COMPSCI 630 Systems **Spring 2017**

Lecture 20: May 1

Lecturer: Emery Berger *Scribe(s): Divyesh Harit, Peter May*

20.1 Pure and Impure Functional Programming Languages

- A pure functional programming language is a language that has no side-effects. This means that in such a language, executing a function does not cause a persistent state change of the program. In a functional language, all objects are immutable.

  · **Example 1:**

  x = 0; //Global

  f() {

  x = x+1; //State change

  return x;

  }

  In this case, value of function depends on history of the function:

  y = f(); //y=1

  z = f(); //z=2

  .

  .

  .

  q = f(); //Depends on how many times f() was called.

  Thus, f() here is not a pure function.

  · **Example 2:**

  sqrt(x) {

  return $\sqrt{x}$;

  }

  y = f(4); //2

  z = f(4); //2

  Same result here, so purity is maintained.

- Examples of impure FPLs: Scala, ML Family Languages (OCaml), Lisp, Scheme. Lisp is impure because it allows state modification.

- In python, everything is a statement which essentially returns void; this is not a functional style. But it does have functions, lambdas, first class expressions, and list comprehensions (which come from Miranda). So we can say that here we have functional features in a traditionally non-functional language.

- Though there has been a steady adaption of FPL concepts into modern languages over time, one of the reasons that functional languages have not been very popular in the mainstream is because it's generally harder to program in FPL. Another reason could be that writing state-changing programs seems to be an intuitive and obvious way.
- Reading/Writing files is also a state-change, yet pure FPL implement it, which bypasses their definition. In OCaml, read/write operations are marked as "unsafe", and allowing the user to work on them at their own risk.
- **MONADS:** The main concept behind monads is to enforce uni-direction: Everything is single-threaded, one can write and read files but they are not allowed to go back. This allows read/write operations without breaking the definition of a pure FPL.

20.2 Memoization

Memoization makes functional programming faster. It basically helps to implement a 1-to-1 mapping:

Argument	**Return**
Some argument	Some value returned by f() for this argument
See the above argument?	Then just return this value!

20.3 Static vs Dynamic languages

- In static languages, the type of an object is predefined, either specified by the user (Java, C, C++), or inferred by the language (Haskell). In dynamic languages (Python, JS, Lisp), there's no such type specification.
- For example, in Python, all the following declarations are valid: x = 12, x = "Hi!", and x = True.
- Haskell uses so called algebraic data types. Popular examples of algebraic data types are product types, (e.g. tuples, classes) and sum types, which are values that can take on different types based on construction.
- In Haskell, there are also type classes:

  sqrt:: float $\rightarrow float$

sqrt:: Num a $\rightarrow a \rightarrow a$

[Type inferred, as long as it's of type Num]

20.4 Strict vs Lazy Evaluation

- f(x) = 12

  In strict evaluation: f(0/0) = 1/"bottom"

  In lazy evaluation, the same would evaluate to the original value, 12. Thus in lazy evaluation, there's an "Evaluation by need".

- Functions are written assuming nothing, so it's like an infinite buffer is being used.

  So, say there's an infinite buffer of all ones [1,1,1,1..] and user needs only 10; this would be very convenient in lazy evaluation as only those 10 would be loaded in memory.

- Drawback of lazy evaluation: Accidental space-leak when requesting, say, the last object of that infinite list.

20.5 The Case for FPL for Big-Data Applications

Reasons:

- The MapReduce paradigm. Easy fast.
- Parallelism; MapReduce is embarrassingly parallel.
- Extremely fault-tolerant (immutability makes this possible). Can run more than once; if something fails, do it again on another processor.