A Brief Introduction To Haskell

With Lazy Evaluation, Infinite Lists and more!

Zarya Mekathotti

Monday 20, September

Outline

Python vs. Haskell

Haskell is a *purely functional* programming language.

Python is a *general-purpose* programming language. (You can use Python for coding in an imperative, procedural, structural, object-oriented or functional style).

Imperative (Python) vs. Functional (Haskell)

Imperative Programming:

- Executes statements, in order to change the state of a program

Functional Programming:

- Evaluates expressions
- Typically avoids using a mutable state
- Functions are 'first-class citizens'

Pure Function:

- A pure function is a function that has *no side effects*

Basic Types

The basic types used in Haskell are:

- `Int` - integer (4)
- `Char` - character (`'a'`)
- `Bool` - `True` or `False`
- `Integer` - arbitrary-precision integers (slow!)
- `Float` - single-precision floating point numbers
- `Double` - double-precision floating point numbers

Basic Operators

Operators in Haskell are quite similar to those in Python.

- `+`, `−`, `∗`, `/` are *infix* operators
- `||`, `&&` are also *infix* operators which work on `Bools`
- Haskell has some built-in mathematical functions like `sin`, `cos`, `tan` which are *prefix* operators
- Prefix function application has higher precedence than infix function application
- To use an *infix* operator `op` as a *prefix* operator, we type `(op)`

Functions

- A function is a rule for associating each element of a source type A with a unique member of a target type B.
- In Haskell, we express this as: `f :: A -> B`
- For functions with multiple arguments, the types are listed in sequence: `f :: A -> B -> C -> ... -> Z` where `Z` is the return type of the function

Some Basic Functions...

Let's write the following functions:

- `successor`: this function takes an `Int` and returns the next number
- `addDigit`: this function takes two `Int`s. The first is a number, the second is a single digit. It returns the numbers 'concatenated together'. E.g `addDigit 123 4` returns `1234`

Using Helper Functions

- How would I write a function to print the nth Fibonacci number?
- `fib 0 = 0, fib 1 = 1, fib 2 = 1, fib 3 = 2 ...`
  - How can I improve the complexity of this function?

Lists

Lists are sequences of objects

- Empty list: `[]`
- `[1,3,5,7]`
- A list of `Char`s are called strings (type `String`):
  `"hello!" == ['h', 'e', 'l' ,'l', 'o']`

Arithmetic Sequences

The special form `[a,b..c]` can help us to build arithmetic sequences. For example: `[2,4..10]` returns `[2,4,6,8,10]`. These work with `Int`s, `Char`s, `Bool`s and `Float`s.

List Construction

Lists are actually constructed using two basic building blocks.

- `[]` (the empty list)
- `:` (the cons operator). This adds a new element to the front of the list.

Some Functions For Lists!

These functions will use **pattern matching**:

- `sum`: Takes a list of `Int`s and calculates the sum of its elements.
- `pos`: Takes a `Char` and a `String` and determines the position of the `Char` in the `String`.

Evaluation

Haskell evaluates an expression by reducing it to its simplest equivalent form. So we can think of evaluation as simply reducing expressions until there are no more expressions to reduce. A reducible expression is called a redex.

Reduction works by repeatedly reducing redexes until there aren't any more. The expression is then in *normal form*

Examples of expressions in normal form:

- `7`
- `"Hello, World"`
- `[True, False, True, True]`
- `1 : 2 : 3 : []`

But how does Haskell know which expressions to evaluate first?

Reduction Strategies

Suppose we've defined the following function:

```
double :: Int -> Int
double x
    = x + x
```

We can evaluate the expression `double (3 + 4)` using certain reduction sequences.
Call-by-value:

```
double (3 + 4)
=> double 7
=> 7 + 7
=> 14
```

Call-by-name:

```
double (3 + 4)
=> (3 + 4) + (3 + 4)
=> 7 + (3 + 4)
=> 7 + 7
=> 14
```

Reduction Strategies

Haskell's reduction strategy is **Lazy Evaluation!**

Lazy evaluation basically chooses the leftmost, outermost redex to be reduced first. Lazy evaluation reduces a redex only if the value of the redex is required to produce the normal form. We can see lazy evaluation working with a function defined as such:

```haskell
f :: Float -> Float -> Float
f x y
    | x < 0     = 0
    | otherwise = y
```

If x is negative, the second argument (y) doesn't need to be evaluated. (Demo)

(Most programming languages choose eager evaluation, which is where function arguments are reduced before reducing the function application itself).

Using Lazy Evaluation In Pattern Matching

Let's look at the definition of the `&&` operator (logical and). (Reminder: `(&&)` is the infix version of the `&&` operator).

```
(&&) :: Bool -> Bool -> Bool
True && x    = x
False && x  = False
```

If we try to evaluate the expression `(1 == 2) && (2 == 2)`, Haskell tries to pattern match with the rules for `&&`. Then the first argument is reduced to `False`. Now we can match the reduced expression to the second line of this definition. So the second argument did not need to be reduced.

Higher-order Functions

Haskell has some higher-order functions which are really powerful!

- `map :: (a -> b) -> [a] -> [b]`
- `filter :: (a -> Bool) -> [a] -> [a]`
- `++ :: [a] -> [a] -> [a]`

Quicksort

Let's write quicksort! We're going to be using:

- A function type signature
- Recursion
- Higher-order functions
- The `++` operator
- Pattern matching

Further Reading

- The `successor` function is greatly related to Peano's axioms and the definition of addition for the real numbers
- There are many higher-order functions in Haskell such as `iterate`, `foldr`, `zipWith` ... You can find more here: http://zvon.org/other/haskell/Outputglobal/index.html
- Higher-order functions in Haskell are also similar to some functions in Kotlin! `map` and `filter` are very useful.

Thanks!

Thanks for listening :)