Formalization in Agda
Part 1

Guillaume Brunerie

Course at the HoTT 2019 Summer School
Carnegie Mellon University, Pittsburgh

August 7, 2019

Introduction

- This course is based on Agda 2.6.0.1 that you hopefully managed to install. Installation instructions: `hott.github.io/HoTT-2019/agda-installation`
- Official documentation: `agda.readthedocs.io`
- These slides: `guillaumebrunerie.github.io/pdf/SummerSchool1.pdf`

Basic concepts

Agda is a dependently-typed programming language which can be used as a proof assistant. Or the other way around?

$$\begin{aligned} \text{statements} &\longleftrightarrow \text{types} \\ \text{proofs} &\longleftrightarrow \text{programs} \end{aligned}$$

```
name : type
name arguments = term
```

For instance

```
modus-ponens : ((P → Q) × P) → Q
modus-ponens (f , x) = f x
```

Here `f` is of type `P → Q` and `x` is of type `P`, therefore the application `f x` is of type `Q`.

Files and comments

- Agda file are text files ending with `.agda`
- One can type-check an Agda file from the command line:

  ```
  agda file.agda
  ```

  or directly from Emacs (see later).
- `-- Comment until the end of the line`
- `{- Multi-line comment (can be nested) -}`

Emacs mode

- Agda can be used interactively with Emacs and the agda-mode
- Input method for Unicode characters
- Key bindings for interactive theorem proving

Emacs

Notation for key bindings in Emacs:

- `C-x` means `Ctrl+x`
- `M-x` means `Alt+x` (or `Option+x` on a Mac)
- To execute something like `C-c C-l`, you do not need to release the `Ctrl` key between `c` and `l`.

Some commands that you may need

`C-x C-f`	New file / Open file
`C-x C-s`	Save file
`C-x b`	Switch to different file
`C-x C-c`	Quit

More commands

`C-a`	Beginning of the line
`C-e`	End of the line
`M-<`	Beginning of the file
`M->`	End of the file
`C-n`	Next line
`C-p`	Previous line
`C-v`	Forward one screen
`M-v`	Backward one screen
`C-s`	Search
`C-r`	Search backwards
`C-_`	Undo
`C-g`	Cancel command

`C-k`	Cut to the end of line
`M-w`	Copy
`C-y`	Paste
`M-%`	Replace
`C-x 0`	Remove buffer
`C-x 1`	Maximize buffer
`C-x 2`	Split buffer (h)
`C-x 3`	Split buffer (v)
`C-x (`	Start recording macro
`C-x )`	Stop recording macro
`C-x e`	Use recorded macro

Agda input method

The Agda input method allows you to type Unicode characters, with LaTeX-like notations. Some examples:

λ	\lambda,\Gl	×	\times,\x	∘	\o
→	\to, \->	◯	\bigcirc	π	\pi
≡	\equiv,\==	τ	\tau	$_4$	_4
≃	\simeq,\~-	⟨	\<	^{2}	\^2
Σ	\Sigma,\GS	⟩	\>	⊥	\bot
∀	\forall,\all	•	\.	ℕ	\bN
∧	\wedge,\and	≤	\le, \<=	ℤ	\bZ
∨	\vee,\or	¬	\neg	↑	\u

Use `M-x describe-char` to see how to input a particular character and `M-x describe-input-method` for the full list.

Interactive proofs

- Instead of writing a term in full and then typecheck it, you can write a term containing a hole and get information about it to fill it more easily later.
- A hole can be written as either

  ```
  ?
  ```

  or

  ```
  {!arbitrary content!}
  ```

Most important commands of the Agda mode

`C-c C-l`	Load the file
`C-c C-f`	Move to the next goal
`C-c C-t`	Show the type of the goal
`C-c C-d`	Show the type of the given term
`C-c C-e`	Show the context
`C-c C-,`	Show the type of the goal and the context
`C-c C-.`	Show the type of the goal, the term, and the context
`C-c C-SPC`	Fill the goal with its content
`C-c C-r`	Fill the goal with its content and enough arguments
`C-c C-c`	Case split
`C-c C-x M-;`	Comment the rest of the file
`C-c C-x C-r`	Kill Agda
`C-c C-n`	Normalize the given term

Lexical structure

- A *name* consists of a sequence of Unicode characters separated by white space or by special characters. The special characters are

  `(){}.;@"`

- In particular, `x+y` is only one symbol, whereas `x + y` is three successive symbols!
- There is a certain number of keywords that cannot be used as names, for instance

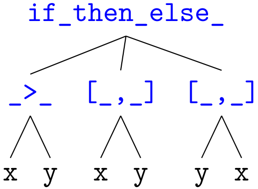
  `λ → = : postulate data record`

Mixfix operators

Mixfix operators (e.g. infix) are names containing underscores.
Each underscore represents the place of one argument.
If we have the symbols `if_then_else_`, `_>_` and `[_,_]`, then

```
if (x > y) then [ x , y ] else [ y , x ]
```

is parsed as



`if_then_else_`

`_>_` `[_,_]` `[_,_]`

x y x y y x

Function types

Non-dependent function types

```
f : A → B   -- or A -> B
f x = [...]  -- or f = λ x → [...]
```

Dependent function types

```
f : (x : A) → B x
f x = [...]
```

Currification can be used for functions with several arguments and parentheses are optional for application.

```
f : A → B → C → D
f a b c = [...]
```

Implicit arguments

An argument can be declared implicit:

```
f : {x : A} → B x
```

This means that you don't need to give it when applying f, and Agda will deduce it automatically (if possible). They can be given explicitely if needed, by name or by position.

```
u : B a
u = f   -- or  f {a}  or  f {x = a}
```

Inductive types and families

We can define inductive types using the `data` keyword.

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

Same for inductive families.

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

Pattern matching

Functions out of inductive types are defined with pattern matching.

```
f : ℕ → ℕ
f zero = zero
f (succ n) = succ (f n)

_•_ : {A : Set} {a b c : A} → a ≡ b → b ≡ c → a ≡ c
refl • refl = refl
```

--without-K

By default, pattern matching is too strong for the purposes of HoTT:

```
UIP : {A : Set} {x y : A} (p q : x ≡ y) → p ≡ q
UIP refl refl = refl
```

But there is an option to make Agda compatible with HoTT

```
{-# OPTIONS --without-K #-}
```

Record types

A record is a data structure containing several other pieces of data.

```
record Monoid : Set₁ where
  field
    X : Set
    m : X → X → X
    e : X
    unit-l : (x : X) → m x e ≡ x
    unit-r : (x : X) → m e x ≡ x
    assoc : (x y z : X) → m (m x y) z ≡ m x (m y z)
```

Copattern matching

Functions into records can be defined by copattern matching.

```
M : Monoid
X M = ℕ
m M = _+_
e M = zero
unit-l M = +-unit-l
unit-r M = +-unit-r
assoc M = +-assoc
```

Sigma types

We can define Σ-types with either records or data types.

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B fst
```

```
data Σ (A : Set) (B : A → Set) : Set where
  _,_ : (x : A) (y : B x) → Σ A B
```

They are quite similar, except that the first one has η and allows for copattern matching.

Universes

The type of “all” types is called `Set`. This is really types, not hSets!
But `Set` itself is of type $\mathtt{Set}_1$ instead.

There is an infinite hierarchy of universes: `Set`, $\mathtt{Set}_1$, $\mathtt{Set}_2$, ...

$$\mathtt{Set}_{\mathtt{n}} \;:\; \mathtt{Set}_{\mathtt{n}+1}$$

If $\mathtt{A} \;:\; \mathtt{Set}_{\mathtt{n}}$ and $\mathtt{B} \;:\; \mathtt{A} \to \mathtt{Set}_{\mathtt{m}}$, then

$$(\mathtt{x} : \mathtt{A}) \to \mathtt{B}\ \mathtt{x} \;:\; \mathtt{Set}_{\mathtt{n} \sqcup \mathtt{m}}$$

Universe polymorphism

- There is an (abstract) type of universe levels

  ```
  Level : Set
  lzero : Level
  lsuc  : Level → Level
  _⊔_    : Level → Level → Level
  ```

  You need to import the correct module

  ```
  open import Agda.Primitive
  ```

- One can quantify over universe levels

  ```
  id : {i : Level} {A : Set i} → (A → A)
  id x = x
  ```

Universe polymorphism

$$\frac{\Gamma \vdash \mathtt{i} : \mathtt{Level}}{\Gamma \vdash \mathtt{Set\ i} : \mathtt{Set\ (lsuc\ i)}}$$

$$\frac{\Gamma \vdash \mathtt{A} : \mathtt{Set\ i} \qquad \Gamma \vdash \mathtt{j} : \mathtt{Level} \qquad \Gamma, \mathtt{x} : \mathtt{A} \vdash \mathtt{B\ x} : \mathtt{Set\ j}}{\Gamma \vdash \mathtt{(x : A)} \to \mathtt{B\ x} : \mathtt{Set\ (i} \sqcup \mathtt{j)}}$$

$$\frac{\Gamma \vdash \mathtt{A}\ \ \mathtt{type} \qquad \Gamma, \mathtt{x} : \mathtt{A} \vdash \mathtt{B\ x}\ \ \mathtt{type}}{\Gamma \vdash \mathtt{(x : A)} \to \mathtt{B\ x}\ \ \mathtt{type}}$$

$$\frac{\Gamma \vdash \mathtt{A} : \mathtt{Set\ i}}{\Gamma \vdash \mathtt{A}\ \ \mathtt{type}}$$

Examples

(demo)