Lecture 7. Loop Analysis

Wei Le

2014.10
Goals of Loop Analysis

- Loop trip count (loop iteration count):
  - Compiler optimization (loop unrolling, loop execution frequencies for feedback-directed optimizations)
  - Worst case execution time
  - Vulnerability detection (denial of service and side channel detection)
- Loop invariant: used for formal verification
- Loop induction variables
- Loop effect on the variables in the loop
- Precondition and postcondition
- Loop termination problems
Examples

```
int j = 9;
for(int i=0; i<10; i++)
    j--;
```

Loop invariant? Precondition? Postcondition?
Examples

Loop Invariant:
- $i + j = 9$
- $i \geq 0 && i < 10 \text{ or } j \leq 9 && j \geq 0$

Precondition: $j = 9, i = 0$
Postcondition: $i = 10, j = 0$

```c
int j = 9;
for(int i=0; i<10; i++)
    j--;
```
Loop Features

Syntactic:
- Single or multi-paths
- Data structure: integer, string, array, containers ...
- Library calls
- Environment: user interactive, networking
- Nested loop

Semantic:
- What a loop computes: test a membership, sort a list of numbers, calculate a mean, traverse a data structure ...
- Loop invariants
- Pre- and post-conditions
- The relations of output variables and input variables are linear

Runtime Characteristics:
- Cache misses
- Performance
Traditional Loop Analysis

Very small state space is covered

- Iterate once [Cadar, Dunbar, Engler 08] [Chipounov, Kuznetsov, Candea 12]
- Report unknown [Xie, Chou, Engler 03]
- Pattern matching [Saxena, Poosankam, McCamant, Song 00]
- ...

Loop Analysis Algorithms [1]

Goal: Loop effects on variables
Solution: Segmented Symbolic Analysis (dynamic)
Loop Analysis Algorithms [2]

Goal: Loop trip count (relate trip count with inputs)
Solution: Abstract Interpretation on X86 machine code (static)

Details:

\[
\omega \quad \text{for } (i=0, \text{ptr}=0; i < \text{uri}_\text{len}; i++, \text{ptr}++)
\]
\[\text{msgbuf}[\text{ptr}] = \text{URI}[i];\]
\[\text{msgbuf}[\text{ptr}++] = ', '.\]

At the end of the first iteration, \(\text{ptr}\) is incremented, and on the loop back edge the two abstract values are joined to give \(0\langle 0, 0, 0 \rangle \sqcup 1\langle 0, 0, 1 \rangle = TC_3\langle 0, 0, * \rangle\). When \(\text{ptr}\) is incremented again on the next iteration, its abstract value after the back edge will be \(1 + TC_3\langle 0, 0, * + 1 \rangle\), which again joins to \(TC_3\langle 0, 0, * \rangle \sqcup 1 + TC_3\langle 0, 0, * + 1 \rangle = TC_3\langle 0, 0, * \rangle\). The effect of the increments on lines 28 and 31 and loop 4 on lines 30–31 are analyzed in a similar way, giving a final abstract value for \(\text{ptr}\) of \(1 + TC_3 + TC_4\langle 0, 0, *, * \rangle\).

\[
[c_1] \circ [c_2] \rightarrow [c_1 \circ c_2] \text{ for any operator } \circ \\
[c_1 + a_1 \cdot TC_1] + [c_2 + a_2 \cdot TC_1] \rightarrow [(c_1 + c_2) + (a_1 + a_2) \cdot TC_1] \\
[c_1] \cdot [c_2 + a_2 \cdot TC_1] \rightarrow [(c_1 \cdot c_2) + (c_1 \cdot a_2) \cdot TC_1] \\
\top \circ E \rightarrow \top \\
E \circ \top \rightarrow \top \\
[c_1 + a_1 \cdot TC_1] \circ [c_2 + a_2 \cdot TC_1] \rightarrow \top \text{ otherwise} \\
raise(E\langle 0 \rangle) \rightarrow E\langle 1 \rangle \\
raise(E\langle * \rangle) \rightarrow E\langle * + 1 \rangle \\
[a\langle 0 \rangle] \sqcup [(a + b)\langle 1 \rangle] \rightarrow [a + b \cdot TC_1\langle * \rangle] \\
[a + b \cdot TC_1\langle * \rangle] \sqcup [(a + b) + b \cdot TC_1\langle * + 1 \rangle] \rightarrow [a + b \cdot TC_1\langle * \rangle]
\]
// assert x \geq 0 & y = 0
while (x \neq y) {
    y = y + 1;
}
// assert x = y

Does “x=y” hold after this loop?
Does this loop terminate?
1) Pre-assertion guarantees that x \geq y
2) Every time through loop
   x \geq y holds at the test – and if body is entered, x > y
   y is incremented by 1
   x is unchanged
   Therefore, y is closer to x (but x \geq y still holds)
3) Since there are only a finite number of integers between x and y, y will eventually equal x
4) Execution exits the loop as soon as x = y (but x \geq y still holds)
Understanding Loops by Induction

We just made an inductive argument
  Inducting over the number of iterations
Computation induction
  Show that conjecture holds if zero iterations
  Show that it holds after $n+1$ iterations
    (assuming that it holds after $n$ iterations)
Two things to prove
  Some property is preserved (known as “partial correctness”),
    if the code terminates
    Loop invariant is preserved by each iteration, if the iteration completes
  The loop completes (known as “termination”)
    The “decrementing function” is reduced by each iteration
    and cannot be reduced forever
How to Choose a Loop Invariant

\{ P \}
while (b) S;
\{ Q \}

Find an invariant, \( LI \), such that

1. \( P \Rightarrow LI \) // true initially
2. \( \{ LI \land b \} S \{ LI \} \) // true if the loop executes once
3. \( (LI \land \neg b) \Rightarrow Q \) // establishes the postcondition

It is sufficient to know that if loop terminates, \( Q \) will hold.

Finding the invariant is the key to reasoning about loops. Inductive assertions is a “complete method of proof”:

If a loop satisfies pre/post conditions, then there exists an invariant sufficient to prove it
An Example of Loop Invariant

```plaintext
// assert x ≥ 0 & y = 0
while (x != y) {
    y = y + 1;
}
// assert x = y
```

A suitable invariant:

\[ \text{LI} = x \geq y \]

1. \(x \geq 0 \quad \& \quad y = 0 \Rightarrow \text{LI}\) // true initially
2. \(\{ \text{LI} \quad \& \quad x \neq y \} \quad y = y+1; \{ \text{LI} \}\) // true if the loop executes once
3. \(\text{LI} \quad \& \quad \neg(x \neq y)) \Rightarrow x = y\) // establishes the postcondition
Total correctness via Well-Ordered Sets

Total correctness = partial correctness + termination
We have not established that the loop terminates
Suppose that the loop always reduces some variable’s value. Does the loop terminate if the variable is a
- Natural number?
- Integer?
- Non-negative real number?
- Boolean?
- ArrayList?
The loop terminates if the variable values are (a subset of) a well-ordered set
- Ordered set
- Every non-empty subset has least element
Decrementing Function

Decrementing function $D(X)$
Maps state (program variables) to some well-ordered set
Tip: always use the natural numbers
This greatly simplifies reasoning about termination

Consider: while (b) S;
We seek $D(X)$, where $X$ is the state, such that

1. An execution of the loop reduces the function’s value:
   \[ \{ L \land b \} S \{ D(X_{\text{post}}) < D(X_{\text{pre}}) \} \]

2. If the function’s value is minimal, the loop terminates:
   \[ (L \land D(X) = \text{minVal}) \Rightarrow \neg b \]
Proving Termination

// assert x ≥ 0 & y = 0
// Loop invariant: x ≥ y
// Loop decrements: (x-y)
while (x != y) {
    y = y + 1;
}
// assert x = y

Is this a good decrementing function?
1. Does the loop reduce the decrementing function’s value?
   // assert (y ≠ x); let d_{pre} = (x-y)
   y = y + 1;
   // assert (x_{post} - y_{post}) < d_{pre}
2. If the function has minimum value, does the loop exit?
   (x ≥ y & x - y = 0) ⇒ (x = y)
Choosing Loop Invariants

For straight-line code, the wp (weakest precondition) function gives us the appropriate property.

For loops, you have to **guess:**
   - The loop invariant
   - The decrementing function

Then, use reasoning techniques to prove the goal property.

If the proof doesn't work:
   - Maybe you chose a bad invariant or decrementing function
     Choose another and try again
   - Maybe the loop is incorrect
     Fix the code

**Automatically choosing loop invariants is a research topic**
Wei Le.
Segmented symbolic analysis.

Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song.
Loop-extended symbolic execution on binary programs.