CSCI 3136

Haskell

Zeh

Alex Safatli

Wednesday, April 3, 2013

Contents

Overview

Everything we will talk about here can be done in C or even assembly language; the question is not whether it can be done but how *easily* it can be done. It's all about expressiveness of the language.

Recall: imperative programming specifies what the computer should do; functional programming specifies what the **value** of a function should be. The exact sequence of steps to compute the value is left unspecified — this is one form of **declarative programming**.

The consequences of under-specifying things like this is that it needs mechanisms to specify execution order when necessary, code correctness (a consequence of side-effect freedom) and memoization (if a function always computes the same value; the potentially length computation does not have to be done again — chance for optimization), lazy evaluation (the order that things happen does not matter, so delaying can happen for computations until it is absolutely necessary), ...

Currying shifts away from a value-oriented view to a function-oriented view. See example.

In control constructs, else-branches are necessary because functions need a value.

Loops make no sense in a functional language because there are no side-effects; there is no way to keep track of a counter – if you want the loops to dos something together, information has to be passed from one information to next which is only necessary with side effects. Iteration becomes recursion.

Haskell allows multiple definitions of the same function; this is known as **patterns**. All must have the same type; it uses the first one that matches the actual parameters. The former parameters are patterns that need to be matched by the actual parameters. This is semantically identical to a switch statement.

Haskell does support arrays but they are slow. See slide.

"Iterating" over lists, or many iterative processes, can be expressed as a combination of a few common idioms. These include **mapping**, **folding**, and **filtering**.

Mapping is the notion of applying a function to each element of a sequence *independently* — these are transformations on the sequences of a list and the creation of a new list (as the original list is immutable). **Folding** is the notion of accumulating elements in a list. And, **filtering** is the notion of computing a sublist of all elements satisfying a certain condition. This is the idea of developing primitives for high-level iterative tasks.

> **NOTE** The first two are typically used in **MapReduce** constructs which can be parallelized efficiently. This is a construct made popular by Google.

Implementing these constructs are fairly *trivial*. Note that the name **functional programming** comes from the fact that functions are first-class values; the entire focus is on functions.

Lists have a limitation: **all elements must be of the same type**. This problem does not arise in Lisp or Scheme; Scheme checks types at runtime and declarations of type cannot be made. Therefore, objects of different values can be put into a given list and manipulated as long as a function is not applied to a list element that it does not support. **Pairs** and **tuples** allow us to group things of different types; see slide — these have a limitation that lists do not have: the number of elements is fixed.

Lists can be **zipped** together; association of the elements in both lists into tuples. Similarly, a list of pairs can be unzipped into a pair of lists. Variants exist (zipping of three-ten lists and zipping with a function).

Haskell also supports **anonymous functions**. When `f x y = x * y` is written, this is syntactic sugar for:

```
f = \ x y -> x * y
```

Overview continued on next page...

Why a backslash? Schemers call anonymous functions lambdas and backslashes are the closest thing in the ASCII alphabet.

Functions

Multi-argument functions are not really multi-argument functions. Functions with more than one argument are functions with one argument of a certain type whose result is a function with ... (see slide). We call:

```
f :: a -> b -> c -> d
```

A **curried** function. `f x y z` really means `((f x) y ) z`. Why are curried functions better? See slides. This is often called **point-free** programming; the focus is on building functions from functions rather than specifying the value a function produces on a particular argument. This cannot work without **function composition** (an operator signified by `.`).

> **NOTE** Scheme's multi-argument functions are not defined as single-argument functions mapped to other functions (e.g., there is no currying). Partial function application in Haskell provides this. Furthermore, lazy evaluation can be present in Scheme, but it must be explicitly forced (Haskell is lazy by default).

Type classes are a very similar notion to Java interfaces (see Quicksort implementation).

The Pitfalls of Laziness

There are **three** kinds of **folds** (accumulating things as one standard idiom): **right-to-left**, **left-to-right** (lazy), and **left-to-right** (strict). The space usage of summing a list of integers is practically $O(n)$; this is even true with `foldl` which is tail-recursive because of laziness (stubs are generated until evaluation is absolutely needed) — a linear-space structure is created. The strict version of `foldl` is, on the other hand, $O(1)$ — constant time.

The Unrealistic Dream of No Side Effects

Disallowing side effects has some very large advantages; values of functions depend only on its arguments (this makes formal reasoning about code correctness easier and has a practical benefit for testing). Side effects *are* necessary when interactions with the real world are needed — without this interaction, programs are useless. Storing state in data structures and updating these data structures requires side effects.

An **action** in the **IO monad** `getChar` reads a character from stdin and returns it (it is not a function). A **monad** is a structure that allows us to sequence actions; the **IO monad** is the monad that allows us to interact with the outside word.

Haskell programs must have a main function; the aim is to create a CLEAR SEPARATION between the part of the computation that has side effects (which needs to be expressed as monadic actions) and the part that does not (which is expressed using pure functions). Monads are generally classes which support four types of operations (see slide). **Do-Notation** makes monad composition of actions easier to write (translating it from this form and compiling it into the expanded code).

Even I/O operators are lazy — they will only operate if they really have to. See slides.