CS510 Software Engineering

Program Representations

Asst. Prof. Mathias Payer

Department of Computer Science
Purdue University

TA: Scott A. Carr
Slides inspired by Xiangyu Zhang

http://nebelwelt.net/teaching/15-CS510-SE

Spring 2015

Why Program Representations?

- Original representations: source code, binary, test cases.
- Hard to analyze and bad fit for automatic reasoning.
- Software is translated (lossy or lossless) into certain representations to help certain analyses.

Table of Contents

Control-Flow Graph (CFG)

- The CFG is an abstract representation of a program that captures all possible flows through the program.
- A CFG is a graph that consists of basic blocks (nodes) and possible control-flow paths (edges).
- A basic block (BB) is a linear sequence of program statements with a single entry and exit. Control-flow cannot exit or halt at any point inside the basic block except at its exit point. Entry and exit nodes coincide if the basic block has only one statement.

Control-Flow Graph: Definition

Control-Flow Graph

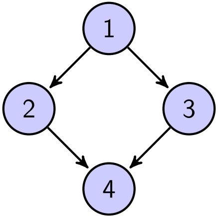
A control flow graph (or flow graph) G is defined as a finite set N of nodes and a finite set E of edges. An edge (i, j) in E connects two nodes n_i and n_j in N. We often write $G = (N, E)$ to denote a flow graph G with nodes given by N and edges by E.

Control-Flow Graph

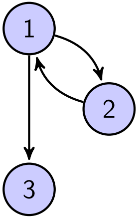
- In a CFG, each BB becomes a node and edges are used to indicate the flow of control between blocks.
- And edge (i, j) connecting blocks b_i and b_j implies that control may flow from block b_i to block b_j[1].
- The graph, by convention, also has a *start* node and an *end* node (also in N). The start node has no incoming edge while the end node has no outgoing edge.

[1] Note that the graph is directed.

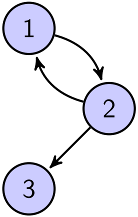
CFG by Example



if-else condition



for/while loop



do-while loop

Path

Path

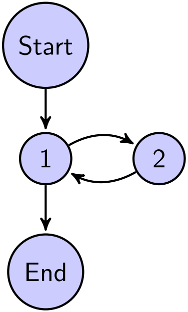
Consider a flow graph $G = (N, E)$. A sequence of k edges $k > 0$, (e_1, e_2, ..., e_k), denotes a path *through the flow graph if the following sequence condition holds:*
Given that n_p, n_q, n_r, n_s are nodes belonging to N, and $0 < i < k$, if $e_i := (n_p, n_q)$ and $e_{i+1} := (n_r, n_s)$ then $n_q \equiv n_r$.

A complete path is a path from start to end. A subpath is a subsequence of a complete path.

Feasible Paths

A path p through a flow graph for program P is considered *feasible* if there exists at least one test case which when input to P produces path p.

```
int func(int n) {
  int i, ret = n;
  for (i = n-1; i>=1; i--) {
    ret = ret * i;
  }
}
```



$p_1 = (Start, 1, 2, 1, End)$
$p_2 = (Start, 1, End)$
$p_{err} = (Start, 1, 2, End)$

Number of Paths

- A program may allow many distinct paths, depending on the conditions in the program. A program without conditions contains exactly one path from Start to End.
- Each condition in the program increments the number of paths by at least 1.
- Conditions can have a multiplicative effect on the number of paths.

Simplified CFG

- Each statement is represented by a node (and each basic block therefore contains only one statement which is the entry and exit statement).
- A simplified CFG is easy to read and implement but not efficient.
- A naive CFG construction algorithm starts with a simplified CFG and merges nodes n_i and n_{i+1} iff node n_i has one outgoing edge and node n_{i+1} has one incoming edge and edge $e := (n_i, n_{i+1})$.

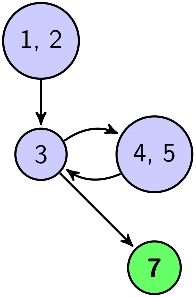
Dominator

Dominators

X `dominates` *Y, iff all possible paths from Start to Y pass through X.*
X `strictly dominates` *Y, iff X dominates Y and* $X != Y$.
X `immediately dominates` *Y, iff X dominates Y and X is the last dominator before Y on a path from Start to Y.*

Dominators: Example

```
1 int sum = 0;
2 int i = 1;
3 while (i<N) {
4   i += 1;
5   sum += i;
6 }
7 printf("Sum: %d", sum);
```



$sdom(7) = \{1, 2; 3\}$
$idom(7) = \{3\}$

Post-dominator

Post Dominators

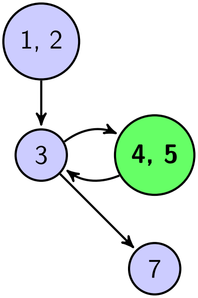
X `post-dominates` *Y, iff all possible paths from Y to End pass through X.*

X `strictly post-dominates` *Y, iff X post-dominates Y and* $X! = Y$.

X `immediately post-dominates` *Y, iff X post-dominates Y and X is the first post-dominator after Y on a path from Y to End.*

Post-dominators: Example

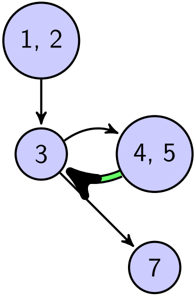
```
int sum = 0;
int i = 1;
while (i<N) {
  i += 1;
  sum += i;
}
printf("Sum: %d", sum);
```



$$spdom(4,5) = \{3;7\}$$
$$ipdom(4,5) = \{3\}$$

Backward Edges

```
int sum = 0;
int i = 1;
while (i<N) {
  i += 1;
  sum += i;
}
printf("Sum: %d", sum);
```



A back edge is an edge whose head dominates its tail[2].

[2] Back edges often identify loops.

Table of Contents

Cyclomatic Complexity

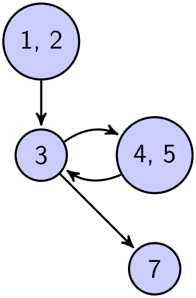
Cyclomatic Complexity

Cyclomatic complexity is a software metric that measures the quantitative complexity of a program by measuring the number of linearly independent paths through a program's source code. The complexity M *is defined as* $M = E - N + 2P$*, whereas* E *is the number of edges,* N *the number of nodes, and* P *the number of connected components (i.e., functions).*

Rule of thumb:
if the complexity M of a function is larger than 10-15 then the function should be split into multiple components.

Cyclomatic Complexity: Example

```c
int sum = 0;
int i = 1;
while (i<N) {
  i += 1;
  sum += i;
}
printf("Sum: %d", sum);
```



$E = 4$, $N = 4$, $P = 1$.

$M = E - N + 2P = 2$.

Table of Contents

Program Dependence Graph (PDG)

- Nodes are formed by single statements, not basic blocks.
- *Data-Dependence Graph* used to track data dependencies.
- *Control-Dependence Graph* used to track control dependencies.
- Widely used program representation!

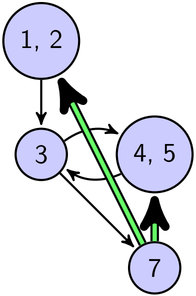
Data Dependence

Data Dependence

X is data dependent on Y, iff (i) there is a variable v defined at Y and used at X and (ii) there exists a path of nonzero length from Y to X along which v is not redefined.

Data Dependence: Example

```
1 int sum = 0;
2 int i = 1;
3 while (i<N) {
4   i += 1;
5   sum += i;
6 }
7 printf("Sum: %d", sum);
```



$$DataDep(sum, 7) = \{5, 1\}$$

Difficulties with Data Dependence

Statically computing data dependencies is hard due to aliasing: a variable can refer to multiple memory locations/objects.

```
int x, y, z, *p;
x = ...;
y = ...;
p = &x;
p = p + z;
... = *p;
```

Control Dependence

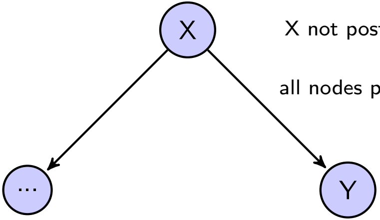
Control Dependence

Y is control dependent on X, iff X directly determines whether Y executes: statements inside one branch of a predicate are usually control dependent on the predicate.

- there exists a path from X to Y so that every node in the path other than X and Y is post-dominated by Y.
  (No such paths for nodes in a path between X and Y).
- Y does not strictly post-dominate X.
  (There is a path from X to End that does not pass Y or X==Y).

Reading assignment:
`http://dl.acm.org/citation.cfm?id=24041`

Control Dependence: Example



X not post-dominated by Y

all nodes post-dominated by Y

Using the PDG

A program dependence graph combines the control dependence graph and the data dependence graph of the program.

- In debugging: what statement possibly induced the fault?
- In security: possible redefinitions?

Table of Contents

Super Control-Flow Graph (SCFG)

- Adds inter-procedural aspects to intra-procedural CFG.
- Connect call sites to entry point of callee.
- Connect return statements back to call site.

Table of Contents

Call Graph (CG)

- Each node represents a function;
- each edge represents a function invocation.

The CG is useful when reasoning across function boundaries (e.g., for profiling or debugging).

Table of Contents

Other Representations

- Points-to Graph
- Static Single Assignment (SSA)

Analysis Tools

- C/C++: LLVM, CIL, CBMC
- Java: SOOT, Wala
- Binary: Valgrind, Pin, Libdetox

Questions?

?