Dependently-Typed Programming

Kazutaka Matsuda

Today's topic

- Dependently-typed programming
  - Overviews
  - Very brief introduction of Agda

Case: List and Vector

- Lists

```
[] : List a
(1 :: []) : List Nat
(1 :: 2 :: []) : List Nat
```

- Vectors

size information in type

```
[] : Vec a 0
(1 :: []) : Vec Nat 1
(1 :: 2 :: []) : Vec Nat 2
```

Manipulation of Lists

```
head : List a -> a

> head (1 :: [])
1
> head (2 :: 1 :: [])
2
> head []
... runtime error ...
```

Manipulation of List

```
_!!_ : List a -> Nat -> a

> (1 :: 2 :: 3 :: []) !! 0
1
> (1 :: 2 :: 3 :: []) !! 1
2
> (1 :: 2 :: 3 :: []) !! 2
3
> (1 :: 2 :: 3 :: []) !! 3
... runtime error ...
```

Manipulation of Vector

```
head : Vec a (1 + n) -> a

> head (1 :: [])
1
> head (2 :: 1 :: [])
2
> head []
... type error ...
```

(1 :: []) :: Vec Nat 1

(2 :: 1 :: []) :: Vec Nat 2

[] :: Vec Nat 0

Manipulation of Vector

{0,1,...,n-1}

```
_!!_ : Vec a n -> Fin n -> a

> (1 :: 2 :: 3 :: []) !! 0
1
> (1 :: 2 :: 3 :: []) !! 1
2
> (1 :: 2 :: 3 :: []) !! 2
3
> (1 :: 2 :: 3 :: []) !! 3
... type error ...
```

Quiz

- Vector-version of the following func?

```
replicate : Nat -> a -> List a

> replicate 3 0
0 :: 0 :: 0 :: []
> replicate 2 "A"
"A" :: "A" :: []
```

Answer

❖ By ***dependent types***

```
replicate : (n:Nat) -> a -> Vec a n

> replicate 3 0
0 :: 0 :: 0 :: []
> replicate 2 "A"
"A" :: "A" :: []
```

... : Vec Nat 3

... : Vec Nat 2

Dependent Types

- Types that depend on terms
  - dependent products: (n:A) -> B
    - function types whose return types depend on input values

e.g.: `replicate : (n:Nat) -> (a -> Vec a n)`

  - ... we ignore dependent sums for a while

(in Agda notation)

Why Dependent Typing?

- Early detection of errors
  - Errors at compilation instead of runtime

e.g.:
```
head : Vec a (1 + n) -> a

_!!_ : Vec a n -> Fin n -> a
```

- Writing (constructive) proofs

e.g.:
```
f : 2 ≦ 3
g : (n : Nat) -> 0 ≦ n
```

Agda

- A dependently-typed programming language
- Actively developed
- Based on Martin Löf Type Theory
  - an extension of simply-typed λ calculus
  - impredicative
- Allowing only total functions
  - so, head and _!!_ are rejected anyways

How to Install Agda

- Step 1. Install stack
  - see https://docs.haskellstack.org

```
$ curl -sSL https://get.haskellstack.org/ | sh
```

- Step 2. Install Agda (may take a few hours)

```
$ stack install Agda
$ agda-mode setup
```

- Step 3. Install Agda standard library
  - https://agda.readthedocs.io/en/latest/tools/package-system.html

Setting up Prog. Env. for Agda

- Step 1. Install Emacs
  - some people use Atom

- Step 2. Edit ./emacs.d/init.el or .emacs
  - insert the following lines.

```
(load-file (let ((coding-system-for-read 'utf-8))
                (shell-command-to-string "agda-mode locate")))
```

"Hello World" in Agda

```
module hello-world where
open import IO

main = run (putStrLn "Hello, World!")
```

- You may forget this, as it is rather rare to write all the things in Agda

Start with Natural Numbers

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

$$\frac{}{\texttt{zero} : \mathbb{N}} \qquad \frac{\texttt{n} : \mathbb{N}}{\texttt{suc n} : \mathbb{N}}$$

```
one : ℕ
one = suc zero

two : ℕ
two = suc (suc zero)
```

Notational Convenience

- It is tedious to write
  suc (suc (suc (suc (...(suc zero)...)))
- The following tells Agda to map digits to your defined natural numbers

```
{-# BUILTIN NATURAL ℕ #-}
```

e.g.:

```
four : ℕ
four = 4 -- suc (suc (suc (suc zero))))
```

Defining Functions

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

Your First Proof (1/2)

- Preparation

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl)
open Eq.≡-Reasoning using (begin_; _≡⟨⟩_; _≡⟨_⟩_; _∎)
```

Your First Proof (2/2)

```
_ : 2 + 3 ≡ 5
_ = begin
     2 + 3
    ≡⟨⟩ -- unfolding definitions
     suc (suc zero) + 3
    ≡⟨⟩ -- definition of + for suc
     suc (suc zero + 3)
    ≡⟨⟩ -- definition of + for suc
     suc (suc (zero + 3))
    ≡⟨⟩ -- definition of + for zero
     suc (suc 3)
    ≡⟨⟩
     5
    ∎
```

2 + 3 ≡ 5 is a type!

Shorter Proof

```
_ : 2 + 3 ≡ 5
_ = refl
```

Agda knows which terms become equivalent after evaluation

Universal Quantification

- By dependent products

```
0+ : (n : ℕ) → 0 + n ≡ n
0+ n = refl
```

The quantification can be written also as:

```
0+ : ∀ (n : ℕ) → 0 + n ≡ n
0+ n = refl
```

or

```
0+ : ∀ n → 0 + n ≡ n
0+ n = refl
```

How about this?

```
+0 : (n : ℕ) → n + 0 ≡ n
```

The following causes a type error

```
+0 = refl
```

```
…
n + 0 != n of type ℕ
when checking that the expression refl has type n + 0 ≡ n
```

because we cannot evaluate n + 0 to n

cf.

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

Induction by Recursive Definition

```
+0 : (n : ℕ) -> n + 0 ≡ n
+0 zero      = refl
+0 (suc n) = Eq.cong suc (+0 n)
```

```
cong : {A B : Set} {x y : A}
       -> (f : A -> B)
       -> x ≡ y -> f x ≡ f y
```

More Human Readable Version.

```
+0 : (n : ℕ) -> n + 0 ≡ n
+0 zero     = refl
+0 (suc n) = begin
   suc n + 0
   ≡⟨⟩
   suc (n + 0)
   ≡⟨ Eq.cong suc (+0 n) ⟩
   suc n
   ∎
```

Interactive Proofs

- By holes

```
+0 : (n : ℕ) -> n + 0 ≡ n
+0 n = ?
```

```
+0 : (n : ℕ) -> n + 0 ≡ n
+0 n = { }0
```

```
+0 : (n : ℕ) -> n + 0 ≡ n
+0 zero = { }0
+0 (suc n) = { }1
```

...

User-Defined Predicates

```
data _≤_ : (n : ℕ) -> (m : ℕ) -> Set where
  z≤n : {m : ℕ}     -> zero ≤ m
  s≤s : {n m : ℕ} -> n ≤ m -> suc n ≤ suc m
```

$$\frac{}{0 \leq n}\ \texttt{z≤n} \qquad \frac{n \leq m}{\text{suc } n \leq \text{suc } m}\ \texttt{s≤s}$$

```
_ : 0 ≤ 5
_ = z≤n
```

```
_ : 2 ≤ 3
_ = s≤s (s≤s z≤n)
```

Pattern Matching on "Proof"

inhabitants: proofs for n ≤ m

```
≤-anti-sym : ∀ {n m} -> n ≤ m -> m ≤ n -> n ≡ m
≤-anti-sym z≤n z≤n = refl
≤-anti-sym (s≤s r1) (s≤s r2) =
  cong suc (≤-anti-sym r1 r2)
```

Notice that the following cases cannot happen

- ≤-anti-sym z≤n (s≤s _)
- ≤-anti-sym (s≤s _) z≤n

Revisiting Vector Example

```
data Vec (A : Set) : ℕ -> Set where
  []  : Vec A 0
  _::_ : ∀ {n} -> A -> Vec A n -> Vec A (suc n)

infixr 4 _::_
```

```
v0 : Vec ℕ 0
v0 = []

v3 : Vec ℕ 3
v3 = 1 :: 20 :: 30 :: []
```

head & tail

```
head : ∀ {A} {n} -> Vec A (suc n) -> A
head (a :: _) = a

tail : ∀ {A n} -> Vec A (suc n) -> Vec A n
tail (_ :: r) = r
```

The input cannot be [].

```
_ : head v3 ≡ 1
_ = refl
```

cf.

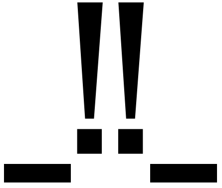
```
v3 : Vec ℕ 3
v3 = 1 :: 20 :: 30 :: []
```

Finite Set

```
data Fin : ℕ -> Set where
  fz : ∀ {n} -> Fin (suc n)
  fs : ∀ {n} -> Fin n -> Fin (suc n)
```

```
_ : Fin 2
_ = fz

_ : Fin 2
_ = fs fz
```




```
_!!_ : ∀ {A n} -> Vec A n -> Fin n -> A
(x :: v) !! fz   = x
(x :: v) !! fs n = v !! n
```

```
_ : v3 !! fz ≡ 1
_ = refl

_ : v3 !! (fs fz) ≡ 20
_ = refl
```

cf.

```
v3 : Vec ℕ 3
v3 = 1 :: 20 :: 30 :: []
```

Replicate

```
replicate : ∀ {A} -> (n : ℕ) -> A -> Vec A n
replicate zero a    = []
replicate (suc n) a = a ∷ replicate n a
```