15-150 Fall 2013
Lab 13

17 Apr 2013

1 Introduction

This lab will give you some practice with lazy programming, which you saw in lecture on Thursday. With lazy programming, we can delay computation of values until we actually need them.

1.1 Getting Started

Update your clone of the `git` repository to get the files for this week's lab as usual by running

```
git pull
```

from the top level directory (probably named `15150`).

1.2 Methodology

You should practice writing requires and ensures specifications, and tests on the functions you write on this assignment. In particular, every function you write should have both specs and tests.

2 I'm Too Lazy To Come Up With a Title

In lecture we covered *lazy lists* - potentially infinite lists where we don't compute an element until we need it. In this lab we will revisit this datatype and write a few more functions that compute results from lazy datatypes.

Here is the datatype for lazy lists:

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

That is, a list can be either empty or a pair containing the first element of the list and a function we can call to get the rest of the list. Because there's a `Nil` constructor it's possible

to create finite lists, but we can also create infinite lists by passing in an appropriate function to the `Cons` constructor.

For example, the function `zeros` from lecture

```
fun zeros () = Cons (0, zeros)
```

returns a lazy list containing infinitely many 0's.

In contrast, this function

```
fun decreasing 0 = Nil
  | decreasing n = Cons(n, fn () => decreasing(n-1))
```

returns a finite lazy list consisting of the integers `n`, ..., 1 in decreasing order (assuming `n` > 0; if `n` $= 0$, the function returns `Nil`; if `n` < 0, the function returns an infinite lazy list).

Task 2.1 (**Warmup**) Write a function `lazy_append : 'a lazylist * 'a lazylist -> 'a lazylist`. This function expects two lazy lists and returns a lazy list consisting of all the elements of the first lazy list followed by all the elements of the second lazy list. The lazy lists involved may be finite or infinite. Please note the type of this function is subtly different than what was discussed in lecture.

In examining lazy lists produced by your code, you may find the following function useful.

```
(* take(L, n) returns the first n elements of L, now as a regular
  list, unless L has fewer than n elements, in which case take raises Fail.
*)
fun take (L : 'a lazylist, 0 : int) : 'a list = nil
  | take (Nil, _) = raise Fail "tried to take from a Nil lazylist"
  | take (Cons(x, f), n) = x::take(f(), n-1)
```

As you might guess, it's tricky to take the length of a potentially-infinite list - if we do it naively you can expect to spend a very long time walking the list! To get around this, we can use a lazy representation of natural numbers to store the result of our length function:

```
datatype lazynat = Zero | Succ of unit -> lazynat
```

This is another version of the Peano natural numbers This definition means a natural number can be either zero or one plus some other number. The only difference is that now we compute "the other number" lazily. Here are some examples of lazy nats:

```
(* returns the lazy representation of infinity *)
fun infinity () = Succ infinity

(* This means 2 + infinity, which is still infinity *)
val silly_infinity = Succ (fn () => Succ (fn () => silly_infinity))

(* Lazy representation of 2 *)
val two = Succ (fn () => Succ (fn () => Zero))
```

Task 2.2 Now write a function `lazy_length : 'a lazylist -> lazynat` that computes the length of a lazy list. This function must terminate even if the list is infinite.

Task 2.3 It's kind of boring if all we can do is take the length of a list, so let's consider some other operations on lazy nats.

Why might it be a bad idea to try and write a function `lazy_to_int` that converts a lazy nat to an int?

Task 2.4 Instead, write a comparison function, `lazy_cmp: lazynat -> int -> order` that returns whether the given lazy nat is less than, equal to or greater than the given integer. Note that only one argument is lazy. This function must terminate even if the given nat is infinite. You may assume the integer argument is nonnegative.

Task 2.5 For the sake of completeness, write a function `int_to_lazy : int -> lazynat` that converts an integer to a lazy nat. You may assume the argument is nonnegative.

Have the TAs check your work before proceeding!

3 Games

Once you have finished the section on lazy lists and a TA has checked your work, spend the remainder of the lab working on homework 11.