Verification in Dafny

Nicekingwei 2017.10.15

Language Specifications

- Variables & Fields

```
var x:Tree;
var y:nat := 0;
var a, b, c := 1, 2, true;
```

- ADT

```
datatype Tree = Empty | Node(left:Tree,value:real,right:Tree)
datatype List<T> = Nil | Cons(head: T, tail: List<T>)
Cons(5, Nil).Cons? && Cons(5, Nil).head == 5
Cons(4, Nil)[tail := Cons(3, Nil)] == Cons(4, Cons(3, Nil))
(true,5).1
```

Language Specifications

- Method

```
method/lemma Abs(x: int) returns (y: int)
        modifies <col>
        requires <expr>
        ensures  <expr>
        decreases <expr>
{
        if x < 0 { return -x;}
        else{ return x;}
}
```

Language Specifications

- Function

```
function/function method abs(x: int): int
        requires <expr>
        ensures <expr>
        reads <col>
        decreases <expr>
{
        if x < 0 then -x else x
}
```

Language Specifications

- Assert/Assume & Forall & Exist

```
assert forall x :: P(x) ==> Q(x)
assert forall (i | 0 <= i < n - m) { b[i] := a[m + i];}
assert forall x | P(x) { Lemma(x); }
assert forall k | P(k) ensures Q(k) { Statements; }
assert (forall n :: 2 <= n ==> (exists d :: n < d && d < 2*n))
```

Language Specifications

- Match

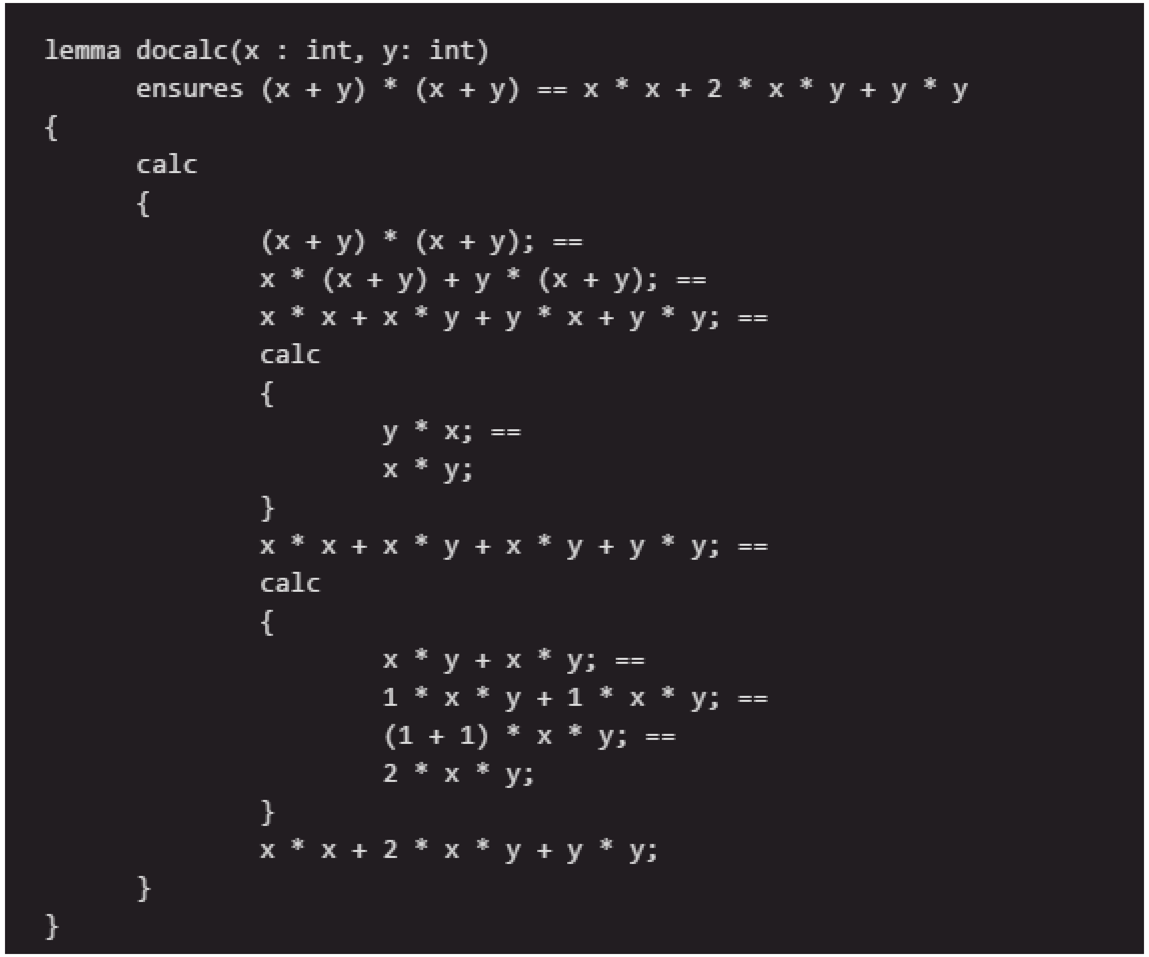
```
match t
        case Empty => …
        case Node(l:Empty,v,r) => …
```

- While & Invariant

```
var i:=0
while i<n
        invariant i<=n
        decreases n-i
{}
```

Language Specifications

- Cal

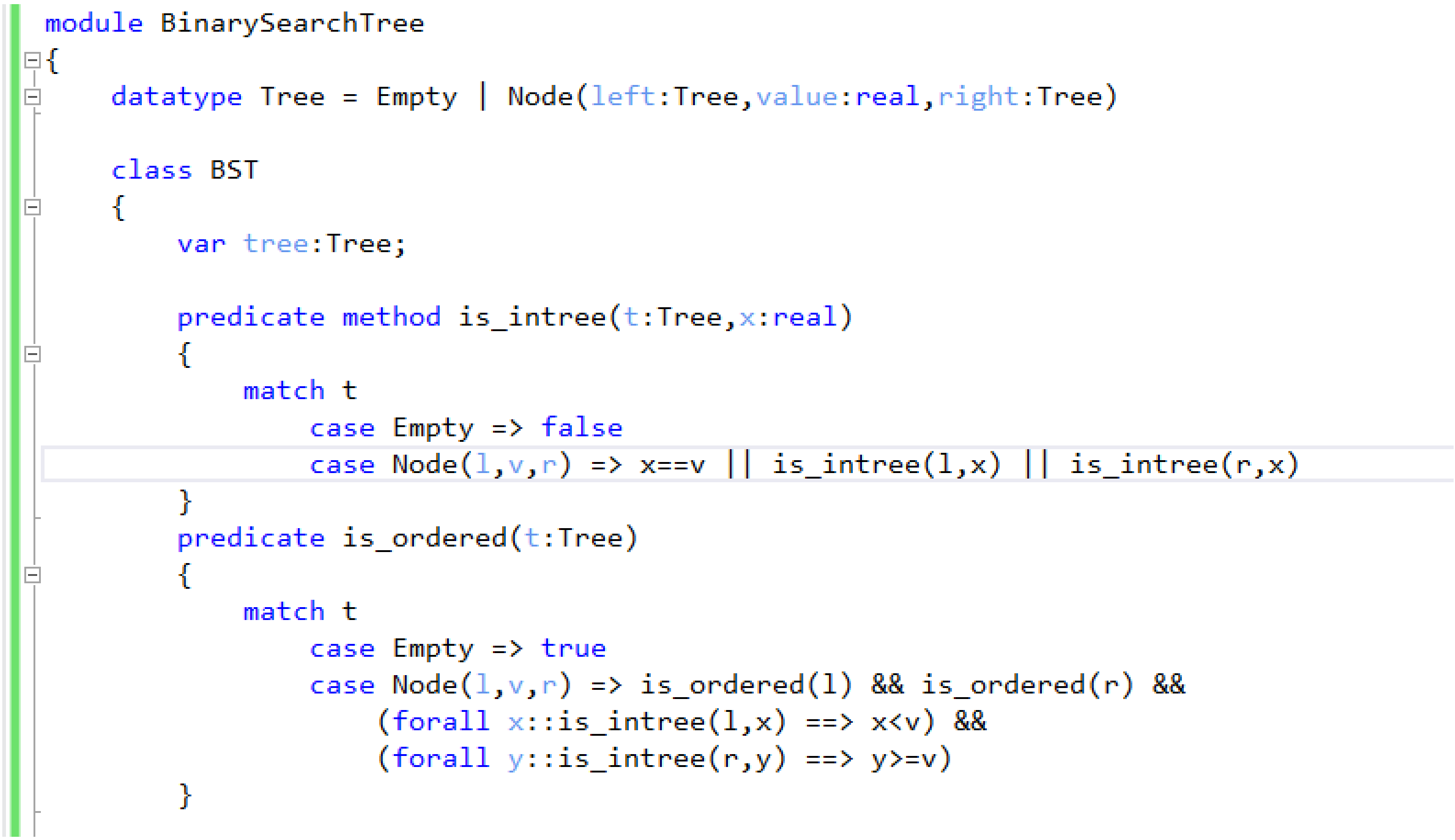


```
lemma docalc(x : int, y: int)
      ensures (x + y) * (x + y) == x * x + 2 * x * y + y * y
{
      calc
      {
              (x + y) * (x + y); ==
              x * (x + y) + y * (x + y); ==
              x * x + x * y + y * x + y * y; ==
              calc
              {
                      y * x; ==
                      x * y;
              }
              x * x + x * y + x * y + y * y; ==
              calc
              {
                      x * y + x * y; ==
                      1 * x * y + 1 * x * y; ==
                      (1 + 1) * x * y; ==
                      2 * x * y;
              }
              x * x + 2 * x * y + y * y;
      }
}
```

Language Specifications(haven’t covered)

- fresh/old
- inductive coinductive
- module system
- trait
- iterator
- type paramerters
- lambda

Example:Binary Search Tree

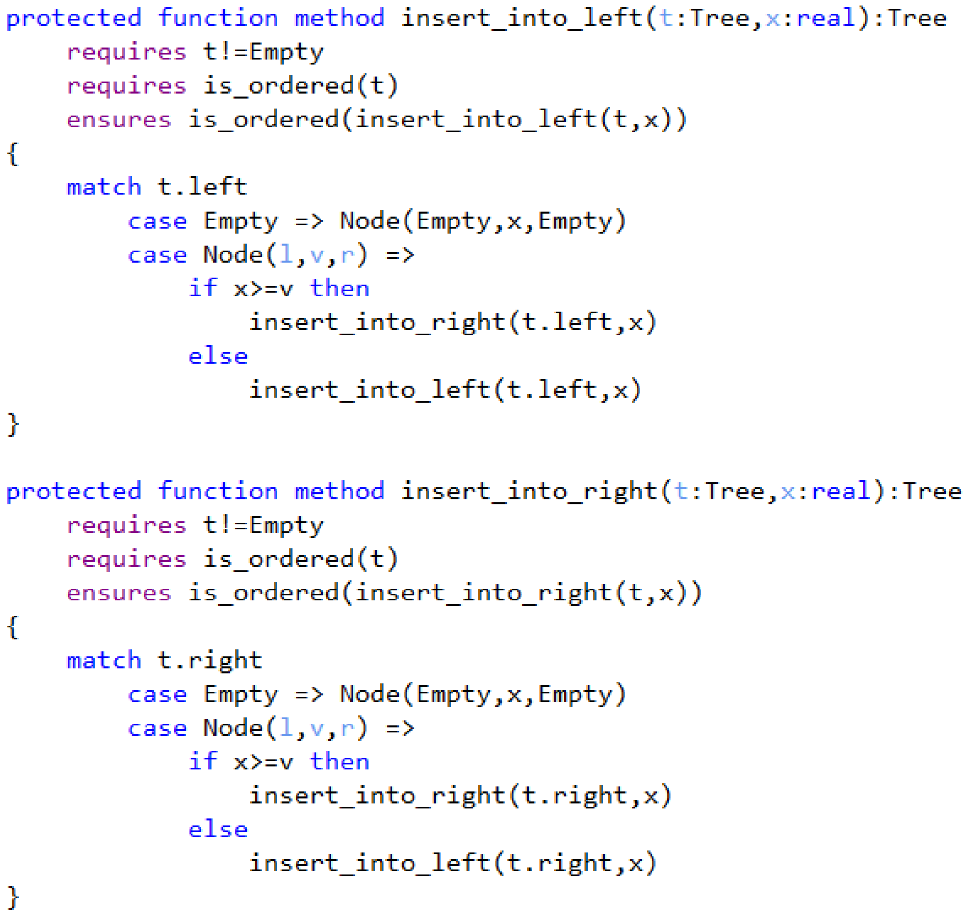


```
module BinarySearchTree
{
    datatype Tree = Empty | Node(left:Tree,value:real,right:Tree)

    class BST
    {
        var tree:Tree;

        predicate method is_intree(t:Tree,x:real)
        {
            match t
                case Empty => false
                case Node(l,v,r) => x==v || is_intree(l,x) || is_intree(r,x)
        }
        predicate is_ordered(t:Tree)
        {
            match t
                case Empty => true
                case Node(l,v,r) => is_ordered(l) && is_ordered(r) &&
                    (forall x::is_intree(l,x) ==> x<v) &&
                    (forall y::is_intree(r,y) ==> y>=v)
        }
```

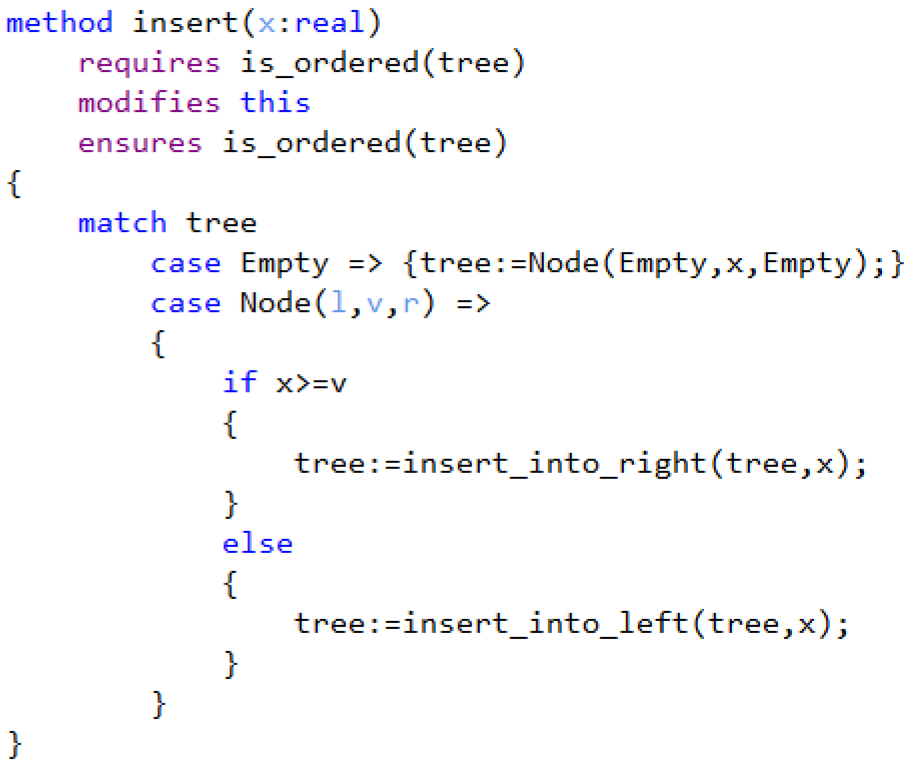
Example:Binary Search Tree



```
protected function method insert_into_left(t:Tree,x:real):Tree
    requires t!=Empty
    requires is_ordered(t)
    ensures is_ordered(insert_into_left(t,x))
{
    match t.left
        case Empty => Node(Empty,x,Empty)
        case Node(l,v,r) =>
            if x>=v then
                insert_into_right(t.left,x)
            else
                insert_into_left(t.left,x)
}

protected function method insert_into_right(t:Tree,x:real):Tree
    requires t!=Empty
    requires is_ordered(t)
    ensures is_ordered(insert_into_right(t,x))
{
    match t.right
        case Empty => Node(Empty,x,Empty)
        case Node(l,v,r) =>
            if x>=v then
                insert_into_right(t.right,x)
            else
                insert_into_left(t.right,x)
}
```

Example:Binary Search Tree



```
method insert(x:real)
    requires is_ordered(tree)
    modifies this
    ensures is_ordered(tree)
{
    match tree
        case Empty => {tree:=Node(Empty,x,Empty);}
        case Node(l,v,r) =>
        {
            if x>=v
            {
                tree:=insert_into_right(tree,x);
            }
            else
            {
                tree:=insert_into_left(tree,x);
            }
        }
}
```

Open Problems

- Why not just define ‘insert_into’ instead of ‘insert_into_left’ and ‘insert_into_right’?

  My answer is about path.

- How to implement the function ‘delete’?

  I have implemented one,but there is an assuming expression(o(╥﹏╥)o)

- What algorithm does the verifier take?What is Z3?What is SMT Solver?

Learning Dafny And Having Fun

- https://github.com/NiceKingWei/algorithm/tree/master/dafny
- Code
- Exercise
- Module
- Notes

References & Resources

- Dafny Tutorial / Verification Corner

https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/

- Github

https://github.com/Microsoft/dafny

- DafnyRef

https://github.com/Microsoft/dafny/blob/master/Docs/DafnyRef/out/DafnyRef.pdf