15-150 Summer 2016
Lab 10

13 April 2016

1 Introduction

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 Linked Lists

Today you're going to write an imperative version of lists commonly called *linked lists*. 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.

```
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 recursive; 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` (as well as the tail of any `Cons`) can be assigned to.

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,ref 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`.

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

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 Think about 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`. You may assume that `l1` is not `Nil`.

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
```

3 New Kids on the Block (or, uh, Reference Cell)

Here's a typeclass that we can only implement using side effects

```
signature NEW = sig
    type t
    val new : unit -> t
    val eq  : t * t -> bool
end
```

`NEW` is the typeclass representing types that have an equality operator, and where *new* elements of the type can be created dynamically, which will always be diffent from every other value of that type that has ever been created.

Fun Bonus Fact: This is a valid implementation of `NEW`

```
structure ExnNew = struct
type t = exn * (exn -> bool)
fun new () =
    let
      exception Eq
    in
      (Eq, fn Eq => true | _ => false)
    end

fun eq ((e1,f1),(e2,f2)) = f1 e2
end
```

Task 3.1 Implement `IntNew : NEW` where `IntNew.t` is an abstract type containing an int. Use a reference to create new values of the abstract type.

Task 3.2

Implement `RefNew : NEW` where `RefNew.t = unit ref`

4 Union-Find Grill

Side Note: Union-Find is a very popular algorithm. Feel free to use the internet to find explanations/demonstrations of and/or pseudocode for union-find (Since this is a lab. If this were a graded assignment, explanations/demonstrations are okay but you should stay from pseudocode).

Union-Find is a famous problem in computer science, which can be phrased as implementing the following signature:

```
signature UNION_FIND =
sig
    type table
    val init  : int -> table
    val unify : table -> int -> int -> unit
    val eq    : table -> int -> int -> bool
end
```

Union-find allows us to create a *table* containing n. Initially, all values are different, so `eq t i j` will return false for all pairs of indices into the table (i, j). The operation `unify t i j` Makes values i and j equal, so that in the future `eq t i j` will be true. What makes this interesting is that equality has to be reflexive, symmetric and transitive, and `unify` preserves these properties. For example, here are some tests for a `UF : UNION_FIND`

```
val uf = UF.init 10
val () = UF.unify uf 0 1
val () = UF.unify uf 2 3
val () = UF.unify uf 0 3
val true = UF.eq uf 0 1
val true = UF.eq uf 1 2
val true = UF.eq uf 2 3
val false = UF.eq uf 3 6
val true = UF.eq uf 6 6
```

Note that the `UNION_FIND` data struture can use any type that supports an equality operation in the implementation, so we can write it as a functor over the `NEW` typeclass.

In the functor `UnionFind`, write the following:

Representation

Our data structure represents the table as a sequence of references (which is intuitively an array). Each reference contains a cell, which is either `Var of N.t` or `Ptr of cell ref` (a.k.a. `Ptr of ptr`). The idea is that if i and j are the same, then when you follow all of their `Ptr`s, you will eventually get to the same `Var` (but you might take different `Ptr`'s to get there)

Assumptions

You can assume all code is run sequentially on this problem. It will be convenient to use some sequence functions in ways that would not be safe in parallel. You should think about why they would not be safe in parallel, but it's okay to write that code anyway.

Task 4.1 Implement `init`, which generates a sequence containing `n` elements, all of which are different.

Task 4.2 Implement a helper function `compress : ptr -> ptr` where `compress r` returns the last reference at the end of the chain of `Ptr`'s starting with `r`. Along the way, you should take all of the pointers on that chain and make them all point directly to `r` (this makes things faster).

Task 4.3 Implement `unify : table -> int -> int -> unit` which makes the elements of the table at position `i` and `j` the same. To do this, use `compress` to follow the `Ptr` chain for each element, then make the end of one pointer chain point to the end of the other pointer chain.

Task 4.4 Implement `eq : table -> int -> int -> bool` to test whether two indices are currently equal. You can do this by following the pointer chains with `compress` and checking whether the elements at the end are equal.

Task 4.5 Implement a test functor `TestIt` which takes in a `UNION_FIND`. Test all 3 implementations of `NEW`