

An Introduction To Dependent Types

Donovan Crichton

February 2018

Preliminaries

- ▶ Slides and Examples available at:
`https://github.com/donovancrichton/talks.git`
- ▶ This talk: `/BFPG/2018-2-13-IntroDepTypes`

A quick note about Idris

- ▶ Idris is a strict-by-default pure functional language that supports first class Types.
- ▶ Idris syntax is very similar to Haskell.
- ▶ Implicits allow references to types implicit in a function, such as:

```
length : Vec n a -> Nat  
length {n} vec = n
```

A Brief Definition.

What are dependent types?

- ▶ A dependent type is a type whose complete definition depends on some value.
- ▶ This is very different to an ordinary parameterised ADT, where the definition depends on the **type** of the parameter(s) only.

Ordinary ADT

```
data MyType a = MyStr String a | MyInt Int a
```

Domain Specific Languages

- ▶ Specific (as opposed to general) purpose language implementation.
- ▶ Used for key domain areas: Salary Calculation, Scripting, etc.
- ▶ Limitations were a key driving force for dependent types[1][2].

Recursive ADTs

```
data Expr a
  = I Int
  | B Bool
  | Add (Expr Int) (Expr Int)
  | And (Expr Bool) (Expr Bool)

--eval Int
f :: Expr a -> Maybe Int
f (I x)      = Just x
f (Add x y)  = pure (+) <*> (f x) <*> (f y)
f _         = Nothing

--eval Bool
g :: Expr a -> Maybe Bool
g (B x)      = Just x
g (And x y)  = pure (&&) <*> (g x) <*> (g y)
g _         = Nothing
```

Type Class to the Rescue!

```
data Expr a
  = I Int
  | B Bool
  | Add (Expr Int) (Expr Int)
  | And (Expr Bool) (Expr Bool)
```

```
class Eval a where
  eval :: Expr a -> a
```

```
instance Eval Int where
  eval (I x)      = x
  eval (Add x y) = (eval x) + (eval y)
```

```
instance Eval Bool where
  eval (B x)      = x
  eval (And x y) = (eval x) && (eval y)
```

Recursive ADTs - Problems

- ▶ We'd like a way to apply a single function (eval) to our class of type constructors (Expr).
- ▶ Type classes work for the previous example, but things start to go pear shaped when we want to constrain our type constructors with type classes.
- ▶ More complicated expressions require multiple type parameters that are only used by a few type constructors.
- ▶ This example requires a deprecated extension, and is generally considered poor practice.

```
data Num a => Expr a
  = N a
  | B Bool
  | Add (Expr a) (Expr a)
  | And (Expr Bool) (Expr Bool)
```


GADTs

- ▶ Generalised Algebraic Data Types (or Dependent Data Types) are a generalisation of ADTs, hence the name.
- ▶ The development of GADTs was strongly motivated by this restriction on type class constraints, particularly on their decomposition.
- ▶ Particularly useful when you want to generalise a function across a group or family of data.

GADTs - An Example

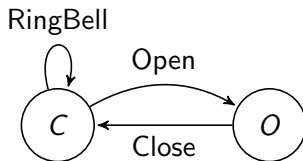
```
{-# LANGUAGE GADTs #-}  
data Expr a where  
    Lift :: (Show a) => a -> Expr a  
    Add  :: Num a => Expr a -> Expr a -> Expr a  
    And  :: Expr Bool -> Expr Bool -> Expr Bool  
  
-- now this works!  
eval :: Expr a -> a  
eval (Lift x) = x  
eval (Add x y) = (eval x) + (eval y)  
eval (And x y) = (eval x) && (eval y)
```

GADTs - Other Information

- ▶ The *a*'s inside the constructor definition are *only* given their explicit types through pattern matching! This can cause problems for the unwary.
- ▶ The type parameter of a GADT is *dependent* on the type constructor used to construct the data type. Thus GADTs are a simple form of a dependent type.
- ▶ GADTs can be used to treat a group of different, but related things in a similar way (but that may be different for each specific thing).

GADTS - Validated State Transitions

GADTs can also be used to have the type-checker validate transitions in a finite state machine. Lets consider an automatic door:



Validated FSMs - Examples

- ▶ Presenter to run through examples.

Closed Type Families

- ▶ Closed Type Families (Type-Level Functions in Idris) allow computations to be expressed at the type level!
- ▶ Haskell does not fully support dependent types yet, but can get very close with a fair amount of work.
- ▶ Key Haskell extensions are: DataKinds, KindSignatures, GADTs, TypeFamilies and TypeOperators, along with knowledge of the singletons library and Template Haskell.

Type-Level Functions

Type-level functions allow a function from a value input, to a type output. These then get paired with an ordinary (value-level) function which returns a type of the type-level function:

```
IntOrString : Bool -> Type
IntOrString True = Int
IntOrString False = String
```

```
intOrString : (x : Bool) -> IntOrString x
intOrString True = 6
intOrString False = "Six"
```

First Class Types

- ▶ In order for the previous slide to work, functions have to be able to return types.
- ▶ This suggests that variables must also accept types.
- ▶ This further suggests that types must be a first class construct!
- ▶ If types are a first class construct then we can use types anywhere we can use a function, which is anywhere we can use a value.
- ▶ This also means we can use functions or values where we can use types!

Vectors - The Obligatory Example

Through both dependent data-types (GADTs) and type-level functions (closed type families) we can express stronger type constraints:

```
infixr 5 :::

data Vec : Nat -> Type -> Type where
  VNil : Vec 0 a
  (:::) : (x : a) -> (xs : Vec n a) -> Vec (n + 1) a

x : Vec 3 Char
x = 'a' ::: 'b' ::: 'c' ::: VNil

-- this wont typecheck!
y : Vec 4 Char
y = 'a' ::: 'b' ::: 'c' ::: VNil
```

Vectors in Haskell

```
{-# LANGUAGE GADTs #-}  
{-# LANGUAGE DataKinds #-}  
{-# LANGUAGE KindSignatures #-}  
  
infixr 5 :::  
  
data Nat = Z | S Nat  
  
data Vec :: Nat -> * -> * where  
  Nil :: Vec Z a  
  (:::) :: a -> Vec n a -> Vec (S n) a  
  
x :: Vec (S (S (S Z))) Char  
x = 'a' ::: 'b' ::: 'c' ::: Nil  
  
--this will not typecheck  
y :: Vec (S (S (S (S Z)))) Char  
y = 'a' ::: 'b' ::: 'c' ::: Nil
```

Constraints as Types

- ▶ Program constraints can be lifted to the type level, through the use of functions as types.
- ▶ This causes the type checker to act as a kind of proof checker! It is no coincidence that the early dependently typed languages were focused on theorem proving!
- ▶ Complexity can be added to the program types as required, in a kind of pay-as-you-go approach.
- ▶ You're paying in verbosity! Type-level functions are required to be total in Idris, which means they must be defined for *every* possible case.
- ▶ You're also paying in design complexity, the more elaborate the type computation, the more difficult it is to reason about the types involved.
- ▶ Still worth it though!

References



CHENEY, J., AND HINZE, R.

First-class phantom types.

Tech. rep., Cornell University, 2003.



XI, H., CHEN, C., AND CHEN, G.

Guarded recursive datatype constructors.

In *ACM SIGPLAN Notices* (2003), vol. 38, ACM, pp. 224–235.