COMP 212: Computer Science II
Homework 9: Functionals and streams

This assignment is all about working with functionals and streams. In the written work, you will investigate two functionals: the list functional `foldl`, which is comparable to the `foldr` functional that we have discussed in class, and the stream functional `unfold`. `foldl` is a functional provided by most functional languages; your job is to describe it as it is used to encapsulate certain recursive definitions, just like `foldr`, and to prove that your description is correct. This gives you more practice with using list functionals and writing proofs. `unfold` plays a role for streams analogous to `foldr` for lists, and you will investigate which stream functionals can be implemented using `unfold`, just like you saw how most list functionals can be implemented using `foldr`. You will need to work through the coding project of this assignment before doing the written work for `unfold`.

You will implement streams with a moderately lazy `datatype` definition, rather than the one described in *MLWP* and that we went over in detail in class. But always keep in mind the main idea when examining your code; we use a function of type `unit` $\to \alpha$ to represent a suspended computation of type α. For the stream type described in *MLWP* and in class, we used a function of type `unit` $\to$ `stream` as the second argument to a `Cons` constructor to represent a suspended computation of the tail of the stream. In this assignment, you will use a function of type `unit` $\to \alpha *$ `stream` as an argument to a `Cons` constructor to represent a suspended computation of both the head and tail of the stream.

1. Written problems (10 points)

Problem 1. *Remember that the foldr (fold-right) functional has the following two equivalent specifications:*

- *foldr g b xs* $= f(xs)$*, where*
$$f([]) = b \qquad f(y :: ys) = g(y, f(ys))$$
- *foldr g b* $[x_0, \ldots, x_{n-1}] = g(x_0, g(x_1, g(\ldots, g(x_{n-1}, b)\ldots)))$*; when* $n = 0$*, this is just notation for* b*.*

We can prove that these two specifications are equivalent by showing that if f *is defined as in the first item, then for all* $n \geq 0$ *and all lists* $[x_0, \ldots, x_{n-1}]$, $f([x_0, \ldots, x_{n-1}]) = g(x_0, g(x_1, g(\ldots, g(x_{n-1}, b)\ldots)))$ *by induction on* n*: if* $n = 0$ *then*

$$f([x_0, \ldots, x_{n-1}]) = f([]) = b = g(x_0, g(x_1, g(\ldots, g(x_{n-1}, b)\ldots)))$$

and when $n > 0$*, then*

$$f([x_0, \ldots, x_{n-1}]) = g(x_0, f([x_1, \ldots, x_{n-1}])) = g(x_0, g(x_1, g(x_2, g(\ldots, g(x_{n-1}, b)\ldots))),$$

with the last equality following from the induction hypothesis (which hasn't been stated for this sketch, but you should be able to fill in).

SML (and most any functional language) also has a foldl (fold-left) functional; one of its specifications is

$$foldl\, g\, b\, [x_0, \ldots, x_{n-1}] = g(x_{n-1}, g(x_{n-2}, g(\ldots, g(x_0, b)\ldots)))$$

(again, when $n = 0$*, this is just notation for* b*). Complete the following to give an equivalent specification for foldl: foldl g b xs* $= f(xs)$*, where .... The "..." is to be filled in with a recursive description of* f*, or (more likely) a description of* f *in terms of a recursive auxiliary function. Hint:*

whereas for foldr, we think of b as a base case and g as a step function, for foldl, we think of g as a function that accumulates a result in its second argument, and b as an initial value of that accumulator. Once you have described f, prove that for all $n \geq 0$ and all lists $[x_0, \ldots, x_{n-1}]$, $f([x_0, \ldots, x_{n-1}]) = g(x_{n-1}, g(x_{n-2}, g(\ldots, g(x_0, b)\ldots)))$.

PROBLEM 2. *One of the functions you must implement in the coding problems is* `unfold`*, a function that is used to create a stream. See the description in the next section before continuing this problem. Some of the functions that you implement in the coding problems can be implemented using* `unfold` *and some cannot (just as many list functions can be implemented using* `foldr` *and* `foldl`*, but some cannot). Identify which functions other than* `cons`*,* `hd`*,* `tl`*, and* `unfold` *that have a stream range can be implemented with* `unfold` *and which cannot. For those that can, write down the implementation (you may use it in your code, too, but it is not necessary). Your implementations will probably make use of* `cons`*,* `hd` *and/or* `tl`*; the step functions must be non-recursive, and must not refer to the function being defined. Can you identify any common feature(s) among the functions that can be implemented with* `unfold` *that is not present among the functions that cannot be implemented with* `unfold`*?*

2. Coding problems (10 points)

For the programming part of this assignment, you will implement a module for potentially-infinite lists as described in *MLWP* pp. 191–204. Paulson calls these *sequences*, but we will use the term *streams*. The way to think about streams (regardless of how they are implemented) is as lists that are possibly infinitely long. We will use the notation $\langle a_0, a_1, \ldots \rangle$ to denote a stream whose n-th element is a_n (0-indexing as always). Streams may be finite, such as $\langle 2, 3, 5, 7 \rangle$, the stream of primes that are < 10, or they may be infinite, such as $\langle 2, 3, 5, 7, 11, 13, \ldots \rangle$, the stream of all prime numbers. Although a finite stream looks like an ordinary list, do not confuse the two: the stream type is different than the list type, and so the finite stream $\langle 2, 3, 5, 7 \rangle$ is not the same as the list $[2, 3, 5, 7]$.

You will use the following `datatype` definition, as per Exercise 5.25:

```
datatype 'a stream = Nil
                   | Cons of unit -> 'a * 'a stream
```

The functions that you must implement are for the most part the ones described in *MLWP*, though with some changes. The main changes to note:

- `take` returns a stream, not a list.
- `interleave` is called `merge`.
- `append` is an ordinary function, not an operator.
- `iterates` has a different type.

Here are a few comments on some of the functions and values you must implement:

- `unfold` creates a stream by making use of a step function and a seed. Intuitively, when the step function is applied to the seed, it returns the next element of the result stream, and the next seed to use in the step function (which you can then use to get the next element of the stream, and another seed, etc.). For example, using `unfold`, one can give a clean definition of the stream of non-negative even integers by using $f(s) = (s, s + 2)$ as the step function and 0 as the initial seed. Since $f(0) = (0, 2)$, 0 is the first element of the result stream, and 2 is the next seed. Since $f(2) = (2, 4)$, 2 is the next element of the result stream, and 4 is the next seed. Since $f(4) = (4, 6)$, 4 is the next element of the result stream, and 6 is the

next seed. In other words, the stream of non-negative even integers is (almost) defined by

$$\texttt{unfold}\,(\underline{\texttt{fn}}\ \texttt{s => (s, s + 2)})\,0.$$

The one complication is that as described, the result stream is necessarily infinite. To allow for the result stream to be possibly finite, the step function returns an `option` value. `NONE` indicates that the stream is terminated, whereas a `SOME` value consists of an element of the stream and the next seed value. So the (infinite) stream of non-negative even integers would in fact be defined by

$$\texttt{unfold}\,(\underline{\texttt{fn}}\ \texttt{s => SOME (s, s + 2)})\,0$$

whereas the stream of non-negative integers from 0 to 10 would be defined by

$$\texttt{unfold}\,(\underline{\texttt{fn}}\ \texttt{s => }\underline{\texttt{if}}\ \texttt{s = 11}\ \underline{\texttt{then}}\ \texttt{NONE}\ \underline{\texttt{else}}\ \texttt{SOME(s, s+1)})\,0.$$

- `iterates` is similar to the corresponding function in *MLWP*, but adds a wrinkle. The function in *MLWP* takes a one-parameter function f and a starting value a_0, and returns the stream $\langle a_0, f(a_0), f(f(a_0)), f(f(f(a_0))), \dots \rangle$. The function you must implement takes a two-parameter function f and a stream of parameters $as = \langle a_0, a_1, \dots \rangle$, and returns the stream $\langle a_0, f(a_0, a_1), f(f(a_0, a_1), a_2), f(f(f(a_0, a_1), a_2), a_3), \dots \rangle$. Notice that if you "ignore" the tail of as and the second argument of f, this looks like the result of the *MLWP* `iterates` function, and in fact, the one you are to implement generalizes that one. For example, if $f(x, y) = x + y$, then

$$\begin{aligned}\texttt{iterates}\, f\, \langle 0, 1, 2, 3, \dots \rangle &= \langle 0, f(0, 1), f(f(0, 1), 2), f(f(f(0, 1), 2), 3), \dots \rangle \\ &= \langle 0, 0 + 1, (0 + 1) + 2, ((0 + 1) + 2) + 3, \dots \rangle \\ &= \langle 0, 1, 3, 6, \dots \rangle.\end{aligned}$$

  Here is a hint about one possible implementation. If you think of the items in the result stream as being computed one-by-one as items in the argument stream are read, then here is how you might think about the computation imperatively:
  - Pull the first item x off the argument stream; this is the item that is produced for the (head of the) result.
  - Pull the next item y off of the argument stream.
  - Compute $f(x, y)$, and push the value back onto the head of the argument stream.
  - Repeat.

  Although `iterates` can be given a very short definition, I found it to be one of the more challenging functions in this project.
- `primes` is the sequence of all prime numbers, in order. Implement this using the Sieve of Eratosthenes, as described in *MLWP*. For full credit, implement `primes` using `unfold` to the extent possible.

3. Going further

You are not required to do any of the tasks described here; of course, I'll be happy to discuss them with you.

The `unfold` operator plays a role in stream programming that is dual to the role of `fold` in list programming. You can think of `fold` as saying how to *consume* a list to *produce* a value. You can think of `unfold` as saying how to *produce* a stream by *consuming* a value. There is a fairly deep mathematical connection here.

You can implement the *MLWP* `iterates` function in terms of the one you write in this project. You might like to think about how.

Computing the terms of the infinite series that is used to define e (see Paulson, Exercise 5.32) is a nice exercise. Even nicer is to compute the stream of terms using `unfold` without using `Math.pow`, and ensuring that the number of multiplications that is performed in computing the n-th element is linear in n.

We have seen that *foldr* (and *foldl* from Problem 1) are functionals that can be used to define functions that are specified by specific forms of list recursions. Is there a functional that can be used to define any function that can be defined by recursion, regardless of the form? That is, is there a functional Y such that if $f(x)$ is defined recursively, then we can define f by $f = Y(f')$ for some (non-recursive) f'? Surprisingly, the answer is yes! Consider the funtcional Y defined by

$$Y\, f\, x = f\,(Y\, f)\, x.$$

You can define the functional Y in SML (really, any higher-order language), and it will be assigned the type $((\alpha \to \beta) \to (\alpha \to \beta)) \to (\alpha \to \beta)$. Here is what is going on. Notice that an equivalent definition for Y is $Y\, f = f\,(Y\, f)$ (although if you use this definition in SML, you will run into non-termination problems because of evaluation order). Notice that $f : (\alpha \to \beta) \to (\alpha \to \beta)$; that is, f maps a function to a function of the same type. Suppose $g = Y\, f$; then $g = Y\, f = f\,(Y\, f) = f\, g$. In other words, g is a *fixed point* of f—a value that f maps to itself. Y is called a *fixed point combinator*, because, given f, Y returns a fixed point of f. Why does this matter? Consider the function $f =$ **fn** h => **fn** n => **if** $n = 0$ **then** 1 **else** $n{*}h(n{-}1)$. Suppose g is a fixed point of f; then $g = f\, g$, so

$$g(n) = (f\, g)(n) = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n{*}g(n{-}1).$$

Hopefully you recognize this as the usual recursive definition of the factorial function. That is, the factorial function is a fixed point of f, and so we can define the factorial function by

$$fact(n) = Y(\textbf{fn } h \Rightarrow \textbf{fn } n \Rightarrow \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n{*}h(n{-}1))(n).$$

As a nice exercise, write down some other simple recursive functions, and see if you can come up with equivalent definitions using the Y combinator.

4. Code distribution, submission, and grading criteria

The code distribution contains the usual skeleton and testing code.

You will submit `hw9.pdf` (your written solutions) and `hw9.sml` (your code solutions).

When grading, we will look for the following:

Written work: Complete, correct, well-written proof for Problem 1 that uses induction appropriately; correct analysis of which stream functions can be implemented with `unfold`, and why.

Coding: Correct and well-written implementations of all functions. As specified above or in comments in the skeleon file, for maximum credit, some functions must be implemented in certain ways.