Algorithm Analysis and Design (CS1.301)

Project Proposal (Revised)
Functional Implementation of Algorithms

Abhinav S Menon

1 Project Plan

The project will consist of an implementation of all the algorithms covered in class in Haskell (a functional programming language).

2 Motivation

The pseudocode for all the algorithms shown in class was based on imperative languages. Functional languages, however, are based on a different set of principles. This means that programming in functional languages can need an entirely distinct perspective from that of imperative coding (see section 5).

Therefore, coding the algorithms in a functional language like Haskell will, I feel, improve my understanding of them by providing an alternate perspective. It will also habituate me to writing functional code, which is a useful skill as functional programs are frequently much shorter[1] than equivalent imperative programs.

3 Use Case

As mentioned above, since functional code is usually smaller than other implementations, it is possible to make use of these programs to a compact software that requires an efficient implementation of them.

4 Tentative Timeline

The following is a tentative timeline of the work involved (covering all algorithms to date, *i.e.*, before 8th October).

[1]Hudak, Paul, and Mark P. Jones. Haskell vs. Ada vs. C++ vs. awk vs.... an experiment in software prototyping productivity. Technical report, Yale University, Dept. of CS, New Haven, CT, 1994.

- October 15th: up to Lecture 5
  - Fibonacci
  - Integer multiplication
  - All divide and conquer algorithms (mergesort, matrix multiplication, order statistics, polynomial multiplication)
- October 22nd: up to Lecture 8, 15th September (all greedy algorithms)
  - Kruskal's algorithm
  - Huffman encoding
  - The set cover problem
- November 10th: up to Lecture 12, 29th September (dynamic programming algorithms)
  - Shortest path in a DAG
  - Longest increasing subsequence
  - Edit distance
  - Chain matrix multiplication
  - The knapsack problem (with and without repetition)
  - Shortest reliable path
  - All pairs shortest path
  - Independent set

5 Functional Programming

5.1 The Lambda Calculus

Functional programming is a paradigm of programming developed from a theoretical computation model called the lambda calculus, created by Alonzo Church. Popular functional programming languages include Haskell, Lisp and Rust.
The lambda calculus precedes the Turing machine model of computation (in fact, Church was Alan Turing's doctoral advisor). Although they are extremely divergent models in terms of their abstractness and the methods they require, they were later proven to be equivalent, substantiating what is now known as the Church-Turing Thesis.

The most basic form of the lambda calculus is built around functions, with no notion of data or datatypes. All information – including numbers, booleans, etc. – is implemented as nameless functions (the imperative programmer will recognise that this forms the basis of Python's lambdas).

5.2 Features of FP Languages

Possibly the most important feature of functional programming is *referential transparency.* Once a variable is declared to have a value, it can always be substituted for that value with no change to the validity of the program. To quote Miran Lipovača, author of "Learn You a Haskell",

> If you say that `a` is 5, you can't say it's something else later because

you just said it was 5. What are you, some kind of liar?

An important result of referential transparency is that the correctness of programs can be proved rigorously by substituting function definitions in their calls until the result is obtained.

This feature is closely related to the lack of *state* in functional programming, so prominent in imperative programs. Other consequences of stateless implementations are

- no iteration (as the counter continuously changes value)
- no side-effects (no memory manipulation or, notably, I/O)
- no global "accumulator" variables

All three of these features are extensively made use of in the pseudo-code shown in class, which means that a functional implementation of the same algorithms will require a considerably different thought-process to code.

5.3 Haskell

Haskell is a general-purpose functional programming language. It has all the features of ordinary functional programming, with some additional ones.

Firstly, Haskell is *strongly typed*, unlike some functional and some imperative languages (Common LISP and Python). Even in the absence of type declarations, types are inferred using a modified version of the Hindley-Milner algorithm.

Secondly, Haskell is *lazy* – roughly, expressions are not evaluated until they absolutely have to be. An important consequence of this is that Haskell has the capacity for infinite lists, as long as one does not attempt to print them.

Although Haskell is pure for the most part, it is possible to simulate state and side-effects (like I/O) using a feature called *monads*. Monads enable us to separate the impure parts of the code (involving side-effects, etc.) from the pure computation. This is not, however, frequently used outside such contexts.
As a side note, monads also enable non-determinism to be simulated and error handling to be handled conveniently.

5.4 An Example

One way to find the Fibonacci numbers (by generating a list, in linear time) is as follows:

```
fibs = 0 : 1 : (zipWith (+) fibs (tail fibs))
```

Note that function application is indicated by simply listing the arguments after the function, *e.g.* if `succ` adds 1 to its argument, then `(succ 5)` evaluates to 6.

The `(:)` operator (sometimes called *cons*) attaches an element to the front of a list. It is right associative; thus the first two elements of `fibs` are 0 and 1.

The `zipWith` function applies a binary operator (in this case `(+)`) to each pair of corresponding elements of two lists. Here, the two lists are `fibs` itself, and `(tail fibs)`, which is `fibs` without its first element.

Since Haskell is lazy, when (say) the third element is requested, it tries to evaluate (the first element of `fibs`) + (the first element of `(tail fibs)`). Both of these are available and it returns 1, as needed.
In this manner, any element of `fibs` (which is an infinite list) can be computed.