15-150 Spring 2012
Lab 14

April 25, 2012

1 Introduction

This lab will give you some practice with writing imperative code, using mutable linked lists.

1.1 Getting Started

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

```
git pull
```

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

1.2 Methodology

You must use the five step methodology for writing functions for every function you write on this assignment. In particular, every function you write should have a purpose and tests.

2 Linked Lists

In lecture, we discussed the difference between persistent and ephemeral data structures. Persistent data structures, the purely functional data we have studied all semester, provide *more functionality*: they can have many futures. On the other hand, ephemeral data structures admit *more implementations*, in particular imperative ones. Imperative implementations can sometimes be more time- and space-efficient, and are a good tool to have in your toolbox—though in most cases purely functional data structures are the right choice. For example, one of the advantages of an imperative implementation is that you can do *in-place updates*: rather than creating a new copy of a value, you modify the original one. This gives you more direct control over what memory gets allocated, which is sometimes important—though, again, in the majority of cases, letting the ML run-time deal with this issue is the right thing to do.

In this lab, we will look at an ephemeral version of lists, which are commonly called *linked lists*.

```
datatype 'a cell = Nil
                 | Cons of 'a * 'a llist
withtype 'a llist = ('a cell) ref
```

The `withtype` syntax means that the `datatype` and `type` definition are mutually defined; this allows `llist` to be used to describe the type of `Cons`, and we will also use it below.

An `llist` is a reference (mutable box) containing a `cell`. A `cell` is either `Nil` or `Cons`, and in the `Cons` case, the tail is another `llist`. This means that the overall `llist` list can be assigned to, as can the tail of any `Cons`.

For example:

```
val example1 : int llist = ref Nil
val example2 : int llist = ref (Cons (1, example1))
val [1] = tolist example2
val () = example1 := (Cons (2, ref Nil))
val [1,2] = tolist example2
```

`tolist` converts an `'a llist` to an `'a list`. In the fourth line, we update the list contained in the box `example1` to `Cons(2,refl Nil)`. *This changes `example2` as well, because `example2` is constructed using `example1`!*

Task 2.1 To make sure you understand how to read the contents of references, define the function

```
tolist : 'a llist -> 'a list
```

2.1 Map

Task 2.2 Write a function

```
val map : ('a -> 'a) -> 'a llist -> unit
```

such that `map f l` modifies `l`, so that each element `x` is replaced with `f x`. `map` should do an in-place update, using only the `ref`s already present in `l`; you will need new `Cons`'es.

Task 2.3 Can `map` be given the type `('a -> 'b) -> 'a llist -> unit`? Why or why not?

Have the TAs check your code before proceeding!

2.2 Filter

Task 2.4 Write a function

```
val filter : ('a -> bool) -> 'a llist -> unit
```

such that `filter p l` modifies `l`, so that only the elements satisfying `p` remain. `filter` should do an in-place update, using only the `ref`s and `cell`s already present in `l`.

Task 2.5 Explain why you cannot write filter with the following type:

```
val filter' : ('a -> bool) -> 'a cell -> unit
```

2.3 Append

Task 2.6 Define a function

```
append : 'a llist * 'a llist -> unit
```

`append(l1,l2)` should modify `l1`, replacing the end of it with `l2`, while keeping `l2` unchanged. `append` should do an in-place update, using only the `ref`s and `cell`s already present in `l1` and `l2`.

For example, the following tests that the *contents* of `l1` and `l2` are correct, using `tolist`:

```
val test1 = ref (Cons (1, ref (Cons (2, ref (Cons (3, ref Nil))))))
val test2end = ref Nil
val test2 = ref (Cons (4, ref (Cons (5, ref (Cons (6, test2end))))))
val () = append(test1,test2)

val [1,2,3,4,5,6] = tolist test1
val [4,5,6] = tolist test2
```

We also need to test that the *sharing* is correct: that `test1`'s tail is in fact `test2`, rather than a copy of it. The following example indicates that, because an update to `test2` changes `test1`:

```
val () = test2end := Cons (7, ref Nil)
val [4,5,6,7] = tolist test2
val [1,2,3,4,5,6,7] = tolist test1
```

Have the TAs check your code before proceeding!

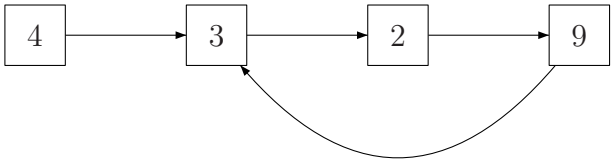

Figure 1: a cyclic linked list of integers

Figure 2: an acyclic linked list of integers

3 Cyclic Lists

We say a linked list is cyclic if there exists a path from any node in the list to itself. A linked list is said to be acyclic if there is no such path. For example, the list in Figure 1 is cyclic but the list in Figure 2 is acyclic.

There's nothing wrong with cyclic lists per se, but code that that assumes acyclic input will almost always fail on cyclic lists. For example, we can define the aforementioned cyclic list by

```
val testend = ref Nil
val testmid = ref (Cons (3, ref (Cons (2, ref (Cons (9, testend))))))
val () = testend := !testmid
val testcyclic = ref (Cons (4, testmid))
```

Task 3.1 What happens when you run `tolist` on `testcyclic`? `printelts` prints the elements of a list. What happens when you run it on `testcyclic`?

It is therefore helpful to have a fast way to test if a list is cyclic.

Task 3.2 [BONUS] Define a function

```
cyclic : 'a llist -> bool
```

The function `cyclic` evaluates to `false` if the argument list is acyclic and `true` if the argument list is cyclic.

Your implementation of `cyclic` must use only a constant amount of space and run in time at most quadratic in the number of unique cells in the argument list. You will need to follow the references in the linked list structure, but you may not destroy the linked list. Your implementation should be purely functional, so you may not have any ephemeral storage.