15–150: Functional Programming

Final Examination

December 12, 2011

- **See the final page for extra material that was distributed during the exam!**
- There are 24 pages in this examination, comprising 7 questions worth a total of 97 points.
- You have 180 minutes to complete this examination.
- Please answer all questions in the space provided with the question.
- You may refer to one double-sided 8.5" x 11" page of notes, but to no other person or source except the course staff, during the examination.
- Unless otherwise indicated, you do not need to give purpose/examples/tests for functions.
- In multi-part coding questions, we will assume the earlier tasks are correct while grading the later tasks. So it is in your interest to attempt the later tasks, even if you don't solve the earlier ones.

Full Name: ______________________________

Andrew ID: ______________________________

Question:	Short Answer	Geometry	TreeNorm	Analysis	Sublist	Profiling	Flattening	Total
Points:	4	11	20	14	16	15	17	97
Score:								

Question 1 [4]: Short Answer

(a) (4 points) Various streaming video services (YouTube, Netflix, Hulu) let you watch video in your Web browser. As you watch the video, your browser sends a requests to a server, which returns *response*s. Suppose that, for each request, the server can return one of the following responses:

- When the video is over, the response is a link to watch it again.
- A video is divided up into frames (individual pictures, which are played consecutively to make the video). When the video is playing, the reponse is a collection of video frames.
- The video services share content, so if you make a request on one service, it may forward that request to another service, and return their response. However, this forwarded response needs to be accompanied by the service to which the request was forwarded (e.g. so their ads can be displayed). For example, if you request a video from Netflix, it might return a respose that says "I checked with Hulu, and here is their response."

Let the type `address` represent Web addresses (URLs).
Let the type `frame` represent a video frame.
Let the type `service` represent a video service (YouTube, Netflix, Hulu, ...).

Define a datatype `response` representing the above responses:

`datatype response =`

Solution:

```
datatype response =
     Over of link
   | Playing of frame Seq.seq
   | Forward of service * response
```

THIS PAGE INTENTIONALLY LEFT BLANK

Question 2 [11]: Geometry

In graphics, people make three-dimensional models of scenes they plan to turn into pictures or movies. One technique used is called *constructive solid geometry*, or CSG. A complicated three-dimensional object (say, Buzz Lightyear) is defined in terms of some basic shapes: spheres, rectangles, etc. This problem is inspired by the idea of CSG but is highly simplified: we are only working in two dimensions, and the only thing your constructions will be able to do is report "black" or "white" for a particular point.

A point is represented in Cartesian coordinates, as an (x, y) pair of `real`s. A construction is represented by a function that returns `true` on all points that should be colored black. Thus:

```
type point = real * real
type constr = point -> bool
```

where

> for any construction `c`, `c p ==> true` iff `p` should be colored black, and
> `c p ==> false` iff `p` should be colored white.

In this problem, you will define some basic shapes, and some ways of composing them to make constructions. For instance, construction in Figure 1 is the union of:

- the intersection of
  - a disk, and
  - the inverse of
    - a rectangle
- a rectangle
- another rectangle

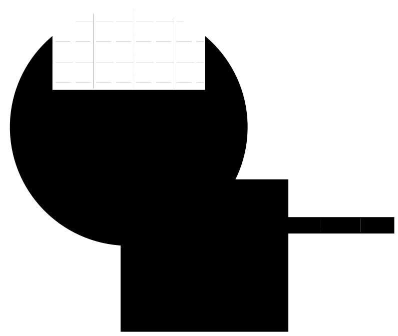

Figure 1: A Simple Construction

(a) Define the following constructions:

i. (3 points) Given two points `min` and `max`, the construction `rect(min,max)` represents a black rectangle where `min` is the lower-left corner, and `max` is the upper-right corner.

```
fun rect ((minx, miny) : point, (maxx, maxy) : point) : constr =
```

Solution:

```
fn (x,y) => xmin <= x andalso x <= xmax andalso
            ymin <= y andalso y <= ymax
```

ii. (3 points) Given two constructions `c1` and `c2`, the construction `intersection(c1,c2)` has, as black points, exactly those points that are colored black by both `c1` and `c2`.

```
fun intersection (c1 : constr, c2 : constr) : constr =
```

Solution:

```
fn p => c1 p andalso c2 p
```

iii. (3 points) Given two constructions `c1` and `c2`, the construction `union(c1,c2)` has, as black points, exactly those points that are colored black by either `c1` or `c2` (or both).

```
fun union (c1 : constr, c2 : constr) : constr =
```

Solution:

```
fn p => c1 p orelse c2 p
```

iv. (2 points) Given a construction `c`, the construction `inverse(c)` has, as black points, exactly those points that are colored white by `c`.

```
fun inverse (c : constr) : constr =
```

Solution:

```
not o c
```

Question 3 [20]: TreeNorm

Recall trees:

```
datatype 'a tree = Empty | Leaf of 'a | Node of 'a tree * 'a tree
```

Often, one uses trees to model sequences, representing a sequence by the leaves of the tree, from left to right. However, one disadvantage of this representation of sequences is that there is no unique representation of the empty sequence. For example, `Empty`, `Node(Empty,Empty)`, `Node(Node(Empty,Empty),Empty)` are all different representations of the empty sequence.

We can solve this problem by restricting attention to normal trees.

A *normal tree* is either

- `Empty`
- a non-empty tree

and that's it!

A *non-empty tree* is either

- `Leaf x`
- `Node(l,r)` where l and r are non-empty trees

and that's it!

There is only one normal empty tree; trees like `Node(Empty,Empty)` are not normal.

The following function normalizes a tree:

```
(* for all t:'a tree, there exists a t' such that
   norm t ==> t' where t' is normal
   and has all the same leaves as t, in the same order *)
fun norm (t : 'a tree) : 'a tree =
    case t of
        Empty => Empty
      | Leaf x => Leaf x
      | Node(l,r) =>
            (case (norm l, norm r) of
                 (Empty, r') => r'
               | (l', Empty) => l'
               | (l', r') => Node(l',r'))
```

You will prove the first part of that spec:

> For all `t`, there exists a `t'` such that `norm t == t'` where `t'` is normal.

The proof is by induction on `t`.

(a) (4 points) Case for `Empty`:
To show:

Solution: there exists a `t'` such that `norm Empty == t'` and `t'` is normal.

Proof:

Solution: Take `t'` to be `Empty`. `norm Empty` steps to `Empty`. `Empty` is normal by definition.

(b) (5 points) Case for `Leaf x`: To show:

Solution: there exists a `t'` such that `norm (Leaf x) == t'` and `t'` is normal.

Proof:

Solution: Take `t'` to be `Leaf x`. `norm (Leaf x)` steps to `Leaf x`. `Leaf x` is non-empty by definition, and therefore normal.

(c) (11 points) Case for `Node(l,r)`: Inductive hypotheses:

Solution: There exists an `l'` such that `norm l == l'` and `l'` is normal.
There exists an `r'` such that `norm r ==> r'` and `r'` is normal.

To show:

Solution: There exists a `t'` such that `norm (Node(l,r)) == t'` and `t'` is normal.

Proof:

Solution:

```
norm (Node(l,r))
== case (norm l , norm r) of ...   step
== case (l' , r') of ...           by IH
```

By the IH, `l'` is either `Empty` or a non-empty tree. If it's `Empty`, the whole expression steps to `r'`, so take `t'` to be `r'`, which is normal by the IH. Otherwise `l'` is non-empty.

By the IH, `r'` is either `Empty` or a non-empty tree. If it's `Empty`, the whole expression steps to `l'`, so take t' to be l', which is normal by the IH.

Otherwise `r'` is non-empty, and the whole expression steps to `Node(l',r')`. So both `l'` and `r'` are non-empty. So take `t'` to be `Node(l',r')` which is is non-empty, and therefore normal.

Disclaimer: this theorem is stated in a style that we used more last semester ("there exists a `t`"), and would be stated a little differently with this semester's tools.

THIS PAGE INTENTIONALLY LEFT BLANK

Question 4 [14]: Analysis

(a) The following function tests whether a predicate holds for all elements of a sequence:

```
fun all (p : 'a -> bool) : ('a Seq.seq) -> bool =
    Seq.mapreduce p true (fn (x,y) => x andalso y)
```

i. (2 points) Assuming `p` has constant work and span on all inputs, give a tight big-O bound for the work of `all p s`, in terms of the length n of `s`.

$W_{\texttt{all}}(n)$ is

Solution: $O(n)$

ii. (2 points) Assuming `p` has constant work and span on all inputs, give a tight big-O bound for the span of `all p s`, in terms of the length n of `s`.

$S_{\texttt{all}}(n)$ is

Solution: $O(\log n)$

(b) The following function tests whether a predicate holds for all elements of a list:

```
fun all_list (p : 'a -> bool) (l : 'a list) : bool =
    case l of
        [] => true
      | x::xs => p x andalso all_list p xs
```

Throughout, assume `p` has constant work and span on all inputs.

i. (3 points) Give a recurrence for the worst-case work of `all_list p l`, in terms of the length n of `l`. Your recurrence should be exact, except you may use constants k_0, k_1, ... to stand for constant numbers of steps of evaluation:

$W_{\texttt{all\_list}}(0) =$

Solution: $k0$

$W_{\texttt{all\_list}}(n) =$

Solution: $k + W_{\texttt{all\_list}}(n-1)$

ii. (2 points) Give a tight big-O bound for the work of `all_list`:

$W_{\texttt{all\_list}}(n)$ is

Solution: $O(n)$

iii. (3 points) Give a tight big-O bound for the span of `all_list p l`, in terms of the length n of `l`. Explain why it is different than the span of `all`.

$S_{\texttt{all\_list}}(n)$ is

Solution: $O(n)$, because it processes the list sequentially.

(c) (2 points) Suppose a function `seqFromList : 'a list -> 'a Seq.seq` that converts a list to a sequence with the same elements in the same order.

Give an example `p` and `l` such that `all_list p l` takes much, much less time than `all p s` does, where `seqFromList l == s`. That is, give an example where `all_list p` is much faster on a list than `all p` is on the corresponding sequence (ignoring the time it takes to convert `l` to `s`).

Solution: `[false,true,true,true,true,..............]` because the `andalso` short-circuits.

Question 5 [16]: Sublist

l is a *sublist* of l' iff l can be obtained by deleting some elements of l':

- `[]` is a sublist of `l` for any `l`
- `x::xs` is a sublist of `y::ys` iff either `x::xs` is a sublist of `ys` (delete `y`) or `x = y` and `xs` is a sublist of `ys` (use `y` to match `x`).

Given two lists l_1 and l_2, a list l is a *longest common sublist* of l_1 and l_2 iff l is a sublist of l_1 and l is a sublist of l_2 and l is at least as long as any other sublist of both l_1 and l_2.

One application of finding a longest common sublist of two lists is to measure DNA similarity: given two strands of DNA, the longest common sublist is a rough estimate of the amount of shared information. For example, a longest common sublist of `AGATT` and `AGTCCAGT` is `AGTT`. `AGAT` is another.

We represent DNA as a `base list`, where

```
datatype base = A | T | C | G
```

Before writing your solution, do the following examples:

(a) (1 point) What is the longest common sublist of `[A,T,G]` and `[T,G]`?

(b) (1 point) What is the longest common sublist of `[C,A,T,G]` and `[C,T,G]`?

(c) (1 point) What is the longest common sublist of `[C,A,T,G]` and `[G,T,G]`?

Question continues on the next page

(d) (11 points) Write a function `lcs` that computes a longest common sublist. Your solution is permitted to take exponential time. You may use the following helper functions, where `longer` returns the longer of two lists, and `baseeq` compares bases for equality:

```
val longer : base list * base list -> base list
val baseeq : base * base -> bool
```

```
    fun lcs (s1 : base list, s2 : base list) : base list =
```

Solution:

```
    fun lcs (s1 : base list, s2 : base list) : base list =
        case (s1,s2) of
            ([],_) => []
          | (_,[]) => []
          | (x::xs, y::ys) =>
                case baseeq(x,y) of
                    true => x :: lcs(xs,ys)
                  | false => longer(lcs(s1,ys),
                                    lcs(xs,s2))

    (* alternative with redundant recursive calls, but that's only
       a constant-factor difference once you memoize *)
    fun lcs' (s1 : base list, s2 : base list) : base list =
        case (s1,s2) of
            ([],_) => []
          | (_,[]) => []
          | (x::xs, y::ys) =>
                longer(case baseeq(x,y) of
                           true => x :: lcs'(xs,ys)
                         | false => lcs'(xs,ys),
                       longer(lcs'(s1,ys),
                              lcs'(xs,s2)))
```

(e) (2 points) Briefly explain how you can use one of the homework problems from this course to obtain a polynomial-time ($O(n)$ or $O(n^2)$ or $O(n^3)$ ...) solution.

Solution: Memoize! There is a lot of duplication of recursive calls.

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

Question 6 [15]: Profiling

Profiling is the process of dynamically tracking some characteristics of a function—such as how long it typically takes to run, how much space it uses, or the number of recursive calls it makes on various arguments.

In this problem, you will use effects and imperative programming to write a higher order function `profile` that produces a profiled version of any function:

> For a function `f`, `profile f` must have the same behavior as `f` on all inputs, except that it additionally prints some profiling information. In particular, it must print out **the total amount of time that has elapsed while running `f` on all the inputs `f` has been called on so far**.

For example, suppose that `f 5` takes 5 seconds and `f 2` takes 6 seconds. Then you should have the following interactions:

```
- val f' = profile f
- f' 5;
Total running time: 5.0 seconds
val it = <value of f 5>
- f' 2;
Total running time: 11.0 seconds
val it = <value of f 2>
```

During this question, you will need to use the type `Time.time`, which is equipped with the following functions:

- `Time.now : unit -> Time.time` which computes a value of type `Time.time` that represents the current time.
- `Time.zeroTime` which represents zero time.
- `Time.+ : Time.time * Time.time -> Time.time` which computes a value of type `Time.time` representing the sum of two times.
- `Time.- : Time.time * Time.time -> Time.time` which computes a value of type `Time.time` representing the difference of two times.
- `Time.toString : Time.time -> string` which computes a nice string representation of a value of type `Time.time`.

(a) (13 points) Complete the function `profile` below as specified.

```
fun profile (f : 'a -> 'b) : 'a -> 'b =
```

Solution:

```
fun profile (f : 'a -> 'b) : ('a -> 'b) =
    let
      val call_times : Time.time ref = ref Time.zeroTime
    in
      fn x =>
          let
            val bef : Time.time = Time.now()
            val result = f x
            val aft : Time.time = Time.now()
            val () = call_times := Time.+ (Time.- (aft,bef) , !call_times)
            val () = print ("Total running time: " ^
                            (Time.toString (!call_times)) ^
                            " seconds\n")
          in
            result
          end
    end
```

Recall that two-argument functions can be represented either by a function whose argument is a pair, or using currying:

```
(* exp : int * int -> int *)
fun exp (e : int, b : int) : int =
    case e of
        0 => 1
      | _ => b * exp (e-1, b)

(* garama_masala_exp : int -> int -> int *)
fun garam_masala_exp (e : int) (b : int) : int =
    case e of
        0 => 1
      | _ => b * garam_masala_exp (e-1) b
```

(b) (2 points) Explain the difference in what is profiled when you call `prof_exp` versus when you call `prof_garama_masala_exp`, as defined by

```
val prof_exp = profile exp
val prof_garama_masala_exp = profile garama_masala_exp
```

Solution: The curried version only profiles the first argument, which immediately returns. The pair version profiles the actual computation, which happens after getting both arguments. A good exercise would be to think about a staged version of the curried version: what would get profiled?

Question 7 [17]: Flattening

All semester, we have assumed that sequence operations provide *nested parallelism*. Thus means that, in code like

```
type 'a matrix = ('a Seq.seq) Seq.seq
fun double_all (m : int matrix) : int matrix =
   Seq.map (Seq.map (fn x => 2 * x)) s
```

there are two sources of parallelism: (1) each element of the sequence computed by `Seq.map` is computed in parallel, and (2) the function supplied as an argument to `Seq.map` may create additional parallel tasks. For example, if the matrix is represented by the sequence of its columns, (1) `Seq.map (fn x => 2 * x)` is applied in parallel to each column and (2) for each column, `(fn x => 2 * x)` is applied in parallel to each entry in that column. Thus, `double_all` has $O(1)$ span for any matrix.

However, many hardware implementations of parallelism allow only *flat parallelism*: a parallel task may not create any additional parallel tasks. We can model this by stipulating that each element of the sequence computed by `Seq.map f` is computed in parallel, but the argument function `f` is executed completely sequentially. In the above example, the inner `(Seq.map (fn x => 2 * x))` would be executed sequentially, so on a matrix of dimensions $n \times n$, the span of `double_all` would be $O(n)$. And similarly for `tabulate`, `reduce`, etc.

For this problem, we assume that `Seq` provides only flat parallelism.

Flattening is a program transformation used to compile nested parallelism to flat parallelism. The key idea is to choose a "flat" representation of types, so that nested parallelism becomes flat parallelism.

In this problem, you will give a flattened implementation of the matrix signature in Figure 2. Matrices are indexed by $(column, row)$, starting at 0. For example, drawing the top-left corner as (0,0), in the matrix

$$\begin{bmatrix} A & B & C \\ D & E & F \end{bmatrix}$$

A is in position (0,0), B is in position (1,0), and F is in position (2,1).

Your task is to find a representation that for which you can implement `map` with $O(1)$ span, using only flat parallelism. For example, suppose you represented `'a matrix` as a sequence of sequences (`'a Seq.seq Seq.seq`), as above, and then implemented

```
fun map f (s : 'a matrix) = Seq.map (Seq.map f) s
```

This does not meet the goal, because this implementation has span proportional to the number of rows, rather than constant.

```
signature MATRIX =
sig
  type 'a matrix (* invariant: dimensions are >= 1 *)

  (* Returns the value of the matrix at the given (col,row) subscript *)
  val sub  : 'a matrix -> (int * int) -> 'a

  (* tabulate f (width, height) returns the matrix with the given dimensions
     such that the value in position (i,j) is f(i,j).
     Here, width means number of columns, and
           height means number of rows. *)
  val tabulate : (int * int -> 'a) -> (int * int) -> 'a matrix

  (* transform each element *)
  val map : ('a -> 'b) -> 'a matrix -> 'b matrix

end
```

Figure 2: Matrix

On the following page, implement the signature `MATRIX`. Your implementation must work for rectangular matrices of dimensions $w \times h$, but to make the time bounds simpler, we state them for the special case of a square $n \times n$ matrix. For higher-order functions, we state the running time for the special case where all function arguments take constant time.

(a) (4 points) Implement `'a matrix`, *as an abstract type*. Hint: use the span for the operations to guide your choice of representation.

(b) (4 points) Implement `sub`, with work $O(1)$ and span $O(1)$.

(c) (5 points) Implement `tabulate`, with work $O(n^2)$ and span $O(1)$.

(d) (4 points) Implement `map`, with work $O(n^2)$ and span $O(1)$.

```
structure FlatMatrix : MATRIX =
struct
```

Solution:

```
type coord = int * int

datatype 'a matrix = M of 'a Seq.seq * int (* number of rows *)
    (* represented by a flat sequence, so

       A B C
       D E F

       is represented by

       (<A,D,B,E,C,F> , 2)

       you need the number of rows for anything indexy (sub, tabulate)

       *)

fun sub (M (s, num_rows)) (c,r) = Seq.nth s (c * num_rows + r)

fun tabulate f (num_cols, num_rows) =
    M (Seq.tabulate (fn i => f (i div num_rows, i mod num_rows))
                    (num_cols * num_rows),
       num_rows)

fun map f (M (s, num_rows)) = M (Seq.map f s, num_rows)
```

```
end
```

THIS PAGE INTENTIONALLY LEFT BLANK

```
signature SEQUENCE =
sig
  type 'a seq

  exception Range

  val length : 'a seq -> int   (* constant work and span *)
  val nth    : 'a seq -> int -> 'a (* constant work and span *)

  (* assuming f has constant work/span,
     tabulate f n has O(n) work and O(1) span *)
  val tabulate : (int -> 'a) -> int -> 'a seq

  val map : ('a -> 'b) -> 'a seq -> 'b seq
  val reduce : (('a * 'a) -> 'a) -> 'a -> 'a seq -> 'a
  val mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b

end
```

Clarifcations given during the exam:

- In Profiling: assume everything is sequential (no parallelism)
- In Profiling: `Time.zeroTime` has type `Time.time`
- In Sublist: note that every list is a sublist of itself
- In Flatting: for `tabulate`, assume that the indices are in bounds