Agda

Catarina Coquand

August 16, 2005

Background

- Agda is an interactive system for developing proofs in a variant of Martin-Löf's type theory
- It is based on the idea of direct manipulation of proof-term and not on tactics. The proof is a term, not a script.
- The language has ordinary programming constructs such as data-types and case-expressions, signatures and records, let-expressions and modules.
- Has an emacs-interface and a graphical interface, Alfa

System

Agda is an interactive system.

- It consists of a type checker and a termination checker
- Implemented in Haskell
- You will use a simpler version of Agda (with a small library)

A proof of $A \to A$

- The proof of $A \to A$ is the term $\lambda x : A.x$
- In Agda

```
\x -> x
-- alternative: \(x::A) -> x
```

- The syntax of Agda is rather close to Haskell

The identity function

- Function definition

```
id (A::Set) :: A -> A
id = \a -> a
```

- Application:

```
id 0
id 'c'
```

Syntactic Sugar for Function Definitions

```
id (A::Set) :: A -> A
id a =  a
```

Inbuilt type: Pairs

- Pairs are written `A × B`
- A pair is written `(a,b)`
- Projection functions
  - `fst :: A × B -> A`
  - `snd :: A × B -> B`
- Corresponds to logical and

Rule for And

$$\frac{\begin{array}{c}[A\&B]\\ C\end{array} \qquad A \quad B}{C}$$

```
curry (A,B,C::Set) :: (A × B -> C) -> A -> B -> C
curry f a b = f (a,b)
```

And-elimination

Stating the $\&$-elimination rule:

$$\frac{\begin{array}{c}[A \quad B] \\ C\end{array} \qquad A\&B}{C}$$

```
uncurry(A,B,C::Set) :: (A -> B -> C)  -> A × B -> C
```

Swap

Bengt's proof of $A\&B \Longrightarrow B\&A$. We use the $\&$-elimination i.e. `uncurry`

```
swap (A,B::Set) :: (A × B) -> B × A
swap p = uncurry (\x y -> (y,x)) p
```

Inbuilt Type: Booleans

- Type is `Bool`
- Constructed by `True` and `False`
- We have the ordinary `if_then_else` construction

Inbuilt Types: Lists

- Type is `List A`
- Constructed by `Nil` and `:`
- The list `[]` is syntactic sugar for `Nil`
- The list `[1,2,5]` is syntactic sugar for `1:[2,5]`
- The list `[1,2,5]` is syntactic sugar for `1:2:5:Nil`

More Inbuilt Types

- `Integer`: Infinite integers with usual operations except division
- `Char`: Characters with some standard operations
- `String`: Strings are lists of characters

Let-expressions

We can also use let-notation

```
ex :: Integer
ex = let {
         big :: Integer;
         big = 12324567891234566789;
         neg :: Integer;
         neg = negate 1000;
         }
     in big*neg
```

Layout rule

```
ex :: Integer
ex = let big :: Integer
         big = 12324567891234566789
         neg :: Integer
         neg = negate 1000
     in big*neg
```

Equality Type

- We write equality as `a == b`
- It is reflexive, symmetric, transitive, and substitutive
- Equivalent to Leibniz-equality

Typechecking a proof of Reflexivity

We have `refId x` is of type `x == x`,

```
refId 6 ::        6 == 6 -- also the inferred type
refId 6 ::        2 * 3 == 4 + 2
```

This is so since $6 == 6$ and $2 * 3 == 4 + 2$ are convertible. (See Herman Geuver's note on type checking)

Stating a Quantified Theorem

State that $==$ is symmetrical: $\forall\ x\ y.x == y \Longrightarrow y == x$

```
symmEq (A::Set):: (x,y::A) -> x == y -> y == x
symmEq x y  =  ....
```

Equivalent to

```
symmEq (A::Set) :: (x::A) -> (y::A) -> x == y -> y == x
symmEq x y  =  ....
```

Defining Type Synonyms

```
Pred :: Set -> Type
Pred X = X -> Set

Rel ::  Set -> Type
Rel X = X -> X -> Set

Symmetrical (X::Set) ::  (R::Rel X) -> Set
Symmetrical R = (x1,x2::X) |-> (x1 `R` x2 -> x2 `R` x1)

symmEq (A::Set) :: Symmetrical (==)
symmEq x1 x2 = ...
```

Language Constructions : Data Types

We introduce a *new* type by data-type construction

```
data Bool = True | False
data List (A::Set) = Nil | (:) (a::A) (l::List A)
```

Language Constructions :Case Expressions

We can introduce implicitly defined constants by case-expressions. (Should be thought of as defining functions with pattern-equations.)

```
(++)(A::Set) :: List A -> List A -> List A
(++) xs ys = case xs of
                (Nil) -> ys
                (x : xs') -> x:xs'++ys
```

Has to cover all possible cases. The term `xs ++ ys` is on normal form.

Elimination Rule for Lists

```
elimList (A ::Set) ::
         (C::List A -> Set) ->
         C [] ->
         ((x::A) -> (xs::List A) -> C xs -> C (x:xs)) ->
         (xs::List A) ->
         C xs
elimList C c_nil c_con xs =
          case xs of
                 (Nil) -> c_nil
                 (x : xs') -> c_con x xs' (elimList  C c_nil c_con xs')
```

Logic

- Or: `data Plus (X,Y::Set)  = Inl (x::X) | Inr (y::Y)`
- Exists: `data Sigma (X::Set) (Y::X -> Set) = dep_pair (x::X)(y::Y x)`
- Truth: `data Unit :: Set  = unit`
- Absurdity: `data Empty :: Set =`

Or- elimination

```
elimPlus (X,Y::Set) ::
    (C::Plus X Y -> Set) ->
    (c_lft::(x::X) -> C (Inl x)) ->
    (c_rgt::(y::Y) -> C (Inr y)) ->
    (xy::Plus X Y) ->
    C xy
elimPlus C c_lft c_rgt xy = case xy of
                                  (Inl x) -> c_lft x
                                  (Inr y) -> c_rgt y

whenPlus (X,Y,Z::Set) ::  (f::X -> Z) -> (g::Y -> Z) -> (Plus X Y -> Z)
whenPlus = elimPlus (\h -> Z)
```

Absurdity

```
data Empty :: Set =

elimEmpty ::  (C::Empty -> Set) -> (z::Empty) -> C z
elimEmpty C z = case z of { }

whenEmpty ::  (X::Set) -> Empty -> X
whenEmpty X z = case z of { }

Not :: Set -> Set
Not X = X -> Empty
absurdElim (A::Set) :: A -> Not A -> (X::Set) -> X
absurdElim h h’ X = whenEmpty X (h’ h)
```

Inductive families

```
idata (==) (X::Set)  :: X  -> X -> Set where
       refId (x::X) :: (==) x  x
```

Use elimination rules and not case for inductive families.

Language Constructions : Structures/Signature

```
PlusSig :: (A::Set) -> Set
PlusSig A = sig
   zer :: A
   plus :: A -> A -> A

IntPluSig :: PlusSig Integer
IntPluSig = struct
              zer ::  Integer
              zer  = 0
              plus ::  Integer -> Integer -> Integer
              plus  = (+)
```

Another Instance

```
ListPluSig :: (A::Set) -> PlusSig (List A)
ListPluSig A = struct
                 zer ::  List A
                 zer  = []
                 plus ::  List A -> List A -> List A
                 plus  = (++)
```

Using Struct/Sig

```
f :: Integer
f = IntPlusSig.plus IntPlusSig.zer (IntPlusSig.zer +1)

f :: Integer
f  = let open IntPlusSig use plus, zer
     in plus zer (zer + 1)
```

Packages

Packages

```
package Natural where
  open Prelude  use  Pred
  open Boolean  use  Bool, False, True

  data Nat   = Zero | Succ (n::Nat)

  natrec (C::Pred Nat)(bc::C Zero)
         (ic::(n::Nat) -> C n -> C (Succ n))
         (m::Nat)
    :: C m =  ....
  isZero (a::Nat) :: Bool
     = ....
```

Examples : typechecking

```
F :: Set
F  = Bool
f :: Bool -> F
f = \a -> a
```

Gives an equality constraint:

```
Bool = F
```

We must compute F to see that they are equal.

Example : Typechecking

```
F :: (A::Set) -> Set
F = \A -> A

f :: (B::Set) -> B -> F B
f = \B -> \a -> a
```

Gives the equality constraint:

```
B = F B
```

Meta-variables

- A meta-variable can only occur in **one** typing constraint.
- The result of typechecking is a set of typing constraints and equality constraints instead of a yes and no answer when type-checking terms with meta-variables.
- Using higher-order unification will sometimes (often) solve the constraints.

Meta-variables

```
f :: (A::Set) -> (a::A) -> A
f  = \(B::Set) -> \(b::B) -> ?
```

Is type correct if

$$B : Set, b : B \vdash\ ? : B$$

Examples Meta Variables ctd

```
f :: (A::Set) -> (a::A) ->  A
f  = \(B::Set) -> \(b::?) -> b
```

Is type correct if

$$B : Set \vdash \ ? \text{ type}$$

and

$$A \equiv \ ?(B = A)$$

Hidden Arguments

We do not have polymorhism, but hidden arguments

```
id (A::Set) :: A -> A
id   a =  a
id 'c'
```

is translated into `id |? 'c'` .

Hidden Arguments ctd

We can write more explicitly

```
id :: (A::Set) |->  A -> A
id   = \(A::Set) |-> \a -> a

id |Char 'c'
```

Emacs-symbols

```
(global-set-key (kbd "C-*") (lambda () (interactive) (insert "\327")))
;;;  Cartesian product
(global-set-key (kbd "C-.") (lambda () (interactive) (insert "\260")))
;;;; Ring
(global-set-key (kbd "C-!") (lambda () (interactive) (insert "\254")))
;;;; not
(global-set-key [f9] (lambda () (interactive) (insert "\330")))
;;;; Empty set
(global-set-key [f10] (lambda () (interactive) (insert "\267"))) ;;;; M
(global-set-key [f11] (lambda () (interactive) (insert "\367")))
```