Verification-Driven Program Development
Exercise Sheet, Week 4

Ștefan Ciobâcă

23.10.2024

References

1. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/krml190.pdf
2. https://dafny.org/teaching-material/Lectures/1-3-Programming-ObjectOriented.html#/

Exercises

1. Fill in the missing bits in following class:

```
class Cell
{
  var contents : int

  constructor()
    ensures contents == 0
  {
    contents := 0;
  }

  constructor fromInt(x : int)
    ...
  {
    ...
  }

  method setContents(x : int)
    ...
  {
    ...
  }

  method getContents() returns (r : int)
  {
    return contents;
  }
}
```

2. Fill in the method `setBoth`:

```
method setCell(cell : Cell, x : int)
  modifies cell
  ensures cell.contents == x
{
  cell.setContents(x);
}

method setBoth(cell1 : Cell, x : int, cell2 : Cell, y : int)
  ...
{
  ...
}

method test1()
{
  var c1 := new Cell.fromInt(9);
  assert c1.contents == 9;
  var c2 := new Cell();
  setBoth(c1, 10, c2, 13);
  assert c1.contents == 10;
  assert c2.contents == 13;
}
```

3. Here is how to create class invariants in Dafny:

```
class EvenNumber
{
  var x : int

  predicate Valid()
    reads this
  {
    x % 2 == 0
  }

  constructor()
    ensures Valid()
  {
    x := 0;
  }

  method increment()
    requires Valid()
    modifies this
    ensures Valid()
  {
    x := x + 2;
  }

  method square()
    requires Valid()
    modifies this
    ensures Valid()
  {
    x := x * 2;
```

```
  }

  method reset()
    requires Valid()
    modifies this
    ensures Valid()
  {
  }

  method getContents() returns (r : int)
    requires Valid()
    ensures r % 2 == 0
  {
    return x;
  }
}
```

Fill in the missing parts in the following class:

```
class NaturalCounter
{
  var counter : int // should be a natural (i.e., >= 0)

  predicate Valid()
    ...
  {
    ...
  }

  constructor()
    ...
  {
    ...
  }

  method increment()
    ...
  {
    ...
  }

  method decrement() ...
    ...
  {
    ...
  }
}
```

4. Fill in the following method:

```
method incrementCounter(c : NaturalCounter?)
  ...
{
  if c != null {
    c.increment();
```

```
  }
}
```

5. Fill in the missing parts in the following class:

```
class Node
{
  var info : int
  var next : Node?

  ghost var footprint : set<Node>
  ghost var contents : seq<int>

  ghost predicate Valid()
    reads this
    reads footprint
    decreases footprint
  {
    this in footprint &&
      (next == null ==> contents == [ info ] ) &&
      (next != null ==>
      (next in footprint &&
      next.footprint < footprint &&
      this !in next.footprint &&
      contents == [info] + next.contents &&
      next.Valid()
      )
      )
  }

  constructor(i : int)
    ensures Valid()
    ensures next == null
    ensures footprint == { this }
    ensures contents == [ i ]
  {
    info := i;
    next := null;
    footprint := { this };
    contents := [ i ];
  }

  method push_front(info : int) returns (result : Node)
    ...
  {
    ...
  }

  method search(info : int) returns (result : Node?)
    ...
  {
    ...
  }
}
```

6. Make `Cell`, `NaturalCounter`, and `Node` extend the following trait:

```
trait Printable
{
  method Print()
}
```

7. Add a method to append an element to the end of the list.

8. Add a method to reverse the list (do not create a new list, just reverse the `next` pointers).

9. Specify, implement, and verify, in object-oriented style, one of the following data structures, at your choice:

   (a) A stack;

   (b) A queue;

   (c) A vector (like an array, but can shrink/grow dynamically, as needed);

   (d) A deque;

   (e) A binary search tree (for extra credit, make it Red/Black, AVL, etc.);

   (f) A skiplist;

   (g) A Bloom filter;

   (h) A heap.