COMP0020 Functional Programming

Lecture 9

Combinators and Higher Order Functions

Contents

- Combinators
  - Definition
  - Examples : S, K, I
- Higher Order Functions
  - Definition
  - Example : composition
- Capturing common forms of recursion on lists
  - Examples : map and filter

Combinators

- Definition
  - A combinator is a function that contains no free variables
- Examples :

$$
\begin{array}{lll}
\mathit{fst}\ (x, y) & = x & \\
\mathit{id}\ x & = x & ||\text{“}I\text{”} \\
\mathit{cancel}\ x\ y & = x & ||\text{“}K\text{”} \\
\mathit{swap}\ f\ x\ y & = f\ y\ x & ||\text{“}C\text{”} \\
\mathit{distribute}\ f\ g\ x & = f\ x\ (g\ x) & ||\text{“}S\text{”}
\end{array}
$$

Combinators (2)

- Basis for implementation
- S & K computationally complete
- All required data available via arguments (passed on the stack) so no need to search for distant bindings

Higher Order Functions

- Definition :
  - A Higher Order Function is a function that either (i) takes a function as an argument, or (ii) returns a function as a result, or (iii) both.

$$f\ g\ x = (g\ x)$$
$$h\ x = (+\ x)$$
$$j\ f\ p\ g = (+\ (f\ p)),\ \textit{if}\ p$$
$$= (+\ (g\ p)),\ \textit{otherwise}$$

Higher Order Functions : types

$$
\begin{aligned}
f &:: (* -> **) -> * -> ** \\
f\ g\ x &= (g\ x)
\end{aligned}
$$

$$
\begin{aligned}
h &:: num -> (num -> num) \\
h\ x &= (+\ x)
\end{aligned}
$$

$$
\begin{aligned}
j &:: (bool -> num) -> bool -> (bool -> num) -> (num -> num) \\
j\ f\ p\ g &= (+\ (f\ p)),\ \textit{if}\ p \\
&= (+\ (g\ p)),\ \textit{otherwise}
\end{aligned}
$$

Function composition

$$compose :: (** -> ***) -> (* -> **) -> * -> ***$$
$$compose\ f\ g\ x = f\ (g\ x)$$

- Can partially apply "compose":

$$fred = (compose\ (+1)\ abs)$$
$$main = fred\ (-3)$$

- Built-in operator "."

$$fred = ((+1)\ .\ abs)$$

Function composition (2)

- twice x = x * 2
- many x = twice (twice (twice (twice x)))
- mymany : : num -> num
- mymany = (twice . twice . twice . twice)

Example HOF

- "myiterate" repeatedly applies its second parameter to its final parameter; the final parameter is an accumulator for the result. The function loops n times, where n is the first parameter :

$$myiterate :: num -> (* -> *) -> * -> *$$

$$myiterate\ 0\ f\ state = state$$

$$myiterate\ n\ f\ state = myiterate\ (n-1)\ f\ (f\ state)$$

Example HOF

- *printdots n = myiterate n ((++) ".") ""*

printdots 3
→ *myiterate* 3 ((++) ".") ""
→ *myiterate* 2 ((++) ".") ((++) "." "")
→ *myiterate* 1 ((++) ".") ((++) "." ((++) "." ""))
→ *myiterate* 0 ((++) ".") ((++) "." ((++) "." ((++) "." "")))
→ ((++) "." ((++) "." ((++) "." "")))
→ "..."

Recursion on lists : capturing common forms

- Mapping a function across the values of a list :

$$notlist\ [] = []$$
$$notlist\ (x : xs) = ((\sim x) : (notlist\ xs))$$

$$inclist\ [] = []$$
$$inclist\ (x : xs) = ((x + 1) : (inclist\ xs))$$

$$abslist\ [] = []$$
$$abslist\ (x : xs) = ((abs\ x) : (abslist\ xs))$$

Recursion on lists : map

- Built-in function “map" makes life easier :

$$
\begin{aligned}
notlist\ any &= map\ (\sim)\ any \\
inclist\ any &= map\ (+1)\ any \\
abslist\ any &= map\ abs\ any
\end{aligned}
$$

- Or, simply :

$$
\begin{aligned}
notlist &= map\ (\sim) \\
inclist &= map\ (+1) \\
abslist &= map\ abs
\end{aligned}
$$

Definition of map

$$map\ f\ []\ \ \ \ \ \ = []$$
$$map\ f\ (x : xs) = ((f\ x) : (map\ f\ xs))$$

- So, what is the type of map?
- Write it here (don't "cheat" yourself by asking Miranda — work it out for yourself!) :

Recursion on lists : capturing common forms

- Filtering some elements out of a list :

$$
\begin{aligned}
firsts\ [] &= [] \\
firsts\ (x : xs) &= (x : (firsts\ xs)),\ if\ (x >= 70) \\
&= firsts\ xs,\ otherwise
\end{aligned}
$$

$$
\begin{aligned}
not34\ [] &= [] \\
not34\ (x : xs) &= (x : (not34\ xs)),\ if\ (x \sim= 34) \\
&= not34\ xs,\ otherwise
\end{aligned}
$$

Recursion on lists : filter

- Recursion on lists : filter

$$firsts\ any\ =\ filter\ (>=70)\ any$$

$$not34\ any\ =\ filter\ (\sim=34)\ any$$

- Or, simply :

$$firsts\ =\ filter\ (>=70)$$

$$not34\ =\ filter\ (\sim=34)$$

Definition of filter

```
filter p []       = []
filter p (x:xs)   = x : (filter p xs), if (p x)
                  = filter p xs, otherwise
```


- So, what is the type of filter ?
- Write it here (don't "cheat" yourself by asking Miranda — work it out for yourself !) :

Challenge

- Can you write a function that takes a list of numbers (containing only the values 1, 2 and 0, where at least one 0 must occur) and returns a three-tuple containing :
  - The number of 1s before the first 0
  - The number of 2s before the first 0
  - The length of the longest sequence of 1s before the first 0
- Notes :
  - The value of this challenge is NOT in knowing the answer — the value is in the process of finding the answer ! So please don't "cheat" yourself by searching for the answer or looking at somebody else's answer.
  - Start by writing down the type of the function (always !)
  - Be prepared to write small "helper" functions, or look in the online manual (Section 28) for built-in operators.
  - If you find this easy, try designing the program a different way so that it makes use of higher order functions (e.g. filter, dropwhile, takewhile)

Summary

- Combinators
  - Definition
  - Example : S, K, I
- Higher Order Functions
  - Definition
  - Example : composition
- Use of example HOF
- Capturing common forms of recursion on lists
  - Examples : map and filter

END OF LECTURE