Testing, Debugging, and Verification

Formal Specification, Part III

Atze van der Ploeg

December 1, 2015

Last Lecture

- Introduced Dafny: An object oriented language with formal specification
- Pre- and postconditions: `requires`/`ensures`
- `modifies` clauses: What fields may method change
- `assert` statements: Dafny tries to prove property.

Remember

Outside method body Dafny only "remembers" annotations (pre- and postconditions).

Methods, Functions and Predicates

- Methods cannot be used in annotations (may change memory).
- `functions` and `predicates` can
  - Cannot write to memory
  - Single statement
  - `reads` keyword states what location functions looks up.

Dafny Functions

- Mathematical functions.
- Cannot write to memory (unlike methods). Safe to use in spec.
- Can only be used in annotations.
- Single unnamed return value, body is single statement (no semicolon).

A function

```
function abs(x : int) : int
{ if x < 0 then -x else x }
```

- Now, can write e.g. `assert abs(3) == 3;`.
- Or, `ensures r == abs(x)`.

Dafny Functions

A function method

```
function method abs(x : int) : int {
  if x < 0 then -x else x
}
```

- Functions are only used for verification.
- Not present in compiled code.
- Functions which does exactly same as a method can be declared `function methods`.
- However, functions need not be efficient.
- Write simple (recursive) function to specify efficient, more complex method.

Recall: Predicates

Functions returning a boolean are called predicates.

A predicate

```
predicate ready()
  reads this; {
  insertedCard == null && wrongPINCounter == 0 &&
  auth == false; }
```

Recall: Predicates

Functions returning a boolean are called predicates.

A predicate

```
predicate ready()
  reads this; {
  insertedCard == null && wrongPINCounter == 0 &&
  auth == false; }
```

Predicates are useful for "naming" common properties used in many annotations:

Example

```
method spitCardOut() returns (card : BankCard)
   modifies this;
   requires insertedCard != null;
   ensures card == old(insertedCard);
   ensures ready();
```

A few words on Framing

Recall from ATM example

```
predicate ready()
  reads this;
  {insertedCard == null && wrongPINCounter == 0 &&
    auth == false;}
```

Reading Frame: memory region allowed to be read by function or predicate (here, all fields of `this` object)

Why?

A few words on Framing

Recall from ATM example

```
predicate ready()
  reads this;
  {insertedCard == null && wrongPINCounter == 0 &&
    auth == false;}
```

Reading Frame: memory region allowed to be read by function or predicate (here, all fields of `this` object)

Why?
Efficiency, if a function does not read a part of memory, we can be sure result is same as before a write.

Framing, Example

```
var atm1 := new ATM;
var atm2 := new ATM;
. . .
assert atm1.ready();
some update to atm2
```

- Here, we know immediately that `atm1` is still ready, as its portion of memory hasn't been changed.
- For simple example, easy to prove anyway, but might be infeasible for more complex data-structures.

Framing: Modifies clauses

Recall

```
method insertCard(c : BankCard)
   modifies `insertedCard;
```

- Methods may read any part of memory
- Must declare what they change
- `reads` and `modifies` allows Dafny to work on one method at the time
- Crucial for efficiency and feasibility of automated proofs.
- The logic about what memory can influence which results is called Seperation Logic.

More built in Data-structures: Sets

- Dafny also support Sets.
- **Set:** Collection of elements, no duplication.
- Immutable, allowed in annotations.
  - Cannot be modified once created.
  - "Modification" by creating a new Set.
  - c.f. strings in Java.

Examples: See Dafny online tutorial (link from lecture page).

Examples: Sets

Basics

```
var s1 := {}; // the empty set
var s2 := {1, 2, 3}; // set contains exactly 1, 2, and 3
assert s2 == {1,1,2,3,3,3,3}; // true, no duplicates.
```

Examples: Sets

Basics

```
var s1 := {}; // the empty set
var s2 := {1, 2, 3}; // set contains exactly 1, 2, and 3
assert s2 == {1,1,2,3,3,3,3}; // true, no duplicates.
```

Union, intersection and set difference

```
var s3, s4 := {1,2}, {1,4};
assert s2 + s4 == {1,2,3,4}; // set union
assert s2 * s3 == {1,2} && s2 * s4 == {1}; // set
 intersection
assert s2 - s3 == {3}; // set difference
```

Examples: Sets

Subset operators

```
assert {1} <= {1, 2} && {1, 2} <= {1, 2}; // subset
assert {} < {1, 2} && !({1} < {1}); // strict, or proper,
 subset
assert {1, 2} == {1, 2} && {1, 3} != {1, 2}; // equality
 and non-equality
```

Examples: Sets

Subset operators

```
assert {1} <= {1, 2} && {1, 2} <= {1, 2}; // subset
assert {} < {1, 2} && !({1} < {1}); // strict, or proper,
 subset
assert {1, 2} == {1, 2} && {1, 3} != {1, 2}; // equality
 and non-equality
```

Set Membership

```
assert 5 in {1,3,4,5};
assert 1 in {1,3,4,5};
assert 2 !in {1,3,4,5};
assert forall x :: x !in {};
```

Recap: Using Quantified Dafny expressions

How to express:

- An array `arr` only holds values ≤ 2

Recap: Using Quantified Dafny expressions

How to express:

- An array `arr` only holds values ≤ 2

```
forall i ::  0 <= i <arr.Length ==> arr[i] <= 2
```

Recap: Using Quantified Dafny expressions

How to express:

- The variable `m` holds the maximum entry of array `arr`

Recap: Using Quantified Dafny expressions

How to express:

- The variable `m` holds the maximum entry of array `arr`

```
forall i :: 0 <= i < arr.Length ==> m >= arr[i]
```

Is this enough?

Recap: Using Quantified Dafny expressions

How to express:

- The variable `m` holds the maximum entry of array `arr`

```
forall i :: 0 <= i < arr.Length ==> m >= arr[i]
```

Is this enough?

```
arr.Length > 0 ==>
exists i :: 0 <= i < arr.Length && m == arr[i]
```

Example: Specifying LimitedIntegerSet

```
class LimitedIntegerSet {

  var limit : int;
  var arr : array<int>;
  var size : int;

 method Init(lim : int)
  {
      limit := lim;
      arr := new int[lim];
      size := 0;
   }
  method Contains(elem : int) returns (res : bool){/*...*/}
  method Find(elem : int) returns (index : int) {/*...*/}
  method Add(elem : int) returns (res : bool) {/*...*/}

}
```

Specifying `Init`: A validity predicate

What are the allowed values for the fields of a `LimitedInSet`?

```
class LimitedIntegerSet {

  var limit : int;
  var arr : array<int>;
  var size : int;

predicate Valid()
  reads this, this.arr;
  {arr != null &&
  0 <= size <= limit &&
  limit == arr.Length}
```

Specifying Init

```
method Init(lim : int)
  modifies this;
  requires lim > 0;
  ensures Valid();
  ensures limit == lim && size == 0;
  ensures fresh(arr);
{. . .}
```

- New objects are indeed valid.
- Parameters set correctly.
- Array is freshly allocated.

Specifying `contains`

```
method contains (elem : int) . . .
```

- Has no effect on the state.
- Returns a boolean.
- Might be useful in specifications.
- Let's make it a function method!

Specifying `contains`

```
method contains (elem : int) . . .
```

- Has no effect on the state.
- Returns a boolean.
- Might be useful in specifications.
- Let's make it a function method!

```
function method contains (elem : int) : bool
reads this, this.arr;
requires this.Valid();
{exists i :: 0 <= i < size && arr[i] == elem}
```

Specifying add

```
method add(elem : int)
modifies this.arr, this`size;
requires this.Valid();
ensures Valid();
ensures (!old(contains(elem)) && old(size) < limit) ==>
             res && contains(elem) && size == old(size)+1 &&
             (forall e :: e!=elem && old(contains(e)) ==>
             contains(e));
ensures (old(contains(elem)) || old(size) >= limit) ==>
             !res && size == old(size) &&
             forall i :: 0 <= i < size ==> arr[i] == old(arr[i
 ]);

{/*...*/}
```

Details of Specification

- How much detail needed in formal specification?
- Depends (to some extent) on what we want to prove about code.
- Recall: Dafny only "remembers" spec of method outside method body.

Specifying **Find**

```
method Find(elem : int) returns (index : int)
 requires Valid();
 ensures 0 <= index < size ==> arr[index] == elem;
 ensures index < 0 ==> forall k :: 0 <= k < size ==>
         arr[k] != elem;
```

- Implemented using linear search (while loop).
- Dafny cannot prove post-condition!
  - How many times do we go through the loop?
  - Will it cover all elements?
- **Solution: Loop invariants**

Introduction to Loop Invariants

- No way of knowing how many times code will loop.
- Need to prove for all paths of program.
  - c.f. proof by induction.

Loop invariant is expression which holds:

- First time entering loop
- At each iteration of loop
- When exiting the loop

Loop Invariant Example I

```
var i := 0;
while (i < n)
   invariant 0 <= i;
   { i := i + 1; }
```

Dafny proves:

- Invariant holds when entering the loop.
- Invariant preserved by the loop.
- If invariant true at beginning of loop, it holds after executing the loop once.
- Proof by induction.

Loop Invariant Example II

Suppose we want to show that when exiting loop, `i == n`:

```
var i := 0;
while (i < n)
   invariant 0 <= i;
   { i := i + 1; }
assert i == n
```

Loop Invariant Example II

Suppose we want to show that when exiting loop, `i == n`:

```
var i := 0;
while (i < n)
   invariant 0 <= i;
   { i := i + 1; }
assert i == n
```

Fails!

All Dafny knows after loop exits is

- that loop guard failed: `!(i < n)` which means (`i >= n`)
- that the invariant holds: `0 <= i`

Loop Invariant Example II

Suppose we want to show that when exiting loop, `i == n`:

```
var i := 0;
while (i < n)
   invariant 0 <= i;
   { i := i + 1; }
assert i == n
```

Fails!

All Dafny knows after loop exits is

- that loop guard failed: `!(i < n)` which means (`i >= n`)
- that the invariant holds: `0 <= i`

Need to revise invariant: `0 <= i < n`

Loop Invariant Example III

Suppose we want to show that when exiting loop, `i == n`:

```
var i := 0;
while (i < n)
   invariant 0 <= i < n;
   { i := i + 1; }
assert i == n
```

Loop Invariant Example III

Suppose we want to show that when exiting loop, `i == n`:

```
var i := 0;
while (i < n)
   invariant 0 <= i < n;
   { i := i + 1; }
assert i == n
```

Fails: Invariant is not preserved

Dafny tries to prove that `0 <= i < n` holds after each iteration

- Holds for every execution except last one.

Loop Invariant Example III

Suppose we want to show that when exiting loop, `i == n`:

```
var i := 0;
while (i < n)
   invariant 0 <= i < n;
   { i := i + 1; }
assert i == n
```

Fails: Invariant is not preserved

Dafny tries to prove that `0 <= i < n` holds after each iteration

- Holds for every execution except last one.
- **Need to revise invariant:** `0 <= i <= n`
- Finding the correct invariant can be challenging.
- Will revisit this topic in Formal Verification part of course.

Loop Invariants for Find method

```
method Find(elem : int) returns (index : int)
 requires Valid();
 ensures index < 0 ==> forall k :: 0<= k<size ==> arr[k]!=elem;
 ensures 0 <= index ==> index < size && arr[index] == elem;
{
  index := 0;
  while (index < size)
   invariant forall i : int :: 0 <= i < index ==> arr[i] !=
 elem
   {
         if(arr[index] == elem) {return index;}
         index := index + 1;
    }
    index := -1;
```

- Dafny needs to know loop covers all elements.
- Everything before current index has been looked at and is not `elem`.

Loop Invariants for `Find` method

```
index := 0;
while (index < size)
 invariant forall k ::  0 <= k < index ==> a[k] != elem
 {
        if(arr[index] == elem) {return;}
        index := index + 1;
  }
  index := -1;
```

- Everything before, but excluding `index` is not `elem`.
- Holds on entry: as `index` is 0, quantification over empty set. Implication trivially true.
- Invariant is preserved: tests value before extending range of non-elem range.
- Dafny complains: `index` may be out of range of array. Need invariant on `index` too.

Loop Invariants for Find method

```
index := 0;
while (index < size)
 invariant forall k :: 0 <= k < index ==> a[k] != elem
 invariant 0 <= index <= size
 {
       if(arr[index] == elem) {return;}
       index := index + 1;
  }
  index := -1;
```

- Holds on entry: as `index` is 0, quantification over empty set. Implication trivially true.
- Invariant is preserved: tests value before extending range of non-elem range.
- Standard bound on `index`: One past end of growing range is a common pattern.
- No array-out-of bound as `k < index`.

Termination

- We know is if we exit the loop, we can assume negation of loop guard and invariants.
- Invariant says nothing about whether loop actually ever exits.
- `decreases` clause:
  - Expression gets smaller at each iteration
  - Is bounded
  - Often (but not always) integer value
- Dafny can often guess this itself.

Example

```
while (0 < i)
  invariant 0 <= i;
  decreases i;
  {
    i := i -1;
  }
```

Termination: Common pattern for `decreases`

Often count up, not down:

Example

```
while (i < n)
  invariant 0 <= i <= n;
  decreases (n - i);
  {
    i := i +1;
  }
```

- Difference between `n` and `i` decrease.
- Bounded from below by zero: `0 <= (n - i)`.
- Very common pattern, Dafny's guess in most situations.

Summary

- Framing: `reads` and `modifies` caluses. Important for efficiency.
- Sets.
- Specifying "valid" objects.
- Using quantifiers in specifications.
- Loops and loop invariant (more in coming lectures).
- Loop termination and `decreases` clauses (more later!).