On Agda

2009.3.12
JAIST/AIST WS
CVS/AIST Yoshiki Kinoshita, Yoriyuki Yamagata

Agenda

- On Agda
- Agda as a programming language
- Agda as a proof system
- Further information.

Agenda

- On Agda
- Agda as a programming language
- Agda as a proof system
- Further information.

History

Chalmers University of Technology (Göteborg, SW) from 1990's

Catarina Coquand, Lennart Augustsson, Thierry Coquand, Makoto Takeyama, Marcin Benke, Thomas Hallgren

Cf. ALF, Alfa etc、theorem prover based on constructive type theory

Joint development by CVS/AIST from 2003

plug-in, application to industry

Cf. Agda Implementers Meeting (AIM): twice a year@ Göteborg /Senri。2008.11 9th@Sendai

Agda2 from 2006

Language changed→old Agda is called Agda1

Ulf Norell, Catarina Coquand, Makoto Takeyama, Nils Anders Danielsson, Andreas Abel, Karl Mehltretter, Marcin Benke

Principle of Agda

- Martin-Löf's constructive type theory
- **Curry-Howard correspondence**
  term : type ⟷ proof : proposition
  - As a dependent-type functional programming language
    Term as a program; Agda is a compiler
  - As a proof system
    Term as a proof; Agda is a theorem prover

Feature of Agda

- Language
  - Dependent type
  - Recursively defined type
  - Unicode
- System
  - Always shows a proof term
    as opposed to script based Coq, Isabelle
  - Compiler to Haskell
  - Meta variables
    Place holder which can be Instantiated by concrete term later.

Agenda

- On Agda
- Agda as a programming language
- Agda as a proof system
- Further information.

Agda Language

- Type declaration
  - data … (recursive type)
  - record … (record)
- Function declaration (definitional equality)
- Postulate
- Module declaration → record type
- fixity

Agda Language

- Type declaration
  - data … (recursive type)
  - record … (record)
- Function declaration (definitional equality)
- Postulate
- Module declaration → record type
- fixity

“data” declaration

- Sole predefined type : Set
  Set: type of all type
- Introducing type
  - data …
  - record …
- “data” declaration
  Enumerate all constructors
- E.g.: Bool type
  true and false are
  constructors of Bool type

```
data Bool : Set where
  true  : Bool
  false : Bool
```

Recursively defined data type

- Recursive type
  Constructor can take arguments of its type
- E.g.: Nat
  - Type of natural numbers
  - Constructors are zero and succ
  - zero is a constant
  - succ is a unary constructor
- Terms of Nat type:
  zero
  succ zero
  succ (succ zero)

```
data Nat : Set where
  zero : Nat
  succ : Nat -> Nat
```


Lexical Convention

- Indentation and new line have meanings
- Ex.2. need indentation.
- Ex.3. need new line
- Ex.4 is correct.
  Semicolon can replace new line
- identifier can contain any unicode
  To separate identifiers, we need spaces
  e.g. Nat: Set are two identifiers.
- mixfix notation
  - See Ex.5
    ′ is declared to be a postfix operator

1.
```
data Nat : Set where
  zero : Nat
  succ : Nat -> Nat
```

2.
```
data Nat : Set where
zero : Nat
succ : Nat -> Nat
```

3.
```
data Nat : Set where
  zero : Nat succ : Nat ->
Nat
```

4.
```
data Nat : Set where
 zero : Nat ; succ : Nat ->
Nat
```

5.
```
data Nat : Set where
  O : Nat
  _′ : Nat -> Nat
```

Parameterized data type

- Finite list of values of type A
  - `[]` nil;
  - `::` cons
  - `_::_` `::` is an infix operator

```
data List (A : Set) : Set
where
  []   : List A
  _::_ : A ->
         List A ->
         List A
```


Dependant type

- Array : type dependent to value.
  Fixed length list of values
- `Array A n` signifies type of array with elements of type A and length n
- `*` is a only value of `Array A 0`
- `cons n a x` adds x to a, an array of length n
- Type of 3rd argument of cons `Array A n` depend on n ∴We name the 1st arg. n for later reference

```
1.  data Array (A : Set) :
         Nat -> Set where
      * : Array A O
      cons : (n : Nat) ->
             A ->
             Array A n ->
             Array A (n ' )
2.  * : {A : Set} ->
        Array A 0
    cons : {A : Set} ->
           (n : Nat) ->
           A ->
           Array A n ->
           Array A (n ' )
```

implicit argument

- 1st arg. Of cons n can be derived of the type of 3rd argument
- write {n : Nat} instead of (n : Nat)
  Then, we can omit n
- Defining as 1. cons can be used without n (cons a x)
- In this case, we can introduce infix^ and defining as 2.
  e.g., a^x : adding x to a

```
1.  data Array (A : Set) :
         Nat -> Set where
      *: Array A O
      cons : {n : Nat} ->
             A ->
             Array A n ->
             Array A (n ’ )

2.  data Array (A : Set) :
         Nat -> Set where
      * : Array A O
      _^_ : {n : Nat} ->
            A ->
            Array A n ->
            Array A (n ’ )
```


Agda Language

- Type declaration
  - data … (recursive type)
  - record … (record)
- Function declaration (definitional equality)
- Postulate
- Module declaration → record type
- fixity

Record type

- A way of combining several values of different types
- E.g.: values of (1.) are pair consisting Bool and Nat.
  f1 and f2 are field names
- Field names can be used to extract values (selector)
  Declaration 1. contains two function definition of 2.
- If b is type Bool and n is type Nat, 3. is a value of type BN.

```
1. record BN : Set
   where
    field
     f1 : Bool
     f2 : Nat

2. BN.f1 : BN -> Bool
   BN.f2 : BN -> Nat

3. record {
    f1 = b ;
    f2 = n
   }
```


Dependant Record Type

- Type of a field can depend on the value of previous fields.
- E.g. Combining Boolean array and its length (1.)
  - Type of array field depends on length
  - Type of the selector becomes like 2.

```
1.  record VR : Set where
     field
      length : Nat
      array  :
       Array Bool length

2.  VR.length : VR -> Nat
    VR.array :
     (r : VR) ->
     Array Bool (VR.length r)
```

Parameterized Record Type

- Record type can take a type parameter.
- E.g. we can define as 1.
  - Types of selectors: 2.
- Type parameter X become an implicit argument of selectors.

```
1.  record PVR (X : Set) : Set
      where
        field
          length : Nat
          array : Array X length

2.  PVR.length :
      {X : Set} -> PVR X -> Nat
    PVR.array :
      {X : Set} ->
      (r : PVR X) ->
      Array X (PVR.length r)
```


Agda Language

- Type declaration
  - data … (recursive type)
  - record … (record)
- Function declaration (definitional equality)
- Postulate
- Module declaration → record type
- fixity

Function Declaration

- Function abstraction becomes 1. it corresponds
  (λ x : Nat) x
- In Function declarations, types and values are specified separately. (2.,3.)
- Can specify values by pattern matching (2.,3.)
- Can use mixfix in function declaration as “data” declarlation (2.,3.)

```
1. ¥ (x : Nat) -> x

2. _+_ :
    Nat -> Nat -> Nat
   n + O = n
   n + (m ’ ) = (n + m)’

3. _==_ :
     Nat -> Nat -> Bool
   O == O      = true
   zero == _ ’  = false
   m ’  ==  O   = false
   m ’  == n ’  = m == n
```


Function Declaration

- By making type parameters as implicit parameters, you can define if-then-else(1.) and Array map(2.)

```
1. if_then_else_fi :
     {X : Set} ->
     Bool -> X -> X -> X
   if true then a else _ fi = a
   if false then _ else b fi = b

2. ArrayMap :
     {X Y : Set} -> {n : Nat} ->
     (X -> Y) -> Array X n ->
   Array Y n
   ArrayMap f * = *
   ArrayMap f (x ^ xs) = (f x) ^
   (ArrayMap f xs)
```

Agenda

- On Agda
- Agda as a programming language
- Agda as a proof system
- Further information.

Embedding Formal System

- Shallow embedding
  Agda treats formulae as atoms
  Substitution etc. → Agda's substitution
- Deep embedding
  Agda can look "inside" of formulae

Shallow Embedding of First Order Predicate Logic

- Proposition as a set of proofs
  - Based on Intuitionistic logic
  - Excluded Middle is introduced by postulate
- Shallow Embedding
  - Proposition : Set
  - Proofs are elements of propositions
- Logical connectives are introduced by 4 rules
  - 1. Formation rules
  - 2. Introduction rules
  - 3. Elimination rules
  - 4. Conversions

We show Agda's construction for1.-3.
4. defines equivalence between proofs (normalization).

⊥ (absurdity)

- Absurdity is a proposition with no proof i.e. Empty Set
- Form: 1.
  - No constructor for ⊥
  - ∴no proof for proposition⊥
- No Introduction rule
- Elim: 2. ⊥Elim
  - For any prop. P, Elim{P} takes a proof of ⊥ and returns a proof of P.
  - `() matches when there is no constructor in the type`

```
1. data ⊥ : Set
   where

2. ⊥Elim :
     {P : Prop} ->
     ⊥ ->
     P
   ⊥Elim ()
```

∧ (conjunction)

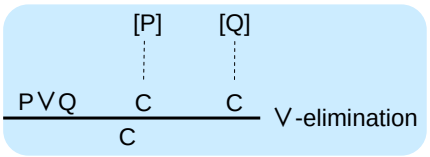
- Form: 1.
  If P and Q are propositions, P ∧ Q is a proposition.
  ∧ is defined as a parameterized record
- Intro: 2.
  Function ∧Intro takes P's proof and Q's proof, returns P ∧ Q's proof.
- Elim:
  Elimination rules for conjunctions are constructed by selectors.

```
1. record _∧_ (P Q : Set) : Set
     where
       field
         Elim1 : P  ;  Elim2 : Q

2. ∧Intro :
     {P Q : Set} -> P -> Q -> P ∧ Q
   ∧Intro a b =
     record { Elim1 = a ; Elim2 = b }

3. ∧Elim1 :
     {P Q : Set} -> P ∧ Q -> P
   ∧Elim1 = _∧_.Elim1
   ∧Elim2 :
     {P Q : Set} -> P ∧ Q -> Q
   ∧Elim2 = _∧_.Elim2
```


∨ (disjunction)

- Form: 1.
  If P and Q are propositions, P ∨ Q is a proposition.
  ∨ is defined as a parameterized data type.
- Intro: 1.
  Two constructors correspond two introduction rules.
  ∨Intro1 takes P's proof and returns P∨Q's proof
  ∨Intro2 takes Q's proof and returns P∨Q's proof
- Elim: 2.
  ∨Elim corresponds ∨-elimination of natural deduction

```
1. data _∨_ (P Q : Set) :
       Set where
     ∨Intro1 : P -> P ∨ Q
     ∨Intro2 : Q -> P ∨ Q

2. ∨Elim :
     {P Q R : Set} -> P ∨ Q ->
     (P -> R) -> (Q -> R) -> R
   ∨Elim (∨Intro1 a) prfP _ =
     prfP a
   ∨Elim (∨Intro2 b) _ prfQ =
     prfQ b
```



⊃ (implication)

- Form: 1.
  - If P and Q are propositions, P⊃Q is a proposition.
  - Using function type P-> Q, it is defined as a parameterized data type.
- Intro: 1.
  - ⊃'s constructor corresponds introduction.
- Elim: 2.
  - Defined by function application.
  - We introduce new type ⊃ instead of using just -> to treat logical connectives uniformly as data types.

```
1. data _⊃_ (P Q : Set)
     : Set
     where
       ⊃Intro :
         (P -> Q) -> P ⊃ Q

2.  ⊃Elim :
     {P Q : Set} ->
     P ⊃ Q ->
     P ->
     Q
    ⊃Elim (⊃ Intro f) a =
      f a
```


¬ (negation)

Negation is defined by using implication and absurdity.

- Form: 1.
  - If P is a proposition, ¬P is a proposition.
- Intro: 2.
  - ¬Intro takes a function from P's proof to absurdity and returns ¬P's proof.
- Elim: 3.
  - ¬Elim receives P's proof and ¬P's proof, returns a proof of absurdity.

```
1. ¬ : Set -> Set
   ¬ P = P ⊃ ⊥

2. ¬Intro :
     {P : Set} ->
     (P -> ⊥) ->
     ¬ P
   ¬Intro pr = ⊃Intro pr

3. ¬Elim :
     {P : Set} ->
     ¬ P -> P ->
     ⊥
   ¬Elim (⊃Intro pr1) pr2 =
     pr1 pr2
```

Example of a proof of tautology

$$\dfrac{\dfrac{\dfrac{[A \wedge B]^{①}}{A}\wedge\text{elim1}}{A \vee B}\vee\text{-intro1}}{A \wedge B \supset A \vee B}\,①\supset\text{-intro}$$

```
1. p1 : (A B : Set) ->
     (A ∧ B) ⊃ (A ∨ B)
   p1 A B =
     ⊃Intro
      (¥ (x : A ∧ B) ->
       ∨Intro1
        (∧Elim1 x))
```


∀ (universal quantifier)

- Form: 1.
  P is a domain of objects.
  Q is a function from P to Propositions.
  A proof of (∀x: P) Q[x] :
  A function receiving an object x of type P and returns a proof of Q x.
  Since, x is a bound variable, its name is meaningless. Hence in Agda, we write ∀ P Q.
- Intro: 1.
  Constructor ∀Intro corresponds introduction rule.
- Elim: 2.
  ∀Elim corresponds elimination-rule

```
1.  data ∀ (P : Set)
           (Q : P -> Set) : Set
     where
      ∀Intro :
       ((a : P) -> Q a) ->
       ∀ P Q

2.  ∀Elim :
     {P : Set} ->
     {Q : P -> Set} ->
     ∀ P Q -> (a : P) -> Q a
    ∀Elim (∀Intro f) a = f a
```

∃ (existential quantifier)

- Form: 1.
  If P is Set and Q is a function from P to propositions, ∃ P Q is a proposition.
  In (∃x:P) Q[x], x is a bound variable. Its name is meaningless, hence in Agda, we write ∃ P Q.
- Intro: 2.
  A proof of (∃x:P) Q[x] :
  A pair of an object a of type P and a proof of Proposition Q a
  Function ∃Intro takes an object a of type P and a proof of Q a, returns a proof of a proof of ∃ P Q
- Elim: 1.

```
1.  record ∃
      (P : Set)
      (Q : P -> Set) : Set
      where
        field
          evidence : P
          ∃Elim : Q evidence

2.  ∃Intro :
      {P : Set} ->
      {Q : P -> Set} ->
      (a : P) ->
      Q a ->
      ∃ P Q
    ∃Intro a prf =
      record { evidence = a ;
               Elim = prf }
```


Agda Language

- Type declaration
  - data … (recursive type)
  - record … (record)
- Function declaration (definitional equality)
- Postulate
- Module declaration → record type
- fixity

Classical Logic, Excluded Middle

- Agda cannot prove Excluded Middle
- postulate:

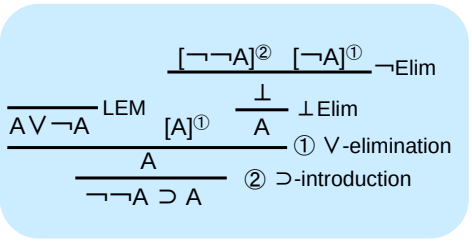
  Introduce a fresh value of the given type.

  Can be lead inconsistency
- LEM: 1.

  A “proof” of Law of Excluded Middle is introduced.

```
1. postulate LEM :
     (P : Set) ->
     P ∨ (¬ P)
```


E.g. Double Negation Elimination




```
DNE :
 {A : Set} ->
 (¬ (¬ A)) ⊃ A
DNE {A} =
 ⊃Intro
 (¥(y : ¬ (¬ A)) ->
  ∨Elim
   LEM
   (¥(w : A) -> w)
   (¥(z : ¬ A) ->
    (⊥Elim
     (¬Elim y z))))
```

Agenda

- On Agda
- Agda as a programming language
- Agda as a proof system
- Further information.

Further Information

- Agda wiki

  Recent Information of Agda (Agda2)

  http://www.cs.chalmers.se/~ulfn/Agda