

CSCC24: Principles of Programming Languages

Notes

Joshua Concon

University of Toronto Scarborough – Winter 2018

This course is taught by Dr. Albert Yu Cheong Lai. If you find any problems in these notes, feel free to contact me at conconjoshua@gmail.com.

Contents

1	Thursday, January 11, 2017	2
1.1	The Root Cause of this Course	2
1.1.1	Higher-order Functions on Aggregates	2
1.1.2	Combining Forms	3
1.1.3	Example Topic: Evaluation Order	4
1.1.4	Example Topic: Scheme Macros	4
1.1.5	Dynamic and Static Typing	4
1.1.6	Parametric Polymorphism	5
1.1.7	What is "Powerful"? – The Tradeoff	6

1 Thursday, January 11, 2017

The purpose of this course is to see the trade-offs between various features in programming languages. This course exists because different programming languages have different features, for example, Java has both class-based OOP and auto-garbage collection while C has neither, but C has union types that Java doesn't have. This means rewriting code into a different language isn't necessarily easy. There may be large semantic differences

1.1 The Root Cause of this Course

A guy named "John Backus" gave a lecture for the acceptance for the Turing Award in 1977. He addressed the question, "Can programming be liberated from the von Neumann style?"

Languages then had been only superficial enhancements to the CPU writing 1 word onto memory at a time i.e:

```
1 s := s + a[i]
```

Backus proposed a new direction for programming languages:

- Higher order functions that work on aggregates (a whole list, an array, a dictionary, etc...)
- Combining forms, for example, function composition ($g \circ f$)
- Reasoning by algebra, for example, the associative law for a function
- If you need a state, use coarse-grained state transitions rather than changing only one word at a time. (So passing an old state into a stateless function that does a lot and returns an answer and a new state.)

1.1.1 Higher-order Functions on Aggregates

Note that the notation to apply a function to several parameters is:
(Haskell)

```
1 f x y z
```

(Scheme:)

```
1 (f x y z)
```

So in Haskell:

```
1 fmap f [x0, x1, ...]
```

will compute

```
1 [f x0, f x1, ...]
```

And

```
1 fmap abs [3, -1, 4]
```

Computes

```
1 [3, -1, 4]
```

And

```
1 foldr (+) 0 [3, 1, 4]
```

Computes

```
1 3+(1+(4+0))
```

Note 2 points:

- "on aggregates" means to work on a whole list at once (such as an array or some "container")
- "Higher-order functions" means that some parameters are functions, so different combinations makes the language more customizable.

Java and MATLAB have the former but lack the latter.

1.1.2 Combining Forms

An obvious example is function composition ($g \circ f$).

In Haskell, this is:

```
1 g . f
```

And in Racket (Scheme) this is:

```
1 compose g f
```

For example, the following code computes the 1-norm of your vector.

```
1 foldr (+) 0 . fmap abs
```

There are other combining forms. There is another example in Haskell.

```
1 (f &&& g) x = (f x, g x)
```

The point is that you can combine functions to perform compound tasks, and this type of language is not about shorter code (although it has that side effect), but about working with building blocks.

1.1.3 Example Topic: Evaluation Order

You can define your own logical "and" in Scheme

```
1 (define (my-and b c) (if b c #f))
2 (my-and #f (list-ref '#t #f #t) 10))
```

The second line fails in Scheme, but if typed in the Haskell version, succeeds.

In most languages, parameters are evaluated before passed into the bodies of functions. In Haskell however, parameters are passed as is. Because of this, in Haskell, many short circuiting operators and control constructs are user-definable, and therefore, very customizable.

1.1.4 Example Topic: Scheme Macros

Scheme offers a macro system for user-defined constructs:

```
1 (define-syntax-rule (my-and b c) (if b c #f))
```

Now if we run the following code, it succeeds.

```
1 (my-and #f (list-ref '#t #f #t) 10))
```

The explanation for this is that this is a macro expansion in Scheme, so the parameters are copy-pasted into the macros. This means that there is a downside, for example:

```
1 (define-syntax-rule (double x) (+ x x))
2 (double (* 3 4))
```

The second line spawns two copies of `(* 3 4)` and performs redundant work, while Haskell's version does not. The Upside is that Scheme's macro system offers other flexibilities not shown in this lecture.

1.1.5 Dynamic and Static Typing

In Scheme:

```

1 (if #f 0 (+ 0 "hello"))
2 (if #t 0 (+ 0 "hello"))

```

The first line fails but the second line succeeds. This is because Types are checked dynamically. When running the program, only the code that is actually run is checked.

In Haskell, the following line fails:

```

1 if True then 0 else 0 + "hello"

```

The reason for this is because types are checked statically, without running, over all the code. (If this code is compiled, then at compile time, if interpreted, then at load time, etc.) So the error of adding 0 to "hello".

Food for thought, Java is both compiled and interpreted.

1.1.6 Parametric Polymorphism

In Haskell, we define:

```

1 trio x = [x, x, x]

```

The inferred type is:

```

1 a -> [a]

```

This is analogous to Java's

```

1 <T> LinkedList<T> trio(<T> x)

```

Note: That the following 2 lines are both legal if we have trio defined

```

1 trio 0
2 trio "hello"

```

"Parametric" means: Supposed you have defined d of type $a \mapsto [a]$, Then you would need one test to know what it does. Say we test $dTrue$ and the answer has length 2. Then we can deduce that dx returns $[x, x]$ for all x .

The basic explanation for this is that d cannot vary behaviour by types. Haskell allows type-determined behaviour, but the function type will look like:

```

1 Foo a => a -> [a]

```

1.1.7 What is "Powerful"? – The Tradeoff

"Macro systems, dynamic typing, ... are powerful." This refers to the flexibility for the implementer or the original author.

"Static typing, parametric polymorphism... are powerful." This refers to the predictability for the user or the maintainer.

Programming is a dialectic class struggle between the user and the implementer. Or between the maintainer and the original author.