A Tour of Dependent Types with Idris

Alex Silva
alex@unendli.ch

What is Idris

- *"What if Haskell had full dependent types?"*
- Pure, strict functional programming language
- Interfaces (has Eq, Ord, Semigroup, Monoid, Functor, Applicative, Monad, Foldable, Traversable etc.)
- Monadic I/O
- *do*-notation not restricted to monads
- Function overloading
- Separate String type
- Type-level functions are just regular functions
- Type-driven development with holes
- Unlike Coq and Agda, goal is safe systems programming instead of theorem proving

Syntax differences

- Haskell

```haskell
reverse :: [a] -> [a]
reverse ls = case ls of
               [] -> []
               (x:xs) -> reverse xs ++ [x]
```

- Idris

```idris
reverse : List a -> List a
reverse ls = case ls of
               [] => []
               x :: xs => reverse xs ++ [x]
```

(this presentation uses semantic highlighting for Idris code)

Dependent types

- A dependent type is a type whose definition depends on a value
- Very expressive types that can model intent very precisely
- The compiler can statically prove strong properties of the code
    - Prove that a list is not empty before taking the first element
    - That a list contain an element before removing it
    - That a sorted list is a permutation of the input list
    - Only close opened files, and don't close a file twice
    - That an email is well-formed

Dependent types

- Π-types (dependent functions)

```
getStringOrInt : (x : Bool) → case x of
                                False ⇒ String
                                True ⇒ Int
getStringOrInt False = "false"
getStringOrInt True = 1
```

- Σ-types (dependent pairs)

```
depPair : (x : Bool ** stringOrInt x)
depPair = (True ** 1)
```

Creating types dynamically

```
data Nat = Z | S Nat

AdderType : Nat -> Type
AdderType 0 = Int
AdderType (S k) = Int -> AdderType k

adder : (numargs : Nat) -> Int -> AdderType numargs
adder 0 acc = acc
adder (S k) acc = \next => adder k (next + acc)
```

GADT syntax

```
data Fin : Nat → Type where
  FZ : Fin (S k)
  FS : Fin k → Fin (S k)

data Vect : Nat → Type → Type where
  Nil : Vect 0 a
  (::) : a → Vect k a → Vect (S k) a

data Parity : Nat → Type where
  Even : Parity (n + n)
  Odd : Parity (S (n + n))
```

Dependent types in functions

```
append : Vect m a → Vect n a → Vect (m + n) a
append [] y = y
append (x :: z) y = x :: append z y

index : Fin len → Vect len elem → elem
index FZ (x :: xs) = x
index (FS k) (x :: xs) = index k xs
```

Records

```
data Colour = Blue | Red | Green
data PriceCategory = A | B | C

record Product where
  constructor MkProduct
  productName : String
  colour      : Colour
  priceCat    : PriceCategory

productClassA : (name : String) →
                (colour : Colour) →
                {auto notGreen : Either (colour = Blue) (colour = Red)} →
                Product
productClassA name colour = MkProduct name colour A
```

Stack language

```
data StackLang : Type -> Vect h1 Type -> Vect h2 Type -> Type where
  Push : a -> StackLang () xs (a :: xs)
  Pop : StackLang a (a :: xs) xs
  Add : Num a => a -> a -> StackLang a v v
  Mul : Num a => a -> a -> StackLang a v v
  Pure : a -> StackLang a v v
  (>>=) : StackLang a v1 v2 ->
          (a -> StackLang b v2 v3) -> StackLang b v1 v3
```

Stack language programs

```
dup : StackLang () (ty :: s) (ty :: ty :: s)
dup = do
  x ← Pop
  Push x
  Push x

swap : StackLang () (t1 :: t2 :: s) (t2 :: t1 :: s)
swap = do
  x ← Pop
  y ← Pop
  Push x
  Push y
```

Stack language programs

```
add : Num ty ⇒ StackLang () (ty :: ty :: s) (ty :: s)
add = do
  x ← Pop
  y ← Pop
  z ← Add x y
  Push z

mul : Num ty ⇒ StackLang () (ty :: ty :: s) (ty :: s)
mul = do
  x ← Pop
  y ← Pop
  z ← Mul x y
  Push z
```

Stack language programs

```
square : Num ty ⇒ StackLang () (ty :: s) (ty :: s)
square = do dup; mul

squareSum : Num ty ⇒ StackLang () (ty :: ty :: s) (ty :: s)
squareSum = do square; swap; square; add
```

Zipper data structure

```
data Zipper : (n : Nat) -> (m : Nat) -> (elem : Type) -> Type where
  ZNil : Zipper 0 0 elem
  ZCons : (front : Vect n elem) -> (back : Vect (S m) elem) ->
            Zipper n (S m) elem

cursor : (z : Zipper n (S m) a) -> a
cursor (ZCons front (x :: xs)) = x

right : (z : Zipper n (S (S m)) a) -> Zipper (S n) (S m) a
right (ZCons front (x :: xs)) = ZCons (x :: front) xs

left : (z : Zipper (S n) (S m) a) -> Zipper n (S (S m)) a
left (ZCons (x :: xs) back) = ZCons xs (x :: back)
```

Zipper from lists and vectors

```
fromList : (l : List a) → Zipper 0 (length l) a
fromList [] = ZNil
fromList l@(x :: xs) = ZCons [] (fromList l)

fromVect : (v : Vect n a) → Zipper 0 n a
fromVect [] = ZNil
fromVect v@(x :: xs) = ZCons [] v
```

Removing element at the cursor

```
delete : Zipper n (S m) a → (p ** q ** (n + m = p + q, Zipper p q a))
delete (ZCons [] (x :: [])) = (_ ** _ ** (Refl, ZNil))
delete (ZCons (y :: xs) (x :: [])) =
    (_ ** _ ** (deleteProof xs, ZCons xs [y]))
  where
    deleteProof : Vect len a → S (plus len 0) = plus len 1
    deleteProof {len} _ =
      rewrite plusCommutative len 1 in
        rewrite plusZeroRightNeutral len in Refl
delete (ZCons front (x :: b@(y :: xs))) = (_ ** _ ** (Refl, ZCons front b))
```

Converting Zipper to Vector

```
toVect : Zipper n m a -> (p ** (n + m = p, Vect p a))
toVect ZNil = (_ ** (Refl, []))
toVect (ZCons front back) =
    (_ ** (toVectProof front back, reverse front ++ back))
  where
    toVectProof : (front : Vect n a) ->
                  (back : Vect (S m) a) ->
                  plus n (S m) = (n + (S m))
    toVectProof {n} {m} _ _ = Refl
```

References

- https://www.idris-lang.org/
- http://docs.idris-lang.org/en/latest/tutorial/index.html
- https://www.manning.com/books/type-driven-development-with-idris
- The Power of Pi http://www.staff.science.uu.nl/~swier004/publications/2008-icfp.pdf
- Dependent Types in the Idris Programming Language https://www.youtube.com/watch?v=zSsCLnLS1hg