15-150 Fall 2014
Lab 02

Thursday 4$^{\text{th}}$ September, 2014

The goal for this lab is to make you more comfortable writing induction proofs on inductively defined objects such as lists and trees. You will also have the opportunity to practice your SML skills by writing functions over inductively defined types.

Take advantage of this opportunity to practice writing 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 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 Inductive Lists

Here is the inductive definition of the set of lists $\mathbb{L}_\mathbb{Z}$ over the domain of the integers $\mathbb{Z}$:

$$\begin{cases} nil \in \mathbb{L}_\mathbb{Z} \\ :: \ : \mathbb{Z} \times \mathbb{L}_\mathbb{Z} \to \mathbb{L}_\mathbb{Z} \end{cases}$$

For convenience, we are using SML's notation for the list constructors rather than *Cons* and *Nil* seen in class. This definition says that every list is either *nil* or $x :: L$ where x is an integer and L is some valid list.

From this definition, we cannot use $\_ :: \_$ to connect two lists, so in lecture we introduced a function $append : \mathbb{L}_\mathbb{Z} \times \mathbb{L}_\mathbb{Z} \to \mathbb{L}_\mathbb{Z}$, where $append(L_1, L_2)$ returns a list obtained by concatenating the elements of L_1 and the elements of L_2:

$$\begin{cases} append(nil, L_2) & = L_2 \\ append(x :: L_1', L_2) & = x :: append(L_1', L_2) \end{cases}$$

With the help of this function, we also defined a function to reverse a list:

$$\begin{cases} rev(nil) & = nil \\ rev(x :: L_1') & = append(rev(L_1'), x :: nil) \end{cases}$$

Finally, we defined the function $length : \mathbb{L}_\mathbb{Z} \to \mathbb{N}$ as

$$\begin{cases} length(nil) & = 0 \\ length(x :: L_1') & = 1 + length(L_1') \end{cases}$$

Task 2.1 Prove the following theorem using structural induction on lists:

$$\text{For all } L \in \mathbb{L}_\mathbb{Z}, \quad length(rev(L)) = length(L)$$

You may freely assume the following property:

Lemma 1

For all $L_1, L_2 \in \mathbb{L}_\mathbb{Z}$, $length(append(L_1, L_2)) = length(L_1) + length(L_2)$.

Task 2.2 Prove the following theorem using structural induction on lists:

$$\text{For all } L \in \mathbb{L}_\mathbb{Z}, \quad rev(rev(L)) = L$$

You may freely assume the following property:

Lemma 2

For all $L_1, L_2 \in \mathbb{L}_\mathbb{Z}$, $rev(append(L_1, L_2)) = append(rev(L_2), rev(L_1))$.

3 Inductive Trees

The set of trees $\mathbb{T}_{\mathbb{Z}}$ over the domain of the integers $\mathbb{Z}$ is inductively defined as:

$$\begin{cases} \mathit{empty} \in \mathbb{T}_{\mathbb{Z}} \\ \mathit{node} : \mathbb{T}_{\mathbb{Z}} \times \mathbb{Z} \times \mathbb{T}_{\mathbb{Z}} \to \mathbb{T}_{\mathbb{Z}} \end{cases}$$

In other words, a tree is either *empty* or $\mathit{node}(T_1, x, T_2)$, where T_1 and T_2 are trees and $x \in \mathbb{Z}$.

From this definition, we consider two functions on trees. First, the function $\mathit{inorder} : \mathbb{T}_{\mathbb{Z}} \to \mathbb{L}_{\mathbb{Z}}$ traverses its argument and collects the data in its node into a list, that is returned. It is defined as follows:

$$\begin{cases} \mathit{inorder}(\mathit{empty}) & = \mathit{nil} \\ \mathit{inorder}(\mathit{node}(T_L, x, T_R)) & = \mathit{append}(\mathit{inorder}(T_L), x :: \mathit{inorder}(T_R)) \end{cases}$$

inorder provides us with the means to reason about the elements of a tree using the structure of a list. The second function we will be using is the familiar $\mathit{size} : \mathbb{T}_{\mathbb{Z}} \to \mathbb{N}$ that determines the number of *node*s in a tree:

$$\begin{cases} \mathit{size}(\mathit{empty}) & = 0 \\ \mathit{size}(\mathit{node}(T_L, x, T_R)) & = 1 + \mathit{size}(T_L) + \mathit{size}(T_R) \end{cases}$$

Task 3.1 The type `tree` is defined for you in the code file for this lab. In this file, complete the code for the functions `inorder` and `size`. Don't forget the specs and the test cases.

Task 3.2 By now, we have defined functions on both lists and trees that determine the size of the structure: that's *length* on lists and *size* on trees. This allows us to show that *inorder* preserves the size of its input.

Prove the following property by structural induction:

$$\text{For all } T \in \mathbb{T}_{\mathbb{Z}}, \qquad \mathit{size}(T) = \mathit{length}(\mathit{inorder}(T))$$

You may use the lemmas given on lists in the previous section.

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 SAT Trees

As seen in Lab 01, given an expression we can draw a computation tree to determine its work and span. For example, in lab 01 we found that the following tree has work 4 and span 3.

```
         *
       /   \
     +       +
    / \     / \
   1   2   3   +
              /  \
             4    5
```

Now that we have the tools to work on inductively defined entities, we will define a specific application of binary trees. Specifically, we will provide a basic structure with which we can evaluate a *Boolean* expression.

The two base values of type `bool` are `true` and `false`. As we saw, Boolean expressions are constructed by means of the logical operations `not`, `andalso`, `orelse`. `not` maps a Boolean expression to its logical negation, while the other two operations map two Boolean expressions into one.

Thus, we can define the set $\mathbb{B}$ of the Boolean expression as an inductive domain, similarly to our definition of binary trees (for brevity, we write *and* and *or* for conjunction and disjunction):

$$\mathbb{B} := \begin{cases} \mathit{true} \in \mathbb{B} \\ \mathit{false} \in \mathbb{B} \\ \mathit{not} : \mathbb{B} \to \mathbb{B} \\ \mathit{and} : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ \mathit{or} : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \end{cases}$$

Using this specific definition for Boolean expressions, assuming that the work of any operator is 1, we can give precise mathematical definitions for the *work* and *span* of evaluating any Boolean expression.

Task 4.1 Using the inductive definition of $\mathbb{B}$, give an inductive definition of $\mathit{work} : \mathbb{B} \to \mathbb{N}$.

Task 4.2 Using the inductive definition of $\mathbb{B}$, give an inductive definition of $\mathit{span} : \mathbb{B} \to \mathbb{N}$.

Task 4.3 Using the above, prove the following theorem:

$$\text{For all } T \in \mathbb{B}, \quad \mathit{work}(T) \geq \mathit{span}(T)$$

Now we will introduce the notion of the *dual* of a Boolean expression. We create the dual of a Boolean expression by swapping every *true* with *false*, and every *false* with *true*. Furthermore, we swap every *and* with *or* as well as every *or* with *and*.

Task 4.4 Provide the formal inductive definition for $dual : \mathbb{B} \to \mathbb{B}$.

Task 4.5 Prove the following theorem:

$$\text{For all } T \in \mathbb{B}, \quad dual(dual(T)) = T$$

Having established some results about Boolean trees, you will find in the code file for this lab a datatype named `btree` that implements the set $\mathbb{B}$ of the Boolean trees.

To avoid naming conflicts with built-in SML types, `btree`s are represented by symbols commonly used in expressing Boolean expressions: $T, F, \wedge, \vee, !$, where $T \cong$ *true*, $F \cong$ *false*, $\wedge \cong$ *and*, $\vee \cong$ *or*, $! \cong$ *not*.

The operators $\wedge$ and $\vee$ are *infix* constructors, meaning they should be placed between their arguments. Three examples have been provided representing a short, medium, and long expression in the code.

Task 4.6 Fill in the partially-written function `dual: btree -> btree` inside the code file for this lab file. Remember to write specs and test cases.

Task 4.7 Write a function `eval: btree -> bool`, which evaluates a `btree` and returns the bool value that the tree evaluates to when following standard rules of Boolean logic.