Using automatic program verifier Dafny

Anton Bannykh

What is Dafny?

Language and program verifier

- Static verification
- Imperative
- Object-based
- Automatic
- Based on Boogie
- SMT solver at the backend

Microsoft Research product

Part of Boogie distribution

Good for beginners

How does it work?

Write a program in Dafny

- Dafny translates it to Boogie
  - Boogie tries to prove it
    - SMT solver (Z3)
  - Boogie gives some responce
- Dafny processes Boogie output

Outcome

- Success → C# code
- Fail → reasonable feedback

How to use

Write a program with specification

Prove

- Possible design-time feedback
  - MS Visual Studio integration

Debug

- Asserts
- Print Boogie code

Execute

- Automatic generation of C# code

If you know Boogie...

Boogie

```
function fib(n: int) returns (int)
{
    if n < 2 then n else fib(n - 1) + fib(n - 2)
}

procedure Fibonacci(n: int) returns (m: int)
    requires n >= 0;
    ensures m == fib(n);
{
    var p, next, tmp : int;
    p, m, next := 0, 0, 1;
    while (p < n)
        invariant p <= n;
        invariant m == fib(p);
        invariant next == fib(p + 1);
    {
        tmp := m + next;
        m := next;
        next := tmp;
        p := p + 1;
    }
}
```

Dafny

```
function fib(n: int) : int
{
    if n < 2 then n else fib(n - 1) + fib(n - 2)
}

method Fibonacci(n: int) returns (m: int)
    requires n >= 0;
    ensures m == fib(n);
{
    var p, next, tmp : int;
    p, m, next := 0, 0, 1;
    while (p < n)
        invariant p <= n;
        invariant m == fib(p);
        invariant next == fib(p + 1);
    {
        tmp := m + next;
        m := next;
        next := tmp;
        p := p + 1;
    }
}
```

Difference with Boogie

High-level

Syntax sugar

Cares much about the heap

Generates executable code

Program in Dafny

Classes

- Generics
- No inheritance, interfaces, etc.

Methods

- Contain the code
- Translated to C# code

Functions

- Differ from methods
- Special notation
- Ghost by default

Specification

Contracts

- Pre- and postconditions
- No class invariants

Assertions

- Debugging
- Hints for proving backend

Ghost variables and methods

- Specification
- Lemmas and theorems

Specification (cont.)

Functions

- Specification
- Pure
- Ghost by default

Termination metrics

- No infinite loops
- Integers, tuples, sets…

Dynamic frames

- At the object grain
- Set of objects method can modify
- Set of objects function can read

Types

Primitive

- Booleans
- Integers
  - mathematical

Classes

- Default class
- Arrays
- User-defined
- No subtyping
  - object

Datatypes

- Recursively defined datastructures
- No subtyping
  - datatype

Sequences

- Functional specification

Sets

- Dynamic frames

Classes

Any class is subtype of object

Arrays

- array<T>
- array2<T>
- …

User-defined

- class C {…}
- class C<T> {…}

Top-level methods are in the _default class

Methods

```
method M(a: A, b: B, c: C) returns (x: X, y: Y, z: Y)
   requires Precondition;
   modifies Frame;
   ensures Postcondition;
   decreases Rank;
{
   Body
}
```

Modular verification: method body is unknown outside its definition.

Example

```
method MultyRet(x: int, y: int) returns (more: int, less: int)
    requires 0 < y;
    ensures less < x < more;
{
    more := x + y;
    less := x - y;
}
```

Functions

```
function F(a: A, b: B, c: C): T
    requires Pre;
    reads Frame;
    ensures Post;
    decreases Rank;
{
    Body
}
```

Non-ghost function: method function

Functions could be used in predicates

Function body is known by Dafny

Example

```
function fib(n: int): int
    requires 0 <= n;
    decreases n;
{
    if n < 2 then n else fib(n-1) + fib(n - 2)
}
```

Note absence of semicolon

Example

```
method Fibonacci(n: int) returns (m: int)
    requires n >= 0;
    ensures m == fib(n);
{
    var i, next := 0, 1;
    m := 0;
    while (i < n)
        invariant i <= n;
        invariant m == fib(i);
        invariant next == fib(i + 1);
    {
        var tmp := m + next;
        m, next, i := next, tmp, i + 1;
    }
}
```

Frames

Definition

- Frame denotes a set of objects whose fields may be updated by the method

```
class MyClass {
    ghost var Repr: set<object>;
    method SomeMethod()
            modifies Repr;
    {...}
    function SomeFunction(): Something
            reads Repr;
    {...}
}
```

Sequences

Definition

- `var s: seq<int>`
- Length: |s|
- Element: s[0]
- Subsequence: s[a..b]
- Tail: s[1..]
- Concatenation s1+s2

Properties

- Immutable

Sets

Definition

- var s: set<int>
- e in s
- e !in s
- s1 < s2
- s1 <= s2
- s1 !! s2

Properties

- Finite
- Could be used as termination metric
  - decreases s;

Datatypes

```
datatype Tree<T> = Empty | Node(Tree<T>, T, Tree<T>);
```

Useful for datastructure definitions

No need to care about the heap

Easy termination proof

Special notations:

```
match (Expr) {
   case Empty => ...
   case Node(l, d, r) => ...
}
```

EXAMPLES

Maximum over array

```
method Max(a: array<int>) returns (m: int)
    requires a != null;
    requires a.Length > 0;
    ensures forall k :: 0 <= k < a.Length ==> a[k] <= m;
    ensures exists k :: 0 <= k < a.Length ==> a[k] == m;
{
    m := a[0];
    var p := 1;
    while (p < a.Length)
        invariant p <= a.Length;
        invariant forall k :: 0 <= k < p ==> a[k] <= m;
        invariant exists k :: 0 <= k < p ==> a[k] == m;
    {
        if (a[p] > m) {
            m := a[p];
        }
        p := p + 1;
    }
}
```

Maximum over array

```
{
    m := a[0];
    ghost var mi := 0;                              //!
    var p := 1;
    while (p < a.Length)
        invariant p <= a.Length;
        invariant 0 <= mi < a.Length;   //!
        invariant m == a[mi];                   //!
        invariant forall k :: 0 <= k < p ==> a[k] <= m;
        invariant exists k :: 0 <= k < p ==> a[k] == m;
    {
        if (a[p] > m) {
                m := a[p];
                mi := p;                        //!
        }
        p := p + 1;
    }
}
```

Two-way maximum

```
method Max(a: array<int>) returns (mi: int)
    requires a != null;
    requires a.Length > 0;
    ensures 0 <= mi < a.Length;
    ensures forall k :: 0 <= k < a.Length ==> a[k] <= a[mi];
{
    var i, j := 0, a.Length - 1;
    while (i < j)
        invariant 0 <= i <= j < a.Length;
        invariant (forall k :: 0 <= k <= i ==> a[k] <= a[j] || a[k] <= a[i]);
        invariant (forall k :: j <= k < a.Length ==> a[k] <= a[j] || a[k] <= a[i]);
    {
        if (a[i] > a[j]) {
                j := j - 1;
        } else {
                i := i + 1;
        }
    }
    mi := i;
}
```

Infinite loop

```
method hail(n: nat)
{
    var i := n;
    while (1 < i)
        decreases *;
    {
        i := if i % 2 == 0 then i / 2 else 3 * i + 1;
    }
}
```

Ackerman

```
function Ack(m: nat, n: nat): nat
  decreases m, n;
{
  if m == 0 then n + 1
  else if n == 0 then Ack(m - 1, 1)
  else Ack(m - 1, Ack(m, n - 1))
}
```

List reversal

```
class Data {}

class Node {
    ghost var heap: set<Node>;
    var value: Data;
    var right: Node;

    function isValid(): bool
        reads this, heap;
        decreases heap;
    {
        if right == null then
                heap == {this}
        else
                this in heap &&
                right in heap &&
                right.heap == heap - {this} &&
                right.isValid()
    }
}
```

List reversal

```
function getHeap(n: Node): set<Node>
    reads n;
{
    if n == null then {} else n.heap
}

function integralRight(n: Node): seq<Node>
    reads n, n.heap;
    requires n != null ==> n.isValid();
    decreases getHeap(n);
{
    if n == null then [] else [n] + integralRight(n.right)
}

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

List reversal

```
method set_right(n: Node, right: Node)
    requires n != null;
    requires n !in getHeap(right);
    requires right != null ==> right.isValid();
    modifies n;
    ensures n.isValid();
    ensures integralRight(n) == [n] + old(integralRight(right));
{
    n.right := right;
    n.heap := {n} + getHeap(right);
}
```

List reversal

```
method reverse(first: Node) returns (result: Node)
    requires first != null ==> first.isValid();
    modifies first, first.heap;
    ensures result != null ==> result.isValid();
    ensures minus(integralRight(result)) == old(integralRight(first));
{
    if (first == null) {
        result := null;
        return;
    }
    var previous: Node, next: Node := null, first;
    while (next != null)
        decreases getHeap(next);
        invariant next != null ==> next.isValid() && previous != null ==> previous.isValid();
        invariant getHeap(next) <= old(first.heap) && getHeap(previous) !! getHeap(next);

        invariant minus(integralRight(previous)) + integralRight(next) ==
                                                        old(integralRight(first));
    {
        var temp := previous;
        previous, next := next, next.right;
        set_right(previous, temp);
    }
    result := previous;
}
```

Links

Dafny

- http://research.microsoft.com/dafny
- http://rise4fun.com/Dafny/tutorial/Guide
- http://research.microsoft.com/dafny/reference.aspx

Rise4Fun

- http://rise4fun.com/Dafny

Verification corner

- http://research.microsoft.com/verificationcorner