CS 457/557 Functional Programming

Lecture 5
Polymorphism; Higher-order functions
Polymorphic Length

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

\[
\text{len} :: [a] \rightarrow \text{Int}
\]
\[
\text{len} [] = 0
\]
\[
\text{len} (x:xs) = 1 + \text{len} \; xs
\]

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

• Consider: \( \text{tag1 } x = (1, x) \)
  
  \[
  ? : \text{type tag1} \\
  \text{tag1} :: a \rightarrow (\text{Int}, a)
  \]

• Other functions have types like this; consider (++)
  
  \[
  ? : \text{type (++)} \\
  (++): [a] \rightarrow [a] \rightarrow [a]
  \]

• What are some other polymorphic functions and their types?
  
  – \text{id} :: \\
  – \text{reverse} :: \\
  – \text{head} :: \\
  – \text{tail} :: \\
  – (:) ::
Polymorphic data structures

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

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

\[ \text{reverse :: } [a] \rightarrow [a] \quad -- \text{lists} \]
\[ \text{tail :: } [a] \rightarrow [a] \]
\[ \text{head :: } [a] \rightarrow a \]
\[ (:)) :: a \rightarrow [a] \rightarrow [a] \]

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

- 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:

```haskell
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.

applyTwice \( f \ x = f(f\ x) \)

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

- What's the type of the following useful function?
flip \( f \ x \ y = f \ y \ x \)
Polymorphism: Functions returned as values

• Consider:
  
  ```haskell
  const x = f
    where f y = x
  ```

  ```haskell
  Main> (const 3) 5
  3
  ```
  
  – What’s the type of `const`?

• Another Example:

  ```haskell
  compose f g x = f (g x)
  ```

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

  ```haskell
  (f . g) x = f (g x)
  ```
Abstraction Over Recursive Definitions

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

  \[
  \begin{align*}
  \text{translist } [ ] &= [] \\
  \text{transList } (p:ps) &= \text{trans } p : \text{translist } ps \\
  \end{align*}
  \]

- Section 3.1:

  \[
  \begin{align*}
  \text{putCharList } [ ] &= [] \\
  \text{putCharList } (c:cs) &= \text{putChar } c : \text{putCharList } cs \\
  \end{align*}
  \]

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

- We can use the abstraction principle to take advantage of this.
Abstraction Yields \textit{map}

- \texttt{trans} and \texttt{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 \texttt{map} (say) such that \texttt{map trans} behaves like \texttt{transList}, and \texttt{map putChar} behaves like \texttt{putCharList}.
- No problem:
  \[
  \begin{align*}
  \text{map } f \; &\; [\;] \quad = \quad [\;] \\
  \text{map } f \; &\; (x:xs) \quad = \quad f \; x \; : \; \text{map } f \; xs
  \end{align*}
  \]
- Given this, it is not hard to see that we can redefine \texttt{transList} and \texttt{putCharList} as:
  \[
  \begin{align*}
  \text{transList} \; &\; xs \quad = \quad \text{map } trans \; \; xs \\
  \text{putCharList} \; &\; cs \quad = \quad \text{map } \text{putChar} \; \; cs
  \end{align*}
  \]
map is Polymorphic

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

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

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

- For example, since \text{trans} :: \text{Vertex} \to \text{Point}, then
  \[
  \text{map} \text{ trans} :: [\text{Vertex}] \to [\text{Point}]
  \]

  and this use of map has type
  \[
  \text{map} :: (\text{Vertex} \to \text{Point}) \to [\text{Vertex}] \to [\text{Point}]
  \]
Another Pattern: Filtering

- Consider extracting the even numbers from a list:
  \[
  \text{evens} :: \text{[Int]} \rightarrow \text{[Int]}
  \]
  \[
  \text{evens} [] = []
  \]
  \[
  \text{evens} (x:xs) \mid \text{even } x = x:(\text{evens } xs)
  \]
  \[
  | \text{otherwise } = \text{evens } xs
  \]

- Or removing the whitespace from a string:
  \[
  \text{nowhite} :: \text{String} \rightarrow \text{String}
  \]
  \[
  \text{nowhite} "" = ""
  \]
  \[
  \text{nowhite} (c:cs) \mid \text{not } (\text{whitespace } c) = x : (\text{nowhite } cs)
  \]
  \[
  | \text{otherwise } = \text{nowhite } cs
  \]
  \[
  \text{where } \text{whitespace } ' ' = \text{True}
  \]
  \[
  \text{whitespace } '\t' = \text{True}
  \]
  \[
  \text{whitespace } _ = \text{False}
  \]
Abstracting to \textit{filter}

- Can define a common function
  \begin{align*}
  \text{filter} &:: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \\
  \text{filter} \ p \ [] & = [] \\
  \text{filter} \ p \ (x:xs) \mid p \ x & = x: (\text{filter} \ p \ xs) \\
  & \mid \text{otherwise} = \text{filter} \ p \ xs
  \end{align*}

- Now can rewrite
  \begin{align*}
  \text{evens} \ xs & = \text{filter} \ \text{even} \ xs \\
  & \quad \text{or just:} \\
  \text{evens} & = \text{filter} \ \text{even}
  \end{align*}

- And
  \begin{align*}
  \text{nowhite} & = \text{filter} \ (\text{not} . \ \text{whitesp}) \\
  & \quad \text{Recall that (.) represents function composition.}
  \end{align*}
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 \text{map} and \text{filter}, e.g.
  \[
  \text{putNonWhiteChars } cs = \\
  [\text{putChar } c \mid c \leftarrow cs, \text{not (whitesp } c)]
  \]
- Can easily define \text{map} and \text{filter} in terms of list comprehension (try it!)
- Actually, list comprehension is defined in terms of \text{map} and \text{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

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.
When to Define Higher-Order Functions

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

\[
\begin{align*}
\text{sum} & : \ [\text{Int}] \rightarrow \text{Int} \\
\text{sum} \ [\] & = 0 \\
\text{sum} \ (x:xs) & = x + \text{sum} \ xs \\
\text{and} & : \ [\text{Bool}] \rightarrow \text{Bool} \\
\text{and} \ [\] & = \text{True} \\
\text{and} \ (x:xs) & = x \land \text{and} \ xs
\end{align*}
\]

\[
\begin{align*}
\text{myminimum} & : \ [\text{Int}] \rightarrow \text{Int} \\
\text{myminimum} \ [\] & = \text{maxBound} \\
\text{myminimum} \ (x:xs) & = x \ `\text{min}` \ \text{myminimum} \ xs
\end{align*}
\]

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

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

- Note that \texttt{foldr} is also \textit{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:
  
  \[
  \begin{align*}
  \text{sum } xs &= \text{foldr } (+) \ 0 \ xs \\
  \text{and } xs &= \text{foldr } (\&\&) \ \text{True} \ xs \\
  \text{myminimum } xs &= \text{foldr } \text{min} \ \text{maxBound} \ xs
  \end{align*}
  \]
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:

\begin{align*}
\text{foldr ~op~ init ~[x1,x2,...,xn]} &= \text{foldr ~op~ init ~(x1 : (x2 : (... (xn : []) ...)))} \\
&= x1 \ `\text{op}` \ (x2 \ `\text{op}` \ (... \ (xn \ `\text{op}` \ init) ...)) \\
\end{align*}

- 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}.

\begin{align*}
\text{foldr ~(+)} ~0 ~\text{(1 \ : \ (2 \ : \ (3 \ : \ []) \ ))} &= \text{1 + (2 + (3 + 0))} \\
\end{align*}

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

• Consider these functions:

  mystery1 xs = foldr (*) 1 xs

  mystery2 xs = foldr k 0 xs
  where k a b = b + 1

  mystery3 q xs = foldr k False xs
  where k x b = q x || b

  mystery4 = 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”:
  \[
  \text{foldl } \text{op } \text{init } (x_1 : x_2 : \ldots : x_n : []) \Rightarrow \\
  (...((\text{init } `\text{op}` x_1) `\text{op}` x_2)\ldots) `\text{op}` x_n
  \]

- 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:
  \[
  \text{foldr } (++) \{} [] [x, y, z] = x ++ (y ++ z) \Rightarrow \\
  \text{foldl } (++) \{} [] [x, y, z] = (x ++ y) ++ z
  \]
  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?):
  \[
  \text{reverse} \; [] = [] \\
  \text{reverse} \; (x::xs) = (\text{reverse} \; xs) ++ [x]
  \]

- Much better (why?):
  \[
  \text{reverse} \; xs = \text{rev} \; [] \; xs \\
  \quad \text{where} \; \text{rev} \; \text{acc} \; [] = \text{acc} \\
  \quad \quad \text{rev} \; \text{acc} \; (x:xs) = \text{rev} \; (x:\text{acc}) \; xs
  \]

- This looks a lot like \text{foldl}; we can redefine \text{reverse} as:
  \[
  \text{reverse} \; xs = \text{foldl} \; \text{revOp} \; [] \; xs \\
  \quad \text{where} \; \text{revOp} \; a \; b = b : a
  \]

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