15-150 Fall 2014
Lab 04

Thursday 18$^{\text{th}}$ September, 2014

The goal for this lab is to give you more practice with analyzing the work and span of functions. Here, you will analyze functions that call other functions and see how that affects the work.

Please take advantage of this opportunity to practice writing code and proofs with the assistance of the TAs and your classmates. You are, as always, 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 Proof Structure

Remember that every proof by induction is structured as follows:

1. The specific *technique* being employed and on what.
2. The *structure* of the proof (number of cases and what they are).
3. For each base case:
   - The statement specialized to this case ("*To show*").
   - The proof of this case.
4. For each inductive case:
   - The statement specialized to this case ("*To show*").
   - The induction hypothesis or hypotheses (*IH*).
   - The proof of this case.

Following this methodology, students have historically submitted proofs that contained fewer errors and were more likely to be correct than otherwise.

1.3 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 Analysis: Traversing trees

For this section, we will be analyzing functions on trees. Here, a tree is defined by the following datatype:

```
datatype tree = Empty
              | Node of tree * int * tree
```

In this section, we will compare the work of two different implementations of

```
inorder: tree -> int list
```

which performs an inorder traversal of the input tree and returns the corresponding list of integers.

2.1 Straight-recursive Traversal

The definition of the `inorder` function that we have used so far is:

```
fun inorder (Empty: tree): int list = []
  | inorder (Node(l,x,r)) = inorder(l) @ (x :: inorder(r))
```

While analyzing this function, you may assume that the tree is balanced. That is, given `Node(l,x,r)`, we know `length(inorder(l))` will be approximately equal to `length(inorder(r))`.

Task 2.1 Find a recurrence to represent the work of `inorder` using the work of `@`.

Task 2.2 We know that the work and span of `@` is linear in the size of `l1`. Find a closed form for the work of `inorder`.

Task 2.3 Find a recurrence and a closed form for the span of `inorder`.

Task 2.4 Suppose we did not assume the tree is balanced. Explain what the worst case input tree would look like.

Task 2.5 Find a recurrence and solve for the work of the worst case input to `inorder`.

Task 2.6 Explain what the input tree would look like in the best case.

Task 2.7 Find a recurrence and solve for the work of the best case input to `inorder`.

2.2 Tail-recursive Traversal

Now we introduce an alternate function for traversing a tree using partial tail recursion.

```
(* inorder2 (T, L) ==> L'
   ENSURES: inorder2(T,L) == inorder(T) @ L
 *)
fun inorder2(Empty: tree, L: int list): int list = L
  | inorder2(Node(l,x,r),L) =
     let
       val L' = x::inorder2(r, L)
     in
       inorder2(l, L')
     end
```

Task 2.8 Before analyzing `inorder2`, do you expect it to perform better, worse, or the same as `inorder`? Why?

Task 2.9 Find and solve a recurrence representing the work of `inorder2`

Task 2.10 We did not assume that the input to `inorder2` is a balanced tree. Give an intuitive reason why the work is the same for inputs to `inorder2`, whereas this was not the case for `inorder`.

3 List permutations

One useful notion when you have two lists is comparing if one list is a *permutation* of the other list. That is, if the two lists both contain the same elements, possibly in different order. In this section, we will present an algorithm for checking if this property holds, and we will implement it in two different ways.

3.1 Using Destructors

To check if some list l_1 is a permutation of l_2, it suffices to show that each element of l_1 corresponds to an element of l_2 without any leftovers. This means that we need to have a function that checks that an element is in a list, a second function that removes it from that list, and a third function that combines them to determine if a list is a permutation of another.

Let's do it!

Task 3.1 Implement the SML function

```
member: int * int list -> bool
```

such that `member (x, l)` returns `true` if the number `x` occurs in list `l`, and `false` otherwise.

Task 3.2 Implement the SML function

```
remove: int * int list -> int list
```

such that, given a list `l` and an integer `x` that occurs in `l`, the call `remove (x, l)` returns the list `l'` obtained by removing the first occurrence of `x` from `l`.

Task 3.3 Implement the SML function

```
isPermutation: int list * int list -> bool
```

such that `isPermutation (l1, l2)` returns `true` if `l1` is a permutation of `l2`, and `false` otherwise.

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.

3.2 Enter `option`'s

Next, we will be exploring the use of a type you have not seen so far, the `option` type.

The `option` type is used in SML when a function does not always have a meaningful value to return. For example, consider the following spec about the function `indexAt`, of type `int * int list -> int`:

> *`indexAt(x,L)` returns the index of the first occurrence of element `x` in list `L`*

The problem with this function is that it does not have a value to return if `x` is not in `L` at all. We could raise an exception, but we might not want to crash our program for some everyday input. Logically, the function should return a statement saying that `x` is not in `L`. This is what the `option` datatype allows us to do.

An option can be one of two things:

- `NONE`
- `SOME(x)`, for some value `x`

The datatype `option` is completely specified by these two values. So to return to the example, if `x` is not in `L`, we can use `NONE` as a return value for `indexAt(x,L)` to indicate that there is no meaningful index to return. This solves the issue we were experiencing. If you want to look at the code for `indexAt` for reference, it is included in the starter code. Observe how it uses a `case` expression to discriminate on partial results.

The type of `indexAt` is updated to `int * int list -> int option`. In fact, options work just like list: `int option` describes that whenever the value `SOME(x)` is returned, then `x` has type `int`. We can have an `int option`, `string option`, etc, just like how we can have an `int list`, `string list`, etc. This isn't something you need to worry about right now — we will look into that in a forthcoming lecture.

3.3 Checking for Permutations using Options

Now that we know about the `option` type, we can write the function that checks if a list is a permutation of another one more directly. We will have just two functions.

Task 3.4 Write the function

```
checkAndGet: int * int list -> int list option
```

such that the following formal specs are satisfied:

- `checkAndGet(x,l)` ≅ `SOME (l1 @ l2)` if `l` ≅ `l1 @ [x] @ l2`;
- `checkAndGet(x,l)` ≅ `NONE` otherwise.

In plain English, `checkAndGet(x,l)` finds `x` in `l`, removes it, then returns `SOME l'` where `l'` has `x` removed; or it returns `NONE` if `x` does not occur in `l`.

Task 3.5 Using `checkAndGet` and nothing else, write the SML function

```
isPermutation2: int list * int list -> bool
```

such that `isPermutation2 (l1, l2)` returns `true` if `l1` is a permutation of `l2`, and `false` otherwise.