Introduction to Machine-Independent Optimizations - 4

Data-Flow Analysis

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

- What is code optimization? (in part 1)
- Illustrations of code optimizations (in part 1)
- Examples of data-flow analysis
- Fundamentals of control-flow analysis
- Algorithms for two machine-independent optimizations
- SSA form and optimizations

Foundations of Data-flow Analysis

- Basic questions to be answered
  1. In which situations is the iterative DFA algorithm correct?
  2. How precise is the solution produced by it?
  3. Will the algorithm converge?
  4. What is the meaning of a "solution"?
- A DFA framework $(D, V, \wedge, F)$ consists of

D	:	A direction of the dataflow, either forward or backward
V	:	A domain of values
$\wedge$	:	A meet operator; $(V, \wedge)$ form a semi-lattice
F	:	A family of transfer functions, $V \longrightarrow V$ F includes constant transfer functions for the ENTRY/EXIT nodes as well

Properties of the Iterative DFA Algorithm

1. If the iterative algorithm converges, the result is a solution to the DF equations
2. If the framework is monotone, then the solution found is the maximum fixpoint (MFP) of the DF equations
    - An MFP solution is such that in any other solution, values of $IN[B]$ and $OUT[B]$ are $\leq$ the corresponding values of the MFP (i.e., less precise)
3. If the semi-lattice of the framework is monotone and is of finite height, then the algorithm is guaranteed to converge
    - Dataflow values decrease with each iteration
      Max no. of iterations = height of the lattice $\times$ no. of nodes in the flow graph

Meaning of the Ideal Data-flow Solution

- Find all possible execution paths from the start node to the beginning of B
- (Assuming forward flow) Compute the data-flow value at the end of each path (using composition of transfer functions)
- No execution of the program can produce a *smaller* value for that program point than

$$IDEAL[B] = \bigwedge_{P,\ a\ possible\ execution\ path\ from\ start\ node\ to\ B} f_P(v_{init})$$

- Answers greater (in the sense of $\leq$) than IDEAL are incorrect (one or more execution paths have been ignored)
- Any value smaller than or equal to IDEAL is conservative, *i.e.,* safe (one or more infeasible paths have been included)
- Closer the value to IDEAL, more precise it is

Meaning of the Meet-Over-Paths Data-flow Solution

- Since finding all execution paths is an undecidable problem, we approximate this set to include all paths in the flow graph

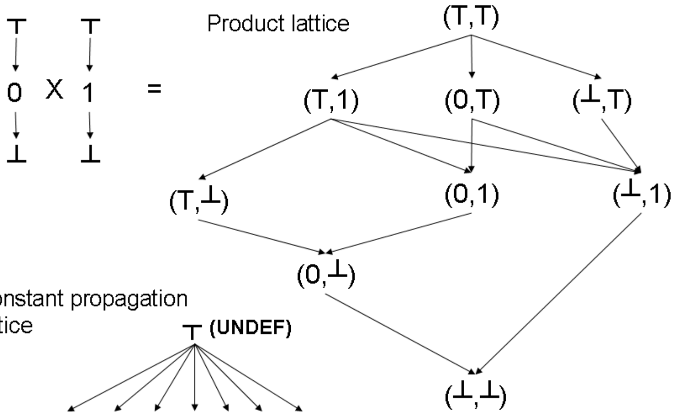
$$MOP[B] = \bigwedge_{P,\ a\ path\ from\ start\ node\ to\ B} f_P(v_{init})$$

- $MOP[B] \leq IDEAL[B]$, since we consider a superset of the set of execution paths

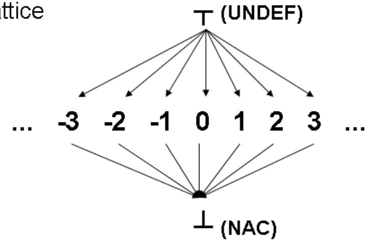
Meaning of the Maximum Fixpoint Data-flow Solution

- Finding all paths in a flow graph may still be impossible, if it has cycles
- The iterative algorithm does not try this
  - It visits all basic blocks, not necessarily in execution order
  - It applies the $\wedge$ operator at each join point in the flow graph
  - The solution obtained is the Maximum Fixpoint solution (MFP)
- If the framework is distributive, then the MOP and MFP solutions will be identical
- Otherwise, with just monotonicity, $MFP \leq MOP \leq IDEAL$, and the solution provided by the iterative algorithm is safe

Product of Two Lattices and Lattice of Constants



Constant propagation lattice



$|S_1 \times S_2| = |S_1| \times |S_2|$
$(a,b) \leq (c,d)$ iff $a \leq c$ & $b \leq d$

The Constant Propagation Framework

- The lattice of the DF values in the CP framework is the product of the semi-lattices of the variables (one lattice for each variable)
- In a product lattice, $(a_1, b_1) \leq (a_2, b_2)$ iff $a_1 \leq_A a_2$ and $b_1 \leq_B b_2$ assuming $a_1, a_2 \in A$ and $b_1, b_2 \in B$
- Each variable v is associated with a map m, and $m(v)$ is its abstract value (as in the lattice)
- Each element of the product lattice has a similar, but "larger" map m
  - Thus, $m \leq m'$ (in the product lattice), iff for all variables v, $m(v) \leq m'(v)$

Transfer Functions for the CP Framework

- Assume one statement per basic block
- Transfer functions for basic blocks containing many statements may be obtained by composition
- $m(v)$ is the abstract value of the variable v in a map m.
- The set F of the framework contains transfer functions which accept maps and produce maps as outputs
- F contains an identity map
- Map for the *Start* block is $m_0(v) = UNDEF$, for all variables v
- This is reasonable since all variables are undefined before a program begins

Transfer Functions for the CP Framework

- Let f_s be the transfer function of the statement s
- If $m' = f_s(m)$, then f_s is defined as follows
  1. If s is not an assignment, f_s is the identity function
  2. If s is an assignment to a variable x, then $m'(v) = m(v)$, for all $v \neq x$, and,
     (a) If the RHS of s is a constant c, then $m'(x) = c$
     (b) If the RHS is of the form $y + z$, then

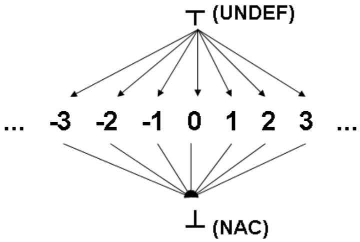
$$\begin{aligned} m'(x) &= m(y) + m(z),\ \textit{if } m(y) \textit{ and } m(z) \textit{ are constants} \\ &= NAC,\ \textit{if either } m(y) \textit{ or } m(z) \textit{ is } NAC \\ &= UNDEF,\ \textit{otherwise} \end{aligned}$$

     (c) If the RHS is any other expression, then $m'(x) = NAC$

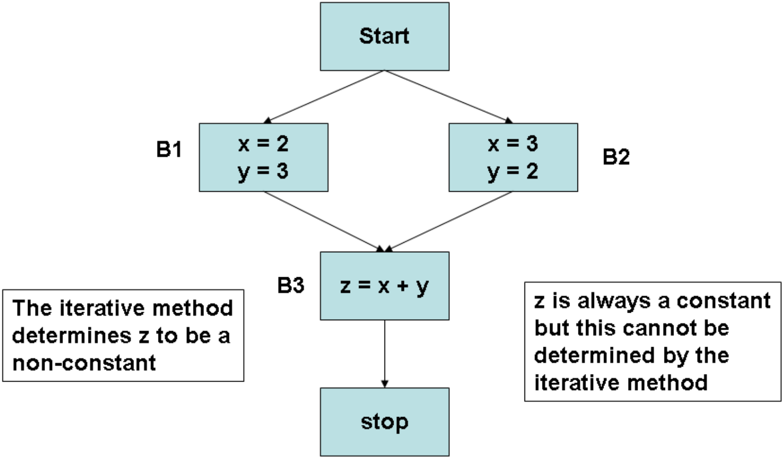
Monotonicity of the CP Framework

It must be noted that the transfer function ($m' = f_s(m)$) always produces a "lower" or same level value in the CP lattice, whenever there is a change in inputs

$m(y)$	$m(z)$	$m'(x)$
UNDEF	UNDEF	UNDEF
	c_2	UNDEF
	NAC	NAC
c_1	UNDEF	UNDEF
	c_2	$c_1 + c_2$
	NAC	NAC
NAC	UNDEF	NAC
	c_2	NAC
	NAC	NAC



Non-distributivity of the CP Framework



Non-distributivity of the CF Framework - Example

- If f_1, f_2, f_3 are transfer functions of $B1, B2, B3$ (resp.), then $f_3(f_1(m_0) \wedge f_2(m_0)) < f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$ as shown in the table, and therefore the CF framework is non-distributive

m	$m(x)$	$m(y)$	$m(z)$
m_0	UNDEF	UNDEF	UNDEF
$f_1(m_0)$	2	3	UNDEF
$f_2(m_0)$	3	2	UNDEF
$f_1(m_0) \wedge f_2(m_0)$	NAC	NAC	UNDEF
$f_3(f_1(m_0) \wedge f_2(m_0))$	NAC	NAC	NAC
$f_3(f_1(m_0))$	2	3	5
$f_3(f_2(m_0))$	3	2	5
$f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$	NAC	NAC	5

Introduction to Control-Flow Analysis

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Why control-flow analysis?
- Dominators and natural loops
- Depth of a control-flow graph

Why Control-Flow Analysis?

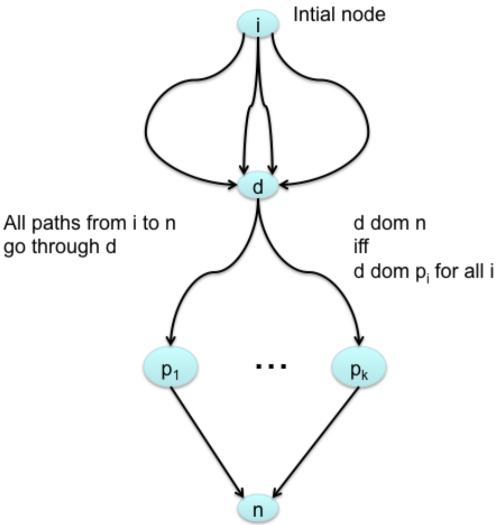
Control-flow analysis (CFA) helps us to understand the structure of control-flow graphs (CFG)

- To determine the loop structure of CFGs
- To compute dominators - useful for code motion
- To compute dominance frontiers - useful for the construction of the static single assignment form (SSA)
- To compute control dependence - needed in parallelization

Dominators

- We say that a node d in a flow graph *dominates* node n, written d *dom* n, if every path from the initial node of the flow graph to n goes through d
- Initial node is the root, and each node dominates only its descendents in the dominator tree (including itself)
- The node x *strictly dominates* y, if x dominates y and $x \neq y$
- x is the *immediate dominator* of y (denoted $idom(y)$), if x is the closest strict dominator of y
- A *dominator tree* shows all the immediate dominator relationships
- Principle of the dominator algorithm
  - If $p_1, p_2, ..., p_k$, are all the predecessors of n, and $d \neq n$, then d *dom* n, iff d *dom* p_i for each i

Dominator Algorithm Principle



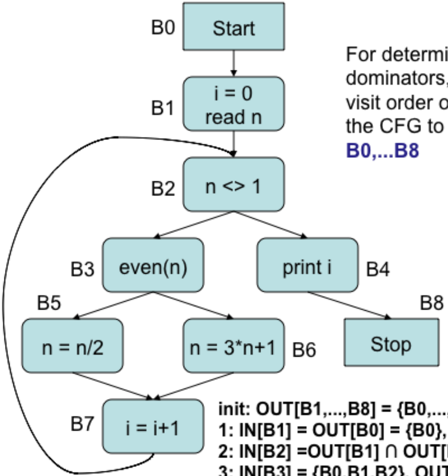
An Algorithm for finding Dominators

- $D(n) = OUT[n]$ for all n in N (the set of nodes in the flow graph), after the following algorithm terminates
- { /* n_0 = initial node; N = set of all nodes; */
  $OUT[n_0] = \{n_0\}$;
  for n in $N - \{n_0\}$ do $OUT[n] = N$;
  while (*changes to any* $OUT[n]$ *or* $IN[n]$ *occur*) do
  for n in $N - \{n_0\}$ do

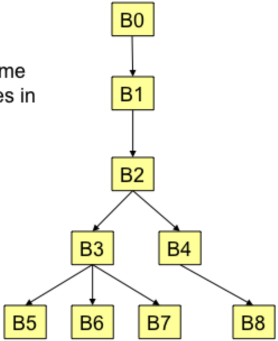
$$\begin{aligned} IN[n] &= \bigcap_{P\ a\ predecessor\ of\ n} OUT[P]; \\ OUT[n] &= \{n\} \cup IN[n] \end{aligned}$$

}

Dominator Example - 1

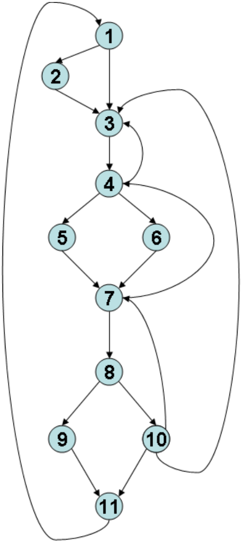


For determining dominators, assume visit order of nodes in the CFG to be **B0,...B8**

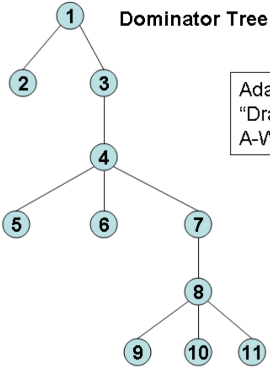


init: OUT[B1,...,B8] = {B0,...,B8}, OUT[B0] = {B0}
1: IN[B1] = OUT[B0] = {B0}, OUT[B1] = {B0,B1}
2: IN[B2] =OUT[B1] ∩ OUT[B7] = {B0,B1} , OUT[B2] = {B0,B1,B2}
3: IN[B3] = {B0,B1,B2}, OUT[B3] = {B0,B1,B2,B3}
IN[B4] = {B0,B1,B2}, OUT[B4] = {B0,B1,B2,B4} = IN[B8]
4: IN[B5] = {B0,B1,B2,B3} = IN[B6], OUT[B5] = {B0,B1,B2,B3,B5}
OUT[B6] = {B0,B1,B2,B3,B6}, OUT[B8] = {B0,B1,B2,B4,B8}
5: IN[B7] = OUT[B5] ∩ OUT[B6] = {B0,B1,B2,B3}
OUT[B7] = {B0,B1,B2,B3,B7}

Dominator Example - 2



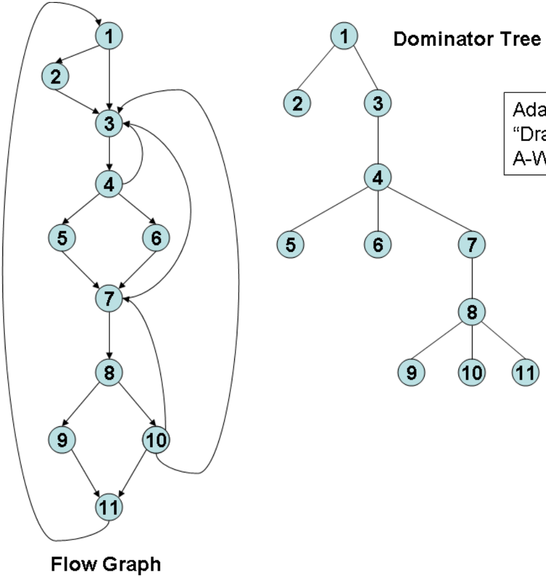
Flow Graph



Dominator Tree

Adapted from the "Dragon Book", A-W, 1986

Dominator Example - 3



Adapted from the "Dragon Book", A-W, 1986

Dominators and Natural Loops

- Edges whose heads dominate their tails are called *back edges* ($a \rightarrow b$: $b = head$, $a = tail$)
- Given a back edge $n \rightarrow d$
  - The *natural loop* of the edge is d plus the set of nodes that can reach n without going through d
  - d is the header of the loop
    - A single entry point to the loop that dominates all nodes in the loop
    - At least one path back to the header exists (so that the loop can be iterated)