def fill[T <: MaybeHot]: C[T]
}
```



Filling paper cups considered harmful

Filler[PaperCup] is illegal (compiler rejects it)

C represents a type constructor that accepts any T <: MaybeHot

But: PaperCup can only be applied to T <: Tepid

Encoding higher-kinded types

```
trait Cup{ type T <: MaybeHot }

trait PaperCup{ type T <: Tepid }

trait Filler {
   type C <: {type T <: MaybeHot}
  def fill[U <: MaybeHot]: C{type T = U}
}
```

Encoding higher-kinded types

```
trait PaperCup{ type T <: Tepid }


class MyFiller extends Filler {
   type C = PaperCup
  def fill[U <: MaybeHot]: C{type T = U}
    = return type is not (necessarily) inhabited!

}
```

Error is not detected until we try to implement `fill`!

How could this have happened?

```
  type C <: {type T <: MaybeHot}

trait PaperCup{ type T <: Tepid }

  type C = PaperCup
```

Solution in pseudo-Scalina

```
trait PaperCup{ untype T <: Tepid }

trait Filler {
  type C <: { untype T <: MaybeHot }
}

class MyFiller extends Filler {
  type C = PaperCup // illegal
  // expected Tepid >: MaybeHot
}
```

Error detected just as earlier as with direct support.

Revenge of the un-members

- As in nuObj, by default, the classifier of an abstract member may be strengthened in a subtype (covariance)
- Instead of writing “untype”, we wrap the member's classifier in Un[...]
- This flips the variance: *contravariant* position
- Essential for the encoding of functions

From an OO perspective

- Un-members were introduced to encode arguments to functions
- Members and un-members form the two halves of the contract implied by a class
  - members are provided (directly or by a subtype)
  - un-members are expected (supplied by client)

Variance and *variance*

- Co/contravariance in previous slides differs from variance in `class List[+T]`
- Scala's definition-site variance annotations specify how *constructed types* *(from the same type constructor)* relate, based on the supplied type arguments
- `A <: B => List[A] <: List[B]`

Variance and *variance*

- Un[...] flips the variance that relates *type constructors* based on the *classifiers* of their members (have no value yet)

```
type List = {type T: Un[*] }
type NumList = {type T: Un[In(Number)] }
```

- `NumList <: List` iff `Un[In(Number)] <: Un[*]`
  - iff `* <: In(Number)` (iff pigs can fly)

Different flavours of abstraction

- value abstracting over value
  - `new {val arg: Un[T]; val apply: T'}`
- value abstracting over type
  - `new {type Arg: Un[K]; val apply: T'}`
- type abstracting over type
  - `{type Arg: Un[K]; type Apply: K'}`

Impact on type system

```
{ u =>
  type Id = {self: []u.Id =>
    type T: Un[*]
    val arg: Un[self.T]
    val apply: self.T = self.arg
  }
  val id: u.Id = new u.Id
  val test: u.Int
    = (id <{T = u.Int} <{arg = 1}).apply
```

this self type reflects the assumption that un-members have been refined before any other member is accessed

Kinds

- `In(T1, T2)`: interval kind
  - depends on types
  - cf. Powertype (Cardelli), singleton kinds (Harper et al)
- `Un[K]`: classifier for type un-member
- `Nominal(R)`: nominal branding
- (`Struct(R)`: replace by singleton kind?)
- `Concrete(R)`: generalising type selection

Flirting with type-level computation

- Concrete kind (cf. objects can't have abstract members)
  - `p.type: Concrete(T)` if `p: T` and `T: Struct(R)`
  - `T: Concrete(R)` if `T: Struct(R)` and all members in `R` are concrete
  - `T#L: Concrete(R)` if `T` declares a type member with label `L` and kind `Concrete(R)`
- Generalising type selection
  - `T#L: K` if `T: Concrete({type L: K})`

Flirting with type-level computation

```
trait TBool { type If[t, f] }
trait TTrue extends TBool {
  type If[t, f] = t }
trait TFalse extends TBool {
  type If[t, f] = f }
// somewhere in the program:
type Flag <: TBool
type Decide = Flag#If[then, else]
```

doesn't work: `Flag = TBool` --> ??

Flirting with type-level computation

```
type Flag : Concrete({type If:
  Concrete({
    type T : Un[*]
    type F : Un[*]
    type Apply : *})})
```

```
(Flag#If<{T=then}<{F=else})#Apply
```

Summary: design goals

- Uniformity
  - (un-)members abstract over types and values
  - both objects and types may contain un-members
  - member: label, classifier, (RHS)
  - soundness at type level *& kind level*
- Faithfully encode FP concepts
  - polymorphic functions are values
  - well-kinded type applications never go wrong

Conclusion

- Scalina's first goal was to improve the integration of FP into Scala's underlying formalism
  - simple idea: introduce contravariance
- Scalina's future includes (I hope)
  - state (notion of paths already in place)
  - virtual classes (type members late bound)
  - type-level computation (kind soundness >important)
  - mechanized meta-theory (Coq tutorial gave me new hope)

Solution in Scalina

```
trait PaperCup{ type T: Un[In(Tepid)] }

trait Filler {
  type C: Struct({type T: Un[In(MaybeHot)]})
}

class MyFiller extends Filler {
  type C = PaperCup // illegal
  // NOT (Un[In(Tepid)] <: Un[In(MaybeHot)])
}
```

Error detected just as earlier as with direct support.