A brief introduction to Agda

December 5, 2022

Edoardo Marangoni
ecmm@anche.no

Agenda

- **Introduction**
  History
  What is it?

- **Agda basics**
  Induction

History

- V1.0 (approx. 1999) by C. Coquand
- V2.0 (approx. 2007) is a complete rewrite by U. Norell
- draws inspiration from previous works such as Alf, Alfa, Cayenne, Coq, ... , Automath

What is it?

- a **total** dependently typed programming language
- an extension of intuitionistic Martin-Löf type-theory
- a proof assistant (due to its dependent typing and Curry-Howard's correspondence)

The anatomy of a simple Agda definition:

```
open import Data.Bool
open import Data.String

-- Define a data type for greetings
data Greeting : Set where           -- Set is somewhat close to Coq's Type
 hello : Greeting   -- no name given
 -- this underscore is for prefix, infix, postfix, mixfix arguments to
 -- constructors
 --     ▼
 hello,_ : String → Greeting -- greet a person

greet : String → Greeting --              see?
--                                          ▼
greet x = if x == "" then hello else (hello, x)
```

Induction

Let's start from the basics: red-black trees. Ok, no, natural numbers:

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

In Agda, data structures need not be tagged with "inductive" or "coinductive" (albeit records do). Let's see a simple example of a function:

```
_+_ : Nat → Nat → Nat
_+_ = ? -- ← This is a goal. It allows interactive development.
```

Induction

Let’s start from the basics: red-black trees. Ok, no, natural numbers:

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

In Agda, data structures need not be tagged with “inductive” or “coinductive” (albeit records do). Let’s see a simple example of a function:

```
_+_ : Nat → Nat → Nat
x + x₁ = ? --  ← Using holes, we can _split_ data types; in this case,
           -- Agda gives us the two arguments x and x₁.
```

Induction

Let's start from the basics: red-black trees. Ok, no, natural numbers:

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

In Agda, data structures need not be tagged with "inductive" or "coinductive" (albeit records do). Let's see a simple example of a function:

```
_+_ : Nat → Nat → Nat
-- Let's split again on x:          (this is Agda's pattern-match)
zero + x₁ = {! !}
suc x + x₁ = {! !}
```

Induction

Let's start from the basics: red-black trees. Ok, no, natural numbers:

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

In Agda, data structures need not be tagged with "inductive" or "coinductive" (albeit records do). Let's see a simple example of a function:

```
_+_ : Nat → Nat → Nat
zero + x₁ = x₁
suc x + x₁ = suc (x + x₁)
```

Sweet, but how do I prove things?

Induction

Let’s prove the (left) identity of +.

```
+-id : ∀ (x : Nat) → ?0 -- <- Yes, we can use holes in type signatures as well!
+-id x = ?1
```

What should the type be?
This is the internal definition of (propositional, intensional, “definitional”) equality.

```
--         ╭ implicit argument with implicit type
--         │  ╭ implicit argument of type 'Set a' - notice the dependent typing
--         │  │           ╭ read this as 'for all x of type A'
--         ▼  ▼---------▼ ▼-----▼
data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  refl : x ≡ x

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

Two terms in agda are said to be definitionally equal when they both have the same normal form up to αη-conversion.

Induction

Let's prove the (left) identity of +.

```
+-id : ∀ (x : Nat) → zero + x ≡ x
+-id zero = ? -- ← must be a term of type '(zero + zero) ≡ zero'
+-id (suc x) = ? -- ← must be a term of type  '(zero + suc x) ≡ suc x'
```

Induction

Let's prove the (left) identity of +.

```
+-id : ∀ (x : Nat) → zero + x ≡ x
+-id zero = refl
+-id (suc x) = ? -- ← must be a term of type  '(zero + suc x) ≡ suc x'
```

Induction

Let's prove the (left) identity of +.

```
+-id : ∀ (x : Nat) → zero + x ≡ x
+-id zero = refl
+-id (suc x) = cong suc (+-id x)
```

Induction

Let’s prove the (left) identity of +.

```
+-id : ∀ (x : Nat) → zero + x ≡ x
+-id zero = refl
+-id (suc x) = cong suc (+-id x)
```

of course, Agda is powerful enough to let this proof be more concise:

```
+-id : ∀ (x : Nat) → zero + x ≡ x
+-id x = refl
```

Induction

Let’s prove something slightly more complicated: the associativity of +. We need to *reason* about equality. From Agda’s `stdlib`:

```
--                  ╭ notice that `y` is implicit!
--                  ▼
_≡⟨⟩_ : ∀ {A} (x {y} : A) → x ≡ y → x ≡ y
_ ≡⟨⟩ x≡y = x≡y

_≡⟨_⟩_ : ∀ (x {y z} : A)  → x ≡ y → y ≡ z → x ≡ z
x ≡⟨ x≡y ⟩ y≡z = trans x≡y y≡z
```

```
+-assoc : ∀ (x y z : Nat) → (x + y) + z ≡ x + (y + z)
+-assoc zero y z = refl
+-assoc (suc x) y z = {! !} -- Goal: ((suc x + y) + z) ≡ (suc x + (y + z))
```

Induction

Let's prove something slightly more complicated: the associativity of +. We need to *reason* about equality. From Agda's `stdlib`:

```
--                  ╭ notice that `y` is implicit!
--                  ▼
_≡⟨⟩_ : ∀ {A} (x {y} : A) → x ≡ y → x ≡ y
_ ≡⟨⟩ x≡y = x≡y

_≡⟨_⟩_ : ∀ (x {y z} : A)  → x ≡ y → y ≡ z → x ≡ z
x ≡⟨ x≡y ⟩ y≡z = trans x≡y y≡z
```

```
+-assoc : ∀ (x y z : Nat) → (x + y) + z ≡ x + (y + z)
+-assoc zero y z = refl
+-assoc (suc x) y z =
  ((suc x) + y) + z
  ≡⟨⟩
  (suc (x + y)) + z
  ≡⟨⟩
  suc ((x + y) + z)
  ≡⟨ ? ⟩ -- Goal :  suc ((x + y) + z) ≡ y'
   ? -- Goal : y' ≡ (suc x + (y + z))
```

Induction

Let's prove something slightly more complicated: the associativity of +. We need to *reason* about equality. From Agda's `stdlib`:

```
--                   ╭ notice that `y` is implicit!
--                   ▼
_≡⟨⟩_ : ∀ {A} (x {y} : A) → x ≡ y → x ≡ y
_ ≡⟨⟩ x≡y = x≡y

_≡⟨_⟩_ : ∀ (x {y z} : A)  → x ≡ y → y ≡ z → x ≡ z
x ≡⟨ x≡y ⟩ y≡z = trans x≡y y≡z
```

```
+-assoc : ∀ (x y z : Nat) → (x + y) + z ≡ x + (y + z)
+-assoc zero y z = refl
+-assoc (suc x) y z =
  ((suc x) + y) + z
  ≡⟨⟩
  (suc (x + y)) + z
  ≡⟨⟩
  suc ((x + y) + z)
  ≡⟨ cong suc (+-assoc x y z) ⟩
   ? -- Goal : suc (x + (y + z)) ≡ (suc x + (y + z))
```

Induction

Let's prove something slightly more complicated: the associativity of +. We need to *reason* about equality. From Agda's `stdlib`:

```
--                    ╭ notice that `y` is implicit!
--                    ▼
_≡⟨⟩_ : ∀ {A} (x {y} : A) → x ≡ y → x ≡ y
_ ≡⟨⟩ x≡y = x≡y

_≡⟨_⟩_ : ∀ (x {y z} : A)  → x ≡ y → y ≡ z → x ≡ z
x ≡⟨ x≡y ⟩ y≡z = trans x≡y y≡z
```

```
+-assoc : ∀ (x y z : Nat) → (x + y) + z ≡ x + (y + z)
+-assoc zero y z = refl
+-assoc (suc x) y z =
  ((suc x) + y) + z
  ≡⟨⟩
  (suc (x + y)) + z
  ≡⟨⟩
  suc ((x + y) + z)
  ≡⟨ cong suc (+-assoc x y z) ⟩
   suc (x + (y + z))
  ≡⟨⟩
    suc x + (y + z)
  ∎ -- ← reflexivity on equality: _∎ : ∀ (x : A) → x ≡ x
```