COMPUTER SCIENCE & ENGINEERING

EECS498-008
Formal Verification of Systems Software

Material and slides created by

Jon Howell and Manos Kapritsos

Opacity

```
function eval_linear(m: int, b: int, x: int) :
int
{
    m * x + b
}
```

```
lemma zero_slope(m: int, b: int, x1: int, x2:int)
{
  if (m == 0) {
    assert eval_linear(m, b, x1) == eval_linear(m, b, x2);
  }
}
```

- This lemma verifies because it can see inside the definition of `function eval_linear()`
- …but lemma bodies are opaque! The result of this verification can’t be used anywhere else.

Opacity

```
lemma zero_slope(m: int, b: int, x1: int, x2:int)
  ensures m == 0 ==>
    eval_linear(m, b, x1) == eval_linear(m, b, x2)
{
}


lemma zero_slope(m: int, b: int, x1: int, x2:int)
  requires m == 0
  ensures eval_linear(m, b, x1) == eval_linear(m, b,
x2)
{
}
```

Returning a value

```
function Add(x: nat, y:nat) : (result:nat)
  ensures result >= 0  // identical to “ensures Add(x,y)>=0”
{
  x + y
}


lemma AddLemma(x: nat, y:nat) returns (result:nat)
  ensures result == Add(x,y)
{
   result := x+y;
}
```

Boolean operators

```
!
&&
||
==
==>
<==>
forall
exists
```

- Shorter operators have higher precedence

```
P(x) && Q(x) ==> R(S)
```

- Bulleted conjunctions / disjunctions

```
&& ( P(x))
&& ( Q(y))
&& ( R(x)) ==>( S(y))
&& ( T(x, y))
```

- Parentheses are a good idea around
  forall, **exists**, **==>**

Quantifier syntax: forall

The type of **a** is typically inferred

```
forall a :: P(a)

forall a :: Q(a) ==> R(a)

forall a | Q(a) :: R(a)
```

expression forms

```
Example: assert forall i | 0 < i < 3 :: i*i < 9;
```

```
forall a | Q(a)
   ensures R(a)
{
}
```

statement form

Quantifier syntax: exists

`forall's` evil twin

`exists a :: P(a)`

E.g. exists n:nat :: 2*n == 4

Dafny cannot prove exists without a witness

```
predicate Human(a: Thing) // Empty body ==> axiom
predicate Mortal(a: Thing)

lemma HumansAreMortal()
  ensures forall a | Human(a) :: Mortal(a)  // axiom

lemma MortalPhilosopher(socrates: Thing)
  requires Human(socrates)
  ensures Mortal(socrates)
{
  assert Human(socrates);
  HumansAreMortal();
  assert Mortal(socrates);
}
```

if-then-else expressions

```
if a < b then P(a) else P(b)

     <==>

( a < b && P(a) ) || ( !(a < b) && P(b) )
```

If-then-else expressions work with other types:

```
if a < b then a + 1 else b - 3
```

Sets

- a: set<int> — set is a templated type
- {1, 3, 5}   {} — set literals
- 7 in a — element membership
- a <= b — subset
- a + b — union
- a - b — difference
- a * b — intersection
- a == b — equality *(works with all mathematical objects)*
- |a| — set cardinality
- set x: nat | x < 100 && x % 2 == 0 — set comprehension

Sequences

```
a: seq<int>, b: seq<int>        seq is a templated type
[1, 3, 5]    []                 sequence literal
7 in a                          element membership
a + b                           concatenation
a == b                          equality (works with all mathematical objects)
|a|                             sequence length
a[2..5]      a[3..]             sequence slice
seq(5, i => i * 2)              sequence comprehension
seq(5, i requires 0<=i
         => sqrt(i))
```

Maps

```
a: map<int, set<int>>                map is a templated type
map[2:={2}, 6:={2,3}]                map literal
7 in a      7 in a.Keys              key membership
a == b                               equality (works with all mathematical objects)
a[5 := {5}]                          map update (not a mutation)
map k | k in Evens()                 map comprehension
    :: k/2
```

The var expression

```
lemma foo()
{
  var set1 := { 1, 3, 5, 3 };
  var seq1 := [ 1, 3, 5, 3 ];

  assert forall i | i in set1 :: i in seq1;
  assert forall i | i in seq1 :: i in set1;
  assert |set1| < |seq1|;
}
```

Algebraic datatypes ("struct" and "union")

```
datatype HAlign = Left | Center | Right
```

new name we're defining — disjoint constructors

```
datatype VAlign = Top | Middle | Bottom

datatype TextAlign = TextAlign(hAlign:HAlign,
vAlign:VAlign)
```

multiplicative constructor

```
datatype Order =   Pizza(toppings:set<Topping>)
                 | Shake(flavor:Fruit, whip: bool)
```

Checking for types

```
predicate IsCentered(va: VAlign) {
  !va.Top? && !va.Bottom?
}

function DistanceFromTop(va: VAlign) : int {
  match va
    case Top => 0
    case Middle => 1
    case Bottom => 2
}
```

Hoare logic composition

```
lemma DoggiesAreQuadrupeds(pet:
Pet)
  requires IsDog(pet)
  ensures |Legs(pet)| == 4 { … }
```

```
lemma StaticStability(pet: Pet)
  requires |Legs(pet)| >= 3
  ensures IsStaticallyStable(pet)
{ … }
```

```
lemma DoggiesAreStaticallyStable(pet: Pet)
  requires IsDog(pet)
  ensures IsStaticallyStable(pet)
{
  DoggiesAreQuadrupeds(pet);
  StaticStability(pet);
}
```

Detour to Imperativeland

```
predicate IsMaxIndex(a:seq<int>, x:int) {
  && 0 <= x < |a|
  && (forall i | 0 <= i < |a| :: a[i] <= a[x])
}
```

Note that the order of conjuncts matters!

And the same is true for ensures/requires: their order matters

Imperativeland

```
method findMaxIndex(a:seq<int>) returns (x:int)
  requires |a| > 0
  ensures IsMaxIndex(a, x)
{
  var i := 1;
  var ret := 0;
  while(i < |a|)
    invariant 0 <= i <= |a|
    invariant IsMaxIndex(a[..i], ret)
  {
    if(a[i] > a[ret]) {
      ret := i;
    }
    i := i + 1;
  }
  return ret;
}
```

```
predicate IsMaxIndex(a:seq<int>, x:int) {
  && 0 <= x < |a|
  && (forall i | 0 <= i < |a| :: a[i] <=
a[x])
}
```

Recursion: exporting ensures

```
function Evens(count:int) : (outseq:seq<int>)
  ensures forall idx :: 0<=idx<|outseq| ==> outseq[idx] == 2 * idx
{
  if count==0 then [] else Evens(count) + [2 * (count-1)]
}
```

Chapter 1 exercises

- …will be released tomorrow
  - Chapter 2 will follow soon (once we have covered specification)
  - Together, they constitute Problem Set 1, due September 23, 23:59pm

- Problem sets are to be done individually
  - No collaboration allowed, except to discuss the problem definition

- You should be already added to autograder.io’s roster
  - Let me know if that’s not the case

The RULES

- You may not use /* */ comments
- You must leave the existing /* */ comments in place
- You may only change text between /*{*/ and /*}*/
- You are not allowed to add axioms

Example: exercise01.dfy

```
//#title Lemmas and assertions

lemma IntegerOrdering()
{
  // An assertion is a **static** check of a boolean expression -- a mathematical
truth.
  // This boolean expression is about (mathematical) literal integers.
  // Run dafny on this file. See where it fails. Fix it.
  assert /*{*/ 5 < 3 /*}*/;
}
```