Homework 02

15-150 Fall 2024

Due: Mon 9th Sept, 2024 at 8:00pm

LaTeX

The written part of your assignments must be written in LaTeX. To make your job easier, a LaTeX template is provided for each assignment where you simply need to fill in your solutions.

If you want to know what is the LaTeX name for a symbol, you can use Detexify. Also, Mathcha can help you type math. For constructs particular to this course (inductive definitions, proofs, etc), we provide a LaTeX guide with templates.

Finally, if you are just lost on how to type something or why your file is not compiling, reach out to the course staff.

Your submission will be penalized if it is not written using LaTeX. The applicable penalties are described in the style guide.

Code Structure

Check out the *basic requirements* for coding tasks in the style guide.

Proof Structure

Check out the *basic structure* for proofs in the style guide.

1 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.

1.1 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 difference between the size of t_R and the size of t_L.

Task 1.1 (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 1.2 (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 1.3 (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 node it visits. It may not use *iSize*.

Task 1.4 (2 points) Give an inductive definition of the function $\mathit{tiltLeft} : \mathbb{TT} \to \mathbb{TT}$ such that $\mathit{tiltLeft}(t)$ is the tree obtained from t by selectively swapping the left and right subtrees so that each inner node has a non-positive imbalance.

1.2 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 $\mathit{validate}(t) = \mathit{true}$, then $\mathit{iSize}'(t) = \mathit{iSize}(t)$.*

Task 1.5 (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 $\mathit{validate}(t) = \mathit{true}$, then $\mathit{validate}(\mathit{tiltLeft}(t)) = \mathit{true}$.*

Task 1.6 (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 $t \in \mathbb{TT}$, $\mathit{iSize}(\mathit{tiltLeft}(t)) = \mathit{iSize}(t)$.

Do these properties hold if the tree t is not valid, i.e., if we drop the assumption "if $\mathit{validate}(t) = \mathit{true}$"? How is this reflected in your proofs?

1.3 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.

Coding Task 1.7 (5 points) Implement the SML function `instrument: tree -> itree` such that `instrument(t)` is the instrumented tree corresponding to `t`.

Coding Task 1.8 (2 points) Implement the SML function `iSize: itree -> int` that realizes the mathematical function *iSize*.

Coding Task 1.9 (2 points) Implement the SML function `validate: itree -> bool` that realizes the mathematical function *validate*.

Coding Task 1.10 (2 points) Implement the SML function `iSize': itree -> int` that realizes the mathematical function iSize'.

Coding Task 1.11 (2 points) Implement the SML function `tiltLeft: itree -> itree` that realizes the mathematical function *tiltLeft*.

2 The SML-Robot

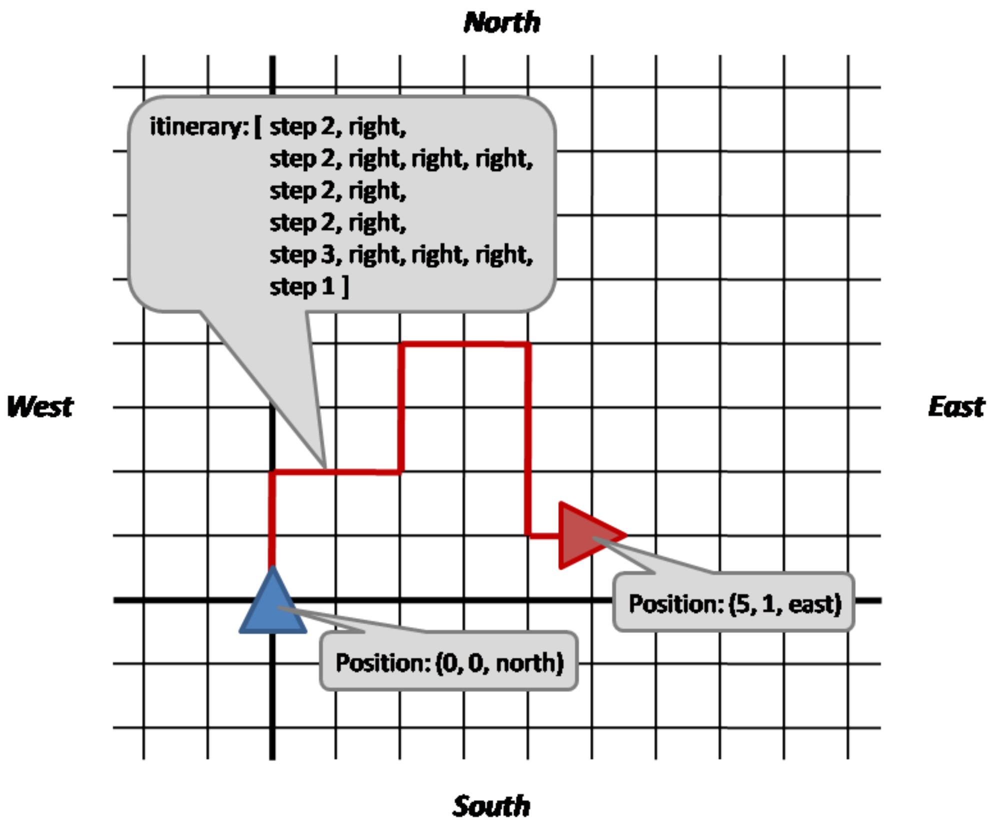

Figure 1: The SML Robot

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 forward n units. 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: $\mathbb{I} \;=\; \begin{cases} \textit{right} & \\ \textit{step } n & \text{where } n \in \mathbb{N} \end{cases}$

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

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

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

2.1 Operating the Robot

Task 2.1 (2 points) Define the function *turnRight* : $\mathbb{P} \rightarrow \mathbb{P}$ such that *turnRight*(p) returns the new position of the robot at position p when receiving the instruction *right*.

Task 2.2 (2 points) Define the function $move : \mathbb{N} \times \mathbb{P} \to \mathbb{P}$ such that $move(n, p)$ returns the new position of the robot at position p when receiving the instruction *step* n.

Task 2.3 (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.

2.2 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* $p \in \mathbb{P}$*,*

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

Task 2.4 (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 @).

After moving by means of *getPosition*, the robot can go back to its original position by turning around, following the exact same itinerary but in reverse, and then turning around again. 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.

Task 2.5 (2 points) Define the function $reverse : \mathbb{L}_{\mathbb{I}} \to \mathbb{L}_{\mathbb{I}}$ so that $reverse(l)$ returns the itinerary that reverses the moves in l, ignoring the initial and final turnarounds. For example,

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

returns the itinerary

$$[step\ 1,\ right,\ right,\ right,\ step\ 2]$$

(Convince yourself that this is indeed correct!)

Task 2.6 (1 points) Using *reverse*, 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. Therefore,

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

returns the itinerary

$$[right,\ right,\ step\ 1,\ right,\ right,\ right,\ step\ 2,\ right,\ right]$$

If you defined the function *reverse* correctly, then you should be able to prove that if a robot moves from p to p' with l, then turns around ending in position p'', then moves from p'' with $reverse(l)$, then it gets back to p except that it is facing the opposite direction it started from. This is expressed by the following mathematical property.

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

$$getPosition(l\ @\ [right, right]\ @\ reverse(l), p) \quad = \quad turnRight(turnRight(p))$$

Task 2.7 (3 points) What is the corresponding property for *goBack*? State it and prove it

2.3 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.

Coding Task 2.8 (2 points) Implement the SML function `turnRight: position -> position` that realizes the mathematical function *turnRight*.

Coding Task 2.9 (2 points) Implement the SML function `move: int * position -> position` that realizes the mathematical function *move*.

Coding Task 2.10 (2 points) Implement the SML function `getPosition: itinerary * position -> position` that realizes the mathematical function *getPosition*.

Coding Task 2.11 (2 points) Implement the SML function `reverse: itinerary -> itinerary` that realizes the mathematical function *reverse*.

Coding Task 2.12 (2 points) Implement the SML function `goBack: itinerary -> itinerary` that realizes the mathematical function *goBack*.

Coding Task 2.13 (2 points) Write at least one test case in such a way that it witnesses the property *Go Back* above.