Bit Vector Data Flow Frameworks

Uday Khedker

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

Jul 2010

Part 1

About These Slides

Copyright

These slides constitute the lecture notes for CS618 Program Analysis course at IIT Bombay and have been made available as teaching material accompanying the book:

- Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press (Taylor and Francis Group). 2009.

Apart from the above book, some slides are based on the material from the following books

- M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland Inc. 1977.
- F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag. 1998.

Outline

- Live Variables Analysis
- Program Execution Model and Semantics
- Soundness of Data Flow Analysis
- Available Expressions Analysis
- Anticipable Expressions Analysis
- Reaching Definitions Analysis
- Common Features of Bit Vector Frameworks
- Partial Redundancy Elimination

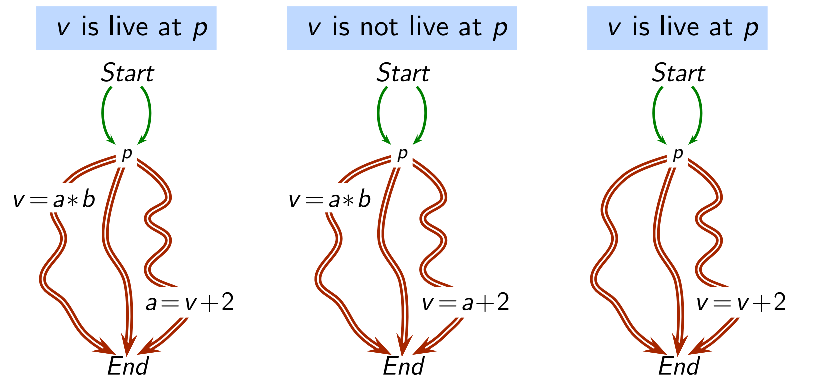
Part 2

Live Variables Analysis

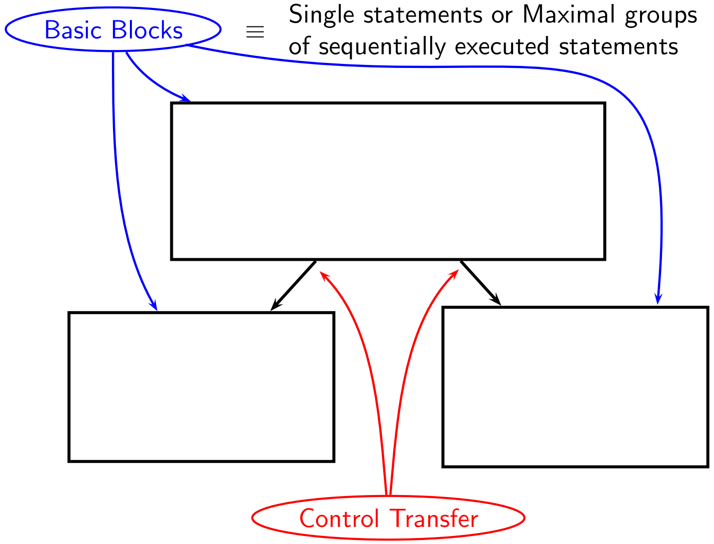
Defining Live Variables Analysis

A variable v is live at a program point p, if some path from p to program exit contains an r-value occurrence of v which is not preceded by an l-value occurrence of v.

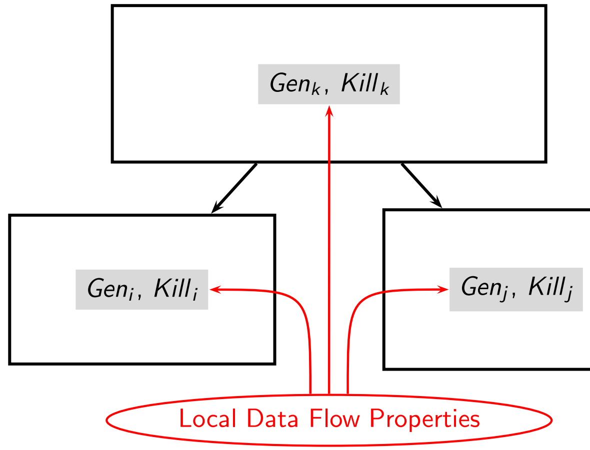
Path based specification



Defining Data Flow Analysis for Live Variables Analysis



Defining Data Flow Analysis for Live Variables Analysis



Local Data Flow Properties for Live Variables Analysis

r-value occurrence

Value is only read, e.g. x,y,z in

x.sum = y.data + z.data

l-value occurrence

Value is modified e.g. y in

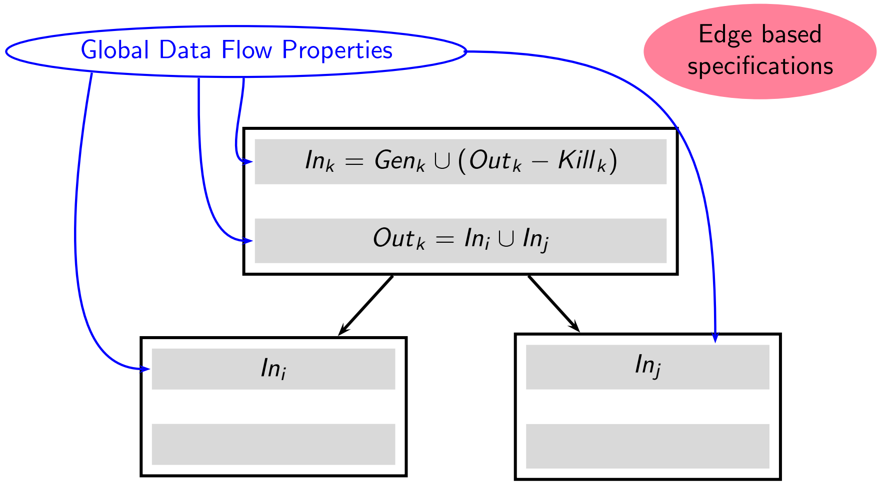
y = x.lptr

$$\begin{aligned} Gen_n &= \{\ v \mid \text{variable } v \text{ is used in basic block } n \text{ and} \\ &\qquad \text{is not preceded by a definition of } v\ \} \\ Kill_n &= \{\ v \mid \text{basic block } n \text{ contains a definition of } v\ \} \end{aligned}$$

within n

anywhere in n

Defining Data Flow Analysis for Live Variables Analysis



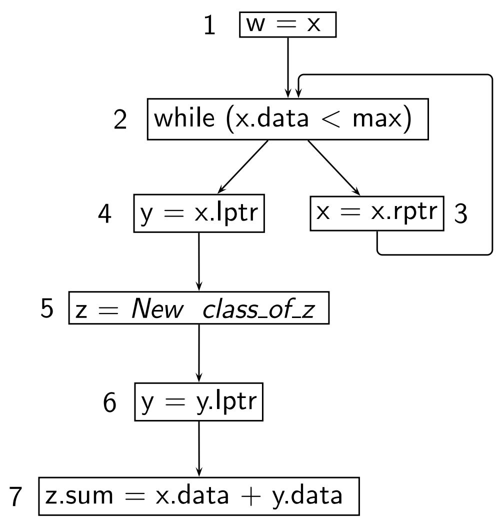
Data Flow Equations For Live Variables Analysis

$$In_n = (Out_n - Kill_n) \cup Gen_n$$

$$Out_n = \begin{cases} BI & n \text{ is } End \text{ block} \\ \bigcup\limits_{s \in succ(n)} In_s & \text{otherwise} \end{cases}$$

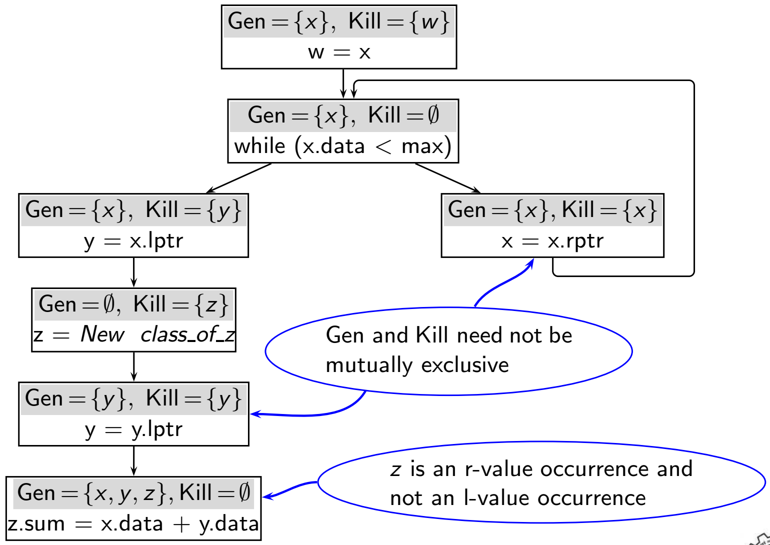
In_n and Out_n are sets of variables.

Data Flow Equations for Our Example

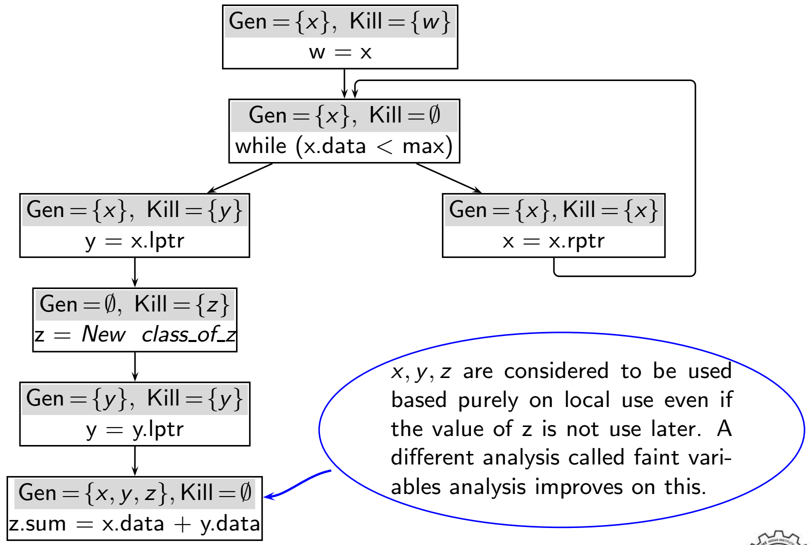


$$\begin{aligned} In_1 &= (Out_1 - Kill_1) \cup Gen_1 \\ Out_1 &= In_2 \\ In_2 &= (Out_2 - Kill_2) \cup Gen_2 \\ Out_2 &= In_3 \cup In_4 \\ In_3 &= (Out_3 - Kill_3) \cup Gen_3 \\ Out_3 &= In_2 \\ In_4 &= (Out_4 - Kill_4) \cup Gen_4 \\ Out_4 &= In_5 \\ In_5 &= (Out_5 - Kill_5) \cup Gen_5 \\ Out_5 &= In_6 \\ In_6 &= (Out_6 - Kill_6) \cup Gen_6 \\ Out_6 &= In_7 \\ In_7 &= (Out_7 - Kill_7) \cup Gen_7 \\ Out_7 &= In_7 \end{aligned}$$

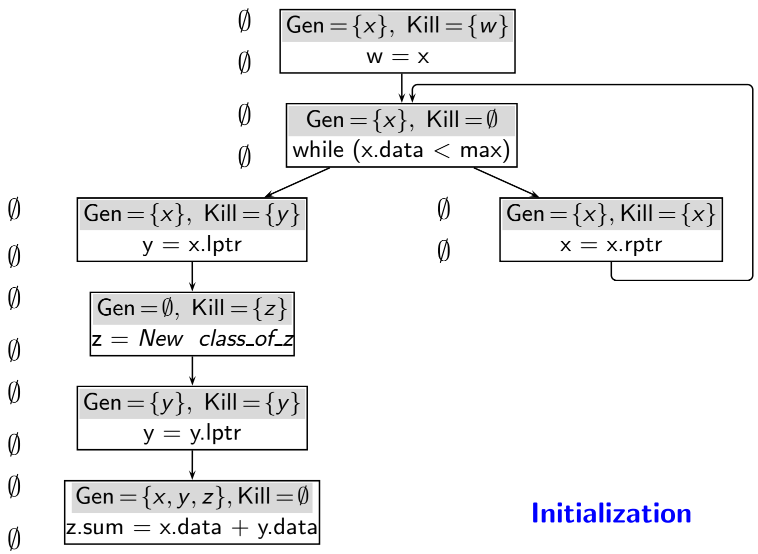
Performing Live Variables Analysis



Performing Live Variables Analysis

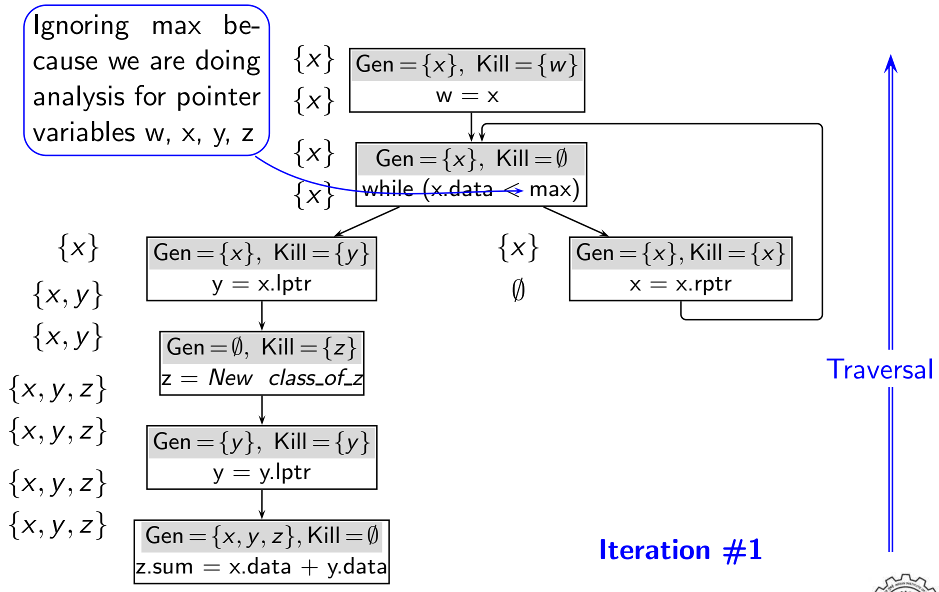


Performing Live Variables Analysis



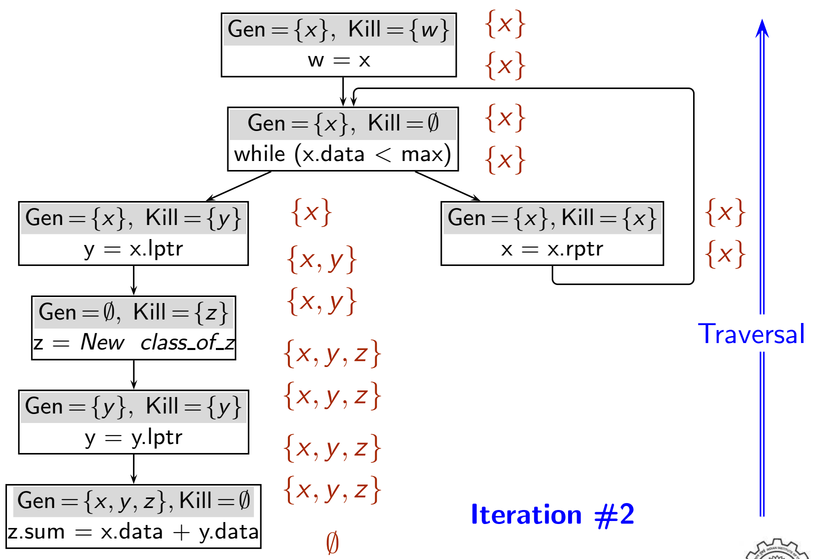
Initialization

Performing Live Variables Analysis

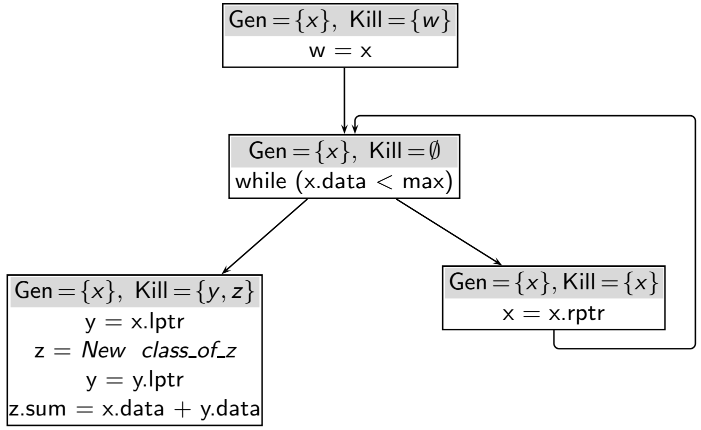


Iteration #1

Performing Live Variables Analysis



Performing Live Variables Analysis



Local Data Flow Properties for Live Variables Analysis

$$In_n = (Out_n - Kill_n) \cup Gen_n$$

- Gen_n : Use not preceded by definition

  Upwards exposed use

- $Kill_n$: Definition anywhere in a block

  Stop the effect from being propagated across a block

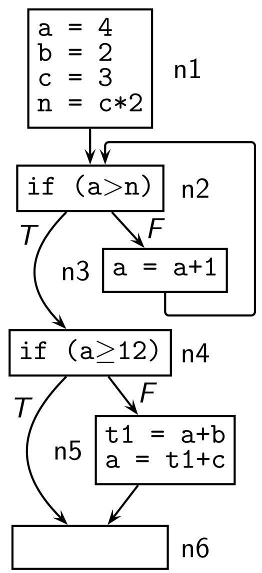
Local Data Flow Properties for Live Variables Analysis

Case	Local Information		Example	Explanation
1	$v \notin Gen_n$	$v \notin Kill_n$	$a = b + c$ $b = c * d$	liveness of v is unaffected by the basic block
2	$v \in Gen_n$	$v \notin Kill_n$	$a = b + c$ $b = v * d$	v becomes live before the basic block
3	$v \notin Gen_n$	$v \in Kill_n$	$a = b + c$ $v = c * d$	v ceases to be live before the statement
4	$v \in Gen_n$	$v \in Kill_n$	$a = v + c$ $v = c * d$	liveness of v is killed but v becomes live before the statement

Using Data Flow Information of Live Variables Analysis

- Used for register allocation.
  If variable x is live in a basic block b, it is a potential candidate for register allocation.
- Used for dead code elimination.
  If variable x is not live after an assignment $x = \ldots$, then the assginment is redundant and can be deleted as dead code.

Tutorial Problem 1 for Liveness Analysis



Local Data Flow Information		
	Gen	*Kill*
n1	$\emptyset$	$\{a, b, c, n\}$
n2	$\{a, n\}$	$\emptyset$
n3	$\{a\}$	$\{a\}$
n4	$\{a\}$	$\emptyset$
n5	$\{a, b, c\}$	$\{a, t1\}$
n6	$\emptyset$	$\emptyset$

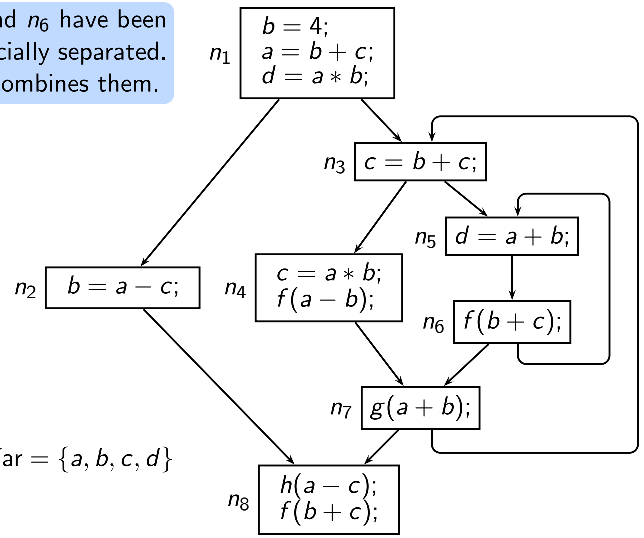
Global Data Flow Information				
	Iteration #1		Iteration #2	
	Out	*In*	*Out*	*In*
n6	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
n5	$\emptyset$	$\{a, b, c\}$	$\emptyset$	$\{a, b, c\}$
n4	$\{a, b, c\}$	$\{a, b, c\}$	$\{a, b, c\}$	$\{a, b, c\}$
n3	$\emptyset$	$\{a\}$	$\{a, b, c\}$	$\{a, b, c\}$
n2	$\{a, b, c\}$	$\{a, b, c, n\}$	$\{a, b, c, n\}$	$\{a, b, c, n\}$
n1	$\{a, b, c, n\}$	$\emptyset$	$\{a, b, c, n\}$	$\emptyset$

Tutorial Problem 2 for Liveness Analysis: C Program

```
int x, y, z;
int exmp(void)
{ int a, b, c, d;
   b = 4;
   a = b + c;
   d = a * b;
   if (x < y)
      b = a -c;
   else
   { do
      { c = b + c;
        if (y > x)
        { do
           { d = a + b;
             f(b + c);
           } while(y > x);
        }
        else
        { c = a * b;
           f(a - b);
        }
        g (a + b);
      } while(z > x);
   }
   h(a-c);
   f(b+c);
}
```


Tutorial Problem 2 for Liveness Analysis: Control Flow Graph

n_5 and n_6 have been artificially separated. gcc combines them.



$\mathbb{V}\text{ar} = \{a, b, c, d\}$

Solution of the Tutorial Problem

Block	Local Information		Global Information			
			Iteration # 1		Iteration # 2	
	Gen_n	$Kill_n$	Out_n	In_n	Out_n	In_n
n_8	$\{a,b,c\}$	$\emptyset$	$\emptyset$	$\{a,b,c\}$	$\emptyset$	$\{a,b,c\}$
n_7	$\{a,b\}$	$\emptyset$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$
n_6	$\{b,c\}$	$\emptyset$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$
n_5	$\{a,b\}$	$\{d\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$
n_4	$\{a,b\}$	$\{c\}$	$\{a,b,c\}$	$\{a,b\}$	$\{a,b,c\}$	$\{a,b\}$
n_3	$\{b,c\}$	$\{c\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$
n_2	$\{a,c\}$	$\{b\}$	$\{a,b,c\}$	$\{a,c\}$	$\{a,b,c\}$	$\{a,c\}$
n_1	$\{c\}$	$\{a,b,d\}$	$\{a,b,c\}$	$\{c\}$	$\{a,b,c\}$	$\{c\}$

Tutorial Problems for Liveness Analysis

- Perform analysis with universal set $\mathbb{V}$ar as the initialization at internal nodes.
- Modify the previous program so that some data flow value computed in second iteration differs from the corresponding data flow value computed in the first iteration.
  (No structural changes, suggest at least two distinct kinds of modifications)
- Modify the above program so that some data flow value computed in third iteration differs from the corresponding data flow value computed in the second iteration.
  Write a C program corresponding to the modified control flow graph

Part 3

Program Execution Model and Semantics

Our Language

- Variables $v \in \mathbb{V}$ar, expressions $e \in \mathbb{E}$xpr and labels $l, m \in \mathbb{L}$abel
  - Expressions compute integer or boolean values
  - A label is an index that holds the position of a statement in a program
- Labelled three address code statements
- We assume that the programs are type correct

Statements in Our Language

- Assignment $l:\ v = e$ where $l \in \mathbb{L}$abel, $v \in \mathbb{V}$ar and $e \in \mathbb{E}$xpr
- Expression computation $l:\ e$ where $l \in \mathbb{L}$abel and $e \in \mathbb{E}$xpr
  (This models use of variables in statements other than assignments)
- Unconditional jump $l:$ goto m where $l, m \in \mathbb{L}$abel
- Conditional jump $l:$ if e goto m where $l, m \in \mathbb{L}$abel, and $e \in \mathbb{E}$xpr
- No operation $l:$ nop

(Other statements such as function calls, returns, heap accesses etc. will be added when required)

Context Free Grammar of Our Language

- program (P), statement (S), label (m)
- expression (E), arithmetic expression (aE), boolean expression (bE)
- binary arithmetic operator (bao), unary arithmetic operator (uao), binary boolean operator (bbo), unary boolean operator (ubo), relational operator (ro)
- arithmetic value (aV), boolean value (bV). variable (v), number (n)

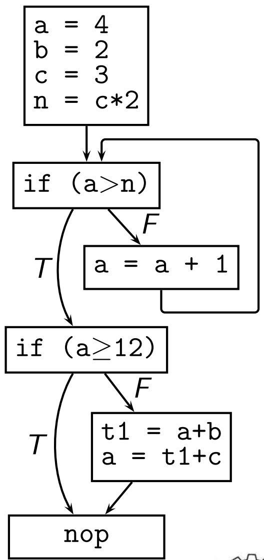
$$
\begin{array}{rcl}
P & \rightarrow & m : S\ \ P \ \mid\ m : S \\
S & \rightarrow & v\ =\ E \ \mid\ E \ \mid\ \text{goto } m \ \mid\ \text{if } E \text{ goto } m \ \mid\ \text{nop} \\
E & \rightarrow & aE \ \mid\ bE \\
aE & \rightarrow & aV \text{ bao } aV \ \mid\ \text{uao } aV \ \mid\ aV \\
bE & \rightarrow & bV \text{ bbo } bV \ \mid\ \text{ubo } bV \ \mid\ aV \text{ ro } aV \ \mid\ bV \\
aV & \rightarrow & v \ \mid\ n \\
bV & \rightarrow & v \ \mid\ T \ \mid\ F
\end{array}
$$

An Example Program

```
int main()
{ int a, b, c, n;

  a = 4;
  b = 2;
  c = 3;
  n = c*2;
  while (a <= n)
  {
    a = a+1;
  }
  if (a < 12)
    a = a+b+c;
}
```

```
1: a = 4
2: b = 2
3: c = 3
4: n = c*2
5: if (a > n)
       goto 8
6: a = a + 1
7: goto 5
8: if (a ≥ 12)
       goto 11
9: t1 = a+b
10: a = t1+c
11: nop
```



Labels and Program Points

```
(1:)  a = 4
(2:)  b = 2
(3:)  c = 3
(4:)  n = c*2
(5:)  if (a > n) goto 8
(6:)  a = a + 1
(7:)  goto 5
(8:)  if (a ≥ 12) goto 11
(9:)  t1 = a+b
(10:) a = t1+c
(11:) nop
```

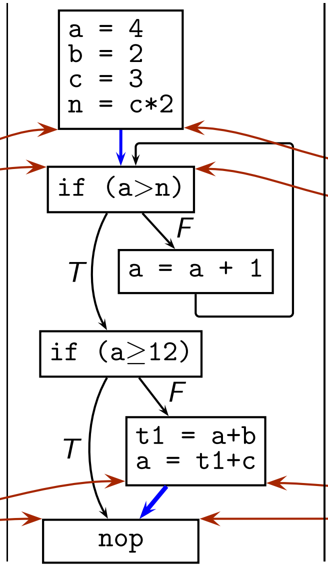
A label of a statement represents

- the program point just before the execution of the statement
- the program point just after the execution of the previous statement
- both the source and the target of the control transfer edge reaching the statement

This is fine if there is no other control transfer to the same program point

Labels and Control Flow

```
1: a = 4
2: b = 2
3: c = 3
4: n = c*2
5: if (a > n)
       goto 8
6: a = a + 1
7: goto 5
8: if (a ≥ 12)
       goto 11
9: t1 = a+b
10: a = t1+c
11: nop
```



- Value of variable *a* could be different at these two points
- Need to distinguish between them
- Blue edges represent implicit goto across block
- We need to explicate all such implicit gotos

Labels and Control Flow

Updating Control Flow

- We assume that all implicit gotos across basic blocks are explicited and labels adjusted appropriately
  This is required only for the purpose of our reasoning about our analysis

Entities in Our Example Program

```
1: a = 4
2: b = 2
3: c = 3
4: n = c*2
5: goto 6
6: if (a > n) goto 9
7: a = a + 1
8: goto 6
9: if (a ≥ 12) goto 13
10: t1 = a+b
11: a = t1+c
12: goto 13
13: nop
```

$$\mathbb{L}\text{abel} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13\}$$

$$\mathbb{V}\text{ar} = \{a, b, c, n, t1\}$$

$$\mathbb{E}\text{xpr} = \{c * 2, a > n, a + 1, a \geq 12, a + b, t1 + c\}$$

The Semantics of Our Language

- $\sigma \in \mathbb{S}\text{tore} : \mathbb{V}\text{ar} \mapsto \mathbb{I} \cup \mathbb{B} \cup \{\bot\}$
  (σ is $\mathbb{V}\text{ar} \mapsto \mathbb{I} \cup \mathbb{B} \cup \{\bot\}$ and $\mathbb{S}\text{tore}$ is the set of σs)
- $(l, \sigma) \in \mathbb{S}\text{tate} : \mathbb{L}\text{abel} \mapsto \mathbb{S}\text{tore}$
  - Q. Why not $\mathbb{L}\text{abel} \times \mathbb{S}\text{tore}$?
  - A. Only one store can be associated with a given label
- Execution of program causes state transitions

Execution of Our Example Program

```
1:  a = 4
2:  b = 2
3:  c = 3
4:  n = c*2
5:  goto 6
6:  if (a > n) goto 9
7:  a = a + 1
8:  goto 6
9:  if (a ≥ 12) goto 13
10: t1 = a+b
11: a = t1+c
12: goto 13
13: nop
```

State $(l, \sigma) =$

14,

Variable	Value
a	12
b	2
c	3
n	6
$t1$	9

Execution terminates when a label $l \notin \mathbb{L}\text{abel}$ is reached

Defining Small Step Semantics

- Goal: Modelling state transitions caused by various statements
- Notation
  - $[\![x]\!]\sigma = \sigma(x)$. Value of x in store σ
  - $[\![e]\!]\sigma$. Value of expression e computed from the values in store σ
  - $\sigma[y \mapsto v]$.
    A new store resulting from replacing the value of y by v. Other values remain the same.

$$(\sigma' = \sigma[y \mapsto v]) \Rightarrow \forall x \in \mathbb{V}\text{ar} : [\![x]\!]\sigma' = \left\{ \begin{array}{ll} [\![x]\!]\sigma & x \text{ is not } y \\ v & \text{otherwise} \end{array} \right\}$$

Defining Small Step Semantics

- Goal: Modelling state transitions caused by various statements
- Syntax of a semantic rule

$$\frac{\textit{Premise}}{(\textit{Oldstate}) \rightarrowtail \textit{Statement} \rightarrowtail (\textit{NewState})} \quad \text{Rule Name}_{ns}$$

Small Step Semantics: Computation

$$\frac{}{(l,\sigma) \rightarrowtail x = e \rightarrowtail (l+1, \sigma[x \mapsto [\![e]\!]\sigma])}\ \text{asgn}_{ns}$$

$$\frac{}{(l,\sigma) \rightarrowtail \text{e} \rightarrowtail (l+1, \sigma)}\ \text{expr}_{ns}$$

$$\frac{}{(l,\sigma) \rightarrowtail \text{nop} \rightarrowtail (l+1, \sigma)}\ \text{nop}_{ns}$$

Small Step Semantics: Computation

$$\frac{}{(l,\sigma) \rightarrowtail x = e \rightarrowtail (l+1, \sigma[x \mapsto [\![e]\!]\sigma])}\ \text{asgn}_{ns}$$

$$\frac{}{(l,\sigma) \rightarrowtail \text{e} \rightarrowtail (l+1, \sigma)}\ \text{expr}_{ns}$$

$$\frac{}{(l,\sigma) \rightarrowtail \text{nop} \rightarrowtail (l+1, \sigma)}\ \text{nop}_{ns}$$

Small Step Semantics: Computation

$$\frac{/ * \textit{unconditionally} * /}{(l,\sigma) \rightarrowtail x = e \rightarrowtail (l+1, \sigma[x \mapsto [\![e]\!]\sigma])}\ \text{asgn}_{ns}$$

$$\frac{}{(l,\sigma) \rightarrowtail \text{e} \rightarrowtail (l+1, \sigma)}\ \text{expr}_{ns}$$

$$\frac{}{(l,\sigma) \rightarrowtail \text{nop} \rightarrowtail (l+1, \sigma)}\ \text{nop}_{ns}$$

Control falls through

The value of x in the store changes to the value of e

Small Step Semantics: Computation

$$\frac{}{(l,\sigma) \rightarrowtail x = e \rightarrowtail (l+1, \sigma[x \mapsto [\![e]\!]\sigma])}\ \text{asgn}_{ns}$$

$$\frac{}{(l,\sigma) \rightarrowtail \text{e} \rightarrowtail (l+1, \sigma)}\ \text{expr}_{ns}$$

$$\frac{}{(l,\sigma) \rightarrowtail \text{nop} \rightarrowtail (l+1, \sigma)}\ \text{nop}_{ns}$$

Small Step Semantics: Computation

$$\frac{}{(l,\sigma) \rightarrowtail x = e \rightarrowtail (l+1, \sigma[x \mapsto [\![e]\!]\sigma])}\ \text{asgn}_{ns}$$

$$\frac{/ * \ \textit{unconditionally} \ * /}{(l,\sigma) \rightarrowtail \text{e} \rightarrowtail (l+1, \sigma)}\ \text{expr}_{ns}$$

$$\frac{}{(l,\sigma) \rightarrowtail \text{nop} \rightarrowtail (l+1, \sigma)}\ \text{nop}_{ns}$$

Control falls through

The store remains same

Small Step Semantics: Computation

$$\frac{}{(l,\sigma) \rightarrowtail x = e \rightarrowtail (l+1, \sigma[x \mapsto [\![e]\!]\sigma])}\ \text{asgn}_{ns}$$

$$\frac{}{(l,\sigma) \rightarrowtail \text{e} \rightarrowtail (l+1, \sigma)}\ \text{expr}_{ns}$$

$$\frac{}{(l,\sigma) \rightarrowtail \text{nop} \rightarrowtail (l+1, \sigma)}\ \text{nop}_{ns}$$

Small Step Semantics: Computation

$$\frac{}{(l,\sigma) \rightarrowtail x = e \rightarrowtail (l+1, \sigma[x \mapsto [\![e]\!]\sigma])}\ \text{asgn}_{ns}$$

$$\frac{}{(l,\sigma) \rightarrowtail \text{e} \rightarrowtail (l+1, \sigma)}\ \text{expr}_{ns}$$

$$\frac{/ * \ \textit{unconditionally} \ * /}{(l,\sigma) \rightarrowtail \text{nop} \rightarrowtail (l+1, \sigma)}\ \text{nop}_{ns}$$

Control falls through

The store remains same

Small Step Semantics: Control Flow

$$\frac{}{(l,\sigma) \rightarrowtail \text{goto } m \rightarrowtail (m, \sigma)}\ \text{goto}_{ns}$$

$$\frac{[\![e]\!]\sigma = T}{(l,\sigma) \rightarrowtail \text{if } \ e \text{ goto } m \rightarrowtail (m, \sigma)}\ \text{ifgotoT}_{ns}$$

$$\frac{[\![e]\!]\sigma = F}{(l,\sigma) \rightarrowtail \text{if } \ e \text{ goto } m \rightarrowtail (l+1, \sigma)}\ \text{ifgotoF}_{ns}$$

Small Step Semantics: Control Flow

$$\frac{}{(l,\sigma) \rightarrowtail \text{goto } m \rightarrowtail (m,\sigma)}\ \text{goto}_{ns}$$

$$\frac{[\![e]\!]\sigma = T}{(l,\sigma) \rightarrowtail \text{if } e \text{ goto } m \rightarrowtail (m,\sigma)}\ \text{ifgotoT}_{ns}$$

$$\frac{[\![e]\!]\sigma = F}{(l,\sigma) \rightarrowtail \text{if } e \text{ goto } m \rightarrowtail (l+1,\sigma)}\ \text{ifgotoF}_{ns}$$

Small Step Semantics: Control Flow

$$\frac{/ * \textit{ unconditionally } * /}{(l,\sigma) \rightarrowtail \text{goto } m \rightarrowtail (m,\sigma)}\ \text{goto}_{ns}$$

$$\frac{[\![e]\!]\sigma = T}{(l,\sigma) \rightarrowtail \text{if } e \text{ goto } m \rightarrowtail (m,\sigma)}\ \text{ifgotoT}_{ns}$$

$$\frac{[\![e]\!]\sigma = F}{(l,\sigma) \rightarrowtail \text{if } e \text{ goto } m \rightarrowtail (l+1,\sigma)}\ \text{ifgotoF}_{ns}$$

Control is transferred to the target statement

The store remains same

Small Step Semantics: Control Flow

$$\frac{}{(l,\sigma) \rightarrowtail \text{goto } m \rightarrowtail (m,\sigma)}\ \text{goto}_{ns}$$

$$\frac{[\![e]\!]\sigma = T}{(l,\sigma) \rightarrowtail \text{if } e \text{ goto } m \rightarrowtail (m,\sigma)}\ \text{ifgotoT}_{ns}$$

$$\frac{[\![e]\!]\sigma = F}{(l,\sigma) \rightarrowtail \text{if } e \text{ goto } m \rightarrowtail (l+1,\sigma)}\ \text{ifgotoF}_{ns}$$

Small Step Semantics: Control Flow

$$\frac{}{(l,\sigma) \rightarrowtail \text{goto } m \rightarrowtail (m,\sigma)}\ \text{goto}_{ns}$$

$$\frac{[\![e]\!]\sigma = T}{(l,\sigma) \rightarrowtail \text{if } e \text{ goto } m \rightarrowtail (m,\sigma)}\ \text{ifgotoT}_{ns}$$

$$\frac{[\![e]\!]\sigma = F}{(l,\sigma) \rightarrowtail \text{if } e \text{ goto } m \rightarrowtail (l+1,\sigma)}\ \text{ifgotoF}_{ns}$$

Control is transferred to the target statement

The store remains same

Small Step Semantics: Control Flow

$$\frac{}{(l,\sigma) \rightarrowtail \text{goto } m \rightarrowtail (m,\sigma)}\ \text{goto}_{ns}$$

$$\frac{[\![e]\!]\sigma = T}{(l,\sigma) \rightarrowtail \text{if } \ e \text{ goto } m \rightarrowtail (m,\sigma)}\ \text{ifgotoT}_{ns}$$

$$\frac{[\![e]\!]\sigma = F}{(l,\sigma) \rightarrowtail \text{if } \ e \text{ goto } m \rightarrowtail (l+1,\sigma)}\ \text{ifgotoF}_{ns}$$

Small Step Semantics: Control Flow

$$\frac{}{(l,\sigma) \rightarrowtail \text{goto } m \rightarrowtail (m,\sigma)}\ \text{goto}_{ns}$$

$$\frac{[\![e]\!]\sigma = T}{(l,\sigma) \rightarrowtail \text{if } \ e \text{ goto } m \rightarrowtail (m,\sigma)}\ \text{ifgotoT}_{ns}$$

$$\frac{[\![e]\!]\sigma = F}{(l,\sigma) \rightarrowtail \text{if } \ e \text{ goto } m \rightarrowtail (l+1,\sigma)}\ \text{ifgotoF}_{ns}$$

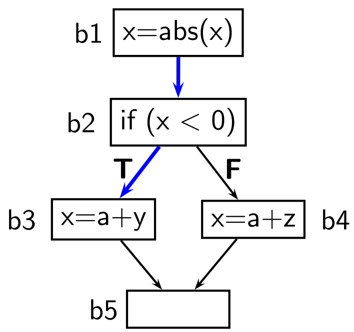
Control falls through

The store remains same

Part 4

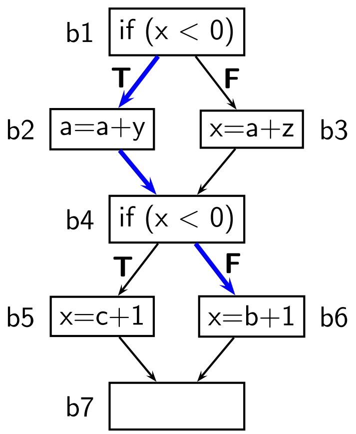
Soundness and Precision of Data Flow Analysis

Conservative Nature of Analysis (1)



- abs(n) returns the absolute value of n
- Is y live on entry to block b2?
- By execution semantics, no
  Path b1→b2→b3 is an infeasible execution path
- A compiler make conservative assumptions: *All branch outcomes are possible*
  ⇒ Consider every path in CFG as a potential execution execution path
- Our analysis concludes that y is live on entry to block b2

Conservative Nature of Analysis (2)



- Is b live on entry to block b2?
- By execution semantics, no
  Path b1→b2→b4→b6 is an infeasible execution path
- Is c live on entry to block b3?
  Path b1→b3→b4→b6 is a feasible execution path
- A compiler make conservative assumptions
  ⇒ our analysis is *path insensitive*
  Note: It is *flow sensitive* (i.e. information is computed for every control flow points)
- Our analysis concludes that b is live at the entry of b2 and c is live at the entry of b3

Conservative Nature of Analysis

Reasons by analysis results may be imprecise

- At intraprocedural level
  - We assume that all paths are potentially executable
  - Our analysis is path insensitive
  - In some cases, sharing of paths generates spurious information (Nondistributive flow functions)
- At interprocedural level
  - Context insensitivity:
    Merging of information across all calling contexts
  - Flow insensitivity: Disregarding the control flow

Showing Soundness of Data Flow Analysis

1. Specify analysis in a notation similar to that of execution semantics
2. Relate analysis rules to rules of execution semantics
3. Syntax of declarative specification of analysis

$$\frac{\textit{Premise}}{(l:\ \text{Info at } l)\ \rightarrow\ l:\textit{Statement}\ \rightarrow\ (m:\ \text{Info at } m)}\ \text{Rule Name}_{lv}$$

Declarative Specification of Liveness Analysis

$$\frac{}{(l : L \cup Var(e))\ \rightarrow\ l : e\ \rightarrow\ (l+1 : L)}\ \text{expr}_{lv}$$

$$\frac{}{(l : (L - \{x\}) \cup Var(e))\ \rightarrow\ l : x = e\ \rightarrow\ (l+1 : L)}\ \text{asgn}_{lv}$$

$$\frac{}{(l : L)\ \rightarrow\ l : nop\ \rightarrow\ (l+1 : L)}\ \text{nop}_{lv}$$

Declarative Specification of Liveness Analysis

$$\frac{}{(l : L \cup Var(e)) \;\rightarrow\; l : e \;\rightarrow\; (l+1 : L)}\ \text{expr}_{lv}$$

$$\frac{}{(l : (L - \{x\}) \cup Var(e)) \;\rightarrow\; l : x = e \;\rightarrow\; (l+1 : L)}\ \text{asgn}_{lv}$$

$$\frac{}{(l : L) \;\rightarrow\; l : nop \;\rightarrow\; (l+1 : L)}$$

Variables occuring in expression e

Declarative Specification of Liveness Analysis

$$\frac{}{(l : L) \;\rightarrow\; l : \text{goto}\ \ m \;\rightarrow\; (m : L)}\ \text{goto}_{lv}$$

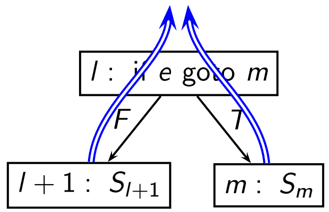
$$\frac{}{(l : L \cup Var(e)) \;\rightarrow\; l : \text{if}\ \ e\ \ \text{goto}\ \ m \;\rightarrow\; (m : L)}\ \text{ifgotoT}_{lv}$$

$$\frac{}{(l : L \cup Var(e)) \;\rightarrow\; l : \text{if}\ \ e\ \ \text{goto}\ \ m \;\rightarrow\; (l+1 : L)}\ \text{ifgotoF}_{lv}$$

Declarative Specification of Liveness Analysis

$$\frac{L'' \supseteq L' \qquad (l : L') \;\rightarrow\; l : S \;\rightarrow\; (m : L)}{(l : L'') \;\rightarrow\; l : S \;\rightarrow\; (m : L)}\ \text{subsumption}_{lv}$$

- The need of subsumption: Adjusting the values at fork nodes

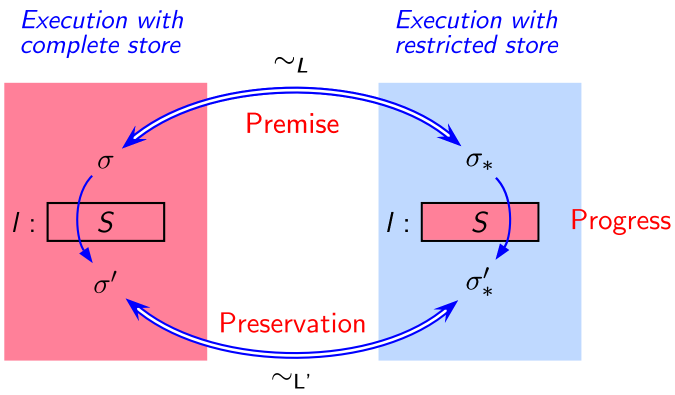


Soundness Criterion for Liveness Analysis

- Equivalence of stores: $\sigma \sim_L \sigma'$
  - σ and σ' "agree" on variables in L. $\forall v \in L$, $[\![v]\!]\sigma = [\![v]\!]\sigma'$
  - Values of other variables do not matter

  σ' simulates σ with respect to L
- Soundness criteria
  - At each program point, restrict the store to the variables that are live
  - Starting from equivalent states, the execution of each statement should cause transition to equivalent states
    - Given that the restricted store is equivalent to the complete store before a statement S
    - If S can be executed without any problem ("*progress*" in program execution) AND
    - The resulting restricted store is equivalent to the complete store ("*preservation* of semantics")
  - By structural induction on the program, the result of liveness analysis is correct

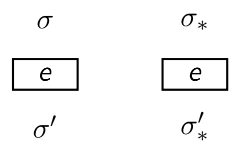
Proving Soundness by Progress and Preservation



- The preservation outcome become premise for the next statement
- It is sufficient to prove the above for each kind of statement

Progress and Preservation for Expression Statement

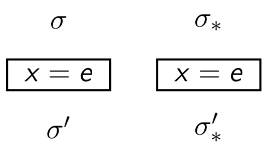
$$\frac{}{(l : L' \cup Var(e)) \;\rightarrow\; l : e \;\rightarrow\; (l+1 : L')}\ \text{expr}_{lv}$$



- Given: $\sigma \sim_L \sigma_*,\ L = L' \cup Var(e)$
- Progress:
  e can be evaluated because variables in $Var(e)$ exist in σ_*
- Preservation:
  Values of all variables remain unchanged
  $\Rightarrow\ \sigma' \sim_{L'} \sigma'_*$

Progress and Preservation for Assignment Statement

$$\frac{}{(l : (L' - \{x\}) \cup Var(e)) \;\rightarrow\; l : x = e \;\rightarrow\; (l+1 : L')}\ \text{asgn}_{lv}$$

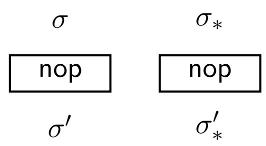


- Given: $\sigma \sim_L \sigma_*,\ L = (L' - \{x\}) \cup Var(e)$
- Progress:
  e can be evaluated because variables in $Var(e)$ exist in σ_*
- Preservation:
  - $\forall v \in (L' - \{x\}) \cup Var(e)$
    $([\![v]\!]\sigma = [\![v]\!]\sigma_*) \Rightarrow ([\![v]\!]\sigma' = [\![v]\!]\sigma'_*)$
  - $([\![e]\!]\sigma = [\![e]\!]\sigma_*) \Rightarrow ([\![x]\!]\sigma' = [\![x]\!]\sigma'_*)$

  $\Rightarrow\ \sigma' \sim_{L'} \sigma'_*$

Progress and Preservation for nop Statement

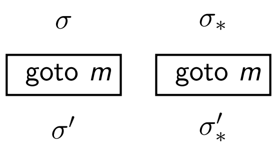
$$\frac{}{(l : L) \;\rightarrow\; l : nop \;\rightarrow\; (l+1 : L)}\ \text{nop}_{lv}$$



- Progress and Preservation follow trivially

Progress and Preservation for Unconditional Goto Statement

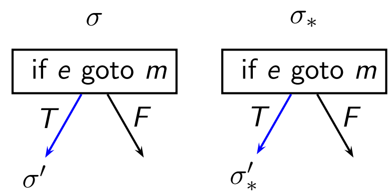
$$\frac{}{(l:L) \;\rightarrow\; l:\text{goto}\;\; m \;\rightarrow\; (m:L)}\;\text{goto}_{lv}$$



- Progress and Preservation follow trivially

Progress and Preservation for Conditional Goto Statement

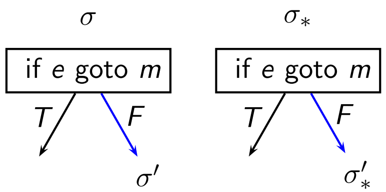
$$\frac{}{(l:L' \cup Var(e)) \;\rightarrow\; l:\text{if } e \text{ goto } m \;\rightarrow\; (m:L')}\;\text{ifgotoT}_{lv}$$



- Given: $\sigma \sim_L \sigma_*$, $L = L' \cup Var(e)$
- Progress:
  - $[\![e]\!]\sigma = [\![e]\!]\sigma_*$
  - Branch outcome is same
- Preservation:
  Values of all variables remain unchanged
  $\Rightarrow\;\; \sigma' \sim_{L'} \sigma'_*$

Progress and Preservation for Conditional Goto Statement

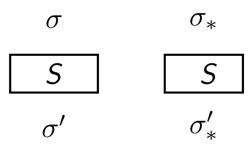
$$\frac{}{(l:L' \cup Var(e)) \;\rightarrow\; l:\text{if } e \text{ goto } m \;\rightarrow\; (m:L')}\;\text{ifgotoF}_{lv}$$



- Given: $\sigma \sim_L \sigma_*$, $L = L' \cup Var(e)$
- Progress:
  - $[\![e]\!]\sigma = [\![e]\!]\sigma_*$
  - Branch outcome is same
- Preservation:
  Values of all variables remain unchanged
  $\Rightarrow\;\; \sigma' \sim_{L'} \sigma'_*$

Progress and Preservation for Subsumption Rule

$$\frac{L'' \supseteq L \quad (l:L) \;\rightarrow\; l:S \;\rightarrow\; (m:L')}{(l:L'') \;\rightarrow\; l:S \;\rightarrow\; (m:L')}\;\text{subsumption}_{lv}$$



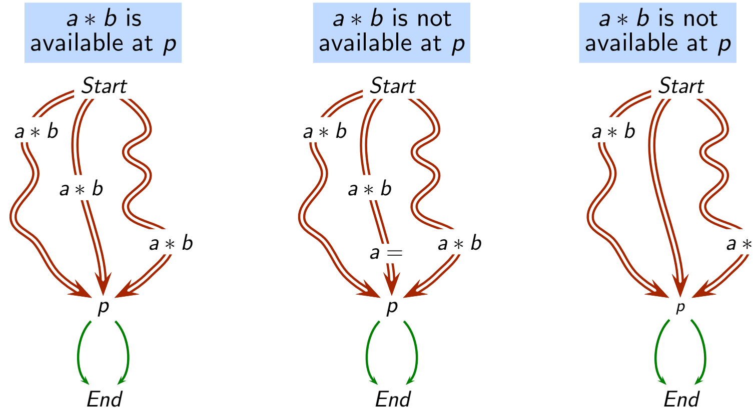
- Given: $\sigma \sim_L \sigma_*$ and $\sigma' \sim_{L'} \sigma'_*$
- Progress: $(\sigma \sim_L \sigma_*) \wedge L'' \supseteq L)$
  $\Rightarrow$ Progress follows trivially
- Preservation:
  $(\sigma \sim_L \sigma_* \Rightarrow \sigma' \sim_{L'} \sigma'_*) \wedge L'' \supseteq L)$
  $\Rightarrow\;\; (\sigma \sim_{L''} \sigma_* \Rightarrow \sigma' \sim_{L'} \sigma'_*)$

Part 5

Available Expressions Analysis

Defining Available Expressions Analysis

An expression e is available at a program point p, if every path from program entry to p contains an evaluation of e which is not followed by a definition of any operand of e.



Local Data Flow Properties for Available Expressions Analysis

$Gen_n = \{ e \mid$ expression e is evaluated in basic block n and this evaluation is not followed by a definition of any operand of $e\}$

$Kill_n = \{ e \mid$ basic block n contains a definition of an operand of $e\}$

	Entity	Manipulation	Exposition
Gen_n	Expression	Use	Downwards
$Kill_n$	Expression	Modification	Anywhere

Data Flow Equations For Available Expressions Analysis

$$In_n = \begin{cases} BI & n \text{ is } Start \text{ block} \\ \bigcap_{p \in pred(n)} Out_p & \text{otherwise} \end{cases}$$

$$Out_n = Gen_n \cup (In_n - Kill_n)$$

Alternatively,

$$Out_n = f_n(In_n), \quad \text{where}$$

$$f_n(X) = Gen_n \cup (X - Kill_n)$$

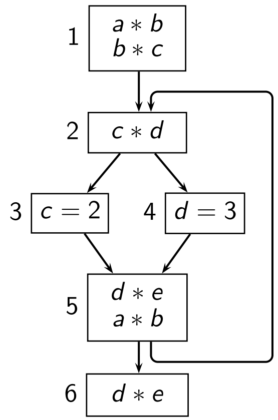
In_n and Out_n are sets of expressions.

Using Data Flow Information of Available Expressions Analysis

- Common subexpression elimination
  - If an expression is available at the entry of a block *b* **and**
  - a computation of the expression exists in *b* **such that**
  - it is not preceded by a definition of any of its operands

  Then the expression is redundant
- Redundant expression must be upwards exposed
- Expressions in Gen_n are downwards exposed

An Example of Available Expressions Analysis

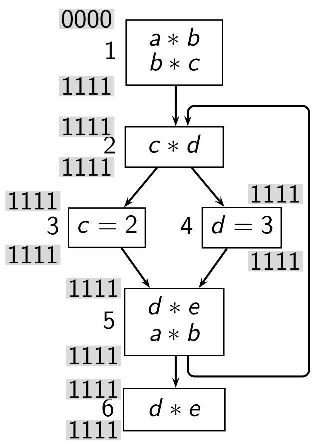


Let $e_1 \equiv a*b$, $e_2 \equiv b*c$, $e_3 \equiv c*d$, $e_4 \equiv d*e$

Node	Computed		Killed		Available		Redund.	
1	$\{e_1, e_2\}$	1100	$\emptyset$	0000	$\emptyset$	0000	$\emptyset$	0000
2	$\{e_3\}$	0010	$\emptyset$	0000	$\{e_1\}$	1000	$\emptyset$	0000
3	$\emptyset$	0000	$\{e_2, e_3\}$	0110	$\{e_1, e_3\}$	1010	$\emptyset$	0000
4	$\emptyset$	0000	$\{e_3, e_4\}$	0011	$\{e_1, e_3\}$	1010	$\emptyset$	0000
5	$\{e_1, e_4\}$	1001	$\emptyset$	0000	$\{e_1\}$	1000	$\{e_1\}$	1000
6	$\{e_4\}$	0001	$\emptyset$	0000	$\{e_1, e_4\}$	1001	$\{e_4\}$	0001

An Example of Available Expressions Analysis

Initialisation

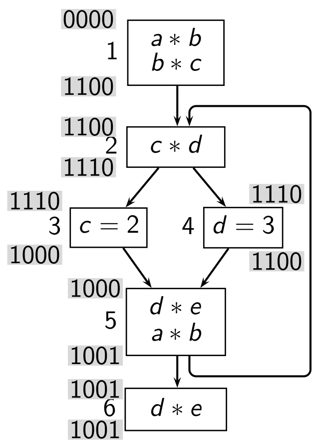


Let $e_1 \equiv a*b$, $e_2 \equiv b*c$, $e_3 \equiv c*d$, $e_4 \equiv d*e$

Node	Computed		Killed		Available		Redund.	
1	$\{e_1, e_2\}$	1100	$\emptyset$	0000	$\emptyset$	0000	$\emptyset$	0000
2	$\{e_3\}$	0010	$\emptyset$	0000	$\{e_1\}$	1000	$\emptyset$	0000
3	$\emptyset$	0000	$\{e_2, e_3\}$	0110	$\{e_1, e_3\}$	1010	$\emptyset$	0000
4	$\emptyset$	0000	$\{e_3, e_4\}$	0011	$\{e_1, e_3\}$	1010	$\emptyset$	0000
5	$\{e_1, e_4\}$	1001	$\emptyset$	0000	$\{e_1\}$	1000	$\{e_1\}$	1000
6	$\{e_4\}$	0001	$\emptyset$	0000	$\{e_1, e_4\}$	1001	$\{e_4\}$	0001

An Example of Available Expressions Analysis

Iteration #1

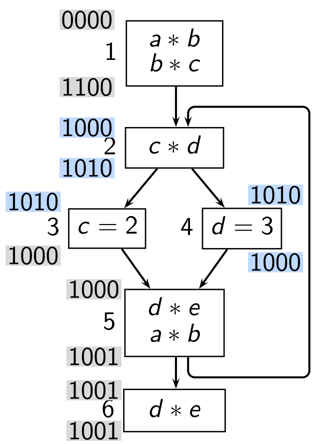


Let $e_1 \equiv a*b$, $e_2 \equiv b*c$, $e_3 \equiv c*d$, $e_4 \equiv d*e$

Node	Computed		Killed		Available		Redund.	
1	$\{e_1, e_2\}$	1100	$\emptyset$	0000	$\emptyset$	0000	$\emptyset$	0000
2	$\{e_3\}$	0010	$\emptyset$	0000	$\{e_1\}$	1000	$\emptyset$	0000
3	$\emptyset$	0000	$\{e_2, e_3\}$	0110	$\{e_1, e_3\}$	1010	$\emptyset$	0000
4	$\emptyset$	0000	$\{e_3, e_4\}$	0011	$\{e_1, e_3\}$	1010	$\emptyset$	0000
5	$\{e_1, e_4\}$	1001	$\emptyset$	0000	$\{e_1\}$	1000	$\{e_1\}$	1000
6	$\{e_4\}$	0001	$\emptyset$	0000	$\{e_1, e_4\}$	1001	$\{e_4\}$	0001

An Example of Available Expressions Analysis

Iteration #2

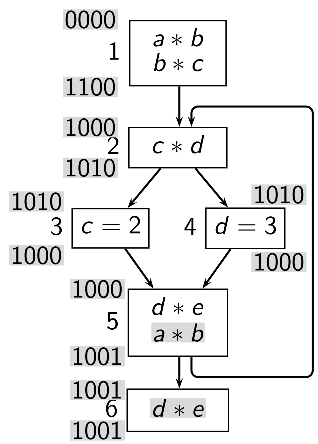


Let $e_1 \equiv a * b$, $e_2 \equiv b * c$, $e_3 \equiv c * d$, $e_4 \equiv d * e$

Node	Computed		Killed		Available		Redund.	
1	$\{e_1, e_2\}$	1100	$\emptyset$	0000	$\emptyset$	0000	$\emptyset$	0000
2	$\{e_3\}$	0010	$\emptyset$	0000	$\{e_1\}$	1000	$\emptyset$	0000
3	$\emptyset$	0000	$\{e_2, e_3\}$	0110	$\{e_1, e_3\}$	1010	$\emptyset$	0000
4	$\emptyset$	0000	$\{e_3, e_4\}$	0011	$\{e_1, e_3\}$	1010	$\emptyset$	0000
5	$\{e_1, e_4\}$	1001	$\emptyset$	0000	$\{e_1\}$	1000	$\{e_1\}$	1000
6	$\{e_4\}$	0001	$\emptyset$	0000	$\{e_1, e_4\}$	1001	$\{e_4\}$	0001

An Example of Available Expressions Analysis

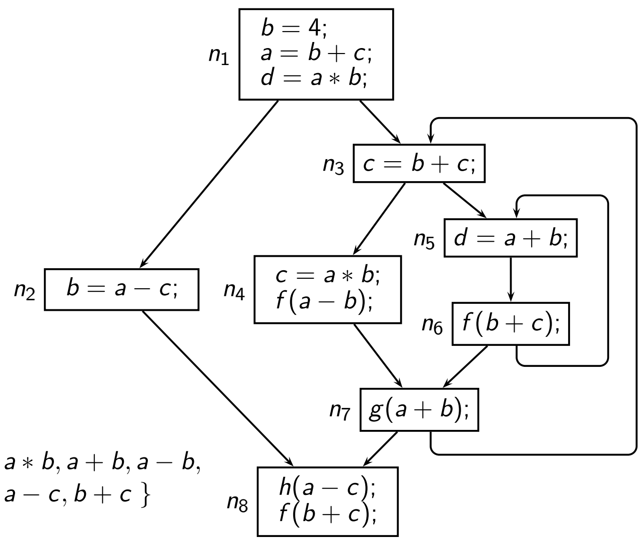
Final Result



Let $e_1 \equiv a * b$, $e_2 \equiv b * c$, $e_3 \equiv c * d$, $e_4 \equiv d * e$

Node	Computed		Killed		Available		Redund.	
1	$\{e_1, e_2\}$	1100	$\emptyset$	0000	$\emptyset$	0000	$\emptyset$	0000
2	$\{e_3\}$	0010	$\emptyset$	0000	$\{e_1\}$	1000	$\emptyset$	0000
3	$\emptyset$	0000	$\{e_2, e_3\}$	0110	$\{e_1, e_3\}$	1010	$\emptyset$	0000
4	$\emptyset$	0000	$\{e_3, e_4\}$	0011	$\{e_1, e_3\}$	1010	$\emptyset$	0000
5	$\{e_1, e_4\}$	1001	$\emptyset$	0000	$\{e_1\}$	1000	$\{e_1\}$	1000
6	$\{e_4\}$	0001	$\emptyset$	0000	$\{e_1, e_4\}$	1001	$\{e_4\}$	0001

Tutorial Problem for Available Expressions Analysis



$\mathbb{E}\text{xpr} = \{\ a * b, a + b, a - b, a - c, b + c\ \}$

Solution of the Tutorial Problem

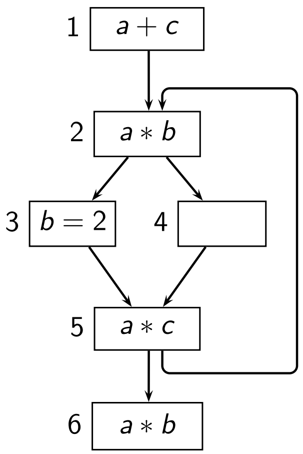
Bit vector	$a * b$	$a + b$	$a - b$	$a - c$	$b + c$

Node	Local Information			Global Information: Iteration # 1		Global Information: Changes in iteration # 2		$Redundant_n$
	Gen_n	$Kill_n$	$AntGen_n$	In_n	Out_n	In_n	Out_n	
n_1	10001	11111	00000	00000	10001			00000
n_2	00010	11101	00010	10001	00010			00000
n_3	00000	00011	00001	10001	10000	10000		00000
n_4	10100	00011	10100	10000	10100			10000
n_5	01000	00000	01000	10000	11000			00000
n_6	00001	00000	00001	11000	11001			00000
n_7	01000	00000	01000	10000	11000			00000
n_8	00011	00000	00011	00000	00011			00000

Further Tutorial Problems

Bit Vector

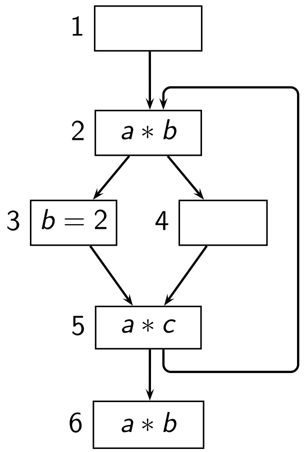
$a+c$	$a*b$	$a*c$



BI	Node	Initialization $\mathbb{U}$		Initialization $\emptyset$	
		In_n	Out_n	In_n	Out_n
$\emptyset$	1	000	100	000	100
	2	100	110	000	010
	3	110	100	010	000
	4	110	110	010	010
	5	100	101	000	001
	6	101	111	001	011
$\mathbb{U}$	1	111	111	111	111
	2	101	111	001	011
	3	111	101	011	001
	4	111	111	011	011
	5	101	101	001	001
	6	101	111	001	011

More Tutorial Problems

Number of iterations assuming that the order of In_i and Out_i computation is fixed (In_i is computed first and then Out_i is computed)



Traversal	Initialization			
	$\mathbb{U}$		$\emptyset$	
	BI		*BI*	
	$\mathbb{U}$	$\emptyset$	$\mathbb{U}$	$\emptyset$
Forward	2	1	2	1
Backward	3	4	3	2

Still More Tutorial Problems ☺

- Partially available expressions at program point p are expressions that are computed and remain unmodified along some path reaching p. The data flow equations for partially available expressions analysis are same as the data flow equations of available expressions analysis except that the confluence is changed to $\cup$.

  Perform partially available expressions analysis for the previous example program.

Result of Partially Available Expressions Analysis

Bit vector

$a*b$	$a+b$	$a-b$	$a-c$	$b+c$

Node	Local Information			Global Information				
				Iteration # 1		Changes in iteration # 2		$ParRedund_n$
	Gen_n	$Kill_n$	$AntGen_n$	In_n	Out_n	In_n	Out_n	
n_1	10001	11111	00000	00000	10001			00000
n_2	00010	11101	00010	10001	00010			00000
n_3	00000	00011	00001	10001	10000	11101	11100	00001
n_4	10100	00011	10100	10000	10100	11100	11100	10100
n_5	01000	00000	01000	10000	11000	11101	11101	01000
n_6	00001	00000	00001	11000	11001	11101	11101	00001
n_7	01000	00000	01000	11101	11101			01000
n_8	00011	00000	00011	11111	11111			00011

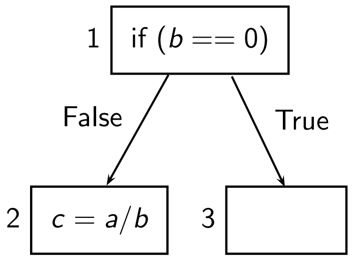
Part 6

Anticipable Expressions Analysis

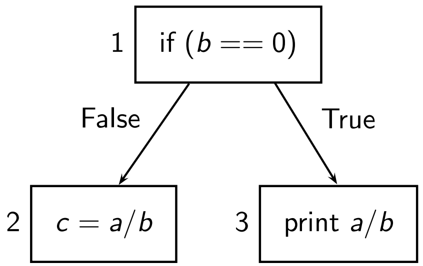
Defining Anticipable Expressions Analysis

- An expression e is anticipable at a program point p, if every path from p to the program exit contains an evaluation of e which is not preceded by a redefinition of any operand of e.
- Application : Safety of Code Hoisting

Safety of Code Motion



Hoisting a/b to the exit of 1 is unsafe ($\equiv$ can change the behaviour of the optimized program)



??

A guarded computation of an expression should not be converted to an unguarded computation

Defining Data Flow Analysis for Anticipable Expressions Analysis

Gen_n = { e | expression e is evaluated in basic block n and this evaluation is not preceded (within n) by a definition of any operand of e}

$Kill_n$ = { e | basic block n contains a definition of an operand of e}

	Entity	Manipulation	Exposition
Gen_n	Expression	Use	Upwards
$Kill_n$	Expression	Modification	Anywhere

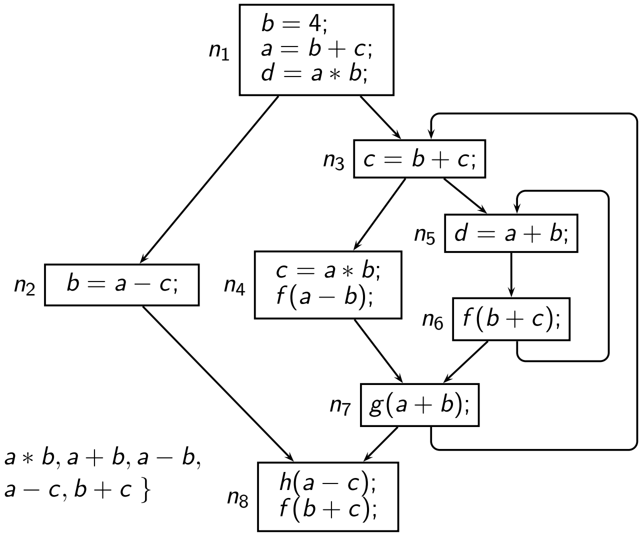
Data Flow Equations for Anticipable Expressions Analysis

$$In_n = Gen_n \cup (Out_n - Kill_n)$$

$$Out_n = \begin{cases} BI & n \text{ is } End \text{ block} \\ \bigcap_{s \in succ(n)} In_s & \text{otherwise} \end{cases}$$

In_n and Out_n are sets of expressions

Tutorial Problem for Anticipable Expressions Analysis



$\mathbb{E}\text{xpr} = \{\ a * b, a + b, a - b, a - c, b + c\ \}$

Result of Anticipable Expressions Analysis

Bit vector | $a * b$ | $a + b$ | $a - b$ | $a - c$ | $b + c$ |

Block	Local Information		Global Information			
			Iteration # 1		Changes in iteration # 2	
	Gen_n	$Kill_n$	Out_n	In_n	Out_n	In_n
n_8	00011	00000	00000	00011		
n_7	01000	00000	00011	01011	00001	01001
n_6	00001	00000	01011	01011	01001	01001
n_5	01000	00000	01011	01011	01001	01001
n_4	10100	00011	01011	11100	01001	11100
n_3	00001	00011	01000	01001	01000	01001
n_2	00010	11101	00011	00010		
n_1	00000	11111	00000	00000		

Part 7

Reaching Definitions Analysis

Defining Reaching Definitions Analysis

- A definition $d_x : x = y$ reaches a program point u if it appears (without a refefinition of x) on some path from program entry to u
- Application : Copy Propagation
  A use of a variable x at a program point u can be replaced by y if $d_x : x = y$ is the only definition which reaches p and y is not modified between the point of d_x and p.

Defining Data Flow Analysis for Reaching Definitions Analysis

Let d_v be a definition of variable v

$$Gen_n = \{ d_v \mid \text{variable } v \text{ is defined in basic block } n \text{ and this definition is not followed (within } n\text{) by a definition of } v\}$$

$$Kill_n = \{ d_v \mid \text{basic block } n \text{ contains a definition of } v\}$$

	Entity	Manipulation	Exposition
Gen_n	Definition	Occurence	Downwards
$Kill_n$	Definition	Occurence	Anywhere

Data Flow Equations for Reaching Definitions Analysis

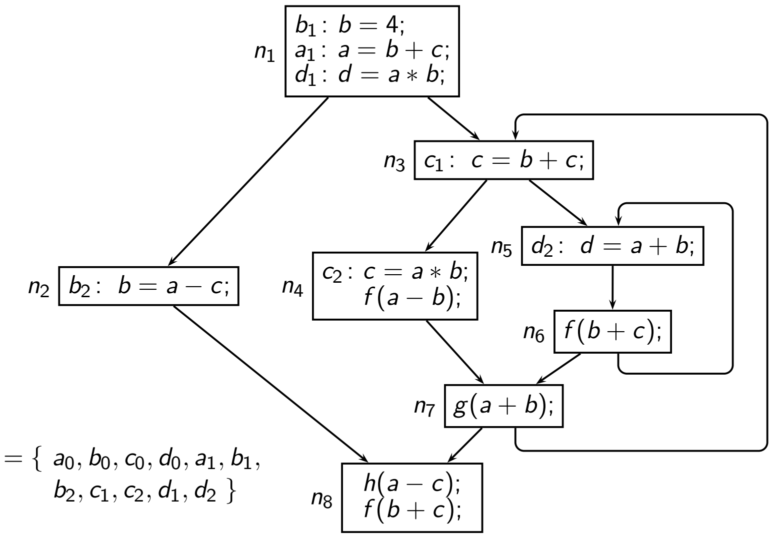
$$In_n = \begin{cases} BI & n \text{ is } Start \text{ block} \\ \bigcup_{p \in pred(n)} Out_p & \text{otherwise} \end{cases}$$

$$Out_n = Gen_n \cup (In_n - Kill_n)$$

$$BI = \{d_x : x = undef \mid x \in \mathbb{V}\text{ar}\}$$

In_n and Out_n are sets of definitions

Tutorial Problem for Reaching Definitions Analysis



$\mathbb{D}\text{efs} = \{ a_0, b_0, c_0, d_0, a_1, b_1, b_2, c_1, c_2, d_1, d_2 \}$

Result of Reaching Definitions Analysis

Block	Local Information		Global Information			
			Iteration # 1		Changes in iteration # 2	
	Gen_n	$Kill_n$	In_n	Out_n	In_n	Out_n
n_1	$\{a_1, b_1, d_1\}$	$\{a_0, a_1, b_0, b_1, b_2, d_0, d_1, d_2\}$	$\{a_0, b_0, c_0, d_0\}$	$\{a_1, b_1, c_0, d_1\}$		
n_2	$\{b_2\}$	$\{b_0, b_1, b_2\}$	$\{a_1, b_1, c_0, d_1\}$	$\{a_1, b_2, c_0, d_1\}$		
n_3	$\{c_1\}$	$\{c_0, c_1, c_2\}$	$\{a_1, b_1, c_0, d_1\}$	$\{a_1, b_1, c_1, d_1\}$	$\{a_1, b_1, c_0, c_1, c_2, d_1, d_2\}$	$\{a_1, b_1, c_1, d_1, d_2\}$
n_4	$\{c_2\}$	$\{c_0, c_1, c_2\}$	$\{a_1, b_1, c_1, d_1\}$	$\{a_1, b_1, c_2, d_1\}$	$\{a_1, b_1, c_1, d_1, d_2\}$	$\{a_1, b_1, c_2, d_1, d_2\}$
n_5	$\{d_2\}$	$\{d_0, d_1, d_2\}$	$\{a_1, b_1, c_1, d_1\}$	$\{a_1, b_1, c_1, d_2\}$	$\{a_1, b_1, c_1, d_1, d_2\}$	
n_6	$\emptyset$	$\emptyset$	$\{a_1, b_1, c_1, d_2\}$	$\{a_1, b_1, c_1, d_2\}$		
n_7	$\emptyset$	$\emptyset$	$\{a_1, b_1, c_1, c_2, d_1, d_2\}$	$\{a_1, b_1, c_1, c_2, d_1, d_2\}$		
n_8	$\emptyset$	$\emptyset$	$\{a_1, b_1, b_2, c_0, c_1, c_2, d_1, d_2\}$	$\{a_1, b_1, b_2, c_0, c_1, c_2, d_1, d_2\}$		

Part 8

Common Features of Bit Vector Data Flow Frameworks

Defining Local Data Flow Properties

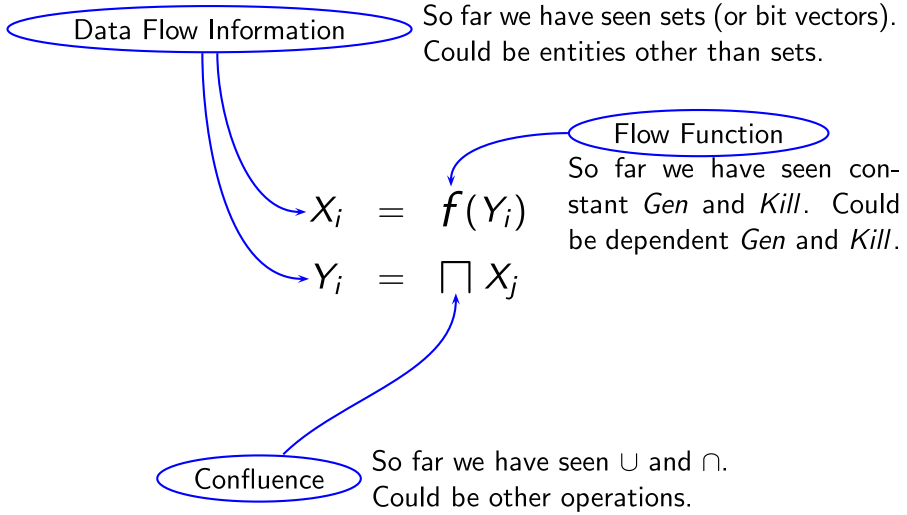
- Live variables analysis

	Entity	Manipulation	Exposition
Gen_n	Variable	Use	Upwards
$Kill_n$	Variable	Modification	Anywhere

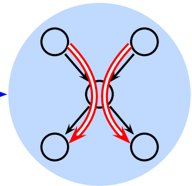
- Analysis of expressions

	Entity	Manipulation	Exposition	
			Availability	Anticipability
Gen_n	Expression	Use	Downwards	Upwards
$Kill_n$	Expression	Modification	Anywhere	Anywhere

Common Form of Data Flow Equations



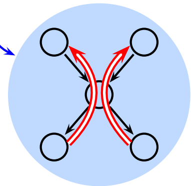
A Taxonomy of Bit Vector Data Flow Frameworks

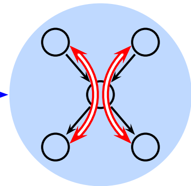


Any Path

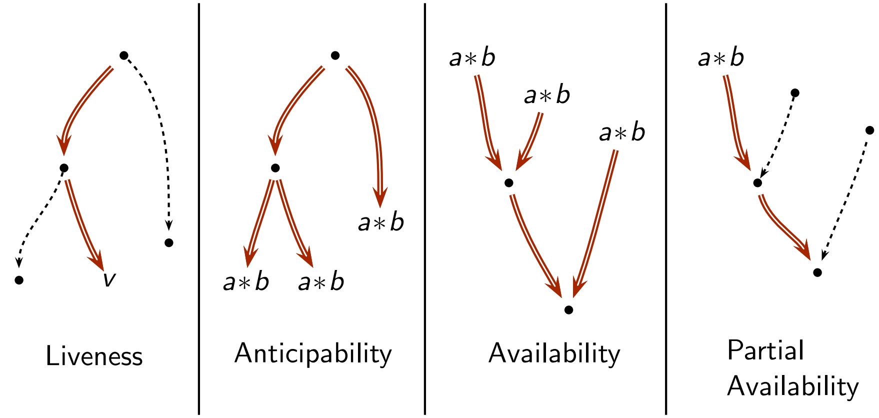
All Paths

	Confluence	
	Union	Intersection
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Anticipable Exressions
Bidirectional (limited)		Partial Redundancy Elimination (Original M-R Formulation)

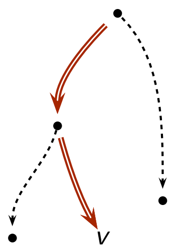




Data Flow Paths Discovered by Data Flow Analysis



Data Flow Paths Discovered by Data Flow Analysis

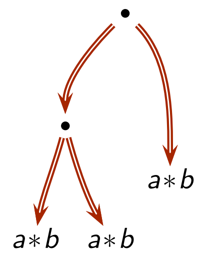


Liveness

Sequence of blocks $(b_1, b_2, \ldots, b_k)$ which is a prefix of some potential execution path starting at b_1 such that:

- b_k contains an upwards exposed use of v, **and**
- no other block on the path contains an assignment to v.

Data Flow Paths Discovered by Data Flow Analysis



Anticipability

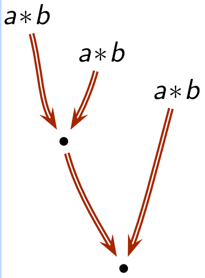
Sequence of blocks $(b_1, b_2, \ldots, b_k)$ which is a prefix of some potential execution path starting at b_1 such that:

- b_k contains an upwards exposed use of $a * b$, **and**
- no other block on the path contains an assignment to a or b, **and**
- every path starting at b_1 is an anticipability path of $a * b$.

Data Flow Paths Discovered by Data Flow Analysis

Sequence of blocks $(b_1, b_2, \ldots, b_k)$ which is a prefix of some potential execution path starting at b_1 such that:

- b_1 contains a downwards exposed use of $a * b$, **and**
- no other block on the path contains an assignment to a or b, **and**
- every path ending at b_k is an availability path of $a * b$.



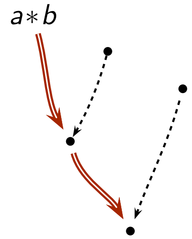
Availability

CS 618 Bit Vector Frameworks: Common Features of Bit Vector Frameworks 74/93

Data Flow Paths Discovered by Data Flow Analysis

Sequence of blocks $(b_1, b_2, \ldots, b_k)$ which is a prefix of some potential execution path starting at b_1 such that:

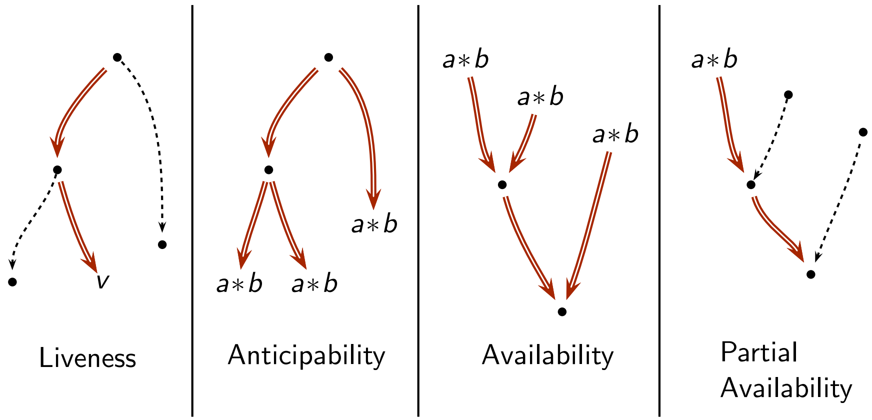
- b_1 contains a downwards exposed use of $a * b$, **and**
- no other block on the path contains an assignment to a or b.



Partial Availability

CS 618 Bit Vector Frameworks: Common Features of Bit Vector Frameworks 74/93

Data Flow Paths Discovered by Data Flow Analysis

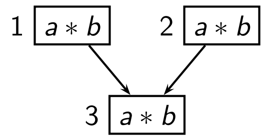


Part 10

Partial Redundancy Elimination

Partial Redundancy Elimination

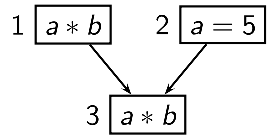
- Precursor: Common Subexpression Elimination (CSE)



- a and b are not modified along paths $1 \longrightarrow 3$ and $2 \longrightarrow 3$
- Computation of $a * b$ in 3 is redundant
- Previous value can be used

Partial Redundancy Elimination

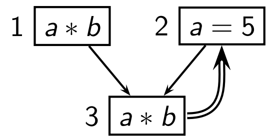
- Motivation: Overcoming the limitation of Common Subexpression Elimination (CSE)



- Computation of $a * b$ in 3 is
  - redundant along path $1 \longrightarrow 3$, but ...
  - not redundant along path $2 \longrightarrow 3$

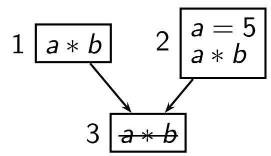
Partial Redundancy Elimination

- The key idea: Code Hoisting



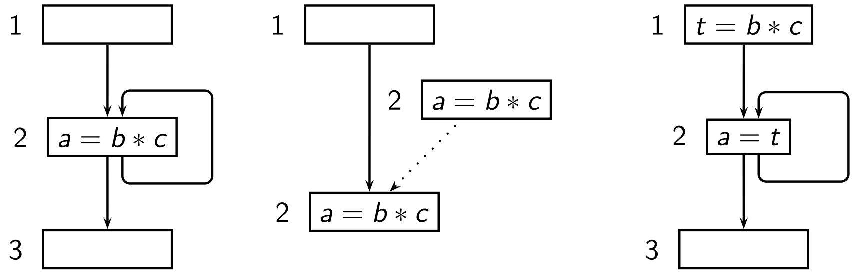
Partial Redundancy Elimination

- The key idea: Code Hoisting

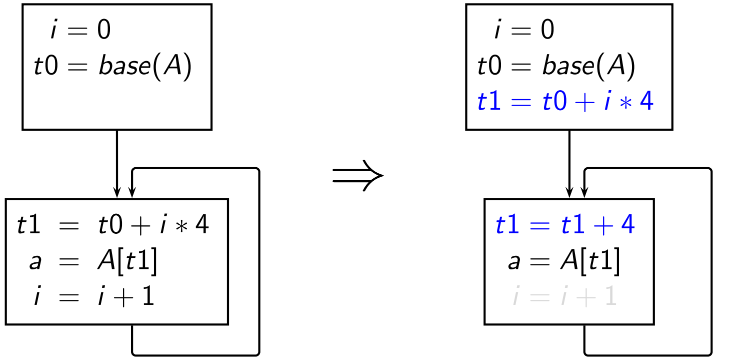


- Computation of $a * b$ in 3 becomes totally redundant
- Can be deleted

PRE Subsumes Loop Invariant Movement

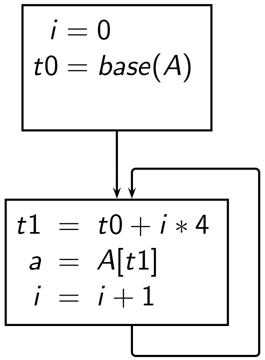


PRE Can be Used for Strength Reduction



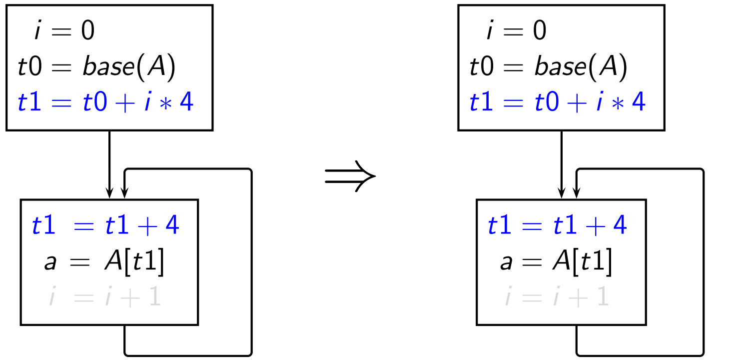
- $*$ and $+$ in the loop have been replaced by $+$
- $i = i + 1$ in the loop has been eliminated

PRE Can be Used for Strength Reduction



- Delete $i = i + 1$
- Expression $t0 + i * 4$ becomes loop invariant
- Hoist it and increment $t1$ in the loop

PRE Can be Used for Strength Reduction



- $*$ and $+$ in the loop have been replaced by $+$
- $i = i + 1$ in the loop has been eliminated

Performing Partial Redundancy Elimination

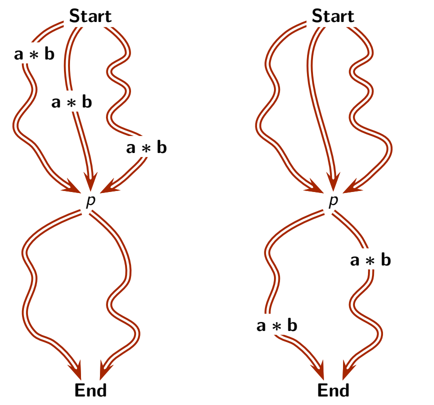
1. Identify partial redundancies
2. Identify program points where computations can be inserted
3. Insert expressions
4. Partial redundancies become total redundancies
   $\Longrightarrow$ Delete them.

Morel-Renvoise Algorithm (*CACM*, 1979.)

Defining Hoisting Criteria

- An expression can be safely inserted at a program point p if it is

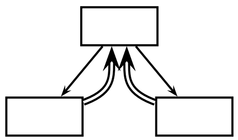
Available at p Anticipable at p



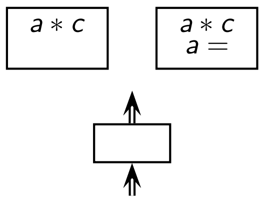
- If it is available at p, then there is no need to insert it at p.
- If it is anticipable at p then all such occurrence should be hoisted to p.
- *An expression should be hoisted to p provided it can be hoisted to p along all paths from p to exit.*

Hoisting Criteria

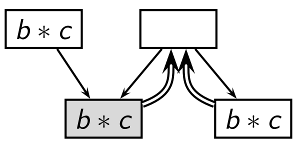
- *Safety of hoisting to the exit of a block*.
  - S.1 Should be hoisted only if it can be hoisted to the entry of all succesors



- *Safety of hoisting to the entry of a block*.
  Should be hoisted only if
  - S.2 it is upwards exposed, or
  - S.3 it can be hoisted to its exit and is transparent in the block



- *Desirability of hoisting to the entry of a block*.
  Should be hoisted only if
  - D.1 it is partially available, and
  - D.2 For each predecessor
    - D.2.a it is hoisted to its exit, or
    - D.2.b is available at its exit.



Applying the Hoisting Criteria

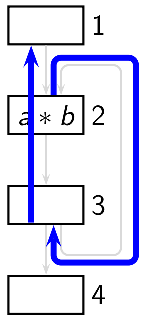
- *Safety of hoisting to the exit of a block.*
  - S.1 Should be hoisted only if it can be hoisted to the entry of all succesors
- *Safety of hoisting to the entry of a block.*
  Should be hoisted only if
  - S.2 it is upwards exposed, or
  - S.3 it can be hoisted to its exit an[d is transparent] in the block
- *Desirability of hoisting to the en[try of a block.]*
  Should be hoisted only if
  - D.1 it is partially available, and
  - D.2 For each predecessor
    - D.2.a it is hoisted to its exit, or
    - D.2.b is available at its exit.

What does this slide show?

- Four examples
- For each example
  - statements in blue enable hoisting
  - statements in red prohibit hoisting

Applying the Hoisting Criteria

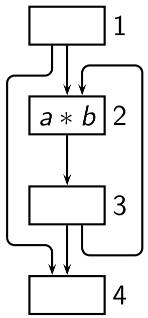
- *Safety of hoisting to the exit of a block.*
  - S.1 Should be hoisted only if it can be hoisted to the entry of all succesors
- *Safety of hoisting to the entry of a block.*
  Should be hoisted only if
  - S.2 it is upwards exposed, or
  - S.3 it can be hoisted to its exit and is transparent in the block
- *Desirability of hoisting to the entry of a block.*
  Should be hoisted only if
  - D.1 it is partially available, and
  - D.2 For each predecessor
    - D.2.a it is hoisted to its exit, or
    - D.2.b is available at its exit.



(Example 1)

Applying the Hoisting Criteria

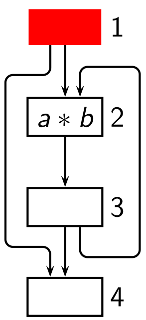
- *Safety of hoisting to the exit of a block.*
  - S.1 Should be hoisted only if it can be hoisted to the entry of all succesors
- *Safety of hoisting to the entry of a block.*
  Should be hoisted only if
  - S.2 it is upwards exposed, or
  - S.3 it can be hoisted to its exit and is transparent in the block
- *Desirability of hoisting to the entry of a block.*
  Should be hoisted only if
  - D.1 it is partially available, and
  - D.2 For each predecessor
    - D.2.a it is hoisted to its exit, or
    - D.2.b is available at its exit.



(Example 2)

Applying the Hoisting Criteria

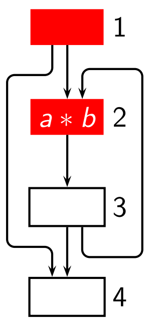
- *Safety of hoisting to the exit of a block.*
  - S.1 Should be hoisted only if it can be hoisted to the entry of all succesors
- *Safety of hoisting to the entry of a block.*
  Should be hoisted only if
  - S.2 it is upwards exposed, or
  - S.3 it can be hoisted to its exit and is transparent in the block
- *Desirability of hoisting to the entry of a block.*
  Should be hoisted only if
  - D.1 it is partially available, and
  - D.2 For each predecessor
    - D.2.a it is hoisted to its exit, or
    - D.2.b is available at its exit.



(Example 2)

Applying the Hoisting Criteria

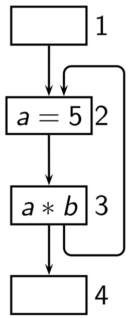
- *Safety of hoisting to the exit of a block.*
  - S.1 Should be hoisted only if it can be hoisted to the entry of all succesors
- *Safety of hoisting to the entry of a block.*
  Should be hoisted only if
  - S.2 it is upwards exposed, or
  - S.3 it can be hoisted to its exit and is transparent in the block
- *Desirability of hoisting to the entry of a block.*
  Should be hoisted only if
  - D.1 it is partially available, and
  - D.2 For each predecessor
    - D.2.a it is hoisted to its exit, or
    - D.2.b is available at its exit.



(Example 2)

Applying the Hoisting Criteria

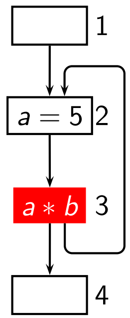
- *Safety of hoisting to the exit of a block.*
  - S.1 Should be hoisted only if it can be hoisted to the entry of all succesors
- *Safety of hoisting to the entry of a block.*
  Should be hoisted only if
  - S.2 it is upwards exposed, or
  - S.3 it can be hoisted to its exit and is transparent in the block
- *Desirability of hoisting to the entry of a block.*
  Should be hoisted only if
  - D.1 it is partially available, and
  - D.2 For each predecessor
    - D.2.a it is hoisted to its exit, or
    - D.2.b is available at its exit.



(Example 3)

Applying the Hoisting Criteria

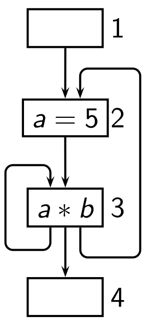
- *Safety of hoisting to the exit of a block.*
  - S.1 Should be hoisted only if it can be hoisted to the entry of all succesors
- *Safety of hoisting to the entry of a block.*
  Should be hoisted only if
  - S.2 it is upwards exposed, or
  - S.3 it can be hoisted to its exit and is transparent in the block
- *Desirability of hoisting to the entry of a block.*
  Should be hoisted only if
  - D.1 it is partially available, and
  - D.2 For each predecessor
    - D.2.a it is hoisted to its exit, or
    - D.2.b is available at its exit.



(Example 3)

Applying the Hoisting Criteria

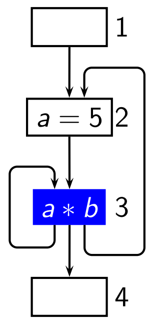
- S.1 Should be hoisted only if it can be hoisted to the entry of all succesors
- *Safety of hoisting to the entry of a block.*
  Should be hoisted only if
  - S.2 it is upwards exposed, or
  - S.3 it can be hoisted to its exit and is transparent in the block
- *Desirability of hoisting to the entry of a block.*
  Should be hoisted only if
  - D.1 it is partially available, and
  - D.2 For each predecessor
    - D.2.a it is hoisted to its exit, or
    - D.2.b is available at its exit.



(Example 4)

Applying the Hoisting Criteria

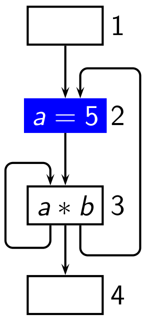
- *Safety of hoisting to the exit of a block.*
  - S.1 Should be hoisted only if it can be hoisted to the entry of all succesors
- *Safety of hoisting to the entry of a block.*
  Should be hoisted only if
  - S.2 it is upwards exposed, or
  - S.3 it can be hoisted to its exit and is transparent in the block
- *Desirability of hoisting to the entry of a block.*
  Should be hoisted only if
  - D.1 it is partially available, and
  - D.2 For each predecessor
    - D.2.a it is hoisted to its exit, or
    - D.2.b is available at its exit.



(Example 4)

Applying the Hoisting Criteria

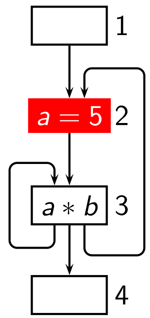
- *Safety of hoisting to the exit of a block.*
  - S.1 Should be hoisted only if it can be hoisted to the entry of all succesors
- *Safety of hoisting to the entry of a block.*
  Should be hoisted only if
  - S.2 it is upwards exposed, or
  - S.3 it can be hoisted to its exit and is transparent in the block
- *Desirability of hoisting to the entry of a block.*
  Should be hoisted only if
  - D.1 it is partially available, and
  - D.2 For each predecessor
    - D.2.a it is hoisted to its exit, or
    - D.2.b is available at its exit.



(Example 4)

Applying the Hoisting Criteria

- *Safety of hoisting to the exit of a block.*
  - S.1 Should be hoisted only if it can be hoisted to the entry of all succesors
- *Safety of hoisting to the entry of a block.*
  Should be hoisted only if
  - S.2 it is upwards exposed, or
  - S.3 it can be hoisted to its exit and is transparent in the block
- *Desirability of hoisting to the entry of a block.*
  Should be hoisted only if
  - D.1 it is partially available, and
  - D.2 For each predecessor
    - D.2.a it is hoisted to its exit, or
    - D.2.b is available at its exit.



(Example 4)

First Level Global Data Flow Properties in PRE

- Partial Availability.

$$PavIn_n = \begin{cases} BI & n \text{ is } Start \text{ block} \\ \bigcup_{p \in pred(n)} PavOut_p & \text{otherwise} \end{cases}$$

$$PavOut_n = Gen_n \cup (PavIn_n - Kill_n)$$

- Total Availability.

$$AvIn_n = \begin{cases} BI & n \text{ is } Start \text{ block} \\ \bigcap_{p \in pred(n)} AvOut_p & \text{otherwise} \end{cases}$$

$$AvOut_n = Gen_n \cup (AvIn_n - Kill_n)$$

PRE Data Flow Equations

Desirability: D.1

$$In_n = PavIn_n \cap \left(AntGen_n \cup (Out_n - Kill_n) \right) \bigcap_{p \in pred(n)} \left(Out_p \cup AvOut_p \right)$$

$$Out_n = \begin{cases} BI & n \text{ is } End \text{ block} \\ \bigcap_{s \in succ(n)} In_s & \text{otherwise} \end{cases}$$

Expressions should be partially available, and

PRE Data Flow Equations

Safety: S.2

$$In_n = PavIn_n \cap \Big(AntGen_n \cup (Out_n - Kill_n) \Big)$$
$$\bigcap_{p\in pred(n)} \Big(Out_p \cup AvOut_p \Big)$$
$$Out_n = \begin{cases} BI & n \text{ is } End \text{ block} \\ \bigcap_{s\in succ(n)} In_s & \text{otherwise} \end{cases}$$

Expressions should be upwards exposed, or

PRE Data Flow Equations

Safety: S.3

$$In_n = PavIn_n \cap \Big(AntGen_n \cup (Out_n - Kill_n) \Big)$$
$$\bigcap_{p\in pred(n)} \Big(Out_p \cup AvOut_p \Big)$$
$$Out_n = \begin{cases} BI & n \text{ is } End \text{ block} \\ \bigcap_{s\in succ(n)} In_s & \text{otherwise} \end{cases}$$

Expressions can be hoisted to the exit and are transparent in the block

PRE Data Flow Equations

Desirability: D.2.a

$$In_n = PavIn_n \cap \Big(AntGen_n \cup (Out_n - Kill_n) \Big)$$
$$\bigcap_{p\in pred(n)} \Big(Out_p \cup AvOut_p \Big)$$
$$Out_n = \begin{cases} BI & n \text{ is } End \text{ block} \\ \bigcap_{s\in succ(n)} In_s & \text{otherwise} \end{cases}$$

For every predecessor, expressions can be hoisted to its exit, or

PRE Data Flow Equations

Desirability: D.2.b

$$In_n = PavIn_n \cap \Big(AntGen_n \cup (Out_n - Kill_n) \Big)$$
$$\bigcap_{p\in pred(n)} \Big(Out_p \cup AvOut_p \Big)$$
$$Out_n = \begin{cases} BI & n \text{ is } End \text{ block} \\ \bigcap_{s\in succ(n)} In_s & \text{otherwise} \end{cases}$$

...expressions are available at the exit of the same predecessor

PRE Data Flow Equations

Safety: S.1

$$In_n = PavIn_n \cap \Big(AntGen_n \cup (Out_n - Kill_n) \Big)$$
$$\bigcap_{p \in pred(n)} \Big(Out_p \cup AvOut_p \Big)$$

$$Out_n = \begin{cases} BI & n \text{ is } End \text{ block} \\ \bigcap_{s \in succ(n)} In_s & \text{otherwise} \end{cases}$$

Expressions should be hoisted to the exit of a block if they can be hoisted to the entry of all succesors

Deletion Criteria in PRE

- An expression is redundant in node n if
  - it can be placed at the entry (i.e. can be "hoisted" out) of n, AND
  - it is upwards exposed in node n.

$$Redundant_n = In_n \cap AntGen_n$$

- A hoisting path for an expression e begins at n if $e \in Redundant_n$
- This hoisting path extends against the control flow.

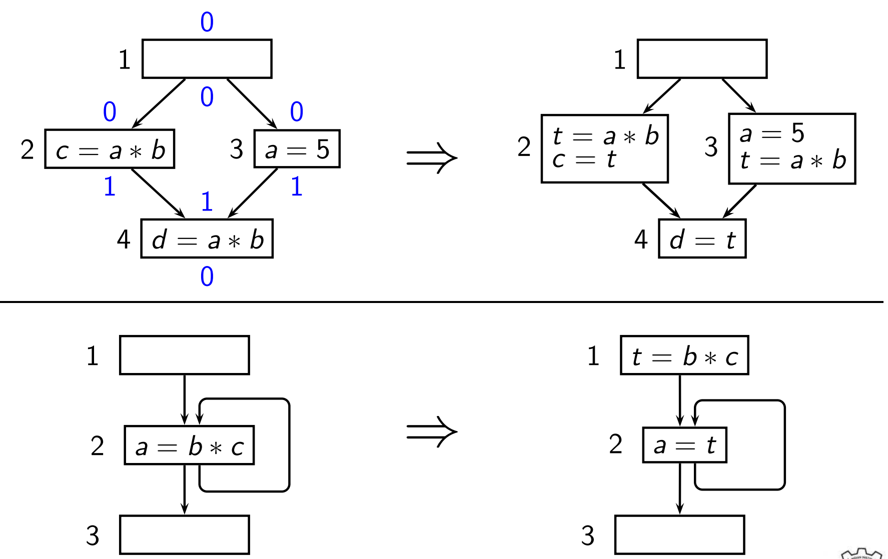
Insertion Criteria in PRE

- An expression is inserted at the exit of node n is
  - it can be placed at the exit of n, AND
  - it is not available at the exit of n, AND
  - it cannot be hoisted out of n, OR it is modified in n.

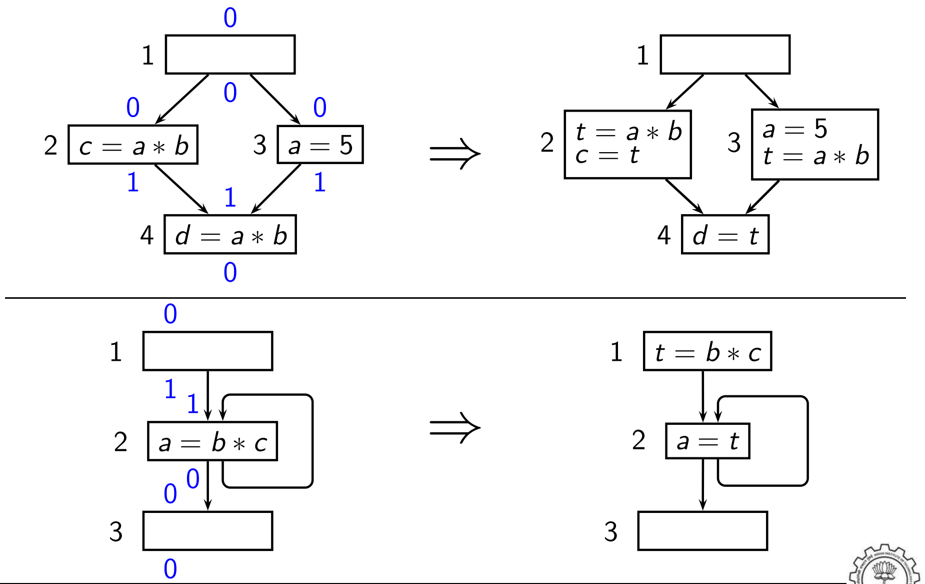
$$Insert_n = Out_n \cap (\neg AvOut_n) \cap (\neg In_n \cup Kill_n)$$

- A hoisting path for an expression e ends at n if $e \in Insert_n$

Performing PRE by Computing In/Out: Simple Cases

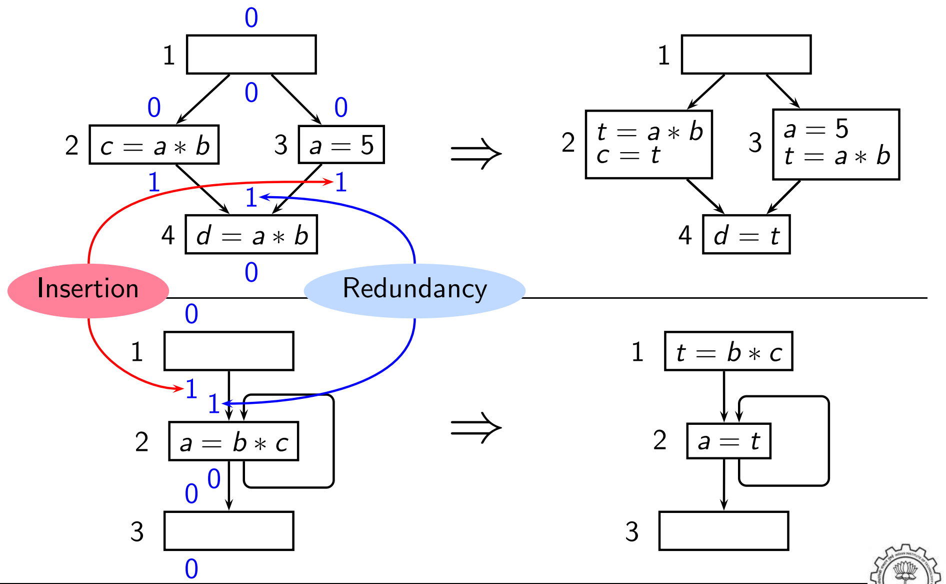


Performing PRE by Computing *In*/*Out*: Simple Cases



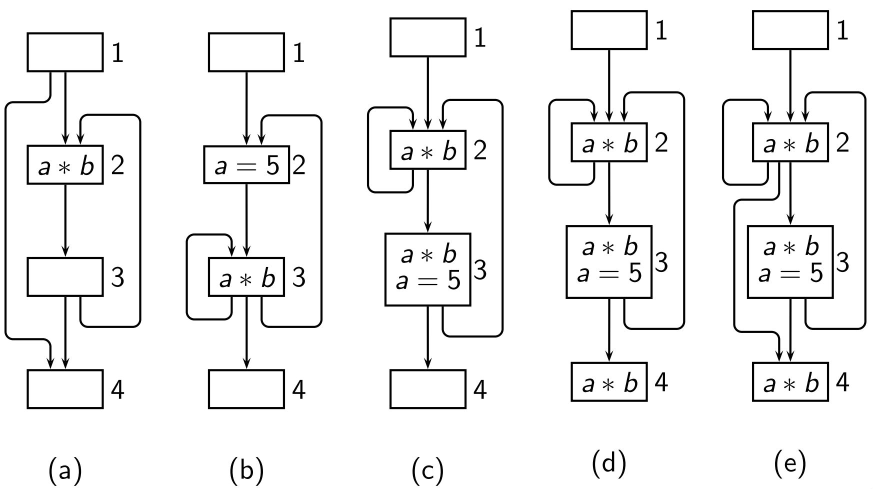
CS 618 Bit Vector Frameworks: Partial Redundancy Elimination 88/93

Performing PRE by Computing *In*/*Out*: Simple Cases



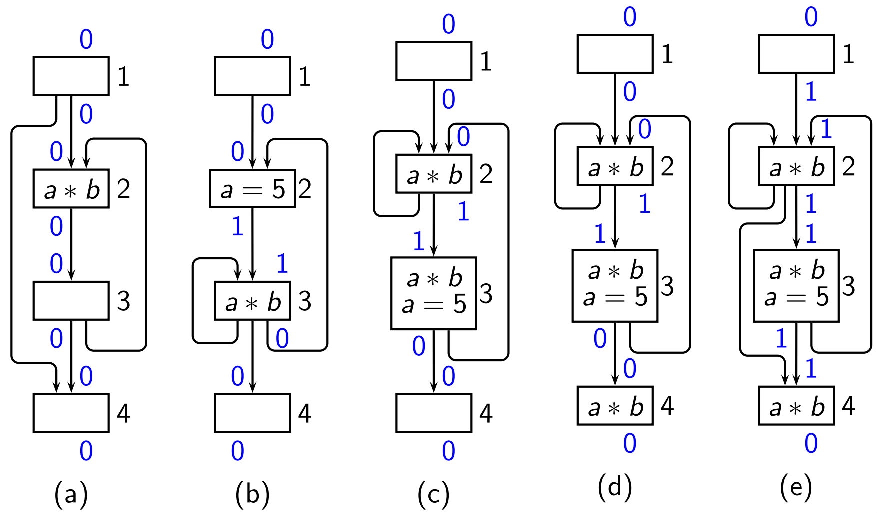
Jul 2010 IIT Bombay

Tutorial Problems for PRE



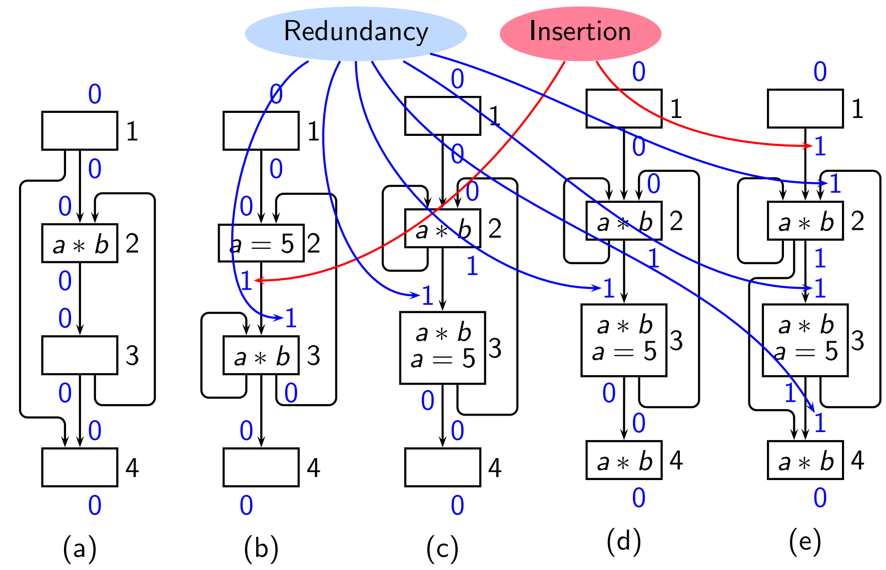
CS 618 Bit Vector Frameworks: Partial Redundancy Elimination 89/93

Tutorial Problems for PRE

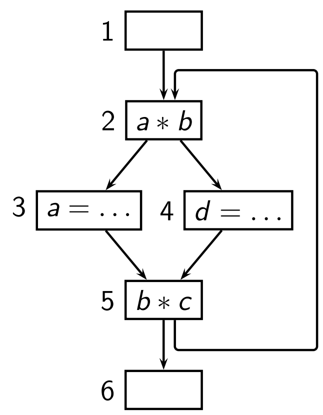


Jul 2010 IIT Bombay

Tutorial Problems for PRE



Further Tutorial Problem for PRE



Let $\{a * b, b * c\} \equiv$ bit string 11

Node n	$Kill_n$	$AntGen_n$	$PavIn_n$	$AvOut_n$
1	00	00	00	00
2	00	10	11	10
3	10	00	11	00
4	00	00	11	10
5	00	01	11	01
6	00	00	11	01

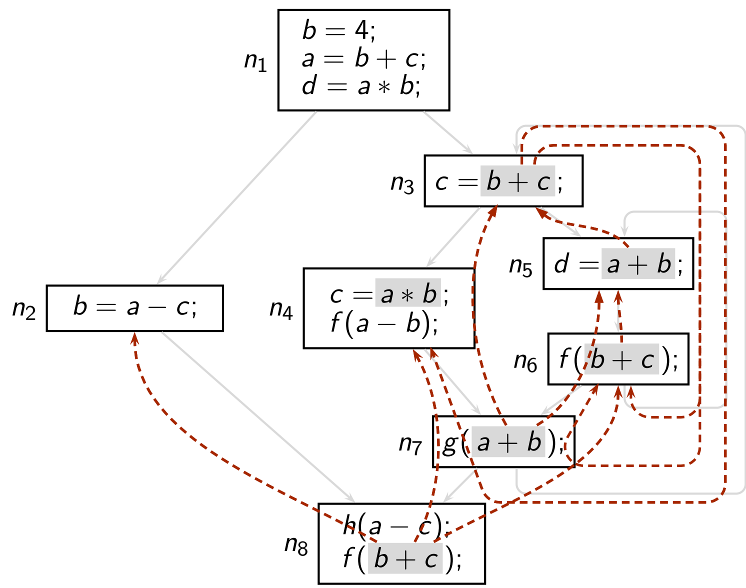
- Compute $In_n/Out_n/Redundant_n/Insert_n$
- Identify hoisting paths

Result of PRE Data Flow Analysis of the Running Example

Bit vector | $a * b$ | $a + b$ | $a - b$ | $a - c$ | $b + c$ |

Block	Global Information							
	Constant information		Iteration # 1		Changes in iteration # 2		Changes in iteration # 3	
	$PavIn_n$	$AvOut_n$	Out_n	In_n	Out_n	In_n	Out_n	In_n
n_8	11111	00011	00000	00011				00001
n_7	11101	11000	00011	01001	00001			
n_6	11101	11001	01001	01001			01000	
n_5	11101	11000	01001	01001		01000		
n_4	11100	10100	01001	11100		11000		
n_3	11101	10000	01000	01001		00001		
n_2	10001	00010	00011	00000			00001	
n_1	00000	10001	00000	00000				

Hoisting Paths for Some Expressions in the Running Example



Optimized Version of the Running Example