Beyond Parametric Polymorphism: Type classes and Monads

Nirav Dave
Computer Science and Artificial Intelligence Laboratory
MIT
October 4, 2011
Hindley-Milner gives us generic functions

- Can generalize a type if the function makes no assumptions about the type:

  \[
  \text{const} :: \forall a b. \ a \to b \to a \\
  \text{const} \ x \ y = x
  \]

  \[
  \text{apply} :: \forall a b. \ (a \to b) \to a \to b \\
  \text{apply} \ f \ x = f \ x
  \]

- What do we do when we need to make an assumption?
A simple sum function

```haskell
-- List data type
data [x] = [] | x : [x]

sum n [] = n
sum n (x:xs) = sum (n + x) xs
```

- *sum* cannot be of type \( a \rightarrow [a] \rightarrow a \), we make use of the type (we need to know how to add to objects in the list).

- Pass in the notion of plus?
Avoiding constraints: Passing in +

\[
\text{sum \ plus \ n \ [] \ = \ n} \\
\text{sum \ plus \ n \ (x:xs) \ = \ sum \ (plus \ n \ x) \ xs}
\]

- Now we can get have a polymorphic type for `sum`

\[
\text{sum :: (a -> a -> a) -> (a -> [a] -> a)}
\]

- When we call `sum` we have to pass in the appropriate function representing addition
Generalizing to other arithmetic functions

- A large class of functions do arithmetic operations (matrix multiply, FFT, Convolution, Linear Programming, Matrix solvers):
  - We can generalize, but we need +, -, *, /, ...

- Create a Numeric “class” type:

```haskell
data (Num a) = Num{
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  (/) :: a -> a -> a
  fromInteger :: Integer -> a
}
```
Generalized Functions w/ “class” types

\[
\begin{align*}
\text{matrixMul} & \quad :: \quad \text{Num } a \rightarrow \text{Mat } a \rightarrow \text{Mat } a \rightarrow \text{Mat } a \\
\text{dft} & \quad :: \quad \text{Num } a \rightarrow \text{Vec } a \rightarrow \text{Vec } a \rightarrow \text{Vec } a
\end{align*}
\]

• All of the numeric aspects of the type has been isolated to the Num type
  – For each type, we built a num instance
  – The same idea can encompass other concepts (Equality, Ordering, Conversion to/from String)

• Issues: Dealing with passing in num objects is annoying:
  – We have to be consistent in our passing of function
  – Defining Num for generic types (\text{Mat } a) requires we pass the correct num a to a generator (\text{num_mat} :: \text{Num } a \rightarrow \text{Num } (\text{Mat } a))
  – Nested objects may require a substantial number of “class” objects

Push “class” objects into type class
Type Classes

Type classes group together related functions (e.g., +, -) that are overloaded over the same types (e.g., Int, Float):

```haskell
class Num a where
   (==), (/=) :: a -> a -> Bool
   (+), (-), (*) :: a -> a -> a
   negate :: a -> a
   ...

instance Num Int where
   x == y = integer_eq x y
   x + y = integer_add x y
   ...

instance Num Float where ...
```
Type Class Hierarchy

```haskell
class Eq a where
    (==), (/=) :: a -> a -> Bool

class (Eq a) => Ord a where
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min :: a -> a -> a
```

- Each type class corresponds to one concept and class constraints give rise to a natural hierarchy on classes.
- Eq is a superclass of Ord:
  - If type `a` is an instance of Ord, `a` is also an instance of Eq.
  - Ord inherits the specification of `(==), (/=)` from Eq.
Laws for a type class

- A type class often has laws associated with it
  - E.g., + in Num should be associative and commutative

- These laws are not checked or ensured by the compiler; the programmer has to ensure that the implementation of each instance correctly follows the law

more on this later
(\text{Num} ~ a) \text{ as a predicate in type definitions}

- We can view type classes as predicates.

- Deals with all the passing we had to do in our data passing fashion.
  - The type implies which objects should be passed in.

Type classes is merely a type discipline which makes it easier to write a class of programs; after type checking the compiler de-sugars the language into pure $\lambda$-calculus.
Novel uses of type classes
Error Handling

- Sometime arithmetic operations may throw an error
  - an error is not a number
  - How should we represent this?

- Maybe type?
  - \((\text{Just } x)\) represents that there was no error and \(x\) is the value
  - \((\text{Nothing})\) represents an error
Handling errors: An Example

\[
\text{quadratic\_sol :: (Num a) => a \to a \to a \to (a,a)}
\]
\[
\text{quadratic\_sol a b c =}
\]
\[
\text{let det = b*b - 4*a*c}
\]
\[
\text{sdet = sqrt det}
\]
\[
\text{in } ((\text{negate b) + sdet)/2),
\]
\[
((\text{negate b) - sdet)/2)
\]

If \(\text{det} < 0\), then we have no solution (error)

Solution: add “error” possibility:
\[
\text{msqrt :: (a \to \text{Maybe}\ a)}
\]
Converting a computation

quadratic_sol :: (Num a) => a -> a -> a
               -> Maybe (a,a)

quadratic_sol a b c =
  let det  = b*b - 4*a*c
      msdet = msqrt det
  in case msdet of
    Nothing -> Nothing
    Just sdet -> Just (((negate b)+sdet)/2),
                   ((negate b)-sdet)/2)

Each use of \textit{msqrt} has to be treated in this manner! What if an input wasn’t valid?

Then all arithmetic has to deal with the Maybe type
Converting the computation - 2

```haskell
quadratic_sol :: (Num a) => Maybe a -> Maybe a -> Maybe a -> Maybe (a,a)
quadratic_sol (Just a) (Just b) (Just c) =
    Just (let det  = b*b - 4*a*c
            sdet = sqrt det
            in ((negate b)+sdet)/2,
                (negate b)-sdet)/2))
quadratic_sol _ _ _ = Nothing
```

This handles errors at inputs, but not "internal" errors (e.g. sqrt)

Error handling has to be "compositional"
New Approach: Convert Everything to Maybe

Instead of doing it ad hoc, let’s lift all of our expressions to Maybe.

\[
pure :: a \rightarrow \text{Maybe } a\\
pure x = \text{Just } x
\]

To do computations on Maybe, we also have to define function application on Maybe:

\[
\Leftarrow :: \text{Maybe } (a \rightarrow b) \rightarrow \text{Maybe } a \rightarrow \text{Maybe } b\\
\Leftarrow (\text{Just } f) (\text{Just } x) = \text{Just } (f x)\\
\Leftarrow _ _ _ _ = \text{Nothing}
\]

\[
\Leftarrow \text{ is analogous to the infix apply } ($) \text{ in Haskell}
\]
Compositional Error Handling

quadratic_sol ::
  (Num a) => Maybe a -> Maybe a -> Maybe a -> Maybe (a,a)
quadratic_sol ma mb mc =
  let mdet = pure (\a b c -> b*b - 4*a*c)
    <*> ma <*> mb <*> mc
  msdet = (pure sqrt) <*> mdet
  in pure (\b sdet -> ((negate b)+sdet)/2,
                 (negate b)-sdet)/2))
    <*> mb <*> msdet

Not quite right - doesn’t handle errors in sqrt
msqrt: sqrt on Maybe inputs

\[
\text{msqrt} :: (\text{Num } a) \Rightarrow \text{Maybe } a \to \text{Maybe } a \\
\text{msqrt} (\text{Just } x) \mid x \geq 0 = \text{Just } (\text{sqrt } x) \\
\text{msqrt} _\_ = \text{Nothing}
\]

msqrt has two types of errors:
1. input is an error (obvious!)
2. internal error because the argument is negative (this error depends on the specific computation)

Because of 2, we can’t represent msqrt as

\[
\text{Maybe } (\text{Int } \to \text{ Int})
\]

only as \( (\text{Maybe } \text{Int } \to \text{ Maybe } \text{Int}) \)!
Compositional Error Handling - 2

quadratic_sol :: (Num a) => Maybe a -> Maybe a -> Maybe a -> Maybe (a,a)
quadratic_sol ma mb mc =
  let mdet = pure (\a b c -> b*b - 4*a*c)
      <*> ma <*> mb <*> mc
  msdet = msqrt mdet
  in pure (\b sdet -> ((negate b + sdet)/2, (negate b - sdet)/2))
      <*> mb <*> msdet

Works, however, it is annoying to think of inputs to all of our functions as Maybes
New Approach: Keep the input pure

So far we considered two options:
1. Write functions which don’t use Maybe and “lift” them to Maybe using pure and <*> (e.g., +)
2. Write functions which explicitly use Maybe, at the cost of having to expose the “Maybe abstraction” (e.g. msqrt)

We’d like to exploit the ease of use of the pure abstraction and simultaneously to be allowed to use the additional Maybe functionality. This can be done with base functions of the form \((a \rightarrow \text{Maybe } b)\)

This requires a “bind” operator:
\[
(\ggg) :: \text{Maybe } a \rightarrow (a \rightarrow \text{Maybe } b) \rightarrow \text{Maybe } b
\]
Bind for the Maybe Type

\[(\ggg) :: \text{Maybe } a \to (a \to \text{Maybe } b) \to \text{Maybe } b \]

\[(\ggg) (\text{Just } x) f = f x \]

\[(\ggg) \text{Nothing } f = \text{Nothing} \]

\[(\ggg)\] allows us to compress multiple layers of Maybe abstraction into a single layer; see the following def

We could have written \((\ggg)\) as:

\[
\begin{align*}
\text{ma} \ggg f &= \text{let } \text{mmb} :: \text{Maybe (Maybe } b) \\
& \quad \text{mmb} = ((\text{pure } f) \lt\lt\lt \text{ma}) \\
& \quad \text{in } \text{join mmb}
\end{align*}
\]

\[
\begin{align*}
\text{join} :: \text{Maybe (Maybe } a) \to \text{Maybe } a \\
\text{join} (\text{Just (Just } x)) &= \text{Just } x \\
\text{join} \_ &= \text{Nothing}
\end{align*}
\]
Compositional Error Handling - 3

```haskell
msqrt :: (Num a) => a -> Maybe a msqrt x | x >= 0 = Just (sqrt x) msqrt _ = Nothing

quadratic_sol :: (Num a) => a -> a -> a -> Maybe (a,a) quadratic_sol a b c = let det = b*b - 4*a*c in (msqrt det) >>= (\sdet -> pure((negate b + sdet)/2, (negate b - sdet)/2))
```

This is much better! We have preserved as much of the pure functional code as possible.
We can also handle Maybe inputs

quadratic_sol :: (Num a) =>
     Maybe a -> Maybe a -> Maybe a -> Maybe (a,a)
quadratic_sol ma mb mc =
   ma >>= \a ->
   mb >>= \b ->
   mc >>= \c ->
      let det = b*b - 4*a*c
          sdet = (sqrt det)
in (msqrt det) >>= (\sdet ->
        pure((negate b + sdet)/2,
               (negate b - sdet)/2))

In this form it’s clear when inputs are “natural” or lifted for the sake of the Maybe “context”
Monads in the type class hierarchy

- **Functor**
  
  ```haskell
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

- **Pointed**
  
  ```haskell
class (Functor f) => (Pointed f) where
  pure :: a -> f a
```

- **Applicative**
  
  ```haskell
class (Pointed f) => Applicative f where
  (<*>) :: f (a -> b) -> f a -> f b
```

- **Monad**
  
  ```haskell
class (Applicative m) => (Monad m) where
  return :: a -> m a -- same as pure
  (>>=) :: m a -> (a -> m b) -> m b
```

Actual hierarchy and functions are different in Haskell because monads preceded other classes.
Another view of Monads: Computational Contexts
Shape preserving computations

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

• similarly
  - `mapTree :: (a -> b) -> (Tree a -> Tree b)`
  - `mapTupleR :: (a -> b) -> ((c, a) -> (c, b))`
  - `mapMaybe :: (a -> b) -> (Maybe a -> Maybe b)`

• `map` does not change the “shape” of a data object - it only affects its contents
Functor type class formalizes maps

class Functor f where
  fmap :: (a -> b) -> (f a -> f b)

instance Functor [] where
  fmap _ [] = []
  fmap g (x:xs) = g x : fmap g xs

instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just (f x)

Notice fmap “lifts” a function to the f “space”
What does shape preservation mean?

- Consider the following `fmap` definition on lists:
  ```haskell
  instance (Functor []) where
      fmap _ [] = []
      fmap f (x:xs) = f x : f x : fmap f xs
  ```

- Does this preserve the shape?
  - No. The shape changes

- All functor instances must obey the following laws:
  ```haskell
  fmap (f . g) = fmap f . fmap g
  fmap id = id
  ```

Remember:

```haskell
id x = x
(f . g) x = f(g x)
```
Pointed type class

- With Functor we have some power to manipulate collections, but we have no way of construct a new collection

```haskell
class (Functor f) => (Pointed f) where
  pure :: a -> f a
```

Pointed Laws:

```
  pure . f = (fmap f) . pure
```
Applicative

- Now we have a way of applying normal functions in the f world, and a way of lofting normal functions to the f world, but can we do anything with f world functions?

- \( f (a \to b) \to f a \to f b \)?

- Is there an full analog for computing in the f world?
Applicative type class

\[
\text{class } (\text{Functor } f) \Rightarrow \text{Applicative } f \text{ where}
\]
\[
\text{pure} :: a \rightarrow f a
\]
\[
(\ltimes\ltimes) :: f (a \rightarrow b) \rightarrow f a \rightarrow f b
\]

- Applicative adds function application in the f “world” (\ltimes\ltimes)
- \text{pure} and \ltimes\ltimes subsume \text{fmap}

Law:
\[
\text{fmap } g \ x = (\text{pure } g) \ltimes\ltimes x
\]
Applicative Examples

```haskell
instance (Applicative Maybe) where
  pure x         = Just x
  Nothing <*> x = Nothing
  (Just f) <*> x = fmap f x

instance (Applicative []) where
  pure x        = [x]
  [] <*> xs = []
  (f:fs) <*> xs = 
  fmap f xs ++ (fs <*> xs)
```
Monad type class: collapsing contexts

class (Monad m) where
  return :: x -> m x  -- pure
  (>>=)  :: m a -> (a -> m b) -> m b

Monad Laws:
Left Identity:
  ((return x) >>= mf) = mf x
Right Identity:
  x >>= (\v -> return v) = x

Associativity:
  (x >>= f) >>= g = x >>= (f >>= g)

Faithfulness of fmap:
  (fmap f xs) = xs >>= (\x -> return (f x))

**do** syntactic sugar for Monads

\[
\begin{align*}
do\ e & \Rightarrow e \\
do\ p \leftarrow e;\ dostmts & \Rightarrow e >> \backslash p \rightarrow do\ dostmts \\
do\ e;\ dostmts & \Rightarrow e >> \backslash_\_ \rightarrow do\ dostmts \\
do\ let\ p = e;\ dostmts & \Rightarrow let\ p = e\ in\ do\ dostmts
\end{align*}
\]

\[
\begin{align*}
do\ (nc1, nw1, nl1) & \leftarrow wc\ filename1 \\
(nc2, nw2, nl2) & \leftarrow wc\ filename2 \\
return\ (nc1 + nc2, nw1 + nw2, nl1 + nl2)
\end{align*}
\]

*versus*

\[
\begin{align*}
wc\ filename1 & >> \backslash (nc1, nw1, nl1) \rightarrow \\
wc\ filename2 & >> \backslash (nc2, nw2, nl2) \rightarrow \\
return\ (nc1 + nc2, nw1 + nw2, nl1 + nl2)
\end{align*}
\]
Comparison of syntax

```haskell
quadratic_sol ma mb mc = do
  a <- ma
  b <- mb
  c <- mc
  let det = b*b - 4*a*c
  sdet <- msqrt det
  return ((negate b)+sdet)/2,(negate b)-sdet)/2)
```

```haskell
quadratic_sol ma mb mc =
  ma >>= \a ->
  mb >>= \b ->
  mc >>= \c ->
    let det = b*b - 4*a*c
    in (msqrt det) >>= (&ndet ->
        return ((negate b + sdet)/2,(negate b - sdet)/2))
```

Next time: Examples

- We’ve seen the underpinnings of Monads and the simple basics of how to use them.

- Next time we’ll discuss more examples of monadic use beyond error and more