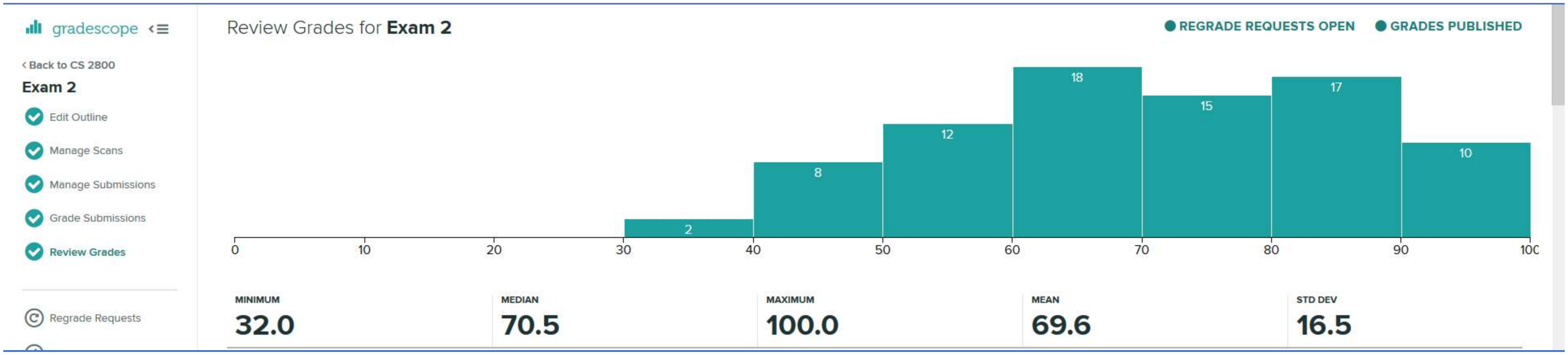
Logic and Computation – CS 2800
Fall 2019

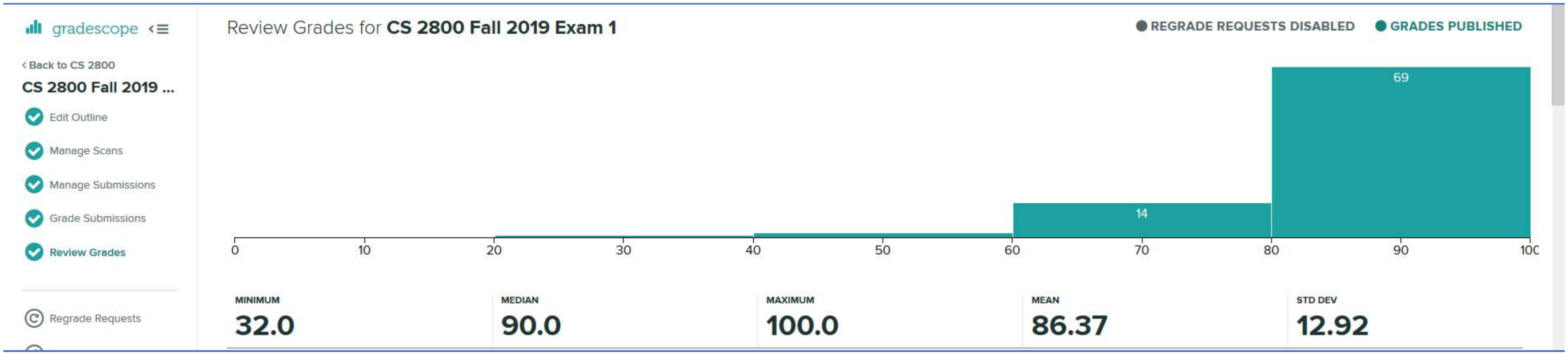
Lecture 34
Reasoning about imperative programs
The invariant game

Stavros Tripakis

Northeastern University
Khoury College of
Computer Sciences

Exam statistics





TRACE evaluations

- **Surveys are anonymous**
- Please respond to the survey!
- Surveys close on Dec 13

Homework 12

- This is an **INDIVIDUAL** (not group) homework
- Each student submits a separate answer

Outline

- Invariants
- Reasoning about imperative code
- The invariant game

Invariants

Invariants: reminder

- Consider this toy program:

```
k := 0 ; // assign 0 to k
```
what condition is true about k here?

```
// say “I love you” ten times:
while (k < 10) {
```
what about here?

```
  printf(“I love you\n”) ;
  k++ ;
```
and here?

```
}
```

Invariants: reminder

- What is an invariant?
  - *A property that is always satisfied in all executions of the program, at a certain location in the program.*
- E.g.:

```
k := 0 ; // assign 0 to k
// k=0 is an invariant here

// say "I love you" ten times:
while (k < 10) {
  // k<10 is an invariant here
  // 0<=k<10 is another (stronger) invariant
  printf("I love you\n") ;
  k++ ;
  // k<=10 is invariant here
  assert(k<=10);  // assertion statement
}
```

Invariants: notation

- We use `{Ix: cond}` to state that condition `cond` is invariant at a certain place in the program.
  - `Ix` is just a label for the invariant
- E.g.:

```
k := 0 ; // assign 0 to k
{I1: k=0}

// say "I love you" ten times:
while (k < 10) {
  {I2: k<10}
  {I3: 0<=k<10}
  printf("I love you\n") ;
  k++ ;
  {I4: k<=10}
}
```

Inductive invariants

Inductive invariants (also called loop invariants)

- A property `I` is an inductive invariant iff:
  1. `I` is an invariant.
  2. `I` is **inductive**: if `I` holds before the loop, then `I` will also hold after the loop.
- E.g.:

```
k := 0 ; // assign 0 to k

// say "I love you" ten times:
while {I: k>=0} (k < 10) {
  printf("I love you\n") ;
  k++ ;
}
```

Inductive invariants

- Is this an inductive invariant?

```
k := 0 ; // assign 0 to k

// say "I love you" ten times:
while {I: k>20} (k < 10) {
  printf("I love you\n") ;
  k++ ;
}
```

- No: it **is** inductive, but it is **not** an invariant.

Inductive invariants

- Is this an inductive invariant?

```
k := 0 ; // assign 0 to k

// say "I love you" ten times:
while {I: k<10} (k < 10) {
  printf("I love you\n") ;
  k++ ;
}
```

- No: if k=9 before the loop, it will be k=10 after the loop.

Inductive invariants

- Is this an inductive invariant?

```
k := 0 ; // assign 0 to k

// say "I love you" ten times:
while {I: k<=10} (k < 10) {
  printf("I love you\n") ;
  k++ ;
}
```

- Yes: if k=10 before the loop, then the loop is not entered, so k will still be 10 after the loop.

Inductive invariants

- Is this an inductive invariant?

```
k := 3 ;

while {I: k>0} (k < 10) {
  k := 3*k - 5 ;
}
```

- No:
  - It **is** an invariant – why?
  - It is **not** inductive – why?
    - Because if `k=1` before the loop, then `k=-2` after the loop.

Proving (or disproving) inductiveness

- We can use any technique that we learned!

```
k := 3 ;

while {I: k>0} (k < 10) {
  k := 3*k - 5 ;
}
```

- Proof obligation: `(integerp k) & k>0 => 3*k-5>0`
  - False
  - Counterexample: `k=1, k>0, 3*k-5=3-5=-2<0`

Proving (or disproving) inductiveness

- We can use any technique that we learned!

```
k := 3 ;

while {I: k>=3} (k < 10) {
  k := 3*k - 5 ;
}
```

- Proof obligation: `(integerp k) & k>=3 => 3*k-5>=3`
  - True
  - Can be shown using induction on natural numbers

Inductiveness is helpful for establishing invariants

- If we know that I is inductive, and we prove that I holds the first time we arrive at the loop, then we know that I is an invariant!

```
k := 3 ;

while {I: k>=3} (k < 10) {
  k := 3*k - 5 ;
}
```

- We don't have to explore all reachable states of the program! (they might even be infinite!)

Reasoning about imperative programs

Example

- To prove the guarantee G, we have to come up with an inductive invariant I such that
  `I & (cnt>=m) => G`
- (cnt>=m) is the negation of the loop condition (cnt < m).
- We include it because in order to reach G, we have to exit the loop.

```
main(n, m: nat)
{ var res, cnt: int;

  res := 0;
  cnt := 0;

  while ⟦I⟧ (cnt < m)
  { print(n, m, cnt, res);
    cnt  := cnt+1;
    res  := res+n;
    ⟦I⟧
  }
  print(n, m, cnt, res);
  ⟦G⟧
}
```

Example

Answer: `I: 0<=cnt<=m & res=cnt*n`

- To prove the guarantee `G`, we have to come up with an inductive invariant `I` such that
  `I & (cnt>=m) => G`
- (cnt>=m) is the negation of the loop condition (cnt < m).
- We include it because in order to reach `G`, we have to exit the loop.

```
main(n, m: nat)
{ var res, cnt: int;

  res := 0;
  cnt := 0;

  while ⟦I⟧ (cnt < m)
  { print(n, m, cnt, res);
    cnt  := cnt+1;
    res  := res+n;
    ⟦I⟧
  }
  print(n, m, cnt, res);
  ⟦G⟧
}
```

What should `I` be for this `G`?

The invariant game

- http://invgame.atwalter.com/

Next time

- Abstract data types and observational equivalence