A Fugue in Agda

Nikita Yurchenko

NURE

20.05.2017

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.

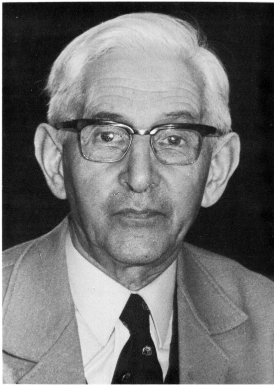
Consider a number $\sqrt{2}^{\sqrt{2}}$. If it is rational, then we can put $\mathrm{a} = \sqrt{2}$, $\mathrm{b} = \sqrt{2}$. In other case if it is irrational, then we can put $\mathrm{a} = \sqrt{2}^{\sqrt{2}}$, $\mathrm{b} = \sqrt{2}$. Therefore, in any case such a and b must exist (due to the law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

Constructive Mathematics

Three dogmas of intuitionism:

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

Luitzen Brouwer



Arend Heyting

Andrey Kolmogorov

BHK-interpretation

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 $A \longrightarrow \bot$
- $\bot$ is an absurd statement $(2 = 3)$ that cannot be proven
- * Ex Falso Quodlibet: $\bot \longrightarrow A$, where A is arbitrary

Curry-Howard isomorphism (propositions-as-types)

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

Interlude

Remarks

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

Per Martin-Löf

Development

Plan

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

MLTT

Π type

```
Π : (A : Set) (B : A → Set) → Set
Π A B = (a : A) → B a

_⇒_ : (A B : Set) → Set
A ⇒ B = Π A (λ _ → B)
```

Σ type

```
record Σ {ℓ₁ ℓ₂} (A : Set ℓ₁) (P : A → Set ℓ₂) : Set (ℓ₁ ⊔ ℓ₂) where
  constructor _,_
  field
    pr₁ : A
    pr₂ : P pr₁

open Σ public

_×_ : (A B : Set) → Set
A × B = Σ A (λ _ → 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
```

Logic

```
data ⊥ : Set where

record ⊤ : Set where
  constructor ⊤-cons

record _∨_ (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} → Π A (λ a → Σ B (λ b → R a b)) →
   Σ (A → B) (λ f → (Π A (λ a → R a (f a))))
ac g = Σ (λ a → pr₁ (g a)) , (λ a → pr₂ (g a))
```

Booleans

Booleans

```
data 𝔹 : Set where
  tt : 𝔹
  ff : 𝔹
```

Operations on Booleans

```
infixl 5 _ _

_∧_ : 𝔹 → 𝔹 → 𝔹
tt ∧ tt = tt
a  ∧ b  = ff

infixl 4 _ _

_∨_ : 𝔹 → 𝔹 → 𝔹
tt ∨ b = tt
ff ∨ b = b
```

Idempotence

```
∧-idemp : ∀ (x : 𝔹) → (x ∧ x) ≡ x
∧-idemp tt = refl
∧-idemp ff = refl
```

Distributivity

$$
\begin{aligned}
&\wedge-\mathrm{distr}-\vee : \forall\ \{x\ y\ z\} \rightarrow (x \wedge (y \vee z)) \equiv ((x \wedge y) \vee (x \wedge z))\\
&\wedge-\mathrm{distr}-\vee\ \{\mathrm{tt}\}\ \{\mathrm{tt}\}\ \{z\} = \mathrm{refl}\\
&\wedge-\mathrm{distr}-\vee\ \{\mathrm{tt}\}\ \{\mathrm{ff}\}\ \{z\} = \mathrm{refl}\\
&\wedge-\mathrm{distr}-\vee\ \{\mathrm{ff}\}\ \{\mathrm{tt}\}\ \{z\} = \mathrm{refl}\\
&\wedge-\mathrm{distr}-\vee\ \{\mathrm{ff}\}\ \{\mathrm{ff}\}\ \{z\} = \mathrm{refl}
\end{aligned}
$$

Absorption

```
∧-∨-absorp : ∀ {a b} → (a ∧ (a ∨ b)) ≡ tt → a ≡ tt
∧-∨-absorp {tt} p = refl
∧-∨-absorp {ff} ()
```

Natural Numbers

Natural numbers

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ

{-# BUILTIN NATURAL ℕ #-}

_+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = suc (m + n)

_*_ : ℕ → ℕ → ℕ
zero * n = zero
suc a * b = b + (a * b)
```

0 is right rero

```
+0 : ∀ (x : ℕ) → x + 0 ≡ x
+0 zero = refl
+0 (suc x) rewrite +0 x = refl
```

+ commutes

```
+suc-lemma : ∀ (x y : ℕ) → x + (suc y) ≡ suc (x + y)
+suc-lemma zero y = refl
+suc-lemma (suc x) y rewrite +suc-lemma x y = refl

+comm : ∀ (x y : ℕ) → x + y ≡ y + x
+comm zero y rewrite +0 y = refl
+comm (suc x) y
  rewrite          -- suc x + y ≡ y + suc x
  +comm x y |      -- suc(y + x) ≡ y + suc x
  +suc-lemma y x   -- suc(y + x) ≡ suc(y + x)
  = refl
```

* left distributive to +

```
*ldistr+ : ∀ (x y z : ℕ) → x * (y + z) ≡ x * y + x * z
*ldistr+ x zero z
  rewrite  -- x * (zero + z) ≡ x * zero + x * z
  *comm x (zero + z) | -- z * x ≡ x * 0 + z * x
  *0 x       -- z * x ≡ 0 + z * x
  = refl
*ldistr+ x (suc y) z
  rewrite                    -- x * (sucy + z) ≡ x * sucy + x * z
  *suc − lemma x (y + z) |   -- x + x * (y + z) ≡ x * sucy + x * z
  *suc − lemma x y |         -- x + x * (y + z) ≡ x + x * y + x * z
  *ldistr+ x y z |           -- x + (x * y + x * z) ≡ x + x * y + x * z
  +assoc x (x * y) (x * z) -- x + x * y + x * z ≡ x + x * y + x * z
  = refl
```

* Some auxiliary proofs are not shown

Lists

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

```
map : ∀ {ℓ} {A B : Set ℓ} → (A → B) → List A → List B
map f [] = []
map f (x :: xs) = (f x) :: map f xs

length : ∀ {ℓ} {A : Set ℓ} → List A → ℕ
length [] = zero
length (x :: xs) = suc (length xs)

reverse :  ∀ {ℓ} {A : Set ℓ} → List A → List A
reverse [] = []
reverse (x :: xs) = reverse xs ++ x :: []
```

Second functor law

```
map-∘ : ∀ {ℓ} {A B C : Set ℓ}(f : B → C)
  (g : A → B) (xs : List A) →
  map (f ∘ g) xs ≡ ((map f) ∘ (map g)) xs
map-∘ f g [] = refl
map-∘ f g (x :: xs) rewrite map-∘ f g xs = 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 familly of datatypes** along with their intrinsic properties, which make it impossible to produce incorrect programs. This side of verificationism is calles **internal** verification.
- An familly T of types indexed with **values** of another type I means that for each $\mathrm{i} : \mathrm{I}$ we have a member of this type familly T_{i}.
- For instance, a type familly 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

Vectors

```
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
  ∞  : ∀ {a} (A : Set a) → Set a
  ♯_ : ∀ {a} {A : Set a} → A → ∞ A
  ♭  : ∀ {a} {A : Set a} → ∞ A → A

infix 1000 _

{-# BUILTIN INFINITY ∞  #-}
{-# BUILTIN SHARP ♯_ #-}
{-# BUILTIN FLAT ♭      #-}

data Stream (A : Set) : Set where
  _∷_ : A → ∞ (Stream A) → Stream A
```

Some recursive functions

```
take : {A : Set} (n : ℕ) → Stream A → Vec A n
take zero  xs  = []
take (suc n) (x :: xs) = x : V : take n (♭ xs)

drop : {A : Set} → ℕ → Stream A → Stream A
drop zero xs = xs
drop (suc n) (x :: xs) = drop n (♭ xs)
```

Some corecursive functions

```
iterate : {A : Set} → (A → A) → A → Stream A
iterate f x = x :: ♯ iterate (f ∘ f) x

map : {A B : Set} → (A → B) → Stream A → Stream B
map f (x :: xs) = f x :: ♯ map f (♭ xs)

zipWith : {A B C : Set} → (A → B → C) →
  Stream A → Stream B → Stream C
zipWith z (x :: xs) (y :: ys) =
  z x y :: ♯ zipWith z (♭ xs) (♭ ys)
```

Equivalence on streams

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

infix 4 _ _
```

Proofs about Stream equivalence

```
srefl : {A : Set} {xs : Stream A} →
    xs ≈ xs
srefl {A} {x :: xs} = x :: ♯ srefl

ssym : {A : Set} {xs ys : Stream A} →
    xs ≈ ys → ys ≈ xs
ssym (x :: xs) = x :: ♯ ssym (♭ xs)

strans : {A : Set} {xs ys zs : Stream A} →
    xs ≈ ys → ys ≈ zs → xs ≈ zs
strans (x :: xs) (.x :: ys) =
    x :: ♯ strans (♭ xs) (♭ ys)
```

Monoids

```
record Monoid {ℓ} (M : Set ℓ) : Set ℓ where
  field
       : M
    _·_ : M → M → M
    · − assoc : (x y z : M) → ((x · y) · z) ≡ (x · (y · z))

mconcat : ∀ {ℓ} {M : Set ℓ} {{_ : Monoid M}} → List M → M
mconcat = foldr _·_
  where open Monoid {{...}}
```

Monoid & Instance arguments

```
instance
  listMonoid : ∀ {ℓ} {A : Set ℓ} → Monoid (List A)
  listMonoid = record {
      = [];
    _·_ = _++_ ;
    ·-assoc = ++-assoc }

t0 : List ℕ
t0 = mconcat ((1 ∷ 2 ∷ 3 ∷ []) ∷ (5 ∷ 6 ∷ 7 ∷ []) ∷ [])
```

Functors

Functor

```
record Functor {ℓ} (F : Set ℓ → Set ℓ) : Set (lsuc ℓ) where
  field
    fmap : ∀ {A B} → (A → B) → F A → F B
    law1 : ∀ {A} → (func : F A) → fmap id func ≡ id func
    law2 : ∀ {A B C} (g : B → C) (h : A → B) (f : F A) →
      fmap (g ∘ h) f ≡ ((fmap g) ∘ (fmap h)) f

fmap2 : {A B : Set } {F : Set → Set} {G : Set → Set}
  {{r1 : Functor G }} {{r2 : Functor F}} →
  (A → B) → G (F A) → G (F B)
fmap2 = fmap ∘ fmap where open Functor {{...}}
```

Monoids & Instance arguments

```
l1 : ∀ {ℓ} {A : Set ℓ} → (xs : List A) → map id xs ≡ xs
l1 [] = refl
l1 (x :: xs) rewrite l1 xs = refl

instance
  functorList : ∀ {ℓ} → Functor (List {ℓ})
  functorList = record {
    fmap = map;
    law1 = l1;
    law2 = map-∘ }
```

Monads

Monoids & Instance arguments

```
record Monad {ℓ₁ ℓ₂} (M : Set ℓ₁ → Set ℓ₂) : Set (lsuc ℓ₁ ⊔ ℓ₂) where
  field
    return : ∀ {A} → A → M A
    _>>=_ : ∀ {A B} → M A → (A → M B) → M B
    lidentity : ∀ {A B} → (a : A) (f : A → M B) →
      (return a) >>= f ≡ f a
    ridentity : ∀ {A} → (m : M A) → m >>= return ≡ m
    assoc : ∀ {A B C} → (m : M A) (f : A → M B) (g : B → M C) →
      (m >>= f) >>= g ≡ m >>= (λ x → f x >>= g)
```

```
instance
  MaybeMonad : ∀ {ℓ} → Monad (Maybe {ℓ})
  MaybeMonad = record {
    return = Just;
    _>>=_ = λ { Nothing _ → Nothing; (Just x) f → f x};
    lidentity = λ x f → refl;
    ridentity = λ { Nothing → refl; (Just _) → refl};
    assoc = λ { Nothing _ _ → refl; (Just _) _ _ → refl} }
```

Finale

Symbols

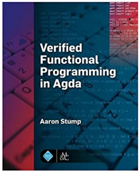
Π	`\Pi`	Σ	`\Sigma`	Σ	`\Sigma`
ℓ	`\ell`	λ	`\Gl`	Σ	`\Sigma`
$\rightarrow$	`\r`	$\langle$	`\<`	Σ	`\Sigma`
∞	`\inf`	$\flat$	`\b`	Σ	`\Sigma`
$\sharp$	`\#`	$\mathbb{B}$	`\bb`	Σ	`\Sigma`
$\mathbb{N}$	`\bn`	$\top$	`\top`	Σ	`\Sigma`
$\perp$	`\bot`	$\equiv$	`\==`		`\glb`
$\cong$	`2 tilda`	$::$	`\::`		`\lub`
$\oplus$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

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

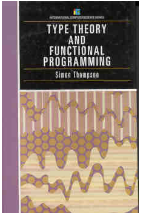
Resources

- 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
- Wonderfull 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

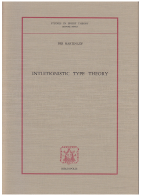
Books



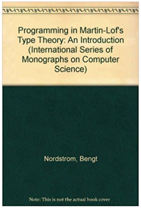
A. Stump



S. Thompson



P. Martin-Löf



B. Nordström

Questions