Type System Extensions

- Haskell utilises an expressive and powerful type system
- It is not final though, new type systems will be considered for the Haskell 2020 language spec

- **Haskell programmers often treat the type of a function as a static specification**
    - **Lightweight**
    - **Machine checked/verified**
    - **Ubiquitous**
- **Type system designers are interested in**
    - Making **more correct programs** get through the type checker
    - Making **fewer incorrect programs** get through

Looking at more interesting extensions to the type system…

Phantom Types

- Types are declared by a **data** or **newtype** declaration that includes some parameters
- ```
  newtype T a b c = TC stuff
  ```
    - a , b and c are the type parameters to the new type being declared
    - **Implicitly these are *universally quantified***
    - The same is true for function types… the real meaning of:
        - ```
          length :: [a] -> Int
          ```
          is
          ```
          length :: ∀a. [a] -> Int
          ```
        - Haskell actually allows this style of definition (with the XRankNTypes GHC flag)
        - ```
          length :: forall a. [a] -> Int
          ```
    - **Important: there can be multiple constructors for a type, <u>and not every type parameter gets mentioned in every constructor</u>**
    - ```
      data Either a b = Left a | Right b
      ```
    - **<u>A so-called "Phantom Type" is a type parameter that isn't mentioned anywhere in the body of the declaration</u>**
    - ```
      newtype Ty a b = T a
      ```
    - This seems like a strange thing to want…
    - **If we have a type like:**
    - ```
      data Maybe a = Nothing | Just a
      ```
    - **Then the data constructors have type…**

```
Nothing :: ∀a . Maybe a
Just    :: ∀a . a -> Maybe a
```

-
- **Ok. Now let's consider another type...**

```
newtype Lis a = Lis [Int]
```

-
- **The type of this constructor is:**

```
Lis :: ∀a. Int -> Lis a
```

-
    - This looks weird, it's not.
    - You give an a and it returns a function which takes an Int and creates a Lis of type a

```
data Even = Even
data Odd = Odd
```

-
    - We can teach the compiler the difference between odd and even length lists using this

```
consE :: Int -> Lis Even -> Lis Odd
consE i (Lis l) = Lis (i:l)

consO i (Lis l) = Lis (i:l)
```

-
    - So now if we write

      ```
      myList = consO 10 (consE 5 nil)
      ```

      - The compiler deduces:

      ```
      myList :: Lis Even
      ```

    -
    - And if we write…

      ```
      consO 1 myList
      ```

      - The compiler complains:

      ```
      Couldn't match expected type `Odd' with actual type `Even'
      Expected type: Lis Odd
        Actual type: Lis Even
      In the second argument of `consO', namely `aList'
      In the expression: consO 1 myList
      ```

    -
    - **So what is this good for? (https://stackoverflow.com/questions/28247543/motivation-behind-phantom-types)**
    - **Typed Pointers?**

      ```
      data Ptr a = MkPtr Addr
      ```

    -
    - With operations like

```
peek :: Ptr a -> IO a
poke :: Ptr a -> a -> IO ()
```

- Now we can't do this…

```
do ptr <- allocate
   poke ptr (42::Int)
   bad::Float <- peek ptr
```

- **We can have polymorphism in those phantom type parameters**

```
lisMap f (Lis x) = Lis (map f x)
```

- Compiler deduces:

```
lisMap :: (Int -> Int) -> (Lis a) -> (Lis b)
```

- Actually, we can do better:

```
lisMap :: (Int -> Int) -> (Lis a) -> (Lis a)
```

  - Other typical uses of Phantom Types include:
    - **Tracking types in an embedded language**
    - **We can say Exp a to be the type of expressions which evaluate to a value a**
      - We can extend this idea to do even better actually
    - **Object Hierarchy models** (this happens in the Haskell GTK lib)

Existential Types

- **Lists are normally homogenous, by which we mean:**

```
data List a = Nil | Cons a (List a)

-- or, in traditional notation
data [a]    = []  | (:)  a [a]
```

```
[ "foo", "bar", 12.3 ]
```

  - There's obviously no single 'a' that can be universally quantified over the body of the list
  - **The type checker correctly rejects this expression**
  - But if we shuffle the quantifier inside the declaration…
  - **Instead of**

```
data forall a . HList a = ...
```

  - **We could say....**

```
data HList = HNil
           | forall a . HCons a HList
```

        - **This is referred to as an *Existential quantification***
        - **The existential type does not appear in the result type**
    - **This is completely useless**
        - Functions like show will find no instance of a type a which will work for a heterogenous type…
        - Maybe we can get around this
```haskell
data HList = HNil
           | forall a . HCons (a, a -> String) HList

f = HCons ("foo",id) (HCons ("bar",id)
          (HCons (12.3,show) HNil))

printHList :: HList -> IO ()
printHList HNil = return ()
printHList (HCons (x,s) xs)
  = do putStrLn (s x)
       printHList xs
```
    -
        - Wrap operations and the values in a way that allows us to confirm they are printable
        - Looks bloated - **We can actually use a type class system here**
```haskell
data HList = HNil
           | forall a . Show a => HCons a HList

f = HCons "foo" (HCons "bar" (HCons 12.3 HNil))

printHList :: HList -> IO ()
printHList HNil = return ()
printHList (HCons x xs)
```
    -
        - **This is used to mimic the notion of Private Fields from OOP**
            - Create a data type with named fields and existential values
            - Create a type class for the methods
        - **This requires the GHC XExistentialQuantification**

GADTs

- **Generalized Algebraic Data Types**
- Really just a data type where we declare the types of the constructors directly
    - **Sometimes called indexed data types**
- **When you declare a data type:**
```haskell
data Either a b = Left a | Right b

Left  :: a -> Either a b
Right :: b -> Either a b
```
-

- **Same structure as a GADT:**

```
data Either a b where
     Left  :: a -> Either a b
     Right :: b -> Either a b
```

-
    - This doesn't look special
    - **The power comes from being able to restrict some of the type variables in the constructors results**
    - **For example, consider this numbers type:**

```
data Z
data S n
```

    - **(Peano numbers)**
    - These are numbers at type level
    - So no constructors exist…
    - **No we can write a type of lists as a GADT**

```
data List a n where
  Nil :: List a Z
  Cons :: a -> List a m -> List a (S m)
```

    -
    - This lets us make a safe head function:

```
hd :: List a (S n) -> a
hd (Cons a _) = a
```

    -

Motivating GADT Example

- **The classic GADT example is a type-safe interpreter (previous exam question)**
- Using this small arithmetic interpreter...

```
data Expr = N Int
          | Add Expr Expr
          | Mult Expr Exper

eval :: Expr -> Int
eval (N x)         = x
eval (Add e0 e1)   = eval e0 + eval e1
eval (Mult e0 e1) = eval e0 * eval e1
```

-
    - What does it look like if we make more types available?
    - Maybe this…

```
data Expr = N Int
          | Add Expr Expr
          | Mult Expr Exper
          | B Bool
          | Eq Expr Expr
          | If Expr Expr Expr
```

-
    - **So what's the type of eval now..?**
    - We could make some messy type using Either and Maybe..

```haskell
eval :: Expr -> Either Int Bool
```

- But this doesn't work without refactoring...

```haskell
eval (Add e0 e1) = eval e0 + eval e1
```

- We can refactor then...

```haskell
eval (Add e0 e1) = v0 + v1 where
        v0 = lefts eval e0
        v1 = lefts eval e1
```

- **But there's still a blatant flaw regarding our types...**

```haskell
eval (Add (B True) (I 3))
```

- **Making it more complex will make it more robust? Right?**
    - **Wrong. This is just introducing Dynamic typing, giving us the possibility of run-time errors (this is not what Haskell strives for)**

```haskell
data Value = NV Int | BoolV Bool

eval :: Expr -> Maybe Value
eval (N x) = NV x
```

```haskell
...
eval (Eq e0 e1)  = case (eval e0, eval e1) of
   (IV v0, IV v1) -> Just BoolV (v0 == v1)
   (BV v0, BV v1) -> Just BoolV (v0 == v1)
   _ -> Nothing
```

- **We can avoid this using GADTs**

```haskell
data Expr a where
   N :: Int -> Expr Int
   B :: Bool -> Expr Bool
   Add :: Expr Int ->  Expr Int ->  Expr Int
   Mult :: Expr Int ->  Expr Int ->  Expr Int
   Eq :: Eq a => Expr a -> Expr a ->  Expr Bool
--   Eq :: Expr Int -> Expr Int ->  Expr Bool
   If :: Expr Bool ->  Expr a -> Expr a -> Expr a
```

-
    - **We are <u>explicitly</u> providing the type signatures for the constructors for each data type of Expr**
    - We are just providing the obvious signature for the 'simple' types (N or B) and **explicitly giving more information to restrict the more complicated types to provide our sought after behaviour**
        - **E.g. Add**
        - Non-GADT definition: Add Expr Expr
            - Compiler infers the type signature to be Expr a -> Expr a -> Expr a
            - This will allow compilation of Add (Expr True) (Expr 1)
        - **GADT definition: Add :: Expr Int -> Expr Int -> Expr Int**
            - Obviously we only want to add numbers

- Type is restricted to Ints now - it'll spit your bools right back at you
- **We now have a type safe evaluator**

```
eval :: Expr a -> a
eval (N x)          = x
eval (B x)          = x
eval (Add e0 e1)   = eval e0 + eval e1
eval (Mult e0 e1) = eval e0 * eval e1
eval (Eq a b) = eval a == eval b
eval (If c t e) = if (eval c) then (eval t) else (eval e)
```

- Values of If are: Condition, True Expr, Else Expr

Type Kinds

- Another extension of haskell
- Allows us to talk about the *kind* of a type: **the type of a type**

```
Int :: *
Char :: *
[Int] :: *
```

More examples:

```
Maybe :: * -> *
[] :: * -> *
StateT :: * -> (* -> *) -> * -> *
```

- **In other words, the kind tells us how many type arguments need to be supplied to produce a type**
- Haskell has the Kind * built in

Data Kinds

- Using the DataKinds extension Haskell will:
  - **Introduce a new type for each constructor**
  - **Introduce a new kind for each type**
- **Can be used with GADTs to further constrain the use of GADT constructors**
- Example - a Vector type which tracks length

```
data Nat where
  Zero :: Nat
  Succ :: Nat -> Nat

data Vector a (l :: Nat) where
  Nil :: Vector a Zero
  Cons :: a -> Vector a n -> Vector a (Succ n)

vec1 ::  Vector Integer (Succ (Succ (Succ Zero)))
vec1 = 1 `Cons` (2 `Cons` (3 `Cons` Nil))
```

  - GADT definition of peano numbers, and Vector
  - **We encoded the type of the vector length within the type**

- How will we write any useful functions?
- **E.g. append**
- **We need to say that the resulting vector type has a length which is the sum of the two original vector lengths**
- How?!
- **TypeFamilies extension gives us _type level functions_**
- We want something like this...
-
```
append :: Vector a x -> Vector a y -> Vector a (x+y)
```
    - **But we encoded length as Nat, not Int, so need an addition function**
- **So we define a type family for the addition operation**

```
type family Add (x :: Nat) (y :: Nat) :: Nat
```
-
- **So we are saying "There is a type-level operation Add that I can define"**

```
type instance Add Zero y = y
type instance Add (Succ x) y = Succ (Add x y)

append :: Vector a x -> Vector a y -> Vector a (Add x y)
append (Cons x xs) ys = x `Cons` (append xs ys)
append Nil         ys = ys
```
-
    - Create Add through instances of this type-level operation
    - And add it to the type signature of the append function
- **What does GHC think the type of append is?**

```
λ> :t append vec1 vec1
append vec1 vec1
  :: Vector
       Integer ('Succ ('Succ ('Succ ('Succ ('Succ ('Succ 'Zero))))))
```
-
    - Huh.

Dependent Types

https://www.schoolofhaskell.com/user/konn/prove-your-haskell-for-great-safety/dependent-types-in-haskell

- Using GADTs as above simulates dependent types
    - Vector is dependent on the value of length, and functions can reject it based on this value
    - **Data Kinds promotes out values into the type level**
    - Shown above, the kind of a type (type of a type) is naturally defined in terms of '*' and '->'
    - Using Data Kinds we can promote our types (Z and S for example) to type level, allowing us to use them to simulate dependency

- **Note: types introduced to type level by DataKinds cannot have an inhabitant value… That is they may only exist in the type signature of a function as the argument to other types, and not by themselves**
- i.e. We cannot define a function in terms of Z or S Nat, but we can define a function in terms of Vector a (l :: Nat)
- So Data Kinds promotes values to type level… **Type Families lets us define Type Level Functions**
- **See Addition above**