Type-Level Computations in Scala

Stefan Zeiger

Typesafe

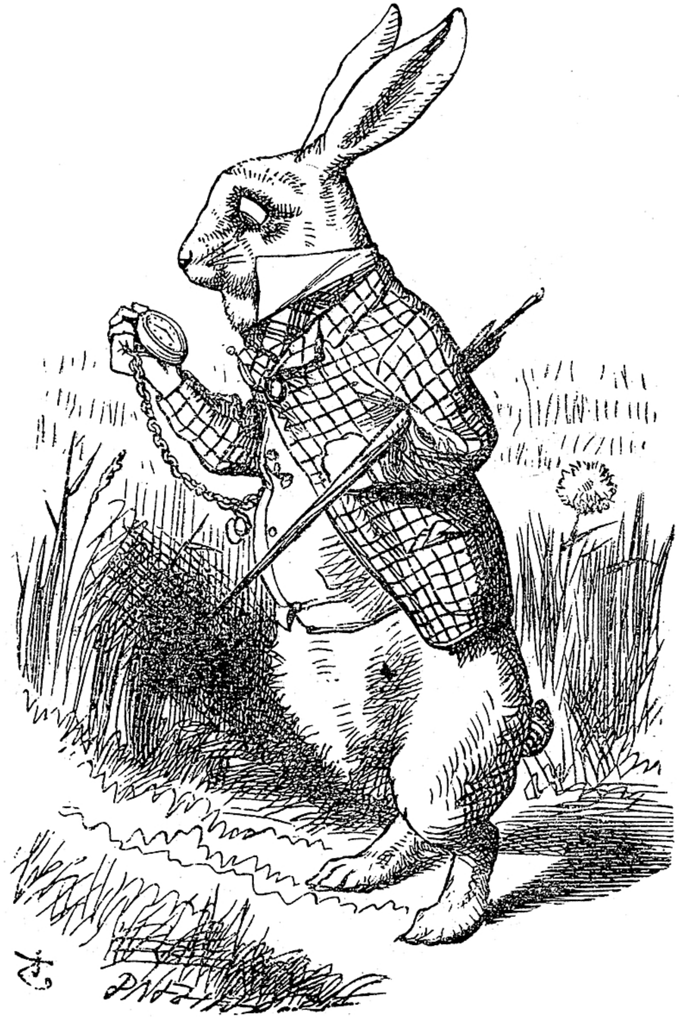
Motivation

- Heterogeneous collection types (HList, HArray)

```scala
val l1 = 42 :: "foo" :: Some(1.0) :: "bar" :: HNil
val i: Int = l1.head
val s: String = l1.tail.head
```

- Unlike Scala's tuples:
  - No size limit
  - Can abstract over arity
- Requirement: No runtime overhead
  - So no implicits!

Down The Rabbit Hole



Booleans

A Simple Boolean Type

Term-Level

```
sealed trait Bool {
  def && (b: Bool): Bool
  def || (b: Bool): Bool
  def ifElse[B](t: => B, f: => B): B
}

lazy val True: Bool = new Bool {
  def && (b: Bool) = b
  def || (b: Bool) = True
  def ifElse[B](t: => B, f: => B) = t
}

lazy val False: Bool = new Bool {
  def && (b: Bool) = False
  def || (b: Bool) = b
  def ifElse[B](t: => B, f: => B) = f
}
```

There's a Smalltalk in your Scala!

From Values to Types

```
sealed trait Bool {


}

lazy val True: Bool = new Bool ...




lazy val False: Bool = new Bool ...
```

From Values to Types

```
sealed trait Bool {

}
```

`: True.type`

```
object True extends Bool ...
```

`: False.type`

```
object False extends Bool ...
```

From Values to Types

```
sealed trait Bool {

}
```

: True.type

```
object True extends Bool ...
```

: False.type

```
object False extends Bool ...

type True = True.type
type False = False.type
```

Translating Methods

```
sealed trait Bool {
  def && (b: Bool): Bool
  def || (b: Bool): Bool
  def ifElse[B](t: => B, f: => B): B
}
```

Translating Methods

```
sealed trait Bool {
  def && (b: Bool): Bool
  def || (b: Bool): Bool
  def ifElse[B](t: => B, f: => B): B
}
```

↓

```
sealed trait Bool {
  type &&
  type ||

}
```

Translating Methods

```
sealed trait Bool {
  def && (b: Bool): Bool
  def || (b: Bool): Bool
  def ifElse[B](t: => B, f: => B): B
}
```

```
sealed trait Bool {
  type && [B <: Bool] <: Bool
  type || [B <: Bool] <: Bool

}
```

Translating Methods

```
sealed trait Bool {
  def && (b: Bool): Bool
  def || (b: Bool): Bool
  def ifElse[B](t: => B, f: => B): B
}
```

```
sealed trait Bool {
  type && [B <: Bool] <: Bool
  type || [B <: Bool] <: Bool
  type IfElse[B, T <: B, F <: B] <: B
}
```

Translating Methods

```scala
sealed trait Bool {
  type && [B <: Bool] <: Bool
  type || [B <: Bool] <: Bool
  type IfElse[B, T <: B, F <: B] <: B
}

object True extends Bool {
  type && [B <: Bool] = B
  type || [B <: Bool] = True
  type IfElse[B, T <: B, F <: B] = T
}

object False extends Bool {
  type && [B <: Bool] = False
  type || [B <: Bool] = B
  type IfElse[B, T <: B, F <: B] = F
}
```

Testing

```
assert    ( (False  &&  False ) ==  False )
assert    ( (False  &&  True  ) ==  False )
assert    ( (True   &&  False ) ==  False )
assert    ( (True   &&  True  ) ==  True  )

assert    ( (False  ||  False ) ==  False )
assert    ( (False  ||  True  ) ==  True  )
assert    ( (True   ||  False ) ==  True  )
assert    ( (True   ||  True  ) ==  True  )

assert    ( False. ifElse     (1, 2)        == 2       )
assert    ( True.  ifElse     (1, 2)        == 1       )
```

Testing

```
assert    ( (False  &&  False ) ==  False )
```

a b c → a.b(c)

```
assert    ( (False. && (False)) ==  False )
```

```
implicitly[ (False# && [False]) =:= False ]
```

A B C → B[A, C]

Testing

```
assert    ( (False. && (False)) ==  False )
assert    ( (False. && (True )) ==  False )
assert    ( (True.  && (False)) ==  False )
assert    ( (True.  && (True )) ==  True  )

assert    ( (False. || (False)) ==  False )
assert    ( (False. || (True )) ==  True  )
assert    ( (True.  || (False)) ==  True  )
assert    ( (True.  || (True )) ==  True  )

assert    ( False. ifElse[Int](1, 2)        == 2        )
assert    ( True.  ifElse[Int](1, 2)        == 1        )
```

Testing

```
implicitly[ (False# && [False]) =:= False ]
implicitly[ (False# && [True ]) =:= False ]
implicitly[ (True#  && [False]) =:= False ]
implicitly[ (True#  && [True ]) =:= True  ]

implicitly[ (False# || [False]) =:= False ]
implicitly[ (False# || [True ]) =:= True  ]
implicitly[ (True#  || [False]) =:= True  ]
implicitly[ (True#  || [True ]) =:= True  ]

implicitly[ False# IfElse[Any, Int, String] =:= String ]
implicitly[ True#  IfElse[Any, Int, String] =:= Int    ]
```

Natural Numbers

Natural Numbers

```
sealed trait Nat {

  def ++ = new Succ(this)
}

final object Zero extends Nat {

}


final class Succ(n: Nat) extends Nat {

}

val _0 = Zero
val _1 = _0.++
val _2 = _1.++
```

Peano Numbers

Natural Numbers

!

```
sealed trait Nat {

  type ++ = Succ[this.type]
}

final object Zero extends Nat {

}
type Zero = Zero.type

final class Succ[N <: Nat] extends Nat {

}

type _0 = Zero
type _1 = _0 # ++
type _2 = _1 # ++
```

Natural Numbers

```
sealed trait Nat {
  type This >: this.type <: Nat
  type ++ = Succ[This]
}

final object Zero extends Nat {
  type This = Zero
}
type Zero = Zero.type

final class Succ[N <: Nat] extends Nat {
  type This = Succ[N]
}

type _0 = Zero
type _1 = _0 # ++
type _2 = _1 # ++
```

Testing

```
assert    ( False. ifElse[Int]( 1,  2) ==   2 )
assert    ( True.  ifElse[Int]( 1,  2) ==   1 )
```

```
implicitly[ False# IfElse[Nat, _1, _2] =:= _2 ]
implicitly[ True#  IfElse[Nat, _1, _2] =:= _1 ]
```

Translation to Types

- ADTs (Algebraic Data Types)
- Different basic values and classes become types
- Purely-functional design
- Polymorphic dispatch (on receiver)
  - No *match*, *if...else*, etc.

Translation Rules

- ADT Values: val → object
- Members: def x / val x → type X
- def f(x) → type F[X]
- a.b → A#B
- x: T → X <: T
- new A(b) → A[B]

Recursion

Thinking Recursively: Addition

```
sealed trait Nat {
  type + [_ <: Nat] <: Nat
}

final object Zero extends Nat {
  type + [X <: Nat] = X
}

final class Succ[N <: Nat] extends Nat {
  type + [X <: Nat] = Succ[N # + [X]]
}
```

Thinking Recursively: Multiplication

```
sealed trait Nat {
  type * [_ <: Nat] <: Nat
}

final object Zero extends Nat {
  type * [X <: Nat] = Zero
}

final class Succ[N <: Nat] extends Nat {
  type * [X <: Nat] = (N # * [X]) # + [X]
}
```

Thinking Recursively: Exponentiation

^ is not commutative (order matters)

```
sealed trait Nat {
  type ^ [X <: Nat] = X # Flip_^ [This]
  type Flip_^ [_ <: Nat] <: Nat
}

final object Zero extends Nat {
  type Flip_^ [X <: Nat] = _1
}

final class Succ[N <: Nat] extends Nat {
  type Flip_^ [X <: Nat] = (N # Flip_^ [X]) # * [X]
}
```

Thinking Recursively: Folds

```scala
sealed trait Nat {
  def + (x: Nat): Nat =
    fold[Nat]((n => new Succ(n)), x)

  def fold[U](f: U => U, z: => U): U
}

final object Zero extends Nat {
  def fold[U](f: U => U, z: => U) = z
}

final class Succ(n: Nat) extends Nat {
  def fold[U](f: U => U, z: => U) = f(n.fold(f, z))
}
```

Church Numerals

Type Functions

```scala
sealed trait Nat {
  def fold[U](f: U => U, z: => U): U
}
```

```scala
sealed trait Nat {
  type Fold[U, F[_ <: U] <: U, Z <: U] <: U
}

final object Zero extends Nat {
  type Fold[U, F[_ <: U] <: U, Z <: U] = Z
}

final class Succ[N <: Nat] extends Nat {
  type Fold[U, F[_ <: U] <: U, Z <: U] = F[N#Fold[U, F, Z]]
}
```

Type Lambdas (Scala 2.11)

```
def + (x: Nat) =
  fold[Nat]((n => new Succ(n)), x)
```

```
def + (x: Nat) =
  fold[Nat]((new { def l(n: Nat) = new Succ(n) }).l _, x)
```

```
type + [X <: Nat] =
  Fold[Nat, ({ type L[N <: Nat] = Succ[N] })#L, X]
```

Type Lambdas (Experimental)

```
def + (x: Nat) =
  fold[Nat]((n => new Succ(n)), x)
```

```
type + [X <: Nat] =
  Fold[Nat, [N <: Nat] => Succ[N], X]
```

Limits of Recursion

```
sealed trait Nat {
  type Switch[U, F[_ <: Nat] <: U, Z <: U] <: U
  type Fold[U, F[_ <: U] <: U, Z <: U] =
    Switch[U,
      ({ type L[P <: Nat] = P#Fold[U, F, Z] })#L, Z]
}
```

Not allowed!

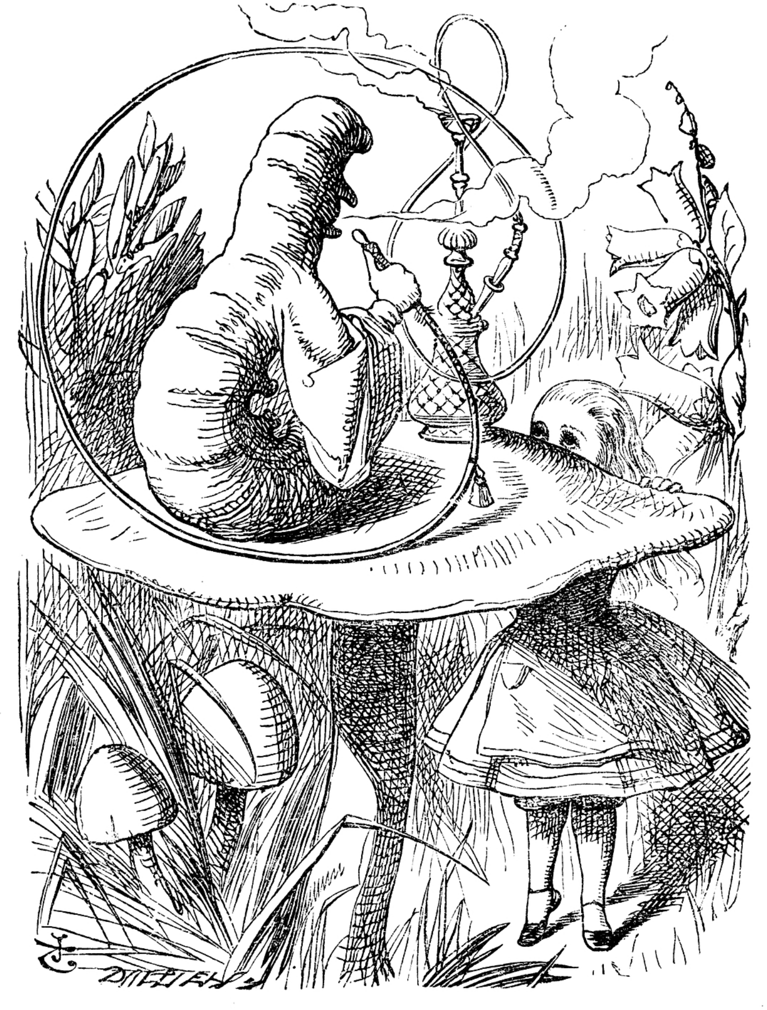
```
final object Zero extends Nat {
  type Switch[U, F[_ <: U] <: U, Z <: U] = Z
}

final class Succ[N <: Nat] extends Nat {
  type Switch[U, F[_ <: U] <: U, Z <: U] = F[N]
}
```

Recursion

- No recursive types allowed
- But recursion through traits
- Fixpoint combinators are possible (but ugly)

"You'll Get Used To It In Time"



Combining Term- And Type-Level

Natural Numbers

```
object Zero extends Nat {
  type This = Zero.type
  type Fold[U, F[_ <: U] <: U, Z <: U] = Z
  def value = 0
}

class Succ[N <: Nat](val value: Int) extends Nat {
  type This = Succ[N]
  type Fold[U, F[_ <: U] <: U, Z <: U] =
    F[N#Fold[U, F, Z]]
}
```

Efficient Natural Numbers

```
sealed trait Nat {
  def value: Int
  // This, Fold, equals, hashCode, toString...

  type + [N <: Nat] =
    Fold[Nat, ({ type L[X <: Nat] = Succ[X] })#L, N]
  def + [T <: Nat](n: T): +[T] =
    Nat._unsafe[+[T]](value + n.value)
}
```

Hybrid operation: Term- & type-level

```
object Nat {
  def _unsafe[T <: Nat](v: Int) =
    ( if(v == 0) Zero else new Succ(v) ).asInstanceOf[T]
}
```

Heterogeneous Lists

HList Basics

```
sealed abstract class HList {
  type This >: this.type <: HList
  type Head
  type Tail <: HList

  def head: Head
  def tail: Tail
}

final class HCons[H, T <: HList](val head: H, val tail: T)
extends HList {
  type This = HCons[H, T]
  type Head = H
  type Tail = T
}

object HNil extends HList { ... }
```

HList Basics

```scala
sealed abstract class HList {
  type This >: this.type <: HList
  type Head
  type Tail <: HList

  def head: Head
  def tail: Tail
}

object HNil extends HList {
  type This = HNil.type
  type Head = Nothing
  type Tail = Nothing
  def head = throw new NoSuchElementException("HNil.head")
  def tail = throw new NoSuchElementException("HNil.tail")
}
```

Cannot fail to compile

HList Length

```
sealed abstract class HList {
  def isDefined: Boolean == this != HNil

  def size: Int = {
    var i = 0
    var h = this
    while(h.isDefined) { i += 1; h = h.tail }
    i
  }

  type Length =
    Fold[Nat, ({ type L[_, Z <: Nat] = Z# ++ })#L, Zero]

  def length: Length = Nat._unsafe[Length](size)
}
```

HList Fold (Type-Level)

```
sealed abstract class HList {
  type Fold[U, F[_, _ <: U] <: U, Z <: U] <: U
}

final class HCons[H, T <: HList](val head: H, val tail: T)
extends HList {
  type Fold[U, F[_, _ <: U] <: U, Z <: U] =
    F[Head, T#Fold[U, F, Z]]
}

object HNil extends HList {
  type Fold[U, F[_, _ <: U] <: U, Z <: U] = Z
}
```

HList Fold (Term-Level)

```
sealed abstract class HList {
  def fold[U](f: (Any, U) => U, z: => U): U
}
```

Function2

```
final class HCons[H, T <: HList](val head: H, val tail: T)
extends HList {
  def fold[U](f: (Any, U) => U, z: => U) =
    f(head, tail.fold[U](f, z))
}

object HNil extends HList {
  def fold[U] (f: (Any, U) => U, z: => U) = z
}
```

Hybrid Functions

```scala
trait Function2[-T1, -T2, +R] {
  def apply(v1: T1, v2: T2): R
}
```

```scala
trait TypedFunction2[-T1, -T2, +R,
                     F[_ <: T1, _ <: T2] <: R] {
  def apply[P1 <: T1, P2 <: T2](p1: P1, p2: P2): F[P1, P2]
}
```

HList Fold (Hybrid)

```
sealed abstract class HList {
  def fold[U, F[_, _ <: U] <: U, Z <: U]
    (f: TypedFunction2[Any, U, U, F], z: Z): Fold[U, F, Z]
}

final class HCons[H, T <: HList](val head: H, val tail: T)
extends HList {
  def fold[U, F[_, _ <: U] <: U, Z <: U]
    (f: TypedFunction2[Any, U, U, F], z: Z): Fold[U, F, Z] =
      f.apply[Head, T#Fold[U, F, Z]]
        (head, tail.fold[U, F, Z](f, z))
}

object HNil extends HList {
  def fold[U, F[_, _ <: U] <: U, Z <: U]
    (f: TypedFunction2[Any, U, U, F], z: Z) = z
}
```

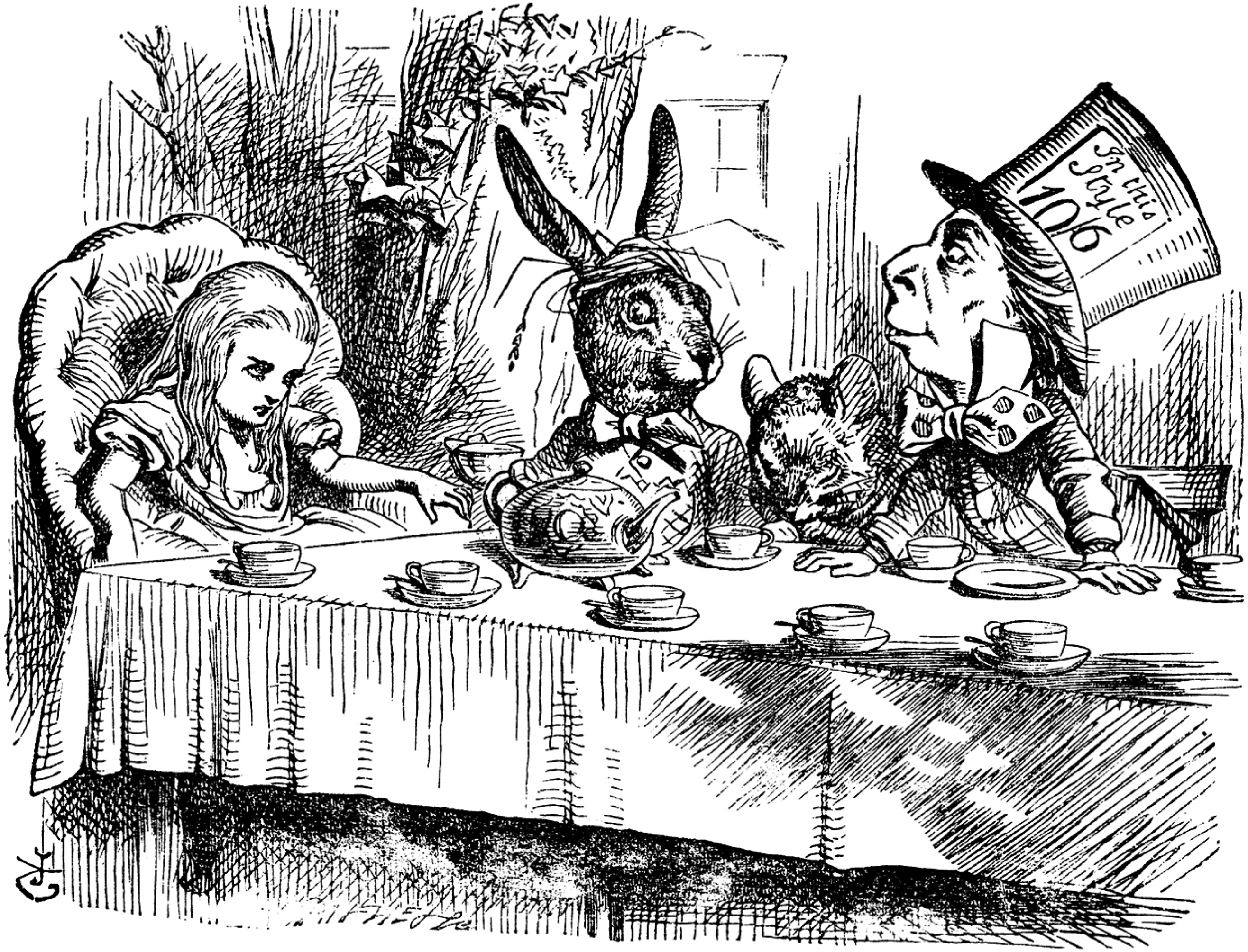
HList Length With Fold

```
sealed abstract class HList {
  def length: Length =
    fold[
      Nat,
      ({ type L[_, Z <: Nat] = Z# ++ })#L,
      Zero.type
    ](
      new TypedFunction2[Any, Nat, Nat,
            ({ type L[_, Z <: Nat] = Z# ++ })#L] {
        def apply[P1 <: Any, P2 <: Nat](p1: P1, p2: P2) =
          p2.++
      },
      Zero
    )
}
```

Limitations

- No recursive types (but recursion through traits)
- Type lambdas and typed functions are cumbersome
- The type language is total
- No unification

Time For Some Tea



Typesafe

@StefanZeiger

https://github.com/szeiger/ErasedTypes