Loop Invariants

CS242

Lecture 17

Approaches to Proving Properties of Programs

Automatic, Low complexity	Automatic, High complexity	Automatic or Semi-automatic Often undecidable	Manual, Undecidable
Simply Typed Lambda Calculus	Static Analysis	Invariant Inference	Dependent Types

Notation: Hoare Triples

{ Precondition } P { Postcondition}

- Precondition and Postcondition are statements in logic
  - Over program variables

- P is a program

- Read: If the precondition holds on entry to P, then the postcondition holds on exit from P

Examples

$\{ x > 0 \}\ x := x + 1\ \{ x > 1 \}$

$\{ \text{true} \}\ \text{if } x \text{ then } y := 1 \text{ else } y := 0\ \{ y = 0 \lor y = 1 \}$

$\{ x = 1 \}\ \ \text{for } i = 1,k \text{ do } x := x * k \quad \{ x = k^k \}$

A Simple Example

```
X = 0
I = 0
while I < 10 do
        X = X + 1
        I = I + 1

assert(X == 10)
```

Loop Invariants

- To verify loops, it suffices to find a sufficiently strong *loop invariant*

- What is a loop invariant?
  - A predicate that holds on every loop iteration
  - (at the same point, usually at the loop head)

- What is "sufficiently strong"?
  - More in a minute …

Loop Invariant (1)

```
X = 0
I = 0
while I < 10 do
        { true }
        X = X + 1
        I = I + 1

assert(X == 10)
```

Loop Invariant (2)

```
Z = 42
X = 0
I = 0
while I < 10 do
        { Z = 42 }
        X = X + 1
        I = I + 1

assert(X == 10)
```

Loop Invariant (3)

```
Z = 42
X = 0
I = 0
while I < 10 do
        { I < 4327 }
        X = X + 1
        I = I + 1

assert(X == 10)
```

Loop Invariant (4)

```
Z = 42
X = 0
I = 0
while I < 10 do
        { X < 11 }
        X = X + 1
        I = I + 1

assert(X == 10)
```

Loop Invariant (5)

```
Z = 42
X = 0
I = 0
while I < 10 do
        { X = I && I < 11 }
        X = X + 1
        I = I + 1

assert(X == 10)
```

Comments

- Loop invariants aren't hard to compute
  - If you don't care about quality
  - true

- What we want is to prove the assertion at the end of the loop
  - Need an invariant strong enough to do this

Comments

- But how can we prove the assertion?

- We need a proof strategy
  - A process that we can apply to reason about *any* loop

Inductive Invariants

Pre

while (B)

I

{

… code …

Post

}

Pre → I

I ∧ B

{ code }

I

I ∧¬B → Post

Inductive Invariants

- $\text{Pre} \rightarrow \text{I}$

  The invariant holds initially

- $\text{I} \wedge \text{B} \{ \text{code} \} \text{I}$

  If the invariant and loop condition hold, executing the loop body re-establishes the invariant

- $\text{I} \wedge \neg\text{B} \rightarrow \text{Post}$

  If the invariant holds and the loop terminates, then the post-condition holds

Loop Invariant (1)

```
X = 0
I = 0
while I < 10 do
        { true }
        X = X + 1
        I = I + 1

assert(X == 10)
```

Loop Invariant (2)

```
Z = 42
X = 0
I = 0
while I < 10 do
        { Z = 42 }
        X = X + 1
        I = I + 1

assert(X == 10)
```

Loop Invariant (3)

```
Z = 42
X = 0
I = 0
while I < 10 do
        { I < 4327 }
        X = X + 1
        I = I + 1

assert(X == 10)
```

Loop Invariant (4)

```
Z = 42
X = 0
I = 0
while I < 10 do
        { X < 11 }
        X = X + 1
        I = I + 1

assert(X == 10)
```

Loop Invariant (5)

```
Z = 42
X = 0
I = 0
while I < 10 do
        { X = I && I < 11 }
        X = X + 1
        I = I + 1

assert(X == 10)
```

A More Realistic Example

```
int A[10];
i = 1
// i = 1
while i < 11 {
      // ∀1 ≤ j < i.  A[j] = 0
      A[i] = 0;
      i += 1
}
// ∀1 ≤ j ≤ 10.  A[j] = 0
```

Three conditions:

$i = 1 \rightarrow \forall 1 \leq j < i.\ A[j] = 0$

$\forall 1 \leq j < i.\ A[j] = 0$
$\{ A[i] = 0; i = i + 1\}$
$\forall 1 \leq j < i.\ A[j] = 0$

$((\forall 1 \leq j < i.\ A[j] = 0) \wedge i \geq 11) \rightarrow$
$\forall 1 \leq j \leq 10.\ A[j] = 0$

First Question

- How do we decide whether these formulas are true?

$\text{Pre} \rightarrow I \qquad I \wedge B \{ \text{code} \} I \qquad I \wedge \neg B \rightarrow \text{Post}$

- Use SMT solvers
  - Satisfiability Modulo Theories
  - Tools that include decision procedures for a wide variety of logical theories relevant to program verification
  - Boolean satisfiability, theory or arrays, bitvectors, integers, ...
- Simply give an SMT a formula and it may
  - Report it is satisfiable (and give an assignment)
  - Report it is unsatisfiable (and give a counter example)
  - Report "I don't know"
  - Run forever

Second Question

Why focus on loop invariants?

First Answer

- Loop invariants are an important concept in everyday programming
- Why is my loop correct?
- You can break the problem into the three conditions stated above

Second Answer: Automated Verification

- Consider a loop-free program P
  - With conditionals
  - Memory references
  - Data structures
  - No function calls

- What is the computational complexity of verifying

  { Precondition } P { Postcondition}

Digression: Automated Reasoning

- Consider the statement X := Y + Z

- How can we reason *automatically* about this statement?
  - Without knowing what specific property we might want to focus on

- Answer
  - We need to encode the entire semantics of the statement
  - In a way that we can usefully query

Boolean Circuits

- Recall that, at bottom, computers are composed of boolean circuits

- These circuits can be represented directly in propositional logic

- For example, assume X, Y, and Z are 1-bit integers
  - $X := Y + Z$
  - $x_0 = y_0 \text{ xor } z_0$
  - $c_1 = y_0 \wedge z_0$

Boolean Circuits

- Assume X, Y, and Z are 2-bit integers
  - $X := Y + Z$

  - $x_0 = y_0 \text{ xor } z_0$
  - $c_1 = y_0 \wedge z_0$
  - $x_1 = y_1 \text{ xor } z_1 \text{ xor } c_1$
  - $c_2 = (y_1 \wedge z_1) \vee (y_1 \wedge c_1) \vee (z_1 \wedge c_1)$

- And so on for any bitwidth of X, Y and Z.

What Are the Queries

- Consider $X := Y + Z$

- We might want to ask whether this addition can overflow.

- The query then is $c_{64} = \text{true}$?

What Can Be Encoded as Boolean Formulas?

- Consider any loop-free, function-call free segment of code

- Consists only of a fixed set of operations working on a fixed set of memory locations
  - We can name every bit that is manipulated
  - And every operation can be represented as boolean operations on bits

Any such program can be encoded as a boolean formula and queried for its possible values.

Nuances …

- Sometimes these formulas might be huge.

- Consider X = Y * Z
  - Encoding multiplication results in a giant circuit

- SMT solvers use higher-level properties of the operations to avoid the worst-case encodings in most cases
  - But at bottom they use boolean representations and solvers

Loops

- Now consider the verification problem
  - Where P can have one loop
  - But still no function calls
- What is the computational complexity of verifying

{ Precondition } P { Postcondition}

Verification of Loops

- Verifying properties of loops is *the* hard problem
  - In any non-trivial loop, we can't name every bit that is manipulated
  - Because we don't know how many times the loop is executed
- Solve this, and everything else is much easier

Invariant Inference

- Find (infer) loop invariants automatically

- An old problem

- Many algorithms in the literature

- We will look at a simple approach

Invariant Inference

- Two ideas:
  1. Separate invariant inference from the rest of the verification problem

  2. Guess the invariant from executions

Why Use Data From Tests?

- Complementary to static reasoning

- "See through" hard analysis problems
  - functionality may be simpler than the code

- Possible to generate many, many tests

Outline

- Guess (many) invariants
    - Run the program
    - Discard candidate invariants that are falsified
    - Attempt to verify the remaining candidates

A Simple Program

```
s = 0;
y = 0;
while( * )
{
  print(s,y);
  s := s + 1;
  y := y + 1;
}
```

- Instrument loop head
- Collect the values of program variables on each iteration

Data Collection Example

```
s = 0;
y = 0;
while( * )
{
  print(s,y);
  s := s + 1;
  y := y + 1;
}
```

- Hypothesize
  - s = y
  - s = 2y
- Data

s	**y**
0	0

Data Collection Example

```
s = 0;
y = 0;
while( * )
{
   print(s,y);
   s := s + 1;
   y := y + 1;
}
```

- Hypothesize
  - s = y
  - ~~s = 2y~~
- Data

s	y
0	0
1	1

Data Collection Example

```
s = 0;
y = 0;
while( * )
{
  print(s,y);
  s := s + 1;
  y := y + 1;
}
```

- Hypothesize
  - s = y
  - ~~s = 2y~~
- Data

s	y
0	0
1	1
2	2
3	3

Another Approach

```
s = 0;
y = 0;
while( * )
{
  print(s,y);
  s := s + 1;
  y := y + 1;
}
```

- Data

s	y
0	0
1	1
2	2
3	3

Arbitrary Linear Invariant

$$as + by = 0$$

- Data

s	y
0	0
1	1
2	2
3	3

Observation

$$as + by = 0$$

s	y
0	0
1	1
2	2
3	3

$$\begin{array}{c} \textbf{w} \\ \begin{bmatrix} a \\ b \end{bmatrix} \end{array} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Observation

$$as + by = 0$$

$\{ w \mid Mw = 0 \}$

s	y
0	0
1	1
2	2
3	3

w
a
b

=

0
0

Observation

$$as + by = 0$$

NullSpace(M)

s	y
0	0
1	1
2	2
3	3

$$\begin{array}{c} \mathbf{w} \\ \begin{bmatrix} a \\ b \end{bmatrix} \end{array} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Linear Invariants

- Construct matrix M of observations of all program variables
- Compute NullSpace(M)
- All invariants are in the null space

Spurious "Invariants"

- All invariants are in the null space
  - But not all vectors in the null space are invariants

- Consider the matrix

s	y
0	0

- Need a check phase
  - Verify the candidate is in fact an invariant

An Algorithm

- Check candidate invariant
  - If an invariant, done
  - If not an invariant, get a *counterexample*
    - Counterexample can be guaranteed to satisfy all invariants
- Add new row to matrix
  - And repeat

Termination

- How many times can the solve & verify loop repeat?

- Each counterexample is linearly independent of previous entries in the matrix

- So at most N iterations
  - Where N is the number of columns
  - Upper bound on steps to reach a full rank matrix

Summary

- Superset of all linear invariants can be obtained by a standard matrix calculation
- Counter-example driven improvements to eliminate all but the true invariants
  - Guaranteed to terminate

What About Non-Linear Invariants?

```
s = 0;
y = 0;
while( * )
{
  print(s,y);
  s := s + y;
  y := y + 1;
}
```

Idea

- Collect data as before

- But add more columns to the matrix
  - For derived quantities
  - For example, y^2 and s^2

- How to limit the number of columns?
  - All monomials up to a chosen degree d

What About Non-Linear Invariants?

```
s = 0;
y = 0;
while( * )
{
  print(s,y);
  s := s + y;
  y := y + 1;
}
```

1	s	y	s^2	y^2	sy
1	0	0	0	0	0
1	1	1	1	1	1
1	3	2	9	4	6
1	6	3	36	9	18
1	10	4	100	16	40

Solve for the Null Space

$$a + bs + cy + ds^2 + ey^2 + fsy = 0$$

1	s	y	s^2	y^2	sy
1	0	0	0	0	0
1	1	1	1	1	1
1	3	2	9	4	6
1	6	3	36	9	18
1	10	4	100	16	40

w
a
b
c
d
e
f

=

0
0
0
0
0
0

Candidate invariant: $-2s + y + y^2 = 0$

Comments

- Same issues as before
  - Must check candidate is implied by precondition, is inductive, and implies the postcondition on termination
  - Termination of invariant inference guaranteed if the verifier can generate counterexamples
- Solvers do well as checkers!

Experiments

Name	#vars	deg	Data	#and	Guess time (sec)	Check time (sec)	Total time (sec)
Mul2	4	2	75	1	0.0007	0.010	0.0107
LCM/GCD	6	2	329	1	0.004	0.012	0.016
Div	6	2	343	3	0.454	0.134	0.588
Bezout	8	2	362	5	0.765	0.149	0.914
Factor	5	3	100	1	0.002	0.010	0.012
Prod	5	2	84	1	0.0007	0.011	0.0117
Petter	2	6	10	1	0.0003	0.012	0.0123
Dijkstra	6	2	362	1	0.003	0.015	0.018
Cubes	4	3	31	10	0.014	0.062	0.076
geoReihe1	3	2	25	1	0.0003	0.010	0.0103
geoReihe2	3	2	25	1	0.0004	0.017	0.0174
geoReihe3	4	3	125	1	0.001	0.010	0.011
potSumm1	2	1	5	1	0.0002	0.011	0.0112
potSumm2	2	2	5	1	0.0002	0.009	0.0092
potSumm3	2	3	5	1	0.0002	0.012	0.0122
potSumm4	2	4	10	1	0.0002	0.010	0.0102

Invariant Inference

- We saw an algorithm for algebraic invariants
  - Up to a given degree
- Guess and Check
  - Hard part is inference done by matrix solve
  - Check part done by standard SMT solver
  - Simple and fast
- In general we have to be concerned with more general invariants
  - Over data structures, disjunctions

Summary

- Loop invariants are an important concept in programming
  - Good to think about invariants for your code!
  - Even without a tool to check or infer invariants

- Automating loop invariant inference is challenging
  - Long-standing research problem
  - Use in practice is still limited