Dafny: Imperative Programming and Verification — Better Together?

Karl Palmskog

02/07/18

Purely Functional vs. Imperative Programming

- mathematical variables vs. program variables (assignables)
- recursion vs. loops
- ease-of-reasoning vs. performance
- monads vs. side effects
- types vs. Hoare logic
- academia vs. industry

Dafny in a Nutshell

Dafny is a programming environment with:

- a purely functional language
- a class-based imperative language
- a specification language for program properties
- integration with program verification facilities
- exporting of verified code to C#

Dafny Overview

- open source, available at `https://github.com/Microsoft/dafny`
- developed mainly at Microsoft Research
- implemented in C#
- relies on the Boogie intermediate verification language and the Z3 theorem prover
- used to develop large-scale systems (e.g., IronFleet, CloudBuild)

Dafny Design Goal

"Provide developers with an immersive experience that feels like programming but encourages thinking about program correctness every step of the way."

In practice: rely heavily on IDE, no conceptual barrier between specification and programming

Recent 4-Page Overview Article of Dafny

"Accessible Software Verification with Dafny", IEEE Computer, Vol 34, Issue 6, Nov 2017
http://ieeexplore.ieee.org/document/8106874/

See also educational videos using Dafny on YouTube, search for "Verification Corner"

Some Alternatives to Dafny

- VCC (C programs, also uses Boogie)
- VeriFast (C programs)
- Frame-C/Jessie (C/Java) using Why
- Coq with FCSL, Iris, or VST

Dafny Interfaces

- Visual Studio (Windows)
- Emacs (everything else)

Emacs setup:

`https://github.com/boogie-org/boogie-friends`

First Example Dafny Program

```
method Mean(x: int, y: int) returns (m : int)


{
  var i := x;
  var j := y;
  while (i ≠ j)



  {
    if (i < j) {
      i := i + 1;
      j := j - 1;
    } else {
      j := j + 1;
      i := i - 1;
    }
  }
  m := i;
}
```

First Example Dafny Program

```
method Mean(x: int, y:int) returns (m : int)
  requires x % 2 == y % 2;
  ensures m == (x + y)/2;
{
  var i := x;
  var j := y;
  while (i != j)




  {
    if (i < j) {
      i := i + 1;
      j := j - 1;
    } else {
      j := j + 1;
      i := i - 1;
    }
  }
  m := i;
}
```

Output

```
Dafny.exe /compile\:0 /nologo mean.dfy
mean.dfy(4,0): Error BP5003: A postcondition might not hold
Execution trace:
    (0,0): anon0
    mean.dfy(7,3): anon13_LoopHead
    (0,0): anon13_LoopBody
    mean.dfy(7,3): anon14_Else
    (0,0): anon16_Then
mean.dfy(7,2): Error: cannot prove termination
Execution trace:
    (0,0): anon0
    mean.dfy(7,3): anon13_LoopHead
    (0,0): anon13_LoopBody
    mean.dfy(7,3): anon14_Else
    mean.dfy(7,3): anon16_Else
    (0,0): anon17_Else
    (0,0): anon11
```

First Example Dafny Program

```
method Mean(x: int, y: int) returns (m : int)
  requires x % 2 == y % 2;
  ensures m == (x + y)/2;
{
  var i := x;
  var j := y;
  while (i != j)




  {
    if (i < j) {
      i := i + 1;
      j := j - 1;
    } else {
      j := j + 1;
      i := i - 1;
    }
  }
  m := i;
}
```

First Example Dafny Program

```
method Mean(x: int, y: int) returns (m : int)
  requires x % 2 == y % 2;
  ensures m == (x + y)/2;
{
  var i := x;
  var j := y;
  while (i != j)
    decreases if i < j then j - i else i - j;


  {
    if (i < j) {
      i := i + 1;
      j := j - 1;
    } else {
      j := j + 1;
      i := i - 1;
    }
  }
  m := i;
}
```

First Example Dafny Program

```
method Mean(x: int, y: int) returns (m : int)
  requires x % 2 == y % 2;
  ensures m == (x + y)/2;
{
  var i := x;
  var j := y;
  while (i != j)
    decreases if i < j then j - i else i - j;
    invariant i < j ==> (x + y)/2 - i == j - (x + y)/2;
    invariant i >= j ==> (x + y)/2 - j == i - (x + y)/2;
  {
    if (i < j) {
      i := i + 1;
      j := j - 1;
    } else {
      j := j + 1;
      i := i - 1;
    }
  }
  m := i;
}
```

Output

```
Dafny.exe /compile\:0 /nologo mean.dfy

Dafny program verifier finished with 2 verified, 0 errors
```

Dafny Datatypes: Sequences

`|s|`	sequence length
`s[i]`	sequence selection
`s[i := e]`	sequence update
`e in s`	sequence membership
`e !in s`	sequence non-membership
`s[lo..hi]`	subsequence
`s[lo..]`	drop
`s[..hi]`	take
`multiset(s)`	sequence conversion to a multiset
`s1 + s2`	concatenation

Functional Specifications

```
function reverse<A>(s: seq<A>) : seq<A>
{
  if |s| = 0 then
    []
  else
    reverse(s[1..]) + [s[0]]
}

predicate nodup<T>(s: seq<T>)
{
  ∀ i, j • 0 ≤ i < |s| ∧ 0 ≤ j < |s| ∧ i ≠ j ⟹ s[i] ≠ s[j]
}
```

Refinement to Imperative Code

```
method Reverse<T>(a: array<T>)
  modifies a;
  ensures a[..] == reverse(old(a[..]));
{




}
```

A Helper Method

```
method Swap<T>(a: array<T>, i: int, j: int)
  requires 0 ≤ i < a.Length;
  requires 0 ≤ j < a.Length;
  modifies a;
  ensures a[i] = old(a[j]);
  ensures a[j] = old(a[i]);
  ensures ∀ k • 0 ≤ k < a.Length ∧ k ≠ i ∧ k ≠ j ⟹
    a[k] = old(a[k]);
  ensures multiset(a[..]) = old(multiset(a[..]));
{
  var tmp := a[i];
  a[i] := a[j];
  a[j] := tmp;
}
```

Using Helper Method

```
method Reverse<T>(a: array<T>)
  modifies a;
  ensures a[..] = reverse(old(a[..]));
{
  var i := 0;
  while (                    )







   {
     Swap(a,  ,                  );
     i := i + 1;
   }




}
```

Executable Program

```
method Reverse<T>(a: array<T>)
  modifies a;
  ensures a[..] = reverse(old(a[..]));
{
  var i := 0;
  while (i ≤ a.Length - 1 - i)







    {
      Swap(a, i, a.Length - 1 - i);
      i := i + 1;
    }



}
```

With Invariants

```
method Reverse<T>(a: array<T>)
  modifies a;
  ensures a[..] = reverse(old(a[..]));
{
  var i := 0;
  while (i ≤ a.Length - 1 - i)
    invariant 0 ≤ i ≤ a.Length - i + 1;
    invariant ∀ k • i ≤ k ≤ a.Length - 1 - i ⟹
      a[k] = old(a[k]);
    invariant ∀ k • 0 ≤ k < i ⟹
      a[a.Length - 1 - k] = old(a[k]) ∧
      a[k] = old(a[a.Length - 1 - k]);
  {
    Swap(a, i, a.Length - 1 - i);
    i := i + 1;
  }
  assert ∀ i • 0 ≤ i < a.Length ⟹
    a[i] = old(a[a.Length - 1 - i]);

}
```

Lemmas

```
function reverse<A>(s: seq<A>) : seq<A>
{
  if |s| == 0 then
    []
  else
    reverse(s[1..]) + [s[0]]
}

lemma reverse_eq<T>(s1: seq<T>, s2: seq<T>)
  requires |s1| == |s2|;
  requires ∀ i • 0 ≤ i < |s1| ⟹ s1[i] == s2[|s2| - 1 - i];
  ensures reverse(s1) == s2;
{



}
```

Lemmas

```
function reverse<A>(s: seq<A>) : seq<A>
{
  if |s| = 0 then
    []
  else
    reverse(s[1..]) + [s[0]]
}

lemma reverse_eq<T>(s1: seq<T>, s2: seq<T>)
  requires |s1| = |s2|;
  requires ∀ i • 0 ≤ i < |s1| ⟹ s1[i] = s2[|s2| - 1 - i];
  ensures reverse(s1) = s2;
{
  if |s1| = 0 { } else {
    reverse_eq(s1[1..], s2[..|s2|-1]);
  }
}
```

Final Version

```
method Reverse<T>(a: array<T>)
  modifies a;
  ensures a[..] = reverse(old(a[..]));
{
  var i := 0;
  while (i ≤ a.Length - 1 - i)
    invariant 0 ≤ i ≤ a.Length - i + 1;
    invariant ∀ k • i ≤ k ≤ a.Length - 1 - i ⟹
      a[k] = old(a[k]);
    invariant ∀ k • 0 ≤ k < i ⟹
      a[a.Length - 1 - k] = old(a[k]) ∧
      a[k] = old(a[a.Length - 1 - k]);
  {
    Swap(a, i, a.Length - 1 - i);
    i := i + 1;
  }
  assert ∀ i • 0 ≤ i < a.Length ⟹
    a[i] = old(a[a.Length - 1 - i]);
  reverse_eq(old(a[..]), a[..]);
}
```

Classes

```
class Node<T> {

  var elem : T;
  var next : Node?<T>;

  constructor Singleton(e: T)
  {
    elem := e;
    next := null;
  }

  constructor Insert(e: T, n: Node<T>)
  {
    elem := e;
    next := n;
  }

}
```

Classes That Represent

```
class Node<T> {

  ghost var list : seq<T>;
  var elem : T;
  var next : Node?<T>;

  predicate Valid()
    reads this, next;
  {
    (next = null ⟹ list = [elem]) ∧
    (next ≠ null ⟹ list = [elem] + next.list)
  }
  constructor Singleton(e: T)
    ensures Valid();
  {
    elem := e;
    next := null;
    list := [e];
  }
}
```

Exporting Code to C#

```
$ Dafny.exe /compile\:1 /spillTargetCode\:1 /nologo LinkedList.dfy

Dafny program verifier finished with 23 verified, 0 errors
Compiled program written to LinkedList.cs
Compiled assembly into LinkedList.dll
```

Running Verified Code

```
public class UseLinkedList
{
    public static void Main()
    {
        Console.WriteLine("creating␣singleton␣list");
        var l1 = new Node<int>();
        l1.Singleton(3);
        Console.WriteLine("inserting␣into␣singleton␣list");
        var l2 = new Node<int>();
        l2.Insert(4, l1);
        Console.WriteLine(l2.next.elem); // outputs "3"
    }
}
```

Applications of Dafny

- IronFleet: distributed system implementations using refinement
- CloudBuild: verified build system (similar to `make`)