CS 326
Formal Reasoning: Loops

Stephen Freund

1

Program Verification

- Weakest Preconditions
  - most permissive assumptions to ensure postcondition is satisfied.
- Verifying functional correctness

```
// requires P
// ensures Q
method test(...) {
  S
}
```

Prove:
P => wp(S, Q)

Example in Dafny

```
method Test(x : int, y : int) returns (c :Point?)
  requires P;
  ensures c != null;
{
  c := null;
  var z;
  if y < 0 {
    z := -2 * y;
  } else {
    z := x;
  }
  if z > 10 {
    c := new Point(z,y);
  }
}
```

Prove: P => wp(..., c != null)

Loops

```
{ n ≥ 0 }
i = 0;
y = 0;
{ P: n ≥ 0 ∧ i = 0 ∧ y = 0 }
while(i != n) {
  i = i+1;
  y = y+i;
}
```

Loops

{ n ≥ 0 }
```
i = 0;
y = 0;
```
{ P: n ≥ 0 ∧ i = 0 ∧ y = 0 }
{ *invariant* I : … }
```
while(i != n) {
  i = i+1;
  y = y+i;
}
```
{ Q: y = sum(1,n) }

Loops in Hoare Logic

{ n ≥ 0 }
```
i = 0;
y = 0;
```
{ P: n ≥ 0 ∧ i = 0 ∧ y = 0 }
{ invariant: y = sum(1,i) }
```
while(i != n) {

  i = i+1;

  y = y+i;

}
```
{ Q: y = sum(1,n) }

```
{P}
while(B) {
  S
}
{Q}
```

Pick invariant I such that

1. P => I
2. {I ∧ B}S{I}
3. (I ∧ !B) => Q

Version 2

{ n ≥ 0 }
```
i = 1;
y = 0;
```
{ P: n ≥ 0 ∧ <u>i = 1</u> ∧ y = 0 }
{ invariant: y = sum(1,i) }
```
while(i != n) {

  i = i+1;

  y = y+i;

}
```
{ Q: y = sum(1,n) }

```
{P}
while(B) {
  S
}
{Q}
```

Pick invariant I such that

1. P => I
2. {I ∧ B}S{I}
3. (I ∧ !B) => Q

Version 2

{ n ≥ 0 }
```
i = 1;
y = 0;
```
{ P: n ≥ 0 ∧ <u>i = 1</u> ∧ y = 0 }
{ invariant: y = sum(1,<u>i-1</u>) }
```
while(i != n) {

  i = i+1;

  y = y+i;

}
```
{ Q: y = sum(1,n) }

```
{P}
while(B) {
  S
}
{Q}
```

Pick invariant I such that

1. P => I
2. {I ∧ B}S{I}
3. (I ∧ !B) => Q

Version 3

{ n ≥ 0 }
```
i = 1;
y = 0;
```
{ P: n ≥ 0 ∧ i = 1 ∧ y = 0 }
{ invariant: y = sum(1,i-1) }
```
while(i != n + 1) {
```
{ y = sum(1,i-1) ∧ i != n + 1 }
```
  i = i+1;
```
{ y = sum(1,i-2) }
```
  y = y+i;
```
{ y = sum(1,i-2) + i = sum(1,i-1) }
```
}
```
{ y = sum(1,i-1) ∧ i = n + 1 } =>
{ Q: y = sum(1,n) }

```
{P}
while(B) {
  S
}
{Q}
```

Pick invariant I such that

1. P => I
2. {I ∧ B}S{I}
3. (I ∧ !B) => Q

This line is wrong!
To fix, we need to flip the two statements in the loop – see Version 4 next.

Version 4: Self Check

{ n ≥ 0 }
```
i = 1;
y = 0;
```
{ P: n ≥ 0 ∧ i = 1 ∧ y = 0 }
{ invariant: y = sum(1,i-1) }
```
while(i != n + 1) {

  y = y+i;

  i = i+1;

}
```
{ Q: y = sum(1,n) }

```
{P}
while(B) {
  S
}
{Q}
```

Pick invariant I such that

1. P => I
2. {I ∧ B}S{I}
3. (I ∧ !B) => Q

Too Strong? Too Weak? Just Right?

- Loop invariant is too *strong*:
  - may not hold on entry.
  - may not be preserved by body
- Loop invariant is too *weak*:
  - can't prove what you want after the loop
- No automatic procedure for conjuring a loop-invariant...
  - Think about invariant while writing the code
  - If proof doesn't work, invariant or code or both may need work

Methodology

1. Decide on the invariant first
   - What describes the milestone of each iteration?
2. Write a loop body to maintain the invariant
3. Write the loop test so "false implies postcondition"
4. Write initialization code to establish invariant

Methodology

Set **max** to hold the largest value in array **items**

1. Decide loop invariant first:

Example

Set **max** to hold the largest value in array **items**

2. Write a loop body to maintain the invariant

```
// I: max holds largest value in items[0..k-1]
while(   ) {
   // I holds
   if(max < items[k]) {
      max = items[k];  // breaks I temporarily
   }
   // max holds largest value in items[0..k]
   k = k+1;  // I holds again
}
```

Example

Set **max** to hold the largest value in array **items**

3. Write the loop test so false-implies-postcondition

```
// I: max holds largest value in items[0..k-1]
while(k != items.count) {
   // I holds
   if(max < items[k]) {
      max = items[k];  // breaks I temporarily
   }
   // max holds largest value in items[0..k]
   k = k+1;  // I holds again
}
// max holds largest value in items[0..items.count-1]
```

Example

Set **max** to hold the largest value in array **items**

4. Write initialization code to establish invariant

```
// I: max holds largest value in items[0..k-1]
while(k != items.count) {
   // I holds
   if(max < items[k]) {
      max = items[k];  // breaks I temporarily
   }
   // max holds largest value in items[0..k]
   k = k+1;  // I holds again
}
// max holds largest value in items[0..items.count-1]
```

Example

Edge case!

```
k = 1; max = items[0];
 // I: max holds largest value in items[0..k-1]
while(k != items.count) {
   // I holds
   if(max < items[k]) {
      max = items[k];   // breaks I temporarily
   }
   // max holds largest value in items[0..k]
   k = k+1;  // I holds again
}
// max holds largest value in items[0..items.count-1]
```

Quotient and Remainder

Set `q` to be the quotient of `x/y` and `r` to be the remainder

{Pre: $x > 0 \wedge y > 0$}

{Invariant }

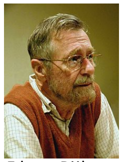
```
while (       ) {

}
```

{Post: $x = q*y + r \wedge 0 \leq r < y$ }

Dutch National Flag (classic)

Given an array of red, white, and blue pebbles, sort the array so the red pebbles are at the front, white are in the middle, and blue are at the end

- [Can only swap pebbles, not count them...]



Edsgar Dijkstra

Pre- and Post-conditions

Precondition: Any mix of red, white, and blue

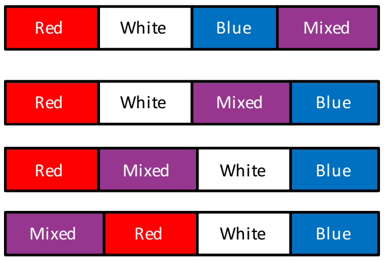
Mixed colors: red, white, blue

Postcondition:

- Red, then white, then blue
- Number of each color same as in original array

Red	White	Blue

Some Potential Invariants



More Precise

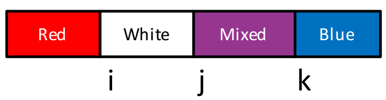
- Precondition: **`a`** contains **`red,white,blue`**
- Postcondition:

```
   0 <= i <= j < a.count
 ∧ a[0..i-1] is red
 ∧ a[i..j-1] is white
 ∧ a[j..a.count-1] is blue
```

- Invariant:

```
   0 <= i <= j <= k <= a.count
 ∧ a[0..i-1] is red
 ∧ a[i..j-1] is white
 ∧ a[k..a.count-1] is blue
```

The Code




```
i = 0;
j = 0;
k = a.count;
while (j!=k) {
  if(a[j] == White) {
     j = j+1;
  } else if (a[j] == Blue) {
     swap(a,j,k-1);
     k = k-1;
  } else { // a[j] == Red
     swap(a,i,j)
     i = i+1;
     j = j+1;
  }
}
```

Termination

- **Quotient-and-remainder**
  - **`r`** (starts positive, gets strictly smaller)
- **Binary search**
  - size of range still considered
- **Dutch-national-flag**
  - size of range not yet partitioned (**`k-j`**)
- **Search in a linked list**
  - length of list not yet considered

Recap

```
{ P }
Point c = null;
int z;
if (y < 0) {
  z = -2*y;
} else {
  z = x;
}
if (z > 10) {
  c = new Point(z,y);
}
{ Q: c != null }
```

From last time:

- weakest == most permissive
- strongest == most restrictive

What is the **weakest precondition** P that ensures postcondition Q?

When to use proofs for loops

- Overkill for "obvious" loops:
  - `for (name in friends) {…}`
- Use logical reasoning:
  - When intermediate state (invariant) is unclear or edge cases are tricky or you need inspiration, etc.
  - As an intellectual debugging tool
    - What *exactly* is the invariant?
    - Is it satisfied on every iteration?
    - Are you sure? Write code to check?
    - Did you check all the edge cases?
    - Are there preconditions you did not make explicit?

Termination

- Two kinds of loops
  - Those we want to always terminate (normal case)
  - Those that may conceptually run forever (e.g., web-server)
- So, proving a loop correct usually also requires proving termination
  - We haven't been proving this: might just preserve invariant forever without test ever becoming false
  - Our Hoare triples say ***if*** loop terminates, postcondition holds
- How to prove termination (variants exist):
  - Map state to a natural number somehow (just "in the proof")
  - Prove the natural number goes down on every iteration
  - Prove test is false by the time natural number gets to 0

Why Reason About Programs?

- Essential complement to testing
  - Testing shows specific result for a specific input
- Proof shows general result for all inputs
  - Can only prove correct code, proving uncovers bugs
  - Provides deeper understanding of why code is correct
- Precisely stating assumptions is essence of spec
  - “Callers must not pass null as an argument”
  - “Callee will always return an unaliased object”

Our Approach

- Hoare Logic, an approach developed in the 70’s
- Rarely use Hoare logic explicitly
  - often overkill for simple code
  - shines for developing code with subtle invariants
- Ideal for introducing program reasoning foundations
  - How does logic “talk about” program states?
  - How can program execution “change what’s true”?
  - What do “weaker” and “stronger” mean in logic?

Weakest Precondition

wp(x = e, Q)	Q[x := e]
wp(S1;S2, Q)	wp(S1,wp(S2,Q))
wp(if b S1 else S2, Q)	(b $\wedge$ wp(S1,Q)) $\vee$ (!b $\wedge$ wp(S2,Q))