A brief introduction to Agda

or, how I learned to stop worrying and love the typechecker

February 17, 2023

Edoardo Marangoni

ecmm@anche.no

Agenda

- **Introduction**
  About Agda
  Syntax
  Interactive development
  Agda’s type system

- **Induction**
  Basic definitions
  Proving theorems

- **Co-induction**
  A brief recap of sized-types
  Basic definitions

About Agda

A bit of 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, Automath, Cayenne, Coq, ...

..and about Agda itself.

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

Syntax

- note that almost any character (including unicode codepoints and ",") except "(" and ")" is valid in identifiers! For example, `3::2::1::[]` is lexed as an identifier, and we must use spaces to make Agda parse it successfully. These are all valid identifiers:
  - `this+is*a-valid[identifier] : ℕ → ℕ`
  - `this,as→well : ℕ → ℕ`

- the character "_" has a special meaning in definitions, as it allows the definition of mixfix operators. For example: `if_then_else_` defines a function which can be used in the following ways:
  - `(if_then_else_) x y z`
  - `if x then y else z`
  - `(if x then_else_) y z`
  - `(if_then y else z) x`
  - `(if x then _ else z) y`
  - ...

Syntax

Let’s see a simple definition of a function:

```
not : Bool → Bool
not false = true
not true = false
```

We can see that pattern matching, in Agda, is similar to that of Haskell. We also can use implicit arguments (notice the dependent typing):

```
id : {A : Set} → A → A
id a = a
```

We can then use `id` both as `(id {Bool} true)` and `(id true)`.

Interactive development

this would be better with live coding

Being a proof assistant, interaction between the developer and Agda itself is really important. The main points of interaction are, in a way similar to Coq, *goals* and *holes*. A hole is syntactically represented by a `?` in the source, and it represents an unknown value:

```
f : ℕ → ℕ
f = ?
```

Agda will track the `?` as representing an unknown value which must have type `ℕ → ℕ`, and graphically show it in the editor as an unsolved goal. We can then ask Agda to automatically solve the goal, refine it or split the definition of the function to match on the possible values of an argument.

```
f : ℕ → ℕ
f = λ z → z
```

For example, when asked to solve the previous goal automatically, Agda may choose to fill it with the identity function `f = λ z → z` .

Agda's type system

- Agda is built upon a dependently typed system. This allows us to model among other things (e.g. being the backbone of its proof-assistant functions) safety conditions such as not accessing the index $n + 1$ of a vector of length n directly in the type system and check this condition at *compile time*.
- The fundamental (read: simple, small) type is `Set`. Not every type in Agda is a `Set`, however; to avoid paradoxes similar to Russel's, Agda uses *universe levels* and provides an infinite number of them, and we have that `Set` is not of type `Set`, instead it is `Set : Set₁`!
  In turn, it is `Set₁ : Set₂, ... : Setₙ`, where the subscript n is its **level**. This allows us to define, for example, `List` to be universe-polymorphic and, instead of being parametrized on types `A : Set`, on `A : Set n`, that is, we can have lists *of types*.

Basic definitions

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

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

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

In Agda, data structures need not be tagged with “inductive” or “coinductive” (albeit records do). Data definitions can be indexed...

```
data NatVec : ℕ → Set where
 ε    : Vec zero
 _::_ : ∀ {n : ℕ} → Nat → Vec n → Vec (suc n)
```

...and parametrized

```
data Vec (A : Set) : ℕ → Set where
 ε    : Vec A zero
 _::_ : ∀ {n : ℕ} → A → Vec A n → Vec A (suc n)
```

Basic definitions

We can also model relations as datatypes. The *mother* of all relations is **propositional equality**; what follows is the internal definition of (propositional, intensional, “definitional”) equality.

```
-- lib/Relation/Binary/PropositionalEquality/Core.agda
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
```

The use of relations and the power of Agda’s dependent type system allows us to write proofs just like function definitions:

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

Basic definitions

We can also model relations as datatypes. The *mother* of all relations is **propositional equality**; what follows is the internal definition of (propositional, intensional, “definitional”) equality.

```
-- lib/Relation/Binary/PropositionalEquality/Core.agda
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
```

The use of relations and the power of Agda’s dependent type system allows us to write proofs just like function definitions:

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

Basic definitions

We can also model relations as datatypes. The *mother* of all relations is **propositional equality**; what follows is the internal definition of (propositional, intensional, “definitional”) equality.

```
-- lib/Relation/Binary/PropositionalEquality/Core.agda
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
```

The use of relations and the power of Agda’s dependent type system allows us to write proofs just like function definitions:

```
+-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, as _+_ is defined recursively on the first element:

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

Equational reasoning

Of course, proofs can be much more complicated than that for `+-id` and might need *chains of equations* to express the various steps of a proof of a theorem. Ideally, every relation should have means to allow reasoning about it. Regarding the equality relation `≡`, the standard library gives us the following:

```
-- lib/Relation/Binary/PropositionalEquality/Core.agda
begin_ : ∀{x y : A} → x ≡ y → x ≡ y
begin_ x≡y = x≡y

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

step-≡ : ∀ (x {y z} : A) → y ≡ z → x ≡ y → x ≡ z
step-≡ _ y≡z x≡y = trans x≡y y≡z

syntax step-≡  x y≡z x≡y = x ≡⟨ x≡y ⟩ y≡z

_∎ : ∀ (x : A) → x ≡ x
_∎ _ = refl
```

(this is generalized with `Setoid` and `PartialSetoid`)

Equational reasoning

Chains of equations can often get difficult to read and follow, so let’s take it slow and prove the associativity of _+_:

```
+-assoc : ∀ (m n p : ℕ ) → (m + n) + p ≡ m + (n + p)
+-assoc m n p = ?
```

Equational reasoning

Chains of equations can often get difficult to read and follow, so let’s take it slow and prove the associativity of _+_:

```
+-assoc : ∀ (m n p : ℕ ) → (m + n) + p ≡ m + (n + p)
+-assoc zero n p = ? -- split on m
+-assoc (suc m) n p = ?
```

Equational reasoning

Chains of equations can often get difficult to read and follow, so let's take it slow and prove the associativity of _+_:

```
+-assoc : ∀ (m n p : ℕ ) → (m + n) + p ≡ m + (n + p)
+-assoc zero n p = refl
+-assoc (suc m) n p = ? -- we must show ((suc m) + n) + p ≡ (suc m) + (n + p)
```

On paper, we could prove this in a way similar to this: suppose

$$(m+n)+p \equiv m+(n+p) \tag{1}$$

then

$$\begin{aligned} & ((\mathrm{suc}\ m)+n)+p \\ \equiv\ & (\mathrm{suc}\ (m+n))+p \\ \equiv\ & \mathrm{suc}\ ((m+n)+p) \\ \textit{by } (1) \equiv\ & \mathrm{suc}\ (m+(n+p)) \\ \equiv\ & (\mathrm{suc}\ m)+(n+p) \end{aligned}$$

We can directly map this chain of reasoning in Agda!

Equational reasoning

Chains of equations can often get difficult to read and follow, so let's take it slow and prove the associativity of _+_:

```
+-assoc : ∀ (m n p : ℕ ) → (m + n) + p ≡ m + (n + p)
+-assoc zero n p = refl
+-assoc (suc m) n p =
 begin
   suc m + n + p
  ≡⟨⟩
   suc (m + n) + p
  ≡⟨⟩
   ? -- goal has type suc (m + n) + p ≡ suc m + (n + p)
```

Equational reasoning

Chains of equations can often get difficult to read and follow, so let's take it slow and prove the associativity of _+_:

```
+-assoc : ∀ (m n p : ℕ ) → (m + n) + p ≡ m + (n + p)
+-assoc zero n p = refl
+-assoc (suc m) n p =
 begin
   suc m + n + p
  ≡⟨⟩
   suc (m + n) + p
  ≡⟨⟩
   suc (m + n + p)
  ≡⟨⟩
   ? -- we now want to use the inductive hypothesis!
```

Equational reasoning

Chains of equations can often get difficult to read and follow, so let's take it slow and prove the associativity of _+_:

```
+-assoc : ∀ (m n p : ℕ ) → (m + n) + p ≡ m + (n + p)
+-assoc zero n p = refl
+-assoc (suc m) n p =
 begin
   suc m + n + p
  ≡⟨⟩
   suc (m + n) + p
  ≡⟨⟩
   suc (m + n + p)
  ≡⟨ cong suc (+-assoc m n p) ⟩
   suc (m + (n + p))
 ∎
```

Equational reasoning

Chains of equations can often get difficult to read and follow, so let’s take it slow and prove the associativity of _+_:

```
+-assoc : ∀ (m n p : ℕ ) → (m + n) + p ≡ m + (n + p)
+-assoc zero n p = refl
+-assoc (suc m) n p =
 begin
   suc m + n + p
  ≡⟨⟩
   suc (m + n) + p
  ≡⟨⟩
   suc (m + n + p)
  ≡⟨ cong suc (+-assoc m n p) ⟩
   suc (m + (n + p))
 ∎
```

Of course, this kind of proof technique can end up in something very difficult to follow...

```
-- lib/Relation/Binary/HeterogeneousEquality/Quotients/Examples.agda
+²-cong {a₁ , b₁} {c₁ , d₁} {a₂ , b₂} {c₂ , d₂} ab~cd₁ ab~cd₂ = begin
    (a₁ + c₁) + (b₂ + d₂) ≡⟨ ≡.cong (_+ (b₂ + d₂)) (+-comm a₁ c₁) ⟩
    (c₁ + a₁) + (b₂ + d₂) ≡⟨ +-assoc c₁ a₁ (b₂ + d₂) ⟩
    c₁ + (a₁ + (b₂ + d₂)) ≡⟨ ≡.cong (c₁ +_) (≡.sym (+-assoc a₁ b₂ d₂)) ⟩
  ...
```

A brief recap of sized-types

- Agda is a total language, therefore all function definitions (and proofs as well) must be demonstrably terminating
- Most termination checks are purely syntactical; this hinders the expressivity and ergonomics of the proof assistant
- Sizes are a semantic help for termination checkers: inductive and co-inductive types are indexed by a size; to check productivity and termination of recursive and co-recursive functions, one can check the behaviour of the sizes of the parameters

Basics of (sized) co-induction

Most of the infrastructure for sized co-induction in Agda is based on the following record:

```
-- lib/Size.agda
SizedSet : (ℓ : Level) → Set (suc ℓ)
SizedSet ℓ = Size → Set ℓ

-- lib/Codata/Sized/Thunk.agda
record Thunk {ℓ} (F : SizedSet ℓ) (i : Size) : Set ℓ where
  coinductive
  field force : {j : Size< i} → F j
```

Basics of (sized) co-induction

Most of the infrastructure for sized co-induction in Agda is based on the following record:

```
-- lib/Size.agda
SizedSet : (ℓ : Level) → Set (suc ℓ)
SizedSet ℓ = Size → Set ℓ

-- lib/Codata/Sized/Thunk.agda
record Thunk {ℓ} (F : SizedSet ℓ) (i : Size) : Set ℓ where
  coinductive
  field force : {j : Size< i} → F j
```

`Thunk`s embody the observational aspect of co-inductive definitions. In practice, this means that other co-inductive (or inductive-co-inductive) data types are defined in terms of `Thunk`:

```
data Stream (A : Set a) (i : Size) : Set a where
  _∷_ : A → Thunk (Stream A) i → Stream A i

head : {A : Set} {i : Size} → Stream A i → A
head (x ∷ x₁) = x

tail : {A : Set} {i : Size} {j : Size< i} → Stream A i → Stream A j
tail (x ∷ xs) = xs .force
```

Basics of (sized) co-induction

```
data Stream (A : Set a) (i : Size) : Set a where
  _::_ : A → Thunk (Stream A) i → Stream A i

repeat : {A : Set} {i : Size} → A → Stream A i
repeat a = a :: λ where .force → repeat a
```

Notice that `Stream` has a constructor! Therefore, stream definitions don't use (explicitly) co-patterns; We can see, in the definition of `repeat`, that the tail of the stream – which must be of type `Thunk (Stream {A}) i` – is a special form of lambda-abstraction which defines operations *on the abstraction itself*.

```
obs-repeat : {A : Set} → A → A
obs-repeat a = head (tail (repeat a))
```