Today

- Occurs Check
- Pattern Matching

Assignments

- HW 9 out
- Proj Status Update due Tues
- Exam 2 Thurs
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]

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

Q: What is the type of flatten?

1. \text{xs} must be a list since we call \text{null } \text{xs} so: \([?] \rightarrow ?
\]
2. \text{head } \text{xs} is a list since we use ++ on it so: \([[[a]]] \rightarrow ?
\]
3. result must have same type as \text{head } \text{xs} so: \([[[a]]] \rightarrow [a]
\]
4. but result must be same type as \text{xs} (from then \text{xs}) so: \([a] \equiv [[a]]
\]
5. repeating this out gives \([[[ \cdots a \cdots ]]]
\]
\text{infinite type!}

Q: How can we fix this?

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

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 (but verbose) myNot definition
    myNot x = if x == True then False else True

• In this case, \( x \) has two (value) patterns: \( \text{True} \) and \( \text{False} \)

    -- 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 ... and stops at first match
A more involved example with lists ...

\[
\text{mix1 } xs \; ys = \begin{cases} 
\text{if } \text{null } xs \; || \; \text{null } ys \\
\text{then } xs \; ++ \; ys \\
\text{else } \text{head } xs : \text{head } ys : \text{mix1 } (\text{tail } xs) \; (\text{tail } ys)
\end{cases}
\]

Q: What do the following return?

\[
\begin{align*}
\text{mix1 } [] \; [] &= [] \\
\text{mix1 } [1,3,5] \; [] &= [1,3,5] \\
\text{mix1 } [] \; [2,4,6] &= [2,4,6] \\
\text{mix1 } [1,3,5] \; [2,4,6] &= [1,2,3,4,5,6]
\end{align*}
\]

Q: What are the patterns?

- \(xs\) empty ... return \(ys\)
- \(ys\) empty ... return \(xs\)
- neither empty ... return else expression

The \text{mix} function defined using patterns

\[
\text{mix2 } [] \; ys = ys \\
\text{mix2 } xs \; [] = xs \\
\text{mix2 } xs \; ys = \text{head } xs : \text{head } ys : \text{mix2 } (\text{tail } xs) \; (\text{tail } ys)
\]

Q: Are these patterns “exhaustive”?

- Yes!
- e.g., calling \text{mix2 } [] \; [] matches the first case
Even fancier patterns ...

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

- 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 ...

Example evaluation of \(mix3\)

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

\[ f (x:\_) = x \quad -- \text{don't care about tail of the list} \]
\[ g (\_:xs) = xs \quad -- \text{don't care about head of the list} \]

\[ \text{fst} (x, \_) = x \quad -- \text{don't care about second elem of pair} \]
\[ \text{snd} (\_, y) = y \quad -- \text{don't care about first elem of pair} \]

• \_\_ stands for any value
• corresponding value cannot be accessed on RHS
• helps readability ... focuses attention on the important stuff

Use wildcards in your homework!!!
Another Example: the `init` and `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?

\[
\begin{align*}
\text{last} & \quad [\ ] = \text{error "empty list"} \quad \text{-- undefined for [\ ]} \\
& \quad [x] = x \quad \text{-- one element list} \\
& \quad (_:xs) = \text{last xs} \quad \text{-- note pattern order} \\
\text{init} & \quad [\ ] = \text{error "empty list"} \\
& \quad [\_] = [] \quad \text{-- one element list} \\
& \quad (x:xs) = x : \text{init xs} \quad \text{-- build up list}
\end{align*}
\]

Order of patterns matters

- e.g., `[x]` and `(x:xs)` both match a one-element list (e.g., `[1]`)
  - the patterns “overlap” (really, `(x:xs)` subsumes `[x]`)
- if you put `(x:xs)` pattern first, `[x]` will never be reached
Exercise: Firsts

Q: Use recursion to define a \texttt{firsts} function that takes a list of pairs \texttt{ps} and returns a list containing the first element of each pair. Give the type of \texttt{firsts}.

\begin{verbatim}
firsts :: [(a,b)] -> [a]
firsts [] = []
firsts ((x,_) : ps) = x : firsts ps
\end{verbatim}

Q: Can you define this function using \texttt{map}? How?

\begin{verbatim}
firsts ps = map fst ps
\end{verbatim}

Exercise: Define \texttt{take} using recursion w/ patterns

\begin{verbatim}
take _ [] = []
take n (x:xs) = if n > 0
  then x : take (n-1) xs
  else []
\end{verbatim}

Exercise: Define \texttt{drop} using recursion w/ patterns

\begin{verbatim}
drop _ [] = []
drop n (x:xs) = if n <= 0
  then x : xs
  else drop (n-1) xs
\end{verbatim}
Guards

Patterns specify “structural” conditions for matching

- Matching on the parts of a data structure

Guards allow us to define **conditions** for a pattern

```haskell
-- previous myDrop function with just patterns
drop _ [] = []
drop n (x:xs) = if n <= 0
  then x : xs
  else drop (n-1) xs
```

- We can rewrite this using guards to remove the if-then-else:

```haskell
drop _ [] = []
drop n xs | n <= 0 = xs
drop n (_:xs) = drop (n-1) xs
```

- The guard gives a condition for applying the pattern
There can be *multiple* guards per pattern

\[
\text{letterGrade } p \\
| p \geq 90 &= "A" \\
| p \geq 80 &= "B" \\
| p \geq 70 &= "C" \\
| p \geq 60 &= "D" \\
| \text{otherwise} &= "F"
\]

- Each guard is an expression of type `Bool`
- `otherwise` is a special variable bound to `True`

**How a guard works**
- For each pattern, check if first guard succeeds
- If so, RHS is result
- Otherwise, check next guard
- If no guards succeed, go to the next pattern

**When calling a function, if no patterns match ...**
- Haskell gives a runtime exception (non-exhaustive pattern)
Another (contrived) example

Q: What does this function do?

\[
\begin{align*}
\text{pairs } [] &= [] \\
\text{pairs } [\_] &= [] \\
\text{pairs } (x:y:zs) &= \begin{cases} 
(x,y) : \text{pairs } (y:zs) \\
\text{otherwise} & \text{pairs } (y:zs) 
\end{cases}
\end{align*}
\]

- Note: patterns and guards can be mixed (as above)
- Also: names in patterns can only appear once!
  - e.g., \(x:x:zs\) wouldn’t work in last pattern

Q: What is the result of \(\text{pairs } [1,2,2,2,3]\)?

\([(2,2),(2,2)]\)

Q: What is the type of \(\text{pairs}\)?

\(\text{pairs} :: (\text{Eq } a) \Rightarrow [a] \rightarrow [(a, a)]\)

Note on where with guards vs. let

\[
\begin{align*}
f \ x \\
| \ g1 &= \text{e1} \\
| \ g2 &= \text{e2} \\
\text{where} \ ... \\
\end{align*}
\]

\[
\begin{align*}
f \ x \\
| \ g1 &= \text{let} \ ... \ \text{in } \text{e1} \\
| \ g2 &= \text{let} \ ... \ \text{in } \text{e2}
\end{align*}
\]