Verification & Testing
Hoare Logic / Deductive Verification

Benedikt Maderbacher
IAIK

Today

- Hoare Logic and Arrays
- Termination
- Dafny

Recap Example

```
{true}
result = 0;

counter = 0;

while (counter != x) {


    counter = counter + 1;


    result = result + y;

}


{result == x*y}
```

Invariant:

Recap Example

```
{true} assign.
result = 0;
{result == 0} assign.
counter = 0;
{result == counter*y} while
while (counter != x) {
    {(result == counter*y) && counter != x} strengthen
    {result == counter*y} rewrite
    {result == ((counter + 1) - 1)*y} assign.
    counter = counter + 1;
    {result == (counter - 1)*y} rewrite
    {result + y == counter*y} assign.
    result = result + y;
    {result == counter*y} while
}
{result == counter*y) && counter == x} rewrite
{result == x*y) && counter == x} strengthen
{result == x*y}
```

Invariant:

result == counter * y

Axioms for Arrays

- $a\{i \mapsto x\}(i) = x$
- $a\{i \mapsto x\}(j) = a(j)$ if $i \neq j$
- $a = b$ if $\forall$ i: $a(i) = b(i)$

SMTLib:

a(i) === (select a i)

a{i↦x} === (store a i x)

Axiom of Array Assignment

$$\frac{}{\{P[a \rightarrow a\{i \mapsto e\}]\}\ a[i]:=e\ \{P\}}$$

Example:

$\{a\{1 \mapsto y\} = [1,2,3]\}$ a[1] := y $\{a = [1,2,3]\}$

$\{a\{2 \mapsto y\}(2) = 5\}$ a[2] := y $\{a(2) = 5\}$

$\{a = []\}$ a[x] := x + 1 $\{a = []\{x \mapsto x+1\}\}$

Termination

Classic Hoare Logic can only prove partial correctness.

How can we extend it to total correctness?

Total correctness = partial correctness + termination

While Rule - Total Correctness

$$\frac{\{I \wedge c \wedge E{=}n\}\ S\ \{I \wedge E{<}n\} \qquad\qquad I \wedge c \Rightarrow E \geq 0}{\{I\}\ \textbf{while}\ c\ \textbf{do}\ S\ \textbf{od}\ \{I \wedge \neg c\}}$$

where E is an integer-valued expression and n is an auxiliary variable not occurring in I , c, S or E.

We will call the expression E the *variant* of the loop.

Example

{0 <= y}
x := 0

while(x != y) {

  x := x + 1

}

{x == y}

Invariant:
Variant:

Example

```
{0 <= y} assign.
x := 0
{x <= y} while
while(x != y) {
  {x <= y && y-x == n && x != y} strengthen
                        (x <= y && x != y) ==> x+1 <= y
  {x+1 <= y && y-x == n}  strengthen (y-x == n) ==> (y-x < n+1)
  {x+1 <= y && y-x < n+1} rewrite
  {x+1 <= y && y-(x+1) < n} assign.
  x := x + 1
  {x <= y && y-x < n} while
}
{x <= y && x == y} strengthen
{x == y}
```

```
Invariant: x <= y
Variant:   y - x

The variant is bounded:
(x <= y && x!=y) ==> y-x >= 0
(x < y) ==> x <= y
[]
```

Verification Conditions

Additional conditions that need to be satisfied to guarantee termination.

vct(while c do {I}[E] S od, P) =

$\{(I \wedge c) \Rightarrow E \geq 0, (I \wedge c \wedge E=n) \Rightarrow pre(S, I \wedge E<n)\}$

$\cup$ vct(S, P)

Lexicographic Variants

Instead of integers one can use tuples of integers as variants.

$0 \equiv (0,0)$

$(a,b) = (c,d) \equiv a=c \land b=d$

$(a,b) < (c,d) \equiv a<c \lor (a = c \land b<d)$

$(a,b) \geq (c,d) \equiv a>c \lor (a = c \land b\geq d)$

Dafny

Verification-aware programming language

Pre-, post-condition specifications

Uses the intermediate verification language Boogie and the Z3 SMT solver as a backend.

Leftpad in Dafny

```
function max(a: int, b: int): int
{
      if a > b then a else b
}

method LeftPad(c: char, n: int, s: seq<char>) returns (v: seq<char>)
requires n > 0
ensures |v| == max(n, |s|)
ensures forall i :: 0 <= i < n - |s| ==> v[i] == c
ensures forall i :: 0 <= i < |s| ==> v[max(n - |s|, 0)+i] == s[i]
{
      var pad, i := max(n - |s|, 0), 0;
      v := s;
      while i < pad
      decreases pad - i
      invariant 0 <= i <= pad
      invariant |v| == |s| + i
      invariant forall j :: 0 <= j < i ==> v[j] == c
      invariant forall j :: 0 <= j < |s| ==> v[i+j] == s[j]
      {
            v := [c] + v;
            i := i + 1;
      }
}
```

Let's prove leftpad

Repository of multiple proofs of the same problem in different languages and frameworks. Curated by Hillel Wayne.

https://github.com/hwayne/lets-prove-leftpad

https://www.hillelwayne.com/post/lpl/

Modular Verifcation

Dafny checks each functions specification independently.

A caller only knows the pre/post conditions and nothing about the internal implementation.

Proofs that are not done automatically by the SMT solver can be written as programs.

Dafny Example 2

```
lemma SkippingLemma(a: array<int>, j: int)
requires forall i :: 0 <= i < a.Length ==> 0 <= a[i]
requires forall i :: 0 < i < a.Length ==> a[i-1]-1 <=
     a[i]
requires 0 <= j < a.Length
ensures forall k :: j <= k < j + a[j] && k < a.Length
     ==> a[k] != 0
{
    var i := j;
    while i < j + a[j] && i < a.Length
    invariant i < a.Length ==> a[j] - (i-j) <= a[i]
    invariant forall k :: j <= k < i && k < a.Length
     ==> a[k] != 0
    {
        i := i + 1;
    }
}
```

```
method FindZero(a: array<int>) returns (index: int)
requires forall i :: 0 <= i < a.Length ==> 0 <= a[i]
requires forall i :: 0 < i < a.Length ==>
     a[i-1]-1 <= a[i]
ensures index < 0 ==> forall i :: 0 <= i < a.Length ==>
     a[i] != 0
ensures 0 <= index ==> index < a.Length &&
        a[index] == 0
{
    index := 0;
    while index < a.Length
    invariant 0 <= index
    invariant forall k :: 0 <= k < index &&
        k < a.Length ==> a[k] != 0
    {
        if a[index] == 0 { return; }
        SkippingLemma(a,index);
        index := index + a[index];
    }
    index := -1;
}
```