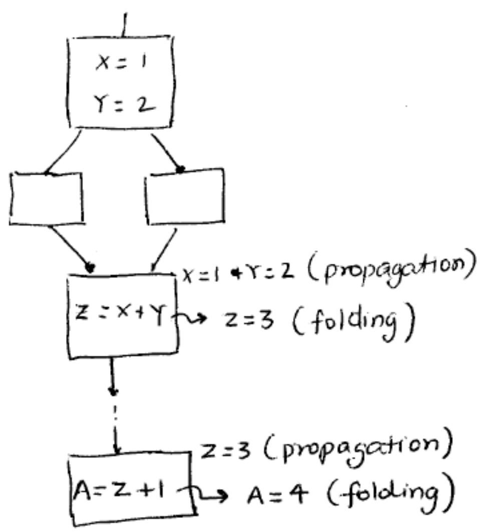
CS 201
Compiler Construction

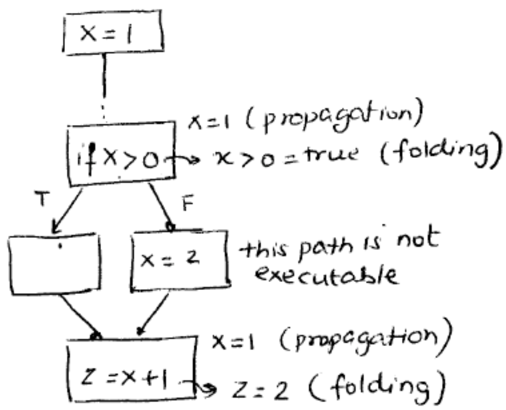
Lecture 6
Code Optimizations:
Constant Propagation & Folding

Constant Propagation & Folding



1. Use of var replaced by a constant value - propagation.
2. Exp. is evaluated or folded if all of its operands are constants.

Constant Propagation & Folding

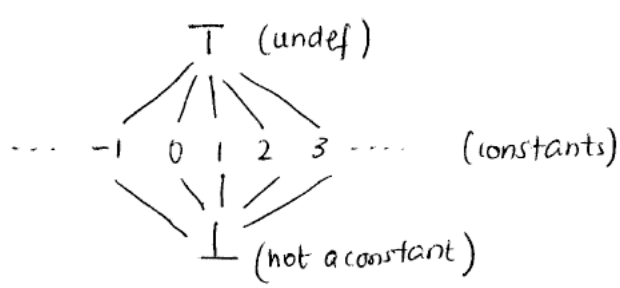


1. Evaluation of branch outcomes may be enabled.

2. Can take advantage of non-executable paths.

3

Data Flow Analysis



Constant Prop. Lattice for a Single variable.

4

Contd..

any $\wedge$ T = any
any $\wedge \perp = \perp$
c $\wedge$ c = c
c $\wedge$ d = $\perp$ ($c \neq d$)

Meet Operator for a Single variable.

5

Contd..

Lattice for all variables (L,∧')

VAR – set of variables
VAL – set of values (all constants, T, $\perp$)

L = {β: β is a total function β: VAR→VAL}
i.e., set of all states of variables.
forall v ε VAR
$\beta_1 \wedge' \beta_2(v) = \beta_1(v) \wedge \beta_2(v)$

β_T + $\beta_\perp$ are top + bottom elements of this new lattice
β_T maps each variable to T
$\beta_\perp$ maps each variable to $\perp$

6

Contd..

Transfer Function

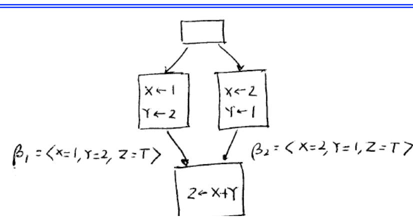
S: A = B op C

$f_S(\beta)$ – transfer function of S.

$$f_S(\beta) = \begin{cases} \beta(v)\ \forall v \in VAR - \{A\} & \text{(no change)} \\ \beta(A) = \bot & \text{if } \beta(B)=\bot \text{ or } \beta(C)=\bot \quad \text{(not a constant)} \\ \beta(A) = \top & \text{if } \beta(B)=\top \text{ or } \beta(C)=\top \quad \text{(still undefined)} \\ \beta(A) = c_i \text{ op } c_j & \text{if } \beta(B)=c_i \text{ and } \beta(C)=c_j \quad \text{(fold)} \end{cases}$$

7

Example



$\beta_1 \wedge' \beta_2 = \langle X=\bot, Y=\bot, Z=\top \rangle$

$f(\beta_1 \wedge' \beta_2) = \langle X=\bot, Y=\bot, Z=\bot \rangle \Rightarrow$ Z is not a constant

$f(\beta_1) = \langle X=1, Y=2, Z=3 \rangle$

$f(\beta_2) = \langle X=2, Y=1, Z=3 \rangle$

$f(\beta_1) \wedge' f(\beta_2) = \langle X=\bot, Y=\bot, Z=3 \rangle \Rightarrow$ Z is a constant

$\therefore f(\beta_1 \wedge' \beta_2) \leq f(\beta_1) \wedge' f(\beta_2)$ (not = but ≤)
not distributive,
just monotonic.

8

Worklist Algorithm

- Each node is a single statement
- IN and OUT contain the lattice values at node entry and exit
- Algorithm will take advantage of resolved branches. How ?
  - Mark edges as executable or non-executable and propagate information only along executable edges
  - Branch (true) (false) (true or false) – mark edges executable accordingly

9

Algorithm: Initialization

Mark all edges as unexecutable

Worklist $\leftarrow n_0$ (initial/start node in the CFG)

$IN[n_0] = \beta_{\perp}$ (none of the variables is a constant)

$\forall n \in N - \{n_0\}$, $IN[n] = OUT[n] = \beta_{T}$ (undefined)

10

Algorithm: Analysis

```
While Worklist ≠ ∅ do
    get n from Worklist
    IN[n] = ∧' OUT[p]  st  p→n is executable
           p∈Pred(n)
    if n is an assignment statement then
        n: A = B op C
        OUT[n] = f_n (IN[n])
        mark outgoing edges as executable
        if OUT[n] changed then add Succ(n) to Worklist
    else (n is a predicate/condition)
        evaluate condition & its value is
            true or false or not a constant
        OUT[n] = f_n (IN[n])
        Mark appropriate outgoing edges as executable
          (true - only true edge, false - only false edge
           not a constant - mark both true & false edges)
        if OUT[n] changed add Succ(n) to Worklist
    endif
endwhile
```

11