

CS:5810 Formal Methods in Software Engineering

Reasoning about Programs with Objects in Dafny

Copyright 2020-21, Graeme Smith and Cesare Tinelli.

Produced by Cesare Tinelli at the University of Iowa from notes originally developed by Graeme Smith at the University of Queensland. These notes are copyrighted materials and may not be used in other course settings outside of the University of Iowa in their current form or modified form without the express written permission of one of the copyright holders. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of one of the copyright holders.

Checksum Objects

An *object* is an instance of a *class*, and like arrays, has a *reference type*

```
class ChecksumMachine {  
  var data: string  
  
  constructor ()  
    ensures data == ""  
  
  method Append(d: string)  
    modifies this  
    ensures data == old(data) + d  
  
  function method Checksum(): int  
    reads this  
    ensures Checksum() == Hash(data)  
  
  ...  
}
```

string is shorthand
for seq<char>

Checksum Objects

...

```
function method Hash(s: string): int {  
    SumChars(s) % 137  
}
```

```
function method SumChars(s: string): int {  
    if |s| == 0 then 0 else  
        var last := |s| - 1;  
        SumChars(s[..last]) + s[last] as int  
    }  
}
```



converts char to int

Test client

```
method Main() {  
    var m := new ChecksumMachine();  
    m.Append("green ");  
    m.Append("grass");  
    var c := m.Checksum();  
    print "Checksum is ", c, "\n";  
}
```

*A method is allowed to allocate new arrays and objects and change their state (that is, the elements of the arrays and the fields of the objects) without mentioning these arrays and objects in the **modifies** clause*

Class Invariant

To write efficient implementation, want to keep track of checksum so far:

```
var cs: int
```

We want to use data in specifications, but not in compiled program:

```
ghost var data: string
```

```
predicate Valid()  
  reads this  
{ cs == Hash(data) }
```

A predicate is a
Boolean function

If a function accesses the fields of an object o, its specification must include reads o

Class Invariant

```
class ChecksumMachine {  
  ghost var data: string  
  
  predicate Valid()  
  reads this  
  
  constructor ()  
  ensures Valid() && data == ""  
  
  method Append(d: string)  
  requires Valid()  
  modifies this  
  ensures Valid() && data == old(data) + d  
  
  function method Checksum(): int  
  requires Valid()  
  reads this  
  ensures Checksum() == Hash(data)  
}
```

Implementation

```
constructor ()  
  ensures Valid() && data == ""  
{ cs := 0;  
  data := ""  
}
```

*A constructor is allowed to assign to the fields of the object being constructed, **this**, without mentioning **this** in the **modifies** clause*

```
function method Checksum(): int  
  requires Valid()  
  reads this  
  ensures CheckSum() == Hash(data)  
{ cs }
```

Implementation

```
method Append(d: string)
  requires Valid()
  modifies this
  ensures Valid()
  ensures data == old(data) + d
{
  var i := 0;
  while i != |d|
    invariant 0 <= i <= |d|
    invariant Valid()
    invariant data == old(data) + d[..i]
    {
      cs := (cs + d[i] as int) % 137;
      data := data + [d[i]];
      i := i + 1;
    }
}
```


Coffee maker components

```
class Grinder {  
  var HasBeans: bool
```

```
  predicate Valid()  
    reads this
```

```
  constructor ()  
    ensures Valid()
```

```
  method AddBeans()  
    requires Valid()  
    modifies this  
    ensures Valid() && HasBeans
```

```
  method Grind()  
    requires Valid() && HasBeans  
    modifies this  
    ensures Valid()  
}
```

```
class WaterTank {  
  var Level: nat
```

```
  predicate Valid()  
    reads this
```

```
  constructor ()  
    ensures Valid()
```

```
  method Fill()  
    requires Valid()  
    modifies this  
    ensures Valid() && Level == 10
```

```
  method Use()  
    requires Valid() && Level != 0  
    modifies this  
    ensures Valid() && Level == old(Level) - 1  
}
```

```
class Cup {  
  constructor ()
```

```
}
```

Coffee maker version 0

```
class CoffeeMaker {  
  predicate Valid() reads this  
  
  constructor () ensures Valid()  
  
  predicate method Ready()  
    requires Valid()  
    reads this  
  
  method Restock()  
    requires Valid()  
    modifies this  
    ensures Valid() && Ready()  
  
  method Dispense(double: bool) returns (c: Cup)  
    requires Valid() && Ready()  
    modifies this  
    ensures Valid()  
}
```

Coffee maker version 0

State:

```
var g: Grinder
var w: WaterTank

predicate Valid()
  reads this
{ g.Valid() && w.Valid() } // error: insufficient
                          // reads clause
```

Require:

```
predicate Valid()
  reads this, g, w
```

Similar change also needed for **reads** of Ready() and **modifies** clauses of Restock and Dispense

Representation sets

The expanded modifies and reads clauses violate the principles of information hiding.

Therefore, we abstract the state of an object to a *representation set*.

For this implementation of the coffee maker, the representation set is

$$\{o, o.g, o.w\}$$

but the coffee maker may also be implemented in terms of different objects.

Coffee maker version 1

Add new variable to state:

```
ghost var Repr: set<object>
```

Change modifies clauses of Restock and Dispense to

```
modifies Repr
```

Change read clauses of Valid and Ready to

```
reads Repr
```

Add the following to the body of Valid

```
this in Repr &&  
g in Repr && g.Valid() &&  
w in Repr && w.Valid()
```

Typically specify
lower bound on
objects in Repr

Coffee maker version 1

In Valid:

```
reads Repr // error: insufficient reads clause
```

This is because `this` is not in Repr unless Valid's predicate holds (and Valid may return `true` or `false`).

We require:

```
predicate Valid()  
  reads this, Repr  
{  
  this in Repr &&  
  g in Repr && g.Valid() &&  
  w in Repr && w.Valid()  
}
```

Class implementation

```
constructor ()
  ensures Valid()
{
  g := new Grinder();
  w := new WaterTank();
  Repr := {this, g, w};
}

predicate method Ready()
  requires Valid()
  reads Repr
{
  g.HasBeans && 2 <= w.Level
}
```

Class implementation

```
method Restock()  
  requires Valid()  
  modifies Repr  
  ensures Valid() && Ready()  
{  
  g.AddBeans(); w.Fill();  
}  
  
method Dispense(double: bool) returns (c: Cup)  
  requires Valid() && Ready()  
  modifies Repr  
  ensures Valid()  
{  
  g.Grind();  
  if double { w.Use(); w.Use(); } else { w.Use(); }  
  c := new Cup();  
}
```


Test harness

```
method CoffeeTestHarness() {  
    var cm := new CoffeeMaker();  
    cm.Restock(); // modifies clause violated  
    var c := cm.Dispense(true); // modifies clause violated  
}
```

The test harness has no **modifies** clause and so is only allowed to modify the fields of fresh objects

Our specification of the coffee maker didn't specify that created objects were fresh

Coffee maker version 2

Add to constructor:

```
ensures fresh(Repr)
```

This removes error with Restock, but not Dispense.

Add to Restock and Dispense:

```
ensures Repr == old(Repr)
```

Alternatively, make Repr *immutable* by declaring it as

```
ghost const Repr: set<object>
```

Changing Repr

What if implementation needs to change Repr, e.g., a method of the coffee maker needs to change the grinder?

Third (and preferred) alternative for `ensures` clauses of methods which mutate Repr:

```
ensures fresh(Repr - old(Repr))
```

That is, any new objects added to Repr are *fresh*

Less common situations

```
method ChangeGrinder()  
  requires Valid()  
  modifies Repr  
  ensures Valid() && fresh(Repr - old(Repr))  
{  
  g := new Grinder();  
  Repr := Repr + {g};  
}
```

Old grinder is still in Repr, but is no longer referenced

The run-time system will eventually reclaim the storage for this object

Less common situations

```
method InstallCustomGrinder(grinder: Grinder)
  requires Valid() && grinder.Valid()
  modifies Repr
  ensures Valid()
  ensures fresh(Repr - old(Repr) - {grinder})
{
  g := grinder;
  Repr := Repr + {g};
}
```

Less common situations

```
method InstallCustomGrinder(grinder: Grinder)
  requires Valid() && grinder.Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr) - {grinder})
{
  g := grinder;
  Repr := Repr + {g};
}
```

Since Repr can dynamically change, this approach to specification is referred to as *dynamic frames*

Dafny is a permutation of certain letters in Dynamic frames

Grinder as an aggregate

```
class Grinder {
  var HasBeans: bool
  ghost var Repr: set<object>
  predicate Valid()
    reads this, Repr
  constructor ()
    ensures Valid() && fresh(Repr)
  method AddBeans()
    requires Valid()
    modifies Repr
    ensures Valid() && HasBeans && fresh(Repr - old(Repr))
  method Grind()
    requires Valid() && HasBeans
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr))
}
```

WaterTank as an aggregate

```
class WaterTank {
  var Level: nat
  ghost var Repr: set<object>
  predicate Valid() reads this, Repr
  constructor () ensures Valid() && fresh(Repr)
  method Fill()
    requires Valid()
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr)) && Level == 10
  method Use()
    requires Valid() && Level != 0
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr))
      && Level == old(Level) - 1
}
```


Coffee Maker

Invariant (in Valid):

```
this in Repr &&  
g in Repr && g.Repr <= Repr && g.Valid() &&  
w in Repr && w.Repr <= Repr && w.Valid()
```

Constructor:

```
constructor ()  
  ensures Valid() && fresh(Repr)  
{  
  g := new Grinder();  
  w := new WaterTank();  
  Repr := {this, g, w} + g.Repr + w.Repr;  
} // illegal first-phase use of fields
```

Constructor

First phase set objects fields and define immutable values

- objects are still being constructed
- so, `this.g.Repr` is not allowed for example

Avoid use of uninitialized fields:

```
var gg := new Grinder();  
var ww := new WaterTank();  
g, w := gg, ww;  
Repr := {this, g, w} + gg.Repr + ww.Repr;
```

Update Repr in second phase:

```
g := new Grinder(); w := new WaterTank();  
new;  
Repr := {this, g, w} + g.Repr + w.Repr;
```

Restock

```
method Restock()  
  requires Valid()  
  modifies Repr  
  ensures Valid() && fresh(Repr - old(Repr)) && Ready()  
{  
  
  g.AddBeans();  
  
  w.Fill(); // precondition violation; modifies violation  
} // postcondition violation
```

Restock

```
method Restock()  
  requires Valid()  
  modifies Repr  
  ensures Valid() && fresh(Repr - old(Repr)) && Ready()  
{
```

```
  g.AddBeans();  
  assert w.Valid(); // assertion violation  
  w.Fill(); // modifies violation  
} // postcondition violation
```

Precondition of
`w.Fill()` not violated
if `w.Valid()` holds

Restock

```
method Restock()  
  requires Valid()  
  modifies Repr  
  ensures Valid() && fresh(Repr - old(Repr)) && Ready()  
{  
  assert w.Valid();  
  
  g.AddBeans();  
  assert w.Valid(); // assertion violation  
  w.Fill(); // modifies violation  
} // postcondition violation
```

Call to AddBeans
affects w.Valid()

Restock

```
method Restock()  
  requires Valid()  
  modifies Repr  
  ensures Valid() && fresh(Repr - old(Repr)) && Ready()  
{  
  assert w.Valid();  
  
  g.AddBeans();  
  assert w.Valid(); // assertion violation  
  w.Fill(); // modifies violation  
} // postcondition violation
```

Call to AddBeans
affects w.Valid()

g.AddBeans only modifies g.Repr, and w.Valid only reads w.Repr
This suggests there is an overlap between g.Repr and w.Repr

Restock

```
method Restock()  
  requires Valid()  
  modifies Repr  
  ensures Valid() && fresh(Repr - old(Repr)) && Ready()  
{  
  assert w.Valid();  
  assert g.Repr !! w.Repr; // assertion violation  
  g.AddBeans();  
  assert w.Valid(); // assertion violation  
  w.Fill(); // modifies violation  
} // postcondition violation
```

(A !! B) states that sets A and B are disjoint ($A * B == \{\}$)

Restock

```
method Restock()  
  requires Valid()  
  modifies Repr  
  ensures Valid() && fresh(Repr - old(Repr)) && Ready()  
{  
  assert this !in g.Repr; // assertion violation  
  assert g in g.Repr; // assertion violation  
  assert w !in g.Repr; // assertion violation  
  assert w.Valid();  
  assert g.Repr !! w.Repr; // assertion violation  
  g.AddBeans();  
  assert w.Valid(); // assertion violation  
  w.Fill(); // modifies violation  
} // postcondition violation
```


Coffee Maker invariant

Valid:

```
this in Repr && g in Repr &&  
g.Repr <= Repr &&  
this !in g.Repr && g.Valid() &&  
w in Repr && w.Repr <= Repr &&  
this !in w.Repr && w.Valid() &&  
g.Repr !! w.Repr
```

If body of Valid() is hidden from clients then they can't see **this** in Repr. Hence, update postcondition of *all* validity predicates as follows

```
predicate Valid()  
  reads this, Repr  
  ensures Valid() ==> this in Repr
```

Back to Restock

```
method Restock()  
  requires Valid()  
  modifies Repr  
  ensures Valid() && fresh(Repr - old(Repr)) && Ready()  
{  
  g.AddBeans();  
  w.Fill();  
  
} // postcondition violation
```

Calls to `AddBeans` and `Fill` may expand `g.Repr` and `w.Repr`

Back to Restock

```
method Restock()  
  requires Valid()  
  modifies Repr  
  ensures Valid() && fresh(Repr - old(Repr)) && Ready()  
{  
  g.AddBeans();  
  w.Fill();  
  Repr := Repr + g.Repr + w.Repr;  
} // postcondition violation
```

Back to Restock

```
method Restock()  
  requires Valid()  
  modifies Repr  
  ensures Valid() && fresh(Repr - old(Repr)) && Ready()  
{  
  g.AddBeans();  
  w.Fill();  
  Repr := Repr + g.Repr + w.Repr;  
} // postcondition violation
```

What we did on the relationships between frames holds for `Dispense` too. We just need to add the following to its body:

```
Repr := Repr + g.Repr + w.Repr;
```

Summary

Representation set:

```
ghost var Repr: set<object>
```

Invariant:

```
predicate Valid()  
reads this, Repr  
ensures Valid() ==> this in Repr  
{ this in Repr && ... }  
  
a in Repr && a.Valid()  
  
b in Repr && b.Repr <= Repr &&  
this !in b.Repr && b.Valid()  
  
a0 != a1 &&  
{a0, a1} !! b0.Repr !! b1.Repr
```

a, a0, a1 are objects with
simple frames

b, b0, b1 are objects
with dynamic frames

Summary

Constructor:

```
constructor ()  
  ensures Valid() && fresh(Repr)  
{ ... ; Repr := {this, a, b} + b.Repr; }
```

Functions:

```
function F(x: X): Y  
  requires Valid()  
  reads Repr
```

(Mutating) method:

```
method M(x: X) returns (y: Y)  
  requires Valid()  
  modifies Repr  
  ensures Valid() && fresh(Repr - old(Repr))
```