Towards Full Dependent Types in Scala

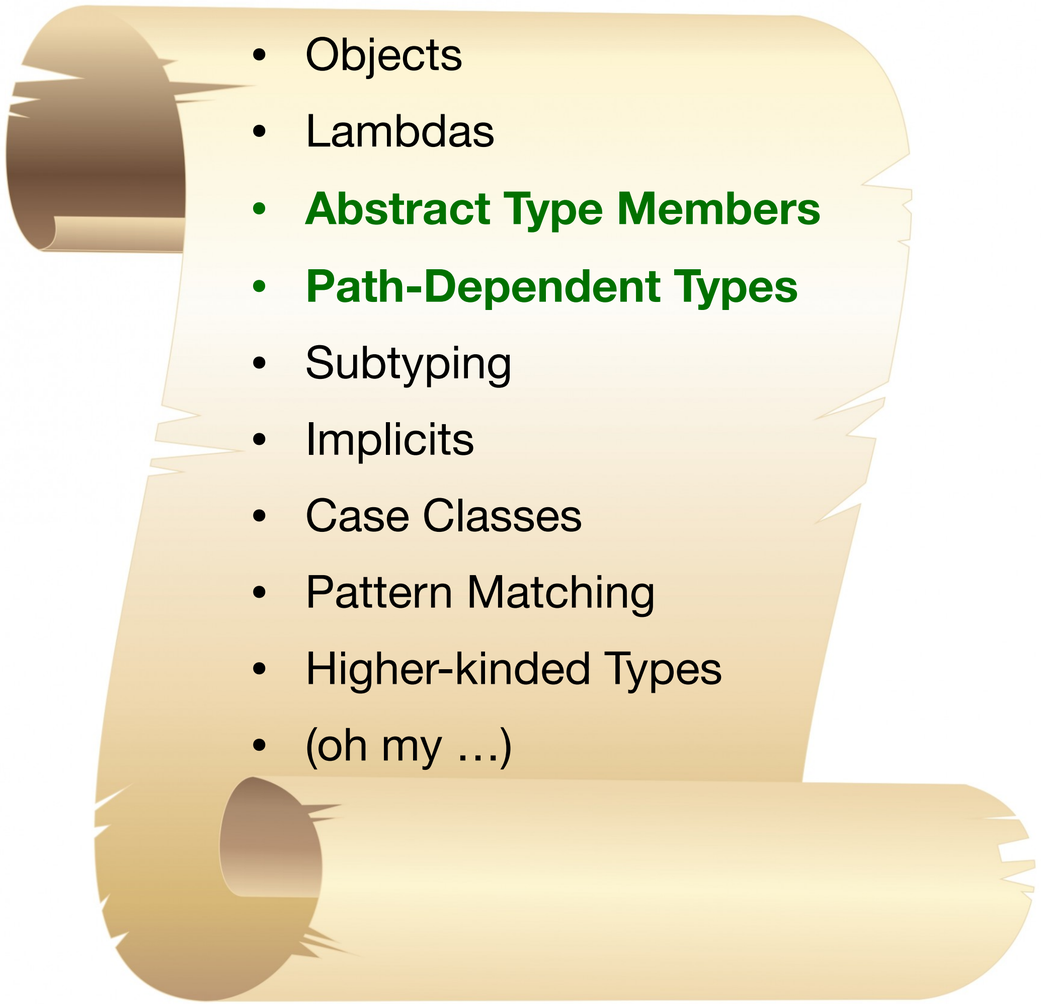
PurPL Retreat, 08/2020

Oliver Bračevac (oliver@bracevac.org)
Nitin John Raj
Guannan Wei
Tiark Rompf

PURDUE UNIVERSITY

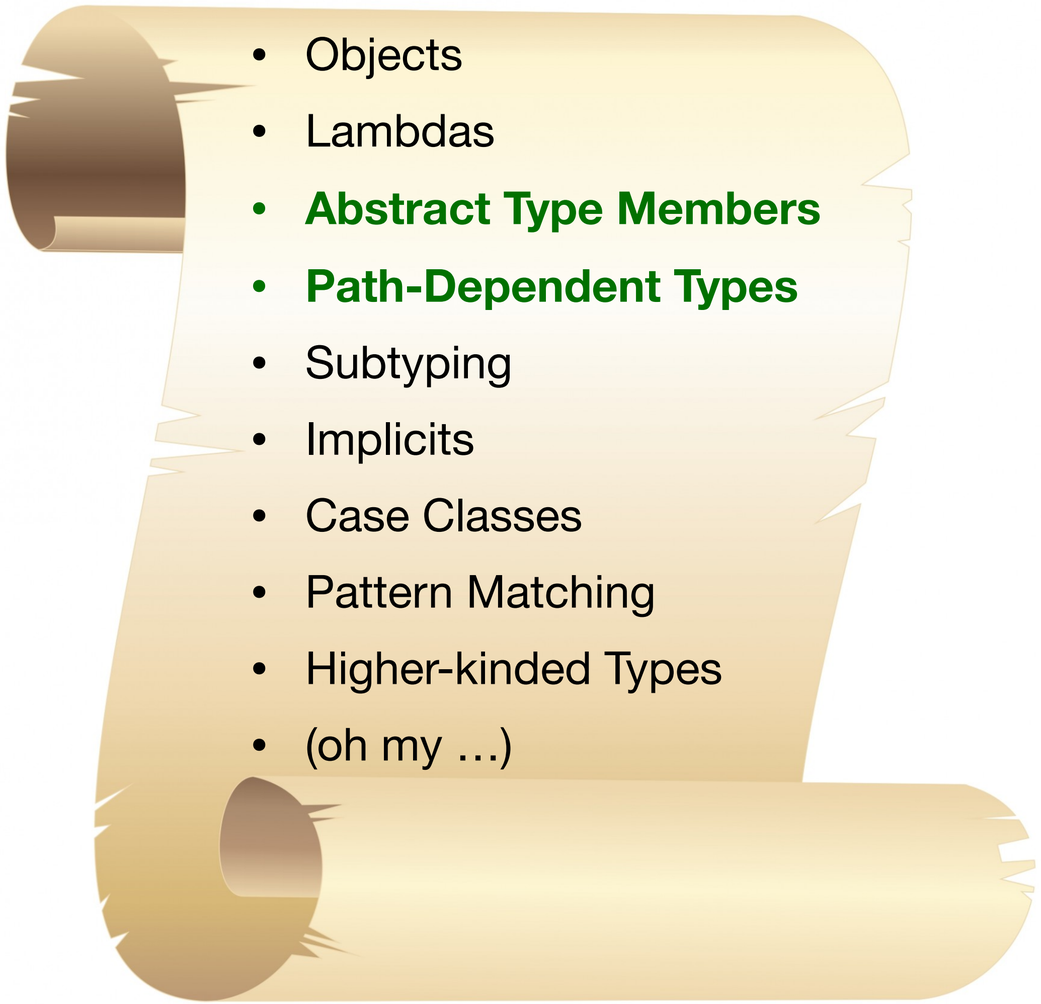
PUR PL

Scala Unifies FP, OO & Module Systems



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

Scala Unifies FP, OO & Module Systems



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

```
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

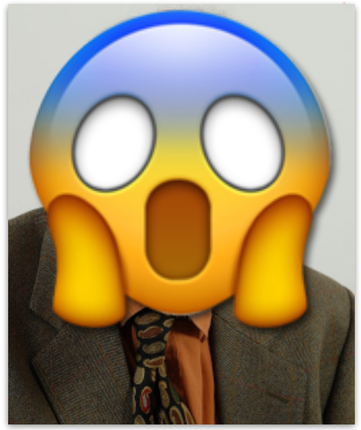
Scala Unifies FP, OO & Module Systems

Key Issue: Mechanized Metatheory/Type Soundness

In Scala 2.11,

> *"Well-typed programs do [redacted] go wrong"*
>
> *– [redacted]*

[Amin & Tate '16]



```
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)
}
```

Nada Amin & Ross Tate: Java and Scala's Type Systems are Unsound - The Existential Crisis of Null Pointers, OOPSLA'16

Dependent Object Types (DOT)

A Foundation for Scala, with Mechanized Soundness Proof

[Namin et al. ’16, Rompf & Namin ’16]

Path-dependent Types

Mind the gap!

Full Dependent Types

Martin-Löf Type Theory/Calculus of Constructions

Scala

DOT

Economy of Concepts

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

Extensions

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

Dotty

http://dotty.epfl.ch

MiniScala

Idris

DOT/Scala

Type Theory

Abstract Types

```
//type member
trait Cfg { type T :> L :< U }
```

```
//dependent pair (Σ-Type)
Cfg = Σ[ X ∈ Set l ](L → X × X → U)
```

Function Types

```
//dependent function
(c : Cfg) => P[c.T] => c.T
```

```
//dependent product (Π-Type)
Π Cfg (λc. Π (P (fst c)) (λ_. (fst c)))
```

Term Dependency

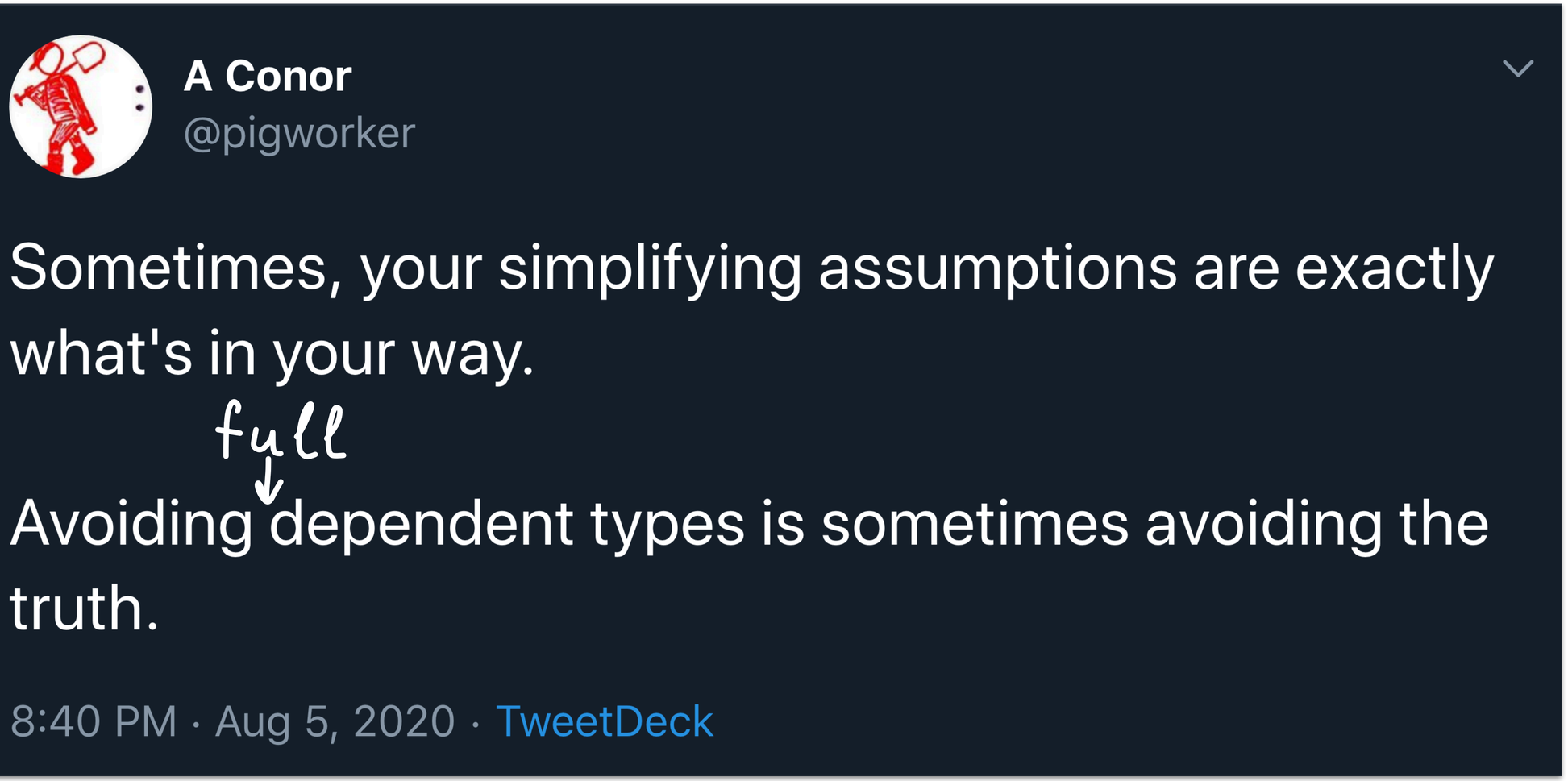
```
//only paths ::= identifier(.field*).field
x.T              //DOT  calculus
x.foo.bar.baz.T //pDOT extension
```

```
//unrestricted
h : Π Bool (λb. if (b & #t) then Int else String)
// h #t == Int, h #f == String
```

- 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:



Why Full Dependent Types?

Example: Type-Level Shapes in TensorFlow Scala

From Dynamic Checks

```
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

```
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!
}
```

To Static Checks

```
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!
}
```

Certified Programming/Theorem Proving

```
// 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>
```

- 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

```
Σ[ X ∈ Set l ](L → X × X → U) : 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”

Going Full Dependent Types in

Stratifying Total & Divergent Computations

```
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

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. */
}
```

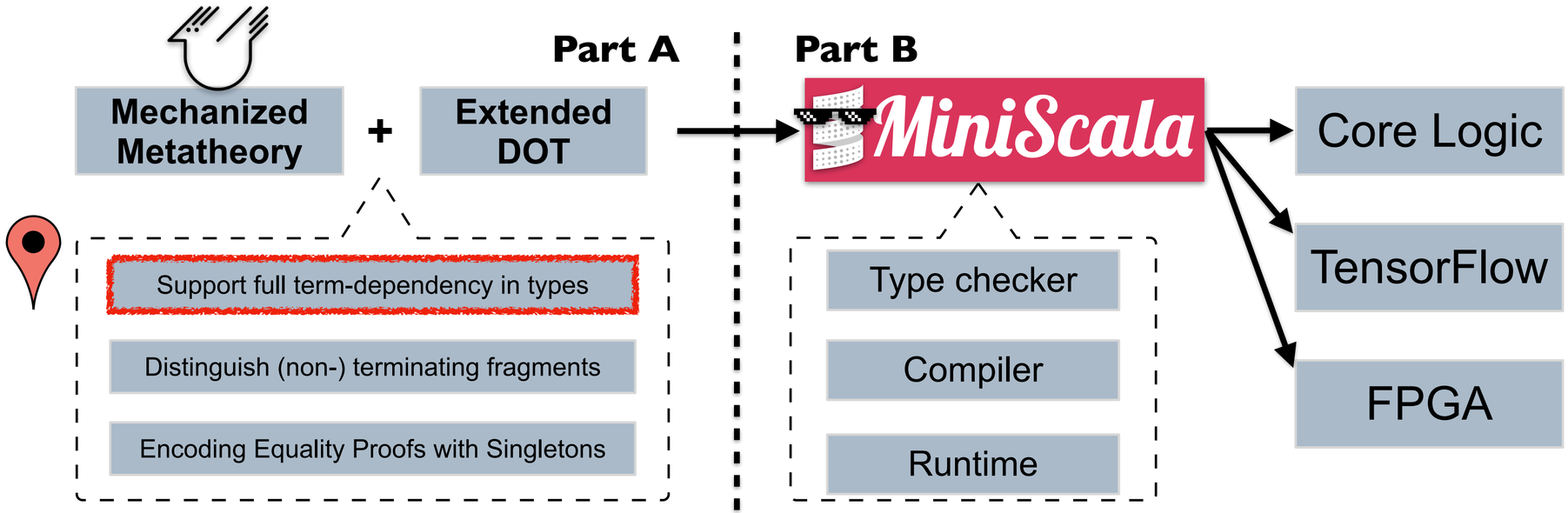
```
def inf(a: Nat): Nat = {
  /* Recursive occurrences of inf typed as:
     inf: (Nat & a.Sub) -> Nat */
  inf(a)
  /*  ^
     Does not type-check, since
     a itself cannot be typed as a.Sub. */
}
```

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

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

Project Roadmap

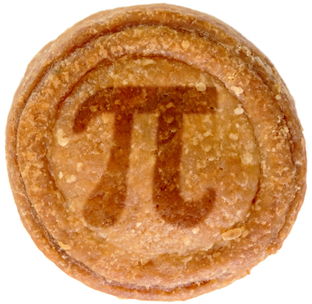
This project can be separated into two parts: the theoretical foundations and a working demonstrator/compiler.



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 Π and prove with it, too!"

References

[1] Ifaz Kabir, Li Yufeng, and Ondrej Lhoták. ιDOT: A DOT calculus with object initializaiton. *Proceedings of the ACM on Programming Languages (PACMPL)*, OOPSLA, 2020.

[2] Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. Normalization by evaluation for sized dependent types. *Proceedings of the ACM on Programming Languages (PACMPL)*, 1(ICFP), 2017.

[3] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in proof theory*. Bibliopolis, 1984.

[4] Daniel Fridlender and Miguel Pagano. A type-checking algorithm for martin-löf type theory with subtyping based on normalisation by evaluation. In *Proceedings of International Conference on Typed Lambda Calculi and Applications (TCLA)*, 2013.

[5] Martin Odersky, Guillaume Martres, and Dmitry Petrashko. Implementing higher-kinded types in Dotty. In *Proceedings of the Symposium on Scala*, 2016.

[6] Luca Cardelli. Typechecking dependent types and subtypes. In *Proceedings of Workshop on Foundations of Logic and Functional Programming*, 1986.

[7] Leo Osvald, Grégory M. Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. Gentrification gone too far? Affordable 2nd-class values for fun and (co-)effect. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016.

[8] Marianna Rapoport and Ondrej Lhoták. A path to DOT: Formalizing fully path-dependent types. *Proceedings of the ACM on Programming Languages (PACMPL)*, 3(OOPSLA), 2019.

[9] Jason Z. S. Hu and Ondrej Lhoták. Undecidability of *D<:* and its decidable fragments. *Proceedings of the ACM on Programming Languages (PACMPL)*, 4(POPL), 2020.

[10] Yanpeng Yang and Bruno C. d. S. Oliveira. Unifying typing and subtyping. *Proceedings of the ACM on Programming Languages (PACMPL)*, 1(OOPSLA), 2017.

[11] Thierry Coquand. An analysis of girard's paradox. In *Proceedings of Annual Symposium on Logic in Computer Science (LICS)*, 1986.

[12] James G. Hook and Douglas J. Howe. Impredicative strong existential equivalent to type:type. Technical report, Cornell University, 1986.

[13] Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, 2017.

[14] Paolo G. Giarusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. Scala step-by-step. *Proceedings of the ACM on Programming Languages (PACMPL)*, ICFP, 2020.

[15] Herman Geuvers. A short and flexible proof of strong normalization for the calculus of constructions. In *International Workshop on Types for Proofs and Programs (TYPES)*, 1994.

[16] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, 2014.

[17] Tiark Rompf and Nada Amin. Type soundness for dependent object types (DOT). In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016.

[18] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 249--272, 2016.

[19] Fei Wang and Tiark Rompf. Towards strong normalization for dependent object types (DOT). In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 2017.