

# An Introduction To Idris

(for the future.)

Donovan Crichton

August 2022

# Preliminaries

- ▶ Slides and Examples available at:  
<https://github.com/donovancrichton/Talks/COMP1100/IntroToIdrisAndGadts>

# Generalised Algebraic Data Types

- ▶ Guarded Recursive DataTypes - HOAS [Xi et al., 2003].
- ▶ First Class Phantom Types - Type Equality from Pattern Matching [Cheney and Hinze, 2003].
- ▶ Generalised Algebraic Data Types - Type Inference [Jones et al., 2004].

# Syntax

What do GADTs actually look like?

```
{-# LANGUAGE GADTs #-}  
-- this is the data constructor  
data Expr a where  
-- these are the type constructors:  
  EInt  :: Int -> Expr Int  
  EBool :: Bool -> Expr Bool  
  EAdd  :: Expr Int -> Expr Int -> Expr Int  
  EAnd  :: Expr Bool -> Expr Bool -> Expr Bool
```

# Limitations of Algebraic Data Types

Imagine a list with a concrete constructor and a polymorphic one.

```
-- this is just fine.  
data List a =  
  Nil  
  | Cons a (List a)  
  | CInt Int (List Int)  
  
-- this is not!  
t1 :: List a -> List a  
t1 Nil = Nil  
t1 (Cons x xs) = xs  
t1 (CInt k ks) = ks
```

## Limitations of Algebraic Data Types 2

- ▶ If we *parameterise* our data types we can't fix that parameter later.
- ▶ In PLT, TT and Logic we have object and meta languages.
- ▶ We would like to use the binding in our meta language for our object language.

# Polymorphic Evaluation

```
{-# Language GADTs #-}  
data Expr a where  
    EInt  :: Int -> Expr Int  
    EBool :: Bool -> Expr Bool  
    EAdd  :: Expr Int -> Expr Int -> Expr Int  
    EAnd  :: Expr Bool -> Expr Bool -> Expr Bool  
  
eval :: Expr a -> a  
eval (EInt x) = x  
eval (EBool b) = b  
eval (EAdd x y) = eval x + eval y  
eval (EAnd p q) = eval p && eval q
```

# Correctness by Construction

- ▶ GADTs now type-check our object language expressions in our meta language.
- ▶ It is *not possible* for me to accidentally write addition on Boolean expressions.
- ▶ Idea - create your data structures in such away that they enforce correctness?.



# Moving Away from Haskell

- ▶ Extensions overload: PolyKinds, DataKinds, KindSignatures, KindAsType, etc.
- ▶ Reification and Reflection: Haskell has two languages!
- ▶ Just to get more expressivity and functionality for GADTS?

# Stephanie Weirich's Dependent Haskell...

Haskell looks different with just *one* extension...

```
{-# LANGUAGE DataKinds, TypeFamilies, PolyKinds,  
    TypeInType, GADTs, RankNTypes, ScopedTypeVariables,  
    TypeApplications, TemplateHaskell,  
    UndecidableInstances, InstanceSigs,  
    MultiParamTypeClasses, TypeOperators,  
    KindSignatures, TypeFamiliyDependencies,  
    AllowAmbiguousTypes, FlexibleContexts,  
    FlexibleInstances #-}
```

# To Idris!

- ▶ Full Support for Dependent Types.
- ▶ Pure Functional Language.
- ▶ Syntactically Similar to Haskell.
- ▶ Term and Type Languages are Identical.
- ▶ It has a book [[Brady, 2017](#)].
- ▶ My Favourite.

## Installation (on Linux)

The README on the repo has good installation instructions.

1. `$ sudo apt-get install chezscheme9,5`
2. `$ git clone https://github.com/idris-lang/idris2`
3. `$ cd idris2`
4. `$ make bootstrap SCHEME=chezscheme9.5`
5. `$ make install`
6. `$ make clean`
7. `$ make all`
8. `$ make install`
9. add `./idris2/bin` to your system path

# DDTS (GADTS) in Idris

Strictness, loss of type inference.

```
data Expr : (a : Type) -> Type where
  Eval  : a -> Expr a
  EAdd  : Num a => Expr a -> Expr a -> Expr a
  EAnd  : Expr Bool -> Lazy (Expr Bool) -> Expr Bool

eval : Expr a -> a
eval (Eval x) = x
eval (EAdd x y) = (eval x) + (eval y)
eval (EAnd p q) = (eval p) && (eval q)
```

Slim instead of Thick. Fat instead of Thin. Full of holes.

- ▶ `:` not `::`
- ▶ `(x :: xs)` not `(x : xs)`
- ▶ `=>` not `->`
- ▶ `?what`  
  
`f : Nat -> Nat`  
`f x =`  
    `case x of`  
        `Z`       `=> ?baseCaseHole`  
        `(S k)` `=> ?stepCaseHole`

# Gentle Dependent Types 1

- ▶ We can *index* our data types as well as *parameterise* them.
- ▶ The index may have a specific element in the return type, the parameter may not.
- ▶ in **List**, *a* is a parameter. In **Expr**, *a* is an index.

## Gentle Dependent Types 2

We can be *precise* about head and tail.

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

hd : Vect (S k) a -> a
hd (x :: _) = x

tail : Vect (S k) a -> Vect k a
tail (_ :: xs) = xs
```



# Scary Dependent Types 1

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

We can state equality between any two types. However we can only construct an element of the equality type if  $a$  and  $b$  are the same thing.

This happens by beta reduction.

## Scary Dependent Types 2

Interactive Demo on defining  $(++)$

- ▶ What is rewrite? Goal changing?
- ▶ Also implicits?
- ▶ More proofs like this, more practice? Port from Coq!
- ▶ Read [logical foundations](#) and skip the tactics!

# Okay...but that's not a proof!

- ▶ A proof based on intuitionistic higher order logic.
- ▶ Thanks to the curry-howard isomorphism.
- ▶ Commonly known as 'Propositions as Types' [[Wadler, 2015](#)].
- ▶ As Phillip Wadler will now [show us](#)!

## Next Week

- ▶ Functor, Applicatives, Monoids and Monads!
- ▶ IO (Finally!)
- ▶ suggestions? (Plan: Parser Combinators, FRP, Lenses, and more)

# References

- E. Brady. *Type-driven development with Idris*. Simon and Schuster, 2017.
- J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.
- S. P. Jones, G. Washburn, and S. Weirich. Wobbly types: type inference for generalised algebraic data types. Technical report, Technical Report MS-CIS-05-26, Univ. of Pennsylvania, 2004.
- P. Wadler. Propositions as types. *Communications of the ACM*, 58 (12):75–84, 2015.
- H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *ACM SIGPLAN Notices*, volume 38, pages 224–235. ACM, 2003.