Today

- Quiz 7
- Occurs Check
- Pattern Matching

Assignments

- R-8, HW-8 due
- R-9, HW-9 out
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]

\[
\text{flatten } xs = \begin{cases} 
  \text{if } \text{null } xs \\
  \text{then } xs \quad \text{-- this doesn’t work!} \\
  \text{else head } xs ++ \text{flatten} \ (\text{tail } xs) 
\end{cases}
\]

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 \([[[[ \ldots \ a \ldots ]]]]\)
   infinite type!

Q: How can we fix this?

\[
\text{flatten } :: [[a]] \rightarrow [a] \\
\text{flatten } xs = \begin{cases} 
  \text{if } \text{null } xs \\
  \text{then } [] \quad \text{-- this works!} \\
  \text{else head } xs ++ \text{flatten} \ (\text{tail } xs) 
\end{cases}
\]

Q: Why does this work?

\(\bullet\) ... 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

  ```haskell
  -- simple (but verbose) myNot definition
  myNot x = if x == True then False else True
  ```

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

  ```haskell
  -- 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:

```haskell
myNot False
```

The Haskell runtime:

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

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

**A more involved example with lists ...**
mix1 xs ys = if null xs || null ys
    then xs ++ ys
    else head xs : head ys : mix1 (tail xs) (tail ys)

Q: What do the following return?

mix1 [] [] ==> []

mix1 [1,3,5] [] ==> [1,3,5]

mix1 [] [2,4,6] ==> [2,4,6]

mix1 [1,3,5] [2,4,6] ==> [1,2,3,4,5,6]

Q: What are the patterns?

- xs empty ... return ys
- ys empty ... return xs
- neither empty ... return else expression

The mix function defined using patterns

mix2 [] ys = ys
mix2 xs [] = xs
mix2 xs ys = head xs : head ys : mix2 (tail xs) (tail ys)

Q: Are these patterns “exhaustive”?

- Yes!
  - e.g., calling mix2 [] [] matches the first case
Even fancier patterns ...

\[
\begin{align*}
  \text{mix3} \ [\ ] \ ys &= ys \\
  \text{mix3} \ xs \ [\ ] &= xs \\
  \text{mix3} \ (x:xs) \ (y:ys) &= x : y : \text{mix3} \ xs \ ys
\end{align*}
\]

- We are “deconstructing” the values in the pattern
- Note the parens around \((x:xs)\) are required
- Using “:” is like calling head and tail on the left-hand side ...
- This is much more succinct and way cooler! ... use in your homework

Example evaluation of mix3

\[
\begin{align*}
  \text{mix3} \ [1,3] \ [2,4] \\
  \Rightarrow \text{mix3} \ (1:[3]) \ (2:[4]) & \quad \text{matches 3rd pattern} \\
  \Rightarrow 1 : 2 : \text{mix3} \ [3] \ [4] \\
  \Rightarrow 1 : 2 : \text{mix3} \ (3:[]) \ (4:[]) & \quad \text{matches 3rd pattern} \\
  \Rightarrow 1 : 2 : 3 : 4 : \text{mix3} \ [] \ [] \\
  \Rightarrow 1 : 2 : 3 : 4 : [\] & \quad \text{matches 1st pattern}
\end{align*}
\]
Another (simpler) example ...

\[ f \; xs = \text{head} \; xs \]

is the same as

\[ f \; (x:xs) = x \]

And:

\[ g \; xs = \text{tail} \; xs \]

is the same as

\[ g \; (x:xs) = xs \]
Wildcards

Use the "wildcard" symbol (_ ) for "don’t care"

For example:

\[
\begin{align*}
  f \ (x_:_) &= x \quad \text{-- don’t care about tail of the list} \\
  g \ (_:xs) &= xs \quad \text{-- don’t care about head of the list} \\
  \text{fst} \ (x, _) &= x \quad \text{-- don’t care about second elem of pair} \\
  \text{snd} \ (_, y) &= y \quad \text{-- don’t care about first elem of pair}
\end{align*}
\]

- _ stands for any value
- corresponding value cannot be accessed on RHS
- helps readability ... focuses attention on the important stuff
- again, use wildcards in your homework!
Another Example: the \textit{init} and \textit{last} functions

Prelude> :type last
last :: [a] -> a

Prelude> last [1,2,3]
3

Prelude> :type init
init :: [a] -> [a]

Prelude> init [1,2,3]
[1,2]

Q: How can we define these using pattern matching?

\begin{verbatim}
last []     = error "empty list"    -- more on error later
last [x]    = x                    -- one element list
last (_:xs) = last xs              -- note pattern order

init []     = error "empty list"
init [x]    = [x]                   -- one element list
init (x:xs) = x : init xs          -- build up list
\end{verbatim}

Order of patterns matters

\begin{itemize}
  \item e.g., [x] and (x:xs) both match a one-element list (e.g., [1])
    \begin{itemize}
    \item the patterns “overlap” (really, (x:xs) subsumes [x])
    \end{itemize}
  \item if you put (x:xs) pattern first, [x] will never be reached
\end{itemize}