Haskell, Part 4

CS 440: Programming Languages and Translators, Spring 2019
Mon 2019-01-28

More Haskell

- Review definition by cases
- Chapter 6: Higher-order functions
  - Revisit currying
  - map, filter
  - Unnamed ("lambda") functions; function definitions as \texttt{function\_name = lambda function}.
  - folding a list
- From Chapter 7: Making our own Types....
  - Algebraic datatypes - enumerations, simple recursive structures, constructor patterns

Review: Last time, Definition by cases

- List clauses of function definition; order of patterns is important: don't put \texttt{vl _ = ...} first).

\begin{verbatim}
Prelude> :{
Prelude| vl []   = "empty"
Prelude| vl [ _ ] = "singleton" -- vl [x] = "singleton" ok too
Prelude| -- Can build lists using :,
Prelude| vl [ _, _ ] = "short"
Prelude| vl _ = "long"
Prelude| :}
Prelude> :t vl
vl :: [a] -> [Char]
Prelude> vl []
"empty"
Prelude> vl [1,2]
"short"
Prelude> vl [1,3,5,2]
"long"
\end{verbatim}

- Can have boolean guards in function definitions (boolean tests)

\begin{verbatim}
Prelude> :{
Prelude| fact n | n <= 0 = 1
Prelude|      | n > 0  = n * fact(n-1)
Prelude| -- order doesn't matter with non-overlapping tests
Prelude| :}
\end{verbatim}
Prelude> fact 5
120

• Local variables in function definitions
  • A where clause lets you calculate a local value and use it in the function

Prelude> :
  Prelude| vl x | len == 0 = "empty"
  Prelude|     | len == 1 = "short"
  Prelude|     | len > 1 = "long"
  Prelude| where len = length x
  Prelude| :
Prelude> vl []
"empty"

• Let expressions are also okay for defining local variables

Prelude> :
  Prelude| vl x = let len = length x in
  Prelude|           if len == 0 then "empty"
  Prelude|           else if len == 1 then "short"
  Prelude|           else "long"
  Prelude| :
Prelude> vl [1,3,5]
"long"

• Case expressions let you take an expression and match it against various patterns

Prelude> :
  Prelude| vl x = case length x of 0 -> "empty"
  Prelude|           1 -> "short"
  Prelude|           _ -> "long"
  Prelude| :
Prelude> vl "string"
"long"

Skip Ch 5 (recursion)

Ch 6 Higher order functions
• Recall: A higher order function is a function that takes another function as a parameter or returns a function as a result.
  • Haskell uses curried functions: Instead of \( f : \text{Int} \times \text{Int} \times \text{Int} \rightarrow \text{Int} \) as in most languages in Haskell, we would have \( f : : : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \).
• Arrow is right-associative, so \( f :: \text{Int} \to (\text{Int} \to (\text{Int} \to \text{Int})) \) is equivalent.

• So \( f \ 17 \) is a function :: that you can use like any other function.

• (It's a partially-applied version of \( f \).)

```haskell
Prelude> :{
Prelude| f :: Int -> Int -> Int -> Int
Prelude| f a b c = a * b + c
Prelude| :}
Prelude> :t f
f :: Int -> Int -> Int -> Int
Prelude> f 3 5 8
23
Prelude> g = f 3
Prelude> :t g
g :: Int -> Int -> Int
Prelude> h = g 5
Prelude> :t h
h :: Int -> Int
Prelude> h 8
23
```

• Recall we needed to give \( g \ 5 \) a name so that we wouldn't try to print its value (causes an error).

```haskell
Prelude> g 5 -- can't print a function value
<interactive>:74:1: error:
  · No instance for (Show (Int -> Int)) arising from a use of ‘print’
    (maybe you haven't applied a function to enough arguments?)
  · In a stmt of an interactive GHCi command: print it
```

• When you say variable = expression, it gives the variable a value but it doesn't print the value out.

```haskell
Prelude> z = 5
Prelude> z -- evaluate z so I can see the value
5
Prelude> h = g 5
Prelude> -- now we have an h; if I ask for it's value, I get an error
Prelude> h
<interactive>:80:1: error:
  · No instance for (Show (Int -> Int)) arising from a use of ‘print’
    (maybe you haven't applied a function to enough arguments?)
  · In a stmt of an interactive GHCi command: print it
```
More functions on lists

Recall map :: (a -> b) -> [a] -> [b] a function on a values, a list of a values, returns the list of results

Prelude> map sqrt [2..5]
[1.4142135623730951,1.7320508075688772,2.0,2.23606797749979]

Similar is filter, which takes a boolean test function (a -> Bool) and an a list and returns the values of the list that pass the test:

Prelude> positive x = if x > 0 then True else False
Prelude> positive x = x > 0 -- is equivalent
Prelude> map positive [3, 5, -1, 2, -9, 7, -2, -3]
[True,True,False,True,False,True,False,False]
Prelude> filter positive [3, 5, -1, 2, -9, 7, -2, -3]
[3,5,2,7]

Another example: find values divisible by 3

Prelude> divisible_by_3 x = mod x 3 == 0 -- using mod as a prefix function
Prelude> divisible_by_3 x = x `mod` 3 == 0 -- equivalent defn, using mod as a binary operator
Prelude> divisible_by_3 6
True
Prelude> filter divisible_by_3 [27..83]
[27,30,33,36,39,42,45,48,51,54,57,60,63,66,69,72,75,78,81]

Filter and find last value

Prelude> largest_multiple_of_3 x = last (filter divisible_by_3 x)
Prelude> largest_multiple_of_3 [27..83]
81

Lambda functions

It can be annoying to write a function like largest_multiple_of_3 just to use it in one spot

We can use unnamed functions instead. Below, \ x -> x `mod` 3 == 0 is a function that takes an x and returns true if x mod 3 is zero.

Prelude> -- divisible_by_3 x = x `mod` 3 == 0
Prelude> -- divisible_by_3 = \ x -> x `mod` 3 == 0 -- same as previous line
Prelude> divisible_by_3 6
True

Backslash is Haskell's representation of greek lowercase lambda.

The actual lambda notation would be \ x . x `mod` 3 == 0.

In fact, function declarations like f x = exp is short for f = \x -> exp
• This makes function definitions like other variable definitions like \( y = 3 \) (variable = value). Warning: If \( f \) is recursive then \( f = \lambda x \rightarrow \text{recursive expression} \) doesn't work in the regular lambda calculus (works okay here)\(^1\).

\begin{verbatim}
Prelude> f a b c = a * b + c Prelude> f = \a -> \b -> \c -> a * b + c -- the function that takes a and returns a function that takes b.... Prelude> f 3 5 8 23
\end{verbatim}

• Lambdas are useful for short things you use once.

\begin{verbatim}
Prelude> filter positive [3, 5, -1, 2, -9, 7, -2, -3] [3,5,2,7] Prelude> filter (\x -> x > 0) [3, 5, -1, 2, -9, 7, -2, -3] [3,5,2,7] Prelude> positive x = x > 0 -- usual definition syntax Prelude> positive = \x -> x > 0 -- using lambda
\end{verbatim}

### Folding lists

• Folding a list lets you combine its elements using some operation, like adding together a list of numbers.

\begin{verbatim}
Prelude> foldl (-) 0 [3,5,8] -16 Prelude> foldl (-) 0 [3,5,8] -16
\end{verbatim}

• For \texttt{foldl}, the starting value goes on the left, and the operations are done left-to-right.

\begin{verbatim}
Prelude> (((0 - 3) - 5) - 8) -16 Prelude> foldr (-) 0 [3,5,8] 6 Prelude> (3 - (5 -(8 - 0))) 6
\end{verbatim}

• The types for \texttt{foldl} and \texttt{foldr} let you fold more than just lists.

• The classtype \texttt{Foldable} is for types you can fold. Lists are built-in \texttt{Foldable}.

• If you define a datatype (see in a bit), you might be able to fold its values too.

\begin{verbatim}
The type for \texttt{foldl} and \texttt{foldr} says \texttt{Foldable t => .... t a.}
Foldable "types" are actually type constructors (take a type and build another type).
List of ... is a built-in type constructor.
\end{verbatim}

\(^1\) Forgot this proviso in lecture -- we'll see later that Haskell's laziness lets \( f = \lambda x \rightarrow \ldots \) use of \( f \ldots \) work recursively.
Prelude> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
Prelude> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b

• In \foldl\ (-) 0 [3, 5, 8], we use Int for the type variables a and b in (b -> a -> b) -> b -> t a -> b, and we use [...] (list-of) as t.
• So our use of \foldl\ had type (Int -> Int -> Int) -> Int -> [Int] -> Int.
• (We take three arguments, one after another: A curried binary function on integers, an integer, and a list of integers, and we return an integer.
  • The plain integer argument is the starting point for the folding (it's what gets returned if you fold []).

Since + is associative, \foldl\ and \foldr\ will return the same result when given the same arguments: foldl (+) 0 [3,5,8] and foldr (+) 0 [3,5,8] both return 16.
  • In lecture I said "since + is associative and commutative" but it's only associativity that matters for flipping foldl / foldr. Commutativity ensures that reordering the elements doesn't change anything: foldl (+) 0 [3, 5, 8] = foldr (+) 0, [5, 8, 3] = 16.
• Since - is not associative, foldl and foldr can return different values. (We saw this above.)
  • And since - is not commutative, reordering the elements can change the result: foldl (-) 0 [1, 2] ≠ foldl (-) 0 [2, 1].

The \Learn You ... book has an example that uses folding to define our own elem function.
  • (elem x y is true if x is a member of list y)
  • With elem' y ys, the foldl function is used to search the ys for a y.
    • For y and ys, we can use any type that supports equality testing (e.g. Int).
    • However, the accumulated result is a boolean (it answers the question "Have we found y yet?")
Prelude> elem' y ys = foldl (\acc x -> if x == y then True else acc) False ys
Prelude> elem' 0 []
False
Prelude> elem' 3 [1,2,3,4]
True
  • If y is 3 then the lambda in elem' becomes \acc x -> if x == 3 then True else acc.
    • Let's give that function a name.
Prelude> f acc x = if x == 3 then True else acc -- always looking for a 3.
Folding left uses f to look (from left-to-right) for a 3 in the list.
Prelude> False `f` 1
False
Prelude> (False `f` 1) `f` 2
False
Prelude> ((False `f` 1) `f` 2) `f` 3
True
Prelude> (((False `f` 1) `f` 2) `f` 3) `f` 4
True
-------------------------------
• Omit flawed discussion of folding using function composition :- ( 
-------------------------------

**Algebraic types**

• We can define our own types like trees, etc.
  • Simple example: Define our own enumerations
    Prelude> data Color = Red | Blue
    Prelude> :info Color
data Color = Red | Blue        -- Defined at <interactive>:224:1
    Prelude> :info Red
data Color = Red | ...  -- Defined at <interactive>:224:14

  • Define a function on Colors:
    Prelude> :{
    Prelude| opposite Red = Blue
    Prelude| opposite Blue = Red
    Prelude| :}

  • Oops -- forgot that datatypes create values that aren't automatically printable.
    Prelude> opposite Red

<interactive>:231:1: error:
  • No instance for (Show Color) arising from a use of 'print'
  • In a stmt of an interactive GHCi command: print it

  • Oops again -- they aren't automatically testable for equality
    Prelude> opposite Red == Blue

<interactive>:232:1: error:
  • No instance for (Eq Color) arising from a use of '=='
  • In the expression: opposite Red == Blue
    In an equation for 'it': it = opposite Red == Blue

  • But Haskell has a very nice feature
The deriving clauses ask Haskell to figure out how to test for equality and print values of type `Color`.

```haskell
Prelude> data Color = Red | Blue deriving (Eq, Show)
Prelude> Red -- Now a color value can be printed
Red

Prelude> #:{}
Prelude| opp Red = Blue
Prelude| opp Blue = Red
Prelude| :}
Prelude> opp Red
Blue
Prelude> opp Red == Blue -- can test for equality
True
```

Here's a simple tree type, with the Haskell-supplied `==` and `show`.

- (Trees where the leafs are labeled by integers.)

```haskell
Prelude> data IntTree = Leaf Int | Node IntTree IntTree deriving (Eq, Show)
Prelude> Leaf 0 == Leaf 0
True
Prelude> Leaf 0 == Leaf 1
False
Prelude> Node (Leaf 0) (Leaf 1) -- create a tree value; it gets printed
Node (Leaf 0) (Leaf 1)
Prelude> t1 = Node (Leaf 0) (Leaf 1)
Prelude> t2 = Node t1 t1
Prelude> t2 -- a tree with t1 and t2 as its left and right subtrees
Node (Node (Leaf 0) (Leaf 1)) (Node (Leaf 0) (Leaf 1))
```

**Datatype constructor patterns**

- You can define functions on `IntTrees` using `Leaf` and `Node` (similarly to how you can use `:` and `[]` on lists)

```haskell
Prelude> #:{}
Prelude| sumTree :: IntTree -> Int
Prelude| sumTree (Leaf x) = x
Prelude| sumTree (Node sub1 sub2) = sumTree sub1 + sumTree sub2
Prelude| :}
Prelude> sumTree (Leaf 0)
0
Prelude> t1
Node (Leaf 0) (Leaf 1)
Prelude> sumTree t1
```

v. Fri 2/1, 21:31
Prelude> t2
Node (Node (Leaf 0) (Leaf 1)) (Node (Leaf 0) (Leaf 1))
Prelude> sumTree t2
2
Prelude>