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]

[1] Edwin Brady (2013). Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming, 23, pp 552-593.

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)

```
Z : Nat
50 : Integer
1.23 : Float
True : Bool
```

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

```
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
```

[1] Programming in Idris: a tutorial, Edwin Brady January 2012

Idris
Data Types[1]

```
data Nat = Z | S Nat

data Bool = True | False
```

[1] Programming in Idris: a tutorial, Edwin Brady January 2012

IDRIS

Data Types[1]

```
data Nat = Z | S Nat

data Bool = True | False

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

[1] Programming in Idris: a tutorial, Edwin Brady January 2012

Idris

Data Types[1]

```
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
```

[1] Programming in Idris: a tutorial, Edwin Brady January 2012

Idris

functions[1]

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

[1] Programming in Idris: a tutorial, Edwin Brady January 2012

Idris

functions[1]

```
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)
```

[1] Programming in Idris: a tutorial, Edwin Brady January 2012

IDRIS

functions[1]

```
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)
```

[1] Programming in Idris: a tutorial, Edwin Brady January 2012

Idris

do,where,let[1]

```
mirror : List a -> List a
mirror xs = let xs' = reverse xs in
                xs ++ xs'
```

[1] Programming in Idris: a tutorial, Edwin Brady January 2012

IDRIS
do,where,let[1]

```
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
```

[1] Programming in Idris: a tutorial, Edwin Brady January 2012

Idris

do,where,let[1]

```
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)
```

[1] Programming in Idris: a tutorial, Edwin Brady January 2012

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]

[1] Programming in Idris: a tutorial, Edwin Brady January 2012

Dependent Types

Example on data types

```
data Vect : Nat -> Type -> Type where
  Nil : Vect Z a
  (::) : a -> Vect k a -> Vect (S k) a
```

Dependent Types

Example on data types

```
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

```
(++) : Vect n a -> Vect m a -> Vect (n + m) a
(++) Nil ys          = ys
(++) (x :: xs) ys = x :: xs ++ ys
```

Dependent Types
Example on functions

```
(++) : Vect n a -> Vect m a -> Vect (n + m) a
(++) Nil ys         = ys
(++) (x :: xs) ys = x :: xs ++ ys
```

```
vecHead : Vect n a -> so (n > 0) -> a
vecHead (x :: xs) _ = x
```

Dependent Types

Example on functions

```
(++) : Vect n a -> Vect m a -> Vect (n + m) a
(++) Nil ys         = ys
(++) (x :: xs) ys = x :: xs ++ ys

vecHead : Vect n a -> so (n > 0) -> a
vecHead (x :: xs) _ = x

vecHead' : Vect (S n) a -> a
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
```

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 : 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
```

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
```

Theorem Proving

```
data (=) : a -> b -> Type where
  refl : x = x
```

Theorem Proving

```
data (=) : a -> b -> Type where
  refl : x = x
```

Now some examples...

Theorem Proving

commands and tactics [1]

- **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

[1] IDRIS-wiki, https://github.com/idris-lang/Idris-dev/wiki/Manual

Theorem Proving

commands and tactics [1]

- **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!

[1] IDRIS-wiki, https://github.com/idris-lang/Idris-dev/wiki/Manual