Idris
A language with dependent types

Alejandro Gómez-Londoño

EAFIT University

31th March, 2014
What is Idris

“What if Haskell had full dependent types?” ¹

Idris features

- Full dependent types
- Type classes
- where clauses, do notation, let bindings
- Monad comprehensions
- Totality checking
- Cumulative universes
- Tactic based theorem proving
- Simple foreign function interface (to C)
Idris
Basic Types

Z : Nat
50 : Integer
1.23 : Float
True : Bool
'a' : Char
"foo" : String

[1,2,3] : List Integer
[1,2,3] : Vect 3 Integer
data Nat = Z | S Nat

data Bool = True | False

infixr 10 ::
data List a = Nil | (::) a (List a)

record Person : Type where
  MkPerson : (name : String) ->
            (age : Int) -> Person

1Programming in Idris: a tutorial, Edwin Brady January 2012
plus : Nat -> Nat -> Nat
plus Z y = y
plus (S k) y = S (plus k y)

mult : Nat -> Nat -> Nat
mult Z y = Z
mult (S k) y = plus y (mult k y)

fact : Nat -> Nat
fact Z = 1
fact (S k) = (S k)* (fact k)
mirror : List a -> List a
mirror xs = let xs' = reverse xs in 
           xs ++ xs'

even : Nat -> Bool
even Z    = True
even (S k) = odd k where
          odd Z    = False
          odd (S k) = even k

greet : IO ()
greet = do
         putStrLn "What is your name? "
         name <- getLine
         putStrLn ("Hello " ++ name)
Dependent Types

Definition

In conventional programming languages, there is a clear distinction between types and values...

In a language with dependent types, however, the distinction is less clear. Dependent types allow types to “depend” on values - in other words, types are a first class language construct and can be manipulated like any other value.¹

¹Programming in Idris: a tutorial, Edwin Brady January 2012
data Vect : Nat -> Type -> Type where
  Nil : Vect Z a
  (::) : a -> Vect k a -> Vect (S k) a

data VectSum : Nat -> Nat -> Type where
  Nil : VectSum Z Z
  (::) : (b : Nat) -> VectSum k a -> VectSum (S k) (a + b)
Dependent Types
Example on functions

\[(+++) : \text{Vect } n \ a \rightarrow \text{Vect } m \ a \rightarrow \text{Vect } (n + m) \ a\]
\[(+++) \ \text{Nil} \ \ y = y\]
\[(+++) \ (x :: \ xs) \ y = x :: \ xs ++ y\]

\[\text{vecHead} : \text{Vect } n \ a \rightarrow \text{so } \ (n > 0) \rightarrow \ a\]
\[\text{vecHead} \ (x :: xs) \ _ = x\]

\[\text{vecHead'} : \text{Vect } (S \ n) \ a \rightarrow \ a\]
\[\text{vecHead'} \ (x :: xs) = x\]
Dependent Types
Examples on Implicit Arguments

vectMap : (A : Type) -> (B : Type) -> (A -> B) -> Vect n A -> Vect n B
vectMap _ _ f Nil = Nil
vectMap t1 t2 f (x::xs) =
  f x :: vectMap t1 t2 f xs

vectMap' : {A : Type} -> {B : Type} -> (A -> B) -> Vect n A -> Vect n B
vectMap' f Nil = Nil
vectMap' f (x::xs) = f x :: vectMap' f xs

vectMap'' : (a -> b) -> Vect n a -> Vect n b
vectMap'' f Nil = Nil
vectMap'' f (x::xs) = f x :: vectMap'' f xs
data (=) : a -> b -> Type where
refl : x = x

Now some examples...
Theorem Proving
commands and tactics

**compute** Normalizes all terms in the goal (note: does not normalize assumptions)

**exact** Provide a term of the goal type directly

**trivial** Satisfies the goal using an assumption that matches its type

**intro** If your goal is an arrow, turns the left term into an assumption

**intros** Exactly like intro, but it operates on all left terms at once

**let** Introduces a new assumption; you may use current assumptions to define the new one

---

Theorem Proving
commands and tactics

- **rewrite** Takes an expression with an equality type \((x = y)\), and replaces all instances of \(x\) in the goal with \(y\). Is often useful in combination with 'sym'

- **state** Displays the current state of the proof

- **term** Displays the current proof term complete with its yet-to-be-filled holes

- **undo** Undoes the last tactic

- **qed** Once the interactive theorem prover tells you “No more goals,” you get to type this in celebration!

---