Proving Loops

Testing debugging and verification

Srinivas Pinisetty

Based on material from Atze van der Ploeg, Wolfgang Aherndt,..

Weakest precondition rules

wp is a *computable* function!

```
wp({}          , R) = R
wp(x := e      , R) = R[x →e]
wp(S1 ; S2     , R) = wp(S1, wp(S2,R))
wp(assert B,     R) = B && R
wp(if B {S1} else {S2}, R) =
    ( B ==> wp(S1,R)) && (!B ==> wp(S2,R))
wp(if B {S1} else {S2}, R) =
    ( B && wp(S1,R)) || (!B && wp(S2,R))
```

While loops

But what about while loops?

`wp(while B { S }, R) = ?`

Is ***not*** computable!

No algorithm ***can*** exist that always computes `wp(while B { S }, R)` correctly!

Now what?

```
while B
{ S }
```


```
while B
invariant I
decreases D
{ S }
```

Verifying programs with loops

- How do we use the invariant and variant to compute the **wp** of a while-loop?

- Partial correctness: prove programs containing loop *if we assume that the loop terminates*
- Total correctness: prove programs containing loop *without assumption*

Recall: What is a loop invariant?

```
method simpleInvariant(n : int) returns (m : int)
requires n >= 0
ensures  n == m {
  m := 0;
  while m < n
  invariant m <= n
  { m := m + 1; }
}
```

A loop invariant is true after *any number* of iterations of the loop (including 0)

- Before entering the loop.

- After each iteration of the loop.

- After exiting the loop.

Another loop invariant

```
method simpleInvariant(n : int) returns (m : int)
requires n >= 0
ensures  n == m
{
  m := 0;
  while m < n
  invariant 0 < 1
  { m := m + 1; }
}
```

To be *useful*, a loop invariant must allow us to prove the program

What is a loop invariant?

Invariant **m<=n** seems more useful! (allows us to prove post condition!)

After the loop **(m < n)** must be **false**

```
method simpleInvariant(n : int) returns (m :
int)
requires n >= 0
ensures  n == m
{
  m := 0;
  while m < n
  invariant m <= n
  { m := m + 1; }
}
```

!(m < n) && m <= n ==> n == m
= m >= n && m <= n ==> n == m
= true!

Partial correctness wp for while

Loop guard

Loop Invariant

Statements

Post-condition

```
wp(while B I S, R) =
    I
    && (B && I ==> wp(S,I))
    && (!B && I ==> R)
```

Invariant holds before loop

If invariant holds before loop, then it also hold afterwards

The failure of the loop guard (!B) and the invariant imply the postcondition R

wp for While - example

```
wp(x := e  , R) = R[x →e]
wp(S1 ; S2 , R) = wp(S1,
wp(S2,R))
wp(assert B, R) = B && R
wp(if B {S1} else {S2}, R) =
    ( B ==> wp(S1,R)) &&
    (!B ==> wp(S2,R))
```

```
wp(while B I S, R) =
    I
    &&(B && I ==> wp(S,I))
    &&(!B && I ==> R)
```

```
wp( while m < n
    invariant m <= n
    { m := m + 1; } , n == m)
```

```
= m <= n
  && (m < n && m <= n ==> wp(m := m + 1, m <= n))
  && (!(m < n) && m <= n ==> n == m)
```

```
(m < n && m <= n ==> wp(m := m + 1, m <= n))
```

```
= (m < n && m <= n ==> m + 1 <= n) (by assignment rule)
```

```
= (m < n  ==> m + 1 <= n) (simplify using m < n ==> m <= n)
```

```
= (m + 1 <= n  ==> m + 1 <= n) (simplify using m < n == m + 1 <= n)
```

```
= true (simplify using p ==> p == true)
```

wp for While - example

```
wp( while m < n
     invariant m <= n
     { m := m + 1; } , n == m)
```

```
wp(x := e  , R) = R[x →e]
wp(S1 ; S2 , R) = wp(S1,
wp(S2,R))
wp(assert B, R) = B && R
wp(if B {S1} else {S2}, R) =
     ( B ==> wp(S1,R)) &&
     (!B ==> wp(S2,R))
```

```
wp(while B I S, R) =
    I
    && (B && I ==> wp(S,I))
    && (!B && I ==> R)
```

```
= m <= n
  && (m < n && m <= n ==> wp(m := m + 1, m <= n))
  && (!(m < n) && m <= n ==> n == m)
```

```
 (!(m < n) && m <= n ==> n == m)
```

```
= (m >= n && m <= n ==> n == m)) (by !(m < n) == m >= n)
```

```
= true
```

wp for While - example

```
wp( while m < n
    invariant m <= n
    { m := m + 1; } , n == m)
```

```
= m <= n
  && true
  && true
```

```
= m <= n (by a && true == a)
```

```
wp(x := e  , R) = R[x →e]
wp(S1 ; S2 , R) = wp(S1,
wp(S2,R))
wp(assert B, R) = B && R
wp(if B {S1} else {S2}, R) =
    ( B ==> wp(S1,R)) &&
    (!B ==> wp(S2,R))
```

```
wp(while B I S, R) =
    I
    && (B && I ==> wp(S,I))
    && (!B && I ==> R)
```

Proving a program with a while loop: partial correctness

```
method simpleInvariant(n : int) returns (m : int)
requires n >= 0
ensures  n == m {
  m := 0;
  while m < n
  invariant m <= n
  { m := m + 1; }
}
```

Compute the *weakest precondition:* wp(S,R)

Check if Q ⇒ wp(S,R)

```
wp(m := 0;
   while m < n
   invariant m <= n
   { m := m + 1; }, n == m)
```

```
= wp(m := 0,
   wp(while m < n
      invariant m <= n
      { m := m + 1; }, n == m)
   ) (by sequential rule)
```

```
= wp(m := 0,  m <= n) (from previous slide)
```

```
= n >= 0 (by assigment rule)
```

Proving a program with a while loop: partial correctness

```
method simpleInvariant(n : int) returns (m : int)
requires n >= 0
ensures  n == m {
  m := 0;
  while m < n
  invariant m <= n
  { m := m + 1; }
}
```

Compute the *weakest precondition:* wp(S,R)

```
= n >= 0
```

Check if Q ⇒ wp(S,R)

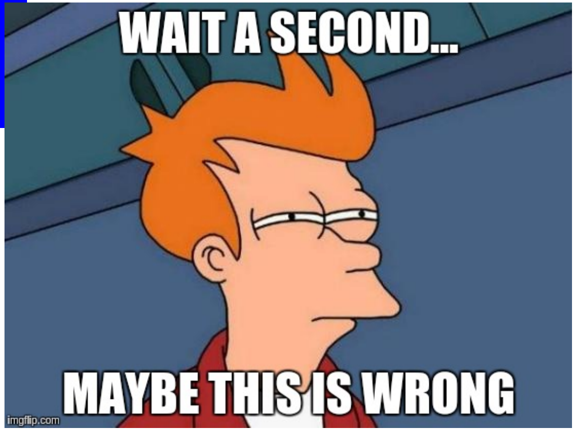
```
= n >= 0 ==> n >= 0
```

```
= true
```

Another proof!

```
wp(while B I S, R) =
    I
    && (B && I ==> wp(S,I))
    && (!B && I ==> R)
```

```
method magic returns ()
requires true
ensures  1 == 0 {
  while 1 != 0
  invariant true
  { ; }
}
```



We need to show:

Compute the *weakest precondition:* wp(S,R)

Check if Q ⇒ wp(S,R)

```
wp( while 1 != 0 true {},1 == 0)
```

```
= true &&
  (true && (1 != 0) ==> wp({}, true)) &&
  (!(1 != 0) && true ==> 1 == 0
```

```
=  (!(1 != 0) && true ==> 1 == 0 (simplify)
```

```
=  true
```

- We proved ***partial correctness***: correct ***assuming*** *that the loop terminates*
- `magic` breaks that assumption!
- Next up: total correctness!

How do we prove termination? (Loop Variant)

Proving termination is also undecidable (need to provide loop variants).

```
method simpleTermination(n : int) returns (m : int)
requires n >= 0
ensures  n == m {
  m := 0;
  while m < n
  decreases (n - m)
  invariant m <= n
  { m := m + 1; }
}
```

Recall: **variants**, Expression which decrease at each loop iteration (Bounded from below by 0).

- Provide decreases expression D (Often derived automatically in Dafny).
- **The value of D is always >= 0**
- **Show that after each iteration of the loop, the value D is less than before the loop iteration**

“Each iteration brings us closer to the last iteration”

How do we prove termination?

```
method x(...) returns (...)
... {
  ...;
  while B
  decreases D
  invariant I
  { S }
}
```

- The value of D is always >= 0

```
B && I ==> D >= 0
```

- Show that after each iteration of the loop, the value D is less than before the loop

```
B && I ==> wp(tmp := D ; S, tmp > D)
```

Termination example

```
method simpleInvariant(n : int) returns (m : int)
requires n >= 0
ensures  n == m {
  m := 0;
  while m < n
  decreases (n - m)
  invariant m <= n
  { m := m + 1; }
}
```

- (a) The value of D is always >= 0

```
B && I ==> D >= 0
```

- (b) Show that after each iteration of the loop, the value D is less than before the loop

```
B && I ==> wp(tmp := D ; S, tmp > D)
```

Proof of (a)

```
m <= n ==> n - m >= 0
```

Simplify

```
= m <= n ==> n >= m
```

Simplify

```
true
```

Proof of (b)

```
m < n && m <= n ==> wp(...)
```

Simplify

```
= m < n ==>
wp(tmp := n - m ; m := m +1, tmp > n - m)
```

```
wp(tmp := n - m ; m := m +1, tmp > n - m)
```

seq rule

```
= wp(tmp := n - m , wp (m := m +1, tmp > n - m))
```

assign

```
= wp(tmp := n - m , tmp > n - (m + 1))
```

assign

```
= n - m > n - (m + 1)
```

simplify

```
= n - m > n - m - 1
```

simplify

```
= true
```

```
= m < n ==> true
```

p ==> true == true

```
= true
```

Total correctness - summary

Loop guard, Loop Invariant, Decreases expression, Statements, Post-condition

```
wp(while B I D S, R) =
    I
    && (B && I ==> wp(S,I))
    && (!B && I ==> R)

    && (B && I ==> D >= 0)
    && (B && I ==>
        wp({tmp := D;S},{tmp > D}))
```

- Invariant holds before loop
- If invariant holds before loop, then it also hold afterwards
- The failure of the loop guard (!B) and the invariant imply the postcondition R
- Decreases expression is always >= 0
- Decreases expression decreases each iteration

Prove m1 correct!

```
wp(x := e  , R) = R[x →e]
wp(S1 ; S2 , R) = wp(S1,
wp(S2,R))
wp(assert B, R) = B && R
wp(if B {S1} else {S2}, R) =
    ( B ==> wp(S1,R)) &&
    (!B ==> wp(S2,R))
```

```
wp(while B I D S, R) =
   I
   && (B && I ==> wp(S,I))
   && (!B && I ==> R)

   && (B && I ==> D >= 0)
   && (B && I ==>
       wp({tmp := D ; S},{tmp > D}))
```

```
method m1(n : nat) returns (i : nat)
requires n >= 0
ensures i == 2*n

{
i := 0;
while (i < n)
 invariant i <= n
 variant n-i
 { i := i + 1; }
i := 2*i;
}
```

Prove fibFast correct!

```
wp(x := e  , R) = R[x →e]
wp(S1 ; S2 , R) = wp(S1,
wp(S2,R))
wp(assert B, R) = B && R
wp(if B {S1} else {S2}, R) =
    ( B ==> wp(S1,R)) &&
    (!B ==> wp(S2,R))
```

```
wp(while B I D S, R) =
   I
   && (B && I ==> wp(S,I))
   && (!B && I ==> R)

   && (B && I ==> D >= 0)
   && (B && I ==>
       wp({tmp := D ; S},{tmp > D}))
```

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

method fibFast(n : nat) returns (c : nat)
requires n >= 1
ensures c == fib(n)
{
  var p := 0;
  c  := 1;
  var i := 1;
  while i < n
  invariant 1 <= i <= n
  invariant p == fib(i - 1) && c == fib(i)
  decreases (n - i)
  { var new := p + c;
    p := c;
    c := new;
    i := i + 1;
  }
}
```

Solution

The correctness of the method fibFast is expressed by the formula
n >= 1 ==> wp(p := 0; ... , c == fib(n))
<=> { sequential composition X4 }
n >= 1 ==> wp(p := 0, wp(c := 1, wp(i := 1, wp(while(i < 1) invariant I do {...}, wp({}, c == fib(n))))))
<=> { empty program }
n >= 1 ==> wp(p := 0, wp(c := 1, wp(i := 1, wp(while(i < 1) invariant I do {...}, c == fib(n)))))

The step is to compute the weakest-precondition of the loop. Since the formula is quite big, let's first take care of some "sub-goals":

Let us prove that the invariant is preserved "`B && I ==> wp(S,I)`":
B && I ==> wp(new := p + c; ..., I)
<=> { sequential composition X5 }
B && I ==> wp(new := p + c, wp(p := c, wp(c := new, wp(i := i + 1, wp({}, I)))))
<=> { empty program }
B && I ==> wp(new := p + c, wp(p := c, wp(c := new, wp(i := i + 1, I))))
<=> { assignment (and unfolding the definition of I) }
B && I ==> wp(new := p + c, wp(p := c, wp(c := new, 1 <= i + 1 <= n && p == fib(i) && c == fib(i + 1))))
<=> { assignment }
B && I ==> wp(new := p + c, wp(p := c, 1 <= i + 1 <= n && p == fib(i) && new == fib(i + 1)))

```
B && I ==> wp(new := p + c, wp(p := c, 1 <= i + 1 <= n && p == fib(i) && new == fib(i + 1)))
<=> { assignment }
B && I ==> wp(new := p + c, 1 <= i + 1 <= n && c == fib(i) && new == fib(i + 1))
<=> { assignment (and unfolding the definition of B and I}
i < n && 1 <= i <= n && p == fib(i - 1) && c == fib(i) ==> 1 <= i + 1 <= n && c == fib(i) && p + c == fib(i + 1)
<=> { definition of fib }
i < n && 1 <= i <= n && p == fib(i - 1) && c == fib(i) ==> 1 <= i + 1 <= n && c == fib(i) && p + c == if (i + 1 <= 1) then i + 1 else
fib(i) + fib(i - 1)
<=> { rewriting the RHS of the implication with equalities on the LHS }
i < n && 1 <= i <= n && p == fib(i - 1) && c == fib(i) ==> 1 <= i + 1 <= n && c == c && p + c == if (i <= 0) then i + 1 else c + p
<=> { we have i <= i on the LHS, so we can simplify the if/then/else expression on the RHS }
i < n && 1 <= i <= n && p == fib(i - 1) && c == fib(i) ==> 1 <= i + 1 <= n && c == c && p + c == c + p
<=> { removing trivial equalities }
i < n && 1 <= i <= n && p == fib(i - 1) && c == fib(i) ==> 1 <= i + 1 <= n
<=> { 1 <= i on the LHS implies 1 <= i + 1, and i < n implies i + 1 <= n }
i < n && 1 <= i <= n && p == fib(i - 1) && c == fib(i) ==> true
<=> { the RHS of the implication is true }
true
```

We will also prove that the failure of loop guard, and invariant implies the post-condition (this is post-condition of the loop, but since the loop is followed by the empty program it is similar to the post-condition of the method):

!B && I ==> c == fib(n)
<=> { definition of B and I }
!(i < n) && 1 <= i <= n && p == fib(i - 1) && c == fib(i) ==> c == fib(n)
<=> { arithmetic }
i >= n && 1 <= i <= n && p == fib(i - 1) && c == fib(i) ==> c == fib(n)
<=> { i >= n and i <= n are equivalent to i == n }
i == n && i <= n && p == fib(i - 1) && c == fib(i) ==> c == fib(n)
<=> { rewriting the RHS of the implication with the equality on the LHS }
i == n && i <= n && p == fib(i - 1) && c == fib(i) ==> c == fib(i)
<=> { the RHS is implied by the LHS }
true

Let us also prove that the decrease expression D is bounded below by 0

```
B && I ==> n - i >= 0
<=> { definition of I (only the relevant bit) }
i < n ==> n - i >= 0
<=> { arithmetic }
i < n ==> n >= i
<=> { trivial arithmetical fact }
true
```

Finally let's prove that the decrease expression actually decreases

B && I ==> wp(tmp := D; new := p + c; ..., tmp > D)
<=> { sequential composition X5 }
B && I ==> wp(tmp := D, wp(new := p + c, wp(p := c, wp(c := new, wp(i := i + 1, wp({}, tmp > D))))))
<=> { empty program }
B && I ==> wp(tmp := D, wp(new := p + c, wp(p := c, wp(c := new, wp(i := i + 1, tmp > D)))))
<=> { assignment }
B && I ==> wp(tmp := D, wp(new := p + c, wp(p := c, wp(c := new, tmp > n - i + 1))))
<=> { assignment X4 }
B && I ==> D > n - i + 1
<=> { definition of D }
B && I ==> n - i > n - i + 1
<=> { arithmetic }
B && I ==> true
<=> { RHS is true }
true

Now we can go back to our original goal, proving the correctness of the method:

```
n >= 1 ==> wp(p := 0, wp(c := 1, wp(i := 1, wp(while(i < 1) invariant I do {...}, c == fib(n)))))
<=> { while rule + facts proven above }
n >= 1 ==> wp(p := 0, wp(c := 1, wp(i := 1, I && true && true && true && true)))
<=> { assignment }
n >= 1 ==> wp(p := 0, wp(c := 1, 1 <= 1 <= n && p == fib(0) && c == fib(1)))
<=> { assignment }
n >= 1 ==> wp(p := 0, 1 <= 1 <= n && p == fib(0) && 1 == fib(1))
<=> { assignment }
n >= 1 ==> 1 <= 1 <= n && 0 == fib(0) && 1 == fib(1)
<=> { definition of fib (computed directly) }
n >= 1 ==> 1 <= 1 <= n && 0 == 0 && 1 == 1
<=> { trivial equalities }
n >= 1 ==> 1 <= 1 <= n
<=> { arithmetic }
true
```

Therefore, the method satisfies its specification!

```
wp({}        , R)          = R
wp(x := e   , R)          = R[x →e]
wp(S1 ; S2 , R)         = wp(S1, wp(S2,R))
wp(assert B, R)  = B && R
wp(if B {S1} else {S2}, R) =
    ( B ==> wp(S1,R)) && (!B ==> wp(S2,R))
wp(if B {S1} else {S2}, R) =
    ( B && wp(S1,R)) || (!B && wp(S2,R))
```

```
wp(while B I D S, R) =
  I
  && ∀ [ B && I ==> wp(S,I) ]
  && ∀ [ !B && I ==> R ]
  && ∀ [ B && I ==> D >= 0 ]
  && ∀ [ B && I ==>  wp({tmp := D;S},{tmp > D}) ]
```

$$wp(\textbf{while}\ b\ \textbf{do}\ S, Q) \equiv \exists k.\ (\ k \geq 0 \wedge P_k\ )$$

where P_k is defined inductively:

$$P_0 \equiv \neg b \wedge Q$$

$$P_{k+1} \equiv b \wedge wp(S, P_k)$$