Haskell Tutorial: Introduction

September 19, 2019

[2]:
```
:opt no-lint
```

1 Introduction

Haskell is a statically typed, purely functional programming language with type inference and lazy evaluation. The first version of Haskell was defined in 1990 and the definition of the Haskell language is a maintained by the Haskell Committee. Haskell is used widely in academia and (to a lesser extend) in the industry. Pandoc, a popular tool to convert between different markup formats, is written in Haskell.

So why is it a neat idea to learn Haskell? There are a couple of possible answers:

- A functional language expands the way you think about programming.
- Haskell makes it easy to reason about programs.
- Haskell is a safe language.
- Haskell is a pure language avoiding side effects.

A popular implementation is the Glasgow Haskell Compiler. It comes with a number of tools. The most important for beginners are:

- `ghci` - an interactive Haskell interpreter
- `ghc` - a Haskell compiler translating Haskell into native machine code
- `runghc` - a program to run Haskell code as scripts

A tool that can be quite helpful in particular for beginners is `hlint`. HLint suggests possible improvements to Haskell source code, often providing suggestions how to simplify code. Another package we like to use is HUnit, a framework for writing unit test cases.

1.1 Expressions

The easiest way to get started with Haskell is to launch `ghci` and to type expressions into the interactive Haskell interpreter. Below are some simple arithmetic expressions. Haskell so far works as a simple calculator. Note that it can do arbitrary precision integer arithmetic.

[3]:
```
2 + 5
```

7

[4]:
```
2 + 5 * 3
```

```
17
```

[5]:
```
(2 + 5) * 3
```

```
21
```

[6]:
```
2^123
```

```
10633823966279326983230456482242756608
```

Haskell is a functional language and that means that pretty much everything in Haskell are functions. Even the simple arithmetic expressions above can be seen as function calls written in *infix notation*. The + operator, for example, is just a syntactic shorthand refering to a function that takes two arguments and returns the sum of them. We can actually write the above expressions in a pure functional representation using *prefix notation*:

[7]:
```
(+) 2 5
```

```
7
```

[8]:
```
(+) 2 ((*) 5 3)
```

```
17
```

[9]:
```
(*) ((+) 2 5) 3
```

```
21
```

[10]:
```
(^) 2 123
```

```
10633823966279326983230456482242756608
```

Of course, writing arithmetic expressions in prefix notation usually does not make the expressions more readable and hence nobody would do this in production code unless there is a very specific reason. At the end, the goal of almost every programming effort is to produce code that is *easy to understand* and *easy to reason about*.

Haskell is also a typed language. So far, we only used integral numbers. Lets see what happens if we devide two numbers.

[11]:
```
6/2
```

```
3.0
```

Apparently, this returns a floating point number. If we want integer division, we have to use the `div` function. Note that in Haskell we write the function name followed by its arguments. If necessary, parenthesis are placed before the function name and after the last argument. This is different from other programming languages where parenthesis are required to enclose the arguments of a function. In C or C++, one would write `div(6, 2)` but in Haskell this is simply `div 6 2`. If parenthesis are necessary in an expression, the function call would be `(div 6 2)`.

[12]:
```
div 6 2
```

```
3
```

Since `div` is a function that takes two arguments and produces a single result value, we can write this expression as well in infix notation by enclosing the function name in backticks.

[13]:
```
6 `div` 2
```

```
3
```

There are predefined functions to test whether a number is `even` or `odd` and we can use the == operator to test whether two values are equal. In a similar way, we can compare numbers. The infix operator notation uses the operators < (less than), > (greater than), <= (less than or equal), >= (greater than or equal), and /= (not equal).

[14]:
```
odd 1
```

```
True
```

[15]:
```
even 1
```

```
False
```

[16]:
```
1 == 2
```

```
False
```

The results show that the comparison operators (functions) return truth values (`True` or `False`).

[17]:
```

```

1.2 Lists

Lists are the most fundamental data type in Haskell. A list is represented by a comma-separated sequence of elements surrounded by square brackets. All elements of a list must be of the same type, i.e., they must all be numbers or they must all be characters and so on. An empty list is represented by opening and closing square brackets with nothing inbetween.

[18]:
```
[]
```

```
[]
```

[19]:
```
[0,1,2,1,3]
```

```
[0,1,2,1,3]
```

Haskell supports an enumeration notation, which is a conveninent way to create longer list (and even infinite lists, as we will see later).

[20]:
```
[1..10]
[0,5..20]
[10,9..1]
```

```
[1,2,3,4,5,6,7,8,9,10]
```

```
[0,5,10,15,20]
```

```
[10,9,8,7,6,5,4,3,2,1]
```

More complicated lists can be created using *list comprehensions*. List comprehensions mimic what we know from mathematics. In mathematics, if we want to define the set of odd numbers between 1 and 10, we could write $\{x | x \in \{1, .., 10\}$ and x is odd$\}$. This is how this idea can be translated into Haskell:

[21]:
```
[ x | x <- [1..10], odd x]
```

```
[1,3,5,7,9]
```

If we want the list of squares of all odd numbers between 1 and 10, we can simply apply a function. In mathematics, we would write $\{x^2 | x \in \{1, .., 10\}$ and x is odd$\}$. For readability, we use the infix operator notation. (As an exercise, rewrite this example in prefix notation.)

[22]:
```
[x^2 | x <- [1..10], odd x]
```

```
[1,9,25,49,81]
```

It is possible to create more complex list comprehensions with variables ranging over multiple lists:

[23]:
```
[ x + y | x <- [1..5], y <- [1..3], x == y]
```

```
[2,4,6]
```

Lists can be concatenated using the ++ operator.

[24]:
```
[1..5] ++ [6..10]
```

```
[1,2,3,4,5,6,7,8,9,10]
```

The elements of a list have a defined order and a certain element can appear multiple times in a list. Hence, lists should not be confused with sets. Since lists maintain an order, it makes sense to talk about the first element of a list or the last element of a list. The `head` function returns the first element of a list while the `tail` function returns the list without its first element. The `last` function returns the last element of a list while the `init` function returns the list without the last element.

[25]:
```
head [1..10]
```

```
1
```

[26]:
```
tail [1..10]
```

```
[2,3,4,5,6,7,8,9,10]
```

[27]:
```
last [1..10]
```

```
10
```

[28]:
```
init [1..10]
```

```
[1,2,3,4,5,6,7,8,9]
```

The `:` operator (often called the cons operator) prepends a head element to a list. In other words, the `(:)` function takes as first argument an element and as second argument a list and it returns the list with the element prepended. Do not confuse this with the `(++)` function, which takes two lists and returns the concatenation of these two lists.

[29]:
```
head [1..10] : tail [1..10]
```

```
[1,2,3,4,5,6,7,8,9,10]
```

Note that the `:` operator is right associative. This makes it easy to use the cons operator repeatedly to create a list.

[30]:
```
1 : 2 : 3 : []
```

```
[1,2,3]
```

The `null` function returns `True` if a list is empty and `False` otherwise. This gives us a convenient (and fast) way to test whether a list is empty. The `length` function returns the number of elements in a list.

[31]:
```
null [1..10]
```

```
False
```

[32]:
```
length [1..10]
```

```
10
```

Since Haskell allows us to create infinite lists (yes it does!), it is strongly recommended to use `null` to test whether a list is empty. The evaluation of the expression `length [1..] == 0` will not terminate since the calculation of the length of an infinite list will not terminate.

[33]:
```
null [1..]
```

```
False
```

[34]:
```
null [x | x <- [1..10], x == 2^x]
```

```
True
```

There are more predefined useful list functions. Some frequently used functions are:

- The `concat` function takes a list of lists and concatenates them
- The `reverse` function takes a list and reverses all elements in it
- The `take` function takes a list and a number `n` and returns the first `n` elements (or fewer if the list has less than `n` elements)
- The `drop` function takes a list and a number `n` and returns the tail after removing the first `n` elements (or an empty list if the list has less than `n` elements)
- The `elem` function returns `True` if a given value is in the list and `False` otherwise
- The `map` function takes a function and list and applies the function to all elements of the list
- The `filter` function takes a function returning either `True` or `False` and a list and returns all elements of the list for which the function applied to the list element returns `True`

The last two functions as examples of higher order functions. Higher order functions take functions as arguments or return functions. Programming with higher order functions is extremely powerful and they often replace simple loops that you may know from imperative programming.

[35]:
```
concat [[1..3], [4..7],[8..10]]
```

```
[1,2,3,4,5,6,7,8,9,10]
```

[36]:
```
reverse "hello world"
```

```
"dlrow olleh"
```

[37]:
```
take 5 [1..10]
```

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

[38]:
```
drop 5 [1..10]
```

```
[6,7,8,9,10]
```

[39]:
```
elem 5 [1..10]
```

```
True
```

[40]:
```
map even [1..10]
```

```
[False,True,False,True,False,True,False,True,False,True]
```

[41]:
```
filter odd [1..10]
```

```
[1,3,5,7,9]
```

[42]:
```
filter (< 7) [1..10]
```

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

The last example shows something rather special. We want to select all numbers from the list that are less than 7. To do this, we take the operator < and we fix the second argument to the constant 7. This gives us a function that has only a single argument and we apply this new function too all list elements in order to produce the result. The technique applied here is called currying and rather fundamental in Haskell. Currying encourages programmers to solve generic problems and then the generic solution can be tailored easily to solve a given task at hand. In the

example, we take the generic "less than" operator and we curry it down to the much more specific "less than 7" function.

We finish this section with introducing the indexing operator `!!`. The indexing operator takes two arguments, the first argument is a list and the second argument is an index number. The operator returns the element of the list at the index position. (The first element of a list has the index position 0.) An index number that is not present in the list leads to an error. Hence, seasoned Haskell programmers often try to avoid the indexing operator in order to write pure functions that have no side effects.

[43]:
```
[1..] !! 9
```

```
10
```

1.3 Characters and Strings

A character value is defined by writing the character surrounded by single quotes. Hence, `'a'` represents the character a. Haskell supports unicode and hence it is no problem to write funny characters in Haskell code. (Whether doing so is a good idea is a different discussion.) Special characters like newlines are represented by escape sequences that start with a backslash. For example, the escape sequence `'\n'` represents a newline character while the escape sequence `'\\'` represents the backslash character.

[44]:
```
'a'
```

```
'a'
```

[45]:
```
'\n'
```

```
'\n'
```

A string is simply a list of characters. Since string literals appear quite frequently, there is a special notation for string literals (lists of characters). A string literal is a possibly empty sequence of characters surrounded by double quotes.

[46]:
```
"this is a string"
```

```
"this is a string"
```

Since a string literal defines a list of characters, we can use list functions on strings.

[47]:
```
length "this is a string"
```

```
16
```

[48]:
```
"this " ++ "is " ++ "a " ++ "string"
```

```
"this is a string"
```

[49]:
```
null ""
```

```
True
```

[50]:
```
head "foo"
tail "foo"
last "foo"
init "foo"
```

```
'f'
```

```
"oo"
```

```
'o'
```

```
"fo"
```

Comparison operators are defined for characters and strings. On strings, the operators do lexicographic comparison.

[51]:
```
'a' < 'b'
'b' < 'a'
"aa" < "ab"
"21" < "111"
21 < 11
```

```
True
```

```
False
```

```
True
```

```
False
```

```
False
```

1.4 Tuples

Tuples are another way to pack multiple values together. Tuples are represented by the values contained in the tuple separated by comma and enclosed in parenthesis. There are some key differences between lists and tuples: - Tuples have a fixed number of values and they are immutable. It is not possible to add values to a tuple or to remove values from a tuple. Tuples are useful in situations where the number of elements is fixed. For example, an edge in a graph can be represented by the two nodes connected by the edge. - The values of a tuple can be of different types. A tuple can easily contain a number, a string, another tuple, and a list.

[52]:
```
(42, "answer", ('a', "string"), [1..10])
```

```
(42,"answer",('a',"string"),[1,2,3,4,5,6,7,8,9,10])
```

Pairs are tuples that have two elements. For pairs, we have predefined functions that can be used to access the first (`fst`) and the second (`snd`) element of a pair.

[53]:
```
fst ("one", 2)
```

```
"one"
```

[54]:
```
snd ("one", 2)
```

```
2
```

A uesful function to create lists of pairs is `zip`. The `zip` function takes two lists and returns a list of pairs: The first element of the first list is paired with the first element of the second list, the second element of the first list is paired with the second element of the second list and so on. This continues until one of the lists has been exhausted.

[55]:
```
zip [1..] ["apple", "peach", "pear"]
```

```
[(1,"apple"),(2,"peach"),(3,"pear")]
```

Lists of tuples can also be created using list comprehensions. Lets create a list of pairs where the second element is a square of the first element and each element is in the range [1..10].

[56]:
```
[(a,b) | a <- [1..10], b <- [1..10], b == a^2]
```

```
[(1,1),(2,4),(3,9)]
```

1.5 Types

Haskell is a strongly and statically typed programming language. This means that every value has an associated type. Think of types as a set of values with defined operations on them. For example, take the set of integral numbers. It is natural to think of these numbers as a type.

- *strongly typed*
  - Haskell does not automatatically cast a value from one type into a different type
  - By avoiding automatic type conversions, some programming error can be cached before they cause problems
- *statically typed*
  - types are known at compilation type by the compiler / interprefer
  - static typing, in combination with strong typing, makes type errors impossible to occur at runtime
- *type inference*
  - Haskell can infer type information and hence the programmer does not have to specify types explicitely for all values (or functions)

Even though Haskell can infer types, we will always specify the type signatures of functions explicitely as this is good practice and captures that intention of the programmer. Some of the core Haskell types are:

- `Char`: characters (unicode)
- `Bool`: one of the two boolean values `True` and `False`
- `Int`: small signed integer number, usually restricted to 64 bits or 32 bits (platform specific)
- `Integer`: integer numbers of arbitrary precision (well, bounded by the available memory)
- `Double`: floating point numbers, usually 64-bits wide
- `Rational`: rational numbers

We can be explicit that a certain value (or expression) has a certain type. The `::` operator indicates that the value on the left side of the operator is to be understood as having the type defined on the right side of the operator.

[57]:
```
42 :: Double
```

```
42.0
```

[58]:
```
3 / 2 - 1 / 4 :: Rational
```

```
5 % 4
```

The last example says that we want the expression to return a rational number. The Haskell notation `5 % 4` represents the rational number commonly written as the fraction $\frac{5}{4}$ in mathematics. While we can define the type of values and expressions, we usally do not do this and rely on Haskell's type inference to determine the type of values and expressions.

In order to get used to types, it is useful to inspect what Haskell knows about types. In `ghci` or this notebook, it is possible to enquire the runtime system about the type information of an expression. This is done by sending the `:type` command to the runtime (the command may be abbreviated to `:t` as long as there is no other runtime command that starts with a t).

[59]:
```
:type 'a'
```

```
Char
```

[60]:
```
:type "hello"
```

```
[Char]
```

[61]:
```
:type []
```

```
forall a. [a]
```

[62]:
```
:type head
```

```
forall a. [a] -> a
```

[63]:
```
:type map
```

```
forall a b. (a -> b) -> [a] -> [b]
```

The last example says that map is a function that takes a function mapping from type `a` to type `b` and a list of values of type `a` and it returns a list of values of type `b`. Since `a` and `b` are not constrained, the map function can be used for arbitrary types `a` and `b`.

1.6 Functions

Since Haskell is a functional language, the programmer primarily defines functions. Defining a function can be very simple. Lets define a function that takes a single number and returns the square of the number, i.e., $f(x) = x^2$.

[64]:
```
f x = x^2
```

[65]:
```
f 4
```

```
16
```

[66]:
```
f 25
```

```
625
```

We can use the newly defined function to define more functions. Lets define $g(x) = f(x) - 8$.

[67]:
```
g x = f x - 8
```

[68]:
```
g 4
```

```
8
```

While this all works as one would expect, we never really specified that the argument of `f` must be a number. Still, if we try to invoke `f` on the character `c`, Haskell provides us with a type error. What happens here is that Haskell has inferred that the function `f` is not defined for characters, because the function definition uses an expression that is not defined for characters. The error message may not be readable yet but the point here is that this is an error that is thrown at compile time and before execution starts. This means that the programmer is required to fix the problem in the code before it can be run and do harm.

[69]:
```
f 'c'
```

```
No instance for (Num Char) arising from a use of `f'
Possible fix: add an instance declaration for (Num Char)
In the expression: f 'c'
In an equation for `it': it = f 'c'
```

It is good practice to not rely on Haskell's type inference for functions and to define the *type signature* of a function explicitly. This way, you also help Haskell's type inference since you declare the intended type of a function. Here is an example how you define an explicit type signature for a function.

[70]:
```
f :: Integer -> Integer
f x = x^2
```

[71]:
```
f 1234
```

```
1522756
```

The type signature of `f` now says: `f` has a type that takes a value of type `Integer` and returns a value of type `Integer`. While we have defined `f` for `Integer` numbers above, it might be useful to define `f` also for `Int` numbers or all possible numbers. We can do this easily in Haskell.

[72]:
```
f :: Num a => a -> a
f x = x^2
```

[73]:
```
f 5
```

```
25
```

[74]:
```
f 5.0
```

```
25.0
```

[75]:
```
f (5 :: Rational)
```

```
25 % 1
```

The type signature now says that `f` receives a value of a numeric type `a` and it returns a value of the same numeric type `a`. The `a` in the type signature is a type variable; it holds the name of a numeric type. Our new definition of `f` is a *polymorphic* function since it can be appliedd to arguments with different types. As the examples above show, the type of the argument of the function `f` determines the type of the value returned by the function: `f` is either a function receiving an integer number and returning an integer number or it is a function receiving a floating point number and returning a floating point number, or it is a function receiving a rational number and returning a rational number.

Since we now know how to define functions, we are essentially ready to start functional programming. But hey, what about all those things that are common in other programming languages like variables, conditional statements, loops? Well, you do not need them because there are functional equivalents (such as pattern matching, guards, recursion, higher order functions, monads) that provide you with a rich toolbox to implement arbitrary algorithms as functions in Haskell. If you are new to programming, you might find that learning functional programming is not very difficult. If, however, you have quite some experience with imperative programming languages, then learning Haskell may become a certain challenge since you have to learn to look at programming from a somewhat different perspective. While things may appear initially in a sense weird or difficult, you will hopefully soon start to realize the power of a functional programming style and carry it over even when you write code in an imperative programming language. Many iterative programming languages have been recently extended to better support functional programming. The recent interest in functional programming is also driven by the fact that functional programs are much easier to turn into concurrent programs than imperative programs since there is no mutable state and functions are pure, i.e., they do not cause side effects.

Here are a few tips on how to think as a functional programmer:

- Do not think about a program as a sequence of operations
- Try to think about the relationship between input and output
- Try to drive simplicity to a maximum
- Think in terms of composition and not in terms of inheritance
- Think about side-effects and how to separate them from the functional core of a program