Loop Invariants
Approaches to Proving Properties of Programs

Automatic, Low complexity
Simply Typed Lambda Calculus

Automatic, High complexity
Static Analysis

Automatic or Semi-automatic Often undecidable
Invariant Inference

Manual, Undecidable
Dependent Types
Notation: Hoare Triples

\{ \text{Precondition} \} \ P \ \{ \text{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 \}

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

\{ x = 1 \} for i = 1,k do x := x * k \{ 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 …
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)

\[
\begin{align*}
Z &= 42 \\
X &= 0 \\
I &= 0 \\
\text{while } I < 10 \text{ do} \\
&\quad \{ X = I \land I < 11 \} \\
&\quad X = X + 1 \\
&\quad I = I + 1 \\
\text{assert}(X == 10)
\end{align*}
\]
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

while (B) {
    ... code ...
}

Pre

I

Post

Pre \rightarrow I

I \land B

\{ code \}

I

I \land \neg B \rightarrow Post
Inductive Invariants

• $\text{Pre} \rightarrow I$
  
The invariant holds initially

• $I \land B \{ \text{code} \} I$
  
If the invariant and loop condition hold, executing the loop body re-establishes the invariant

• $I \land \neg B \rightarrow \text{Post}$
  
If the invariant holds and the loop terminates, then the post-condition holds
Loop Invariant (1)

\[
\begin{align*}
X &= 0 \\
I &= 0 \\
\text{while } I < 10 \text{ do} & \\
\quad \{ \text{true} \} \\
\quad X &= X + 1 \\
\quad I &= I + 1 \\
\text{assert}(X == 10)
\end{align*}
\]
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
\[ \{ X = I \land I < 11 \} \]
\[ X = X + 1 \]
\[ I = I + 1 \]

assert(\( X == 10 \))
A More Realistic Example

```c
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:

- $i = 1 \Rightarrow \forall 1 \leq j < i. A[j] = 0$
- $\forall 1 \leq j < i. A[j] = 0$
  - $\{A[i] = 0; \ i = i + 1\}$
  - $\forall 1 \leq j < i. A[j] = 0$
- $((\forall 1 \leq j < i. A[j] = 0) \land \ i \geq 11) \Rightarrow \forall 1 \leq j \leq 10. A[j] = 0$
First Question

• How do we decide whether these formulas are true?

\[
\text{Pre} \rightarrow I \quad I \land B \{ \text{code} \} \quad I \land \neg B \rightarrow \text{Post}
\]

• 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

$$\{\text{Precondition}\} \ P \ \{\text{Postcondition}\}$$
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
  • $X := Y + Z$
  • $x_0 = y_0 \text{xor} z_0$
  • $c_1 = y_0 \land z_0$
Boolean Circuits

• Assume X, Y, and Z are 2-bit integers
  • X := Y + Z
    • $x_0 = y_0 \text{ xor } z_0$
    • $c_1 = y_0 \land z_0$
    • $x_1 = y_1 \text{ xor } z_1 \text{ xor } c_1$
    • $c_2 = (y_1 \land z_1) \lor (y_1 \land c_1) \lor (z_1 \land c_1)$

• 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 $X = Y \times Z$
  • 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

\[
\begin{align*}
s &= 0; \\
y &= 0; \\
\text{while( } & \text{ )} \\
\{ \\
& \quad \text{print}(s,y); \\
& \quad s := s + 1; \\
& \quad y := y + 1; \\
\}
\end{align*}
\]

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

\[
\begin{align*}
s &= 0; \\
y &= 0; \\
\text{while}(\ * \ ) & \\
\{ \\
    \text{print}(s,y); \\
    s := s + 1; \\
    y := y + 1; \\
\}
\end{align*}
\]

- Hypothesize
  - \( s = y \)
  - \( s = 2y \)
- Data

\[
\begin{array}{c|c}
  s & y \\
  \hline
  0 & 0 \\
\end{array}
\]
Data Collection Example

\[
\begin{align*}
  s &= 0; \\
  y &= 0; \\
  \text{while}(\ast) \\
  \{ \\
    \text{print}(s, y); \\
    s &:= s + 1; \\
    y &:= y + 1;
  \}
\end{align*}
\]

- Hypothesize
  - \(s = y\)
  - \(s = 2y\)

- Data

<table>
<thead>
<tr>
<th>s</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Data Collection Example

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

- Hypothesize
  - $s = y$
  - $s = 2y$

- Data

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>y</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Another Approach

\[
s = 0;
y = 0;
\text{while}(\ast)
\{
    \text{print}(s,y);
    s := s + 1;
    y := y + 1;
\}
\]

Data

<table>
<thead>
<tr>
<th>s</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Arbitrary Linear Invariant

\[ a s + b y = 0 \]

• Data

<table>
<thead>
<tr>
<th>s</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Observation

\[ ax + by = 0 \]

<table>
<thead>
<tr>
<th>s</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

\[ w = a = 0 \]
\[ w = b = 0 \]
Observation

\[ as + by = 0 \]

\[ \{ w \mid Mw = 0 \} \]

<table>
<thead>
<tr>
<th>s</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

\[ w = a = 0 \]

\[ w = b = 0 \]
Observation

\[ as + by = 0 \]

**NullSpace(M)**

<table>
<thead>
<tr>
<th>s</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c}
\text{w} \\
a \\
b \\
\end{array} \quad \begin{array}{c}
= \\
0 \\
0 \\
\end{array}
\]
Linear Invariants

• Construct matrix $M$ of observations of all program variables

• Compute $\text{NullSpace}(M)$

• 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

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

• 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;
}

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>s</th>
<th>y</th>
<th>s^2</th>
<th>y^2</th>
<th>sy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>3</td>
<td>36</td>
<td>9</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>4</td>
<td>100</td>
<td>16</td>
<td>40</td>
<td></td>
</tr>
</tbody>
</table>
Solve for the Null Space

\[ a + bs + cy + ds^2 + ey^2 + fsy = 0 \]

\[
\begin{array}{cccccc}
1 & s & y & s^2 & y^2 & sy \\
1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 \\
1 & 3 & 2 & 9 & 4 & 6 \\
1 & 6 & 3 & 36 & 9 & 18 \\
1 & 10 & 4 & 100 & 16 & 40 \\
\end{array}
\]

\[
\begin{array}{c}
w \\
a \\
b \\
c \\
d \\
e \\
f \\
\end{array}
\]

\[
\begin{array}{cccccc}
& & & & w & \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

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

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