

CSC2/455 Software Analysis and Improvement

Proving Programs Correct

Sreepathi Pai

April 24, 2024

URCS

Outline

Proofs of Program Correctness

Loop Invariants

Theorem Proving

Postscript

Outline

Proofs of Program Correctness

Loop Invariants

Theorem Proving

Postscript

What is a correct program?

- A program that meets its specification is a correct program
- What is the correct specification for a program?
 - The scope of this question is beyond this course
 - Not entirely technical
- Our goal is only to study methods that check if a program meets its provided specification
 - Technical only

Our simple program

```
void fn(int k) {  
    int x = k;  
    int c = 0;  
  
    while(x > 0) {  
        c = c + 1;  
        x = x - 1;  
    }  
  
    assert(x == 0);  
    assert(c == k);  
}
```

Will those assertions always be true? [i.e. are they always valid?]

CBMC: Try #1

```
$ cbmc --function fn --trace simple1.c
CBMC version 5.10 (cbmc-5.10) 64-bit x86_64 linux
Parsing simple1.c
Converting
Type-checking simple1
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Generic Property Instrumentation
Running with 8 object bits, 56 offset bits (default)
Starting Bounded Model Checking
Unwinding loop fn.0 iteration 1 file simple1.c line 14 function fn thread 0
Unwinding loop fn.0 iteration 2 file simple1.c line 14 function fn thread 0
Unwinding loop fn.0 iteration 3 file simple1.c line 14 function fn thread 0
Unwinding loop fn.0 iteration 4 file simple1.c line 14 function fn thread 0
Unwinding loop fn.0 iteration 5 file simple1.c line 14 function fn thread 0
Unwinding loop fn.0 iteration 6 file simple1.c line 14 function fn thread 0
Unwinding loop fn.0 iteration 7 file simple1.c line 14 function fn thread 0
Unwinding loop fn.0 iteration 8 file simple1.c line 14 function fn thread 0
Unwinding loop fn.0 iteration 9 file simple1.c line 14 function fn thread 0
Unwinding loop fn.0 iteration 10 file simple1.c line 14 function fn thread 0
Unwinding loop fn.0 iteration 11 file simple1.c line 14 function fn thread 0
Unwinding loop fn.0 iteration 12 file simple1.c line 14 function fn thread 0
...
```

Whoops, infinite loop!

CBMC: Try #2

```
$ cbmc --function fn --unwind 10 --trace simple1.c
CBMC version 5.10 (cbmc-5.10) 64-bit x86_64 linux
...
Starting Bounded Model Checking
Unwinding loop fn.0 iteration 1 file simple1.c line 14 function fn thread 0
Unwinding loop fn.0 iteration 2 file simple1.c line 14 function fn thread 0
...
Unwinding loop fn.0 iteration 8 file simple1.c line 14 function fn thread 0
Unwinding loop fn.0 iteration 9 file simple1.c line 14 function fn thread 0
Not unwinding loop fn.0 iteration 10 file simple1.c line 14 function fn thread 0
...

** Results:
[fn.assertion.1] assertion x == 0: FAILURE
[fn.assertion.2] assertion c == k: FAILURE

...

k: -2147483648 (10000000 00000000 00000000 00000000)
x:-2147483648 (10000000 00000000 00000000 00000000)
c=0 (00000000 00000000 00000000 00000000)

Violated property: assertion x == 0
x == 0

Violated property: assertion c == k
c == k
```

- if k is negative (note: output is reformatted to fit)
 - x will not be zero
 - c will not be equal k

Specifying k must always be greater than zero

- We check our specifications, and notice that `fn` should only work on non-negative k

```
void fn(int k) {  
    __CPROVER_assume(k >= 0);  
    ...  
}
```


CBMC: Try #3

```
$ cbmc --unwinding-assertions --function fn --unwind 10 --trace simple1.c
CBMC version 5.10 (cbmc-5.10) 64-bit x86_64 linux
...
Unwinding loop fn.0 iteration 9 file simple1.c line 14 function fn thread 0
Not unwinding loop fn.0 iteration 10 file simple1.c line 14 function fn thread 0
...
** Results:
[fn.assertion.1] assertion x == 0: SUCCESS
[fn.assertion.2] assertion c == k: SUCCESS
[fn.unwind.0] unwinding assertion loop 0: FAILURE

Trace for fn.unwind.0:

INPUT k: 12 (00000000 00000000 00000000 00001100)
c=10 (00000000 00000000 00000000 00001010)
x=2 (00000000 00000000 00000000 00000010)

Violated property:
  unwinding assertion loop 0
```

- CBMC can't show loop terminates for a (fixed) finite number of unwindings
 - Here unwind=10 and CBMC says more unwindings would be needed for $k = 12$
- Conclusions may be unsound

Try out all possible unwindings

- For C , k is still an integer.
 - Finite number of values
 - Could try out all possible unwindings by fixing an upper bound
- Might be feasible for `simple`
 - But add more loops, and time/space increases significantly
- Strategy not even feasible for languages like Python
 - Python has infinite precision integers
- Can we try something else?

```
void fn(int k) {
    int x = k;
    int c = 0;

    while(x > 0) {
        c = c + 1;
        x = x - 1;
    }

    assert(x == 0);
    assert(c == k);
}

int main(void) {
    int k;

    klee_make_symbolic(&k, sizeof(k), "k");
    klee_assume(k >= 0);
    fn(k);
}
```

KLEE, contd.

```
clang -I ~/ext/klee-2.1/include/ -emit-llvm -c -g -O0 -Xclang -disa  
~/ext/klee-2.1/build/bin/klee simple1.bc  
KLEE: output directory is "src/klee-out-0"  
KLEE: Using Z3 solver backend  
^CKLEE: ctrl-c detected, requesting interpreter to halt.  
KLEE: halting execution, dumping remaining states  
  
KLEE: done: total instructions = 5174  
KLEE: done: completed paths = 273  
KLEE: done: generated tests = 273
```

- Symbolic execution using KLEE doesn't seem to work either
 - I interrupted after a minute or so.
 - Without `klee_assume`, KLEE also detects the assertion failure of `x == 0`

What about abstract interpretation?

```
x := k;  
c := 0;  
while(x > 0) {  
    x := (x - 1);  
    c := (c + 1)  
}
```

- Input:
 - $M^\# = \{k \mapsto [0, +\infty), x \mapsto \top, c \mapsto \top\}$
- Output:
 - $\{ 'k': (0, +\text{inf}), 'x': (0, 0), 'c': (0, +\text{inf}) \}$
 - $M^\# = \{k \mapsto [0, +\infty), x \mapsto [0, 0], c \mapsto [0, +\infty)\}$

Does M^\sharp allow us to prove our assertions?

$$M^\sharp = \{k \mapsto [0, +\infty), x \mapsto [0, 0], c \mapsto [0, +\infty)\}$$

- Logically $P : (k \geq 0) \wedge (x = 0) \wedge (c \geq 0)$
 - We want to prove $a_0 : x = 0$
 - We want to prove $a_1 : c = k$
- For a_0
 - If P is valid, then so is a subset of P , in particular $P_0 : (x = 0)$
 - (This is because $a \wedge b \wedge c \implies a$ is valid)
 - $P_0 \implies a_0$ is valid (also written as $P_0 \models a_0$)
- This won't work for a_1
 - $P_1 : (k \geq 0) \wedge (c \geq 0)$ [any subset can be chosen]
 - $P_1 \not\models (c = k)$
 - Not strong enough. Counterexample: $k = 6, c = 5$
 - Recall intervals domain is not relational, so can't relate c to k

simple.c, logically deriving P_0

```
void fn(int k) {
    int x = k;
    int c = 0;

    while(x > 0) {
        c = c + 1;
        x = x - 1;
    }

    assert(x == 0);
    assert(c == k);
}
```

- Clearly, P_0 captures the state of the program at the end of the loop well enough to allow us to prove $x = 0$
- Can we derive P_0 (logically)?
 - First glance, *only* from loop condition, all we can say is that $x \leq 0$ if loop executes and exits.
 - Not strong enough to prove $x = 0$

Loops ...

- Loops may execute zero, a finite number, or an infinite number of iterations
 - Bounded Model Checkers: Can't handle loops soundly without a fixed upper bound
 - Symbolic checkers: same
 - Abstract interpretation: Approximation may prevent us from verifying some properties
- But if we can find a P that captures the state of the program at the end of a loop
 - executing zero, finite or infinite number of iterations
 - P may be strong enough to prove properties we're interested in
 - without having to model the loop iteration by iteration

Outline

Proofs of Program Correctness

Loop Invariants

Theorem Proving

Postscript

Loop Invariants

- A loop invariant is a condition over the program state that holds:
 - Before the loop
 - At the beginning of each iteration
 - At the end of each iteration

A loop invariant in `simple.c`

```
assert(x >= 0);  
  
while(x > 0) {  
    assert(x >= 0);  
  
    c = c + 1;  
    x = x - 1;  
  
    assert(x >= 0);  
}
```

- $x \geq 0$ holds before the loop (since $x = k$, and $k \geq 0$)
- $x \geq 0$ holds at beginning of iteration, since $x > 0$ (from loop condition)
- $x \geq 0$ holds at end of iteration
 - x is reduced by 1 each iteration
 - $x > 0 \implies x \geq 1 \implies x - 1 \geq 0$

Using the loop invariant to prove $x == 0$

- At end of loop
 - $x \leq 0$ (from loop condition, if loop exits, then $\neg(x > 0)$ holds)
 - $x \geq 0$ (from loop invariant)
 - $x \leq 0 \wedge x \geq 0 \implies x = 0$
- What about $c == k$?

Trying out some candidate loop invariants for $c == k$

- Will $c \leq k$ work?

```
assert(c <= k);  
while(x > 0) {  
    assert(c <= k);  
  
    c = c + 1;  
    x = x - 1;  
    assert(c <= k);  
}
```

- Definitely holds before loop ($k \geq 0$, and $c = 0$)
- But harder to show that c won't exceed k during loop
 - We *know* it is true, but hard to prove!
 - We only know $x > 0$ at the beginning of each iteration
 - Hard to show that $c + 1 \leq k$ from that premise (even assuming $c \leq k$)
 - In fact $c \leq k$ allows $c = k$ which would mean $c + 1 > k$!

Change the loop condition?

```
assert(c <= k);  
while(c < k) {  
    assert(c <= k);  
  
    c = c + 1;  
    x = x - 1;  
  
    assert(c <= k);  
}
```

- Definitely holds before loop ($k \geq 0$, and $c = 0$)
- Holds on entry to loop as well $c < k \implies c \leq k$
- Holds after each iteration as well:
 - $c + 1 \leq k + 1$, (from invariant)
 - $c < k$ (from loop condition)
 - $c + 1 \leq k$

Using the loop invariant to prove $c == k$

- At end of loop
 - $c \geq k$ (from loop condition, if loop exits, then $\neg(c < k)$ holds)
 - $c \leq k$ (from loop invariant)
 - $c \leq k \wedge c \geq k \implies c = k$
- What about $x == 0$?
 - Back to square one?
- How about combining the loop conditions and the invariants?

Combining the loop invariants and loop conditions

```
assert(x >= 0 && c <= k);
while(x > 0 && c < k) {
    assert(x >= 0 && c <= k);

    c = c + 1;
    x = x - 1;

    assert(x >= 0 && c <= k);
}
```

- This doesn't seem to work
 - Not strong enough to imply either assertion after combination with loop exit condition!
 - If you work it out, you may be tempted to change the loop condition...

Let's look at some concrete program executions

- $k = 5$

entry : k: 5, x: 5, c: 0

end: k: 5, x: 4, c: 1

entry: k: 5, x: 4, c: 1

end: k: 5, x: 3, c: 2

entry: k: 5, x: 3, c: 2

end: k: 5, x: 2, c: 3

entry: k: 5, x: 2, c: 3

end: k: 5, x: 1, c: 4

entry: k: 5, x: 1, c: 4

end: k: 5, x: 0, c: 5

exit: k: 5, x: 0, c: 5

- Do you see a relation between x , c , and k ?
- Do you see a pattern that is unchanging (i.e. invariant)?

Invariant candidate #4: $x + c == k$

```
assert(x + c == k);
while(x > 0) {          /* note original loop condition */
    assert(x + c == k);

    c = c + 1;
    x = x - 1;

    assert(x + c == k);
}
```

- Clearly holds before entering loop and on first iteration
 - $x = k \wedge c = 0 \implies x + c = k$
- Assume holds at some iteration
 - $x + c = k$
- Then, it still holds at end of iteration (and next iteration)
 - $x - 1 + c + 1 = k$
- (Inductive argument)

Proving a_0 and a_1

- $P : \neg(x > 0) \wedge (x + c = k)$
 - For a_0 : $(x \leq 0) \wedge (x + c = k) \implies x = 0$
 - For a_1 : $(x \leq 0) \wedge (x + c = k) \implies c = k$
- Can't prove these using P as derived, since P admits $x < 0$.
 - We want $x = 0$ for the proof to go through
 - Without $x = 0$, setting $x = -1$ is a counterexample for both $P \implies a_0$ and $P \implies a_1$
- But we can derive that $x \geq 0$
 - We are given that $k \geq 0$
 - $\{k \geq 0\} x := k \{x \geq 0\}$ (*assignment axiom*)
- We can therefore strengthen P by adding $x \geq 0$
 - Allowed since $P \wedge \text{true}$ is still true
- This allows both the proofs to go through!
 - $(x \leq 0) \wedge (x + c = k) \wedge (x \geq 0) \implies x = 0$

Summary

- A loop invariant captures the effects of a loop on the program state
 - Without having to “run” or approximate states
 - Just have to prove the invariant satisfies the definition
- A useful loop invariant allows us to prove properties
 - May require additional facts given or derived from other parts of the program
- How to find loop invariants?
 - Use the Feynman “Algorithm” [not serious]
 - No general technique to find loop invariants!

Partial Correctness

- Big elephant in the room
 - Our proofs only hold if the loops terminate!
- Do the loops in the programs so far terminate?
 - Easy to show that they do
 - All have a strictly decreasing variable
 - Loop terminates when that variable reaches zero
- But, revisit the loop condition $x > 0 \ \&\& \ c < k$
 - with loop invariant $x \geq 0 \wedge c \leq k$
 - the negation of the loop condition prevents us from proving a_0 and a_1
- Might be tempted to use $x > 0 \ || \ c < k$ after figuring that out

Example

```
void fn(int k) {
    int x = k;
    int c = 0;

    assert(x >= 0 && c <= k);

    while(x > 0 || c < k) {
        assert(x >= 0 && c <= k);
        c = c + 1;
        x = x - 1;
        assert(x >= 0 && c <= k);
    }

    assert(x == 0);
    assert(c == k);
}
```

- Assume the loop invariant is still valid
- How do you prove the loop terminates?
 - Need to show that x becomes zero at the same time as c becomes k

Total Correctness

Total correctness = Partial Correctness + Termination

Outline

Proofs of Program Correctness

Loop Invariants

Theorem Proving

Postscript

Interactive Theorem Provers

- Sometimes called Proof Assistants
 - Isabelle
 - Coq
 - Lean
- Allow you to write proofs
 - Assist you in solving them
 - Proof writing is undecidable in general
- Verify your proofs are correct!
- Actually make writing proofs fun
 - Though still very tedious?





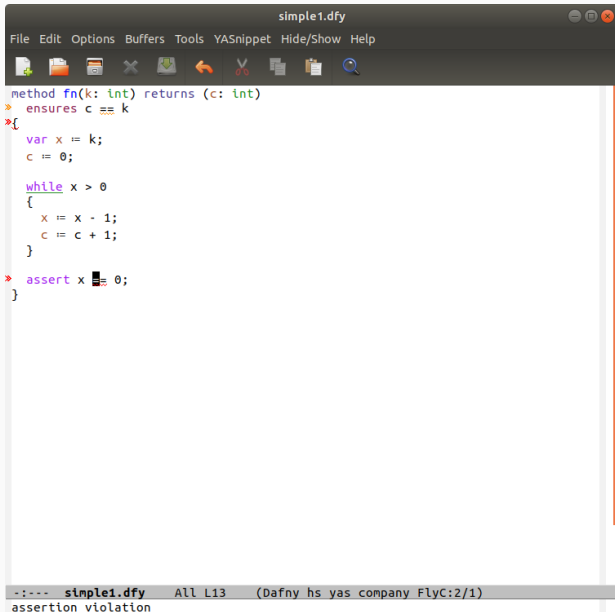
```
method DutchFlag(a: array<Color>)
  requires a ≠ null modifies a
  > ensures ∀ i,j · 0 ≤ i < j < a.Length ⇒ Ordered(a[i], a[j])
  ensures multiset(a[..]) == old(multiset(a[..]))
{
  var r, w, b = 0, 0, a.Length;
  > while w ≠ b
    invariant 0 ≤ r ≤ w ≤ b ≤ a.Length;
    invariant ∀ i · 0 ≤ i < r ⇒ a[i] == Red
    invariant multiset(a[..]) == old(multiset(a[..]))
  {
    match a[w]
    case Red ⇒
      a[r], a[w] = a[w], a[r];
      r, w = r + 1, w + 1;
    case White ⇒
      w = w + 1;
    case Blue ⇒
      b = b - 1;
  }
```

- A free programming language that only compiles programs that can be verified
 - Generates C#, JS or Go
- Termed “auto-active program verifier”
 - Can verify your programs as you type them
- Can be used as a batch style compiler
 - But best used with an Editor
 - VS Code and Emacs supported

Obtaining Dafny

- Available for free for Windows, Linux and macOS
 - Even in Debian/Ubuntu repository
 - (though old version)
- I'm using the version from the Github repo

Simple program in Dafny



The screenshot shows a window titled "simple1.dfy" with a menu bar (File, Edit, Options, Buffers, Tools, YASnippet, Hide/Show, Help) and a toolbar. The code in the editor is as follows:

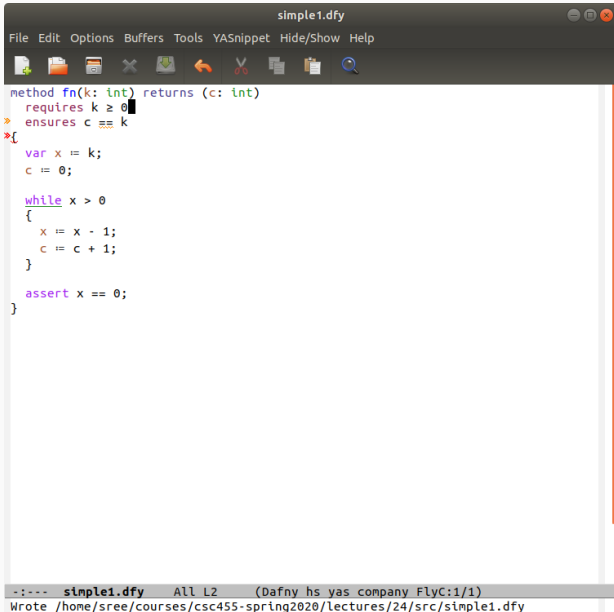
```
method fn(k: int) returns (c: int)
  > ensures c == k
  >{
    var x := k;
    c := 0;

    while x > 0
    {
      x := x - 1;
      c := c + 1;
    }

    > assert x == 0;
  }
```

At the bottom of the window, a status bar displays the error message: "-:--- simple1.dfy All L13 (Dafny hs yas company FlyC:2/1) assertion violation".

Adding requires



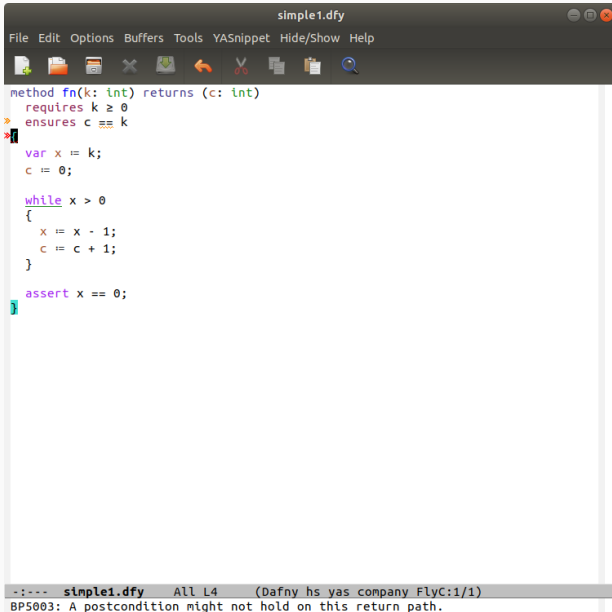
```
simple1.dfy
File Edit Options Buffers Tools YASnippet Hide/Show Help
method fn(k: int) returns (c: int)
  requires k ≥ 0
  ensures c == k
*{
  var x := k;
  c := 0;

  while x > 0
  {
    x := x - 1;
    c := c + 1;
  }

  assert x == 0;
}
```

--- simple1.dfy All L2 (Dafny hs yas company FlyC:1/1)
Wrote /home/sree/courses/csc455-spring2020/lectures/24/src/simple1.dfy

Postcondition might not hold



The screenshot shows a window titled "simple1.dfy" with a menu bar (File, Edit, Options, Buffers, Tools, YASnippet, Hide/Show, Help) and a toolbar. The code in the editor is as follows:

```
method fn(k: int) returns (c: int)
  requires k ≥ 0
  ensures c == k
{
  var x := k;
  c := 0;

  while x > 0
  {
    x := x - 1;
    c := c + 1;
  }

  assert x == 0;
}
```

The status bar at the bottom displays: `-:--- simple1.dfy All L4 (Dafny hs yas company FlyC:1/1)` and the error message: `BP5003: A postcondition might not hold on this return path.`

Add $x \geq 0$ invariant

```
simple1.dfy
File Edit Options Buffers Tools YASnippet Hide/Show Help
method fn(k: int) returns (c: int)
  requires k ≥ 0
  ensures c == k
{
  var x = k;
  c = 0;

  while x > 0
    invariant x ≥ 0
    {
      x = x - 1;
      c = c + 1;
    }

  assert x == 0;
}
```

--- simple1.dfy All L4 (Dafny hs yas company FlyC:1/1)
BP5003: A postcondition might not hold on this return path.

Add $c \leq k$ invariant

```
simple1.dfy
File Edit Options Buffers Tools YASnippet Hide/Show Help
method fn(k: int) returns (c: int)
  requires k ≥ 0
  > ensures c == k
  >{
    var x := k;
    c := 0;

    > while x > 0
    >   invariant c ≤ k
    {
      x := x - 1;
      c := c + 1;
    }

    assert x == 0;
  }
}

-:--- simple1.dfy All L9 (Dafny hs yas company FlyC:2/1)
BP5005: This loop invariant might not be maintained by the loop.
```


Add $c \leq k$ invariant, contd

```
simple1.dfy
File Edit Options Buffers Tools YASnippet Hide/Show Help
method fn(k: int) returns (c: int)
  requires k ≥ 0
  ensures c == k
{
  var x := k;
  c := 0;

  while c < k
    invariant c ≤ k
    {
      x := x - 1;
      c := c + 1;
    }
  > assert x == 0;
}
```

```
:-**- simple1.dfy All L15 (Dafny hs yas company FlyC:1/0)
assertion violation
```

Combining invariants

```
simple1.dfy
File Edit Options Buffers Tools YASnippet Hide/Show Help
method fn(k: int) returns (c: int)
  requires k ≥ 0
  > ensures c == k
  >{
    var x := k;
    c := 0;

    while x > 0 ∧ c < k
      invariant x ≥ 0 ∧ c ≤ k
      {
        x := x - 1;
        c := c + 1;
      }
  }
  > assert x == 0;
}
```

-.**-* simple1.dfy All L15 (Dafny hs yas company FlyC:2/1)
assertion violation

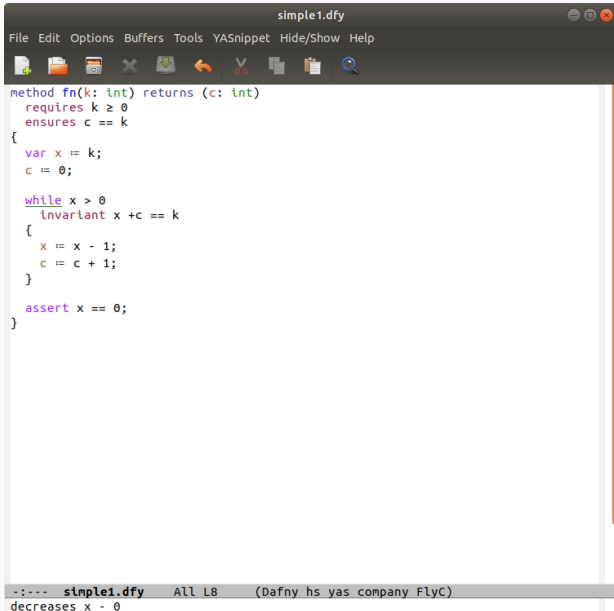
Combining invariants - Part #2

```
simple1.dfy
File Edit Options Buffers Tools YASnippet Hide/Show Help
method fn(k: int) returns (c: int)
  requires k ≥ 0
  > ensures c == k
  x|
  var x = k;
  c := 0;

  while x > 0 ∧ c < k
    invariant x ≥ 0 ∧ c ≤ k
    {
      x = x - 1;
      c := c + 1;
    }
  > assert x == 0;
  |

-:***- simple1.dfy All L4 (Dafny hs yas company FlyC:2/1)
BP5003: A postcondition might not hold on this return path.
```

Final Invariant



The screenshot shows a window titled "simple1.dfy" with a menu bar (File, Edit, Options, Buffers, Tools, YASnippet, Hide/Show, Help) and a toolbar. The code in the editor is as follows:

```
method fn(k: int) returns (c: int)
  requires k ≥ 0
  ensures c == k
{
  var x := k;
  c := 0;

  while x > 0
    invariant x + c == k
  {
    x := x - 1;
    c := c + 1;
  }

  assert x == 0;
}
```

At the bottom of the window, a status bar displays: `-- simple1.dfy All L8 (Dafny hs yas company FlyC)` and `decreases x - 0`.

Non-terminating loop

```
simple1.dfy
File Edit Options Buffers Tools YASnippet Hide/Show Help
method fn(k: int) returns (c: int)
  requires k ≥ 0
  ensures c == k
{
  var x := k;
  c := 0;
  > while x > 0 ∨ c < k
    invariant x ≥ 0 ∧ c ≤ k
    {
      x := x - 1;
      c := c + 1;
    }
  assert x == 0;
}

-:***- simple1.dfy All L8 (Dafny hs yas company FlyC:1/0)
decreases
cannot prove termination; try supplying a decreases clause for the loop
```

Outline

Proofs of Program Correctness

Loop Invariants

Theorem Proving

Postscript

Further Resources

- We focused entirely on loop invariants today
 - Their utility and ability to model entire loop executions
 - Their use in proving properties
 - Furia, Meyer, Velder, Loop Invariants: analysis, classification, and examples, ACM Computing Surveys
- Introduced you to Dafny
 - The Dafny Project at Microsoft Research
 - More reading (including 4-part video lectures)
- Next week: Hoare Logic
 - Source of the assignment axiom, and other rules for deriving program facts
 - Strongly recommend reading Background Reading on Hoare Logic, by Mike Gordon

Homework

- Let $popcount(x)$ be the number of bits set to 1 in x
- Show that $popcount(x) - popcount(x \& (x - 1)) = 1$
 - (where $\&$ is bitwise and)
- Example:
 - 5 is 0b101, 4 is 0b100, $5 \& 4 = 0b100 = 4$