15-819O Program Analysis

Spring 2013

Jonathan Aldrich

Topic: OO Verification with Dafny

Readings:

K. Rustan M. Leino. Specification and Verification of Object-Oriented Software. Lecture notes for the Marktoberdorf International Summer School, 2008.

Reading Question Set 3 [due March 27th, 2013 at 11:59am]

Read the paper cited above. Answer the questions below in a text file that you email to the instructor (aldrich@cs.cmu.edu) with subject “RQ 3”:

1. Download Dafny or try it on rise4fun.com. The sum of the odd numbers from 1 through $n*2-1$ is equal to n^2. Write a program that computes n^2 using this series. Check the correctness of your program using Dafny, i.e. that it really computes n^2. Your solution should use appropriate requires, ensures, and invariant clauses. Turn in your code.

2. In Dafny, define a linked list class in which each node in the linked list keeps an integer value, and also keeps the sum of its value and the values of all subsequent nodes in the list. Thus if we have a list [1, 2, 3, 4], the sum values of each node would be 10, 9, 7, and 4, respectively. Define an *init* method that initializes a new list cell, and define an *insert* method that adds a number such as 5 to the list (e.g. by appending it at the end, putting it in a sorted order, or some other strategy). Specify *insert* to declare its effect on the current node’s sum. Define a *valid* function that constrains the sum field to be the sum of the subsequent nodes in the list. Use Dafny to verify the class implementation against the specification, and to verify a client program that creates a list node, does a couple of insert operations, and asserts that the sum is as expected. Turn in your code.

3. Extending your answer to question 2, write a second client program that traverses partway down the list to an interior node, and does an insert operation on that interior node. Now try to do an operation on the head of the list. Turn in your code. Also answer the following: Does Dafny believe the list is still valid based on your specification, and permit the operation? Why or why not?

4. **[optional, difficult]** Modify your linked list implementation to be a doubly linked list. In the insert operation, use the backwards links to notify earlier nodes in the list when their sum should change. Get the client program from question 3 to verify.

5. Consider the Dafny code for a doubly-linked list cell below. What is the problem with the Valid() function, and why is it a problem for verification? Write an improved Valid() function that does not have this problem, yet still ensures the list is linked up consistently.

```
class DList {
  var val:int;
  var next:DList;
  var prev:DList;
  var footprint:set<DList>;

  function Valid():bool
    reads this,footprint;
  {
    this in footprint
    && (next != null ==> next in footprint && next.footprint == footprint
          && next.prev == this && next.Valid())
    && (prev != null ==> prev in footprint && prev.footprint == footprint
          && prev.next == this && prev.Valid())
  }
}
```

6. Consider the partial Dafny code listed below, in particular the client method. The full implementation of class C is not shown, but the important parts of the specification are.

   a) Explain, using the definition of fresh from the Dafny paper, how Dafny knows that c1 and c2's footprints are disjoint.

   b) Explain, using the relevant part of Figure 9 in the Dafny paper, how Dafny knows that no objects in c1's footprint changed when c2.increase() was called.

   c) Explain, using an axiom defined in the Dafny paper, how Dafny knows that c1.count() is unchanged after the call.

```
class C {
  var footprint:set<C>;

  function count():int
    reads this,footprint;
  { … }
  function valid():bool
    reads this,footprint;
  {…}

  method increase()
    requires valid();
    modifies this, footprint;
    ensures fresh(footprint - old(footprint));
    ensures valid();
    ensures count() == old(count())+1;
  { … }
  // other methods and fields not shown
}
method makeC(x:int) returns (c:C)
  requires x > 0;
  ensures c != null;
  ensures fresh(c);
  ensures fresh(c.footprint);
  ensures c.valid();
  ensures c.count() == x;
{ … }
method client()
{
  var c1 := makeC(1);
  assert c1.count()==1;
  var c2 := makeC(2);
  assert c2.count()==2;
  c2.increase();
  assert c1.count()==1;
}
```