Computing x^p: Reasoning with a Loop Invariant Leads to Discovery of an Efficient Implementation

Bruce W. Weide
Computer Science and Engineering
Ohio State University




Overview

- Example suggests a variety of in-class activities
  - Tried so far only in our CS1 course, but you might use it in a Discrete Math course
- Today's example: computing a ***product*** rather than a ***power***
  - These problems have completely parallel structure and illustrate the same points

Background

- A “components-first” approach is used in our CS1 and CS2
  - Language is Java
  - Design-by-contract with ***formal*** specifications

Novelty

- You can now demo the application of web-based public ***tools*** that can leverage ***formal*** specifications to reason automatically about software behavior
  - Dafny (used in this presentation)
  - RESOLVE (used in the next presentation)

`NaturalNumber` Components

- Our CS1 introduces a `NaturalNumber` component family with which students practice applying some key concepts:
  - Design-by-contract
  - Recursion (that's not "in your face")
  - Interval-halving/bisection/binary search

`NaturalNumber` Components

- Start with three simple methods:

```
void multiplyBy10(int d)
int divideBy10()
boolean isZero()
```

- Write code, layer by layer, to do various arithmetic calculations
  - Add
  - Power
  - Root

Reasoning About Method Calls

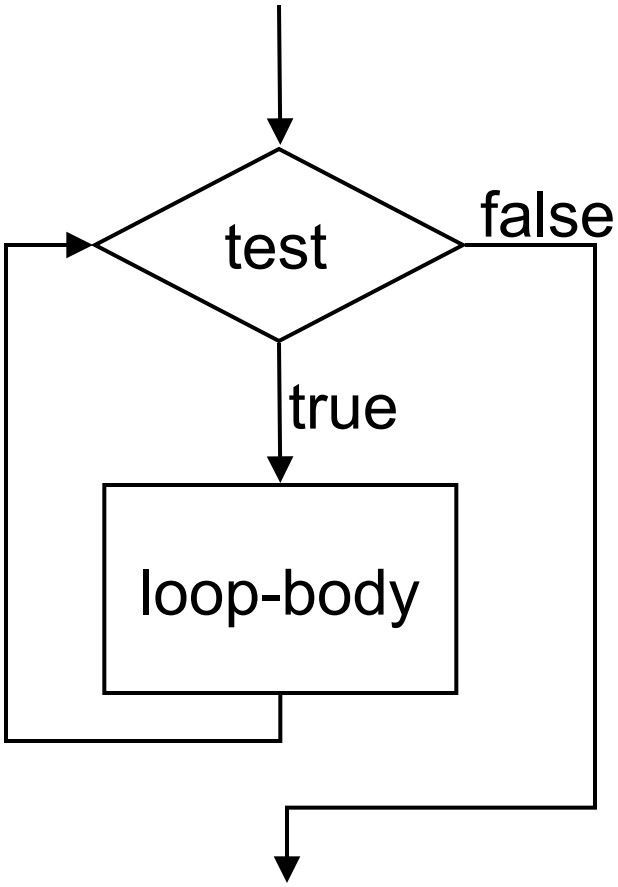
- What a method call does is described by its ***contract***
  - ***Precondition***: a property that is true *before* a call is made (client's obligation)
  - ***Postcondition***: a property that is true *after* a call returns (implementer's obligation)

Reasoning About Loops

- What a **`while`** loop does is described by its ***loop invariant***
  - ***Invariant***: a property that is true *every time* some code reaches a certain point—in the case of a loop invariant, at the loop condition test
  - ***Progress metric***: a non-negative integer-valued expression that decreases every time through the loop body

`while` Statement Control Flow

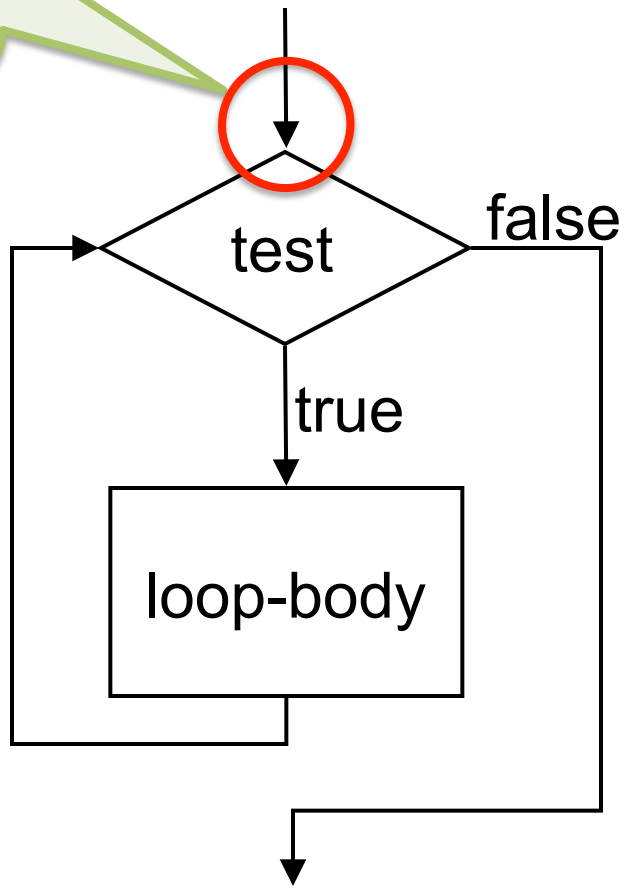
```
while (test) {
  loop-body
}
```



Control Flow

The loop invariant is a property that is true here, just before the loop begins...

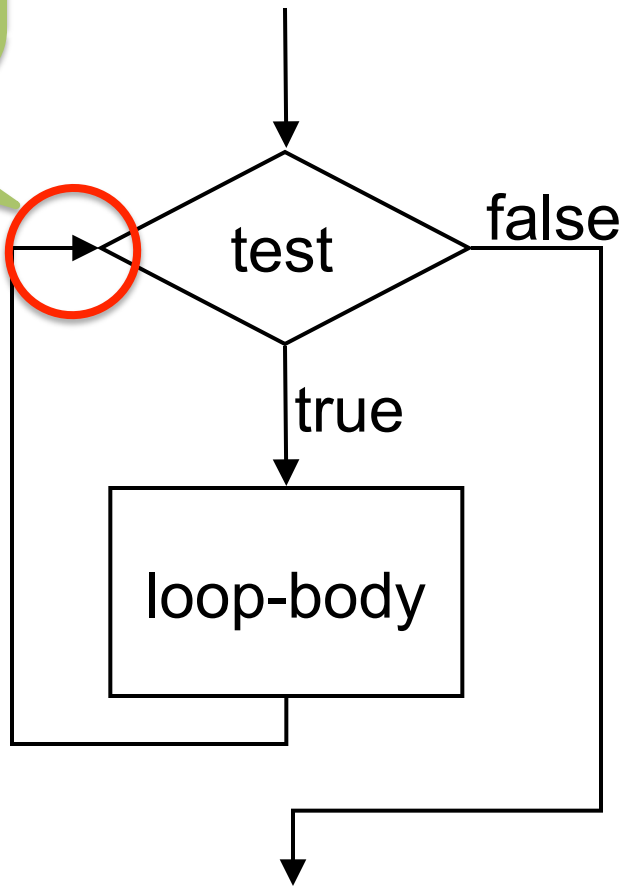
```
while (test) {
  loop-body
}
```



Control Flow

... and is true here, just after *every* execution of the loop body …

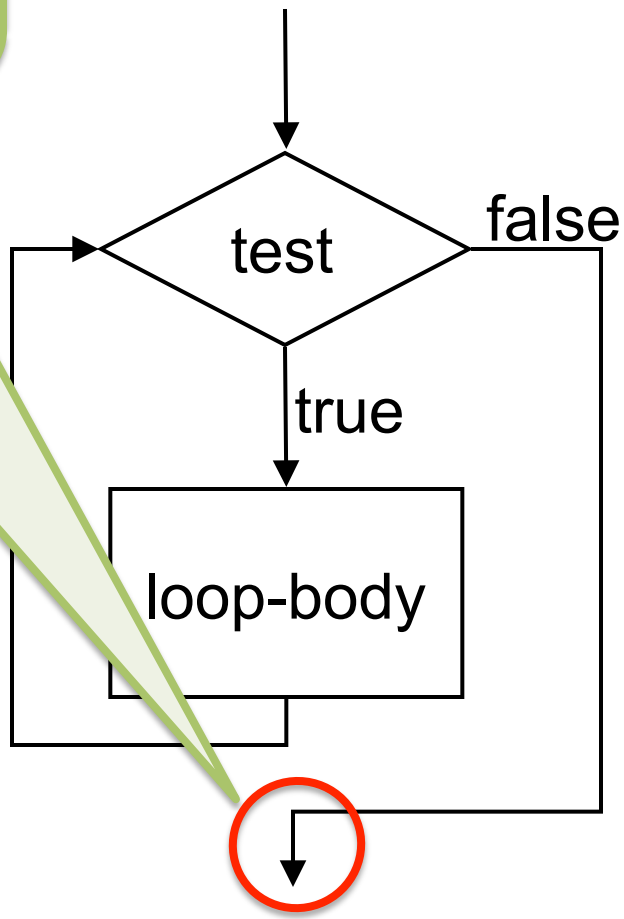
```
while (test) {
  loop-body
}
```



Control Flow

... hence is true here (assuming the test does not change values of variables).

```
while (test) {
  loop-body
}
```



http://rise4fun.com/Dafny

dafny

Microsoft Research

Is this program correct?

```
1 method Multiply(x: int, y: int) returns (product: int)
2   requires x >= 0 && y >= 0;
3   ensures product == x * y;
4 {
5
6 }
7
```

http://rise4fun.com/Dafny

Microsoft Research

dafny

Is this program correct?

```
1 method Multiply(x: int, y: int) returns (product: int)
2   requires x >= 0 && y >= 0;
3   ensures product == x * y;
4 {
5
6 }
7
```

Dafny's **`int`** is considered unbounded, just like our `NaturalNumber`.

http://rise4

dafny

Ask Dafny to verify the program, and it says:
postcondition might not hold.
Why not?

Is this program correct?

```
1 method Multiply(x: int, y: int) returns (product: int)
2   requires x >= 0 && y >= 0;
3   ensures product == x * y;
4 {
5
6 }
7
```

Provide a Method Body

```
 5   var xIncr: int := x;
 6   var yLeft: int := y;
 7   product := 0;
 8   while (yLeft != 0)
 9   {
10     product := product + xIncr;
11     yLeft := yLeft - 1;
12   }
```

Provide a

```
 5   var xIncr:
 6   var yLeft:  int := y;
 7   product := 0;
 8   while (yLeft != 0)
 9   {
10     product := product + xIncr;
11     yLeft := yLeft - 1;
12   }
```

Note: A previous homework was to write the code for this for `NaturalNumber`; this is a direct translation to Dafny.

Provide a

Ask Dafny to verify the program, and it says:
postcondition might not hold.
Why not?

```
 5   var xIncr:
 6   var yLeft:
 7   product := 0;
 8   while (yLeft != 0)
 9   {
10     product := product + xIncr;
11     yLeft := yLeft - 1;
12   }
```

Activity: Given the Invariant, Prove It Works

```
 5   var xIncr: int := x;
 6   var yLeft: int := y;
 7   product := 0;
 8   while (yLeft != 0)
 9     invariant product + xIncr * yLeft == x * y;
10     decreases yLeft;
11   {
12
13   }
```

Activity: Given the Invariant, Prove It Works

```
5   var xIncr: int := x;
6   var yLeft: int := y;
7   product := 0;
8   while (yLeft != 0)
9     invariant product + xIncr * yLeft == x * y;
10    decreases yLeft;
11  {
12
13  }
```

This equality is ***invariant*** …

Activity: Given the Invariant, Prove It Works

```
 5   var xIncr: int := x;
 6   var yLeft: int := y;
 7   product := 0;
 8   while (yLeft != 0)
 9     invariant product + xIncr * yLeft == x * y;
10     decreases yLeft;
11   {
12
13   }
```

… so this expression is ***invariant*** (since `x` and `y` do not change).

Activity: Giv
Prov

Ask Dafny to verify the program, and it says:
decreases expression might not decrease.
Why not?

```
5   var xIncr: int :
6   var yLeft: int := y;
7   product := 0;
8   while (yLeft != 0)
9     invariant product + xIncr * yLeft == x * y;
10    decreases yLeft;
11  {
12
13  }
```

Activity: Write the Loop Body

```
 5   var xIncr: int := x;
 6   var yLeft: int := y;
 7   product := 0;
 8   while (yLeft != 0)
 9     invariant product + xIncr * yLeft == x * y;
10     decreases yLeft;
11   {
12     product := product + xIncr;
13     yLeft := yLeft - 1;
14   }
```

Activity: Wri

Ask Dafny to verify the program, and it says:
Dafny program verifier finished with 2 verified, 0 errors.

```
 5   var xIncr: int :
 6   var yLeft: int :
 7   product := 0;
 8   while (yLeft != 0)
 9     invariant product + xIncr * yLeft == x * y;
10     decreases yLeft;
11   {
12     product := product + xIncr;
13     yLeft := yLeft - 1;
14   }
```

What's New?

- If you discuss loop invariants, it's just like any other example you might do in class
  - An example we have done for many years
  - Except now, there's this tool that automatically verifies the code is correct…

Next Question

- Is there any way to “decrease yLeft” other than by decrementing it?

```
 5   var xIncr: int := x;
 6   var yLeft: int := y;
 7   product := 0;
 8   while (yLeft != 0)
 9     invariant product + xIncr * yLeft == x * y;
10     decreases yLeft;
11   {
12
13   }
```

Next

Some student:
"You could divide it by 2."

- Is there any way to "d ase yLeft" other than by decrementi :

```
 5   var xIncr: int :=
 6   var yLeft: int  ;
 7   product := 0;
 8   while (yLeft  = 0)
 9     invariant product + xIncr * yLeft == x * y;
10     decreases yLeft;
11   {
12
13   }
```

Activity: Finish the Loop Body

```
 5   var xIncr: int := x;
 6   var yLeft: int := y;
 7   product := 0;
 8   while (yLeft != 0)
 9     invariant product + xIncr * yLeft == x * y;
10     decreases yLeft;
11   {
12
13     yLeft := yLeft / 2;
14   }
```

Activity: Finisl

```
 5   var xIncr: int :=
 6   var yLeft: int :=
 7   product := 0;
 8   while (yLeft != 0)
 9     invariant product +     cr * yLeft == x * y;
10     decreases yLeft;
11   {
12
13     yLeft := yLeft / 2;
14   }
```

Note: A previous homework was to implement a method to divide a `NaturalNumber` by 2; so, you really are allowed to do this!

Activity: Finish th

The rest is tough…
Hint: Try the case where `yLeft` is even.

```
5   var xIncr: int := x;
6   var yLeft: int := y;
7   product := 0;
8   while (yLeft != 0)
9     invariant prod + xIncr * yLeft == x * y;
10    decreases y eft;
11  {
12
13    yLeft := yLeft / 2;
14  }
```

`assert` Statements

```
 8   while (yLeft != 0)
 9     invariant product + xIncr * yLeft == x * y;
10     decreases yLeft;
11   {
12     if (yLeft % 2 == 0) {
13       assert product + (xIncr * 2) * (yLeft / 2) ==
14              product + xIncr * yLeft;
15       xIncr := xIncr + xIncr;
16     } else {
17
18     }
19     yLeft := yLeft / 2;
20   }
```

`assert` Sta

```
 8   while (yLeft != 0)
 9     invariant product +
10     decreases yLeft;
11   {
12     if (yLeft % 2 == 0) {
13       assert product + (xIncr * 2) * (yLeft / 2) ==
14              product + xIncr * yLeft;
15       xIncr := xIncr + xIncr;
16     } else {
17
18     }
19     yLeft := yLeft / 2;
20   }
```

This uses Dafny as a symbolic algebra system to check that your claim is correct; not *needed* in the code.

Completed Activity

```
 8  while (yLeft != 0)
 9    invariant product + xIncr * yLeft == x * y;
10    decreases yLeft;
11  {
12    if (yLeft % 2 != 0) {
13      product := product + xIncr;
14    }
15    xIncr := xIncr + xIncr;
16    yLeft := yLeft / 2;
17  }
```

Compl

Ask Dafny to verify the program, and it says:
Dafny program verifier finished with 2 verified, 0 errors.

```
 8   while (yLeft !=
 9     invariant product + xIncr * yLeft == x * y;
10     decreases yLeft;
11   {
12     if (yLeft % 2 != 0) {
13       product := product + xIncr;
14     }
15     xIncr := xIncr + xIncr;
16     yLeft := yLeft / 2;
17   }
```

Observation

- This is *essentially* the multiplication algorithm students (used to?) learn in elementary school
  - Most people, not just students, cannot recognize it…

Dafny “Permalinks”

http://rise4fun.com/Dafny/ycFB	(Multiply specification)
http://rise4fun.com/Dafny/sL0Q4	(slow, loop body only)
http://rise4fun.com/Dafny/Nm7i	(slow, invariant only)
http://rise4fun.com/Dafny/632H	(slow, completed)
http://rise4fun.com/Dafny/K1US	(fast, yLeft only)
http://rise4fun.com/Dafny/pEcl	(fast, even case)
http://rise4fun.com/Dafny/RKAH	(fast, both cases)
http://rise4fun.com/Dafny/hhJ3	(fast, completed)

e-mail: weide.1@osu.edu

web: http://cse.osu.edu/rsrg