Towards Full Dependent Types in Scala

PurPL Retreat, 08/2020

Oliver Bračevac (oliver@bracevac.org)
Nitin John Raj
Guannan Wei
Tiark Rompf
Scala Unifies FP, OO & Module Systems

• Objects
• Lambdas
• Abstract Type Members
• Path-Dependent Types
• Subtyping
• Implicit
• Case Classes
• Pattern Matching
• Higher-kinded Types
• (oh my …)
Scala Unifies FP, OO & Module Systems

- Objects
- Lambdas
- **Abstract Type Members**
- Path-Dependent Types
- Subtyping
- Implicit
- Case Classes
- Pattern Matching
- Higher-kind Types
- (oh my …)

```scala
trait Tree {
  type Node
  val root: Node
  def add(node: Node): Tree
}

val appleTree : Tree
val orangeTree : Tree

appleTree.add(orangeTree.root)
// type error orangeTree.Node =/= appleTree.Node
```

[Rapoport '19]
Scala Unifies FP, OO & Module Systems
Key Issue: Mechanized Metatheory/Type Soundness

“Well-typed programs do not go wrong”
- Robin Milner
Scala Unifies FP, OO & Module Systems
Key Issue: Mechanized Metatheory/Type Soundness

In Scala 2.11,

“Well-typed programs do not go wrong”
— Robbin Milner

object unsound {
  trait LowerBound[T] { type M >: T }
  trait UpperBound[U] { type M <: U }
  def coerce[T, U](t : T): U = {
    def upcast(lb: LowerBound[T], t : T) : lb.M = t
    val bounded : LowerBound[T] with UpperBound[U] = null
    return upcast(bounded, t)
  }
}
def main(args: Array[String]): Unit = {
  val zero : String = coerce[Integer,String](0)
}
Dependent Object Types (DOT)
A Foundation for Scala, with Mechanized Soundness Proof

Path-dependent Types

Full Dependent Types

Martin-Löf Type Theory/Calculus of Constructions

DOT

Economy of Concepts

- class List[Elem] {} ~> class List { type Elem }
- List[String] ~> List { type Elem = String }
- List[T] forSome { type T }
- List ~> List

Extensions

- pDOT [Rapoport '19]
- gDOT [Giarrusso et al. '20]
- iDOT [Kabir et al. '20]

http://dotty.epfl.ch

Scala

[Dottor et al. '16, Rompf & Namin '16]
• DOT & Co. feature a restricted/simplified form of “types dependent on terms”

• Extensions with more features tend to be hard/span entire PhDs, e.g., pDot [Rappoport ’19]

• Metatheory of Dependent Types and Subtyping believed to be hard. Actively being researched [Fridlender & Pagano ’13, Abel et al. ’17, Yang & Oliveira ’17]

• Type Theory has been around for much longer (Frege & Russell 1900s, Martin-Löf 1970s, Barendregt 1990s)
Why Full Dependent Types?

According to Conor McBride:

Sometimes, your simplifying assumptions are exactly what's in your way. Avoiding dependent types is sometimes avoiding the truth.
Why Full Dependent Types?
Example: Type-Level Shapes in TensorFlow Scala

From Dynamic Checks

```scala
val inputs =
  tf.placeholder[Float](Shape(-1, 10)) // Tensor shape: [M, 10]

val predictions = tf.nameScope("Linear") {
  val weights =                        // Tensor shape: [5, 1]
    tf.variable[Float]("weights", Shape(5, 1), tf.ZerosInitializer)

  tf.matmul(inputs, weights)           // Runtime error!
}
```
Why Full Dependent Types?

Example: Type-Level Shapes in TensorFlow Scala

From Dynamic Checks

```scala
val inputs = tf.placeholder[Float](Shape(-1, 10)) // Tensor shape: [M, 10]
val predictions = tf.nameScope("Linear") {
  val weights = tf.variable[Float]("weights", Shape(5, 1), tf.ZerosInitializer)
  tf.matmul(inputs, weights) // Runtime error!
}
```

Certified Programming/Theorem Proving

```scala
// Matrices, operations, and laws
type M : Nat => Nat => Type

def mul[n <: Nat, m <: Nat, k <: Nat]: M[n,m] => M[m,k] => M[n,k] = ...

def trans[n <: Nat, m <: Nat]: M[n,m] => M[m,n] = ...

def mul_trans_law[n <: Nat, m <: Nat, k <: Nat]:
  (a: M[n,m]) => (b: M[n,k]) =>
  trans(mul(a,b)) == mul(trans(b),trans(a))
  = <proof of the theorem>
```

To Static Checks

```scala
val inputs = tf.placeholder[Float, Shape(-1, 10)]()
val predictions = tf.nameScope("Linear") {
  val weights = tf.variable[Float, Shape(5, 1)]("weights", tf.ZerosInitializer)
  tf.matmul(inputs, weights) // Type error: Shape mismatch!
}
```

• By virtue of the Curry-Howard Correspondence: Propositions as Types, Proofs as Programs

• Scala/Dotty does not permit such an interpretation of types
Why/When can we “Depend” on Dependent Types?

Logical Consistency: Type Soundness + Strong Normalization

**Well-typed programs do not go wrong**

**All well-typed programs reduce to a normal form**

(Coq proofs for a DOT subset: [Wang & Rompf '17])

Avoid Logical Paradoxes

E.g. abstract types require predicativity/cumulative universe hierarchy

$$\Sigma[ X \in \text{Set} \ l ](L \to X \times X \to U) : \text{Set}(l + 1)$$

Otherwise, Girard’s Paradox derivable in the type system [Hook & Howe ’86, Coquand ’86]

Can’t use problematic features in proofs

- General Recursion, Side Effects
  
  ```
  def inf(a: Nat): Nothing = inf(a)
  ```

- Otherwise could derive a proof of “falsity”
object Application {
  var global: Any = _
  def main(args: Array[String])(implicit c: CanIO) {
    println("ok") // ok, CanIO capability
    def rec(x) = rec(x) // ok, CanIO <: CanDiverge
    global = rec // error, c would escape via
      // closure of rec
  }
}

• Inspired by Zombie/Trellys [Casinghino et al. ’14], which distinguishes logical and general purpose programming via types.

• (Co)effect system via 2nd-class capabilities [Osvald et al. ’16] stratifies into logical and programmatic sublanguage
  • Syntax is shared between both sublanguages, i.e., just need to learn one language
  • You can hack in Scala the way you are used to
  • Optionally, you can do serious verification
  • The type system/capabilities will sort it out for you
    • Prevent logically inconsistent features where it matters
Going Full Dependent Types in MiniScala

Structural Recursion, via Subtyping and Intersections

// Assume Scott encoding of data
trait Nat {
  def switch[A](z: () => A)(s: Nat => A): A
}
def suc: Nat => Nat
def plus(a: Nat, b: Nat): Nat = {
  /* Recursive occurrences of plus typed as:
     plus: (Nat & a.Sub, Nat) -> Nat */
  a.switch { () => b } { n => suc(plus(n, b)) } /* ^
    ^
    Extracted value n has type Nat & a.Sub.
  Hence, method plus is well typed. */
}

// It’s safe to use plus in theorems
def plusComm(x: Nat, y: Nat): plus(x, y) == plus(y, x)

def inf(a: Nat): Nat = {
  /* Recursive occurrences of inf typed as:
     inf: (Nat & a.Sub) -> Nat */
  inf(a) /* ^
    Does not type-check, since
    a itself cannot be typed as a.Sub. */
}

// Rejected, encodes a false proposition
def wrong(x: Nat): inf(x) == x
Project Roadmap

This project can be separated into two parts: the theoretical foundations and a working demonstrator/compiler.

**Part A**
- Mechanized Metatheory
  - Support full term-dependency in types
  - Distinguish (non-) terminating fragments
  - Encoding Equality Proofs with Singletons

**Part B**
- Extended DOT
- MiniScala
- Type checker
- Compiler
- Runtime

**Part C**
- Core Logic
- TensorFlow
- FPGA
Conclusion

• Scala/DOT:
  • Metatheory is hard, due to subtyping and path-dependent types
• Working hypothesis: Full dependent types simplify the theory
• MiniScala
  • Scala with full dependent types
  • 2nd-class capabilities: pragmatic separation of verification and programming
  • Requires logically consistent core
    • Mechanization effort currently underway in Agda
• Backwards compatibility a non-goal
Conclusion

“You can have your \( \pi \) and prove with it, too!”
References