Haskell, Part 3
CS 440: Programming Languages and Translators, Spring 2019
Wed 2019-01-23

More Haskell

• More lists: map and zip
• Types and typeclasses
  • Primitive types, constructed types, type variables, parametric polymorphism
  • Typeclasses (aren't classes). Instances of typeclasses.
  • Built-in typeclasses Num, Fractional, Eq, Show, Read, Integral.

List functions map and zip

GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Prelude> map sqrt [2.3, 4.5, 7.2]
[1.51657508881031, 2.1213203435596424, 2.6832815729997477]
Prelude> [sqrt 2.3, sqrt 4.5, sqrt 7.2]
[1.51657508881031, 2.1213203435596424, 2.6832815729997477]
Prelude> zip [1,2,3,4] "wxyzabc"
[(1,'w'),(2,'x'),(3,'y'),(4,'z')]

Chapter 3 Types and typeclasses

• We've seen primitive types like Int, Float, Double, Char, Bool plus type constructors: list of..., tuple of ...
  and function types (type_of_arg -> type_of_result)

Prelude> :t sqrt
sqrt :: Floating a => a -> a

• It's common to specify the type of user-defined function (the typechecker will verify it for you).

Prelude> :{
Prelude| f :: Int -> (Int, [Int])
Prelude| f x = (x+1, [x, x-1, x+5])
Prelude| :}
Prelude> :t f
f :: Int -> (Int, [Int])
Prelude> :{
Prelude| g :: Int -> Int -- error: result is pair of Int
Prelude| g x = (x, x)
Prelude| :}
Haskell is pretty good at inferring the type if you don't specify one:

Prelude> h x = [ x+3 ]
Prelude> :t h
h :: Num a => a -> [a]

Type variable: In Num a => a -> [a], a is a type variable: h can have any type you get by replacing a with a numeric type, so if you give it a Fractional argument, it returns a list of Fractional result.

Prelude> :t h 3.5
h 3.5 :: Fractional a => [a]

This kind of polymorphism (with type variables) is parametric polymorphism.

The Haskell typechecker tries to infer the most general type it can for an expression

  Here, whatever type of value you pass to j, it returns a list of those kind of values.

Prelude> j x = [x, x, x]
Prelude> :t j
j :: a -> [a]

You can explicit declare the type a function result. Here we're restricting the return type of k to be [Float], so the argument must be a Float.

Prelude> k y = j y :: [Float]
Prelude> :t j
j :: a -> [a]
Prelude> :t k
k :: Float -> [Float]

**Num and Fractional are Typeclasses**

- Typeclasses are a generalization of overloading. Instead of using + to mean different functions depending on the context, in Haskell, there's only one + operator, but it can work on any value of a specified set of types, Num.

- So + can take any argument that is in typeclass Num (technical term is "is an instance of Num").

  - "Class" in "typeclass" is unrelated to class in the OO sense.

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

- There are a bunch of various typeclasses built into Haskell.

  - More-specific versions of Num include Floating, Fractional
Fractional types support division. Floating types support sqrt.

Prelude> :t (/)
(/) :: Fractional a => a -> a -> a

Prelude> :t sqrt
sqrt :: Floating a => a -> a

In ghci, the command :info name of typeclass tells you what functions the typeclass supports (below, +, -, ...., fromInteger).

Prelude> :info Num

class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a

  -- (Next, if you want to create your own type and make it a Num, you have to include at least the specified minimal set of operations. In Num, you can specify either negate or subtraction; the other operation is defined automatically.)
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}

  -- Defined in ‘GHC.Num’

  -- (Next, we see the built-in types Word, Integer, etc. support Num — they’re said to be instances of Num.)

instance Num Word -- Defined in ‘GHC.Num’
instance Num Integer -- Defined in ‘GHC.Num’
instance Num Int -- Defined in ‘GHC.Num’
instance Num Float -- Defined in ‘GHC.Float’
instance Num Double -- Defined in ‘GHC.Float’

You can also ask for info on a built-in function / operator.

Prelude> :info (+)

class Num a where
  (+) :: a -> a -> a

  -- Defined in ‘GHC.Num’
  infixl 6 +

Eq typeclass

Another useful built-in typeclass is Eq — it’s for types that support == and /= for equality testing. (Note: It's /=, not !=.) You have to define either == or /= if you want a newly created type to be an instance of Num.

Prelude> :info Eq
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    {-# MINIMAL (==) | (/=) #-}
    -- Defined in 'GHC.Classes'
instance Eq a => Eq [a] -- Defined in 'GHC.Classes'
    (If a is an equality type, then so list "list of a".)
instance Eq Word -- Defined in 'GHC.Classes'
instance Eq Ordering -- Defined in 'GHC.Classes'
instance Eq Int -- Defined in 'GHC.Classes'
instance Eq Float -- Defined in 'GHC.Classes'
instance Eq Double -- Defined in 'GHC.Classes'
instance Eq Char -- Defined in 'GHC.Classes'
instance Eq Bool -- Defined in 'GHC.Classes'
    (The next line says that if you have 15 equality types, then their 15-tuple is also an equality type.)
instance (Eq a, Eq b, Eq c, Eq d, Eq e, Eq f, Eq g, Eq h, Eq i, Eq j, Eq k, Eq l, Eq m, Eq n, Eq o) =>
    Eq (a, b, c, d, e, f, g, h, i, j, k, l, m, n, o)
    -- Defined in 'GHC.Classes'
    (I'm omitting a bunch of lines that declare similarly that 14-, 13-, ...., 3-tuples are equality types.)
instance (Eq a, Eq b) => Eq (a, b) -- Defined in 'GHC.Classes'
    (We've seen a lot of Eq types; I'm omitting the declarations of the rest of them)

Show typeclass

• I think in lecture I said Show supports printing, but it's a little more subtle than that. Show supports a show
  function that takes a value of your type and returns a string representation of it. The actual print functions are
  somewhere else.)
Prelude> :info Show
class Show a where
    showsPrec :: Int -> a -> ShowS
    show :: a -> String
    showList :: [a] -> ShowS
    {-# MINIMAL showsPrec | show #-}
    -- Defined in 'GHC.Show'
    (Some omissions ...., then we see that the low-level built-in types are all Showable.)
instance Show a => Show [a] -- Defined in 'GHC.Show'
    (If you can show some type, you can show a list of that type.)
instance Show Word -- Defined in 'GHC.Show'
instance Show GHC.Types.RuntimeRep -- Defined in 'GHC.Show'
instance Show Ordering -- Defined in ‘GHC.Show’
instance Show a => Show (Maybe a) -- Defined in ‘GHC.Show’
instance Show Integer -- Defined in ‘GHC.Show’
instance Show Int -- Defined in ‘GHC.Show’
instance Show Char -- Defined in ‘GHC.Show’
instance Show Bool -- Defined in ‘GHC.Show’

(.... more omissions, concerning 2-... through 15-tuples of showable types being showable...)
instance Show Float -- Defined in ‘GHC.Float’
instance Show Double -- Defined in ‘GHC.Float’

• So what’s not showable? Functions, primarily. That’s why you can’t ask for the value of a function like sqrt; it
  has a value, you just can’t print it.
Prelude> sqrt -- functions are not showable

Functions aren’t showable or == testable

<interactive>:67:1: error:
  • No instance for (Show (Double -> Double))
    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
Prelude> :t sqrt
sqrt :: Floating a => a -> a
    (You can skip the next two lines if you want. If you’re curious, the "+d" on a :t type request ask Haskell
     to use some default concrete types instead of the general type variables.)
Prelude> :t +d sqrt
sqrt :: Double -> Double

• Function types are not in Eq either (in addition to not in Show).
Prelude> sqrt == sqrt

<interactive>:71:1: error:
  • No instance for (Eq (Double -> Double))
    arising from a use of ‘==’
    (maybe you haven’t applied a function to enough arguments?)
  • In the expression: sqrt == sqrt
    In an equation for ‘it’: it = sqrt == sqrt

Read typeclass

• Typeclass Read is the opposite of Show; it supports a read function that takes a string and gives you a value of
  your type. Unfortunately, you often have to specifically say what kind of value to return
Prelude> read "2" -- what kind of result should we get? Don’t know
*** Exception: Prelude.read: no parse
Prelude> -- You can specify what kind of 2 you want
Prelude> read "2" :: Int
2
Prelude> read "2" :: Float
2.0

• The read routine doesn't return a polymorphic value, it wants a specific type.

Prelude> x = read "2" :: Num a => a

<interactive>:79:5: error:
  * Could not deduce (Read a1) arising from a use of ‘read’
    from the context: Num a
    bound by the inferred type of x :: Num a => a
    at <interactive>:79:1-26
    or from: Num a1
    bound by an expression type signature:
      forall a1. Num a1 => a1
    at <interactive>:79:17-26
Possible fix:
  add (Read a1) to the context of
  an expression type signature:
    forall a1. Num a1 => a1
  • In the expression: read "2" :: Num a => a
    In an equation for ‘x’: x = read "2" :: Num a => a

• Though you can't treat read "2" as any kind of Num, you can treat the constant 2 as any kind of Num.

• 2 is a polymorphic constant

Prelude> x = 2 :: Num a => a  -- some sort of number
Prelude> :t x
x :: Num a => a

Integral typeclass

• The typeclass Integral isn't for integers, it generalizes real numbers.

  • Prelude> -- In the class header, (Real a, Enum a) => Integral a means that for a to be an
    integral type, it also has to be part of typeclasses Real and Enum.

Prelude> :info Integral
class (Real a, Enum a) => Integral a where
  quot :: a -> a -> a
  rem :: a -> a -> a
  div :: a -> a -> a
  mod :: a -> a -> a
  quotRem :: a -> a -> (a, a)
  divMod :: a -> a -> (a, a)
toInteger :: a -> Integer
{-# MINIMAL quotRem, toInteger #-}
-- Defined in ‘GHC.Real’
instance Integral Word -- Defined in ‘GHC.Real’
instance Integral Integer -- Defined in ‘GHC.Real’
instance Integral Int -- Defined in ‘GHC.Real’

• The fromIntegral function in Num takes an Integral and gives you an equivalent Num. The type of
  fromIntegral has two polymorphic types, one for the parameter and one for the result.
  • When you define a type and make it a Num, you define fromIntegral to show how your type is connected
to everyday Num values.
• Prelude> :t fromIntegral
  fromIntegral :: (Integral a, Num b) => a -> b
Prelude> fromIntegral 12
12
Prelude> :t fromIntegral 12
  fromIntegral 12 :: Num b => b

• There exist other built-in typeclasses, we can worry about them if we need them.

Chapter 4: Syntax in Functions

• What syntax do we use for defining functions? More than id paramvar = expr
• There’s definition by case. Below f is defined on all values (the f x case takes care of things that aren’t 0, 1, 2).
Prelude> :{
Prelude| f 0 = 0
Prelude| f 1 = 1
Prelude| f 2 = 2
Prelude| f x = 3
Prelude| :}
Prelude> f 17
3
Prelude> f 0
0
• The order of classes matters:
Prelude> :{
Prelude| g x = 2
Prelude| g 1 = 3
Prelude| :}

<interactive>:110:1: warning: [-Woverlapping-patterns]
    Pattern match is redundant
In an equation for ‘g’: g 1 = ...

Prelude> g 1
2

• Simple factorial
Prelude> :{
Prelude| fact 0 = 1
Prelude| fact n = n * fact(n-1)
Prelude| :}
Prelude> -- f works ok on values ≥ 0
Prelude> fact -1

(Oops! I forgot that - is a function; I asked for (fact -) 1

<interactive>:118:1: error:
  • Non type-variable argument in the constraint: Num (p -> p)
    (Use FlexibleContexts to permit this)
  • When checking the inferred type
    it :: forall p. (Eq p, Num p, Num (p -> p)) => p -> p

• On unix, ^C breaks an infinite loop
Prelude> fact (-1)
^C^C^C^CInterrupted.

• You can use underscore (_ ) instead of a variable if you don't care what value it has.
Prelude> f (x, _) = x
Prelude> :t f
f :: (a, b) -> a

• If you declare a variable like y here, the user might think you plan to use it. Not necessarily a big problem.
Prelude> f (x, y) = x
Prelude> :t f
f :: (a, b) -> a
Prelude> fst (3,'a')
3
Prelude> snd (3, 'a')
'a'
Prelude> s (_, y) = y --- our own version of snd
Prelude> :t s
s :: (a, b) -> b
Prelude> :t snd
snd :: (a, b) -> b
Prelude> s (3, 'a')
'a'

• The parameters can have more complicated types than is obvious
Prelude> h p = (snd p, fst p) -- reverse an ordered pair
Prelude> :t h
h :: (b1, b2) -> (b2, b1)
Prelude> let x = (3, 'a') in h x
('a',3)
Prelude> h (3, 'a')
('a',3)

You can use [] and _ to build functions by cases on lists.

Prelude> :{
Prelude| j [] = 0
Prelude| j [_] = 5
Prelude| j [_,_] = 7
Prelude| j [1, _, _] = 8 -- exactly 3 elements, first one being 1
Prelude| j [_, _, _] = 9 -- other 3-long lists return 9
Prelude| j (x : y) = x + j y -- if the list is of length 4 or greater...
Prelude| :}
Prelude> :t j
j :: (Num p, Eq p) => [p] -> p
Prelude> j [1,2,3]
8
Prelude> j [3,2,1]
9
Prelude> j [10,1,2,3]
18

Pattern Matching

* These definitions above all used pattern matching.
  * Many patterns look like expressions; the _ pattern is an exception

Prelude> k _ = 0 -- k of anything is zero
Prelude> k 7
0

* Here's an error: you can't evaluate a pattern. Haskell is actually trying to parse this as a function definition and it
  thinks you've left off the rest of the definition.

Prelude> k _

<interactive>:156:3: error:
  * Found hole: _ :: p20
    Where: ‘p20’ is an ambiguous type variable
  * In the first argument of ‘k’, namely ‘_’
    In the expression: k _
    In an equation for ‘it’: it = k _
  * Relevant bindings include it :: p1 (bound at <interactive>:156:1)
Valid substitutions include

\[
\begin{align*}
  f :: & \forall a \, b. \, (a, b) \rightarrow a & (\text{defined at <interactive>:125:1}) \\
  s :: & \forall a \, b. \, (a, b) \rightarrow b & (\text{defined at <interactive>:130:1}) \\
  k :: & \forall p1 \, p2. \, \text{Num} \, p1 => p2 \rightarrow p1 \\
  & \quad (\text{defined at <interactive>:154:1}) \\
  h :: & \forall b1 \, b2. \, (b1, b2) \rightarrow (b2, b1) & (\text{defined at <interactive>:135:1}) \\
  x :: & \forall a. \, \text{Num} \, a => a & (\text{defined at <interactive>:80:1}) \\
  \$trModule :: & \text{GHC.Types.Module} & (\text{defined at <interactive>:1:1}) \\
\end{align*}
\]

(Some substitutions suppressed; use \texttt{-fmax-valid-substitutions=N} or \texttt{-fno-max-valid-substitutions})

• You can use literal constants \((1, 2, \ldots)\) and the empty list \([\ ]\) and you can build up later patterns using \([\ldots, \ldots, \ldots]\) and colon \(\cdot\), and you can use tuples \((\ldots, \ldots, \ldots)\), and also various "data constructors" that we'll see in a bit.

• Things you can't use in patterns: regular functions like \(+, -, \ldots\), etc., \texttt{sqrt} \(++\) (list concatenation).

\[
\begin{align*}
\text{Prelude}> & f \, ([1,2,3] ++ y) = 2 \quad -- \text{fails} \\
& \text{Prelude}> \text{f} (1 \, : \, 2 \, : \, 3 \, : \, y) = 2 \\
& \text{Prelude}> -- \text{colon} \, (\cdot), \, \text{so we can specify "list of length} \geq 3 \, \text{beginning with} \\
& \text{Prelude}> -- 1, 2, 3" \, \text{that way} \\
& \text{Prelude}> -- \text{Can also have boolean test expressions as part of function} \\
& \text{Prelude}> -- \text{definitions} \\
& \text{Prelude}> -- \text{these are "guards"} \\
& \text{Prelude}> :\{ \\
& \text{Prelude}> \quad f \, n \mid n < 0 = 1 \\
& \text{Prelude}> \quad \mid n == 1 = 1 \\
& \text{Prelude}> \quad \mid \text{otherwise} = n * f(n-1) \\
& \text{Prelude}> :\} \\
& \text{Prelude}> --\text{should've been n == 0 = 1} \\
& \text{Prelude}> f \, (-3) \\
& 1 \\
& \text{Prelude}> f \, 6 \\
& 720 \\
& \text{Prelude}> f \, 0 \quad -- \text{last case makes this} \, 0 * f \, (0-1) = 0 * 1 = 0 \\
& 0 \\
& \text{• Otherwise just means True} \\
& \text{Prelude}> :\{ \\
& \text{Prelude}> \quad g \, n \mid n <= 0 = 1 \\
& \text{Prelude}> \quad \mid \text{True} = n * g(n-1) \\
& \text{Prelude}> :\} \\
\end{align*}
\]
Prelude> g 5
120

• Do have some issues with where the vertical bars can be:

Prelude> :{
Prelude| g n | n <= 0 = 1
Prelude| | True = n * g(n-1)
Prelude| :}

<interactive>:189:1: error: parse error on input ‘|’
(Haskell thought the | meant we were defining a whole different thing.)