Compilerconstructie

najaar 2013

http://www.liacs.nl/home/rvvliet/coco/

Rudy van Vliet
kamer 124 Snellius, tel. 071-527 5777
rvvliet(at)liacs(dot)nl

college 9, dinsdag 26 november 2013

Code Optimization

1

9.2 Introduction to Data-Flow Analysis

- Optimizations depend on data-flow analysis, e.g.,
  - Global common subexpression elimination
  - Dead-code elimination
- Execution path yields program state
- Extract information from program state for data-flow analysis
- Usually infinite number of execution paths / program states
- Different analyses extract different information

2

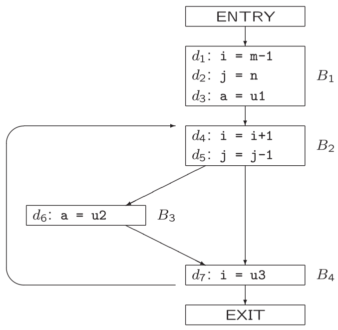
Data-Flow Analysis (Examples)

Extract information from program states at program point

- Reaching definitions: which definitions (assignments of values) of variable a reach program point?
  Useful for debugging
- Can variable x only have one constant value at program point?
  Useful for constant folding

3

Computing Reaching Definitions



Reaching definitions
- Before B_1: $\emptyset$
- After B_1: $\{d_1, d_2, d_3\}$
- Before B_2: . . .

4

Data Flow Values

- IN$[s]$: before statement s
- OUT$[s]$: after statement s
- Transfer function f_s
  - forward: OUT$[s] = f_s($IN$[s])$
  - backward: IN$[s] = f_s($OUT$[s])$

5

Computing Reaching Definitions

- Effect of single definition $d: u = v$ *op* w:
  - OUT$[d] = \{d\} \cup ($IN$[d] - \ldots)$

6

Computing Reaching Definitions

Effect of single definition $d: u = v$ *op* w:

- OUT$[d] = \{d\} \cup ($IN$[d] - \{$all other definitions of u in program$\})$
- Hence,

$$f_d(x) = \{d\} \cup (x - \{\text{all other definitions of } u \text{ in program}\}) = gen_d \cup (x - kill_d)$$

where

$$gen_d = \{d\}$$
$$kill_d = \{\text{all other definitions of } u \text{ in program}\}$$

7

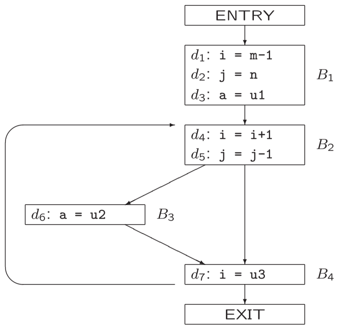
Computing Reaching Definitions

Effect of block B, with definitions $1, 2, \ldots, n$:

$$gen_B = \{n, n-1, \ldots, 1\} - \{\text{ definitions killed afterwards }\}$$
$$= gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \ldots$$
$$kill_B = kill_1 \cup kill_2 \cup \ldots \cup kill_n$$

8

Computing Reaching Definitions



$gen_{B_1} = \{d_1, d_2, d_3\}$
$kill_{B_1} = \{d_4, d_5, d_6, d_7\}$

$gen_{B_2} = \{d_4, d_5\}$
$kill_{B_2} = \{d_1, d_2, d_7\}$

$gen_{B_3} = \{d_6\}$
$kill_{B_3} = \{d_3\}$

$gen_{B_4} = \{d_7\}$
$kill_{B_4} = \{d_1, d_4\}$

Iterative Algorithm for Computing Reaching Definitions

OUT[ENTRY] $= \emptyset$
for each basic block B other than ENTRY
    OUT[B] $= \emptyset$

while (changes to any OUT occur)
    for each basic block B other than ENTRY
    {   IN[B] $= \cup_{\text{predecessors } P \text{ of } B}$ OUT[P]

        OUT[B] $= gen_B \cup ($IN[B] $- kill_B)$
    }

Typical form of algorithm for forward data-flow analysis

Example with $B = B_1, B_2, B_3, B_4,$ EXIT...

Implementation of Iterative Algorithm for Computing Reaching Definitions

With bit vectors

Block B	OUT[B]^0	IN[B]^1	OUT[B]^1	IN[B]^2	OUT[B]^2
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	000 0000	001 0111	001 0111	001 0111

Live-Variable Analysis

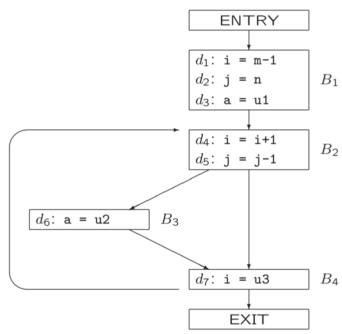
- Variable x is live at program point p, if value of x at p could be used later along *some* path
- Otherwise x is dead at p
- Information useful for register allocation (see college 7)
- Information about later use must be propagated backwards

Live-Variable Analysis

Effect of block B on live variables

- def_B: variables *defined* in B
- use_B: variables that may be *used* in B prior to any definition in B

Computing Liveness



$def_{B_1} = \{i, j, a\}$
$use_{B_1} = \{m, n, u1\}$

$def_{B_2} = \{i, j\}$
$use_{B_2} = \{i, j\}$

$def_{B_3} = \{a\}$
$use_{B_3} = \{u2\}$

$def_{B_4} = \{i\}$
$use_{B_4} = \{u3\}$

Iterative Algorithm for Computing Liveness

IN[EXIT] $= \emptyset$
for each basic block B other than EXIT
    IN[B] $= \emptyset$

while (changes to any IN occur)
    for each basic block B other than EXIT
    {   OUT[B] $= \cup_{\text{successors } S \text{ of } B}$ IN[S]

        IN[B] $= use_B \cup ($OUT[B] $- def_B)$
    }

Typical form of algorithm for backward data-flow analysis

Available expressions

- Is (value of) expression x *op* y available?
- Useful for global common subexpression elimination
- Can be decided with data-flow analysis

Available Expressions (Example)

Computing Available Expressions (Example)

Statement	Available Expressions
	$\emptyset$
a = b + c	
	$\{b+c\}$
b = a - d	
	$\{a-d\}$
c = b + c	
	$\{a-d\}$
d = a - d	
	$\emptyset$

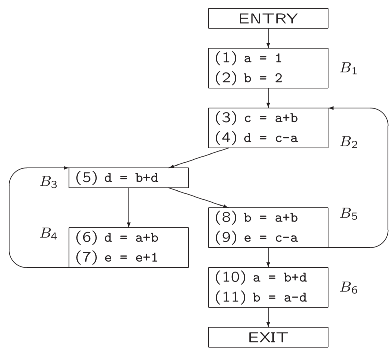
Computing Available Expressions

OUT[ENTRY] $= \emptyset$
for each basic block B other than ENTRY
    OUT[B] $= U$

while (changes to any OUT occur)
    for each basic block B other than ENTRY
    { IN[B] $= \cap_{\text{predecessors } P \text{ of } B}$ OUT[P]

        OUT[B] $= e\_gen_B \cup$ (IN[B] $-$ e_kill_B)
    }

Flow Graph For Data Flow Analysis



Efficient Iterative Data-Flow Analysis

Example: computing reaching definitions

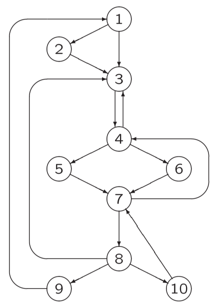
OUT[ENTRY] $= \emptyset$
for each basic block B other than ENTRY
    OUT[B] $= \emptyset$

while (changes to any OUT occur)
    for each basic block B other than ENTRY
    { IN[B] $= \cup_{\text{predecessors } P \text{ of } B}$ OUT[P]

        OUT[B] $= gen_B \cup$ (IN[B] $-$ $kill_B$)
    }

Order of blocks in second for-loop matters

Efficient Iterative Data-Flow Analysis



Order of blocks in second for-loop matters

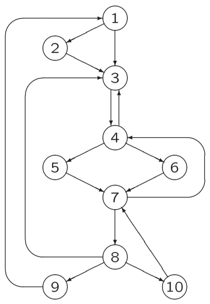
9.6 Loops in Flow Graphs

- Optimizations of loops have significant impact
- Essential to identify loops
- Used in region based analysis (not for exam)

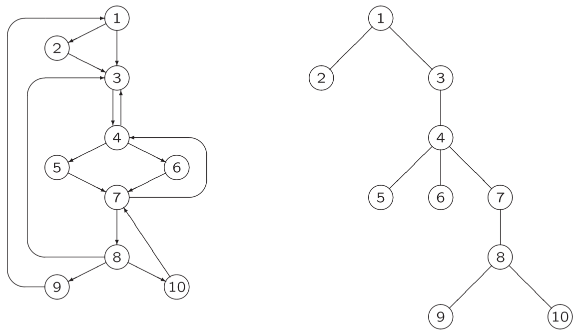
Dominators

- Dominators:
  - Node d dominates node n if every path from ENTRY node to n goes through d: d *dom* n
  - Node n dominates itself
  - Loop entry dominates all nodes in loop
- Immediate dominator m of n:
  last dominator on (any) path from ENTRY node to n
  - if $d \neq n$ and d *dom* n, then d *dom* m

Dominators (Example)



Dominator Trees (Example)



Finding Dominators

Forward data-flow analysis

N is set of all nodes

OUT[ENTRY] = {ENTRY}
for each node n other than ENTRY
    OUT$[n] = N$

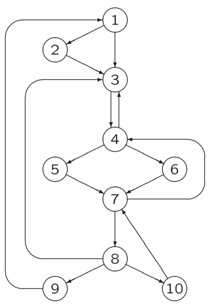
while (changes to any OUT occur)
    for each node n other than ENTRY
    {   IN$[n] = \cap_{\text{predecessors } m \text{ of } n}$ OUT$[m]$

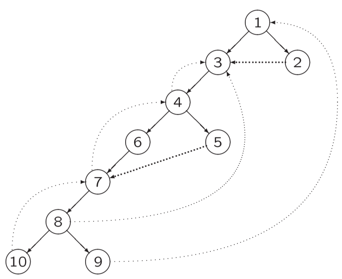
        OUT$[n]$ = IN$[n] \cup \{n\}$
    }

Depth-First Traversal

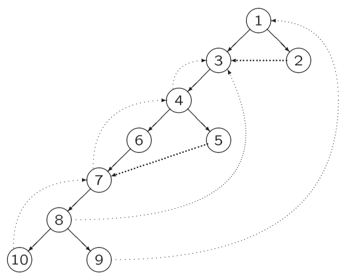
- Depth-first traversal of graph
  - Start from entry node
  - Recursively visit neighbours (in any order)
  - Hence, visit nodes far away from the entry node as quickly as it can (DF)

A Depth-First Spanning Tree





A Depth-First Spanning Tree



- Advancing edges
- Retreating edges
- Cross edges
- Back edge $a \to b$, if b dominates a (regardless of DFST)
- Each back edge is retreating edge in DFST
- Flow graph is reducible, if each retreating edge in any DFST is back edge

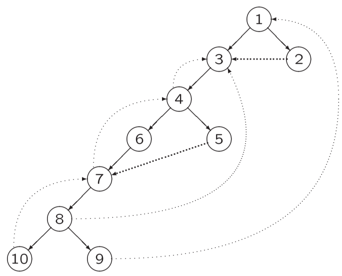
(Non)Reducible flow graphs

- In practice, almost every flow graph is reducible
- Example of nonreducible flow graph (with advancing edges)
- To decide on reducibility:
  1. Remove back edges
  2. Is remaining graph acyclic?

Natural loops

- If loop has single-entry node, then compiler can assume certain initial conditions
- Natural loop
  1. Has single-entry node: header
  2. Has back edge to header
- Each back edge $n \to d$ determines natural loop, consisting of
  - d
  - all nodes that can reach n without going through d
- Constructing natural loop of back edge. . .

Natural Loops (Example)



No Natural Loops

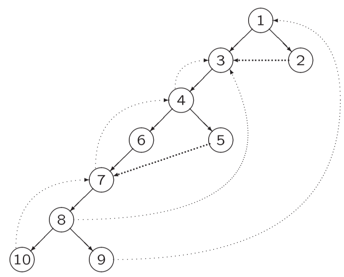
Natural Loops

- Useful property: unless two natural loops have same header
  - either they are disjoint
  - or one is nested within other

  Allows for inside-out optimization

- Assumption: if necessary, combine natural loops with same header. . .

A Depth-First Ordering

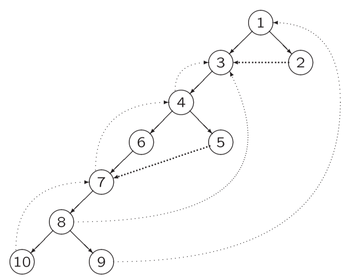


- Depth-First Ordering: nodes in DFST in WRL order ≈ reverse of postorder
- Example: 1,2,3,4,5,6,7,8,9,10
- Edge $m \rightarrow n$ is retreating, if and only if n comes before m in depth-first ordering

Depth of Flow Graph

- Depth of DFST is largest number of retreating edges on any cycle-free path
- If flow graph is reducible, then depth is independent of DFST: depth of flow graph
- Depth $\leq$ depth of loop nesting in flow graph

Depth of Flow Graph (Example)



Depth is 3, because of path $10 \rightarrow 7 \rightarrow 4 \rightarrow 3$

Speed of Convergence of Iterative Data-Flow Algorithms

In data-flow analysis, can significant events be propagated to node along acyclic path?

- Yes for
  - Reaching definitions
  - Live-variable analysis
  - Available expressions
- No for
  - Copy propagation

If yes, then fast convergence possible

Efficient Iterative Data-Flow Analysis

Example: computing reaching definitions

```
OUT[ENTRY] = ∅
for each basic block B other than ENTRY
    OUT[B] = ∅

while (changes to any OUT occur)
    for each basic block B other than ENTRY
    {   IN[B] = ∪_{predecessors P of B} OUT[P]

        OUT[B] = gen_B ∪ (IN[B] − kill_B)
    }
```

Order of blocks in second for-loop matters

Fast Convergence

- Forward data-flow problem: visit nodes in depth-first-order
- Recall: edge $m \to n$ is retreating, if and only if n comes before m in depth-first ordering
- Example: path of propagation of definition d:

$$3 \to 5 \to 19 \to 35 \to 16 \to 23 \to 45 \to 4 \to 10 \to 17$$

- Number of iterations: $1 +$ depth $(+\ 1)$
- Typical flow graphs have depth 2.75
- Backward data-flow problem: visit nodes in reverse of depth-first-order

En verder. . .

- Dinsdag 3 december: practicum over opdracht 4
- Maandag 9 december: inleveren opdracht 4
- Dinsdag 17 december, 10:00–13:00: tentamen
- Vragenuur ?

Compiler constructie

college 9
Code Optimization

Chapters for reading:
9.2, 9.6