Overview of the Agda proof assistant

Guillaume Brunerie

November 28, 2012

Introduction

- Agda is a dependently typed functional programming language (can be compiled to Haskell, Epic or Javascript)
- Agda is a proof assistant based on Martin–Löf dependent type theory

Contents

1. The type theory behind Agda
2. The user interface

Contents

1. The type theory behind Agda

2. The user interface

The core theory

- Infinite hierarchy of universes à la Russel (no coercions)
  $\texttt{Set} = \texttt{Set}_0, \texttt{Set}_1, \texttt{Set}_2, \ldots, \texttt{Set}_\texttt{n}, \ldots$

$$\frac{}{\texttt{Set}_\texttt{n} \; : \; \texttt{Set}_{\texttt{n+1}}}$$

The core theory

- Infinite hierarchy of universes à la Russel (no coercions)
  $\mathtt{Set} = \mathtt{Set}_0, \mathtt{Set}_1, \mathtt{Set}_2, \ldots, \mathtt{Set}_n, \ldots$

$$\frac{}{\mathtt{Set}_n : \mathtt{Set}_{n+1}}$$

- Dependent product

$$\frac{\mathtt{A} : \mathtt{Set}_n \qquad \mathtt{x} : \mathtt{A} \vdash \mathtt{P}\ \mathtt{x} : \mathtt{Set}_m}{(\mathtt{x} : \mathtt{A}) \to \mathtt{P}\ \mathtt{x} : \mathtt{Set}_{max(n,m)}}$$

The core theory

- Infinite hierarchy of universes à la Russel (no coercions)
  $\texttt{Set} = \texttt{Set}_0, \texttt{Set}_1, \texttt{Set}_2, \ldots, \texttt{Set}_n, \ldots$

$$\frac{}{\texttt{Set}_n : \texttt{Set}_{n+1}}$$

- Dependent product

$$\frac{\texttt{A} : \texttt{Set}_n \qquad \texttt{x} : \texttt{A} \vdash \texttt{P x} : \texttt{Set}_m}{(\texttt{x} : \texttt{A}) \to \texttt{P x} : \texttt{Set}_{\max(n,m)}}$$

- Variables, application ($\texttt{u v}$), abstraction ($\lambda\ \texttt{x} \to \texttt{u}$)

The core theory

- $\beta\eta$-equality for functions, type directed conversion algorithm:

$$\frac{\mathtt{x} : \mathtt{A} \vdash \mathtt{u}\ \mathtt{x} \equiv \mathtt{v}\ \mathtt{x} : \mathtt{P}\ \mathtt{x}}{\vdash \mathtt{u} \equiv \mathtt{v} : (\mathtt{x} : \mathtt{A}) \to \mathtt{P}\ \mathtt{x}}$$

The core theory

- $\beta\eta$-equality for functions, type directed conversion algorithm:

$$\frac{\mathtt{x : A} \vdash \mathtt{u\ x} \equiv \mathtt{v\ x : P\ x}}{\vdash \mathtt{u} \equiv \mathtt{v : (x : A)} \to \mathtt{P\ x}}$$

- No cumulativity, `A : Set i` $\not\Rightarrow$ `A : Set (suc i)`
(a lifting operation can be defined using records)

The core theory

- $\beta\eta$-equality for functions, type directed conversion algorithm:

$$\frac{\mathtt{x\ :\ A} \vdash \mathtt{u\ x} \equiv \mathtt{v\ x\ :\ P\ x}}{\vdash \mathtt{u} \equiv \mathtt{v\ :\ (x\ :\ A)} \to \mathtt{P\ x}}$$

- No cumulativity, `A : Set i` $\not\Rightarrow$ `A : Set (suc i)` (a lifting operation can be defined using records)
- No impredicativity

Inductive types

- Strictly positive inductive types and inductive families

Inductive types

- Strictly positive inductive types and inductive families
- No dependent eliminator, no match, no fix, but definitions by pattern matching and termination checker

```
f : ℕ → A
f 0 = x
f (S n) = h n (f n)
```

Inductive types

- Strictly positive inductive types and inductive families
- No dependent eliminator, no match, no fix, but definitions by pattern matching and termination checker

```
f : ℕ → A
f O = x
f (S n) = h n (f n)
```

- Default pattern matching algorithm is too strong for HoTT (it implies K)

Inductive types

- Strictly positive inductive types and inductive families
- No dependent eliminator, no match, no fix, but definitions by pattern matching and termination checker

```
f : ℕ → A
f 0 = x
f (S n) = h n (f n)
```

- Default pattern matching algorithm is too strong for HoTT (it implies K), use the option `--without-K`

```
$ agda --help
[...]
     --without-K      disable the K rule (maybe)
[...]
```

Records

- Enjoy definitional η-equality (gives η for `unit` and Σ)

Records

- Enjoy definitional η-equality (gives η for `unit` and Σ)

$$\frac{\vdash \pi_1\ \mathtt{u} \equiv \pi_1\ \mathtt{v} : \mathtt{A} \qquad \vdash \pi_2\ \mathtt{u} \equiv \pi_2\ \mathtt{v} : \mathtt{P}\ (\pi_1\ \mathtt{u})}{\vdash \mathtt{u} \equiv \mathtt{v} : \Sigma\ \mathtt{A}\ \mathtt{P}}$$

Records

- Enjoy definitional η-equality (gives η for `unit` and Σ)

$$\frac{\vdash \ \pi_1 \ \mathtt{u} \equiv \pi_1 \ \mathtt{v} : \mathtt{A} \qquad \vdash \ \pi_2 \ \mathtt{u} \equiv \pi_2 \ \mathtt{v} : \mathtt{P} \ (\pi_1 \ \mathtt{u})}{\vdash \ \mathtt{u} \equiv \mathtt{v} : \Sigma \ \mathtt{A} \ \mathtt{P}}$$

- Gives lifting operation for universes

```
record lift {i} (A : Set i) : Set (suc i) where
  constructor ↑
  field ↓ : A
```

Universe polymorphism

- There is an abstract type of universe levels

```
Level : Set₀   (= Set zero)
zero : Level
suc : Level → Level
max : Level → Level → Level
```

Universe polymorphism

- There is an abstract type of universe levels

```
Level : Set₀   (= Set zero)
zero : Level
suc : Level → Level
max : Level → Level → Level
```

- One can quantify over universe levels

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

Universe polymorphism

- There is an abstract type of universe levels

```
Level : Set₀   (= Set zero)
zero : Level
suc : Level → Level
max : Level → Level → Level
```

- One can quantify over universe levels

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

- Not all types belong to some universe

Universe polymorphism

$$\frac{\texttt{i : Level}}{\texttt{Set i : Set (suc i)}}$$

$$\frac{\texttt{A : Set i} \qquad \texttt{x : A} \vdash \texttt{P x : Set j} \qquad \texttt{j : Level}}{\texttt{(x : A)} \to \texttt{P x : Set (max i j)}}$$

(j does not depend on x)

$$\frac{\texttt{A type} \qquad \texttt{x : A} \vdash \texttt{P x type}}{\texttt{(x : A)} \to \texttt{P x type}}$$

$$\frac{\texttt{A : Set i}}{\texttt{A type}}$$

Instance arguments

- Agda's version of type classes

Instance arguments

- Agda's version of type classes
- Arguments declared as instance arguments are inferred from the context if there is exactly one matching value

Instance arguments

- Agda's version of type classes
- Arguments declared as instance arguments are inferred from the context if there is exactly one matching value

```
postulate
  group-structure : Set → Set
  _●_ : {G : Set} {{G-str : group-structure G}}
        → G → G → G
  H : Set
  H-str : group-structure H

function : H → H → H → H
function x y z = (x ● y) ● (z ● (x ● y))
```

Instance arguments

- They can have other uses

```
axiom-of-choice : Set
axiom-of-choice = [...]

lemma : {{ac : axiom-of-choice}} → [...]
lemma {{ac}} = [...] ac [...]

theorem : {{ac : axiom-of-choice}} → [...]
theorem = [...] lemma [...]
```

Instance arguments

- They can have other uses

```
axiom-of-choice : Set
axiom-of-choice = [...]

lemma : {{ac : axiom-of-choice}} → [...]
lemma {{ac}} = [...] ac [...]

theorem : {{ac : axiom-of-choice}} → [...]
theorem = [...] lemma [...]
```

- Main drawback: instance arguments are non-recursive (design choice)

Abstract blocks

In the following situation

```
abstract
  f : [...]
  f = [...]

  g : [...]
  g = [...]

h : [...]
h = [...]
```

g can access the definition of f but h cannot access the definition of either f or g.

Other features

- Induction-recursion
- Irrelevant arguments
- Coinduction
- Reflection
- Positivity checking can be disabled
- Termination checking can be disabled
- Coverage checking can be disabled
- Type in type can be enabled

Contents

1 The type theory behind Agda

2 The user interface

Emacs mode

- The only supported way to use Agda interactively is emacs with the agda-mode

Emacs mode

- The only supported way to use Agda interactively is emacs with the agda-mode
- Input method for Unicode characters

Emacs mode

- The only supported way to use Agda interactively is emacs with the agda-mode
- Input method for Unicode characters
- Key bindings for interactive edition of proofs

Input method

λ	\lambda, \Gl	$\times$	\times, \x	$\circ$	\o
$\rightarrow$	\to, \->	$\bigcirc$	\bigcirc	π	\pi
$\equiv$	\equiv, \==	τ	\tau	$_4$	_4
$\simeq$	\simeq, \~-	$\langle$	\<	^{2}	\^2
Σ	\Sigma, \GS	$\rangle$	\>	$\bot$	\bot
$\forall$	\forall, \all	$\bullet$	\.	$\mathbb{N}$	\bn
$\wedge$	\wedge, \and	$\leq$	\le, \<=	$\mathbb{Z}$	\bz
$\vee$	\vee, \or	$\neg$	\neg	$\uparrow$	\u

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

Interactive proofs

- There are no tactics, you write λ-terms directly
- You can write λ-terms with holes, which will be filled later

Common commands

`C-c C-l`	Load the file
`C-c C-SPC`	Fill the current goal
`C-c C-a`	Try to automatically fill the current goal
`C-c C-c`	Case split
`C-c C-r`	Introduction of λ or record constructors
`C-c C-t`	Gives the type of the goal
`C-c C-d`	Gives the type of the given term
`C-c C-.`	Gives the type of the goal and of the given term
`C-u C-c C-t`	Same without normalizing
`C-u C-c C-d`	Same without normalizing
`C-u C-c C-.`	Same without normalizing

Examples

(examples)