Interactive Programming with Idris

Elliot Potts

March 2, 2017

What is Idris?

- General purpose
- Purely functional
- Eagerly evaluated
- Dependently typed

```
data List : Type -> Type where
  Nil : List elem
  (::) : elem -> List elem -> List elem
```

```
data [a] = [] | a : [a]
```

Semantic Highlighting

data
type
bound variable
function
keyword
implicit

```
data List : Type -> Type where
  Nil : List elem
  (::) : elem -> List elem -> List elem
```

```
data [a] = [] | a : [a]
```

```
listsum : List Integer -> Integer
listsum [] = 0
listsum (x :: xs) = x + (listsum xs)

mean : List Integer -> Integer
mean xs = (listsum xs) `div` (toIntegerNat (length xs))

main : IO ()
main = do putStrLn (show (mean [1,2,3]))
          putStrLn (show (mean [1,1,1]))
          putStrLn (show (mean []))
```

```
data Nat : Type : where
  Z : Nat
  S : Nat -> Nat
```

```
data Vect : Nat -> Type -> Type where
  Nil : Vect 0 elem
  (::) : elem -> Vect len elem -> Vect (S len) elem
```

```
vectsum : Vect k Integer -> Integer
vectsum [] = 0
vectsum (x :: xs) = x + (vectsum xs)

mean : Vect (S k) Integer -> Integer
mean xs = (vectsum xs) `div` (toIntegerNat (length xs))

main : IO ()
main = putStrLn (show (mean [1,2,3]))
       putStrLn (show (mean [1,1,1]))
       putStrLn (show (mean []))
```

```
myZipWith : (a -> b -> c) -> Vect k a -> Vect k b -> Vect k c
myZipWith f [] [] = []
myZipWith f (x :: xs) (y :: ys) = f x y :: myZipWith f xs ys

myMap : (a -> b) -> Vect k a -> Vect k b
myMap f [] = []
myMap f (x :: xs) = f x :: myMap f xs

append : Vect n a -> Vect m a -> Vect (n + m) a
append [] ys = ys
append (x :: xs) ys = x :: append xs ys
```

```
readAllWords : (len : Nat) -> IO (Vect len String)
readAllWords 0 = pure []
readAllWords (S k) =
  do s <- getLine
     case trim s of
       "" => do putStrLn ("Please input " ++ (show (S k)) ++
                          " more words.")
                readAllWords (S k)
       x => xs <- readAllWords k
            pure (x :: xs)
```

- $\forall (w \in A).P(w)$
- $\prod_{(w:A)} P(w)$
- `(w : A) -> P w`

- $\exists(w \in A).P(w)$
- $\Sigma_{(w:A)}P(w)$
- `(w : A ** pf : P w)`

```
readSomeWords : IO (len : Nat ** Vect len String)
readSomeWords = do getLine
  case trim s of
    "" => pure (0 ** [])
     x => do (predLen ** xs) <- readSomeWords
             pure (S predLen ** x :: xs)
```

```
joinWithSpace : String -> String -> String
joinWithSpace a b = a ++ " " ++ b

main : IO ()
main = do
  putStrLn "Input first names, leave a blank line to finish"
  (n ** firstNames) <- readSomeWords
  putStrLn ("Now input " ++ (shown) ++ " last names")
  lastNames <- readAllWords n
  putStrLn "Your full names are: " >>=
  let fullNames = zipWith joinWithSpace firstNames lastNames
  putStrLn (show fullNames)
```