



Control Flow Graph

Classic program representation
And operations



References

- *Compilers: Principles, Techniques and Tools* by Aho, Sethi, Ullman; Addison Wesley, 1986
- *Advanced Compiler Design & Implementation* by Steven S. Muchnick; Morgan Kaufman, 1997
- *Data Structures and Algorithms* by Aho, Hopcroft and Ullman; Addison Wesley 1987

2

Why a graph rep of program?

- Easier to analyze a graph
- Front-end analysis
 - Reverse engineering
 - Re-engineering
- Back-end analysis & optimization

3

Back-end optimizations

Common subexpression Elimination:

```
a = b + c
b = a - d
c = b + c
d = a - d
```

```
a = b + c
b = a - d
c = b + c
d = b
```

Dead-code elimination:

```
While ( i < MAX ) {
    sum = sum + i;
    i = i + 1;
    x = sum + 1;
}
```

// never use x!

How do I know when I can perform these optimizations?

4

Control & Data flow analysis

- Basic block: maximal sequence of instructions whose only entry is through the first instruction and only exit is from the last instruction
- First instruction (leader):
 - Entry point of routine
 - Target of a branch
 - Instruction following a branch or return
- BB is All instructions from a leader, up to but not including the next leader

5

Call instruction

- In most cases need not be considered a branch – longer and fewer bb's
- If language allows multiple or alternate return points – consider call to be a branch
 - Fortran
 - C – setjmp, longjmp
- Some optimizations make it necessary to consider call as a jump: e.g., instruction scheduling

6

Getting edges in cfg

- There is a directed edge from block B1 to B2 if B2 can immediately follow B1 in some execution sequence; that is:
 - There is a conditional or unconditional jump from B1 to B2, or
 - B2 immediately follows B1 in the order of the program
 - B1 is a predecessor of B2, B2 is a successor of B1

7

Flow graph

- $G = (N, E)$
 - N – set of nodes, each node a basic block
 - E – set of edges $\in N \times N$
 - (a, b)
 - $a \rightarrow b$
 - Sometimes *entry* & *exit* $\in N$
 - $\text{Succ}(b) = \{n \in N \mid \exists e \in E \ni e = b \rightarrow n\}$
 - $\text{Pred}(b) = \{n \in N \mid \exists e \in E \ni e = n \rightarrow b\}$

8

Example: computing fib

```

unsigned int fib(unsigned int m) {
    unsigned int f0=0, f1=0, f2=0;
    if ( m <= 1) {
        return m;
    }
    else {
        for (int i = 2; i <= m; ++i) {
            f2 = f0 + f1;
            f0 = f1;
            f1 = f2;
        }
        return f2;
    }
}
    
```

Compliments of
S. Muchnick

9

Intermediate code for fib

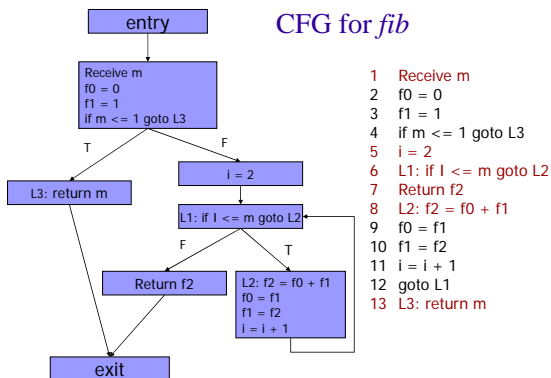
```

1  Receive m
2  f0 = 0
3  f1 = 1
4  if m <= 1 goto L3
5  i = 2
6  L1: if i <= m goto L2
7  Return f2
8  L2: f2 = f0 + f1
9  f0 = f1
10 f1 = f2
11 i = i + 1
12 goto L1
13 L3: return m
    
```

Leaders:
1, 5, 6, 7, 8, 13

10

CFG for fib



11

Representing CFG's

- Set of records, each consisting of:
 - count of the number of statements in the block
 - Pointer to leader of the block
 - Lists of predecessors and successors of the block

12

Loops

- Excellent candidates for optimizations
- Collection of nodes in G such that
 - All nodes in the collection are **strongly connected**: there is a path from any node, a , to any other node, b .
 - Collection has a unique entry
- A loop that contains no other loops is an inner loop

13

Algorithm to find $SCC \in G$

- Get Reverse Depth First numbering
- Build G_R (G -reverse)
- Start with highest DF number in G_R :
 - Mark each node until can't go any further, or
 - Find a node already marked
- Reverse edges in G_R for each $|SCC| > 1$

14

Algorithm for DF search

$L[v]$ is an adjacency list of vertices

```
void dfs( v : vertex ) {
    static count = 0;
    vertex w;
    mark[v] = visited;
    dfnumber[v] = count++;
    for ( each vertex w on L[v] ) {
        if ( mark[w] = unvisited ) {
            dfs(w);
        }
    }
}
```

15

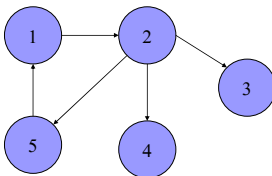
Algorithm to find Reverse DF numbering

$L[v]$ is an adjacency list of vertices

```
void dfs( v : vertex ) {
    static count = 1;
    vertex w;
    mark[v] = visited;
    for ( each vertex w on L[v] ) {
        if ( mark[w] = unvisited ) {
            dfs(w);
        }
    }
    reverse_dfnumber[v] = count++;
}
```

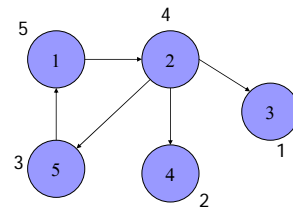
16

Example: consider graph G



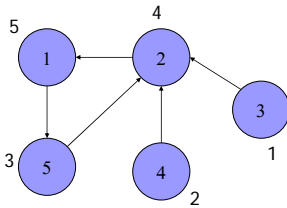
17

Reverse DF numbering for G



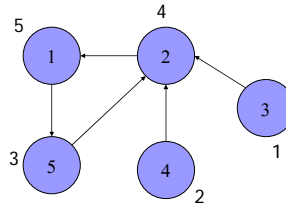
18

Find G_R



19

Find SCC's

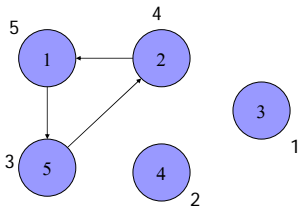


Starting with highest
Numbered node in G_R
Traverse until find an
Already numbered
Node, or can go no
Further. Do it again...

How do I know if I've
been to a node?

20

3 SCC's



21

Summary of SCC

- Can find loops in CFGs for structured and unstructured programs
- What's the running time?

$O(n + e)$



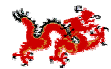
22

Using CFG to find CSE

- For each BB, construct a directed acyclic graph: dag
 - Leaves are labeled w/ unique ids: variable names or constants
 - Interior nodes are labeled by an operator symbol
 - Nodes are optionally given a sequence of ids: interior nodes compute values, ids have that value

23

DAGS



- Are not cfgs, but each node of a cfg can be represented as a dag
- Tell how the values in each BB are used by subsequent statements in the block
- Can be used to determine which values in the block might be used outside the block
- Can be built at various granularities: statement or operation

24

Dag construction

- Process each statement of a block in turn
- When we see $x = y + z$, look for nodes that represent the "current" values of y & z
 - these could be leaves or interior nodes
 - Create a node labeled "+" and give it 2 children: left node is for y and right node for z
 - Label this new node x
 - However if there is already denoting $y + z$, do not add a new node, simply label existing node x

25

Dag construction (cont)

- Two details:
 - If x had previously labeled some other node, we remove the label
 - For statement such as $x = y$ do not create a new node, rather append the label x to node for "current" value of y

26

Dag construction algorithm:

Perform steps 1 thru 3 for each statement in the bb
 Intuitively, $node(id)$ is the node of the dag that represents the value of id at the current point in the dag construction process

Initially, assume no nodes and $node$ is undefined for all arguments
 Suppose the current statement is:

- (i) $x = y \text{ op } z$
- (ii) $x = \text{op } y$
- (iii) $x = y$



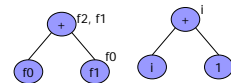
Treat a relop like if $i \leq 20$ goto as case (i) with x undefined.

1. If $node(y)$ is undefined, create a leaf labeled y and let $node(y)$ be this Node. In case (i), if $node(z)$ is undefined, create a leaf labeled z and let that leaf be $node(z)$.
2. In case (i), determine if there is a node labeled op , whose left child is $node(y)$ and whose right child is $node(z)$. (This check is to catch common subexpressions). If not, create such a node. In either event, let n be the node found or created. In case (ii), determine whether there is a node labeled op , whose lone child is $node(y)$. If not, create such a node, and let n be the node found or created.
3. Delete x from the list of attached identifiers for $node(x)$. Append x to the List of attached identifiers for the node n found in (2) and set $node(x)$ to n .

27

Fib example

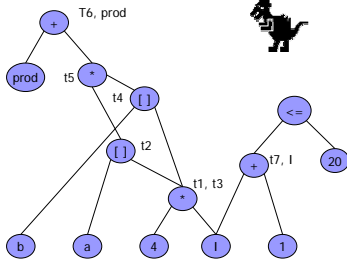
```
L2: f2 = f0 + f1
f0 = f1
f1 = f2
i = i + 1
```



28

More complicated example

- (1) $t1 = 4 * i$
- (2) $T2 = a [t1]$
- (3) $T3 = 4 * i$
- (4) $T4 = b [t3]$
- (5) $T5 = t2 * t4$
- (6) $T6 = \text{prod} + t5$
- (7) $\text{prod} = t6$
- (8) $T7 = i + 1$
- (9) $i = T7$
- (10) If $i \leq 20$ goto (1)



- (1) $t1 = 4 * i$
- (2) $T2 = a [t1]$
- (3) $T4 = b [t1]$
- (4) $T5 = t2 * t4$
- (5) $\text{prod} = \text{prod} + t5$
- (6) $i = i + 1$
- (7) If $t7 \leq 20$ goto (1)

29

Applications of dags

- Automatically detects cse
- Determine which ids have their values used in the block: leaves
- Which statements compute values that could be used outside the block
- Generate optimized code – exploit cse and don't perform assignment unless necessary: which node has attached id used outside block (need live variable analysis)

30

Arrays

```
x = a [i]
a[j] = y
z = a [i]
```

But, suppose $i = j$ and $y \neq a[i]$

Similar problem occurs if we have
An assignment such as $*p = w$;

```
x = a[i]
a[j] = y
z = x
```

A procedure call kills all nodes, since
In the absence of knowledge about
What happened in the procedure, we
Must assume that any variable may be
Changed by a side effect.

31

Data Flow Analysis



32

Intro to global data flow analysis

- The computation of data flow information can be automated
- Need info about where definitions occur (l-values) and uses occur (r-values)
- “global” means *local* to a function but *global* (across blocks) to the cfg
- uses data flow equations

33

Uses of data flow analysis

- Code optimization:
 - Dead code elimination
 - Better register usage
- Data flow testing
- Debugging
- Alias analysis – better memory usage

34

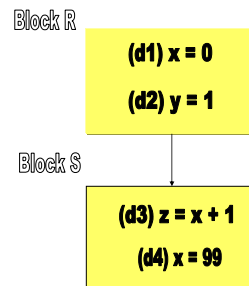
Data flow equations

$$out[S] = gen[S] \cup (in[S] - kill[S])$$

- $Out[S]$ is the set of all defs that leave a block S (said to be “live”)
- $gen[S]$ all new defs that are generated by the block S
- $in[S]$ all defs that enter block S
- $kill[S]$ all defs that are killed by a def in block S

35

Find $in[S]$, $out[S]$, $gen[S]$, $kill[S]$



36

Details

- When we write $out[S]$ we imply that there is a unique **end point** from which control flows out of a block S
- There are subtleties attached to *procedure calls, pointer variables and arrays*

37

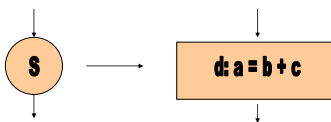
Reaching definition

- A *definition* of x is a statement that assigns to x , or may assign to x
- a definition d *reaches* a point p if there is a path from the point immediately following d to p , such that d is not killed along that path

38

More examples of *gen, kill, out*

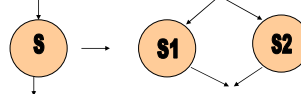
- $Gen[s] = \{d\}$
- $kill[S] = D - \{d\}$
- $out[S] = gen[S] \cup (in[S] - kill[S])$



39

More examples of *gen, kill, out*

- $Gen[S] = gen[S1] \cup gen[S2]$
- $kill[S] = kill[S1] \cap kill[S2]$
- $in[S1] = in[S]$
- $in[S2] = in[S]$
- $out[S] = out[S1] \cup out[S2]$



40

Representing sets

- Sets of definitions, such as $gen[S]$ can be represented compactly using bit vectors
 - assign a number to each definition of interest in the cfg
 - the bit vector representing a set of definitions will have 1 in position i if the definition numbered i is in the set
- the C++ standard library has an efficient implementation of sets

41

Algorithm to compute reaching definitions



Input: CFG for which $kill[B]$ & $gen[B]$ have been computed for each block B
output: $in[B], out[B]$ for each block B

method: use an iterative approach, starting with the estimate that $in[B]$ is empty for all B . We use a boolean variable, *change*, to record on each pass through the blocks whether *in* has changed; if not, we're finished.

42

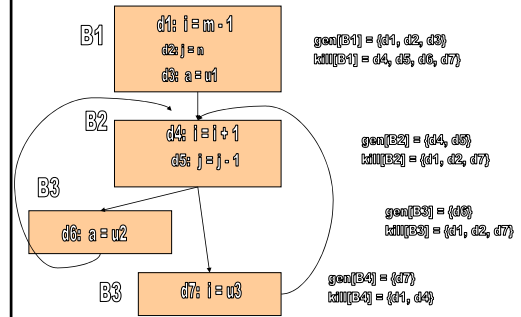
Algorithm to compute reaching definitions (cont)

```

For each block  $B$  do  $out[B] = gen[B]$ 
 $change = true$ 
while  $change$  do begin
   $change = false$ 
  for each block  $B$  do
     $in[B] = \cup out[P]$ ,  $P$  a predecessor of  $B$ 
     $oldout = out[B]$ 
     $out[B] = gen[B] \cup (in[B] - kill[B])$ 
    if  $out[B] \neq oldout$  then  $change = true$ 
  end for
end while
  
```

43

Flow graph to illustrate reaching definitions



44

Computation of *in* and *out* sets

Block B	initial		pass 1		pass 2	
	in[B]	out[B]	in[B]	out[B]	in[B]	out[B]
B1	000 0000	111 000	000 0000	111 0000	000 0000	111 0000
B2	000 0000	000 1100	111 0011	001 1110	111 1111	001 1110
B3	000 0000	000 0010	001 1110	000 1110	001 1110	000 1110
B4	000 0000	000 0001	001 1110	001 0111	001 1110	001 0111

45