

A Fugue in Agda

Nikita Yurchenko

NURE

20.05.2017

- 1 Exposition
- 2 Interlude
- 3 Development
- 4 MLTT
- 5 Booleans
- 6 Natural Numbers
- 7 Lists
- 8 Vector
- 9 Coinduction
- 10 Finale
- 11 Questions

Exposition

LEM if false, and here is why:

Theorem (van Dalen, 1973)

There exist such irrational numbers a and b that the number a^b is rational.

Proof.

Consider a number $\sqrt{2}^{\sqrt{2}}$. If it is rational, then we can put $a = \sqrt{2}$, $b = \sqrt{2}$. In other case if it is irrational, then we can put $a = \sqrt{2}^{\sqrt{2}}$, $b = \sqrt{2}$. Therefore, in any case such a and b must exist (due to the law of excluded middle).

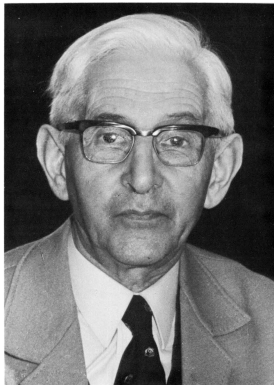


Three dogmas of intuitionism:

- ❶ No LEM (Therefore, proofs by contradiction are not valid).
- ❷ Replace actual infinity with potential realizability (Brouwer sequences).
- ❸ Proofs are mathematical objects. To prove something means to construct a proof of it.



Luitzen Brouwer



Arend Heyting



Andrey Kolmogorov

Brouwer-Heyting-Kolmogorov interpretation of logical constants

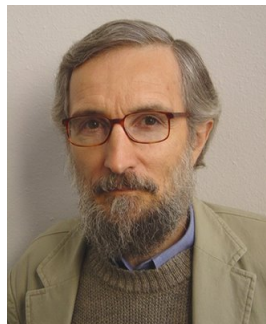
- A proof of $A \wedge B$ is a pair (a, b) where a is a proof of A , and b is a proof of B
- A proof of $A \vee B$ is a pair $(a, 1)$ where a is a proof of A , or a pair $(0, b)$ where b is a proof of B
- A proof of $A \longrightarrow B$ is a function that takes proofs of A as inputs and produces proofs of B as outputs
- A proof of $\exists x \in A : \phi(x)$ is a pair (a, b) where $a \in A$, b is a proof of $\phi(a)$
- A proof of $\forall x \in A : \phi(x)$ is a function $f : A \longrightarrow \Phi$, $a \mapsto \phi(a)$ that converts elements a of A into proofs $\phi(a)$
- Negation defined as $A \longrightarrow \perp$
- \perp is an absurd statement ($2 = 3$) that cannot be proven
- * Ex Falso Quodlibet: $\perp \longrightarrow A$, where A is arbitrary

Curry-Howard isomorphism (propositions-as-types)

Proof Theory	Type Theory
Proposition A Proof of $A \wedge B$ $A \vee B$ $A \supset B$ $\neg A$ (i.e. $A \rightarrow \perp$) true, false	Type $\Gamma \vdash a : A$ Product $A \times B$ Sum $A + B$ $A \rightarrow B$ $A \rightarrow \perp$ \top, \perp
$\forall x \in A. B(x)$ $\exists x \in A. B(x)$	$\prod_{x:a} B(x)$ $\sum_{x:a} B(x)$
Induction	Inductive type (e.g. \mathbb{N})
Pierce's law $((P \rightarrow Q) \rightarrow P) \rightarrow P$	Continuation
double-negation translation	Continuation-passing style

Interlude

- de Bruijn principle
- MLTT
- Girard's Paradox
- HoTT
- Coq
- Agda
- Idris
- NuPRL
- ...



Per Martin-Löf

Development

- Theoretical basics (Π , Σ , ac , \top , \perp , \equiv , \cong)
- Types & theorems (\mathbb{B} , \mathbb{N} , List , Vector , Stream)
- Monoid, Functor, Monad

MLTT

Π type

$\Pi : (A : \text{Set}) (B : A \rightarrow \text{Set}) \rightarrow \text{Set}$

$\Pi A B = (a : A) \rightarrow B\ a$

$_ \Rightarrow _ : (A\ B : \text{Set}) \rightarrow \text{Set}$

$A \Rightarrow B = \Pi A (\lambda _ \rightarrow B)$

Σ type

```
record  $\Sigma$  { $\ell_1$   $\ell_2$ } (A : Set  $\ell_1$ ) (P : A  $\rightarrow$  Set  $\ell_2$ ) : Set ( $\ell_1 \sqcup \ell_2$ ) where
  constructor  $\Sigma$ _,_
  field
    pr1 : A
    pr2 : P pr1
```

```
open  $\Sigma$  public
```

```
_  $\times$  _ : (A B : Set)  $\rightarrow$  Set
A  $\times$  B =  $\Sigma$  A ( $\lambda$  _  $\rightarrow$  B)
```

Identity type

```
data _≡_ {ℓ} {A : Set ℓ} (a : A) : A → Set ℓ where  
  refl : a ≡ a
```

```
sym : ∀ {ℓ} {A : Set ℓ} {a b : A} →  
      a ≡ b → b ≡ a  
sym refl = refl
```

```
trans : ∀ {ℓ} {A : Set ℓ} {a b c : A} →  
        a ≡ b → b ≡ c → a ≡ c  
trans refl refl = refl
```

```
cong : ∀ {ℓ} {A : Set ℓ} {a b : A} {B : Set} {f : A → B} →  
        a ≡ b → (f a) ≡ (f b)  
cong refl = refl
```



```
data  $\perp$  : Set where
```

```
record  $\top$  : Set where  
  constructor  $\top$  – cons
```

```
record  $\_ \vee \_$  (A : Set) (B : Set) : Set where  
  constructor  $\_ + \_$   
  field  
    in1 : A  
    in2 : B
```

Axiom of Choice

`data R {A B : Set} (a : A) (b : B) : Set where`

`ac : {A B : Set} → $\prod A (\lambda a \rightarrow \sum B (\lambda b \rightarrow R a b)) \rightarrow$
 $\sum (A \rightarrow B) (\lambda f \rightarrow (\prod A (\lambda a \rightarrow R a (f a))))$`

`ac g = $\sum (\lambda a \rightarrow pr_1 (g a)) , (\lambda a \rightarrow pr_2 (g a))$`

Booleans

Booleans

```
data  $\mathbb{B}$  : Set where  
  tt :  $\mathbb{B}$   
  ff :  $\mathbb{B}$ 
```

Operations on Booleans

infixl 5 $_ _$

$_ \wedge _ : \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}$

$tt \wedge tt = tt$

$a \wedge b = ff$

infixl 4 $_ _$

$_ \vee _ : \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}$

$tt \vee b = tt$

$ff \vee b = b$

Idempotence

$\wedge - \text{idemp} : \forall (x : \mathbb{B}) \rightarrow (x \wedge x) \equiv x$

$\wedge - \text{idemp } \text{tt} = \text{refl}$

$\wedge - \text{idemp } \text{ff} = \text{refl}$

Distributivity

$\wedge - \text{distr} - \vee : \forall \{x\ y\ z\} \rightarrow (x \wedge (y \vee z)) \equiv ((x \wedge y) \vee (x \wedge z))$

$\wedge - \text{distr} - \vee \ \{\text{tt}\} \ \{\text{tt}\} \ \{z\} = \text{refl}$

$\wedge - \text{distr} - \vee \ \{\text{tt}\} \ \{\text{ff}\} \ \{z\} = \text{refl}$

$\wedge - \text{distr} - \vee \ \{\text{ff}\} \ \{\text{tt}\} \ \{z\} = \text{refl}$

$\wedge - \text{distr} - \vee \ \{\text{ff}\} \ \{\text{ff}\} \ \{z\} = \text{refl}$

Absorption

$\wedge - \vee - \text{absorp} : \forall \{a\ b\} \rightarrow (a \wedge (a \vee b)) \equiv \text{tt} \rightarrow a \equiv \text{tt}$
 $\wedge - \vee - \text{absorp} \{\text{tt}\} p = \text{refl}$
 $\wedge - \vee - \text{absorp} \{\text{ff}\} ()$

Natural Numbers

Natural numbers

```
data  $\mathbb{N}$  : Set where
```

```
  zero :  $\mathbb{N}$ 
```

```
  suc :  $\mathbb{N} \rightarrow \mathbb{N}$ 
```

```
{-# BUILTIN NATURAL  $\mathbb{N}$  #-}
```

```
 $_ + _$  :  $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ 
```

```
zero + n = n
```

```
suc m + n = suc (m + n)
```

```
 $_ * _$  :  $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ 
```

```
zero * n = zero
```

```
suc a * b = b + (a * b)
```

0 is right zero

$+0 : \forall (x : \mathbb{N}) \rightarrow x + 0 \equiv x$

$+0 \text{ zero} = \text{refl}$

$+0 (\text{suc } x) \text{ rewrite } +0 x = \text{refl}$

+ commutes

+suc - lemma : $\forall (x\ y : \mathbb{N}) \rightarrow x + (\text{suc } y) \equiv \text{suc } (x + y)$

+suc - lemma zero y = refl

+suc - lemma (suc x) y rewrite +suc - lemma x y = refl

+comm : $\forall (x\ y : \mathbb{N}) \rightarrow x + y \equiv y + x$

+comm zero y rewrite +0 y = refl

+comm (suc x) y

rewrite -- sucx + y \equiv y + sucx

+comm x y | -- suc(y + x) \equiv y + sucx

+suc - lemma y x -- suc(y + x) \equiv suc(y + x)

= refl

* left distributive to +

`*ldistr+` : $\forall (x\ y\ z : \mathbb{N}) \rightarrow x * (y + z) \equiv x * y + x * z$

`*ldistr+` x zero z

rewrite $-- x * (zero + z) \equiv x * zero + x * z$

`*comm` x (zero + z) | $-- z * x \equiv x * 0 + z * x$

`*0` x $-- z * x \equiv 0 + z * x$

= refl

`*ldistr+` x (suc y) z

rewrite $-- x * (suc\ y + z) \equiv x * suc\ y + x * z$

`*suc - lemma` x (y + z) | $-- x + x * (y + z) \equiv x * suc\ y + x * z$

`*suc - lemma` x y | $-- x + x * (y + z) \equiv x + x * y + x * z$

`*ldistr+` x y z | $-- x + (x * y + x * z) \equiv x + x * y + x * z$

`+assoc` x (x * y) (x * z) $-- x + x * y + x * z \equiv x + x * y + x * z$

= refl

* Some auxiliary proofs are not shown

Lists

```
data List {ℓ} (A : Set ℓ) : Set ℓ where
```

```
  [] : List A
```

```
  _ :: _ : (x : A) (xs : List A) → List A
```

```
_ ++ _ : ∀ {ℓ} {A : Set ℓ} → List A → List A → List A
```

```
[] ++ ys = ys
```

```
(x :: xs) ++ ys = x :: (xs ++ ys)
```

Functions on Lists

$\text{map} : \forall \{\ell\} \{A\ B : \text{Set } \ell\} \rightarrow (A \rightarrow B) \rightarrow \text{List } A \rightarrow \text{List } B$

$\text{map } f [] = []$

$\text{map } f (x :: xs) = (f\ x) :: \text{map } f\ xs$

$\text{length} : \forall \{\ell\} \{A : \text{Set } \ell\} \rightarrow \text{List } A \rightarrow \mathbb{N}$

$\text{length } [] = \text{zero}$

$\text{length } (x :: xs) = \text{suc } (\text{length } xs)$

$\text{reverse} : \forall \{\ell\} \{A : \text{Set } \ell\} \rightarrow \text{List } A \rightarrow \text{List } A$

$\text{reverse } [] = []$

$\text{reverse } (x :: xs) = \text{reverse } xs ++ x :: []$

Second functor law

$\text{map} \circ : \forall \{ \ell \} \{ A \ B \ C : \text{Set } \ell \} (f : B \rightarrow C)$
 $(g : A \rightarrow B) (xs : \text{List } A) \rightarrow$
 $\text{map } (f \circ g) \ xs \equiv ((\text{map } f) \circ (\text{map } g)) \ xs$
 $\text{map} \circ f \ g \ [] = \text{refl}$
 $\text{map} \circ f \ g \ (x :: xs) \text{ rewrite } \text{map} \circ f \ g \ xs = \text{refl}$

Reverse preserves length

```
-- length - homo : length(xs ++ ys) ≡ length xs + length ys
reverse - preserves - length : ∀ {ℓ} {A : Set ℓ} → (xs : List A)
  → length (reverse xs) ≡ length xs
reverse - preserves - length [] = refl
reverse - preserves - length (x :: xs) rewrite
  reverse - preserves - length xs |
  length - homo (reverse xs) (x :: []) |
  reverse - preserves - length xs |
  +comm (length xs) 1
= refl
```

Vector

Internal vs External

- The way to reason about programs shown in previous examples is called **external** verification: we define non-dependent types and reason about them using \equiv .
- Another way to produce verified programs is to define a **dependent family of datatypes** along with their intrinsic properties, which make it impossible to produce incorrect programs. This side of verificationism is called **internal** verification.
- A family T of types indexed with **values** of another type I means that for each $i : I$ we have a member of this type family T_i .
- For instance, a type family indexed over \mathbb{B} would have precisely two members.
- A classic example of a dependent type is a dependent vector: a List-like container which can store a precise number of elements
- In recent versions of Haskell some "dependent types" can be simulated using type families and type-level programming

```
data Vec {a} (A : Set a) : ℕ → Set a where
  [] : Vec A zero
  _ :: _ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)

_ ++ _ : ∀ {a m n} {A : Set a} →
  Vec A m → Vec A n → Vec A (m + n)
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

Coinduction

Basic operations & Streams

postulate

$$\begin{aligned}\infty &: \forall \{a\} (A : \text{Set } a) \rightarrow \text{Set } a \\ \sharp_ &: \forall \{a\} \{A : \text{Set } a\} \rightarrow A \rightarrow \infty A \\ b &: \forall \{a\} \{A : \text{Set } a\} \rightarrow \infty A \rightarrow A\end{aligned}$$

infix 1000 $_$

$$\begin{aligned}\{-\# \text{ BUILTIN INFINITY } \infty \#-\} \\ \{-\# \text{ BUILTIN SHARP } \sharp_ \#-\} \\ \{-\# \text{ BUILTIN FLAT } b \#-\}\end{aligned}$$

data Stream (A : Set) : Set where
 $_ :: _ : A \rightarrow \infty (\text{Stream } A) \rightarrow \text{Stream } A$

Some recursive functions

$\text{take} : \{A : \text{Set}\} (n : \mathbb{N}) \rightarrow \text{Stream } A \rightarrow \text{Vec } A \ n$

$\text{take } \text{zero} \ xs = []$

$\text{take } (\text{suc } n) (x :: xs) = x : V : \text{take } n \ (b \ xs)$

$\text{drop} : \{A : \text{Set}\} \rightarrow \mathbb{N} \rightarrow \text{Stream } A \rightarrow \text{Stream } A$

$\text{drop } \text{zero} \ xs = xs$

$\text{drop } (\text{suc } n) (x :: xs) = \text{drop } n \ (b \ xs)$

Some corecursive functions

$\text{iterate} : \{A : \text{Set}\} \rightarrow (A \rightarrow A) \rightarrow A \rightarrow \text{Stream } A$
 $\text{iterate } f \ x = x :: \# \text{iterate } (f \circ f) \ x$

$\text{map} : \{A \ B : \text{Set}\} \rightarrow (A \rightarrow B) \rightarrow \text{Stream } A \rightarrow \text{Stream } B$
 $\text{map } f \ (x :: xs) = f \ x :: \# \text{map } f \ (b \ xs)$

$\text{zipWith} : \{A \ B \ C : \text{Set}\} \rightarrow (A \rightarrow B \rightarrow C) \rightarrow$
 $\text{Stream } A \rightarrow \text{Stream } B \rightarrow \text{Stream } C$
 $\text{zipWith } z \ (x :: xs) \ (y :: ys) =$
 $z \ x \ y :: \# \text{zipWith } z \ (b \ xs) \ (b \ ys)$

Equivalence on streams

```
data _≈_ {A : Set} : Stream A → Stream A → Set where
  _ :: _ : (x : A) {xs ys : ∞ (Stream A)} →
    ∞ (b xs ≈ b ys) → x :: xs ≈ x :: ys

infix 4 _ _
```

Proofs about Stream equivalence

$\text{srefl} : \{A : \text{Set}\} \{xs : \text{Stream } A\} \rightarrow$

$xs \approx xs$

$\text{srefl } \{A\} \{x :: xs\} = x :: \# \text{srefl}$

$\text{ssym} : \{A : \text{Set}\} \{xs\ ys : \text{Stream } A\} \rightarrow$

$xs \approx ys \rightarrow ys \approx xs$

$\text{ssym } (x :: xs) = x :: \# \text{ssym } (\text{b } xs)$

$\text{strans} : \{A : \text{Set}\} \{xs\ ys\ zs : \text{Stream } A\} \rightarrow$

$xs \approx ys \rightarrow ys \approx zs \rightarrow xs \approx zs$

$\text{strans } (x :: xs) (\text{.}x :: ys) =$

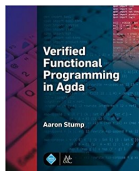
$x :: \# \text{strans } (\text{b } xs) (\text{b } ys)$

Finale

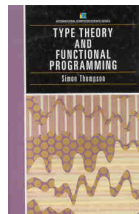
Π	<code>\Pi</code>	Σ	<code>\Sigma</code>	Σ	<code>\Sigma</code>
ℓ	<code>\ell</code>	λ	<code>\G1</code>	Σ	<code>\Sigma</code>
\rightarrow	<code>\r</code>	\langle	<code>\<</code>	Σ	<code>\Sigma</code>
∞	<code>\inf</code>	\flat	<code>\b</code>	Σ	<code>\Sigma</code>
\sharp	<code>\#</code>	\mathbb{B}	<code>\bb</code>	Σ	<code>\Sigma</code>
\mathbb{N}	<code>\bn</code>	\top	<code>\top</code>	Σ	<code>\Sigma</code>
\perp	<code>\bot</code>	\equiv	<code>\==</code>		<code>\glb</code>
\cong	<code>2 tilda</code>	$::$	<code>\::</code>		<code>\lub</code>
\oplus	<code>\oplus</code>	\otimes	<code>\otimes</code>	\mapsto	<code>\mapsto</code>
\circ	<code>\o</code>	ω	<code>\om</code>	Ω	<code>\Omega</code>

* To find out how to type arbitrary symbol,
use `M-x describe-char` in Agda mode.

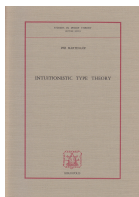
- A bit outdated: <http://oxij.org/note/BrutalDepTypes/>
- Wiki: <http://wiki.portal.chalmers.se/agda>
- Classic tutorial:
www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf
- <http://www.cse.chalmers.se/~peterd/papers/DependentTypesAtWork.pdf>
- Wonderful tutorial here
<http://people.inf.elte.hu/divip/AgdaTutorial/Index.html>
- Norell's PhD
- Agda Standard Library
<https://github.com/agda/agda-stdlib>
- NAL (NURE Agda Library)
<https://github.com/zelinskiy/NAL>



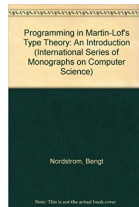
A. Stump



S. Thompson



P. Martin-Löf



B. Nordström

Questions