The **DOT** Calculus

(**D**ependent **O**bject **T**ypes)

Nada Amin

Scala Days

June 18, 2014

DOT: Dependent Object Types

- DOT is a core calculus for path-dependent types.
- Goals
  - simplify Scala's type system by desugaring to DOT
  - simplify Scala's type inference by relying on DOT
  - prove that DOT is type-safe

Types in Scala and DOT

Types in Scala

modular	named type	`scala.collection.BitSet`
	compound type	`Channel with Logged`
	refined type	`Channel { def close(): Unit }`
functional	parameterized type	`List[String]`
	existential type	`List[T] forSome { type T }`
	higher-kinded type	`List`

Reducing Functional to Modular?

- type parameter to type member

```
class List[Elem] {} /*vs*/ class List { type Elem }
```

- parameterized type to refined type

```
List[String] /*vs*/ List { type Elem = String }
```

- existential type?

```
List[_] /*or*/ List[T] forSome { type T } /*~vs~*/ List
```

- higher-kinded type?

```
List /*~vs~*/ List
```

Bonus: **D**on't **R**epeat **Y**ourself

```
        new HashMap[String, List[Int]]
with SynchronizedMap[String, List[Int]]

          /*vs*/

        new HashMap[String, List[Int]]
with SynchronizedMap
```

Type Members and Modularity (Example)

```
trait Symbols {
  type Type
  // a symbol has a type ...
  trait Symbol { def tpe: Type }
}
trait Types {
  type Symbol
  // ... and a type has a symbol
  trait Type { def sym: Symbol }
}
// ... but the references are not hard-coded!
// putting the cake together:
trait SymbolTable extends Symbols with Types
```

Type Members and Data Abstraction (Example)

```
trait HeapModule {
  type Heap
  type Elem
  def ord: Ordering[Elem]
  def empty: Heap
  def isEmpty(h: Heap): Boolean
  def insert(x: Elem, h: Heap): Heap
  def findMin(h: Heap): Elem
  def deleteMin(h: Heap): Heap
  def merge(h1: Heap, h2: Heap): Heap
}
trait BinomialHeapModule extends HeapModule {
  type Rank = Int
  case class Node(x: Elem, r: Rank, c: List[Node])
  override type Heap = List[Node]
  // ... implementation ...
}
```

Path-Dependent Types and Binary Methods (Example)

```
val m1: HeapModule = ...
val m2: HeapModule = ...
// in Scala REPL
> m1.merge(m1.empty, m2.empty)
// error: type mismatch;
// found   : m2.Heap
// required: m1.Heap
//               m1.merge(m1.empty, m2.empty)
//                                  ^
```

Types in DOT

types `S, T, U`

- path-dependent type `p.L`
- refined type `T { z => D }`
- intersection `T & T`
- union `T | T`
- top `Any`
- bottom `Nothing`

declarations `D`

- type declaration `type L >: S <: U`
- field declaration `val l: U`
- method declaration `def m(x: S): U`

Type Inference

Type Inference in Scala (Least Upper Bound?)

```
trait A { type T <: A }
trait B { type T <: B }
trait C extends A with B { type T <: C }
trait D extends A with B { type T <: D }
// in Scala, lub(C, D) is an infinite sequence
A with B { type T <: A with B { type T <: ... } }

// in Scala REPL
> val o = if (true) (new C{}) else (new D{})
o: A with B{type T <: A with B} = ...
> val o:A with B{type T<:A with B {type T<:A with B}} =
           if (true) (new C{}) else (new D{})
o: A with B{type T <: A with B{type T <: A with B}} = ...
```

Type Inference in Scala (Working too Hard too Soon?)

```
scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[scala.col
lection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[scala.collection.m
utable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[scala.collection.mutable.It
erable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[Cloneable with Mutable with scala.co
llection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with scala.collection.mutable.Builder[(Int, scala.col
lection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clear
able with Equals] with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: Cloneable with Mutable with scala.collection.gener
ic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}] with scala.collection.mutable.Builder[(Int, scala.collection.mutab
le.Set[Int]) with Int,scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutab
le.Cloneable[Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with
 scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collection.generic.Sh
rinkable[Int] with scala.collection.generic.Clearable with Equals] with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: C
loneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}] with Int => Any w
ith scala.collection.generic.Shrinkable[Int]{def seq: scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with
 Int] with scala.collection.mutable.Cloneable[Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.gen
eric.Clearable with Equals] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable with Mutable
 with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with Int => Any with scala.collection.ge
neric.Shrinkable[Int]{def seq: Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable
 with Equals}}] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,scala.collection.mutable.Iterable[_
 >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[scala.collection.mutable.Iterable[_ >: (Int,
 scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[Cloneable with Mutable with scala.collection.generic.
Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.S
et[Int]) with Int,Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals]
 with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: Cloneable with Mutable with scala.collection.generic.Shrinkable[Int
] with scala.collection.generic.Clearable with Equals}] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with
 Int,scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[Clon
eable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with scala.collection
.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] wit
h scala.collection.generic.Clearable with Equals] with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: Cloneable with Mut
able with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}] with Int => Any with scala.collect
ion.generic.Shrinkable[Int]{def seq: scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.c
ollection.mutable.Cloneable[Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable wi
th Equals] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collec
tion.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with Int => Any with scala.collection.generic.Shrinkable[
Int]{def seq: Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}}] w
ith Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable
.Set[Int]) with Int] with scala.collection.mutable.Cloneable[scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]
) with Int] with scala.collection.mutable.Cloneable[Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collect
ion.generic.Clearable with Equals] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable with M
utable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with Int => Any with scala.collec
tion.generic.Shrinkable[Int]{def seq: Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Cl
earable with Equals}] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,scala.collection.mutable.Iter
 -:---  lub.scala      Top (1,0)      (Scala Hi Undo-Tree)
Mark set
```

Type Inference in Scala (Working too Hard too Soon?)

```
der[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.colle
ction.generic.Clearable with Equals] with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: Cloneable with Mutable with sca
la.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}] with Int => Any with scala.collection.generic.S
hrinkable[Int]{def seq: scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mut
able.Cloneable[Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] wi
th scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collection.generic.
Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq:
 Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}}] with scala.col
lection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mut
able.Set[Int]) with Int] with scala.collection.mutable.Cloneable[scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[
Int]) with Int] with scala.collection.mutable.Cloneable[Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.col
lection.generic.Clearable with Equals] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable wi
th Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with Int => Any with scala.co
llection.generic.Shrinkable[Int]{def seq: Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generi
c.Clearable with Equals}] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,scala.collection.mutable.
Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[Cloneable with Mutable with scala.
collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with scala.collection.mutable.Builder[(Int, scala.c
ollection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Cle
arable with Equals] with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: Cloneable with Mutable with scala.collection.gen
eric.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}] with Int => Any with scala.collection.generic.Shrinkable[Int]{de
f seq: scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[Cl
oneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with scala.collecti
on.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] w
ith scala.collection.generic.Clearable with Equals] with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: Cloneable with M
utable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}}] with Int => Any with scala.coll
ection.generic.Shrinkable[Int]{def seq: scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scal
a.collection.mutable.Cloneable[scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collect
ion.mutable.Cloneable[Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equ
als] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collection.g
eneric.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with Int => Any with scala.collection.generic.Shrinkable[Int]{d
ef seq: Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}] with sca
la.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,scala.collection.mutable.Iterable[_ >: (Int, scala.collecti
on.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[Cloneable with Mutable with scala.collection.generic.Shrinkable[Int]
 with scala.collection.generic.Clearable with Equals] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with I
nt,Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with Int => An
y with scala.collection.generic.Shrinkable[Int]{def seq: Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.co
llection.generic.Clearable with Equals}] with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: scala.collection.mutable.It
erable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[Cloneable with Mutable with scala.co
llection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with scala.collection.mutable.Builder[(Int, scala.col
lection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clear
able with Equals] with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: Cloneable with Mutable with scala.collection.gener
ic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}}}}
-:---  lub.scala      Bot (1,21480)  (Scala Undo-Tree)
```

Type Inference in Scala (Working too Hard too Soon?)

```
import scala.collection.mutable.{Map => MMap, Set => MSet}
val ms: MMap[Int, MSet[Int]] = MMap.empty

// in Scala REPL
> if (!ms.contains(1)) ms += 1 -> MSet(1) else ms(1) += 1
res0: ... (796 characters) ...
> :t res0
    : ... (21481 characters) ...
```

- Inspired by a bug report
  SI-5862: very slow compilation due to humonguous LUB.
- The character lengths reported are for Scala 2.11 (*after* the fix).
- In Dotty, type inference can be lazy thanks to the native unions (for least upper bounds) and intersections (for greatest lower bounds) of the core calculus (DOT).

Meta-Theory

Type Safety: "Well-typed programs can't go wrong."

```
// ... at least in some ways ...
// JavaScript REPL
> var x = {foo: 'hello'}
undefined
> x.foo
"hello"
> x.bar
undefined
> x.bar.boo
// TypeError: Cannot read property 'boo' of undefined
> x.foo('hello')
// TypeError: string is not a function
```

Type Safety: Syntactic Recipe (Concepts)

small-step	operational semantics ("show all your steps")
irreducible	terms, stuck terms vs values
progress	theorem: if a term type-checks, then it can take a step or it is a value (but not stuck!)
preservation	theorem: if a term type-checks and steps, then the new term also type-checks with the same type (or subtype)
type safety	= progress + preservation

Preservation Challenge: Branding

```
trait Brand {
  type Name
  def id(x: Any): Name
}
// in REPL
val brand: Brand = new Brand {
  type Name = Any
  def id(x: Any): Name = x
}
brand.id("hi"): brand.Name // ok
"hi": brand.Name // error but probably sound
val brandi: Brand = new Brand {
  type Name = Int
  def id(x: Any): Name = 0
}
brandi.id("hi"): brandi.Name // ok
"hi": brandi.Name // error and probably unsound
```

Theory: Subtyping of Path-Dependent Types

- `/*If*/ p.L /*has lower bound*/ Sp /*and upper bound*/ Up`
  - `S <: p.L /*if*/ S <: Sp`
  - `p.L <: U /*if*/ Up <: U`

Preservation Challenge: Path equality

```
trait B { type X; val l: X }
val b1: B = new B { type X = String; val l: X = "hi" }
val b2: B = new B { type X = Int;    val l: X = 0    }
trait A { val i: B }
val a = new A { val i: B = b1 }
println(a.i.l : a.i.X) // ok
println(b1.l  : b1.X)  // ok
println(b1.l  : a.i.X) // error: type mismatch;
// found   : b1.l.type (with underlying type b1.X)
// required: a.i.X
// abstractly, would need to show
Any  <: Nothing // lattice collapse!
// to show
b1.X <: a.i.X
// because
U of b1.X = Any <: Nothing = S of a.i.X
println(b2.l  : a.i.X) // error and probably unsound
```

Challenge: Subtyping Transitivity

- `S <: T <: U /*imply*/ S <: U`

Subtyping Transitivity and Path-Dependent Types

- `S <: p.L <: U`
- `p.L /*has lower bound*/ Sp /*and upper bound*/ Up`
- `S  <: Sp`
- `Up <: U`

Subtyping Transitivity and Path-Dependent Types with Bad Bounds

- `/*Suppose*/ p.L /*has bad bounds, e.g.*/`
  - `p.L /*has lower bound*/ Any /*and upper bound*/ Nothing`
  - `S  <: Any     /*for any*/ S`
  - `U  <: Nothing /*for any*/ U`
  - `S <: p.L <: U /*imply*/ S <: U /*for any*/ S, U`
- Lattice collapse!

Path-Dependent Types (Recap Example)

```
trait Animal { type Food; def gets: Food
               def eats(food: Food) {}; }
trait Grass; trait Meat
trait Cow extends Animal with Meat {
  type Food = Grass; def gets = new Grass {} }
trait Lion extends Animal {
  type Food = Meat;  def gets = new Meat {} }
val leo = new Lion {}
val milka = new Cow {}
leo.eats(milka) // ok
val lambda: Animal = milka
lambda.eats(milka) // type mismatch
// found : Cow
// required: lambda.Food
lambda.eats(lambda.gets) // ok
```

Path-Dependent Types (Recap Example Continued)

```
def share(a1: Animal)(a2: Animal)
         (bite: a1.Food with a2.Food) {
  a1.eats(bite)
  a2.eats(bite)
}
val simba = new Lion {}
share(leo)(simba)(leo.gets) // ok
share(leo)(lambda)(leo.gets) // error: type mismatch
// found : Meat
// required: leo.Food with lambda.Food

// Observation:
// We don't know whether the parameter type of
share(lambda1)(lambda2)_
// is realizable until run-time.
```

Realizability of Intersection Types at Run-Time

```
val lambda1: Animal = new Lion {}
val lambda2: Animal = new Cow  {}
lazy val p: lambda1.Food & lambda2.Food = ???
// for illustration, say we re-defined the following:
trait Food { type T }
trait Meat extends Food  { type T = Nothing }
trait Grass extends Food { type T = Any }
// statically
p.T /*has fully abstract bounds*/
// at runtime
lambda1.Food /*is*/ Meat /*&*/ lambda2.Food /*is*/ Grass
p /*has type*/ Meat & Grass
// lower bound is union of lower bounds
p.T /*has lower bound*/ Nothing | Any /*is*/ Any
// upper bound is intersection of upper bounds
p.T /*has upper bound*/ Nothing & Any /*is*/ Nothing
p.T /*has bad bounds at run-time!*/
```

Summary

- DOT is a core calculus for path-dependent types.
- Benefits for Scala
  - simpler type system
  - better type inference
  - (hopefully) easier reasoning
- Challenges in Meta-Theory
  - type preservation
    - run-time path equality
    - inlining branding
  - path-dependent types with "bad bounds"
    - subtyping transitivity
    - run-time "bad bounds" via intersection types

Thank You!