A Compendium of Correct Proofs

1 Introduction

This document gives a number of example proofs to demonstrate our expected style for CS3110 proofs of correctness and efficiency.

Note: do not read the "Black Book of Bogus Proofs".

2 Proofs of correctness

2.1 Length $\geq$ 0

```
let rec length l = match l with
  | []     -> 0
  | h::tl -> 1 + length tl
```

Claim: for all `l : list`, `length l >= 0`.

Proof: By structural induction on `l`. There are two cases: (i) `l = []` and (ii) `l = h::tl`. In case (i), we see that `length l -->* 0`. Clearly `0 >= 0`, so this case is satisfied. In case (ii), we may assume inductively that `length tl >= 0`. Clearly, `length l -->* 1 + length tl`. Since `length tl >= 0`, `1 + length tl >= 0`, so `length l >= 0`.

2.2 Depth ≥ 0

```
type 'a tree = Node of 'a tree * 'a * 'a tree | Leaf

let rec depth t = match t with
  | Node (l, _, r) -> 1 + max (depth l) (depth r)
  | Leaf -> 0
```

Claim: for all `t : tree`, `depth t >= 0`.

Proof: By structural induction on `t`. There are two cases: (i) `t = Node (l, v, r)`, and (ii) `t = Leaf`

In case (i), we may inductively assume that `depth l >= 0` and `depth r >= 0`. In this case, we see that `depth t = 1 + max (depth l) (depth r)`. Since `max a b >= a` and `max a b >= b`, we see that `max (depth l) (depth r) >= (depth l) >= 0`. Therefore `1 + max (depth l) (depth r) >= 0` as required.

In case (ii), we see directly that `depth t --> 0 >= 0`, as required.

2.3 Inorder traversal preserves membership

```
let rec tree_mem x t = match t with
  | Node (l, v, r) -> if v = x
                      then true
                      else tree_mem x l || tree_mem x r
  | Leaf           -> false
```

```
let rec inorder_list t = match t with
  | Node (l, v, r) -> (inorder_list l) @ [v] @ (inorder_list r)
  | Leaf           -> []
```

Claim: for all `x` and `t`, if `tree_mem x t` then `List.mem x (inorder_list t)`.

For convenience, let $P(\mathrm{t})$ denote the proposition "if `tree_mem x t` then `List.mem x (inorder_list t)`".

Proof: By induction on the structure of `t`. There are two cases: (i) `t = Leaf`, and (ii) `t = Node(l,v,r)`.

In case (i), we see that `tree_mem x Leaf` is false, so $P(\mathrm{Leaf})$ is vacuously true.

In case (ii), we may assume inductively that $P(\mathrm{l})$ and $P(\mathrm{r})$ hold, and we wish to show that $P(\mathrm{t})$ holds (where `t = Node(l,v,r)`). Since $P(\mathrm{t})$ says "if `tree_mem x t` then ...", we assume that `tree_mem x t` is true, and we wish to show that `List.mem x (inorder_list t)` evaluates to true. By the definition of `inorder_list`, this is equivalent to

```
List.mem x ((inorder_list l)@[v]@(inorder_list r))
```

We have assumed that `tree_mem x t` is true; by examining the code, we see that there are three ways this could happen: either (a) `v = x`, (b) `tree_mem x l`, or (c) `tree_mem x r`.

In case (a), since `v = x`, it is clear that `List.mem x (_@[x]@_)`, as required.

In case (b), we can apply our inductive hypothesis: we know that `tree_mem x l` so we can conclude that `List.mem x (inorder_list l)`. Therefore (by the specification for `List.mem`), `List.mem x ((inorder_list l)@_@_)` as required.

Case (c) is completely analogous to case (b): we have `tree_mem x r`, so we can apply $P(\mathrm{r})$ to conclude `List.mem x (inorder_list r)`, and thus `List.mem x (_@_@(inorder_list r))`, as required.

We have now considered every possible case, so we conclude that whenever `tree_mem x t` holds, `List.mem x (inorder_li` is also true. This completes our second inductive case, and thus our inductive proof.

2.4 BST

```
let rec bst_insert t x =
  match t with
  | Leaf -> Node (Leaf, x, Leaf)
  | Node (l, v, r) ->
    if v <= x then Node (l, v, insert r x)
    else Node (insert l x, v, r)
```

Define the function C from BSTs to $\mathcal{P}(\mathbb{Z})$ as follows:

$$C(Leaf) = \{\}$$
$$C(Node(l,v,r)) = C(l) \cup C(r) \cup \{v\}$$

We say that a BST t is *valid* if $t = Leaf$ or if $t = Node(l,v,r)$, then for all $x \in C(l)$ and $y \in C(r)$, $x < v \leq y$.

Claim: Let t be a valid BST with elements from $\mathbb{Z}$ and $x \in \mathbb{Z}$. Denote t' by $t' = insert\ t\ v = Node(l,v,r)$. Then $C(t') = C(t) \cup \{v\}$, and t' is a valid BST.

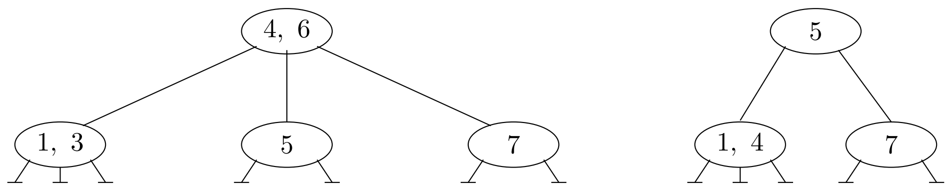
Proof: The proof is by structural induction on t. Suppose that $t = Leaf$. By the substitution model, $t' = Node(Leaf, x, Leaf)$, and by definition of C, $C(t') = \{x\} = C(t) \cup \{x\}$. Clearly t' is a valid BST, so the base case holds. Suppose that $t = Node(l,v,r)$ is a valid BST. Suppose that $v \leq x$. By the substitution model, $t' = Node(l, v, insert\ r\ x)$. Denote t'' by $t'' = insert\ r\ x$. By induction hypothesis on t'', $C(t'') = C(r) \cup \{x\}$, so by definition of C, $C(t') = C(l) \cup C(t'') = C(l) \cup C(r) \cup \{v\} \cup \{x\} = C(t) \cup \{x\}$ as desired. Furthermore, note that for $x \in C(l)$, $x < v$ by assumption of t being a valid BST. By the same argument, for all $y \in C(r)$, $v \leq y$, and given $v \leq x$, for all $z \in C(t'') = C(r) \cup \{x\}$, $v \leq z$, so t' is a valid BST. The case when $v > x$ is completely symmetric. Thus the inductive step holds, which finishes the proof.

2.5 Tree Induction

A **2-3 tree** is a tree in which every node has either two or three children. Here is the OCaml type we will be using for 2-3 trees:

```
type 'a tree23 =
  | Leaf
  | Node2 of 'a tree23 * 'a * 'a tree23
  | Node3 of 'a tree23 * 'a * 'a tree23 * 'a * 'a tree23
```

Here are some example 2-3 trees:



In the following exercise we will be considering **2-3 search trees**, which obey the following invariants. Note that the examples above are 2-3 search trees.

1. 2-nodes have a single value and two subtrees. All values in the left subtree are less than the node's value and all values in the right subtree are greater than the node's value. 2-nodes are effectively identical to regular binary search tree nodes.

2. 3-nodes have two values and three subtrees. All values in the left subtree are less than both of the node's values. All values in the right subtree are greater than both of the node's values. All values in the *middle* subtree are greater than the node's smaller value and less than the node's bigger value.

To make our code little tidier, we define the following type and function for comparing arbitrary values:

```
type comparison = LT | GT | EQ

let simple_compare (x:'a) (y:'a) : comparison =
  let z = compare x y in
  if z < 0 then LT
  else if z > 0 then GT
  else EQ
```

With an order relation in hand the next step in implementing the search tree is implementing a "contains" function that determines whether a value is present in a tree. You could scan the entire tree, comparing every value with the value you are searching for, but this would be terribly inefficient. The whole point of search trees is to avoid this kind of inefficiency. We can avoid this inefficiency by using the search tree invariants.

Here is an OCaml implementation of a contains function:

```
let rec contains23 (x : 'a) (t : 'a tree23) : bool =
  match t with
  | Leaf -> false
  | Node2 (t1,x1,t2) ->
    (match simple_compare x x1 with
     | LT -> contains23 x t1
     | EQ -> true
     | GT -> contains23 x t2)
  | Node3 (t1,x1,t2,x2,t3) ->
    (match (simple_compare x x1, simple_compare x x2) with
    | (LT,LT) -> contains23 x t1
```

```
    | (GT,LT) -> contains23 x t2
    | (GT,GT) -> contains23 x t3
    | (EQ,LT) | (GT,EQ) -> true
    | _ -> failwith "Malformed tree" )
```

We will use induction to prove that `contains23` returns true if the value `x` exists somewhere in the tree `t` and false otherwise.

More formally, let $P(t)$ be the statement that for `(t : a tree23)` that satisfies the invariants, `(contains23 x t)` returns true if `x` is equal to a value contained in `t` and false otherwise for all `(x:a)`.

Base case: For `(t : s tree23) = Leaf` and an arbitrary `(x : a)`

`contains23 x t` $\to^*$ `false`

Since `t` is a `Leaf`, it does not contain any value so `x` does not exist anywhere in `t`. Hence P(`Leaf`) holds.

Inductive Hypothesis:

Suppose P(`t1`), P(`t2`) and P(`t3`) all hold for some `(t1,t2,t3 : a tree23)` that satisfies the invariants.

Inductive Step:

We will prove that for all `(x1, x2 : a )` such that `x1` is greater than all the values in `t1` and less than all the values in `t2`, and `x2` is greater than all the values in `t2` and less than all the values in `t3`, if `t4 = Node2 (t1,x1,t2)` and `t5 = Node3 (t1,x1,t2,x2,t3)` then `P(t4)` and `P(t5)` also hold.

`contains23 x t4` $\to^*$ (*)

```
match simple_compare x x1 with
  | LT -> contains23 x t1
  | EQ -> true
  | GT -> contains23 x t2}
```

There are 3 cases each corresponds to a match case:

- Case 1: `simple_compare x x1` evaluates to `LT` then (*) evaluates to `contains23 x t1`.

  If `contains23 x t1` returns `true` then `x` exists in `t1` by the induction hypothesis so `x` exists in `t4`.

  If `contains23 x t1` returns `false` then `x` does not exist in `t1`.

  `x` also does not exist in `t2` because `x` < `x1` < all the values in `t2` so `x` does not exist in `t4`.

- Case 2: `simple_compare x x1` evaluates to `EQ` then (*) evaluates to `true`. In this case `x` exists in `t4` as it is exactly `x1`.

- Case 3: `simple_compare x x1` evaluates to `GT` then (*) evaluates to `contains23 x t2`.

  If `contains23 x t2` returns `true` then `x` exists in `t2` by the induction hypothesis so `x` exists in `t4`.

  If `contains23 x t2` returns `false` then `x` does not exist in `t2`.

  `x` also does not exist in `t1` because `x` > `x1` > all the values in `t2` so `x` does not exist in `t4`.

  In all cases P(`t4`) holds.

`contains23 x t5` $\to^*$ (**)

```
match (simple_compare x x1, simple\_compare x x2) with
  | (LT,LT) -> contains23 x t1
  | (GT,LT) -> contains23 x t2
  | (GT,GT) -> contains23 x t3
  | (EQ,LT) | (GT,EQ) -> true
  | _ -> failwith "Malformed tree"
```

- Case 1: `simple_compare x x1` and `simple_compare x x2` both evaluate to `LT` then (**) evaluates to `contains23 x t1`.

  If `contains23 x t1` returns `true` then `x` exists in `t1` by the induction hypothesis so `x` exists in `t5`.

If `contains23 x t1` returns `false` then `x` does not exist in `t1`.

`x` does not exist in `t2` because `x` < `x1` < all the values in `t2`.

`x` does not exist in `t3` because `x` < `x2` < all the values in `t3`.

So `x` does not exist in `t5`.

- Case 2: `simple_compare x x1` evaluates to `EQ` and `simple_compare x x2` evaluates to `LT` then (**) evaluates to `contains23 x t2`.

  If `contains23 x t2` returns `true` then `x` exists in `t2` by the induction hypothesis so `x` exists in `t5`.

  Else `contains23 x t2` returns `false` then `x` does not exist in `t2`.

  `x` does not exist in `t1` because `x` > `x1` > all the values in `t2`.

  `x` does not exist in `t3` because `x` < `x2` < all the values in `t3`.

  So `x` does not exist in `t5`.

- Case 3: `simple_compare x x1` and `simple_compare x x2` both evaluate to `GT` then (**) evaluates to `contains23 x t3`.

  If `contains23 x t3` returns `true` then `x` exists in `t3` by the induction hypothesis so `x` exists in `t5`.

  Else `contains23 x t3` returns `false` then `x` does not exist in `t2`.

  `x` does not exist in `t1` because `x` > `x1` > all the values in `t2`.

  `x` does not exist in `t3` because `x` > `x2` > all the values in `t3`.

  So `x` does not exist in `t5`.

- Case 4: Either `simple_compare x x1` or `simple_compare x x2` evaluates to `EQ`, since `x1` < `x2` it follows that `simple_compare x x1, simple_compare x x2` must match either `(EG,LT)` or `(LT,EG)` so (**) evaluates to `true`. In this case `x` exists in `t4` as it is either `x1` or `x2`.

- Case 5: `x1` is greater than all the values in `t1` and less than all the values in `t2`, `x2` is greater than all the values in `t2` and less than all the values in `t3`, and `t1, t2, t3` all satisfy the invariants. It follows that `simple_compare x x1,simple_compare x x2` cannot match neither one of these patterns: `(LT,GT) (LT,EQ) (EQ,EQ) (EQ,GT)`. Hence (**) does not fail

Conclusion: From the base case, inductive hypothesis and inductive step, by the principal of structural induction P(`t`) holds for all `(t : a tree23)`.

3 Proofs of worst-case complexity

See Recitation 15.

4 Proofs of amortized complexity

See Lecture 16.