Module 15: Reaching Definition

Lecture 30: Analysis

The Lecture Contains:

- Interval Analysis
- Backward Analysis
- Available Expression
- Live Variable Analysis
- Very Busy Expression
- Common Sub-expression Elimination
- Copy Propagation
- Loop Invariant Computations
- Performing Code Motion
- Elimination of Induction Variable
- Detection of Induction Variables
- Strength Reduction
- Pointers
- A Simple Pointer Language
- Transfer Function

Previous Next

Module 15: Reaching Definition

Lecture 30: Analysis

For B_{3a} block

$$in(B_3) = in(B_{3a})$$

$$in(B_{4a}) = F_{B_3}\ (in(B_3))$$

$$in(B_5) = F_{B_{4a}}\ (in(B_{4a}))$$

For the while loop

$$in(B_4) = \left(F_{B_6} \text{ o } F_{B_4}\right) * (in(B_{4a}))$$

$$= (id \sqcap \left(F_{B_6} \text{ o } F_{B_4}\right))\ (in(B_{4a}))$$

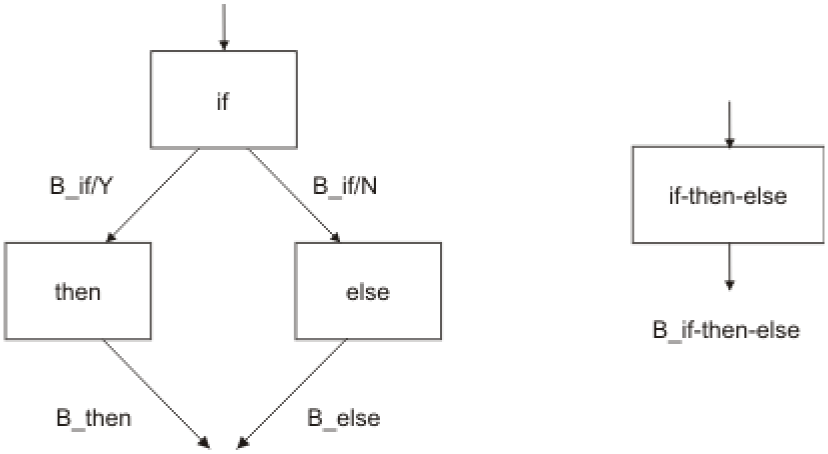
$$in(B_6) = F_{B_4}(in(B_4))$$

Interval Analysis

- Interval analysis is trivial; it is identical to structural analysis.
- Only three kinds of regions appear: general acyclic, proper, and improper.

Backward Analysis

- Harder to model as single exit is not guaranteed in programs
- For constructs with single exit we can 'turn the equations around'



Bottom up equation:

$$B_{if} - then - else = (B_{if}/Y \circ B_{then}) \sqcap (B_{if}/N \circ B_{else})$$

Top down equation:

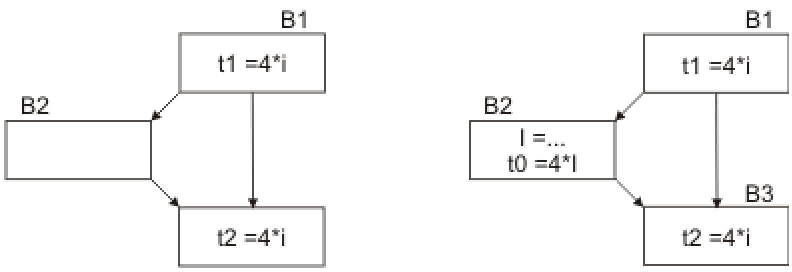
Out (then) = out (if –then –else)

Out (else) = out (if-then-else)

Out (if) = $B_{then}(out(then)) \sqcap B_{else}(out(else))$

Available Expression

- Used in detecting common subexpressions



- Expression 4*i in B_3 is a common subexpression if 4*i is available the entry point of B_3
- It will be available if i is not assigned a value in B_2 or 4*i is re-computed after i is assigned in B_2
- How to compute set of generated expressions:
    - At a point prior to block no expressions are available
    - If at a point p set A of expressions is available and q is a point after p with statement x:=y+z then set of expressions available at q is:
        - Add to A expression y+z
        - Delete from A any expression involving x

Previous Next

Example

Statement	Available expression
a:=b+c	
	b+c
b:=a-d	
	a-d
c:=b+c	
	a-d
d:=a-d	
	none

- U is the universal set of all expressions appearing in a program
- In[B] and out[B] are sets of expressions available at the beginning/end of B
- E-gen[B] and e-kill[B] are sets of expressions generated and killed in B
- Out[B] = in[B] - e-kill[B] $\cup$ e-gen[B]

  In[B] = $\bigcap$ out[P] for B not initial

  In[B] = $\emptyset$ where B_0 is the initial block
- Initialization:

  In[B_0] = $\emptyset$

  Out[B_0] = e-gen[B_0]

  Out[B] = U - e-kill[B] if B is not an entry block

Live Variable Analysis

- Used for dead code elimination
- In[B] and out[B] are sets of live variables at entry and exit
- Def[B] set of variables assigned value in B prior to use in B
- Use[B] set of variables whose value may be used before definition in B
- In[B] = use[B] S (out[B] - def[B])

  Out[B] = S in[S] where S is successor of B
- A variable is live coming into a block if EITHER is it used in the block before re-definition OR it is live coming out and not re-defined
- A variable is live coming out of a block if it is live coming into one of its successors
- Initialization:

  in[B]= $\emptyset$ for all B

Previous Next

Very Busy Expression

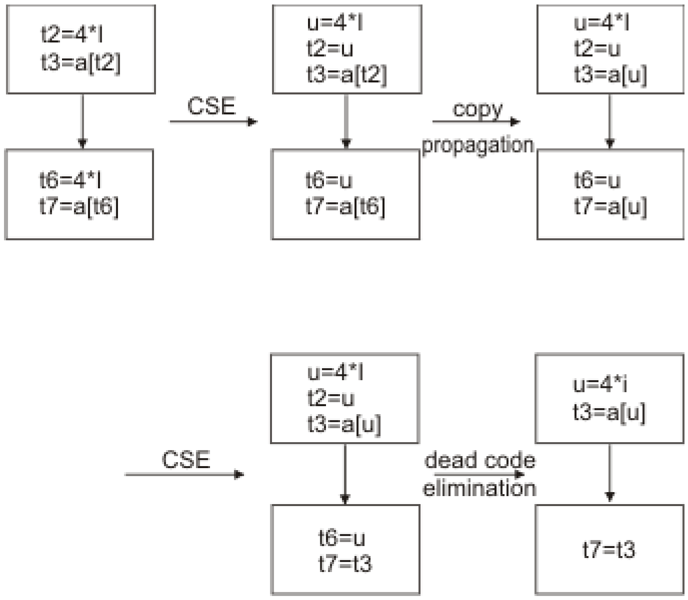
- In[B] and Out[B] are sets of VBE at the beginning and end of B
- Use[B] set of expressions b+c computed in B with no prior definition of b or c
- Def[B] set of expression b+c for which either b or c is defined in block B prior to computation of b+c
- In[B] = out[B] - def[B] $\cup$ use[B]

  out[B] = $\cap$ in[S] where S is successor of B
- An expression is VBE coming into a block if either it is used in B or it is live coming out and not defined in B
- An expression is VBE coming out of a block if it is live going into all the successors of B
- Initialization:
  in[B] = U for all B

Common Sub-expression Elimination

For every statement s of the form x=y+z such that y+z is available at the beginning of the block and y and z are not re-defined
prior to s

1. Find all definitions which have y+z that reach s' block
2. Create a new variable u
3. Replace each w=y+z found in (1) by
   u=y+z; w=u
4. Replace statement s by x=u



Module 15: Reaching Definition

Lecture 30: Analysis

Copy Propagation

Assignment s:x=y may be eliminated if at all the places where x is used we replace x by y

- Statement s must be the only definition of x reaching where substitution is to be made
- On every path from s to target there are no assignments to y (additional data flow analysis needs to be done)

Algorithm: for each copy s:x=y do the following:

1. Determine those uses of x that are reached by this definition
2. Determine whether it is the only definition of x reaching and there is no definition of y on the path
3. If s meets the above conditions then remove s and replace all uses of x found in (1) by y

Loop Invariant Computations

If for an assignment x=y+z all the definitions of y and z are outside loop then x=y+z is invariant of loop.

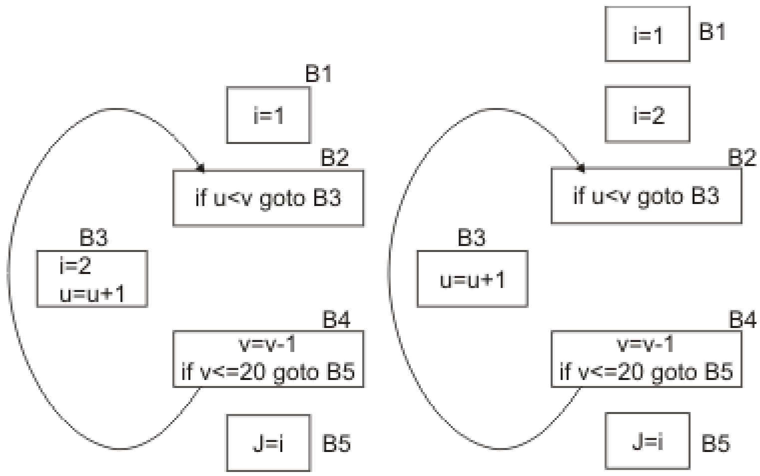
Input: A loop L with basic blocks. Assume that ud chains are available for individual statements.

1. Mark invariant statements whose operands are all either constants or or have their reaching definitions outside L
2. Repeat step (3) until no new statements are marked invariant
3. Mark invariant whose operands either are constant, have all their reaching definitions outside L, or have exactly one reaching definition and that definition is a statement in L marked invariant

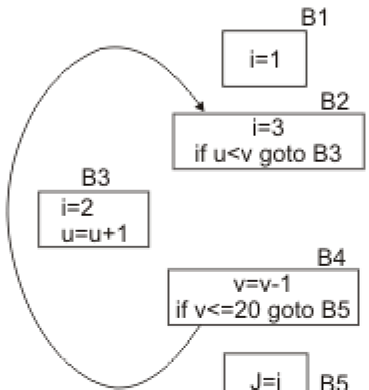
Performing Code Motion

- Move an invariant statement s to pre-header if following conditions are met:
    1. The block containing s dominates all exit nodes of the nodes
    2. There is no other statement in the loop that assign to x
    3. No use of x in the loop is reached by any definition of x other than s
- Maintaining dataflow information
    1. Ud chains: does not change by code motion
    2. Dominator information: changes by code motion; it needs to be recomputed.
- More general code motion:
  If none of the three conditions are satisfied then for a loop invariant statement
  A=B+C define T=B+C in the pre header and replace A=B+C by A=T

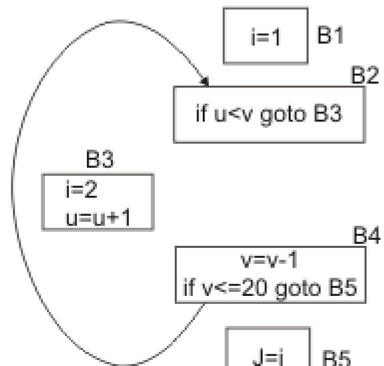
Previous Next



Example of condition 1: Illegal code motion



Example of condition 2



Example of condition 3

Elimination of Induction Variable

- A variable X is called induction variable of a loop if in every iteration value of X is changed by a constant value.
- Basic induction variables: as defined i=i ± c
- Secondary induction variable: a basic function of basic induction variable

Previous Next

Module 15: Reaching Definition

Lecture 30: Analysis

Detection of Induction Variables

Input: A loop L with reaching definition information and loop invariant computation information
Output: A set of induction variables. Associated with each induction variable j is a triple (i, c, d) such that j=c*i+d. i is assumed to be basic induction variable, and j is said to belong to family of i.

1. Find all basic induction variables of L (using loop invariant information). Each basic induction variable has a triple (i, 1, 0).
2. Search for variable k with single assignment to k within L having one of the following forms:
   k=j*const, k=j/const, k=j ± const
   where j is an induction variable.

1. If j is basic induction variable then k is in family of j. if j is not basic and is in family of i then
   - There is no assignment to i between j and k
   - No definition of j outside L reaches k
2. Modify instructions computing induction variable such that ± are used rather than multiplication (strength reduction).

Strength Reduction

Consider each basic induction variable. For every induction variable j in family of i with triple (i, c, d)

1. Create a new variable s
2. Replace all assignments to j by j=s
3. Immediately after each assignment i=i+n append s=s+c*n
   Place s in the family of i with triple (i,c,d)
4. Initialize s to s=c*i+d in the pre-header
   Eliminate induction variables

Pointers

A := B + C
*P := D
F := *P
E := B+C

No definitions of B or C.
Is B+C available at E := B+C
depends whether *P changes B or C

Safe Assumption : Indirect assignment can change any variable, indirect use can use any name
Therefore,

- More live variable and reaching definitions than realistic.
- Fewer available expressions than realistic

Previous Next

Module 15: Reaching Definition

Lecture 30: Analysis

A Simple Pointer Language

The language consists of

- Elementary data types (integers and reals) requiring one word each
- Array of these types
- Pointer is used as cursor to run through an array
- Pointer p points to an element of an array
- Variables that could be used as pointers are those declared to be pointers and temporaries that received a value that is pointer plus or minus a constant
- If there is a statement s: p:=&a then after s, p points to a
- If there is a statement s: p:= q ± c where c is an integer, and p and q are pointers then after s, p points to an array that q could point to before s
- If there is a statement s: p:=q then after s, p points to whatever q could point to before s
- In[B] is a set of pairs (p,a) where p is a pointer and a is a variable
- Out[B] is defined in a similar manner for set of values after a block B

Transfer Function

- If s is p *: =* $\& a \ or \ p := \&a = \pm c \ in \ the \ case \ a \ is \ an \ array, then$

  $trans_s(s) = (s - \{(\mathrm{p}, \mathrm{b}) | \text{any variable b}\})$

  $\cup \{(\mathrm{p}, \mathrm{a})\}$

- If s is p: $\pm$ c for pointer q and nonzero integer c then

  $\text{trans}_s(s) = (s - \{\mathrm{p}, \mathrm{b})|\text{any varible b}\})$

  $\cup \{(\mathrm{p}, \mathrm{b}) | (\mathrm{q}, \mathrm{b}) \text{ is in S and b is in an}$ Array variable}
- If s is p: = q then

  $\text{trans}_s(S) = (S - \{(\mathrm{p}, \mathrm{b}) | \text{any varible b}\})$

  $\cup \{(\mathrm{p}, \mathrm{b}) | (\mathrm{q}, \mathrm{b}) \text{is in S}\}$
- If s assign to pointer p any other expression then

  $trans_s(S) = (S - \{(\mathrm{p}, \mathrm{b}) | \text{any variable b}\})$
- If s is not an assignment to a pointer then

  $trans_s(S) = S$

Previous Next