Today

- More on recursion
- Pattern matching (intro)

Assignments

- HW-4, R-4 due
- HW-5, R-5 out

Announcements

- Project status report due Thurs
Recursion (Review)

Sum: One way to write a list sum function in Haskell

\[
\text{mySum } \text{xs} = \text{if } \text{null } \text{xs} \\
\quad \text{then } 0 \\
\quad \text{else head } \text{xs } + \text{mySum } \text{tail } \text{xs}
\]

How it works

\[
\text{mySum } [2,4,6] \\
\quad \Rightarrow 2 + \text{mySum } [4,6] \\
\quad \Rightarrow 2 + 4 + \text{mySum } [6] \\
\quad \Rightarrow 2 + 4 + 6 + \text{mySum } [] \\
\quad \Rightarrow 2 + 4 + 6 + 0
\]

Q: What is the type of mySum?

\[
\text{Prelude}\gg \text{:type mySum} \\
\text{mySum } :: (\text{Num } a) \Rightarrow [a] \Rightarrow a
\]

Q: How does Haskell know this?

- by looking at the function definition!
- for example:
  - \text{null } :: [a] \Rightarrow \text{Bool} \hspace{1cm} \ldots \text{xs must be a list}
  - \text{0 } :: \text{Num } a \Rightarrow a \hspace{1cm} \ldots \text{return type is a Num instance}
  - \text{head } :: [a] \Rightarrow a \hspace{1cm} \ldots \text{xs element}
  - \text{(+)} :: (\text{Num } a) \Rightarrow a \Rightarrow a \hspace{1cm} \ldots \text{list elem types a Num instance}
Append: One way to write an append function in Haskell

\[
\text{append } xs \text{ ys} = \begin{cases} 
\text{null } xs & \text{then } ys \\
\text{else } (\text{head } xs) : \text{append } (\text{tail } xs) \text{ ys} 
\end{cases}
\]

How it works:

\[
\text{append } [1,2] [3,4] \\
\quad \Rightarrow 1 : \text{append } [2] [3,4] \\
\quad \Rightarrow 1 : 2 : \text{append } [] [3,4] \\
\quad \Rightarrow 1 : 2 : [3,4] \\
\quad \Rightarrow [1,2,3,4]
\]

Another example:

\[
\text{append } [1,2] [\ldots] \\
\quad \Rightarrow (\text{head } [1,2]) : \text{append } (\text{tail } [1,2]) [\ldots] \\
\quad \Rightarrow (\text{head } [1,2]) : (\text{head } [2]) : \text{append } (\text{tail } [2]) [\ldots] \\
\quad \Rightarrow (\text{head } [1,2]) : (\text{head } [2]) : [\ldots]
\]

- second list never has to (is never asked to) be evaluated
- result is called a “thunk” (“suspension”, “delayed computation”, “future”)

Why does append have the type \([a] \rightarrow [a] \rightarrow [a]\)?

- \text{null} :: [a] \rightarrow \text{Bool} \\
- (:) :: b \rightarrow [b] \rightarrow [b] \\
- \text{then } ys \\
- \text{head } xs :: a

\[
\ldots \text{xs} :: [a] \\
\ldots \text{result must be of type } [b] \\
\ldots \text{ys must of type } [b] \\
\ldots \text{so } a = b
\]
Type Inference and the “Occurs Check” Error

Q: How can we define a flatten function in Haskell?

e.g., flatten \[[[1,2],[3,4]]\] should return \[1,2,3,4\]

    flatten xs = if null xs
        then xs -- this doesn’t work!
        else head xs ++ flatten (tail xs)

Q: What is the type of flatten?

1. \(xs\) must be a list since we call null \(xs\) 
   so: \([?]\) \(\rightarrow\) ?

2. head \(xs\) is a list since we use ++ on it 
   so: \([[a]]\) \(\rightarrow\) ?

3. result must have same type as head \(xs\) 
   so: \([[a]]\) \(\rightarrow\) [a]

4. but result must be same type as \(xs\) (from then \(xs\)) 
   so: [a] \(\equiv\) [[a]]

5. repeating this out gives \([[[[ \cdots a \cdots ]]]]\) 
   infinite type!

Q: How can we fix this?

    flatten :: [[a]] \(\rightarrow\) [a]
    flatten xs = if null xs
        then [] -- this works!
        else head xs ++ flatten (tail xs)

Q: Why does this work?

  • ... we broke the connection between the input type and the output type
Pattern Matching

Functions are defined as a series of equations

- Each equation has a different “pattern” of input

  -- simple myNot definition
  myNot x = if x == True then False else True

- In this case, x has two (value) patterns: True and False

  -- myNot definition w/out if-then-else
  myNot True = False
  myNot False = True

- Here we are defining the function using “pattern matching”

How does this work?

Say we call:

myNot False

The Haskell runtime:

- Checks the value supplied (False) against the first pattern
- In this case, it isn’t a match (False \(\neq\) True)
- The second pattern is checked, which succeeds
- The right-hand side of the second equation is returned

Haskell tries patterns in order ... stops at first match