Haskell, Part 5

CS 440: Programming Languages and Translators, Fall 2019

9/15: Act. Q’s added

A. More Haskell — Miscellaneous topics

• Simple I/O, do blocks, actions
• Modules
• data vs type declaration
• Instances of class types
• User-defined operators
• Sections
• The $ operator

Next time: Regular Expressions (Not Haskell!)

Simple I/O

• I/O doesn’t preserve referential transparency.
• If getline returns the string read in, then getline ≠ getline
• Need to sequester I/O off to the side so that the referentially transparent part remains state-free
  • You can run code that does I/O, but it can’t return results to the rest of your program.
  • Your I/O code can access the rest of your program, just not the other way

How to model printing/reading

• putStrLn "some string" prints and returns a value of type IO ()
  • By the way, putStrLn "string" is like putStrLn ("string" ++ "\n")
  • Printing modifies the printer - captured by the IO () value
  • Also has side effect of making printer print.
• There are routines that take IO t data and create other IO t’ data
  • But no way to take IO t and return (e.g.) an integer
  • (Unless you ignore the IO t data.)
• Reading via getline produces a value of type IO String
  • There’s a string hidden in there, but you can only access it certain ways
  • To pull string out involves fairly complicated code that uses operator >>= ("bind") and lambda functions
  • Haskell supports simpler syntactic sugar that maps to that code.
do block

- Use `variable <- IO String` expression to set `var = the string`
- A do block consists of a sequence of "actions" reading, printing, setting `x <- getLine` etc.
- Haskell translates do block into the more complicated format that uses `>>=` and lambdas [9/14]

```haskell
> :{
| do
|   putStr "data? 
|   str <- getLine
|   putStr str
|   putStr "...
|   putStrLn str
| :}
data? user types in data
user types in data...user types in data
```

- If you want a plain calculation like `y = str ++ str`, you use a `let`.

```haskell
> :{
| do
|   str <- getLine
|   let y = str ++ str
|   putStrLn y
| :}
argle
argleargle
```

- You can define a function using `let`:

```haskell
> :{
| do
|   let f x = x + 3
|   putStrLn (show (f 5))
| :}
8
```

- Note: Last part of a do block needs to be an action (not a `let` or an expression)

  - A `let` or expression would allow you to define a do block that returns a non-IO value, which would break referential transparency.

```haskell
> :{
| do s <- getLine
|    s ++ " " ++ s -- an expression!
| :}
```

```
<interactive>:115:4: error:
```
• Couldn't match type '[]' with 'IO'
  Expected type: IO Char
  Actual type: [Char]
• In a stmt of a 'do' block: s ++ " " ++ s
  In the expression:
    do s <- getLine
    s ++ " " ++ s
  In an equation for 'it':
    it
    = do s <- getLine
       s ++ " " ++ s

• [9/14] A do block is expected to have an overall type of IO a, so the last line of one is most often a
  putStrLn or putStrLnln, which both have the type IO ()
• [9/14] Entering a do block immediately evaluates it. You can delay evaluation by giving a name to
  the do block. This also lets you run the block multiple times.

> :{                        -- [9/14] this code example
|   pgm =                  -- you can also say pgm = do on one line so
|     do                   -- the putStrLnln gets only one level of
|       putStrLn "oops"   -- indentation
|   putStrLnln "oops"
| :}
> pgm
  oops
> pgm
  oops

• You can use if-else with actions; if a branch has > 1 action you need an inner do block for it

> :{                        -- if-else with actions
|   do s <- getLine
|   | if s == ""
|   | then
|   |     putStrLnln "empty"
|   | else do
|   |     putStrLn "String wasn't empty, it was "
|   |     putStrLnln s
| :}
hi
  String wasn't empty, it was hi
>

• You can use read to turn a string into a number
> :{  
| do s <- getLine  
|    let n = read s :: Int  
|    let dbl = n + n  
|    putStrLn ("Double " ++ show n ++ " is " ++ show dbl)  
| :}
1234
Double 1234 is 2468
>
------------------------ ended here 2019-09-04

Modules

- A module is a named collection of various definitions
  - Prelude is the library module loaded by default
  - In code get access to module using import module_name
    import Data.Char
- ghci commands to import and unimport :m +Data.Char, :m -Data.Char
  > :m +Data.Char
  Prelude Data.Char>
  - Note after importing Data.Char, it appears in the prompt as a loaded module.
- ghci command :browse Data.Char
  - Prints out all the names associated with the module.
    Prelude Data.Char> :browse Data.Char
    Data.Char.digitToInt :: Char -> Int
    Data.Char.isLetter :: Char -> Bool
    toUpper :: Char -> Char
    ... -- lots omitted
    GHC.Unicode.toUpper :: Char -> Char
    Prelude Data.Char> toUpper 'a' -- a function in Data.Char
    'A'
    Prelude Data.Char> map toUpper "abcDE"
    "ABCDE"

- If a module isn't loaded, you can use a qualified name
  Prelude Data.Char> :m -Data.Char
  Prelude> map Data.Char.toUpper "afsfs97423wruefsd"
  "AFSFS97423WRUEFSD"
  Prelude>

**data declaration vs type declaration**

- We've seen data declarations - they get you enumerations, algebraic types
• A type declaration just gives you another name for a type

  • **Structural equivalence**, not name equivalence

    > :{
    | type P1 = (Int, Int)
    | type P2 = (Int, Int)
    | p1 = (3, 5) :: P1 -- P1 just stands for (Int, Int)
    | p2 = (3, 5) :: P2
    | :}
    > p1 == p2
    True

• Datatypes use **name equivalence** -- even if two datatypes hold data the same way, they're of different types.

  data D1 = D1con Int
  data D2 = D2con Int
d1 = D1con 17
d2 = D2con 17
d1 == d2 -- type error

• (For even better example, I tried to define D1 and D2 to both use Dcon as a constructor, but that causes an error)

  data D1 = Dcon Int
d1 = Dcon 17
data D2 = Dcon Int -- error-- renaming constructor Dcon
d2 = Dcon 17 -- can't do
d1 == d2 -- can't do

• (Let's do this: Define D1 twice. Note below, Haskell uses a different name for the old version of D1, and data from the two versions isn't comparable, even though they are built the same way.)

    > :{
    | data D1 = Dcon Int
    | d1 = Dcon 17 -- uses first version of Dcon
    | :}
    > :{
    | data D1 = Dcon Int
    | d2 = Dcon 17 -- uses first version of Dcon
    | :}
    > :t d1
d1 :: Ghci19.D1
    > :t d2
d2 :: D1
    > d1 == d2 -- error

• Because type declarations are just synonyms, this example works on type D1 = Int:
Instances of class types

- Recall that an instance of a type class instance is a type that belongs to the class.
  - E.g., Eq Int, Eq Char, etc. But not Eq (Int -> Int)
- To create your own type and make it an instance, the definitions of various functions or operators on the type go inside the instance declaration, not with the datatype definition. Here’s part of an example with a Fraction type:

```hs
data Fraction = Frac Int Int deriving (Eq, Show, Read)
-- numerator over denominator

instance Eq Fraction where -- the code for == goes here
    (Frac n1 d1) == (Frac n2 d2)
    = n1 * d2 == d1 * n2

instance Num Fraction where
    (Frac n1 d1) * (Frac n2 d2)
    = ... omitted ... (+, -, etc.)
```

- You can declare the functions with the data declaration by defining synonyms for the operations and using those in the instance declaration. E.g.,

```hs
data Fraction = Frac Int Int deriving (Eq, Show, Read)
eqfrac (Frac n1 d1) (Frac n2 d2) = n1 * d2 == d1 * n2

instance Eq Fraction where
    (==) = eqfrac
```
User-defined operators

- Use sequence of special symbols (certain chars not useable)
- By default, user-defined binary [9/15] operators are infix and left associative (you can change this; not shown)

```
(/) :: Float -> Float -> Float
x / y = if y == 0.0 then x else x / y
```

-- If you prefer prefix notation, you can use that instead:
```
(/) x y = if y == 0 ....
```

- **Danger**: Don’t give existing operators definitions outside of their type class instance declaration

```
data Fraction = ....
(Frac n1 d1) + (Frac n2 d2) == ....
```

- When + is defined this way, outside the instance declaration for Num, this + overrides the old definition of +. E.g., 2+2 fails. (Very socially unacceptable!)

- Again, you can define a synonym function for (+) and refer to it in the instance declaration

```
data Fraction = ....
fracadd (Frac n1 d1) (Frac n2 d2) = ...
```

```
instance Num Fraction where
  (+) == fracadd -- as in earlier example
  -- definitions of other operators omitted
```

Sections

- A section is a partially-applied operator: \((expr \ optr)\) = \(\lambda y \rightarrow expr \ optr \ y\)

  - It works as a unary function.
```
> f = (1 +) -- adds one to something
> :t f
f :: Num a => a -> a
> f 2
3
```

- Can be useful for mapping across lists:
```
> map (\ _ \) [2..10]
[2,3,4,5,6,7,8,9]
```

- There’s also \((\ optr \ expr)\) = \(\lambda y \rightarrow y \ optr \ expr\), where you’ve specified the right-hand operand.
```
> map (/ 2) [2..10]
[1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0]
```

- You can use sections on a named function via backquotes
```
> f x y = 2 * x + y
> (5 `f` 3) == f 5 3 -- section with first argument
True
> (`f` 5) 3 == f 3 5 -- section with second argument
```
True

- The `-` operator is different because it has unary and binary interpretations: \((-1)\) is just \(-1\), not a section.

\[
\begin{align*}
&> \texttt{(:t (+ 3)} \\
&(\texttt{(+ 3)} :: \texttt{Num a => a -> a)} \\
&> \texttt{(:t (- 3)} \\
&(\texttt{(- 3)} :: \texttt{Num a => a)}
\end{align*}
\]

- You can use \((\texttt{expr }-)\) as a section, however.

\[
\begin{align*}
&> \texttt{(:t (1 -)} \\
&(\texttt{(1 -)} :: \texttt{Num a => a -> a)} \\
&> \text{\texttt{map (1 -) [2..10]}} \\
&\texttt{[-1,-2,-3,-4,-5,-6,-7,-8,-9]}
\end{align*}
\]

- You can parenthesize `-` to use it as a prefix function

\[
\begin{align*}
&> \texttt{\text{\texttt{map (\texttt{(- 1)} [2..10]}} \\
&\texttt{[-1,-2,-3,-4,-5,-6,-7,-8,-9]}
\end{align*}
\]

- You can also define a synonym for binary `-` and use that

\[
\begin{align*}
&> \texttt{sub = (-)} \\
&> \text{\texttt{map (`sub` 1) [2..10]}} \\
&\texttt{[1,2,3,4,5,6,7,8,9]} \\
&> \texttt{map (1 `sub`) [2..10] -- works like (1 -)} \\
&\texttt{[-1,-2,-3,-4,-5,-6,-7,-8,-9]}
\end{align*}
\]

The Function Application Operator `$`

- The `\$` operator is defined as \(f \$ x = f \ x\)

- Why bother? `\$` has very low precedence, normal function app has very high precedence
  - Using `\$` can let you omit some parens when you have complicated function expressions
  - You can write \((\texttt{expr}_1 \texttt{ \$ } \texttt{expr}_2)\) as \(\texttt{expr}_1 \texttt{ \$ } \texttt{expr}_2\) (without the parentheses)
  - `\$` can also be useful as a section

\[
\begin{align*}
&> \text{\texttt{\texttt{map \$ (256) [\texttt{sqrt, sqrt . sqrt, sqrt.sqrt.sqrt}]} \ \texttt{[16.0,4.0,2.0]}}}
\end{align*}
\]

Okay, that's it for Haskell for now — Next time: Regular Expressions
Activity Questions, Lecture 5 [added 9/15]

I/O vs referential transparency

1. Does the concept of I/O preserve referential transparency? (Hint: if read_a_line() returned the read-in string, would read_a_line() == read_a_line() in general?

2. The getline routine is of type IO String. Use :info IO in ghci to find out: Does IO String (or IO a in general) support Eq, Read, or Show? What happens if you run getline == getline in ghci?

Do blocks

3. Fix the do block below. It's intended to prompt the user, read in a Double and print it and its square root out, as shown by the sample run.

```haskell
> :{
>     pgm = do
>         putStrLn "Enter a double > ":info
>         s <- getLine -- (Hint: indentation?)
>         let nbr = read s in -- (Hint 1: declare a type?)
>         root <- sqrt nbr
>         putStrLn "The number was " ++ nbr ++ " and its sqrt was " ++ root -- Hint: similar with root
>     :}
> pgm
Enter a double > 2.58
The number was 2.58 and its sqrt was 1.606237840420901
```

Modules

4. How do you load the module Data.Char into ghci? Unload it? What happens if you use (e.g.) isLetter 'z' when Data.Char is unloaded? What happens if you use Data.Char.isLetter when Data.Char unloaded? What about when it's loaded?

5. What Haskell statement do you use to import a module like Data.Char, and does putting it in mean you can should still load the module in ghci?

Datatypes vs type synonyms

6. Say we have data T1 = C1 | C2 C1 | C3 [Char]. Which names are types and which ones are datatype constructors? Which names are being defined and which are being used? What are the types of C1, C2, and C3? What names can you use :info on in ghci? What about :browse? Does it matter that we left out a deriving (Eq, Show, Read) clause?
7. What's the difference between data and type declarations when it comes to equality testing?

**Class Type Instances**
8. What does it mean to declare that a datatype is an instance of a class type? How do you do this? What's required to get this to work?
9. If we want to add a new meaning to an existing operator (e.g., a new version of +), why is it important to do this within a class type instance declaration?

**Sections**
10. Say $f = \text{(/)}$ (or more generally, any binary infix operator). Which of the following mean the same thing (and what do they mean): $(f \; x \; y)$, $((/ \; x) \; y)$, $((/ \; x) \; y)$, $(x \; (/)) \; y)$, $((`f` \; x) \; y)$, and $((x`f`) \; y)$? Which of them are use sections?

**Juxtapositioning vs $**
11. What are the meanings of $a \; b \; (c \; d)$, $(a \; b \; c \; d)$, $(a \; b) \; (c \; d)$, $(a \; b) \; ($ $c \; d)$, $a \; (b \; ($ $c \; d)$, and $a \; b \; ($ $c \; d$? Which have different meanings? Which parentheses are redundant?
Solutions to Selected Activity Questions, Lecture 5

3. (do block)

```haskell
pgm = do
    putStrLn "Enter a double > 
    s <- getLine
    let nbr = read s :: Double
    let root = sqrt nbr
    putStrLn ("The number was " ++ show nbr
                ++ " and its sqrt was " ++ show root)
```

6. With `data T1 = C1 | C2 T1 | C3 [Char]`, `T1` is a type, `C1` is a constructor constant, and `C2` and `C3` are constructor functions. `T1`, `C1`, `C2`, and `C3` are being defined; Beside `Char`, `T1` is being used and makes `T1` a recursive datatype. `:info` can be used on all of `T1` and `C1 - C3`. You can't use `:browse` on any of them (`:browse` is for modules). Leaving out the `deriving` clause means that `T1` does not automatically support equality testing, printing `T1` values, or reading `T1` values.

7. Types declared using `data` can only be tested using *name equivalence* — with a type `data T = ...`, you can only check `T` values for equality with other `T` values. (And that's assuming all the underlying types on the r.h.s. support `Eq`.) Types declared using type use *structural equivalence* — with a type `type T = TE` (where `TE` is a type expression), you can check `T` values for equality against `TE` values (as long as `Eq TE`) because even though `T` and `TE` are different as pieces of text, they done values that are put together the same way.

11. (Function application: Juxtapositioning vs `$`)

- Expressions `a b (c d)`, `(a b) (c d)`, `(a b) $ (c d)`, and `a b $ c d` all mean the same thing: `a` is a binary function with argument 1 being `b` and argument 2 being the result of calling `c` on `d`.
- The expression `(a b c d)` has a different meaning, where `a` is a 3-parameter curried function being passed `b`, `c`, and `d`.
- The expression `a (b $ c) d` is different from all the others: `a` is a binary function but first argument is the result of calling function `b` on `c` (since `$` means function application). The second argument is `d`.
- The parens for `(a b)` in `(a b) (c d)`, `(a b) $ (c d)` are redundant because function application is left-associative. In addition, in `(a b) $ (c d)`, the parens in `(c d)` are also redundant because `$` has lower precedence than function application. (In `a b (c d)` and `(a b) (c d)`, the parens in `(c d)` are required.
- The parens in `a (b $ c) d` are not redundant because without them, you get the same meaning as `a b (c d)`.