CS 457/557 Functional Programming

Lecture 5
Polymorphism; Higher-order functions
Polymorphic Length

len :: [a] -> Int
len [] = 0
len (x:xs) = 1 + len xs

• Polymorphic functions don’t “look at” their polymorphic arguments.
• They use the same code now matter what the type of their polymorphic arguments.

“a” is a type variable. It is lowercase to distinguish it from types, which are uppercase.
Polymorphism

• Consider: \texttt{tag1 \mathit{x} = (\mathit{1}, \mathit{x})}
  \[
  \begin{array}{l}
  > \textbf{:type tag1} \\
  \texttt{tag1 :: a -> (Int,a)}
  \end{array}
  \]

• Other functions have types like this; consider \texttt{(++)}
  \[
  \begin{array}{l}
  ? \textbf{:type (++)} \\
  \texttt{(++) :: [a] -> [a] -> [a]}
  \end{array}
  \]

• What are some other polymorphic functions and their types?
  \begin{itemize}
  \item \texttt{id ::}
  \item \texttt{reverse ::}
  \item \texttt{head ::}
  \item \texttt{tail ::}
  \item \texttt{(:) ::}
  \end{itemize}
Polymorphic data structures

• Polymorphism originates from data structures that don’t care what kind of data they store.

\[
\text{id} :: \text{a} \rightarrow \text{a} \quad -- \text{The ultimate}
\]

\[
-- \text{polymorphic function}
\]

\[
\text{reverse} :: \text{[a]} \rightarrow \text{[a]} \quad -- \text{lists}
\]

\[
\text{tail} :: \text{[a]} \rightarrow \text{[a]}
\]

\[
\text{head} :: \text{[a]} \rightarrow \text{a}
\]

\[
(\_ : \text{a} \rightarrow \text{[a]} \rightarrow \text{[a]}
\]

\[
\text{fst} :: (\text{a, b}) \rightarrow \text{a} \quad -- \text{tuples}
\]

\[
\text{swap} :: (\text{a, b}) \rightarrow (\text{b, a})
\]

• How do we define new data structures with “holes” that can be polymorphic?
Maybe is polymorphic

data Maybe a = Just a | Nothing

Note the types of the constructors:

Nothing :: Maybe a
Just    :: a -> Maybe a

Thus:

Just 3 :: Maybe Int
Just "x" :: Maybe String
Just (3,True) :: Maybe (Int,Bool)
Just (Just 1) :: Maybe (Maybe Int)

Example of its use:

lookup :: a -> [(a,b)] -> Maybe b

lookup k [] = Nothing
lookup k ((k',v):rest) | k == k' = Just v
| otherwise = lookup k rest
Polymorphism from functions as arguments

- Another source of polymorphism comes from functions which take functions as arguments.

```haskell
applyTwice f x = f(f x)
```

Main> :t applyTwice
applyTwice :: (a -> a) -> a -> a

- What's the type of the following useful function?

```haskell
flip f x y = f y x
```
Polymorphism: Functions returned as values

• Consider:

\[
\text{const } x = f \\
\text{where } f \ y = x
\]

Main> (const 3) 5
3

– What’s the type of `const`?

• Another Example:

\[
\text{compose } f \ g \ x = f \ (g \ x)
\]

– What’s the type of `compose`?
– Note: Prelude defines compose as an infix operator

\[
(f \ . \ g) \ x = f \ (g \ x)
\]
Abstraction Over Recursive Definitions

• Recall some definitions from previous chapters.
  • Section 4.1:

        translist []       = []
    transList (p:ps)   = trans p : translist ps

• Section 3.1:

        putCharList []     = []
    putCharList (c:cs) = putChar c : putCharList cs

• There is something strongly similar about these definitions. Indeed, the only thing different about them (besides the variable names) is the function trans vs. the function putChar.

• We can use the abstraction principle to take advantage of this.
Abstraction Yields map

- `trans` and `putChar` are what’s different; so they should be arguments to the abstracted function.
- In other words, we would like to define a function called `map` (say) such that `map trans` behaves like `transList`, and `map putChar` behaves like `putCharList`.
- No problem:

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

- Given this, it is not hard to see that we can redefine `transList` and `putCharList` as:

  ```
  transList   xs = map trans   xs
  putCharList cs = map putChar cs
  ```
**map is Polymorphic**

- The key thing about map is that it is *polymorphic*. Its most general (“principal”) type is:

  \[
  \text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
  \]

- Every use of `map` has a type that is an *instance* of the principal type (obtained by substituting for \(a\) and \(b\)).

- For example, since `trans :: Vertex \rightarrow Point`, then
  \[
  \text{map trans} :: [Vertex] \rightarrow [Point]
  \]
  
  and this use of map has type
  \[
  \text{map} :: (\text{Vertex} \rightarrow \text{Point}) \rightarrow [\text{Vertex}] \rightarrow [\text{Point}]
  \]
Another Pattern: Filtering

• Consider extracting the even numbers from a list:

```haskell
evens :: [Int] -> [Int]
evens [] = []
evens (x:xs) | even x = x:(evens xs)
             | otherwise = evens xs
```

• Or removing the whitespace from a string:

```haskell
nowhite :: String -> String
nowhite "" = ""
nowhite (c:cs) | not (whitesp c) = x : (nowhite cs)
                | otherwise = nowhite cs

where whitesp ' ' = True
     whitesp '\t' = True
     whitesp _ = False
```
Abstracting to filter

- Can define a common function

  ```haskell
  filter :: (a -> Bool) -> [a] -> [a]
  filter p []                 = []
  filter p (x:xs) | p x      = x:(filter p xs)
  | otherwise = filter p xs
  ```

- Now can rewrite

  ```haskell
  evens xs = filter even xs
  ```
  - or just:

    ```haskell
    evens = filter even
    ```

- And

  ```haskell
  nowhite = filter (not . whitesp)
  ```
  - Recall that (.) represents function composition.
List comprehensions revisited

• Recall some uses of the list comprehension notation
  \[
  \text{putCharList } cs = [\text{putChar } c \mid c \leftarrow cs]
  \]
  \[
  \text{evens } xs = [y \mid y \leftarrow xs, \text{ even } y]
  \]

• Observe that this notation incorporates both \textit{map} and \textit{filter}, e.g.
  \[
  \text{putNonWhiteChars } cs =
  \]
  \[
  [\text{putChar } c \mid c \leftarrow cs, \text{ not } (\text{whitespace } c)]
  \]

• Can easily define \textit{map} and \textit{filter} in terms of list comprehension (try it!)

• Actually, list comprehension is defined in terms of \textit{map} and \textit{filter} (and a few other things...)
When to Define
Higher-Order Functions

• Recognizing repeating patterns is the key, as we did for `map`. As another example, consider:

```haskell
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs
```

```haskell
and :: [Bool] -> Bool
and [] = True
and (x:xs) = x && and xs
```

```haskell
myminimum :: [Int] -> Int
myminimum [] = maxBound
myminimum (x:xs) = x \text{ `min` } myminimum xs
```

• Note the similarities. Also note the differences (circled), which need to become parameters to the abstracted function.
When to Define
Higher-Order Functions

• Recognizing repeating patterns is the key, as we did for `map`. As another example, consider:

```haskell
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs

and :: [Bool] -> Bool
and [] = True
and (x:xs) = x && and xs

myminimum :: [Int] -> Int
myminimum [] = maxBound
myminimum (x:xs) = x `min` myminimum xs
```

• Note the similarities. Also note the differences (circled), which need to become parameters to the abstracted function.
Abstracting to foldr

• This leads to:

\[
\begin{align*}
\text{foldr~op~init~[]} &= \text{init} \\
\text{foldr~op~init~}(x:xs) &= x \ `\text{op}` \ (\text{foldr~op~init~xs})
\end{align*}
\]

• Note that foldr is also polymorphic:

\[
\text{foldr~:~:~:~:~(a~\rightarrow~b~\rightarrow~b)~\rightarrow~b~\rightarrow~[a]~\rightarrow~b}
\]

  – We'll see the full power of this polymorphism shortly.

• Previous functions can now be redefined:

\[
\begin{align*}
\text{sum~xs} &= \text{foldr~}(+)~0~xs \\
\text{and~xs} &= \text{foldr~}(\&\&)~\text{True}~xs \\
\text{my\-\minimum~xs} &= \text{foldr~min~max\-Bound~xs}
\end{align*}
\]
Visualizing the effect of \texttt{foldr}

- One useful way to think about what \texttt{foldr} does is to observe what it does on an arbitrary list written using explicit constructors:

\[
\text{foldr op init } \texttt{[x}_1,\texttt{x}_2,\ldots,\texttt{x}_n] \\
= \text{foldr op init } (\texttt{x}_1 : (\texttt{x}_2 : (\ldots (\texttt{x}_n : []) \ldots))) \\
= \texttt{x}_1 \ ` \texttt{op` (x}_2 \ ` \texttt{op` (\ldots (\texttt{x}_n \ ` \texttt{op` init) \ldots))}
\]

- So we can think of \texttt{foldr} as taking a list and replacing each \texttt{(:)} by \texttt{op} and the final \texttt{[]} by \texttt{init}.

\[
\texttt{foldr (+) 0} (\texttt{1 : (2 : (3 : []))})
\]

\[
= 1 + (2 + (3 + 0))
\]

- The \texttt{r} in \texttt{foldr} is because it “folds from the right”.
Mystery folds

• Consider these functions:

\[
\text{mystery1 } xs = \text{foldr } (*) \ 1 \ xs
\]

\[
\text{mystery2 } xs = \text{foldr } k \ 0 \ xs
\quad \text{where } k a b = b + 1
\]

\[
\text{mystery3 } q \ xs = \text{foldr } k \ \text{False } xs
\quad \text{where } k x b = q x \ | \ | \ b
\]

\[
\text{mystery4 } = \text{foldr } (: ) \ [ ]
\]

• What are their types?
• What do they do?
Two Folds are Better than One

• In addition to \texttt{foldr}, the Haskell Prelude defines another function \texttt{foldl} which “folds from the left”:

\[
\text{foldl } \text{op} \text{ init } (x_1 : x_2 : \ldots : x_n : []) = (...((\text{init } \ `\text{op}` \ x_1) \ `\text{op}` \ x_2)\ldots) \ `\text{op}` \ x_n
\]

• Exercise: define \texttt{foldl} using recursion.

• Why two folds? Often they are equivalent, but sometimes using one can be more efficient than the other. For example:

\[
\text{foldr} \ (++) \ [] \ [x,y,z] = x \ ++ \ (y \ ++ \ z) \\
\text{foldl} \ (++) \ [] \ [x,y,z] = (x \ ++ \ y) \ ++ \ z
\]

The former is more efficient than the latter (see textbook).

• In general, one or the other of \texttt{foldl} and \texttt{foldr} may be more efficient and/or lazier in any given circumstance.

• Choosing between them is non-trivial!
Reversing a List

- Obvious but inefficient (why?):
  
  \[
  \begin{align*}
  \text{reverse } [] &= [] \\
  \text{reverse } (x::xs) &= (\text{reverse } xs) ++ [x]
  \end{align*}
  \]

- Much better (why?):
  
  \[
  \begin{align*}
  \text{reverse } xs &= \text{rev } [] xs \\
  \text{where } \text{rev } acc [] &= acc \\
  \text{rev } acc (x:xs) &= \text{rev } (x:acc) \text{ } xs
  \end{align*}
  \]

- This looks a lot like \texttt{foldl}; we can redefine \texttt{reverse} as:
  
  \[
  \begin{align*}
  \text{reverse } xs &= \text{foldl } \text{revOp } [] \text{ } xs \\
  \text{where } \text{revOp } a \text{ } b &= b : a
  \end{align*}
  \]

- Or just as
  
  \[
  \text{reverse } = \text{foldl } (\text{flip } (:)) \text{ } []
  \]