CS4012 Topics in Functional Programming

Glenn Strong

Type system extensions

The Haskell type system is expressive and quite powerful.

But it is hardly the last word. Many different extensions have been explored, and some of these are likely to be at least considered for the Haskell 2020 language specification.

- Haskell programmers often treat the type of a function as a kind of static specification
  - Lightweight
  - Machine checked
  - Ubiquitous
- Type system designers are interested in
  - Making more *correct* programs get through the type checker
  - Making fewer *incorrect* programs get through the type checker
- We will examine some of the more interesting extensions to the Haskell type system

Phantom types

- Recall that 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
```

- In fact, you can write this in Haskell if you like (ghc needs the `-XRankNTypes` flag)

```
length :: forall a. [a] -> Int
```

- There can be multiple constructors for a type
- not every type parameter gets mentioned in every constructor

```
data Either a b = Left a | Right b
```

- A so-called "Phantom type" is a type parameter that isn't mentioned *anywhere* in the body of the declaration

```
newtype Ty a b = T a
```

- Which seems a pretty odd thing to want.
- If we have a type like:

```
data Maybe a = Nothing | Just a
```

- The data constructors have types:

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

- Consider this type

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

- What type does the constructor have here?

```
Lis :: ∀a. Int -> Lis a
```

- Now include these:

```
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)
```

- Now if I write

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

- The compiler deduces:

```
myList :: Lis Even
```

- And if I 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
```

- Is this good for anything practical?
- How about *typed pointers*?

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

- With operations like:

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

- If I write:

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

- The compiler will smack me down!
- We can be polymorphic 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 embedded languages
- We can say `Exp a` be the type of expressions evaluating to a value a
- (we can extend this idea to do even better, actually)
- Object Hierarchy models (this happens in the Haskell `GTK` library)

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 is (obviously) no single 'a' that can be universally quantified over the body of the list.
- The type checker has to reject this expression (and rightly so!)
- But if we shuffle the quantifier *inside* the declaration...
- Instead of

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

- We were to say:

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

- This is referred to as an *Existential quantification* (I know, you expected to see "∃" somewhere).
- The existential type does not appear anywhere in the result type

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

- Sadly, it is useless

```haskell
printHList :: HList -> IO ()
printHList HNil = return ()
printHList (HCons x xs)
  = do putStrLn (show x)
       printHList xs
```

```
No instance for (Show a)
      arising from a use of `show'
...
```

- There is no direct way through this
- But we can go around...

```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 the operations and the values in a way that allow us to confirm they are safe
- Looks a bit bloated
- Actually, with this idea we *can* use type classes to help!

```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)
  = do putStrLn (show x)
       printHList xs
```

- This approach has often been used to model the notion of *Private fields* in object-like type systems
  - Create a data type with named fields and existential values
  - Create a type class for the methods
  - You can't have `.` for structure access but lots of people define an infix `#` operation instead.
- p.s. you need `-XExistentialQuantification` in GHC to enable this
- Or put `{-# LANGUAGE ExistentialQuantification #-}` at the top of your module

GADT's

- We have seen several extensions to the Hindley-Milner system in the past week or two.
- One more, which subsimes some of these approaches, is the notion of a *Generalized Algebraic Data Type*
- This is 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:

```
data Either a b = Left a | Right b
```

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

- As a GADT we declare the same structure thus:

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

- So what?
  - (apart from helping us to realise that data constructors are just functions)
- The extra power comes from being able to *restrict* some of the type variables in the constructors result
- As an example, here is a type for numbers:

```
      data Z
      data S n
```

- I mean, of course, Peano numbers.
- And I mean numbers at the type level.
- So there are no constructors...
- Now we can write a type of lists as a GADT:

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

- This lets us write a safe `head` function:

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

There's a price to pay, of course. We can't do this any more:

```haskell
f 0 = Nil
f 1 = Cons 1 Nil
```

Motivating GADT example

- The classic small GADT example is a type-safe interpreter
- Last year we built a small arithmetic interpreter.

```haskell
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 try to extend the types this has available?
- We might try:

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

- What type should `Eval` have?
- How about:

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

- Nope:

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

- We can make it messy:

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

- But it won't really solve the problem:

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

- So we might try:

```
eval :: Expr -> Maybe (Either Int Bool)
```

- We could go for some kind of complicated expression-result type:

```
data Value = NV Int | BoolV Bool

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

- In other words
- Dynamic typing and possible run-time type errors.
- (boo, hiss!)

- We can do better using GADT's
- The GADT definition:

```
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
```

- Now we can write an evaluator that is type-safe!

```
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)
```

Type Kinds

There is one more extension in Haskell that I want to talk about

Haskell allows us to talk about the *Kind* of a type - that is, 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

When we use the `DataKinds` extension Haskell will

- Introduce a new *type* for each data constructor
- Introduce a new *kind* for each data type

We can use these with GADTs to further constrain the use of GADT data constructors.

The usual example is to create 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))
```

But wait - if the type length of the vector is encoded in the type then how will we write useful functions?

For example, what's the type of `append`?

We would need a way to say that the resulting vector type has a length that is the sum of the two original vector lengths.

The `TypeFamilies` extension lets us write *type level functions*!

I want to write a type that's something like:

```
append :: Vector a x -> Vector a y -> Vector a (x+y)
```

Of course, since my lengths are `Nat` types rather then `Int` values I can't just use `(+)`.

If I introduce a type family for my addition operator:

```
type family Add (x :: Nat) (y :: Nat) :: Nat
```

I am saying that 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
```

So:

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

Dependent types

The final frontier?

Knowing a program is correct

- How do we know a program is correct?
  - We might write it, then verify the correctness
  - Or, we might prove that a solution exists and then extract a program from that.
  - Or, we might write the program in a language that allows us to state properties
- Types matter to us because they let us check (automatically) properties

```
map :: (a -> b) -> [a] -> [b]
```

- The richer the type structure the more one can say in those properties
- So, the more flexibility one has to write programs while preventing errors
- Types represent static guarantees
- The Simply Typed Lambda Calculus parallels the deduction rules of zeroth-order logic

  - So we could say types correspond to propositions
  - Terms correspond to proofs
  - In the Hindley-Milner type system we have more:
  - We can abstract over propositions (polymorphism) and introduce axiom schemes (parameterised datatypes)

Theorem Proving in Types

- If we take this view then:
  - Encode our propositions as types
  - If the program type checks (i.e. if the checker accepts our definition `f :: T`
  - Then we can conclude that `T` is proven)
- However, we can't express predicates; in brief, our logic is not expressive enough

- *Terms* can depend on *Terms*
  - Function definitions
- *Types* can depend on *Types*
  - Parameterised data types
- *Terms* can depend on *Types*
  - Polymorphic terms

- The missing part of the grid is a way for *Types* to depend on *Terms*
- The name for this is a *dependent type* system

- Agda is
- "a dependently typed functional programming language"
- Haskell-like, but with many important differences too
- "a proof assistant"
- A tool for writing and checking constructive proofs.
- Today we will give a very brief taste of Agda, to demonstrate dependent types

- The classic Agda example is a vector type which maintains vector lengths

```
data ℕ: Set where
  zero : ℕ
  suc : ℕ  → ℕ

data Vec (A : Set) : ℕ → Set where
[] : Vec A zero
_::_ : ∀ {n} → A → Vec A n → Vec A (suc n)
```

Giving us a vector type which contains it's own length

Using this we can give a definition of Map that retains a length invariant

```
map : ∀ {A B n} → (A → B) → Vec A n → Vec B n
```

```
map f [] = []
map f (x :: xs) = f x :: map f
```

We can do some of this with GADT's But we needed type families for this:

```
_+_ : ℕ → ℕ  → ℕ
zero + n = n
(suc n) + m = suc (n + m)
```

Using this...

```
_++_ : ∀ {A n m} → Vec A n → Vec A m → Vec A (n + m)
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

During type checking this equation:

```
[] ++ ys = ys
```

gets the inferred type

```
_ ++ _ : ∀ {A n m} → Vec A n → Vec a m → Vec A (zero + m)
```

While the type of the r.h.s. is:

```
ys : Vec A m
```

The typechecker has to reduce `zero + m` at compile time to find the type!

That's not all we can do. Here's a solution to the "printf" problem we saw described in the "Functional Unparsing" paper.

Some types for describing format strings:

```
data ValidFormat : Set₁ where
   argument : (A : Set) → (A → String) → ValidFormat
   literal  : Char      → ValidFormat

data Format : Set₁ where
   valid   : List ValidFormat → Format
   invalid : Format
```

We can convert a string containing a format string into a `Format` value using the `parse` function. This will produce a *valid* format for any correct format string, and an *invalid* format otherwise

```
parse : String → Format
parse s = parse' [] (toList s)
  where
    parse' : List ValidFormat → List Char → Format
    parse' l ('%' :: 's' :: fmt) = parse' (argument String (λ x → x) :: l) fmt
    parse' l ('%' :: 'c' :: fmt) = parse' (argument Char (λ x → fromList [ x ]) :: l) fmt
    parse' l ('%' :: 'd' :: fmt) = parse' (argument ℕ showNat :: l) fmt
    parse' l ('%' :: '%' :: fmt) = parse' (literal '%' :: l) fmt
```

```
    parse' l ('%' :: c   :: fmt) = invalid
    parse' l (c         :: fmt) = parse' (literal c :: l) fmt
    parse' l []                 = valid (reverse l)
```

A function that converts a format string to a *type*:

```
Args : Format → Set
Args invalid                     = ⊥ → String
Args (valid (argument t _ :: r)) = t → (Args (valid r))
Args (valid (literal _ :: r))    = Args (valid r)
Args (valid [])                  = String

FormatArgs : String → Set
FormatArgs f = Args (parse f)
```

The actual `printf` function. Note that the *type* is computed from the input parameter `f`:

```
sprintf : (f : String) → FormatArgs f
sprintf = sprintf' "" ∘ parse
  where
    sprintf' : String → (f : Format) → Args f
    sprintf' accum invalid                    = λ t → ""
    sprintf' accum (valid [])                 = accum
    sprintf' accum (valid (argument _ s :: l)) = λ t →  (sprintf' (accum ++ s t) (valid l))
    sprintf' accum (valid (literal c :: l))    = sprintf' (accum ++ fromList [ c ]) (valid l
```