15-150 Fall 2014
Lab 09

Thursday 30^{th} October, 2014

The goal for this lab is to make you even more familiar with continuations. Please take advantage of this opportunity to practice writing functions and proofs with the assistance of the TAs and your classmates. You are encouraged to collaborate with your classmates and to ask the TAs for help.

1 Introduction

1.1 Getting Labs

We will be distributing the text and any starter code for the labs using Autolab. Each week's lab will start being available at the beginning of class.

On the Autolab page for the course, the current lab is the last entry under `Lab`. Say it is called "`labnn`". Click on this link. There, two links matter

- View writeup: this is the text of the lab in PDF format.
- Download handout: this is the starter code, if any, for the lab. It will always be distributed as a compressed archive (in `.tgz` format). Uncompressing it will create the following directories:

`lab`*nn*`/`	Directory for lab *nn*
`code/`	Code directory for lab *nn*
`*.sml`	Starter files for lab *nn*
`handout.pdf`	Copy of writeup for lab *nn*

1.2 Code Structure

On this and future assignments, we will be grading your programs on more than just their input-output behavior. It's not enough to have programs that happen to work: they need to clearly state what they do, have some empirical evidence that they work as advertised, and be easy for other people to read and reason about.

You must use the following five step methodology for writing functions, for *every* function you write in this assignment:

1. In the first line of comments, write a call template of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns).
4. Implement the function (include type annotations for the arguments and result of the function)
5. Provide test cases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function presented in lecture:

```
(*  factorial (n) ==> res
 *  REQUIRES:  n >= 0
 *  ENSURES: res is  n!
 *)
fun factorial (0: int): int = 1
  | factorial n = n * factorial (n-1)

(* Tests: *)

val 1 = factorial 0
val 720 = factorial 6
```

2 Tagged Trees

In this lab we will be solving the problem of searching on tagged trees. A *tagged tree* is a tree in which every node stores an entry consisting of both a piece of data and an identifying *tag*. The tags are not necessarily unique, so there may be a lot of entries with the same tag. Entries are not sorted by tags either. From this arise two obvious search problems: First, return some piece of data associated with a given tag, and second return the whole list of data associated with a tag.

We define a tagged tree using the following SML types:

```
type tag = int
type 'a entry = tag * 'a
datatype 'a tree = Empty
                 | Node of 'a tree * 'a entry * 'a tree
```

An example of a `string tree` is the following:

```
val x: string tree = Node(Node(Empty,
                               (1,"bar"),
                                Empty),
                          (2,"foo"),
                          Empty)
```

Tagged data and functions that retrieve information on the basis of tags are key aspects of social networking applications such as Facebook, Twitter and many others. They are also closely related to databases and how some type of information in them is accessed.

Now, you will implement four different functions to search tagged trees.

2.1 Finding tagged data

Task 2.1 Implement the following function in ttrees.sml

```
search1: tag * 'a tree -> 'a option
```

using a regular recursive search. The call `search1 (tag, t)` return `NONE` if no entry in the tree has the given tag, or `SOME d` if `d` is any piece of data with tag `tag` in `t`.

2.2 Finding tagged data with continuations

Next, we will implement the same task using continuations. Recall that a continuation is an additional functional argument that stores the remaining work to be done, effectively postponing it till later. In the case where we do not find the tag we were looking for, we call the continuation to explore the parts of the tree we had postponed.

This kind of continuations, where we call the continuation when we don't find what we were looking for, are called *failure continuations* — we call the continuation when our search fails. Note that we do not have any interesting data to pass to a failure continuation — we just need to trigger it. One good way to do so is to call it with `()`, the one and only value of type `unit`.

Task 2.2 To do so, we will begin by implementing the following helper function:

```
search2': tag * 'a tree -> (unit -> 'a option) -> 'a option
```

where `search2' (tag, t) k` returns `SOME d` if

- either `t` contains an entry with tag `tag` and `d` is any associated data,
- or there is no entry with tag `tag` in `t`, but `k ()` returns `SOME d`.

`NONE` is returned in all other cases.

Task 2.3 Using `search2'`, define the function

```
search2: tag * 'a tree -> 'a option
```

such that `search2 (tag,t)` returns `SOME d` iff `t` contains the entry `(tag,d)` for any `d`.

2.3 Finding all tagged data

Up until this point, we only needed a single solution. However, tags are not unique, so next we will modify our search to find the list of all the data associated with a tag, rather than just one. Now, the continuation will be invoked also when we find what we were looking for. Moreover, the continuation will need the partial solution found by our current exploration and stitch it together with its own findings. This time, we do have something interesting to call it with — the partial solution — and therefore it will take an `'a list` as input instead of `unit`.

Continuations that are invoked even when the search succeeds are called *success continuations*. Their input type is typically not `unit`. All the continuations we have seen in class were success continuations.

Task 2.4 Implement the following helper function:

```
search3': tag * 'a tree -> ('a list -> 'a list) -> 'a list
```

where `search3' (tag,t) k` returns `l` such that, if `l'` is the list of data with tag `tag` in `t`, then `k l'` returns `l`.

Task 2.5 Using `search3'`, define the function

```
search3: tag * 'a tree -> 'a list
```

such that `search3 (tag,t)` returns the list of all the data with tag `tag` in `t`.

2.4 Finding tagged data, one at a time

Finally, we explore one more way to solve the search problem on tagged trees: what if now we need one solution, and maybe later we'll need more ... or maybe not. We could do this with `search3` — pick the head of the returned list now and maybe later look at the rest. This is inefficient, though, if all we will be using is the first solution: the rest of the list was computed for nothing.

What we want back is the first result (if there is one) together with a function that, when called, will return the second result and a function that, when called, will return the third result, and so on. To do this, we define a new type:

```
datatype 'a result = NoMore
                   | Next of 'a * (unit -> 'a result)
```

where `NoMore` signals that there are no more results (in a way, it does the job that `NONE` did in `search1`).

`Next(d,f)` returns one piece of data found during the search (that's `d`), and a function `f` to be called to find the next solution (or `NoMore` if there isn't any).

Task 2.6 Write the following helper function

```
search4': tag * 'a tree -> (unit -> 'a result) -> 'a result
```

such that `search4' (tag,t) k` returns `NoMore` if there are no entries with tag `tag` in `t` and `k ()` returns `NoMore`. Moreover, it returns `Next(d,f)` if

- either `t` contains an entry with tag `tag` and `d` is the associated data, and `f` is a function that returns the next result,
- or `t` contains no such entry, but `k ()` evaluates to `Next(d,f)`.

The important part to note here is that we are taking the encapsulation provided by continuations and using them to stop doing work altogether until we next need another search result. Your code should not use lists.

Task 2.7 Using `search4'`, write the function

```
search4: tag * 'a tree -> result
```

such that `search4(tag,t)` returns either `NoMore` if `t` does not contain entries with tag `tag`, or `Next(d,f)` where `d` is one such piece of data and `f` is a function that returns the next result.

Checkout point!

Completing everything up to here in the lab assignment will guarantee credit for this lab.

Click here or go to the class schedule and click on a `Check me in` button.