

# Functional Data Structures

## Exercise Sheet 7

### Exercise 7.1 Round wrt. Binary Search Tree

The distance between two integers  $x$  and  $y$  is  $|x - y|$ .

1. Define a function  $round :: int\ tree \Rightarrow int \Rightarrow int\ option$ , such that  $round\ t\ x$  returns an element of a binary search tree  $t$  with minimum distance to  $x$ , and  $None$  if and only if  $t$  is empty.

Define your function such that it does no unnecessary recursions into branches of the tree that are known to not contain a minimum distance element.

2. Specify and prove that your function is correct. Note: You are required to phrase the correctness properties yourself!

Hint: Specify 3 properties:

- $None$  is returned only for the empty tree.
  - Only elements of the tree are returned.
  - The returned element has minimum distance.
3. Estimate the time of your  $round$  function to be linear in the height of the tree

```
fun round :: "int tree => int => int option"  
fun T_round :: "int tree => int => nat"
```

### Exercise 7.2 Interval Lists

Sets of natural numbers can be implemented as lists of intervals, where an interval is simply a pair of numbers. For example the set  $\{2, 3, 5, 7, 8, 9\}$  can be represented by the list  $[(2, 3), (5, 5), (7, 9)]$ . A typical application is the list of free blocks of dynamically allocated memory.

We introduce the type

```
type_synonym intervals = "(nat*nat) list"
```

Next, define an *invariant* that characterizes valid interval lists: For efficiency reasons intervals should be sorted in ascending order, the lower bound of each interval should

be less than or equal to the upper bound, and the intervals should be chosen as large as possible, i.e. no two adjacent intervals should overlap or even touch each other. It turns out to be convenient to define *inv* in terms of a more general function such that the additional argument is a lower bound for the intervals in the list:

```
fun inv' :: "nat  $\Rightarrow$  intervals  $\Rightarrow$  bool"
```

```
definition inv where "inv  $\equiv$  inv' 0"
```

To relate intervals back to sets define an *abstraction function*

```
fun set_of :: "intervals  $\Rightarrow$  nat set"
```

Define a function to add a single element to the interval list, and show its correctness

```
fun add :: "nat  $\Rightarrow$  intervals  $\Rightarrow$  intervals"
```

```
lemma add_correct_1:
```

```
"inv is  $\implies$  inv (add x is)"
```

```
lemma add_correct_2:
```

```
"inv is  $\implies$  set_of (add x is) = insert x (set_of is)"
```

Hints:

- Sketch the different cases (position of element relative to the first interval of the list) on paper first
- In one case, you will also need information about the second interval of the list. Do this case split via an auxiliary function! Otherwise, you may end up with a recursion equation of the form  $f(x\#xs) = \dots \text{case } xs \text{ of } x'\#xs' \Rightarrow \dots f(x'\#xs')$  ... combined with *split: list.splits* this will make the simplifier loop!

## Homework 7.1 Construction of a tree from a preorder and inorder traversals

*Submission until Thursday, June 23, 23:59pm.*

A tree can be uniquely identified by a preorder and an inorder traversals of its members. These traversals can be computed using the functions *preorder* and *inorder*.

```
thm preorder.simps
```

```
thm inorder.simps
```

One way to construct a tree from those two traversals is via the function *reconstruct* below:

```
reconstruct [] [] =  $\langle$ 
```

```
reconstruct (y # ys) zs = (let ls = lefts y zs; rs = rights y zs in  $\langle$ reconstruct (filter ( $\lambda y$ . y  $\in$  set ls) ys) ls, y, reconstruct (filter ( $\lambda y$ . y  $\in$  set rs) ys) rs)
```

```
value "reconstruct [3, 1, 6, 5, 8, 7, 9::int] [1, 3, 5, 6, 7, 8, 9]"
```

```
value "reconstruct [5, 3, 4, 1, 2] [1, 2, 3, 4, 5::int]"
```

Prove that this function works as desired.

**theorem** *reconstruct\_correct*:

*“distinct (preorder t)  $\implies$  reconstruct (preorder t) (inorder t) = t”*