Agda

Samuel Mimram

2025

École polytechnique

Agda

Agda is a programming language / proof assistant:

- one directly write programs of given type (no tactics),
- in order to help the user one gradually fills *holes*.

First proof

As a first proof, consider:

```
open import Prelude

-- The product is commutative
×-comm : {A B : Type} (A × B) → (B × A)
×-comm (a , b) = (b , a)
```

We then type `C-c C-l` in order to have Agda check this for us.

Agda syntax

We can import functions from other modules with

```
open import ModuleName
```

Comments are of the form

```
-- one line
```

or

```
{-
multiple
lines
-}
```

Agda syntax

For all functions, we declare the type and then the value:

```
f : {A : Type} → A → B
f x = ...
```

Named standard arguments are `(x : A)` and implicit arguments are `{x : A}`.

All functions are recursive.

We can use the notation `_` in order to have Agda guess part of the term.

Agda syntax

Agda allows for UTF-8 symbols which are mostly typed like in $\LaTeX$:

×	`\times`	⊤	`\top`	→	`\to`	Π	`\Pi`	λ	`\Gl`	ℕ	`\bN`
⊎	`\uplus`	¬	`\neg`	∃	`\ex`	Σ	`\Sigma`	≡	`\equiv`	∎	`\qed`

Agda is picky about **spaces**:

- `m + n` is an addition
- `m+n` is an identifier name

The reason is that we can define new infix operators, e.g.

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

Agda syntax

In practice it is almost impossible to write a whole proof directly.

Agda offers holes which are typed as `?`.

Agda shortcuts

We then have shortcuts to help us in proofs:

`C-c C-l`	typecheck and highlight the current file
`C-c C-,`	get information about the hole under the cursor
`C-c C-.`	same as above + the type of the proposed filler
`C-c C-`*space*	give a solution
`C-c C-c`	case analysis on a variable
`C-c C-r`	refine the hole
`C-c C-a`	automatic fill
middle click	go to the definition of the term

Inductive types

We can define inductive types

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

Constructors are injective: Agda “knows” that

- `zero` is always different from `suc n`
- `suc m` is the same as `suc n` iff `m` is the same as `n`

Inductive types

The identity type can also be defined as an inductive type:

```
data _≡_ {A : Type} (x : A) : (y : A) → Type where
  refl : x ≡ x
```

Local definitions

We can have local definitions with `where`:

```
quadruple : ℕ → ℕ
quadruple n = double (double n)
  where
  double : ℕ → ℕ
  double zero = zero
  double (suc n) = suc (suc (double n))
```

Universes

Agda features universes:

```
id : {ℓ : Level} {A : Type ℓ} → A → A
id a = a
```

We have supremum of levels:

```
arr : {ℓ ℓ' : Level} (A : Type ℓ) (B : Type ℓ') → Type (ℓ-max ℓ ℓ')
arr A B = A → B
```

Agda can generate implicit arguments:

```
private variable
  ℓ ℓ' : Level

arr : (A : Type ℓ) (B : Type ℓ') → Type (ℓ-max ℓ ℓ')
arr A B = A → B
```

Capturing implicit arguments

We can capture implicit arguments

```
id : {ℓ : Level} {A : Type ℓ} → A → A
id {ℓ} {A} a = a
```

or

```
id : {ℓ : Level} {A : Type ℓ} → A → A
id {A = A} a = a
```

Unnamed functions

It is possible to define unnamed functions:

```
id : {ℓ : Level} {A : Type ℓ} → A → A
id = λ x → x
```

We have a special syntax for pattern matching:

```
not : Bool → Bool
not = λ { true → false ; false → true }
```

(it is generally preferable to use `where`).

Modules

We can define modules with

```
module Int where
  int : Type
  int = ...

  add : int → int → int
  add = ...
```

Note the two spaces at the beginning of lines!

By default all files define modules.

Modules

Anonymous modules with parameters are sometimes useful:

```
module _ {ℓ ℓ' : Level} (A : Type ℓ) (B : Type ℓ') where
  f : A → B
  f = ...

  g : B → A
  g = ...
```

equational reasoning