第17章：Agda的基本概念(1)

胡振江，张伟

信息学院计算机科学技术系

2021年11月10日

What is Agda?

- A dependently typed programming language
  - Implemented in Haskell
- A proof assistant: propositions-as-types
  - Types: propositions
  - Terms (functional programs): proofs

Agda的安装

- 在GHC上用 stack 安装

  > stack install --system-ghc --stack-yaml stack-8.2.2.yaml

  对应GHC的版本

- 关于安装的其他信息，请参照下面的网页：

  https://plfa.github.io/GettingStarted/

一个简单的Agda程序

hello.agda

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

not : Bool -> Bool
not true = false
not false = true
```

与Haskell很相似

Load:	C-c C-l
Compile:	C-x C-c
Compute:	C-c C-n

Dependently Typed Programming in Agda

Reference:

Ulf Norell, Dependently Typed Programming in Agda. Advanced Functional Programming 2008: 230-266.

数据类型和函数定义

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

_+_ : Nat -> Nat -> Nat
zero + m = m
suc n + m = suc (n + m)

_*_ : Nat -> Nat -> Nat
zero * m = zero
suc n * m = m + (n * m)
```

Agda函数一定是全函数，覆盖输入数据的所有可能

操作符

- 中缀操作符

```
_or_ : Bool -> Bool -> Bool
false or x = x
true or _ = true
```

- 三元操作符

```
if_then_else_ : {A : Set} -> Bool -> A -> A -> A
if true then x else y = x
if false then x else y = y
```

参数化(多态) 类型

- 元素可以是任意的A类型的列表

```
infixr 40 _::_
data List (A : Set) : Set where
    [] : List A
    _::_ : A -> List A -> List A
```

```
l1 : List Bool
l1 = true :: false :: true :: []
l2 : List (List Bool)
l2 = (true :: []) :: []
```

依赖函数 (Dependent Functions)

在Agda，我们用 (x : A) -> B 来表示接受A类型的x作为输入，返回一个B类型的值作为输出。需要注意的是，这里的x可以出现在B里.

```
identity : (A : Set) -> A -> A
identity A x = x
```

```
apply : (A : Set) -> (B : A -> Set)
        -> ((x : A) -> B x) -> (a : A) -> B a
apply A B f a = f a
```

(x : A) -> (y : B) -> C 可简写为 (x : A)(y : B) -> C

(x : A) -> (y : A) -> C 可简写为 (x , y : A) -> C

数据类型族 (Data Type Family)

```
data Vec (A : Set) : Nat -> Set where
 [] : Vec A zero
 _::_ : {n : Nat} -> A -> Vec A n -> Vec A (suc n)
```

Vec A n 表示一个长度为 n 的向量类型

```
head : {A : Set}{n : Nat} -> Vec A (suc n) -> A
head (x :: xs) = x
```

head 是一个全函数的定义

```
vmap : {A B : Set}{n : Nat} -> (A -> B)
         -> Vec A n -> Vec B n
vmap f [] = []
vmap f (x :: xs) = f x :: vmap f xs
```

隐式参数

```
id : {A : Set} -> A -> A
id x = x
```

A : Set 可以自动推导出来

```
_o_ :  {A : Set}
       {B : A -> Set}
       {C : (x : A) -> B x -> Set}
       (f : {x : A}(y : B x) -> C x y)
       (g : (x : A) -> B x)
       (x : A) -> C x (g x)
(f o g) x = f (g x)
```

多态函数定义

```
map : {A B : Set} -> (A -> B) -> List A -> List B
map f [] = []
map f (x :: xs) = f x :: map f xs


_++_ : {A : Set} -> List A -> List A -> List A
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

点模式匹配：静态可推导信息

```
data Vec2 (A : Set) : Nat -> Set where
    nil : Vec2 A zero
    cons : (n : Nat) -> A -> Vec2 A n -> Vec2 A (suc n)
```

```
vmap2 : {A B : Set}(n : Nat)
          -> (A -> B) -> Vec2 A n -> Vec2 B n
vmap2 .zero f nil = nil
vmap2 .(suc n) f (cons n x xs) = cons n (f x) (vmap2 n f xs)
```

```
vmap3 : {A B : Set}(n : Nat)
          -> (A -> B) -> Vec2 A n -> Vec2 B n
vmap3 zero f nil = nil
vmap3 (suc n) f (cons .n x xs) = cons n (f x) (vmap2 n f xs)
```

荒谬模式匹配

定义一个类型族：给定一个自然数n，它定义比n小的数字。

```
data Fin : Nat -> Set where
    fzero : {n : Nat} -> Fin (suc n)
    fsuc : {n : Nat} -> Fin n -> Fin (suc n)
```

显然Fin Zero类型的数据是无法构造的。

```
magic : {A : Set} -> Fin zero -> A
magic ()
```

```
_!_ : {n : Nat}{A : Set} -> Vec A n -> Fin n -> A
[] ! ()
(x :: xs) ! fzero = x
(x :: xs) ! (fsuc i) = xs ! i
```

作为证明的程序

- Agda的类型系统足以将任意命题表示一个类型，而其类型中的元素是该命题的证明。
- 假命题、真命题

```
data False : Set where
record True : Set where
```

- 将可判定命题作为布尔值并定义

```
isTrue : Bool -> Set
isTrue true = True
isTrue false = False
```

isTrue b 是“b等于true”的证明的类型

- 应用:安全的列表查询

```
lookup : {A : Set}(xs : List A)(n : Nat)
       -> isTrue (n < length xs) -> A
lookup [] n ()
lookup (x :: xs) zero p = x
lookup (x :: xs) (suc n) p = lookup xs n p
```

- 定义恒等关系

```
data _==_ {A : Set}(x : A) : A -> Set where
    refl : x == x
```

对于A类型的元素x，我们定义一个“和x相等“证明族：
这个族是以x为索引，而且仅有一个证明构造方法refl。

- 自然树上的“小于或等于”关系

```
data _≤_ : Nat -> Nat -> Set where
    leq-zero : {n : Nat} -> zero ≤ n
    leq-suc : {m n : Nat} -> m ≤ n -> suc m ≤ suc n
```

- 简单的例子

```
2≤3 : suc (suc zero) ≤ suc (suc (suc zero))
2≤3 = leq-suc (leq-suc leq-zero)
```

- 证明≤的传递性

```
leq-trans : {l m n : Nat} ->
              l ≤ m -> m ≤ n -> l ≤ n

leq-trans leq-zero _ = leq-zero
leq-trans (leq-suc p) (leq-suc q) = leq-suc (leq-trans p q)
```

作业

17-1 给定两个自然数，定义 _<_ 判断第一个数小于第二个数的函数。

17-2 定义函数filter p xs从列表xs中去除不满足p的元素。

17-3 假设n x m 矩阵建模为向量的向量：

```
Matrix : Set -> Nat -> Nat -> Set
Matrix A n m = Vec (Vec A n) m
```

给出矩阵转置的定义。

```
transpose : {A : Set}{n  m : Nat} ->
            Matrix A n m -> Matrix A m n
transpose [] = ...
transpose ( v :: mat) = ...
```