15-150 Fall 2015 Homework 12

Lazy Functional Programming

Released: Wednesday, 2 December 2015
Due: Tuesday, 8 December 2015 at 23:59 EST

Notes

- This is the last homework assignment of the semester. It is a longer assignment, so start early!
- The files you will be handing in are `Lazy.sml`, `Change.sml`, `Frobenius.sml`, `Regular.sml`, `Hamming.sml`, `MemoKnuth.sml`, and `hw12.pdf`.
- You should write your test cases in a separate test structure in each file.
- Supporting files and signatures are in `lib` and `signatures` respectively. Do not modify any files in `lib` or `signatures`. You may use them for reference.
- You may use helper functions for each task, but only if they make your code easier to read and/or more efficient. In addition, you must provide specs and tests (if possible) for each helper function. Failure to do so will result in point deductions.
- You may use code from lecture to help solve the problems, but make sure to cite it!
- Make sure to read the handout carefully.

Quotes

"Ambition is a poor excuse for not having sense enough to be lazy." Milan Kundera

"Periods of wholesome laziness, after days of energetic effort, will wonderfully tone up the mind and body." Grenville Kleiser

"The three chief virtues of a programmer are laziness, impatience and hubris." Larry Wall

"The lazy man vies with the industrious." William Shatner

Background

In class we discussed a datatype whose values can represent lazy lists. The in-class datatype only allowed for infinite lazy lists. In this homework we will use the following datatype definition, which allows for both finite and infinite lazy lists:

```
datatype 'a lazylist = Nil
| Cons of 'a * (unit -> 'a lazylist)
```

We also refer to the function `show : int -> 'a lazylist -> 'a list`, which can be used to extra the first few members of a lazy list:

```
fun show 0 _ = [ ]
|   show n Nil = [ ]
|   show n (Cons(x, xs)) = x :: show (n-1) (xs ( ))
```

In this homework, you will explore how to use lazy lists to represent finite and infinite sets if values drawn from an ordered type of data; for short, ordered sets of data. There are many applications for ordered sets, since we often need to deal with infinite amounts of data in a structured manner, and we can take advantage of the order to enable efficient operations such as finding the smallest member of a set that satisfies some special property.

We say that a value `L` of type `t lazylist` (for some type `t`) is finite iff there is a non-negative integer `N` such that for all $\texttt{n} \geq \texttt{N}$,

```
length (show n L) = N
```

and we say that `L` is infinite iff for all $\texttt{n} \geq 0$,

```
length (show n L) = n
```

For example. if we define

```
fun nats x = Cons(x, fn ( ) => nats(x+1))
```

the expression `nats 42` is infinite

Ordered Types

An ordered type is an equality type `t`, together with a comparison function that is linear. The ML family of "equality types" includes any type built from basic types like `int`, `string`, `unit`, `bool`, using tupling (*) and `list`. A comparison function `compare : t * t -> order` satisfies the properties given when we talked about sorting lists and trees. The linearity property is simply that `compare(x,y) = EQUAL` if and only if `(x = y) = true`. Since `t` is an equality type, the expression `x = y` is legal and has type `bool`.

Examples of ordered types:

- `string`, with `compare` defined to be `String.compare`
- `string`, with `compare` given by:

```
fun compare(x, y) =
  case Int.compare(String.size x, String.size y) of
                EQUAL => String.compare(x, y)
              | LESS => LESS
              | GREATER => GREATER
```

  Here of course `String.size` is used to get the number of characters in a string, so this `compare` function is like a length-sensitive variant of `String.compare`.

- `int`, with `compare` defined to be `Int.compare`. This is the integers, with the usual (increasing) order.

We will use the following signatures for ordered types:

```
signature ORD =
sig
eqtype t
   val compare : t * t -> order (* a linear comparison function  *)
end
```

We will refer to the following structures that implement ordered types of strings and integers.

```
structure Strings : ORD =
struct
  type t = string
  fun compare(x,y) =
    case Int.compare(String.size x, String.size y) of
      EQUAL => String.compare(x,y)
    | other => other
end

structure Strings2 : ORD =
struct
  type t = string
  val compare = String.compare
end

structure Ints : ORD =
struct
  type t = int
  val compare = Int.compare
end
```

Given a comparison function `compare` for type `t`, we say that a function

```
g : t * t -> t
```

is compare-monotone iff for all values `x1, x2, y` of type `t`,

```
  compare (x1, x2) = LESS implies compare (g(x1, y), g(x2, y)) = LESS
and
  compare (x1, x2) = LESS implies compare (g(y, x1), g(y, x2)) = LESS
```

NOTE: The string concatenation function

```
(op ^) : string * string -> string
```

is `compare`-monotone, for each of the `compare` functions above, i.e. `Strings.compare` and `Strings2.compare`.

Ordered Sets

An ordered set consists of an ordered type (as above) together with functions for building and manipulating sets, as specified in the signature `ORDSET`, including ways to test for membership and to extract the smallest (finitely many) elements. In our representation we will ensure that a set is represented by a lazy list of values in increasing order, with no repeated elements.

In describing ordered sets we use standard set-notation like $\{x_1, ..., x_n\}$ (also allowing infinite cases), and we usually list the elements in increasing order with respect to the relevant comparison function.

Examples:

- The set of strings `{"", "foo", "foofoo", . . .}` of all strings that consist of `"foo"` concatenated some finite number of times, with `Strings2` as the ordered type. The first 4 elements, in order, form the list
  `["", "foo", "foofoo", "foofoofoo"]`

- The set of strings `{"", "foo", "foofoo", . . .}` of all strings that consist of `"foo"` concatenated some finite number of times, with `Strings` as the ordered type. The first 4 elements, in order, again form the list
  `["", "foo", "foofoo", "foofoofoo"]`

- The set of strings `{"bar", "barf", "car"}`, with `Strings2` as the ordered type. The first three elements, in order, are:
  `["bar", "barf", "car"]`

- The set of strings `{"bar", "barf", "car"}`, with `Strings` as the ordered type. The first three elements, in order, are:
  `["bar", "car", "barf"]`

- The set of integers `{42, 43, 44, . . .}`, with `Ints` as the ordered type. The first three elements are
  `[42, 43, 44]`

- The set of even positive integers, with `Ints` as the ordered type. The first three elements are:
  `[2, 4, 6]`

Here is the signature (also in `signatures/ORDSET.sig`) that specifies an ordered set. In question 1 it will be your job to implement a functor that creates a structure ascribing to `ORDSET` given an ordered type.

```
signature ORDSET =
sig
  eqtype t
  val compare : t * t -> order (* a linear comparison function *)
  type set (* ordered sets of values of type t *)
  val empty : set (* the empty set *)
  val singleton : t -> set (* singleton(x) represents {x} *)

(* first S returns NONE if S is empty and SOME(x) if S is Cons(x, xs) *)
  val first : set -> t option

(* rest S returns Nil if S is empty and xs() if S is Cons(x, xs) *)
  val rest : set -> set

(* REQUIRES f is total and compare-monotone *)
(* ENSURES (enum f) represents { f(n) | n >= 0 } *)
  val enum : (int -> t) -> set

(* union (X, Y) represents { z | z in X or z in Y }, i.e. X U Y *)
  val union : set * set -> set

(* intersect (X, Y) represents {z | z in X and z in Y } *)
  val intersect : set * set -> set

(* REQUIRES g is total and compare-monotone
(* ENSURES pairwith g (X, Y) represents { g(x,y) | x in X and y in Y } *)
  val pairwith : (t * t -> t) -> set * set -> set

(* REQUIRES g is total and compare-monotone,
     z is compare-LESS than or EQUAL to every value in X,
     z is an identity element for g  *)
(* ENSURES closure g z X represents the smallest set S
     that contains z and X and is closed under application of g,
     i.e. S = {z} U { g(x, y) | x in X and y in S }      *)
```

```
  val closure : (t * t -> t) -> t -> set -> set

(* member X x = true if x is in X, false otherwise *)
  val member : set -> t -> bool


(* show n X = list of the n compare-least members of X *)
  val show : int -> set -> t list

(* REQUIRES f is total, X is finite or infinitely many x in X satisfy f *)
(* ENSURES (filter f X) represents {x in X | f x = true } *)
  val filter : (t -> bool) -> set -> set
end
```

QUESTION 1

> "Inspiration is a guest that does not willingly visit the lazy."
> Pyotr Ilyich Tchaikovsky

Task 1.1 (20 pts). In `Lazy.sml` complete the functor definition for `Lazy` that implements ordered sets as finite or infinite lazy lists enumerated in increasing order, with no repeated elements. You can include any helper functions that you need, but you must also indicate clearly their types and specifications, and remember that only the data mentioned in the signature will be available to users of the functor.

Here we only show a template for the functor definition, to remind you that the functor `Lazy` is intended to be applicable to any ordered type (any structure `S` with signature `ORD`), and produces a signature that ascribes to the signature `ORDSET`, transparently.

```
functor Lazy(S : ORD) : ORDSET =
struct
  type t = S.t
  val compare = S.compare
  datatype set = Nil | Cons of t * (unit -> set)
  val empty = Nil
```

```
  (* the rest of your code goes here *)
end
```

For testing, you may use the provided ordered type structures `Ints, Strings,` and `Strings2` (same from above). Some test cases have been provided for you in the `Lazy.sml` file. You must add additional test cases as well.

QUESTION 2

> "Never put off until tomorrow what you can do the day after tomorrow." Mark Twain

Task 2.1 (10 pts). In `Change.sml` use your `Lazy` functor from question 1 to write a non-recursive ML function

```
change : int list * int -> bool
```

such that for all lists of positive integers `L` and all positive integers `n`, `change (L, n)` evaluates to `true` if it is possible to make change for `n` using (unlimited quantities of) coins drawn from the list `L`. `change (L, n)` evaluates to `false`, otherwise. For example:

```
change ([6, 9], 21) = true
change ([9, 6], 22) = false
```

You should convert an integer list into the ordered set of all integers that are obtainable from it using addition, then check for membership.

Task 2.2 (10 pts). A positive integer n is a "Hamming number" if it is only divisible by 2, 3 and/or 5. For example, 20 is a Hamming number ($20 = 225$), but 42 is not. The set of Hamming numbers contains every number that is a power of 2, every number that is a power of 3, and every number that is a power of 5. Also note that this set is closed under multiplication.

In `Hamming.sml` use your `Lazy` functor to write a non-recursive ML function

```
ham : int -> int list
```

such that for all $n \geq 0$, ham n evaluates to the list consisting of the first n Hamming numbers, in increasing order, without repetitions. For example:

```
ham 10 = [1,2,3,4,5,6,8,9,10,12]
```

You should build the ordered set of all Hamming numbers, then extract the smallest n members.

QUESTION 3

"Laziness is the first step towards efficiency." Patrick Bennett

A certain fast-food restaurant chain sells Chicken McNuggets in boxes of 6, 9, or 20. Obviously one could by exactly 15 McNuggets by buying a box of 6 and a box of 9. But it's also easy to see that you cannot buy exactly 17 McNuggets. It's not so clear whether or not you could obtain exactly 53 McNuggets. (You can't.)

The McNuggets Problem asks, when given a list of box sizes, to find the largest number for which it is impossible to purchase exactly that number of McNuggets. We generalize to allow for the possibility that at some future time the fast-food chain might change box sizes, for example to begin selling only in boxes of 7, 11, and 17.

Possibly since mathematicians and computer scientists love their fastfood, a lot is known about this problem. It's also known as the Postage Stamp Problem, or the Frobenius Coin Problem, which asks for the largest monetary amount that cannot be obtained using only coins of specified denominations.

Suppose we have boxes of positive integer sizes given by a list $[a_1, ..., a_n]$. If the greatest common divisor of $a_1, ..., a_n$ is not 1, there is no largest inexpressible integer. (Every multiple of the g.c.d. would be inexpressible.) If the g.c.d. $a_1, ..., a_n$ is 1, the following facts are relevant:

- If $n = 1$, so $a_1 = 1$, every positive integer is expressible.
- If $n > 1$ and the g.c.d of $a_1, ..., a_n$ is 1, there is a largest inexpressible positive integer and it is equal to the smallest positive integer N such that N is inexpressible and each of $N+1, N+2, ..., N+m$ is expressible, where m is the smallest box size.

You may, if you wish, consult the Wikipedia article on the Frobenius Coin Problem, but if you use any information derived from there you should be careful! Statements appearing in a Wikipedia article may not be as clear as they should be and might mislead you.

Task 3.1 (15 pts). In `Frobenus.sml`, use your `Lazy` functor to write an ML function

```
frobenius : int list -> int option
```

such that when `L` is a non-empty list of positive integers,

```
frobenius L = SOME N,
         where N is the largest positive integer not expressible using L,
         if there is one.
      frobenius L = NONE otherwise.
```

You should check the g.c.d. of the list of box sizes, and if this is 1, build the ordered set of all positive integers expressible as combinations of these sizes; then look for the smallest N with the properties mentioned above.

You may use the following g.c.d function in your code if you would like:

```
fun gcd(m,n) = case Int.compare(m,n) of
                                LESS => gcd(m, n-m)
                              |  EQUAL => m
                              |  GREATER => gcd(m-n, n)
```

Task 3.2 (5 pts). Using your `frobenius` function, find the values of:

```
(a)   frobenius [6, 9, 20]
(b)   frobenius [6, 10, 20]
(c)   frobenius [4, 6, 9, 11]
(d)   frobenius [15, 14, 13]
(e)   frobenius [19, 20]
```

It's not sufficient for you too merely state the values - your function must actually produce the correct answer for each case.

Please put these test cases in your `Frobenius.sml` in a separate testing structure. Make sure to label each test case so that they are easy to find.

QUESTION 4

> "I will always choose a lazy person to do a difficult job, because he will find an easy way to do it." Bill Gates

Here is a datatype for regular expressions, similar to the one discussed in class and on an earlier homework. Important changes to note: we include `Both`, which corresponds to intersection of regular languages; and we use `Char : string -> regex` rather than `Char : char -> regex`, mainly to avoid dealing with character syntax (and hashtags!).

```
datatype regex = Zero | One | Char of string
                           | Plus of regex * regex
                           | Times of regex * regex
                           | Star of regex
                           | Both of regex * regex
```

The language of a regular expression `R : regexp` is the set of strings (`R`) given by the following clauses:

$\mathcal{L}(\texttt{Zero}) = \emptyset$
$\mathcal{L}(\texttt{One}) = \{\texttt{""}\}$
$\mathcal{L}(\texttt{Char s}) = \{\texttt{s}\}$
$\mathcal{L}(\texttt{Plus(R1,R2)}) = \mathcal{L}(\texttt{R1}) \cup \mathcal{L}(\texttt{R2}) = \{s \mid s \in \mathcal{L}(\texttt{R1}) \vee s \in \mathcal{L}(\texttt{R2})\}$
$\mathcal{L}(\texttt{Times(R1, R2)}) = \{\texttt{s1\^{}s2} \mid \texttt{s1} \in \mathcal{L}(\texttt{R1}) \wedge \texttt{s2} \in \mathcal{L}(\texttt{R2})\}$
$\mathcal{L}(\texttt{Star R}) = \{\texttt{""}\} \cup \mathcal{L}(\texttt{R}) \cup \{\texttt{s1\^{}s2} \mid \texttt{s1} \in \mathcal{L}(\texttt{R}), \texttt{s2} \in \mathcal{L}(\texttt{Star R})\}$
$\mathcal{L}(\texttt{Both(R1,R2)}) = \mathcal{L}(\texttt{R1}) \cap \mathcal{L}(\texttt{R2}) = \{s \mid s \in \mathcal{L}(\texttt{R1}) \wedge s \in \mathcal{L}(\texttt{R2})\}$

Note that `""` is the empty string, `^` is string concatenation, and `""` is the identity element for concatenation, i.e. `("" ^ s) = s = (s ^ "")` for all strings `s`. Concatenation is associative, but not commutative.

The clause for $\mathcal{L}$(`Star R`) is recursive, and says that the language of `Star R` is the smallest set of strings that contains the empty string `""` and the language of `R`, and is closed under strings concatenation.
EXAMPLES

- $\mathcal{L}$(`Char "foo"`) = {`"foo"`}
- $\mathcal{L}$(`Plus(Char "foo", Char "bar")`) = {`"foo"`, `"bar"`}
- $\mathcal{L}$(`Times(Char "foo", Char "bar")`) = {`"foobar"`}

- $\mathcal{L}$(`Star(Char "foo")`) = {`""`, `"foo"`, `"foofoo"`, ...}
- $\mathcal{L}$(`Star One`) = {`""`}
- $\mathcal{L}$(`Star(Star One)`) = {`""`}

Consider the following signature `REGEX`:

```
signature REGEX = sig
  datatype regex = Zero | One | Char of string
                 | Plus of regex * regex
                 | Times of regex * regex
                 | Star of regex
                 | Both of regex * regex
  type set
  val language : regex -> set
  val member : set -> string -> bool
  val show : int -> set -> string list
end
```

Task 4.1 (10 pts). In `Regular.sml` use your `Lazy` functor to complete the following structure that implements regular languages as ordered sets of strings.

```
structure Regular : REGEX =
struct
  datatype regex = Zero | One | Char of string
                 | Plus of regex * regex
                 | Times of regex * regex
                 | Star of regex
                 | Both of regex * regex

  structure S = Lazy(Strings)
  open S

  (* your code goes here *)
end
```

HINT: This problem is (perhaps surprisingly) easy to solve using the lazy implementation of ordered sets of strings. If you weren't able to develop correct solutions to the `Lazy` functor earlier, you can still get credit here if you use its components properly.

Task 4.2 (5 pts). Using your `Regular` structure from the previous question, test your language function by using the test functor `RegExTest` (that we supply). Run `test` on each of the 18 provided test cases and show us what results your code produces in your PDF.

Task 4.3 (5 pts). Now cut-and-paste your code into the following structure definition, which is exactly as above except that it builds on `Strings2` rather than `Strings`:

```
structure Regular : REGEX =
struct
   datatype regex = Zero | One | Char of string
                  | Plus of regex * regex
                  | Times of regex * regex
                  | Star of regex
                  | Both of regex * regex

  structure S = Lazy(Strings2)
  open S

  (* your code goes here *)
end
```

Test the language function from this `Regular2` structure by running the same test file (that we supply). In your PDF, show us what results your code produces for each of the provided test cases. Explain why the results differ!

QUESTION 5

> "Progress isn't made by early risers. It's made by lazy men trying to find easier ways to do something." Robert A. Heinlein

In class we explored the minimax algorithm, and we introduced mutually recursive functions `F, G : state -> int` for outcome assessment from the perspective of a Maxie player and a Minnie player, respectively. We defined a functor `Knuth` that takes a structure `Game` of signature `GAME` and builds a structure `Knuth(Game)` with the signature `PLAYER`. When we use these functions for a game with lots of branching and long sequences of legal moves, it isn't feasible to evaluate `F s` for all states. For example, with the simple Nim game `F 40` takes a very long time! Luckily there is an easy way to exploit references and effects that will speed things up, even in games where there isn't a "genius strategy." The idea is to use memoization.

Suppose we have a structure Table that implements the signature

```
signature TABLE =
sig
type (''a, 'b) table
val empty : unit -> (''a, 'b) table
val insert : ''a * 'b -> (''a, 'b) table -> (''a, 'b) table
val lookup : ''a -> (''a, 'b) table -> 'b option
end
```

The type constructor `table` is parameterized by a pair of types, one of which stands for an equality type. Suppose we have a game for which the `state` type is an equality type. A value of type `(state, int) table` represents a collection of state-outcome entries, with at most one entry for each state. Such a table can be used to summarize the results of a number of minimax calculations. The value `empty` represents the empty collection. The `insert` and `lookup` functions behave in the obvious way; in particular `lookup s T` returns `NONE` if there is no entry for state s in the table `T`, and returns `SOME v` if the table contains the pair `(s,v)`.

We can build a memoizing version of `F` that uses a ref cell to store a table of minimax results and looks up the state in this table first, and when applied to a state for which there is no entry in the current table, computes the result and stores it. When applied to a state for which the table already has an entry, the memoizing function "remembers" the result and returns it.

The effect of a call to this memoizing function can thus help speed up later calls.

This is only a partial description of what you should do; there are several degrees of freedom in how you actually go about the task! Obviously we allow you to use `ref` types, the function `ref : 'a -> 'a ref` for creating a fresh cell initialized with a given value, the function `! : 'a ref -> 'a` for getting the contents of a cell, the infix assignment operator `:=`, and `;` for sequential evaluation of expressions.

Task 5.1 (15 pts). Complete the body of the functor called `Memo_Knuth` that implements the minimax algorithm using memoization, as described above, given structures `Table` and `Game` with the relevant signatures.

Your answer should fit into the following template:

```
functor Memo_Knuth (Table : TABLE) (Game : GAME) : PLAYER =
struct
  structure Game = Game

  (* your code goes here *)

end
```

Use the Minimax algorithm code from lecture notes to help you with this task.

Task 5.2 (5 pts). Using your `Memo_Knuth` functor and a suitable `Table` structure, calculate the best move for Maxie from an initial state of 100 matchsticks. You can use the following syntax to work with the curried functor, assuming you have already defined structures `Table` and `Nim` with signatures `TABLE` and `GAME`:

```
structure S = Memo_Knuth(Table)(Nim);
S.Player 100
```

Put your answer in your PDF.