15-150 Fall 2014
Lab 05

Thursday 25th September, 2014

The goal for this lab is to give you familiarity with type inference, scope, polymorphism, and proofs of polymorphic statements.

Please take advantage of this opportunity to practice writing 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 Warmup: Types in SML

Task 2.1 For each of the following expressions, determine its type, if it has one. If the expression does not typecheck, answer “DNT”. Do this exercise without the help of the SML interpreter.

1. `[]`
2. `[[]]`
3. `NONE`
4. `if false then 1.0 else 1`
5. Assume you have determined that the functions `f`, `g`, `h` have types `'a -> 'b`. What is the type of
   - `fun w x = if h x then f x else g (f x)`
   - `fun f (x,y) = if x=1 then y else "asdf"`

Task 2.2 Determine the type of each of the following expressions. Also, determine the values of the variables at each line, and the value of all the *let*-expression.

-
  ```
  1.  let
  2.    val x = 4
  3.  in
  4.    (let
  5.       val x = 5
  6.     in
  7.       x*20
  8.     end) div x
  9.  end
  ```

-
  ```
  1.  let
  2.    val x = 0
  3.    val y = 7
  4.  in
  5.    (let
  6.       val x = 5
  7.       val y = nil
  8.     in
  9.       x::y
  10.    end) @ [x - y]
  11. end
  ```

3 Polymorphic Trees

For these tasks, we will use the following datatype:

```
datatype 'a tree = Empty
                 | Node of 'a tree * 'a * 'a tree
```

You can find the code for the function `inorder: 'a tree -> 'a list` carries out an in-order traversal of its argument.

Task 3.1 Write a function

```
merge: 'a tree * 'a tree -> 'a tree
```

such that the call `merge(t1,t2)` returns a tree `t` so that `(inorder t1) @ (inorder t2)` is equivalent to `(inorder t)`.

Task 3.2 Prove that

For all `t,t':'a tree`*,*

$$\texttt{inorder(merge(t,t'))} \cong \texttt{(inorder t) @ (inorder t')}$$

The polymorphic type variable `'a` can be instantiated to any type. Equality, however, is not meaningful on all types (you can't use `=` on expressions of type `real` for example — we will soon encounter other types that do not admit equality). However, in SML we can write

```
''a
```

for a type variable that can only be instantiated to types that admit equality (so that replacing it with `real` is disallowed). While it slightly restricts polymorphism, this gives us a way to write a lot of useful functions that, directly or indirectly, rely on equality. Here are some.

Task 3.3 Write the function

```
member: ''a * ''a list -> bool
```

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

Task 3.4 Write the function

```
indexOf: ''a * ''a list -> int option
```

defined so that `indexOf(x,l)` returns `NONE` if `x` is not in `l`, and otherwise returns `SOME n` where `n` is the index of the first occurrence of `x` in `l`.

Task 3.5 Write a function

```
    remove: ''a * ''a list -> ''a list
```

such that `remove(x,l)` returns the list obtained by removing all occurrences of `x` from `l`.

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.

4 More Polymorphic Trees

In this exercise, we will consider the same notion of tree as in the previous section.

Task 4.1 Write a function

```
removeT: ''a * ''a tree -> ''a tree
```

so that `removeT(x,t)` returns the tree obtained by removing all occurrences of `x` from `t`. For this task, you should use `merge` from the previous section.

Task 4.2 Prove that

For all `x:''a` and `t:''a tree`,

$$\texttt{inorder(removeT(x,t))} \cong \texttt{remove(x, inorder t)}$$

You may use the property you proved earlier in this proof as well as the following lemma:

For all `x:''a` and `l1,l2:''a tree`,

$$\texttt{remove(x, l1 @ l2)} \cong \texttt{remove(x, l1) @ remove(x, l2)}$$

For the next task, we use the following datatype:

```
datatype direction = L | R
```

Task 4.3 Write a function

```
path: ''a * ''a tree -> direction list option
```

such that `path(x,t)` returns `NONE` if `x` is not in `t`, and otherwise returns `SOME l` where `l` is the list representing the path from the root of `t` to `x`. This list is composed of `L`s and `R`s, representing moving left and right on the branches of `t`