Datatypes and Functional Programming in Dafny

Datatypes

- For instance, to define a binary tree:

```
datatype Tree = Leaf | Node(Tree, Tree)
```

- Declare type

```
datatype D<T> = Ctor1 | Ctor2 | ...

datatype List<T> = Nil | Cons(head: T, tail: List<T>)
datatype Semaphore = Green | Yellow | Red
```

- Construct instance

```
var list1 := Cons(5, Nil); var sem1 := Green;
```

- Update an instance

```
d[CtorParam := Value]

list1 := list1[head := 1];
```

- Constructor check

```
d.Ctor?
```

- Field selector

```
d.CtorParam

function Length(x: List<T>) : nat {
if x.Cons? then 1 + Length(x.tail) else 0
}
```

- Case analysis:

  Match expression (for a match statement, use "=> stmt;" instead of "=> expression") {} are optional

```
function Length(x: List<T>) : nat { match x {
case Nil => 0
case Cons(h, t) => 1 + Length(t) } }
```

Copy tree values to an array

We may start in a given array position and transverse the tree in preorder.

```
type T = int
datatype Tree<T> = Leaf | Node(Tree, T, Tree)
method Fill(t: Tree<T>, a: array<T>, start: int)
                returns (end: int)
{
match t {
    case Leaf => end := start;
    case Node(left, x, right) =>
      end := Fill(left, a, start);
    if end < a.Length {
        a[end] := x;
        end := Fill(right, a, end + 1);
        }
    }
}
```

Verify the correctness of Fill

```
function contains<T(==)>(t: Tree<T>, v: T): bool
{
 match t {
   case Leaf => false
   case Node(left, x, right) =>
    x == v || contains(left, v) || contains(right, v)
 }
}

method Main()
{ var q := Node(Node(Leaf,4,Leaf),5, Node(Leaf,3,Leaf));
  var t := new int[10];
  var n := Fill(q,t,0);
  assert contains(q,3);
  print t[0], t[1], t[2];
}
```

Pre and Pos Conditions

```
method Fill(t: Tree<T>, a: array<T>, start: int)
          returns (end: int)
requires 0 <= start <= a.Length
decreases t
```

```
modifies a
ensures  start <= end <= a.Length
ensures forall i:: 0 <= i < start ==> a[i] == old(a[i])
ensures forall i:: start <= i < end ==> contains(t, a[i])
```

Field selectors/Destructors

To access the fields of a datatype one can name them

```
type T = int
datatype Tree<T> = Leaf | Node(left:Tree, key:T, right:Tree)
```

and

```
function countLeaf(t: Tree<T>): nat {
 if t.Leaf? then 1
 else countLeaf(t.left) + countLeaf(t.right)
}
```

Lists

```
datatype List<T> = Nil | Cons(head: T, tail: List<T>)
function Length<T>(xs: List<T>): nat {
  match xs
  case Nil => 0
  case Cons(_, tail) => 1 + Length(tail)
}
function Length'<T>(xs: List<T>): nat {
  if xs == Nil then 0 else 1 + Length'(xs.tail)
}

lemma  LengthLength'<T>(xs: List<T>)
  ensures Length(xs) == Length'(xs)
{}
```

See file about Lists

Unary Numbers

```
datatype Nat = Z | S(Nat)

function   Plus(x:Nat, y:Nat): Nat
decreases x;
{
match x {
```

```
    case Z => y
    case S(z) => S(Plus(z,y))
}
}

lemma {:induction n} example(n:Nat)
decreases n;
ensures Plus(n,Z) == n
{}

lemma {:induction false} example0 (n: Nat)
decreases n
ensures Plus(n,Z) == n
{
    match n {
    case Z => { assert Plus(Z,Z) == Z;}
    case S(k) => {
        assert Plus(n,Z) == S(Plus(k,Z));
        example0(k);
        assert S(Plus(k,Z)) == S(k);
        assert  S(k) == n;
        }
    }

}
```

See file about unary numbers