Program Analysis

Readings

- Notes on Representation and Analysis of Software (Sections 1--5)
- The Program Dependence Graph and Its Use in Optimization
- Dragon book
Program Analysis

Control-flow Analysis

Control Flow: Basic Blocks

• **Basic block**: a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt of possibility of branch except at the end
• A basic block may or may not be **maximal**
Computing Control Flow: Algorithm

*Input:* a list of program statements in some form  
*Output:* A list of control-flow graph (CFG) nodes and edges  

*Method:*  
- Construct basic blocks  
- Create entry and exit nodes; create edge (entry, B1); create (Bk, exit) for each Bk that represents an exit from program  
- Add CFG edge from Bi to Bj if Bj can immediately follow Bi in some execution, i.e.,  
  - There is conditional or unconditional goto from last statement of Bi to first statement of Bj or  
  - Bj immediately follows Bi in the order of the program and Bi does not end in an unconditional goto statement  
- Label edges that represent conditional transfers of control

---

Computing Control Flow Graphs (CFGs)

**Procedure AVG**

```
S1  count = 0
S2  fread(fptr, n)
S3  if EOF goto S11
S4  if (n >= 0) goto S7
S5  return (error)
S6  goto S9
S7  nums[count] = n
S8  count ++
S9  fread(fptr, n)
S10 goto S3
S11 avg = mean(nums,count)
S12 return(avg)
```
### CFG with Maximal Basic Blocks

**Procedure AVG**

1. **S1**  \( \text{count} = 0 \)
2. **S2**  \( \text{fread(fptr, n)} \)
3. **S3**  while (not EOF) do
4. **S4**     if \((n < 0)\) do
5. **S5**         return (error)
6. **else**
7. **S6**         nums[count] = n
8. **endif**
9. **S7**     count ++
10. **endif**
11. **S8**  \( \text{fread(fptr, n)} \) endwhile
12. **S9**  avg = mean(nums,count)
13. **S10** return(avg)

---

### CFGs exercise

**Procedure Trivial**

1. **S1**  read (n)
2. **S2**  switch (n)
3. **case 1:**
4. **S3**     write ("one")
5. **break**
6. **case 2:**
7. **S4**     write ("two")
8. **case 3:**
9. **S5**     write ("three")
10. **break**
11. **default**
12. **S6**     write ("Other")
13. **endswitch**
14. **end Trivial**

---

Build a CFG with maximal basic blocks for the program.
CFGs exercise

Procedure Trivial

S1 read (n)
S2 switch (n)
    case 1:
        S3 write ("one")
        break
    case 2:
        S4 write ("two")
        break
    case 3:
        S5 write ("three")
        break
    default
        S6 write ("Other")
        endswitch
end Trivial
CFG: Terminology

- **CFG** = \( <N, E> \), rooted directed graph
  - \( N \) = set of nodes
  - \( E \subseteq N \times N \) = set of edges
  - entry \( \in N \), exit \( \in N \)
- **Successors/predecessors** of a basic block
- **Branch node**
- **Join node**

![Diagram of CFG]

Some Useful Concepts

**Depth-First Search (DFS):** Visits descendants before visiting siblings

**Depth-first spanning tree:** All nodes, only edges traversed in the DFS

**Depth-first presentation:** spanning tree + remaining edges (marked)
  - Forward edges: node \( \rightarrow \) direct descendant
  - Back edges: node \( \leftarrow \) ancestor in the tree
  - Cross edges: node \( \leftrightarrow \) neither ancestor nor descendant

![Diagrams of DFS concepts]
Some Useful Concepts

**Preorder traversal:** Traversal of the depth-first spanning tree in which each node is processed before its descendants

**Postorder traversal:** Traversal of the depth-first spanning tree in which each node is processed after its descendants

---

Control Flow Analysis: Dominance

A node n **dominates** a node m (n dom m) if every path from the **entry** to m includes n

A node n **postdominates** a node m (n pdom m) if every path from the m to the **exit** includes n

Antisymmetric, reflexive, transitive

- Can be represented using a tree whose root is the entry node for dominance and the exit node for postdominance
Control Flow Analysis: Dominance

CFG

entry
S1
T
F
S2
S3
S4
F
T
S5
S6
exit

Dominance Tree

entry
S1
S2
exit
S3
S4
S5
S6

Control Flow Analysis: Dominance

CFG

entry
S1
T
F
S2
S3
S4
F
T
S5
S6
exit

Postdominance Tree

exit
S2
S1
S5
entry
S4
S3
S6
Dominance (properties)

n dominates m iff
- n = m, or
- n = unique immediate predecessor of m, or
- m has more than one predecessor and for all immediate predecessors l of m (n ≠ l)
  - n dominates l

Algorithm

Dominance (algorithm)

Input: N, pred, entry
Output: domin: n → {n}
- domin(entry) = {entry}
- foreach n ∈ N - {entry}
  - domin(n) = N
- label: change = false
- foreach n ∈ N - {entry}
  - T = ∩ p ∈ pred(n) domin(p)
  - D = {n} ∪ T
  - if D ≠ domin(n)
    - change = true
    - domin(n) = D
- if change = true goto label
- return domin
Finding Loops

- Loops are very important (why?)
- How do we identify loops?
- Not every cycle is a loop (for optimization)
- Different kinds of loops
  - Irreducible loops
  - Reducible loops
- We’ll see how to identify natural loops, which account for most loop in real programs
- Definition:
  "single entry, head dominates all nodes"

Natural Loops

- Given G = {N, E}, subgraph G' = {N', E'} and s' in N'
  define a natural loop with entry s' if s' dominates all nodes in N'

- Any natural loop?
- Why?
Using Dominance to Find Loops

Dominance trees can be used to identify loops

**Back edge** = $t \rightarrow h$, $h$ dominates $t$

**Natural loop** = given a back edge $t \rightarrow h$, its natural loop is the subgraph consisting of:

- node $h$ (loop header)
- all nodes dominated by $h$ that can reach $t$ w/o traversing $h$
- all edges that connect nodes in this set

Example

Find back edges and associated loops
Applications of Control Flow

Further analyses
- Data-flow, reachability, …

Program understanding
- Program structure and flow are made explicit

Software complexity
- From the structural standpoint

Structural coverage in testing
- Statement, branch, path, …

Program Analysis

Data-Flow Analysis
Data-flow Analysis

- Motivation
- Data-flow problems (reaching definitions, reachable uses, …)
- Iterative data-flow analysis
- Other types of data-flow analysis: worklist, …
- DU-chains, UD-Chains

Uses of Data-Flow Analyses

**Compiler Optimization**

E.g., *Constant propagation*

\[ a = c + 10 \]

suppose every assignment to \( c \) that reaches this statement assigns 5

then \( a \) can be replaced by 15

⇒ need to know *reaching definitions*: which definitions of variable \( c \) reach a statement
Uses of Data-Flow Analyses

Software Engineering Tasks
E.g., Data-flow testing
suppose that a statement assigns a value but the use of that value is never executed under test

\[ a = c + 10 \]
\[ d = a + y \]

need definition-use pairs (du-pairs): associations between definitions and uses of the same variable or memory location

Uses of Data-Flow Analyses

Software Engineering Tasks
E.g., Debugging
suppose that a has the incorrect value in the statement

\[ a = c + y \]

need data dependence information: statements that can affect the incorrect value at a given program point
Basic Definitions

**Definition** and **Use**
Consider statement $X = Y + Z$
- Definitions?
- Uses? (p-use or c-use?)

**Kill** and **Reach**
- A definition $d$ of a variable $x$ is *killed* at a statement $s$ iff $s$ redefines $x$ and the last assignment to $x$ was $d$
- A definition $d$ of $x$ *reaches* $s$ if there is at least a path from $d$ to $s$ along which $x$ is not killed (def-clear path)

Goal of data-flow analysis

- Data-flow analysis computes the flow of different data throughout the program
- Wide range of analyses, from reaching definitions to slicing
- In general, performed on the CFG
- Exact solutions to most problems are undecidable (Why?)
  ➔ Need to approximate
Safety and Precision

- Can you define **safety** and **precision**?
- Are they related?
- Can you give some examples?
- Is imprecision a problem only for data-flow analysis?

Approximate analysis can **overestimate** the solution:
- Solution contains all actual information plus some spurious information
- This type of analysis is safe or conservative

Approximate analysis can **underestimate** the solution:
- Solution may not contain all actual information
- This type of analysis is unsafe

For optimization, need conservative, safe analysis
For software engineering tasks, may be able to use unsafe analysis information

Major challenge for data-flow analysis: provide **safe** yet **precise** (i.e., minimize the spurious information) information in an **efficient** way
Data-Flow Problems

Compute the flow of data to points in the program — e.g.,
- Where does the assignment to I in statement 1 reach?
- Where does the expression computed in statement 2 reach?
- Which uses of variable J are reachable from the end of B1?

Points in a basic block:
- before first statement
- after last statement
- between statements

Reaching Definitions

Problem:
Determine the set of definitions that reach a point in the program

Where are the definitions in the program?
- Of variable I:
- Of variable J:

Which basic blocks (before block) do these definitions reach?
- Def 1 reaches
- Def 2 reaches
- Def 3 reaches
- Def 4 reaches
- Def 5 reaches
Reaching Definitions

**Problem:**
Determine the set of definitions that reach a point in the program

Where are the definitions in the program?
- Of variable I: 1, 3
- Of variable J: 2, 4, 5

Which basic blocks (before block) do these definitions reach?
- Def 1 reaches B2
- Def 2 reaches B1, B2, B3
- Def 3 reaches B1, B3, B4
- Def 4 reaches B4
- Def 5 reaches exit

Typically solved by creating a set of **data-flow equations**

Iterative Reaching Definitions

**Iterative Method:**
1. Compute two kinds of local information (i.e., within a basic block)
   - \( \text{GEN}[B] \) is the set of definitions that are created (generated) within B
   - \( \text{KILL}[B] \) is the set of definitions in the program that are killed if they reach B's entry

2. Compute two other sets by propagation
   - \( \text{IN}[B] \) is the set of definitions that reach the beginning of B
   - \( \text{OUT}[B] \) is the set of definitions that reach the end of B
Iterative Reaching Definitions

Iterative Method (cont’d):

Propagation method:

- **Initialize** \( \text{IN}[B] \), \( \text{OUT}[B] \) sets for all \( B \)
- **Iterate** over all \( B \) until there are no changes in \( \text{IN}[B] \) and \( \text{OUT}[B] \), computed as
  - \( \text{IN}[B] = \bigcup \text{OUT}[P] \), \( P \) is a pred. of \( B \)
  - \( \text{OUT}[B] = \text{GEN}[B] \cup (\text{IN}[B] – \text{KILL}[B]) \)

---

Iterative Reaching Definitions

algorithm ReachingDefinitions
Input: CFG w/ \( \text{GEN}[B], \text{KILL}[B] \) for all \( B \)
Output: \( \text{IN}[B], \text{OUT}[B] \) for all \( B \)
begin ReachingDefinitions
  \( \text{IN}[B] = \text{empty}; \text{OUT}[B] = \text{GEN}[B] \), for all \( B \); change = true
  while change do begin
    change = false
    foreach \( B \) do begin
      \( \text{IN}[B] = \text{union} \text{OUT}[P] \), \( P \) is a predecessor of \( B \)
      \( \text{Oldout} = \text{OUT}[B] \)
      \( \text{OUT}[B] = \text{GEN}[B] \cup (\text{IN}[B] – \text{KILL}[B]) \)
      if \( \text{OUT}[B] \neq \text{Oldout} \) then change = true
    endfor
  endwhile
end Reaching Definitions
Iterative Reaching Definitions

• Where is imprecision affecting the computation?
• Is the algorithm guaranteed to converge? Why or why not?
• What is the worst-case time complexity?
• What is the worst-case space complexity?
• Which visiting order could improve worst-case time complexity? By how much?

Depth-first ordering (preorder)
Depth-first ordering (preorder)

Given a depth-first spanning tree for the graph, the **depth** is the largest number of back edges on any acyclic path.

Reachable Uses

A use of A is reachable from a point p if there exists a def-clear path wrt A from p to the use.

Reachable uses also called **upwards exposed uses**.
Reachable Uses

Where are the uses in the program?
- Of variable I: 2
- Of variable J: 4, 5

From which basic blocks (end of block) are the uses reachable?
- Use 2 is unreachable
- Use 4 is reachable from B1, B2
- Use 5 is reachable from B3
Iterative Reachable Uses

Iterative Method:

1. Local information:
   - GEN[B]: the set of uses that are created (generated) within B and can be reached from the beginning of B (upwards exposed uses for B)
   - KILL[B]: the set of uses such that there is a def of the corresponding variable in B

2. Propagation:
   - IN[B] =
   - OUT[B] =

Iterative Reachable Uses

Iterative Method:

1. Local information:
   - GEN[B]: the set of uses that are created (generated) within B and can be reached from the beginning of B (upwards exposed uses for B)
   - KILL[B]: the set of uses such that there is a def of the corresponding variable in B

2. Propagation:
   - IN[B] =
   - OUT[B] =
Iterative Reachable Uses

**Iterative Method:**

1. Local information:
   - \( GEN[B] \) is the set of uses that are created (generated) within \( B \) and can be reached from the beginning of \( B \) (upwards exposed uses for \( B \)).
   - \( KILL[B] \) is the set of uses such that there is a def of the corresponding variable in \( B \).

2. Propagation:
   - \( \text{IN}[B] = \) 
   - \( \text{OUT}[B] = \) 

![Diagram](source_image_url)
Iterative Reachable Uses

algorithm ReachableUses
Input: CFG w/ GEN[B], KILL[B] for all B
Output: IN[B], OUT[B] for all B
begin ReachableUses
    OUT[B] = GEN[B], IN[B] = empty, for all B; change = true
    while change do begin
        change = false
        foreach B do begin
            IN[B] = union OUT[S], S is a successor of B
            OldOUT = OUT[B]
            if OUT[B] != OldOUT then change = true
        endfor
    endwhile
end reachableUses

Algorithm Comparison

Similarities between RD and RU

Differences between RD and RU
Algorithm Comparison

Similarities between RD and RU

- Local information (GEN and KILL) computed for each B
- Flow into block computed as union of predecessors in flow
- Iteration until no more changes

Differences between RD and RU

- RD flow is forward; RU flow is backward
- RD best ordering is depth-first (topological); RU best ordering is reverse depth-first (reverse topological)

Other DF Approaches (worklist)

Data-flow for nodes 1, 2, 3 never changes but is computed on every iteration of the algorithm

In general, nodes involved in the computation may be a small subset of the nodes in the graph
Other DF Approaches (worklist)

algorithm RDWorklist
Input: GEN[B], KILL[B] for all B
Output: reaching definitions for each B
Method:
initialize IN[B], OUT[B] for all B; add successors of B initially involved in computation to worklist W
repeat
  remove B from W
  Oldout=OUT[B]
  compute IN[B], OUT[B]
  if oldout \neq OUT[B] then add successors of B to W endif
until W is empty

Compute RD for f1 using RDWorklist
• GEN[3] is \{3\}, GEN[10] is \{10\}, KILL[3] is \{10\}, KILL[10] is \{3\}
• Add successors of 3, 10 to W
• remove 4 from W, compute IN[4], OUT[4], etc.