Program Verification Exam

PROJECT A: IVL1 ➔ DSA
PROJECT B: BINARY SEARCH TREES

Leila Dardouri
Group 13

08/12/2023

PROJECT A:
IVL1 → DSA

IVL1 ➔ DSA

Every assignment on a path through C assigns to a fresh variable.

So, we need to:

- Introduce **multiple versions** of each variable
    - **Left-side** of **assignment** is always **fresh**
    - Make sure to always **read latest version**

- In C1 [ ] C2, **synchronize choices** on the last version of the variables

IVL1 ➔ DSA: example

```
method foo(x: Int) returns (y: Int)
    requires x > 0
    ensures y > x
{
    var z: Int := x + 1

    if(z > 20){
        y := z + 1
    } else {
        z := z + z
        y := z + 2
        y := y + 3
    }
    y := y + 4
}
```

```
-----IVL1----
method foo{
my_x_foo: Int
my_y_foo: Int
orig_my_x_foo: Int := my_x_foo
assume my_x_foo > 0
my_z_foo: Int := my_x_foo + 1
{assume my_z_foo > 20
my_y_foo := my_z_foo + 1}
[]
{assume !(my_z_foo > 20)
my_z_foo := my_z_foo + my_z_foo
my_y_foo := my_z_foo + 2
my_y_foo := my_y_foo + 3}
my_y_foo := my_y_foo + 4
assert my_y_foo > orig_my_x_foo
}
```

```
----------DSA---------
method foo{
my_x_foo_0: Int
my_y_foo_0: Int
orig_my_x_foo_0: Int
my_z_foo_1: Int
my_y_foo_1: Int
my_y_foo_2: Int
my_z_foo_0: Int
orig_my_x_foo_0 := my_x_foo_0
assume my_x_foo_0 > 0
my_z_foo_0 := my_x_foo_0 + 1
{assume my_z_foo_0 > 20
my_y_foo_0 := my_z_foo_0 + 1
my_z_foo_1 := my_z_foo_0
my_y_foo_1 := my_y_foo_0}
[]
{assume !(my_z_foo_0 > 20)
my_z_foo_1 := my_z_foo_0 + my_z_foo_0
my_y_foo_0 := my_z_foo_1 + 2
my_y_foo_1 := my_y_foo_0 + 3}
my_y_foo_2 := my_y_foo_1 + 4
assert my_y_foo_2 > orig_my_x_foo_0
}
```

IVL1 ➔ DSA: update assignments

To keep track of the variables versions:

id   counter

```
varsMap: Map<string, int>
```

```
// translates a IVL1 statement into a DSA statement
let rec IVL1_stmt_to_DSA (cmd: v1Statement, varsMap: Map<string, int>) : DSAStatement list * Map<string, int> =
    match cmd with
        | V1Assignment(i, e) ->

        let (e', varsMap') = update_vars_in_expression (Some(i), e, varsMap)

        let count =
            if Map.containsKey i varsMap' then
                Map.find i varsMap' + 1
            else
                0

        let varsMap' = Map.add i count varsMap

        ([ DSAAssignment($"{i}_{count}", e') ], varsMap')
```

IVL1 ➔ DSA: update expressions

```
// update the vars ids in an expression, according to the varsMap, to translate them from IVL1 into DSA
let rec update_vars_in_expression
    (
        idAssign: ident option,
        e: expr,
        varsMap: Map<string, int>
    ) : expr * Map<string, int> =
    match e with
    | Ref id ->
        let count =
            if Map.containsKey id varsMap then
                Map.find id varsMap
            else
                0

        let varsMap' = Map.add id count varsMap

        (Ref($"{id}_{count}"), varsMap')
```

IVL1 ➔ DSA: syncronize choice branches

```
let rec fix_choice_branches (c1VarsMap, c2VarsMap) =
    let c1Seq = Map.toSeq c1VarsMap |> List.ofSeq // convert maps into seq to iterate over them
    let c2Seq = Map.toSeq c2VarsMap |> List.ofSeq

    let c1StmtList = add_assignments_to_choice_branch1 (c1Seq, c2Seq, [])
    let c2StmtList = add_assignments_to_choice_branch2 (c1Seq, c2Seq, [])

    (c1VarsMap, c2VarsMap, c1StmtList, c2StmtList)
```

IVL1 ➔ DSA: syncronize choice branches

```
// add the assignements needed in the first branch of the choice to align the variables counter in the branches
let rec add_assignments_to_choice_branch1 (c1Seq, c2Seq, c1StmtList) =
    match c1Seq, c2Seq with
    | _, (id2, c2Value) :: rest2 ->
        match find_value_by_id id2 c1Seq with
        | Some c1Value when c2Value > c1Value ->
            let c1Stmt = DSAAssignment($"{id2}_{c2Value}", Ref($"{id2}_{c1Value}"))
            add_assignments_to_choice_branch1 (c1Seq, rest2, c1Stmt :: c1StmtList)

        | Some c1Value when c2Value <= c1Value -> add_assignments_to_choice_branch1 (c1Seq, rest2, c1StmtList)

        | _ ->
            let c1Stmt = DSAAssignment($"{id2}_{c2Value}", Ref($"{id2}_{0}"))

            add_assignments_to_choice_branch1 (
                c1Seq,
                rest2,
                if (c2Value <> 0) then c1Stmt :: c1StmtList else c1StmtList
            )
    | _ -> (c1StmtList)
```

PROJECT B:
BINARY SEARCH TREES

Binary Search Trees (BST): definition

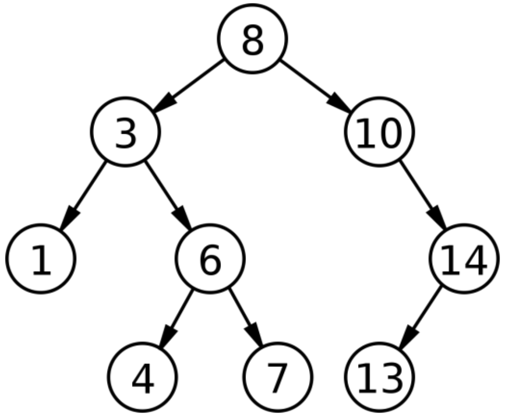
```
predicate bst(node: Ref) {
    acc(node.elem) && acc(node.left) && acc(node.right) &&

    (node.left != null ==> bst(node.left) &&
     |tree_content(node.left)| > 0 &&
      node.elem > max_seq(tree_content(node.left))) &&

    (node.right != null ==> bst(node.right) &&
     |tree_content(node.right)| > 0 &&
      node.elem < min_seq(tree_content(node.right)))
}
```

```
function max_seq(s : Seq[Int]) : Int
    requires |s| > 0
    ensures Seq(s[0]) ++ s[1..] == s
    ensures forall x : Int :: x in s ==> x <= result
    ensures result in s
    {
        |s| == 1 ? s[0] : max(s[0], max_seq(s[1..]))
    }
```

```
field elem : Int
field left : Ref
field right : Ref
```




```
function tree_content(tree: Ref) : Seq[Int]
    requires bst(tree)
    ensures |result| > 0
    {
        unfolding bst(tree) in
        (tree.left == null && tree.right == null) ? Seq(tree.elem) :
        (tree.left == null && tree.right != null) ? Seq(tree.elem) ++ tree_content(tree.right) :
        (tree.left != null && tree.right == null) ? tree_content(tree.left) ++ Seq(tree.elem) :
         tree_content(tree.left) ++ Seq(tree.elem) ++ tree_content(tree.right)
    }
```

Binary Search Trees (BST): insert

```
method bst_insert(tree: Ref, val: Int)
 requires bst(tree)
 requires acc(time_credit(), (height(tree) + 1)/1)
 ensures bst(tree)
 ensures old(seq_to_set(tree_content(tree))) union
           Set(val) == seq_to_set(tree_content(tree))
 {
     consume_time_credit()
     unfold bst(tree)

     if(val < tree.elem) {
         if(tree.left == null) {
                 var new_tree_left : Ref := new(*)
                 new_tree_left.elem := val
                 new_tree_left.left := null
                 new_tree_left.right := null

                 tree.left := new_tree_left
                 fold bst(tree.left)
         } else {
             bst_insert(tree.left, val)
         }
     } else {
       if(val > tree.elem) { ... }
     }
     fold bst(tree)
 }
```

```
function height(tree: Ref) : Int
    requires bst(tree)
    ensures result >= 0
    {
        unfolding bst(tree) in (tree.left == null) ?
            ((tree.right == null) ? 0 : 1 + height(tree.right)) :
               ((tree.right == null) ? 1 + height(tree.left) :
                                        1 + max(height(tree.left), height(tree.right)))
    }
```

THANK YOU