Haskell, Part 2

CS 440: Programming Languages and Translators, Spring 2019
Mon 2019-01-14

More Haskell

• More on lists, lists of lists, list functions including take, drop, elem and !! (get n'th element)
• Defining variables at top level
• Backquotes and parentheses for making functions infix or prefix
• Infinite lists
• :t expr to get the type of an expression.
• Use of Num and Fractional to discuss general collections of numbers.

Sample Interactive Run of Haskell

unix > ghci
CS 440, S'19, Lecture 01, Mon 2019-01-14

unix > ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/ :? for help

Continue on with lists

Prelude> [] -- empty list
[]
Prelude> [1] -- singleton list
[1]
Prelude> [1, 'a'] -- need the elements of the list to all have the same type

<interactive>:3:2: error:
  • No instance for (Num Char) arising from the literal '1'
    • In the expression: 1
      In the expression: [1, 'a']
      In an equation for ‘it’: it = [1, 'a']

Prelude> [ [2], [3,5] ] -- okay to have list containing sublists of different lengths
[[2],[3,5]]

* These are "typeclasses" [unrelated to OO classes]. We'll study them more later.
Various List Functions

- Is list null? (== [])?
- length of list
- reverse a list
- minimum and maximum elements of list
- sum and product of list values
- take and drop initial segment of a list

Prelude> length [3,5,6,7,8]
5
Prelude> null [3,5,6,7,8]
False
Prelude> null []
True
Prelude> reverse [1,2,3,5,7]
[7,5,3,2,1]
Prelude> reverse (reverse [1,2,3,5,7])
[1,2,3,5,7]
Prelude> minimum [1,2,3,5,7]
1
Prelude> maximum [1,2,3,5,7]
7
Prelude> sum [1,2,3,5,7]
18
Prelude> product [1,2,3,5,7]
210

-- take n x returns list of first n elements of list x
-- drop n x returns what remains after omitting first n elements of list x

Prelude> take 3 [1,2,3,5,7]
[1,2,3]
Prelude> take 4 [1,2,3,5,7]
[1,2,3,5]
Prelude> take 0 [1,2,3,5,7]
[]
Prelude> take (-1) [1,2,3,5,7]
[]
Prelude> take 4 []
[]
Prelude> drop 3 [1,2,3,5,7] -- list without first three items
[5,7]
Prelude> drop 8 [1,2,3,5,7]
[]

**Defining interactive variable**
- At top level, we can define a variable good to the end of the session (unless you redefine it.)
- Note defining x doesn’t print out its value. To get the value, use x as an expression

Prelude> x = [1,2,3,5,7] -- Note no "let" or "in"
Prelude>
Prelude> x  -- use x as an expression to print its value
[1,2,3,5,7]

**More functions: !!, elem**
- Use list !! n to return n’th element of a list

Prelude> x !! 0 -- return first element
1
Prelude> x !! 1
2
Prelude> x !! 8
*** Exception: Prelude.!!: index too large

- Is a value an element of a list?

Prelude> elem 2 [1,2,3,5,7] -- is 2 in the list?
True
Prelude> elem 8 [1,2,3,5,7] -- but not 8
False

**Prefix to Infix; Infix to Prefix**
- We can take a binary function id and use it as an infix operator by surrounding with back quotes

Prelude> 2 `elem` [1,2,3,5,7]
True

- If you want to use a binary operator as a value, surround it with parentheses

Prelude> 3 : x -- usual notation: infix operator
[3,1,2,3,5,7]
Prelude> (+) 3 x -- as a prefix function
[3,1,2,3,5,7]
Fancy: List of functions; can't print out functions

- Let's try something fancy, a list of arithmetic functions

Prelude> fl = [(+), (-), (*), div, rem, (/)] -- Give it the name fl

Prelude> (fl !! 0) 3 5 -- fl !! 0 is the (+) function
Prelude> [fcn 10 4 | fcn <- fl] -- do (+) 10 4, (-) 10 4, etc
[14,6,40,2,2]
Prelude> [(+) 10 4, (-) 10 4, (*) 10 4, div 10 4, rem 10 4]
[14,6,40,2,2]

-- You can't print out functions
-- Calculating fl !! 0 gives you (+), but you can't print it.
Prelude> fl !! 0

<interactive>:35:1: error:
  • No instance for (Show (Integer -> Integer -> Integer))
    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

Infinite Lists

- Infinite lists can be created repeat and cycle. It's ok to calculate them but don't try to print them.

Prelude> r = repeat 17 -- r = [17, 17, 17, etc]

-- (Just using Prelude> r here would try to print an infinite list)

Prelude> take 20 r -- first 20 elements of r
[17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17]
Prelude> s = drop 20 r
Prelude> take 10 s
[17,17,17,17,17,17,17,17,17,17]
Prelude> s = cycle [1,3,5,6] -- [1,3,5,6, 1,3,5,6, 1,3,5,6, etc]
Prelude> take 20 s
[1,3,5,6,1,3,5,6,1,3,5,6,1,3,5,6,1,3,5,6]
Prelude> sum (take 20 s) -- add up first 20 elements of s = 5 occurrences of 1,3,5,6
75
Ask ghci for the Type of an Expression

-- At top level, :t expr will print the type of the expression
-- s is a list of numbers, but Haskell is vague as to what kind of number (Int, Float, ... ?)
Prelude> :t s
s :: Num a => [a]

Some Kinds of Numbers

-- 3.5 * 4.2 is a fractional number (Float, Double?)
-- Fractionals are numbers that support / (division)
Prelude> :t (3.5 * 4.2)
(3.5 * 4.2) :: Fractional a => a

-- Binary operations like + aren't of type a x a -> a
-- where a is a number type. + is of type a -> a -> a.
-- We'll see the difference in a bit.
Prelude> :t (+)
(+) :: Num a => a -> a -> a

-- Division is a binary operation on fractional numbers
Prelude> :t (/)
(/) :: Fractional a => a -> a -> a

-- [(+), (-), (*), (/)] is a list of binary operations on numbers
-- (Postpone discussing Integral and Fractional parts that specify what kind of number)

-- A list of binary operations on numbers
Prelude> :t [(+), (-)]
[(+), (-)] :: Num a => [a -> a -> a]

Pause here for activity questions (moved to end of document)

Tuples

- Tuples are similar to lists but have fixed length. They aren't the same as lists.
- In Haskell, the pair type of a and b is written (a, b), not a × b

Prelude> (1,2)
(1,2)
Prelude> :t (1,2)
(1,2) :: (Num a, Num b) => (a, b)
Prelude> [1,2] == (1,2)

<interactive>:24:10: error:
  * Couldn't match expected type '[Integer]'
    with actual type '(Integer, Integer)'
  * In the second argument of '==', namely '(1, 2)'
    In an equation for 'it': it = [1, 2] == (1, 2)
Prelude> fst (1,2) -- first element of pair
1
Prelude> snd (1,2) -- second element of pair
2

• In a typical language, a function like + is of type Int × Int → Int. I.e., it takes a pair of values and returns a value. In Haskell, this type is written (Int, Int) → Int, but Haskell actually uses a different type.

• In Haskell, the type of (+) is a → a → a (where a is a Number type).

• A function like + is a one-place function that takes the left operand and returns a function. This function is a takes the right operand of the + and returns the sum. So + takes its arguments one after another.

• Functions like + that take their arguments one after another are said to be "curried".

• The name has nothing to do with spices, it comes from Haskell Curry, the mathematician / logician / CS person for whom the language Haskell is named.

Prelude> div 23 6 -- Apply div to 23, then apply the result to 6
2
Prelude> ((+) 5) 7 -- Apply + to 5, then apply the result to 5
12

Prelude> :t (+)
(+ :: Num a => a -> a -> a)

• The binary functions we're familiar with are uncurried; they take their two arguments at the same time. The function g below is the uncurried version of +.

Prelude> g (x,y) = x + y -- add two elements of a pair
Prelude> :t g
g :: Num a => (a, a) -> a
In Haskell, we pretty much always use curried functions. There are `curry` and `uncurry` functions that convert from one to the other.

- `curry g` takes its arguments one after another: `(curry g) 2 4 = 2 + 4 = 6.`
- `uncurry (+)` takes it arguments at the same time: `(uncurry (+))(2, 4) = 2 + 4 = 6.

```haskell
Prelude> g (x,y) = x + y  -- Uncurried addition
Prelude> h x y = x + y    -- Curried addition (just like +)
Prelude> :t g
Prelude> :t h
Prelude> :t (curry g)    -- Same type as +
(curry g) :: Num c => c -> c -> c
Prelude> :t (uncurry (+)) -- Same type as g
(uncurry (+)) :: Num c => (c, c) -> c

-- Call g on pair of arguments, call h one arg after another
Prelude> g(1,6) 7
Prelude> h 1 6 7

-- The argument to g doesn't have to be a literal pair; it can be calculated
Prelude> z = head [(1,6),(2,4)]
Prelude> g z 7

More List functions
- `zip` takes two lists and returns a list of pairs: a pair with the first elements of the two lists, a pair with the second elements, etc.

```haskell
Prelude> z = zip [1,2,3] ['a','b','c']
Prelude> head z
(1,'a')
Prelude> z = zip [1,2,3] [6,7,9]
Prelude> z
[(1,6),(2,7),(3,9)]
```

- `map` applies a function to every element of a list

```haskell
Prelude> map sqrt [1..5]
[1.0,1.4142135623730951,1.7320508075688772,2.0,2.23606797749979]
Prelude> k x = 8 + x      -- a function on one argument
Prelude> map k [1..5]    -- add 8 everywhere
[9,10,11,12,13]
Prelude> map ((+) 8) [1..5]  -- k is same as function result of (+) 8
[9,10,11,12,13]
```
Lecture 2 Activity Questions

1. What is type of value is \[(+, (-), (*), \text{div, rem, (/)}]\?
   
   (Added after class -- ignore the Integral and Fractional parts of the answer)

Write definitions for three list-handling functions

2. \texttt{firsthalf} \(x\) = first half of list \(x\). E.g., \texttt{firsthalf} \([1,2,3,4]\) = \([1,2]\)
   
   If the list is of odd length, ignore the middle element: \texttt{firsthalf} \([1,2,3,4,5]\) = \([1,2]\)
   
   Use \texttt{length} \(x\), division by 2, and \texttt{take}. \texttt{div y 2} is division by 2 dropping remainder

3. \texttt{secondhalf} \(x\) = second half of list \(x\). E.g., \texttt{secondhalf} \([1,2,3,4]\) = \([3,4]\)
   
   If the list is of odd length, ignore the middle element: \texttt{secondhalf} \([1,2,3,4,5]\) = \([4,5]\)
   
   \texttt{rem y 2} is remainder of integer division.
   
   Possible hint: \(z + \texttt{rem y 2}\) is \(z+0\) or \(z+1\) depending on \(y\) being even or odd

4. \texttt{pal} \(x\) = Is \(x\) a palindrome? (does first half equal reverse of second half?)
Activity Solution

1. $[(+, \text{ (-)}, (\text{ *}), \text{ div}, \text{ rem}, (/))]$ is of type $(\text{ Integral } a, \text{ Fractional } a) \Rightarrow [a \rightarrow a \rightarrow a]$. I.e., a list of binary operations on certain kinds of numbers

2. \texttt{firsthalf} \( x = \text{ take} (\text{ div} (\text{ length} x) 2) x \)
\hspace{1cm} Take first \( N \) items of \( x \) where \( N \) is \( \text{ div} (\text{ length} x) 2 \), i.e., half the length of 2 ignoring remainder from division of 2.

3. For \texttt{secondhalf}, let’s look at an initial buggy version then fix it.
\hspace{1cm} \text{ (Buggy): secondhalf} \( x = \text{ drop} (\text{ div} (\text{ length} x) 2) x \). This works okay for even-length lists but it includes the middle element for odd-length lists. I.e., \texttt{secondhalf} \([1,2,3,4,5] = [3,4,5]\).
\hspace{1cm} If \( N = \text{ div} (\text{ length} x) 2 \), then for even-length \( x \), we want to drop the first \( N \) elements. For odd-length \( x \), we want to drop the first \( N+1 \) elements. Turns out we can drop the first \( N + \text{ rem} (\text{ length} x) 2 \) elements because the remainder is 0 for even lengths and 1 for odd lengths. So the correct solution is
\hspace{1cm} \text{ secondhalf} \( x = \text{ drop} (\text{ div} (\text{ length} x) 2 + \text{ rem} (\text{ length} x) 2) x \)

4. The palindrome testing is straightforward. We need the parentheses below because function application is left associative.
\hspace{1cm} \texttt{pal} \( x = \text{ firsthalf} x == \text{ reverse} (\text{ secondhalf} x) \)

Some tests - you can verify that

- \texttt{secondhalf} \([1,2,3,4,5,6] = [4,5,6]\)
- \texttt{buggy secondhalf} \([1,2,3,4,5,6,7] = [4,5,6,7]\) (bug: includes middle element 4)
- \texttt{fixed secondhalf} \([1,2,3,4,5,6,7] = [5,6,7]\) (omits middle element)
- Return true: \texttt{pal} [] , \texttt{pal} [1] , \texttt{pal} [1,1] , \texttt{pal} [1,2,1] , \texttt{pal} [1,2,3,2,1]
- Also true (since strings are lists of characters): \texttt{pal "", pal "1", pal "11", pal "121"}
- Return false: \texttt{pal "12", pal "1312"}

\( \dagger (\text{ Integral } \text{ numbers support div and rem, Fractionals support } /) \)