15-150 Fall 2014
Homework 02

Out: Tuesday 2nd September, 2014
Due: Tuesday 9th September, 2014

1 Introduction

This assignment focuses on inductive domains other than the natural numbers. You will be defining inductive function definitions on such domains, express them in Standard ML, and prove properties using structural induction. Examples of such inductive domains are trees and lists.

1.1 Getting the Homework Assignment

The starter files for the homework assignment have been distributed through Autolab at

`https://autolab.cs.cmu.edu`

Select the page for the course, click on "hw02", and then "Download handout". Uncompressing the downloaded file will create the directory `hw02-handout` containing the starter files for the assignment. The directory where you will be doing your work is called `hw/02` (see below): copy the starter files there.[1]

1.2 Submitting the Homework Assignment

Submissions will be handled through Autolab, at

`https://autolab.cs.cmu.edu`

In preparation for submission, your `hw/02` directory will need to contain the file `hw02.pdf` with your written solutions and the files `itree.sml` and `robot.sml` with your code. The starter code for this assignment contains incomplete versions of these files. You will need to complete them with the solution to the various programming tasks in the homework.

To submit your solutions, run

[1] The download and working directory have different names so that you don't accidentally overwrite your work were the starter files to be updated. **Do not do your work in the download directory!**

```
make
```

from the `hw/02` directory on any Unix machine (this include Linux and Mac). This will produce a file `hw02-handin.tgz`, containing the files to be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw02-handin.tgz` file via the "Handin your work" link. If you are working on AFS, you can alternatively run

```
make submit
```

from the `hw/02` directory. That will create `hw02-handin.tgz`, containing the files to be handed in for this homework assignment and directly submit this to Autolab.

All submission will go through Autolab's "autograder". The autograder simply runs a series of tests against the reference solution. Each module has an associated number of points equal to the number of functions you need to complete. For each such function, you get 1.0 points if your code passes all tests, and 0.0 if it fails at least one test. Click on the cumulative number for a module for details. Obtaining the maximum for a module does not guarantee full credit in a task, and neither does a 0.0 translate into no points for it. In fact, the course staff will be running additional tests, reading all code, and taking into account other aspects of the submitted code such as structured comments and tests (see below), style and elegance.

To promote good programming habits, your are limited to a maximum of 5 submissions for this homework. Use them judiciously! In particular, make sure your code compiles cleanly before submitting it. Also, make sure your own test suite is sufficiently broad.

Remember that your written solutions must be submitted in PDF format—we do not accept MS Word files or other formats.

The SML files `itree.sml` and `robot.sml` must contain all the code that you want to have graded for this assignment, and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Due Date

This assignment is due on Tuesday 9$^{\text{th}}$ September, 2014. Remember that there are no late days for this course and you will get 2 bonus points for every 12 hours that you submit early.

1.4 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.

1.5 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
```

1.6 Style

Programs are written for people to read — it's convenient that they can be executed by machines, but their high level text is a way for one person to explain an idea to another person. Your code should reflect this, and we will grade your code on how easy it is to understand.

The published style guide is your primary resource here. Strive to write clear, concise, and elegant code. If you have any questions about style, or just have a feeling that a piece of code could be written more simply, don't hesitate to ask!

2 Instrumented Trees

In this exercise, we will consider binary trees that can be empty, or a leaf carrying a string, or an inner node with two subtrees. Such trees constitute the inductive domain $\mathbb{T}$ defined as follows:

$$\begin{cases} \mathit{Empty} \in \mathbb{T} \\ \mathit{Leaf} : \mathbb{S} \to \mathbb{T} \\ \mathit{Node} : \mathbb{T} \times \mathbb{T} \to \mathbb{T} \end{cases}$$

Therefore, the empty tree is represented as *Empty*, the leaf annotated with string $s \in \mathbb{S}$ is written $\mathit{Leaf}(s)$, and the tree with left and right subtrees t_L and t_R respectively takes the form $\mathit{Node}(t_L, t_R)$.

The *size* of a tree is the number of strings it contains. Specifically, the size of the empty tree is 0, the size of a leaf is 1, and the size of a tree starting with an inner node is the sum of sizes of its subtrees.

2.1 Instrumented Trees

Given $t \in \mathbb{T}$, the *degree of imbalance* (or more briefly just *imbalance*) of t is 0 if t is empty or a leaf, and is the difference between the size of its left and right subtrees otherwise. In this last case, t's imbalance will be negative if its left subtree has more leaves than its right subtree, positive if its right subtree is has more leaves, and 0 if they have the same number of leaves.

An *instrumented tree* is a tree whose inner nodes record the imbalance of the subtree starting with them, recursively. (There is no need to record the imbalance of empty nodes and leafs.) This gives rise to the inductive domain $\mathbb{TT}$, defined as follows:

$$\begin{cases} \mathit{iEmpty} \in \mathbb{TT} \\ \mathit{iLeaf} : \mathbb{S} \to \mathbb{TT} \\ \mathit{iNode} : \mathbb{TT} \times \mathbb{Z} \times \mathbb{TT} \to \mathbb{TT} \end{cases}$$

In particular, in the instrumented tree $t = \mathit{iNode}(t_L, i, t_R)$, the number i is the imbalance of t, i.e., the different between the size of t_R and the size of t_L.

Task 2.1 (2 points) Give an inductive definition for the function $\mathit{iSize} : \mathbb{TT} \to \mathbb{N}$ that computes the size of its argument. The inductive clause should make two recursive calls.

Task 2.2 (2 points) Give an inductive definition of the function $\mathit{validate} : \mathbb{TT} \to \mathbb{B}$ such that $\mathit{validate}(t)$ returns *true* if the imbalance at every inner node of t is correct, and *false* otherwise.

Task 2.3 (2 points) Give an inductive definition to the function $\mathit{iSize}' : \mathbb{TT} \to \mathbb{N}$ such that $\mathit{iSize}'(t) = \mathit{iSize}(t)$ for a valid instrumented tree t. Your definition of iSize' must be such that the evaluation of $\mathit{iSize}'(t)$ makes *at most one* recursive call at each nodes it visits.

Task 2.4 (2 points) Give an inductive definition of the function $tiltLeft : \mathbb{TT} \to \mathbb{TT}$ such that $tiltLeft(t)$ is the tree obtained from t by selectively swapping the left and right subtrees so that each inner node has a non-positive imbalance.

2.2 Properties

If your definition of $iSize'$ is correct, for valid instrumented trees, it should compute the same value as $iSize$ on valid instrumented trees. Mathematically,

Property 1. *For all* $t \in \mathbb{TT}$, *if* $validate(t) = true$, *then* $iSize'(t) = iSize(t)$.

Task 2.5 (8 points) Prove this property using an appropriate form of induction.

Another property of the above definitions is that, if your definition of *tiltLeft* is correct, it should transform valid instrumented trees into valid instrumented trees. That is, the following property should hold.

Property 2. *For all* $t \in \mathbb{TT}$, *if* $validate(t) = true$, *then* $validate(tiltLeft(t)) = true$.

Task 2.6 (10 points) Prove this property using an appropriate form of induction. You may assume the following lemma without proof (but you need to cite its use):

Lemma: For all $t \in \mathbb{TT}$, $iSize(tiltLeft(t)) = iSize(t)$.

2.3 Implementation

The inductive definitions of trees and instrumented trees translate into the following `datatype` declarations:

```
datatype tree = Empty
              | Leaf of string
              | Node of tree * tree

datatype itree = iEmpty
               | iLeaf of string
               | iNode of itree * int * itree
```

These declarations are given to you in the starter file `itree.sml` of this homework.

Task 2.7 (5 points) Implement the SML function `instrument: tree -> itree` such that `instrument(t)` is the instrumented tree corresponding to `t`.

Task 2.8 (2 points) Implement the SML function `iSize: itree -> int` that realizes the mathematical function *iSize*.

Task 2.9 (2 points) Implement the SML function `validate: itree -> bool` that realizes the mathematical function *validate*.

Task 2.10 (2 points) Implement the SML function `iSize': itree -> int` that realizes the mathematical function *iSize′*.

Task 2.11 (2 points) Implement the SML function `tiltLeft: itree -> itree` that realizes the mathematical function *tiltLeft*.

3 The SML-Robot

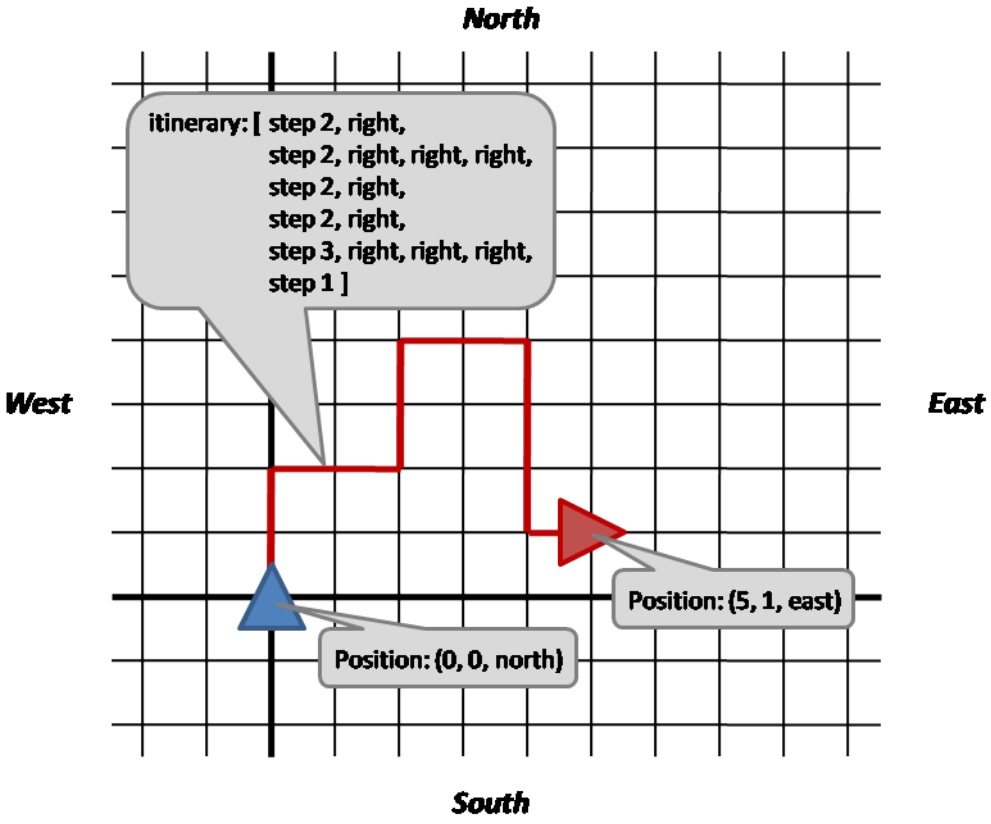

Figure 1: The SML Robot

The SML-Robot is a very simple robot that moves around on a flat surface. It is controlled with two *instructions*: *right* tells the robot to turn right and *step n* instructs the robot to move n units. If $n > 0$ then the robot will move forward. If $n < 0$ the robot will move backwards. An *itinerary* is a list of instructions that will make the robot move around. When the robot is moving, it is important to know where it is heading. The *orientation* of the robot is either *north*, *east*, *south* or *west*. The *position* of the robot in space is then a pair of coordinates (x, y) on an infinite bi-dimensional grid (the Cartesian plane), combined with its orientation.

We can give a precise mathematical definition of these notions as the following sets:

Instructions: $\mathbb{I} = \begin{cases} right \\ step\ n \quad \text{where } n \in \mathbb{Z} \end{cases}$

Itineraries: An itinerary is just a list of instructions. We will denote it as $\mathbb{L}_{\mathbb{I}}$.

Orientations: $\mathbb{O} \;=\; \{north, east, south, west\}$

Positions: $\mathbb{P} \;=\; \mathbb{Z} \times \mathbb{Z} \times \mathbb{O}$

Task 3.1 (2 points) Are $\mathbb{I}$, $\mathbb{L}_{\mathbb{I}}$, $\mathbb{O}$ and $\mathbb{P}$ inductive domains? Justify your answer. You may assume that $\mathbb{Z}$ is an inductive domain (there is indeed a way to define it on the basis of elementary ingredients and simple rules).

3.1 Operating the Robot

Task 3.2 (2 points) Define the function $turnRight : \mathbb{P} \to \mathbb{P}$ such that $turnRight(p)$ returns the new position of the robot at position p when receiving the instruction *right*.

Task 3.3 (2 points) Define the function $move : \mathbb{Z} \times \mathbb{P} \to \mathbb{P}$ such that $move(n, p)$ returns the new position of the robot at position p when receiving the instruction *step* n.

Task 3.4 (3 points) Define the function $getPosition : \mathbb{L}_{\mathbb{I}} \times \mathbb{P} \to \mathbb{P}$ such that, given itinerary l and initial robot position p, the function invocation $getPosition(l, p)$ returns its final position. For example (abbreviating *right* as "R" and *step* n as "$s\ n$" for conciseness),

$$getPosition([s\ 2, R, s\ 2, R, R, R, s\ 2, R, s\ 2, R, s\ 3, R, R, R, s\ 1], (0, 0, north))$$

has value $(5, 1, east)$. This is visualized in Figure 1.

3.2 Properties

If the function *getPosition* is defined correctly, we should be able to prove that moving the robot from position p to position p' along an itinerary l, and then moving it from p' to p'' with itinerary l' is equivalent to moving from p to p'' with the combined itinerary $l@l'$. This is expressed mathematically by the following property.

Property 3 (Appended Itineraries)**.** *For all* $l_1, l_2 \in \mathbb{L}_{\mathbb{I}}$ *and for all* $p \in \mathbb{P}$*,*

$$getPosition(l_1@l_2, p) \;=\; getPosition(l_2, getPosition(l_1, p))$$

Task 3.5 (6 points) Prove this property using an appropriate form of induction. In your proof, you may (of course) rely on the definition of *append* (written @).

Task 3.6 (2 points) After moving by means of *getPosition*, the robot can go back to its original position by following the exact same itinerary but backward. To do that, we want to write a function that calculates the itinerary for the robot to go back to its initial position

based on the original itinerary. Define the function $goBack : \mathbb{L}_{\mathbb{I}} \to \mathbb{L}_{\mathbb{I}}$ so that $goBack(l)$ returns the itinerary that allows the robot to go back to the position it was at before executing l. For example,

$$goBack([step\ 2, right, step\ 1])$$

returns the itinerary

$$[step\ (-1), right, right, right, step\ (-2)]$$

(Convince yourself that this is indeed correct!)

If you defined the function *goBack* correctly, then you should be able to prove that if a robot moves from p to p' with l and then moves from p' with $goBack(l)$ then it gets back to p. This is expressed by the following mathematical property.

Property 4 (Go Back)**.** *For all* $l \in \mathbb{L}_{\mathbb{I}}$ *and* $p \in \mathbb{P}$*,*

$$getPosition(l@(goBack(l)), p) \quad = \quad p$$

Task 3.7 (10 points) Prove this property. In you proof, you may utilize of any of the following properties without proving them (as long as you cite their use):

- [Associativity of @] *For all* $l_1, l_2, l_3 \in \mathbb{L}_{\mathbb{I}}$*,* $l_1@(l_2@l_3) = (l_1@l_2)@l_3$
- [360] *For all* $p \in \mathbb{P}$*,* $turnRight(turnRight(turnRight(turnRight(p)))) = p$
- [Back&Forth] *For all* $n \in \mathbb{Z}$ *and* $p \in \mathbb{P}$*,* $move(n, move(-n, p)) = p$

3.3 Implementation

Instructions, itineraries, orientations and positions are implemented by the following SML types:

```
datatype instruction = Right
                     | Step of int
type itinerary = instruction list
datatype orientation = North | East | South | West
type position = int * int * orientation
```

These declarations are given to you in the starter file `robot.sml` of this homework.

Task 3.8 (2 points) Implement the SML function `turnRight: position -> position` that realizes the mathematical function *turnRight*.

Task 3.9 (2 points) Implement the SML function `move: int * position -> position` that realizes the mathematical function *move*.

Task 3.10 (2 points) Implement the SML function `getPosition: itinerary * position -> position` that realizes the mathematical function *getPosition*.

Task 3.11 (2 points) Implement the SML function `goBack: itinerary -> itinerary` that realizes the mathematical function *goBack*.

Task 3.12 (2 points) Write at least one test case in such a way that it witnesses the property *Go Back* above.

