Hello Haskell!

Patrick D. Elliott

April 11, 2023

Preliminaries

Homework

- Optionally, read chapters 2-3 of *Haskell programming from first principles*, covering basic Haskell syntax, and basic string/list manipulation.
    - You should be able to skim, since we're covering most of this material today.
- Do the building functions exercises, at the end of chapter 3, from p83-85.
    - The exercises simply involve writing a function to produce the desired results, so you should be able to test your answers yourselves using GHCi. If you get stuck send me an email before next week's class.

Organizational

- Class on May 9th will take place remotely (I'll distribute a webex link via rocketchat closer to the time).
- Class on May 16th will be cancelled - I'm in America for a conference.

Lambda calculus summary

- The lambda calculus is a *formal system* (i.e., a logic) for reasoning about *functions*.
- In the lambda calculus, computation is modelled as a form of simplification, using the following rules:
  - β-reduction. $(\lambda x.f(x))(y) \Rightarrow f(y)$
  - α-conversion. $\lambda x.x \Rightarrow \lambda y.y$
  - η-reduction. $\lambda x.f(x) \Rightarrow f$
- Haskell can be thought of as a kind of lambda calculi, where *running a program* amounts to reducing a complex expression until we reach normal form.
- Reduction doesn't always converge on a normal form; sometimes expressions diverge; this corresponds to *non-terminating computations* (imagine, for example, a program implementing a timer that runs indefinitely).

Chapter 1 exercises, p18 n7

$$(\lambda xyz.xz(yz))(\lambda x.z)(\lambda x.a)$$

1. Curry arguments: $(\lambda x.\lambda y.\lambda z.xz(yz))(\lambda x.z)(\lambda x.a)$
2. α-conversion: $(\lambda x.\lambda y.\lambda z_1.xz_1(yz_1))(\lambda x.z)(\lambda x.a)$
3. β-reduce: $(\lambda y.\lambda z_1.(\lambda x.z)z_1(yz_1))(\lambda x.a)$
4. β-reduce: $\lambda z_1.(\lambda x.z)z_1((\lambda x.a)z_1)$
5. β-reduce: $\lambda z_1.(\lambda x.z)z_1 a$
6. β-reduce: $\lambda z_1.za$
7. Normal form!

Getting started with Haskell

Expressions and declarations

- Everything you write in Haskell is either an *expression* or a *declaration*.
    - Expressions can be values, functions, functions applied to values, etc.
    - Declarations are bindings that allow us to name complex expressions.

Here are some examples of expressions in Haskell:

```haskell
1
1 + 1
"Icarus"
```

The REPL

- The GHCi REPL stands for the *Glasgow Haskell Compiler interactive Read-Eval-Print-Loop*.
- It allows us to evaluation Haskell expressions directly without the need to save the program in a source file.
- There are a few different ways to get a GHCi instance:
    - In the browser: `https://tryhaskell.org/`
    - By installing GHC and running `ghci` in the terminal.

REPL cont.

- When we type an expression into the REPL it automatically evaluates it for us.
- The following expressions are already in normal form, so they simply evaluate to themselves.

```
ghci> 1
1
ghci> "Icarus"
"Icarus"
```

A complication

- In reality, it's a bit more complex than that.
- An expression like 1 evaluates to an *integer*, but technically speaking integers aren't the kind of things that can be printed to an output, rather their *string representations*.
- Under the hood, GHCi exploits Haskell's type system to determine whether an expression is *showable*; what we see is given by the function associated with the showable type class.
    - We'll learn more about what this means later in the semester.

Evaluating arithmetic expressions

- GHCi can be used as a basic calculator by inputting arithmetic expressions.
- Complex expressions are evaluated until we reach normal form:

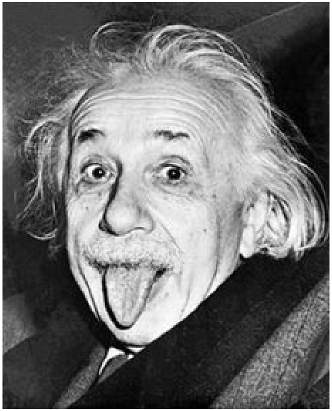
```
ghci> ((1 + 2) * 3) + 100
109
```

- Note that GHCi doesn't show us any of the intermediate steps.
- N.b. expressions that can be reduced are called redexes (i.e., *reducible expressions*).

Functions

- *Functions* in haskell are particular kinds of expressions, which play a very important role.
- Just like mathematical functions, they map inputs to outputs, in a determinate fashion.
- A Haskell function always evaluates to the same result when given the same argument values.
    - This property is known as *referential transparency*, and makes Haskell programs extremely straightforward to reason about.
    - For those of you with some experience programming in an imperative language like C, this is quite a departure! In imperative languages, evaluating a line of code might affect the state in a way which changes subsequent evaluations.

Referential transparency



"Insanity is doing the same thing over and over and expecting different results." (Albert Einstein)

More on functions

There are a number of different ways of declaring functions in haskell. Here is the simplest way:

```
ghci> triple x = x * 3
ghci> triple 4
12
```

Function names always start with lower case letters in haskell. It's good practice to use descriptive function names, which conventionally use camel case, e.g.:

```
ghci> multiplyByThree x = x * 3
ghci> multiplyByThree 4
12
```

Abstractions

- Note that the equals sign = indicates that this is a *declaration* rather than an expression.
- Note that declarations are much like abstractions, in the sense that the variable(s) to the left of the = bind the corresponding variable(s) to the right.
- In fact it's also possible to define functions directly as abstractions, using the following syntax:

```
ghci> triple = \x -> x * 3
ghci> triple 4
12
ghci> (\x -> x * 3) 4
12
```

Call back

- Remember when I said that printing values in GHCi is more complicated than it first appears?
- Try evaluating an abstraction, e.g.,

```
ghci> (\x -> x * 3)
```

Intermission

How would we declare a function that has one parameter and words for al the following expressions?

```
pi * (5 * 5)
pi * (10 * 10)
pi * (2 * 2)
pi * (4 * 4)
```

Note that `pi` is an expression that is given by the Haskell `Prelude`. The prelude is a module (i.e., a set of declarations) that is implicitly imported by default.

Solution

```
ghci> circleArea radius = pi * (radius * radius)
ghci> circleArea 5
78.53981633974483
```

Note that as well descriptive function names, we can also use descriptive *variable* names; there's no reason (aside from brevity) that we have to use single letters as variable names.

Prefix vs. infix

As you've probably gathered, the syntax for *function application* in Haskell just involves whitespace, i.e.., `f x` means $f(x)$.

The arithmetic operators like + are *infix operators*; they can be used as ordinary functions by enclosing them in paretheses:

```
ghci> 200 + 300
500
ghci> (+) 200 300
500
ghci> ((+) 200) 300
500
```

Declarations in the REPL

We can define functions and later use them with a single REPL session; the REPL has a limited form of state.

```
ghci> y = 10
ghci> x = 10 * 5 + y
ghci> myResult = x * 5
ghci> myResult
300
```

You can quit the REPL by typing `:q`; declarations won't persist between REPL sessions, so typing `myResult` in a new session will give you the following error:

```
ghci> myResult
error: Variable not in scope: myResult
```

Declarations in source files

In order to get your declarations to persist, you need to write them into source files (called *modules*). Try saving the following as `learn.hs`.

```haskell
module Learn where

y = 10
x = 10 * 5 + y
myResult = x * 5
```

You can now *load* the module in GHCi.

```
ghci> :l learn.hs
Ok, one module loaded.
ghci> myResult
300
```

Tips for writing source files

A module must always start with a module declaration `module MyModule where`; the module name should always start with a capital letter, unlike a function declaration.

White space and line-breaks are *significant*; the following won’t compile; the second line should be indented:

```
x = 10 *
5 + y
```

Comments are lines starting with a double dash.

```
-- a random declaration serving no apparent purpose:
x = 10 * 5 + y
```

More tips

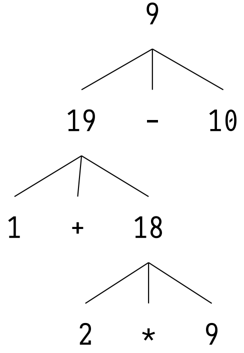
Using a text editor with support for Haskell syntax highlighting will be a big help. Some options:

- VS Code.
  - Probably the most popular text editor right now, with excellent haskell support built in.
- Emacs (with `haskell-mode`).
  - This is what I use. If you're not already familiar with emacs, I definitely wouldn't recommend it.
- Notepad++
  - I don't really know anything about this, but apparently it's a good option if you're running Windows.

You can also just use the online Haskell playground, which has syntax highlighting baked in.

Basic arithmetic

Basic arithmetic can help us get a feel for how haskell expressions are evaluated, e.g., `1 + 2 * 9 - 10`.



Associativity and precedence

Arithmetic infix operators in haskell:

- `+` : addition
- `-`: subtraction
- `*`: multiplication
- `/`: fractional division

You can get information about operator *associativity* and *precedence* using the `:info` command in GHCi.

```
ghci> :i (+)
infixl 6 +
```

N.b. this will also give you information about the *type* of the expression. This won't be relevant yet, but will be important soon.

The $ operator

This $ is an important infix operator that is often used to write terse haskell code without parentheses. Here is its definition:

```haskell
f $ a = f a
```

This is an `infixr` operator with the lowest possible precedence:

```haskell
(2^) $ 2 + 2
(2^) (2 + 2)
```

`let` and `where`

- `let` is used to introduce an expression.
- `where` is a *declaration* that is bound in its containing syntactic construct.

```haskell
printInc n = print plusTwo
  where plusTwo = n + 2
```

```haskell
printInc n = let plusTwo = n + 2
              in print plusTwo
```

Intermission

```haskell
z = 7
x = y ^ 2
waxOn = x * 5
y = z + 8
```

Write out what will happen when you run the following:

- `10 + waxOn`
- `(+ 10) waxOn`
- `(-) 15 waxOn`
- `(-) waxOn 15`

Types and strings

Types in formal semantics

Types in Haskell are a way of *categorizing values*; they provide a syntactic restriction on how complex expressions are built.

You might be familiar with types if you've ever taken a semantics course before.

- *is happy*: $\langle e, t \rangle$
- *Henning*: e

Types in Haskell

- Haskell has a more complex and powerful type-system than the one you might be used to from formal semantics.
    - Formal semantics typically uses the *simply-typed lambda calculus* as a basis.
    - Haskell is based on System F, i.e., the *polymorphic lambda calculus*, which allows for universal quantification over types.
    - Various *language extensions* exist to make Haskell's type system even more powerful (dependent types, linear types, etc).
    - In this course, we won't go much beyond simple types and some basic polymorphism.

Getting information about types

You can find out the type of any haskell expression quite easily using the `:type` command in GHCi:

```
ghci> :t "hello haskell!"
"hello haskell!" :: String
ghci> :t 'a'
'a' :: Char
```

- Note that single characters are enclosed in single quotes.
- The double colon `::` is interpreted as *has the type*.

Type annotations

We explicitly annotate expressons with their type using `::`.

```
ghci> :t ("hello haskell!" :: String)
"hello haskell!" :: String
```

If we annotate an expression with the wrong type, we'll get an error:

```
ghci> :t ("hello haskell!" :: Char)
<interactive>:1:2: error:
    • Couldn't match type ‘[Char]’ with ‘Char’
      Expected: Char
        Actual: String
```

String types

`String` is actually a name for a *complex type*, `[Char]`.

That is to say, strings in haskell are actually just *lists of characters*.

In general, for any type `a`, the type `[a]` is the type of a list of things of type `a`.

Printing strings

We can print strings to the standard output in GHCi using the `putStrLn` or `putStr` functions.

```
ghci> putStrLn "hello haskell!"
hello haskell!
```

Examine the type of `putStrLn`. You'll notice something quite interesting.

```
ghci> :t putStrLn
putStrLn :: String -> IO ()
```

In Haskell, we use arrow notation for function types (we'll come back to this later). `IO ()` is a special type to indicate that the program has some effect beyond evaluation of functions and arguments.

Printing strings from a source file

```haskell
  -- print1.hs

module Print1 where

main :: IO ()
main = putStrLn "hello world!"
```

If we load `print1.hs` from GHCi and execute `main`, *hello world!* will be printed to the standard output.

The `main` function

In haskell `main` is the default action when building an executable, or running it in GHCi, and it must always be of type `IO ()`.

Input/output is much more complicated in Haskell than in most other programming languages, since it involves exploiting Haskell's type system to reason about *side effects*. This will be a topic for later in the semester.

Concatenating strings

There are two functions for concatenating strings in the haskell prelude:

```
(++) :: [a] -> [a] -> [a]
concat :: [[a]] -> [a]
```

- ++ is an infix operator, whereas ~concat is just an ordinary function.
- Note that a in the type signature is a *type variable*. Free variables in type signatures are implicitly universally quantified in Haskell.
- This means that both ++ and `concat` are *polymorphic* functions; they can be used to combine lists more generally.

Types primer i

In formal semantics, functional types are often written using angled-brackets (e.g., $\langle e, t\rangle$), following the convention used by (Heim, Irene and Kratzer, Angelika, 1998).

Haskell uses arrow notation, which is more commonly found in the computer science/programming language literature, although some semantics texts use arrow notation (Carpenter, Bob, 1998).

Arrow notation in Haskell is *right associative*:

- `a -> b -> c` $\iff$ `a -> (b -> c)`

Types primer ii

Let's look again at the type for list concatenation:

```
(++) :: [a] -> [a] -> [a]
```

- `(->)` is a type *constructor*. It takes two types `a`, `b` and returns the type of a function from *as* to *bs*.
- One important feature of haskell is the possibility of defining arbitrary constructors; `([.])` takes a type `a` and returns the type of a list of *as*.
- Remember, free type variables are implicitly universally quanitified, which means that list concatenation is defined for something of type `[a]`, where `a` can be *any type*.

Strings as lists of chars

```
"hello haskell!"
['h','e','l','l','o',' ','h','a','s','k','e','l','l','!']
```

- Strings surrounded by double quotes are really just *syntactic sugar* for lists of characters.
- Syntactic sugar is just a notational convention built into the language that makes our lives as programmers easier.
- Lists are actually also syntactic sugar! We'll learn what lists really are in a bit.

Polymorphism

What do you think the following evaluates to?

```
[1,2,3] ++ [4,5,6]
```

What happens if we try to evaluate the following:

```
"hello" ++ [4,5,6]
```

More list manipulation

```
ghci> head "Henning"
'H'
ghci> tail "Henning"
"enning"
ghci> take 0 "Henning"
""
ghci> take 3 "Henning"
"Hen"
ghci> drop 3 "Henning"
"ning"
ghci> "Henning" !! 2
'n'
```

Totality and safety

What happens when you run the following in GHCi:

```
ghci> "yo" !! 2
```

Let's examine the type of `!!`; as expected, its a function from a list of as, to an integer, to an a.

```
(!!) :: [a] -> Int -> a
```

Note however, that this isn't a *total* function; there are some lists and integers for which this function will be undefined.

Partial functions in haskell are considered *unsafe*, because the type system doesn't prevent us from providing an illicit value as an argument to the function.

Building lists with cons

The final list manipulation function we'll look at is an important one: cons.

```
ghci> 'h' : []
[h]
ghci> 'h' : "enning"
"henning"
```

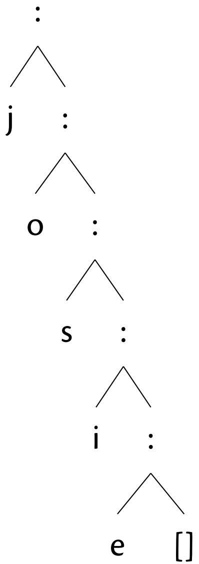
In haskell, lists are built up by successive application of cons:

```
'h' : ('e' : ('n' : ('n' : ('i' : ('n' : ('g' : []))))))
```

Since : is right associative we can drop the parentheses.

Lists in haskell are therefore *singly-linked lists of characters*.

Singly-linked lists



An aside on performance

- For most industrial applications, singly-linked lists of chars would be a terrible choice.
- On the other hand, this means that strings ”come for free” on the basis of chars and extremely general list manipulation functions.
- For anything we do in this class, performance won’t be an issue. For serious work with strings, the standard is the Haskell `text` library.

Prolegomenon to types

The simply-typed lambda calculus

- In preparation for next week's class with type.
- Types are syntactic categories used to restrict what counts as a valid expression.
- Basic ingredients:
    - A set of primitive types.
    - A recursive rule for constructing complex (i.e., functional) types.
    - Rules for computing the type of a complex expression from the types of its parts.

Primitive types

- Let's keep things simple, and start with just two primitive types:

$$\textbf{Typ} := \{\, \texttt{Int}, \texttt{Bool} \,\}$$

- We'll assume that integers are possible values and have the type `Int`:

$$\texttt{73 :: Int}$$

- We'll also assume two primitive values with the type `Bool`:

$$\textbf{true} :: \texttt{Bool}, \textbf{false} :: \texttt{Bool}$$

Functional types

We'll now state a recursive rule for complex (functional) types, using the Haskell convention for types.

- If `a` $\in$ **Typ**, then `a` is a type.
- If `a` is a type, and `b` is a type, then `a → b` is a type.
- Nothing else is a type.

This means that we have many complex types like the following:

- `(Bool → Bool) → Int`
- `Int → Int`

Functions and their types

- We can assign some useful operations their types:

$$(+) :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}$$

$$(-) :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}$$

$$\textbf{factorial} :: \texttt{Int} \to \texttt{Int}$$

$$\textbf{odd} :: \texttt{Int} \to \texttt{Bool}$$

$$\textbf{even} :: \texttt{Int} \to \texttt{Bool}$$

$$\textbf{and} :: \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}$$

Types of complex expressions

Functional applications: Let $\beta :: \mathtt{a} \rightarrow \mathtt{b}$, $\alpha :: \mathtt{a}$ be an expression of the SLTC. $\beta(\alpha)$ is an expression of type $\mathtt{b}$.

Abstractions: Let $\beta :: \mathtt{b}$ be an expression of the SLTC, and υ a variable of type $\mathtt{a}$. $\lambda\upsilon.\beta$ is an expression of type $\mathtt{a} \rightarrow \mathtt{b}$.

Exercise

Can you infer the types of the following expressions? Go step by step.

$$\textbf{and}(\textbf{odd}(4))(t)$$

$$\lambda x.\textbf{odd}(\textbf{factorial}(x))$$

$$\lambda f.f(\lambda x.(+)(x)(2))$$

Type inference

Often, you can *infer* the type of an expression without specifying the type of all of its sub-parts.

When you try to compile a haskell source file, or evaluate an expression in GHCi, the compiler will attempt to check that it is well-typed, by inferring the types of any expressions that don’t have an explicit type provided.

Since haskell’s type system is more expressive than we have here, the type-inference algorithm is quite complicated (the compiler is based on an algorithm called *Hindley-Milner*).

Restrictions of a first-order type system

In a first order type-system, we can only state typed identity functions. What is the type of *the* identity function?

$$\lambda x.x :: ?$$

Restrictions of a first-order type system cont.

Consider the following functions:

$$\mathbf{not} :: \texttt{Bool} \rightarrow \texttt{Bool}$$
$$\mathbf{not}' :: \lambda f.\lambda x.\mathbf{not}(f(x))$$
$$\mathbf{not}'' :: \lambda r.\lambda x.\lambda y.\mathbf{not}(r(x)(y))$$

- What are the types of not' and not"?
- Is there a way of expressing all three functions as a single-operation? If not, why not?

Bonus: recursion

Remember the expression ω:

$$(\lambda x.xx)(\lambda x.xx)$$

- Try to give it a concrete type.
- This problem is related to the lack of Turing completeness of the SLTC.
- On the other hand, because the SLTC is relatively constrained it has some extremely nice logical properties:
    - The SLTC is a sound and complete logic.
    - *Type-checking* (checking whether an expression is well-typed), and *type inference* are decidable.

Next time

- Next time we'll learn much more about Haskell's type system.
- Haskell's type system is more expressive than the SLTC - we can do everything we can in the SLTC and more.
- We'll learn about *polymorphic functions* corresponding *polymorphic datatypes*; a first step in understanding the kinds of powerful abstractions that Haskell provides to reason about computation.

$\mathcal{F}in$

References

Carpenter, Bob (1998). *Type-Logical Semantics*, MIT Press.

Heim, Irene and Kratzer, Angelika (1998). *Semantics in Generative Grammar*, Blackwell.