An Introduction to Agda

Curtis Dunham
February 1, 2019

Agenda

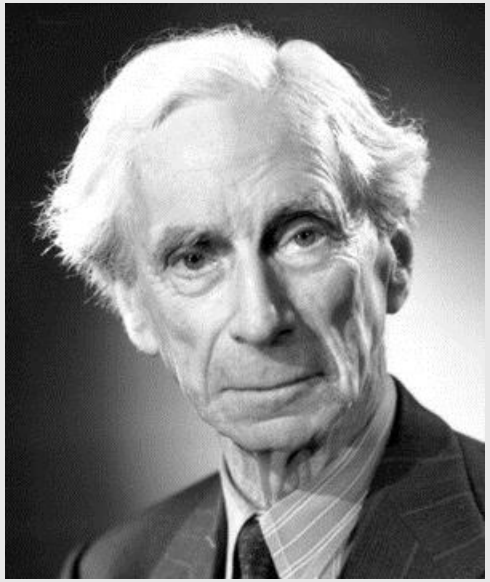
- History
- Agda
  - What it is
  - Why it’s interesting
  - Some basic definitions and proofs
- Demo
  - Emacs interaction
  - Typed holes
  - Short proofs

Intuitionistic Type Theory: The Forefathers

Brouwer

Intuitionism

Russell



Types

Intuitionism

- Briefly: mathematics <u>without</u>
  The Law of the Excluded Middle (LEM)

- LEM: All propositions are either true or false;
  $\forall$ P, P $\vee$ $\neg$P.
- Demands construction of witnesses:
  $\exists$x : P(x) can only be proven by *constructing* an
  object x such that P(x).

Russell's Types

- Russell's Paradox:
  "the set of all sets that do not contain themselves"
- Self-reference is problematic
- Types enforce a hierarchy in which self-reference is impossible

BHK Interpretation$_1$

- The Brouwer–Heyting–Kolmogorov Interpretation: interpretation of the logical operators in intuitionistic logic

- A ∧ B requires a proof of A and a proof of B
- A ∨ B requires a proof of A or a proof of B

…

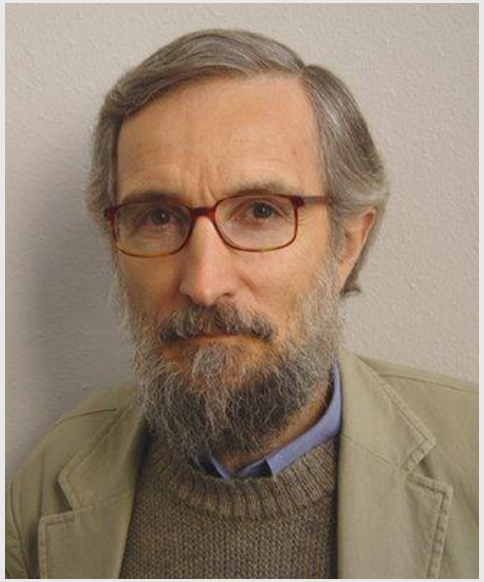
BHK Interpretation$_2$

- $A \to B$ requires a construction that transforms any proof of A into a proof of B
  - *i.e.* evidence $a : A$ transformed by function f such that $f(a) : B$
- $\bot$ (absurdity) has no proof
- $\neg A$ means $A \to \bot$

Curry-Howard Correspondence

- *and* $\Leftrightarrow$ *pairing*
- *or* $\Leftrightarrow$ *tagged union*
- *implication* $\Leftrightarrow$ *function application*
- *false/absurdity* $\Leftrightarrow$ *type with no members*

Intuitionistic Type Theory



- Per Martin-Löf:
  Martin-Löf Type Theory (MLTT) (1972)

Some key contributions towards Agda:

- Calculus of Constructions, Coquand
- Calculus of Inductive Constructions, Paulin-Mohring
- UTT, Luo
- Agda 2, Ulf Norell

What is Agda?

From the website [1] :

- A dependently-typed functional programming language
- A proof assistant

A product of Sweden – Chalmers, Gothenburg University

[1] http://wiki.portal.chalmers.se/agda/

Similar Systems

- Coq (CIC), Ocaml
- Matita (CIC), Ocaml
- Lean (CIC), C++
- Idris, Haskell

Agda and Haskell

Agda is…

- Written in Haskell
- Compiles to Haskell
- Liberally borrows Haskell syntax

Haskell influence brings:

- Fancy lambda calculus with pattern matching
- Significant indentation

Normal dependently typed features

- Types and terms share hierarchy of universes
  - Terms in types, types in terms – "full lambda cube"
  - Type functions
- "Propositions as Types", "Proofs are Programs"
  - A theorem is the type of its proofs
  - A proof "proves" the theorem by inhabiting/having the type
- Dependent product (Π), dependent sum (Σ)
  - Constructive "for all" and "there exists" quantifiers
- Type inference: arguments can often be inferred

Programming Language or Prover?

Recall: Agda is both

How?

- A dependently-typed functional programming language
- A proof assistant

In this logical system, **type checking = proof checking**

When using Agda as a prover, programs are not "compiled"; type checking is sufficient.

Distinct features

- Interactive editing of typed holes in Emacs
- Unicode

- Proof terms – deBruijn criterion ✓
  - Unlike tactic-oriented provers (*e.g.* Coq, HOL), in Agda the proof terms are written **directly**
  - A brief aside for the next few slides: This attribute receives undeserved negative prejudice

Proof Terms

- Back in 2010, Ben Delaware gave a Coq introduction to this audience
- He suggested that writing proof terms (as in Agda) is unpleasant

e.g. proof of associativity of list append:

```
Definition app_assoc :=
list_ind
 (fun a0 : list A => forall b c : list A, a0 ++ b ++ c = (a0 ++ b) ++ c)
 (fun b c : list A => refl_equal (b ++ c))
 (fun (a0 : A) (a1 : list A)
 (IHa : forall b c : list A, a1 ++ b ++ c = (a1 ++ b) ++ c)
 (b c : list A) =>
 let H :=
 eq_ind_r (fun l : list A => a0 :: (a1 ++ b) ++ c = a0 :: l)
 (refl_equal (a0 :: (a1 ++ b) ++ c)) (IHa b c) in
 eq_ind_r (fun l : list A => a0 :: a1 ++ b ++ c = l)
 (eq_ind_r (fun l : list A => a0 :: l = a0 :: l)
 (refl_equal (a0 :: (a1 ++ b) ++ c)) (IHa b c)) H) a
```

😬

Proof Tactics

- But that proofs by tactics was more pleasant

e.g. proof <u>script</u> for associativity of list append:

```
Lemma app_assoc : forall A (a b c : list A), a ++ (b ++ c) = (a ++ b) ++ c.
 induction a; simpl; intros.
 reflexivity.
 cut (a :: (a0 ++ b) ++ c = a :: (a0 ++ b ++ c)).
 intros; rewrite H; rewrite IHa; reflexivity.
 rewrite IHa; reflexivity.
Qed.
```

Counterpoint

- This distinction *is* true of Coq
  - Avoid writing Gallina proof terms directly
  - Ltac (tactic language) is dirty, but expedient

- But in Agda …
  - Writing proofs as Agda functions isn't so bad...
  - Typed holes provide equivalent interactivity!

Associativity of **append** in Agda

From the Agda standard library (agda-stdlib):

```
module _ {a} {A : Set a} where

  ++-assoc : Associative {A = List A} _≡_ _++_
  ++-assoc []       ys zs = refl
  ++-assoc (x ∷ xs) ys zs = cong (x ∷_) (++-assoc xs ys zs)
```

Some Definitional Backchaining...

```
-- Algebra/FunctionProperties.agda
module Algebra.FunctionProperties
  {a ℓ} {A : Set a} (_≈_ : Rel A ℓ) where
Associative : Op₂ A → Set _
Associative _·_ = ∀ x y z → ((x · y) · z) ≈ (x · (y · z))

-- Algebra/FunctionProperties/Core.agda
Op₂ : ∀ {ℓ} → Set ℓ → Set ℓ
Op₂ A = A → A → A
```

Definition of ++ (list concatenation)

```
-- Data/List/Base.agda
infixr 5 _++_

_++_ : ∀ {a} {A : Set a} → List A → List A → List A
[]       ++ ys = ys
(x ∷ xs) ++ ys = x ∷ (xs ++ ys)
```

Definition of ≡ (equality)

```
-- Agda/Builtin/Equality.agda
infix 4 _≡_
data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  instance refl : x ≡ x
```

Associativity of **append**, again$_1$

```
++-assoc : Associative {A = List A} _≡_ _++_
++-assoc []          ys zs = refl
++-assoc (x ∷ xs) ys zs = cong (x ∷_) (++-assoc xs ys zs)
```

After applying Associative, the type signature is roughly
λ (x y z : List _) → (x ++ y) ++ z ≡ x ++ (y ++ z)

Associativity of **append**, again$_2$

```
++-assoc : Associative {A = List A} _≡_ _++_
++-assoc []        ys zs = refl
++-assoc (x :: xs) ys zs = cong (x ::_) (++-assoc xs ys zs)
```

Proof proceeds by case analysis on the first argument.

Associativity of **append**, again$_3$

```
++-assoc : Associative {A = List A} _≡_ _++_
++-assoc []           ys zs = refl
++-assoc (x ∷ xs) ys zs = cong (x ∷_) (++-assoc xs ys zs)
```

Base case is trivial ('refl' means proof by reflexivity):
Recall that (by definition of ++), [] ++ ys ≡ ys. So
([] ++ y) ++ z ≡ [] ++ (y ++ z)
y ++ z ≡ y ++ z
refl (y ++ z)

Associativity of **append**, again$_4$

```
++-assoc : Associative {A = List A} _≡_ _++_
++-assoc []           ys zs = refl
++-assoc (x ∷ xs) ys zs = cong (x ∷_) (++-assoc xs ys zs)
```

When using proof by induction, *the proof is recursive*!

(xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)

x ∷ ((xs ++ ys) ++ zs) ≡ x ∷ (xs ++ (ys ++ zs))

■

Agenda

- Agda
  - What it is
  - Why it's interesting
  - Some basic definitions and proofs
- Demo
  - Emacs interaction
  - Typed holes
  - Short proofs

Agda Strengths

- Interactivity
- Brevity: Unicode, mixfix
- Proof terms
  - Powerful formalism, direct Curry-Howard
- Active community and developers

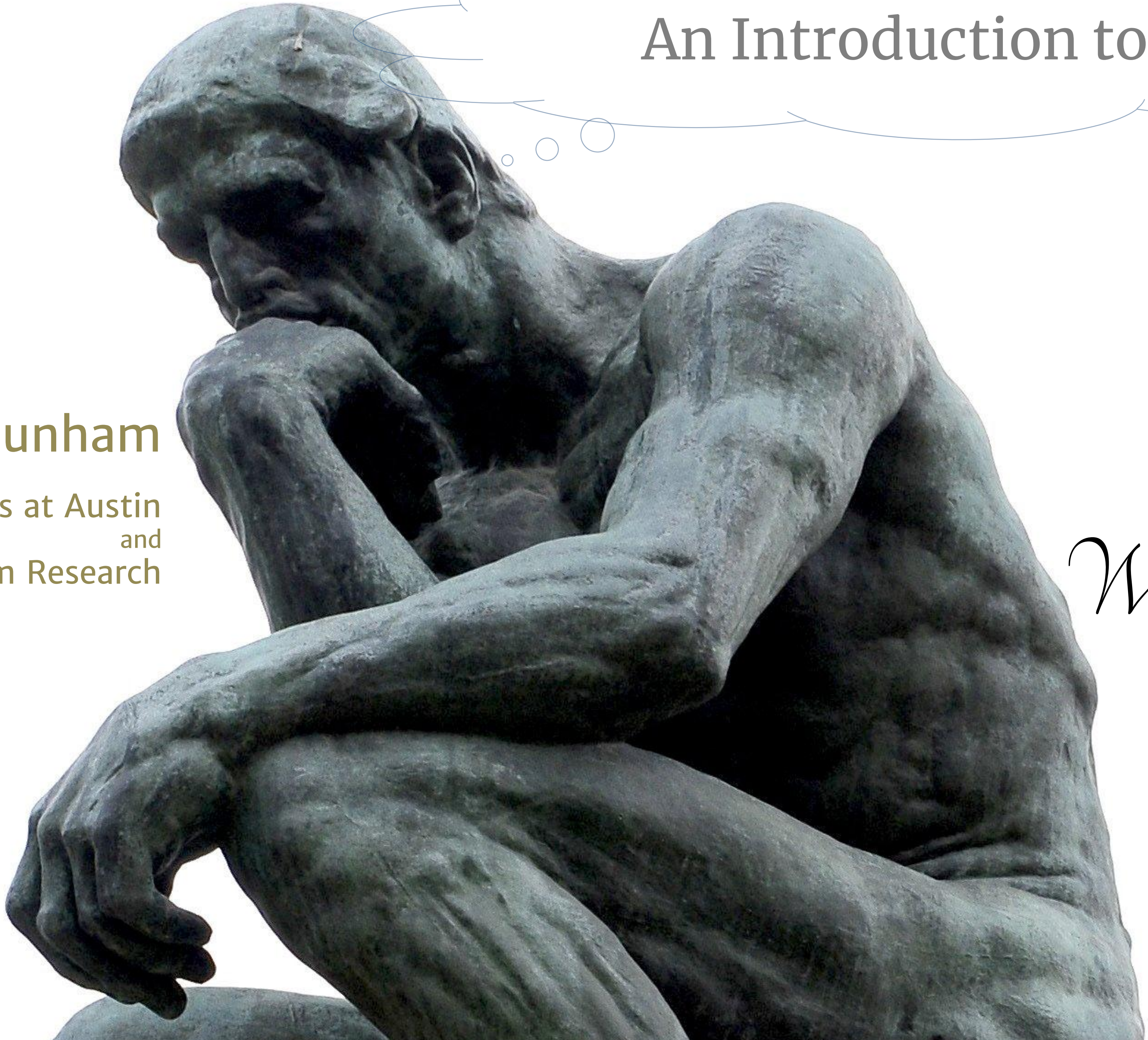
Agda Weaknesses

- Large body of background knowledge
- Poor error messages
- Proof automation functionality is minimal
  - Counterpoint: mature Reflection API allows self service
- Incomplete documentation
- Slow

“Agda-Curious”?

- Programming Language Foundations in Agda
  - https://plfa.github.io/
  - Port of Software Foundations (Coq) by Pierce, *et al.*

An Introduction to Agda



Thank you!

What questions do you have?

Curtis Dunham

University of Texas at Austin
and
Arm Research

Backup / Slide Graveyard