An Introduction To Dependent Types

Donovan Crichton

February 2018

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 paramterised ADT, where the definition depends on the type of the paramter(s) only.

Ordinary ADT

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

Recursive ADTs

```haskell
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!

```haskell
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, is generally considered poor practice, and more constraints = more type parameters!

```
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 class or family of data.

GADTs - An Example

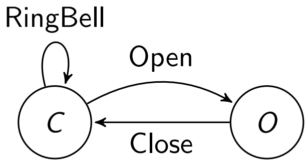
```haskell
{-# LANGUAGE GADTs #-}
data Expr a where
  Lift :: (Show a) => a -> Expr a
  Add  :: Num a => Expr a -> Expr a -> Expr a
  And  :: Expr Bool -> Expr Bool -> Expr Bool

-- 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 - An Example

```haskell
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}

data DoorState = DoorOpen | DoorClosed

data DoorCmd :: DoorState -> DoorState -> * -> * where
  Open     :: DoorCmd DoorClosed DoorOpen   ()
  Close    :: DoorCmd DoorOpen   DoorClosed ()
  RingBell :: DoorCmd DoorClosed DoorClosed ()
  Pure     :: a -> DoorCmd state state a
  Bind     :: DoorCmd state1 state2 a ->
              (a -> DoorCmd state2 state3 b) ->
              DoorCmd state1 state3 b

-- this will throw a type error!
doorProg :: DoorCmd DoorClosed DoorClosed ()
doorProg = Open `Bind` \x ->
           RingBell
```

Validated FSMs - A (Correct) Example

```haskell
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}

data DoorState = DoorOpen | DoorClosed

data DoorCmd :: DoorState -> DoorState -> * -> * where
  Open     :: DoorCmd DoorClosed DoorOpen   ()
  Close    :: DoorCmd DoorOpen   DoorClosed ()
  RingBell :: DoorCmd DoorClosed DoorClosed ()
  Pure     :: a -> DoorCmd state state a
  Bind     :: DoorCmd state1 state2 a ->
              (a -> DoorCmd state2 state3 b) ->
              DoorCmd state1 state3 b

-- this works!
doorProg :: DoorCmd DoorClosed DoorClosed ()
doorProg = RingBell `Bind` \x ->
           Open `Bind` \y ->
           Close
```

Closed Type Families

- A stronger form of dependent type than a GADT. Closed Type Families (Type-Level Functions in Idris) allow computations to be expressed at the type level!
- This in practice allows you to write more expressive types, and exclude a greater number of invalid programs at compile time.
- Haskell does not fully support dependent types yet, but can get fairly close with a fair amount of work.
- Closed type families require a lot of haskell extentions, additional syntax, and boilerplate in the form of singletons, so we'll switch to Idris for the remainder of the talk.

Type-Level Functions

Typel-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

```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 elbarate the type computation, the more difficult it is to reason about the types involved.
- Still worth it though!