Deriving Loop Invariants and Loop Variants

Srinivas Pinisetty\textsuperscript{1}

17 December 2018

\textsuperscript{1}Lecture slides based on material from Wolfgang Aherndt,..
Example

requires \( n \geq 0 \)
ensures \( i = 2n \)
i := 0;
while (i < n) {
    i := i + 1;
}
i := i \times 2;

What needs to be true after the loop?

\[ wp(i:=i*2, i == 2*n) \] which is equal to \( i = n \)
Example

requires n >= 0
ensures i == 2*n
i := 0;
while (i < n) {
    i := i + 1;
}
i := i * 2;

What needs to be true after the loop?

- wp(i := i*2, i == 2*n) which is equal to (i == n)
- What, in addition to negated guard (i >= n), is needed to prove this?
How to Derive Loop Invariants?

Example

requires n >= 0
ensures i == 2*n
i := 0;
while (i < n) {
    i := i + 1;
}
i := i * 2;

What needs to be true after the loop?

- wp(i := i*2, i == 2*n) which is equal to (i == n)
- What, in addition to negated guard (i >= n), is needed to prove this?
- (i <= n) (why isn’t (i < n) suitable?)
Example

requires n >= 0
ensures i == 2*n
i := 0;
while (i < n) {
  i := i + 1;
}
i := i * 2;

What needs to be true after the loop?

- wp(i:= i*2, i == 2*n) which is equal to (i == n)
- What, in addition to negated guard (i >= n), is needed to prove this?
  - (i <= n) (why isn’t (i < n) suitable?)
  - (i <= n) is established before loop, and is preserved.
How to Derive Loop Variants?

Example

requires \( n \geq 0 \)
ensures \( i = 2*n \)
i := 0;
while (i < n) {
    i := i + 1;
}
i := 2*i;

What happens to the loop counter?

Look at the loop counter, \( i \). It starts at 0 and increments by one on each iteration, until reaching \( n \). Hence, the difference between \( i \) and \( n \) shrink each time. Candidate variant: \( n-i \)
How to Derive Loop Variants?

Example

requires n >= 0
ensures i == 2*n
i := 0;
while (i < n) {
    i := i + 1;
}
i := 2*i;

What happens to the loop counter?

Look at the loop counter, \(i\). It starts at 0 and increments by one on each iteration, until reaching \(n\). Hence, the difference between \(i\) and \(n\) shrink each time. Candidate variant: \(n-i\)

Is \((n-i)\) bounded from below by 0?

Yes! We have found a suitable loop variant!
Example

requires n >= 0
ensures i == 2*n
i := 0;
while (i < n)
  invariant i <= n
  variant n-i
  { i := i + 1; }
i := 2*i;
Example (Silly Addition)

method SillyAdd (x:int, y:int) returns (z:int)
ensures z==x+y
{
    var i := y;
    z := x;
    while (i > 0) {
        z := z + 1;
        i := i - 1;
    }
}
Example (Silly Addition)

method SillyAdd (x:int, y:int) returns (z:int)
ensures z==x+y
{
    var i := y;
    z := x;
    while (i > 0) {
        z := z + 1;
        i := i - 1;
    }
}

Finding the invariant
First attempt: use postcondition z == x+y
Finding the invariant: Example

Example (Silly Addition)

method SillyAdd (x:int, y:int) returns (z:int)
ensures z==x+y
{
var i := y;
z := x;
while (i > 0) {
    z := z + 1;
i := i - 1;}}

Finding the invariant
First attempt: use postcondition $z == x+y$
  ▶ Not true at start whenever $y \neq 0$
  ▶ Not preserved by loop, because $z$ is increased
Example (Silly Addition)

method SillyAdd (x:int, y:int) returns (z:int)
ensures z==x+y
{
    var i := y;
    z := x;
    while (i > 0) {
        z := z + 1;
        i := i - 1;
    }
}
Example (Silly Addition)

method SillyAdd (x:int, y:int) returns (z:int)
ensures z==x+y
{
  var i := y;
  z := x;
  while (i > 0) {
    z := z + 1;
    i := i - 1;}

Finding the invariant

What stays invariant?
- The sum of z and i: \( z + i = x + y \) “Generalization”
- Can help to think of partial result: “\( \delta \)” between z and \( x + y \)
Finding the invariant: Example

Example (Silly Addition)

method SillyAdd (x:int, y:int) returns (z:int)
ensures z==x+y
{
var i := y;
z := x;
while (i > 0) {
z := z + 1;
i := i - 1;}}

Checking the invariant
Is z + i = x + y a good invariant?
Example (Silly Addition)

method SillyAdd (x:int, y:int) returns (z:int)
ensures z==x+y
{
    var i := y;
    z := x;
    while (i > 0) {
        z := z + 1;
        i := i - 1;
    }
}

Checking the invariant

Is $z + i = x + y$ a good invariant?

- Holds in the beginning and is preserved by loop
Example (Silly Addition)

method SillyAdd (x:int, y:int) returns (z:int)
ensures z==x+y
{
  var i := y;
  z := x;
  while (i > 0) {
    z := z + 1;
    i := i - 1;
  }
}

Checking the invariant
Is $z + i = x + y$ a good invariant?
- Holds in the beginning and is preserved by loop
- But postcondition not achieved by $z + i = x + y \land i \leq 0$
Finding the invariant: Example

Example (Silly Addition)

method SillyAdd (x:int, y:int) returns (z:int)
ensures z==x+y
{
  var i := y;
  z := x;
  while (i > 0) {
    z := z + 1;
    i := i - 1;
  }
}

Strengthening the invariant

Postcondition holds if y >= 0
  ▶ Sufficient to add i >= 0 to z + i = x + y && i <= 0
  ▶ Hints at missing precondition: y >= 0
Exercise

- Patch the specification contract for SillyAdd.
- In addition to the invariant from the example, also state a variant.
- Formally prove SillyAdd correct using the invariant and variant by following the "Checklist for loop correctness discussed earlier".
Solution

method SillyAdd (x:int, y:int) returns (z:int)
  requires y >= 0;
  ensures z==x+y;
  {
    var i := y;
    z := x;
    while (i > 0)
      invariant z + i == x + y && i >= 0;
      variant i;
      {
        z := z + 1;
        i := i - 1;
      }
  }

Formally prove SillyAdd correct
Some Tips On Finding Invariants

General Advice

- Invariants must be developed!
- Be as systematic in deriving invariants as when debugging a program
- Don’t forget: the program or contract (more likely) can be buggy
  - In this case, you won’t find an invariant!
Some Tips On Finding Invariants, Cont’d

- The desired **postcondition** is a good starting point
  - What, in addition to negated loop guard, is needed for it to hold?
The desired **postcondition** is a good starting point
- What, in addition to negated loop guard, is needed for it to hold?
- If the invariant candidate is not preserved by the loop body:
  - Does it need strengthening?
  - Can you add stuff from the precondition?
  - Try to express the relation between partial and final result
Some Tips On Finding Invariants, Cont’d

- The desired postcondition is a good starting point
  - What, in addition to negated loop guard, is needed for it to hold?
- If the invariant candidate is not preserved by the loop body:
  - Does it need strengthening?
  - Can you add stuff from the precondition?
  - Try to express the relation between partial and final result
- Simulate a few loop body executions to discover invariant patterns
Some Tips On Finding Invariants, Cont’d

- The desired postcondition is a good starting point
  - What, in addition to negated loop guard, is needed for it to hold?
- If the invariant candidate is not preserved by the loop body:
  - Does it need strengthening?
  - Can you add stuff from the precondition?
  - Try to express the relation between partial and final result
- Simulate a few loop body executions to discover invariant patterns
- If the invariant is not initially valid:
  - Can it be weakened such that the postcondition still follows?
  - Did you forget an assumption in the precondition?
Some Tips On Finding Invariants, Cont’d

- The desired **postcondition** is a good starting point
  - What, in addition to negated loop guard, is needed for it to hold?
- If the invariant candidate is **not preserved** by the loop body:
  - Does it need strengthening?
  - Can you add stuff from the precondition?
  - Try to express the relation between partial and final result
- Simulate a few loop body executions to discover invariant patterns
- If the invariant is **not initially valid**:
  - Can it be weakened such that the postcondition still follows?
  - Did you forget an assumption in the precondition?
- Several “rounds” might be required
Some Tips On Finding Invariants, Cont’d

- The desired **postcondition** is a good starting point
  - What, in addition to negated loop guard, is needed for it to hold?
- If the invariant candidate is **not preserved** by the loop body:
  - Does it need strengthening?
  - Can you add stuff from the precondition?
  - Try to express the relation between partial and final result
- Simulate a few loop body executions to discover invariant patterns
- If the invariant is **not initially valid**:
  - Can it be weakened such that the postcondition still follows?
  - Did you forget an assumption in the precondition?
- Several “rounds” might be required
- Use Dafny
  - Check which case of the loop invariant cannot be proved by the verifier
Exercise: Find the loop invariant

The Max method should return the maximum element in the array.

- Provide appropriate pre-conditions to prevent the method being called on inputs that would cause exceptions.
- Provide post-conditions stating that max indeed is the maximum value of arr
- Provide appropriate loop invariants which allows the post-conditions to be proved (hint: there are three).

```plaintext
method Max(arr : array<int>) returns (max : int)
{
    var i := 1;
    max := arr[0];
    while(i < arr.Length)
    {
        if(arr[i] > max)
        {
            max := arr[i];
        }
        i := i +1;
    }
}
```
method Max(arr : array<int>) returns (max : int)
requires arr != null && arr.Length > 0;
ensures forall i :: 0 <= i < arr.Length ==> max >= arr[i ];
ensures exists i :: 0 <= i < arr.Length && max == arr[i ];
{
    var i := 1;
    max := arr[0];
    while(i < arr.Length)
    invariant 0 < i <= arr.Length;
    invariant forall j :: 0 <= j < i ==> max >= arr[j ];
    invariant exists j :: 0 <= j < i && max == arr[j ];
    { 
        if(arr[i] > max)
        {max := arr[i];}
        i := i +1;
    }
}
The method \texttt{kthEven} is supposed to return the $k^{th}$ even number, where 0 is considered as the first.

Provide a suitable pre- and postconditions, as well as a loop invariant.

Prove correctness.

\begin{verbatim}
method kthEven(k : int) returns (e : int)
{
    e := 0;
    var i := 1;
    while (i < k)
    {
        e := e + 2;
        i := i + 1;
    }
}
\end{verbatim}
Exercise: Solution

method kthEven(k : int) returns (e : int)
requires k > 0;
ensures e == 2 * (k-1)
{
e := 0;
var i := 1;
while (i < k)
  invariant e == 2*(i-1) && i <= k
  {
    e := e + 2;
    i := i + 1;
  }
}
Invariant rule has three parts:

- The invariant must hold at the beginning of the loop.
- The invariant must be preserved by an arbitrary execution of the loop body provided that the guard is true.
- The negated guard plus the invariant imply the desired postcondition.

Loop invariants can be developed systematically:

- Start with the desired postcondition.
- Discover patterns through execution of a few loop bodies.
- Use strengthening, weakening, generalisation..

Remember, your program or contract might be wrong!