CS 201
Compiler Construction

Lecture 3
Data Flow Analysis

1

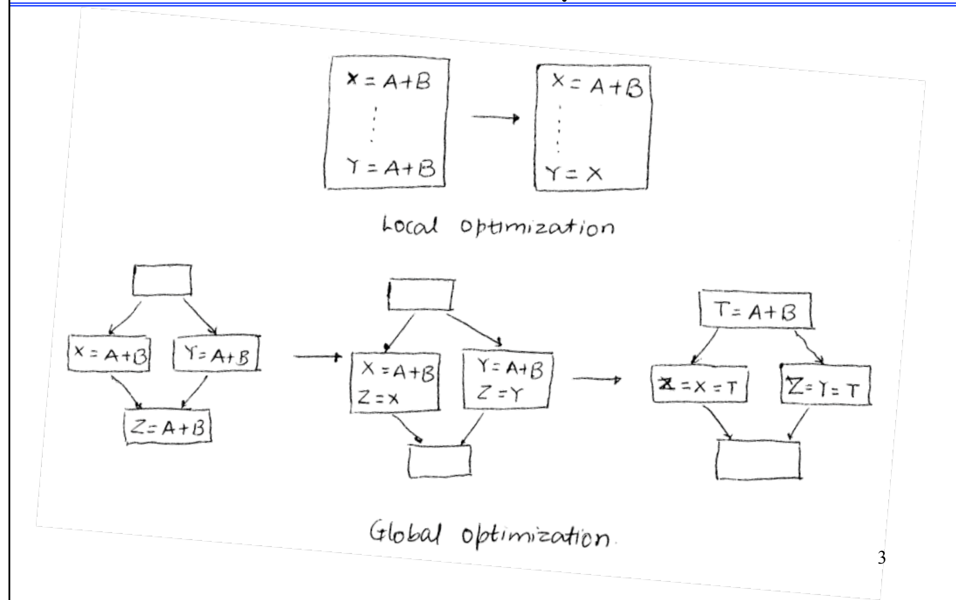
Data Flow Analysis

Data flow analysis is used to collect information about the flow of data values across basic blocks.

- Dominator analysis collected global information regarding the program's structure
- For performing global code optimizations global information must be collected regarding values of program values.
  - Local optimizations involve statements from same basic block
  - Global optimizations involve statements from different basic blocks → data flow analysis is performed to collect global information that drives global optimizations

2

Local and Global Optimization



3

Applications of Data Flow Analysis

- Applicability of code optimizations
- Symbolic debugging of code
- Static error checking
- Type inference
- …….

4

1. Reaching Definitions

Definition d of variable v: a statement d that assigns a value to v.

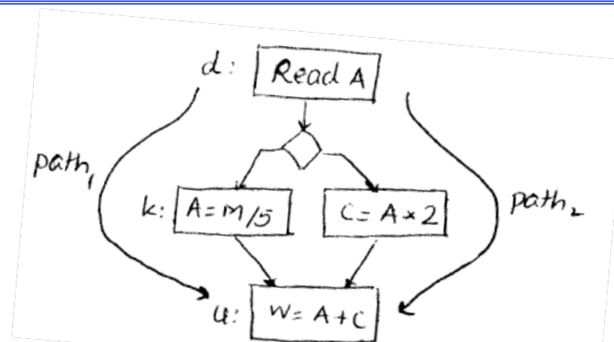
Use of variable v: reference to value of v in an expression evaluation.

Definition d of variable v reaches a point p if there exists a path from immediately after d to p such that definition d is not killed along the path.

Definition d is killed along a path between two points if there exists an assignment to variable v along the path.

5

Example



d reaches u along $path_2$ & d does not reach u along $path_1$

Since there exists a path from d to u along which d is not killed (i.e., $path_2$), d reaches u.

6

Reaching Definitions Contd.

Unambiguous Definition: X = ….;
Ambiguous Definition: *p = ….; p may point to X

For computing reaching definitions, typically we only consider kills by unambiguous definitions.

X=..

*p=..

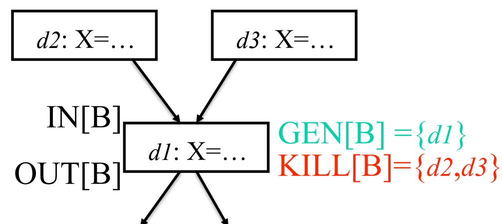
Does definition of X reach here ? Yes

7

Computing Reaching Definitions

At each program point p, we compute the set of definitions that reach point p.

Reaching definitions are computed by solving a system of equations (data flow equations).



8

Data Flow Equations

IN[B]: Definitions that reach B's entry.
OUT[B]: Definitions that reach B's exit.

$$IN[B] = \bigcup_{p \in pred(B)} OUT(p)$$

$$OUT(B) = GEN(B) \cup (IN(B) - KILL(B))$$

GEN[B]: Definitions within B that reach the end of B.
KILL[B]: Definitions that never reach the end of B due to redefinitions of variables in B.

9

Reaching Definitions Contd.

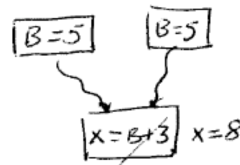
- Forward problem – information flows forward in the direction of edges.
- May problem – there is a path along which definition reaches a point but it does not always reach the point.

  Therefore in a May problem the meet operator is the Union operator.

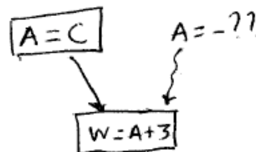
10

Applications of Reaching Definitions

- Constant Propagation/ folding



- Copy Propagation



11

2. Available Expressions

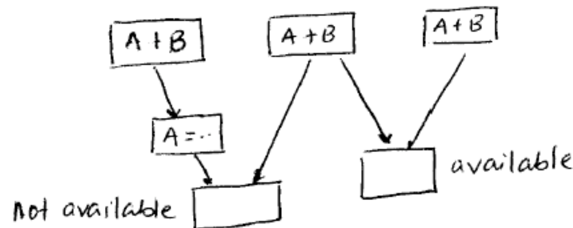
An expression is generated at a point if it is computed at that point.

An expression is killed by redefinitions of operands of the expression.

An expression A+B is available at a point if every path from the start node to the point evaluates A+B and after the last evaluation of A+B on each path there is no redefinition of either A or B (i.e., A+B is not killed).

12

Available Expressions



Available expressions problem computes: at each program point the set of expressions available at that point.

Data Flow Equations

IN[B]: Expressions available at B's entry.
OUT[B]: Expressions available at B's exit.

$$IN[B] = \bigcap_{p \in pred(B)} OUT(p)$$

$$OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$$

GEN[B]: Expressions computed within B that are available at the end of B.
KILL[B]: Expressions whose operands are redefined in B.

Available Expressions Contd.

- Forward problem – information flows forward in the direction of edges.
- Must problem – expression is definitely available at a point along all paths.

  Therefore in a Must problem the meet operator is the Intersection operator.
- Application:

X = A+B
Y = A+B
Z = A+B
A+B is available here
T = X = A+B
T = Y = A+B
Z = T

15

3. Live Variable Analysis

A path is X-clear is it contains no definition of X.

A variable X is live at point p if there exists a X-clear path from p to a use of X; otherwise X is dead at p.

X = . .
X is live
.. = X
X is dead
Read(x)
X is live
.. = X
.. = X
X is dead

Live Variable Analysis Computes:
At each program point p identify the set of variables that are live at p.

16

Data Flow Equations

IN[B]: Variables live at B's entry.
OUT[B]: Variables live at B's exit.

$$OUT[B] = \bigcup_{S \in Succ(B)} IN[S]$$

$$IN[B] = GEN[B] \cup (OUT[B] - KILL[B])$$

GEN[B]: Variables that are used in B prior to their definition in B.
KILL[B]: Variables definitely assigned value in B before any use of that variable in B.

17

Live Variables Contd.

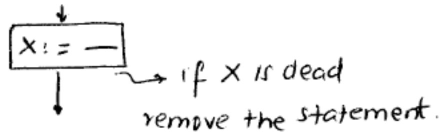
- Backward problem – information flows backward in reverse of the direction of edges.
- May problem – there exists a path along which a use is encountered.

  Therefore in a May problem the meet operator is the Union operator.

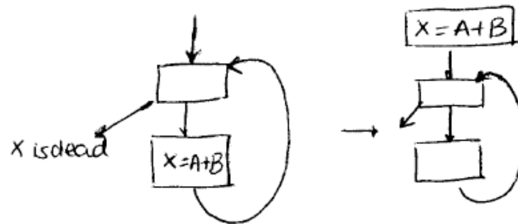
18

Applications of Live Variables

- Register Allocation
- Dead Code Elimination

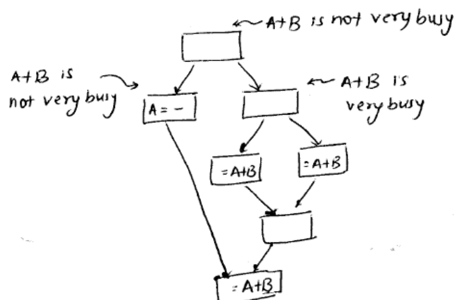


- Code Motion Out of Loops

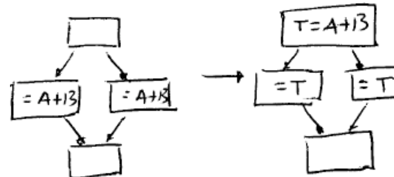


4. Very Busy Expressions

A expression A+B is very busy at point p if for all paths starting at p and ending at the end of the program, an evaluation of A+B appears before any definition of A or B.



Application:
Code Size Reduction



Compute for each program point the set of very busy expressions at the point.

Data Flow Equations

IN[B]: Expressions very busy at B's entry.
OUT[B]: Expressions very busy at B's exit.

$$OUT[B] = \bigcap_{S \in Succ(B)} IN[S]$$

$$IN[B] = GEN[B] \cup (OUT[B] - KILL[B])$$

GEN[B]: Expression computed in B and variables used in the expression are not redefined in B prior to expression's evaluation in B.
KILL[B]: Expressions that use variables that are redefined in B.

21

Very Busy Expressions Contd.

- Backward problem – information flows backward in reverse of the direction of edges.
- Must problem – expressions must be computed along all paths.

  Therefore in a Must problem the meet operator is the Intersection operator.

22

Summary

	May/Union	Must/ Intersection
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Very Busy Expressions

23

Conservative Analysis

Optimizations that we apply must be Safe => the data flow facts we compute should definitely be true (not simply possibly true).

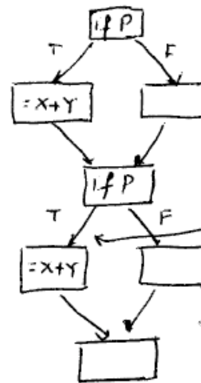
Two main reasons that cause results of analysis to be conservative:

1. Control Flow
2. Pointers & Aliasing

24

Conservative Analysis

1. Control Flow – we assume that all paths are executable; however, some may be infeasible.



X+Y is always available if we exclude infeasible paths.

25

Conservative Analysis

2. Pointers & Aliasing – we may not know what a pointer points to.

1. X = 5
2. *p = … // p may or may not point to X
3. … = X

Constant propagation: assume p does point to X (i.e., in statement 3, X cannot be replaced by 5).

Dead Code Elimination: assume p does not point to X (i.e., statement 1 cannot be deleted).

26

Representation of Data Flow Sets

- Bit vectors – used to represent sets because we are computing binary information.
  - Does a definition reach a point ? T or F
  - Is an expression available/very busy ? T or F
  - Is a variable live ? T or F
- For each expression, variable, definition we have one bit – intersection and union operations can be implemented using bitwise and & or operations.

27

Solving Data Flow Equations

Iterative approach

- initialize sets
- iterate over the sets till they stabilize

Example: Forward problem (Available Expressions)

```
IN[B0] = ∅ ;  OUT[B0] = GEN[B0]
For i = 1 to N do  OUT[Bi] = {all expressions} - KILL[Bi]
change = true
while change do
    change = false
    for each block B ≠ B0 do
        OLDOUT = OUT[B]
        IN[B] = ∩ OUT[P]
              P ∈ pred(B)
        OUT[B] = GEN[B] ∪ (IN[B] - KILL[B])
        IF OUT[B] ≠ OLDOUT then change = true
    endfor
endwhile
```

$\cap$ – start with largest estimate & iteratively shrink the solution till it stabalizes.

28

Solving Data Flow Equations

Iterative Approach

Example - backward problem (live variables)

```
for i = 1 to N do   IN[Bi] = GEN[Bi]
OUT[Bexit] = ∅
change = true
while change do
    change = false
    for each block B do
        OLDIN = IN[B]
        OUT[B] =     U      IN[S]
                 S∈succ(B)
        IN[B] = GEN[B] U (OUT[B] - KILL[B])
        If OLDIN ≠ IN[B] then change = true
    end for
endwhile
```

U - start with smallest solution & keep expanding till it stops expanding.

29

Solving Data Flow Equations

Alternative Approach - Worklist Algorithm

Example - backward problem (very busy expressions)

```
for i=1 to N do   IN[Bi] = {All expressions} - KILL[Bi]
OUT[Bexit] = ∅
Worklist ← All blocks
while Worklist ≠ ∅ do
        get B from Worklist
        OLDIN = IN[B]
        OUT[B] =     ∩      IN[S]
                 S∈succ(B)
        IN[B] = GEN(B) U (OUT[B] - KILL[B])
        If OLDIN ≠ IN[B] then
              Add Pred(B) to Worklist
endwhile
```

∩ - start with largest solution & keep iterating till it stops shrinking.

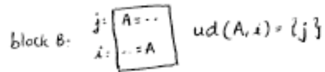
30

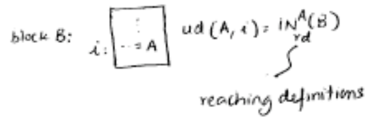
Use-Def & Def-Use Chains

Directly link instructions that produce values with instructions that consume values.

use-def

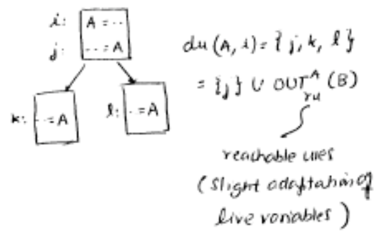
ud chain for some variable use u is the list of pointers to all definitions of the variable that reach u.





def-use

du chain for some variable definition d is the list of pointers to all uses of the variable that are reachable from d.



31