Machine-Independent Optimizations

Y.N. Srikant

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

NPTEL Course on Compiler Design

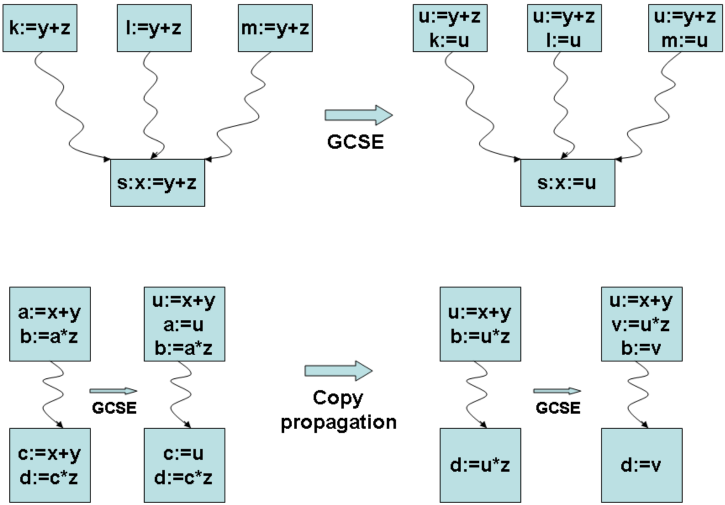
Outline of the Lecture

- Global common sub-expression elimination
- Copy propagation
- Loop invariant code motion
- Induction variable elimination and strength reduction
- Region based data-flow analysis

Elimination of Global Common Sub-expressions

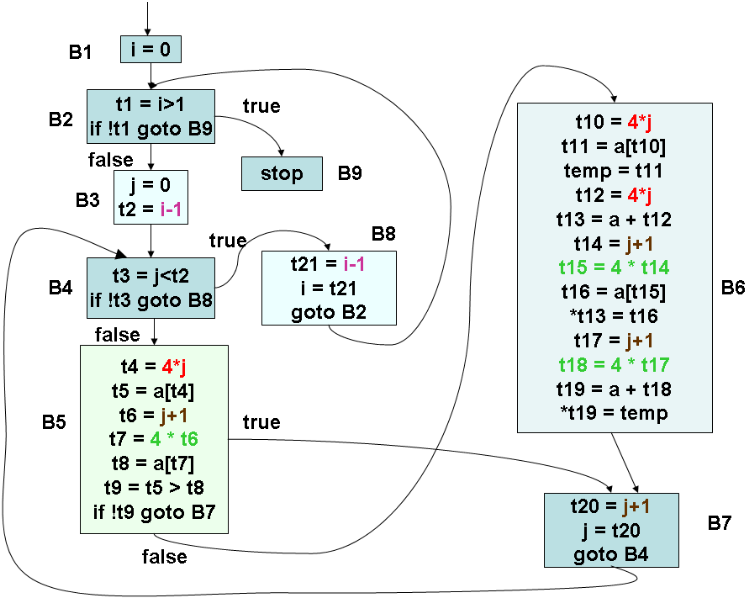
- Needs available expression information
- For every $s: \ x := y + z$, such that $y + z$ is available at the beginning of s' block, and neither y nor z is defined prior to s in that block, do the following
  1. Search backwards from s' block in the flow graph, and find first block in which $y + z$ is evaluated. We need not go *through* any block that evaluates $y + z$.
  2. Create a new variable u and replace each statement $w := y + z$ found in the above step by the code segment $\{u := y + z; w := u\}$, and replace s by $x := u$
  3. Repeat 1 and 2 above for every predecessor block of s' block
- Repeated application of GCSE may be needed to catch "deep" CSE

GCSE Conceptual Example

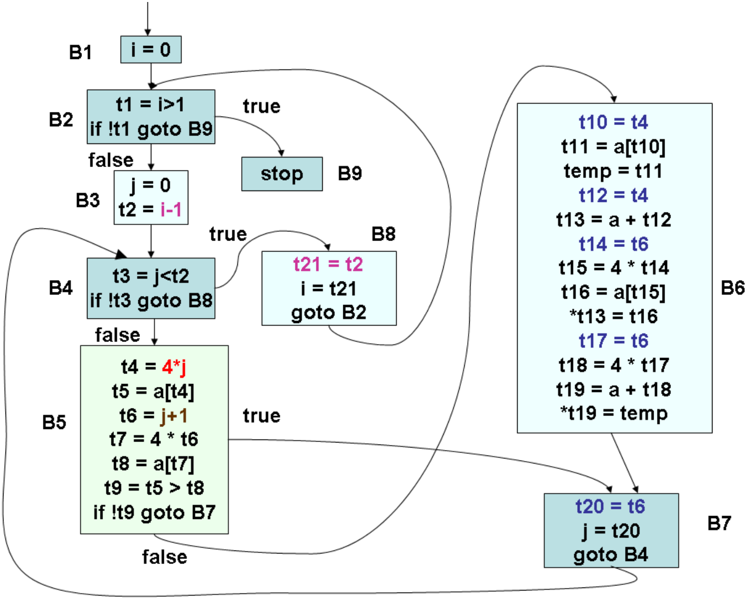


Demonstrating the need for repeated application of GCSE

GCSE on Running Example - 1



GCSE on Running Example - 2



Copy Propagation

- Eliminate copy statements of the form $s: \; x := y$, by substituting y for x in all uses of x reached by this copy
- Conditions to be checked
  1. u-d chain of use u of x must consist of s only. Then, s is the only definition of x reaching u
  2. On every path from s to u, including paths that go through u several times (but do not go through s a second time), there are no assignments to y. This ensures that the copy is valid
- The second condition above is checked by using information obtained by a new data-flow analysis problem
  - $c\_gen[B]$ is the set of all copy statements, $s: x := y$ in B, such that there are no subsequent assignments to either x or y within B, after s
  - $c\_kill[B]$ is the set of all copy statements, $s: x := y$, s not in B, such that either x or y is assigned a value in B
  - Let U be the universal set of all copy statements in the program

Copy Propagation - The Data-flow Equations

- $c\_in[B]$ is the set of all copy statements, $x := y$ reaching the beginning of B along every path such that there are no assignments to either x or y following the last occurrence of $x := y$ on the path
- $c\_out[B]$ is the set of all copy statements, $x := y$ reaching the end of B along every path such that there are no assignments to either x or y following the last occurrence of $x := y$ on the path

$$
\begin{aligned}
c\_in[B] &= \bigcap_{P\ is\ a\ predecessor\ of\ B} c\_out[P],\ B\ not\ initial \\
c\_out[B] &= c\_gen[B] \bigcup (c\_in[B] - c\_kill[B]) \\
c\_in[B1] &= \phi,\ where\ B1\ is\ the\ initial\ block \\
c\_out[B] &= U - c\_kill[B],\ for\ all\ B \neq B1\ (initialization\ only)
\end{aligned}
$$

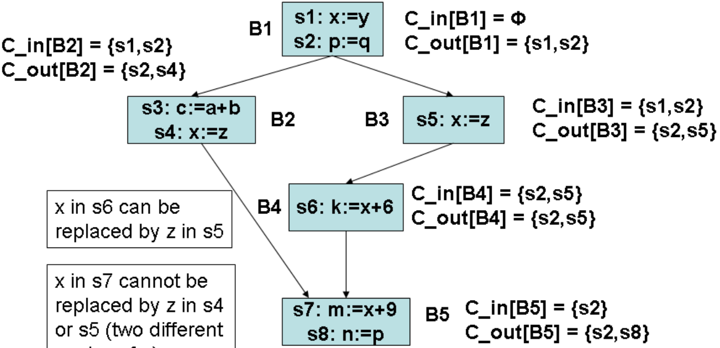
Algorithm for Copy Propagation

For each copy, $s : x := y$, do the following

1. Using the $du - chain$, determine those uses of x that are reached by s
2. For each use u of x found in (1) above, check that
   (i) u-d chain of u consists of s only
   (ii) s is in $c\_in[B]$, where B is the block to which u belongs. This ensures that
      - s is the only definition of x that reaches this block
      - No definitions of x or y appear on this path from s to B

   (iii) no definitions x or y occur within B prior to u found in (1) above
3. If s meets the conditions above, then remove s and replace all uses of x found in (1) above by y

Copy Propagation Example 1



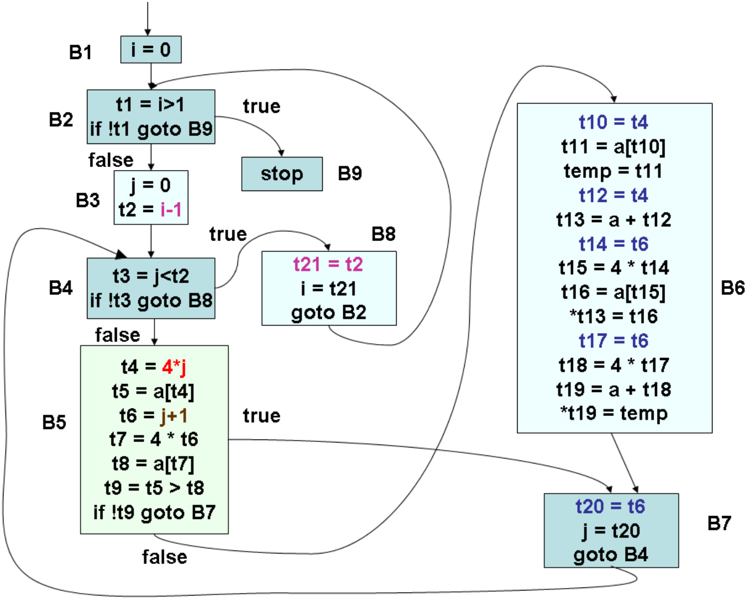
x in s6 can be replaced by z in s5

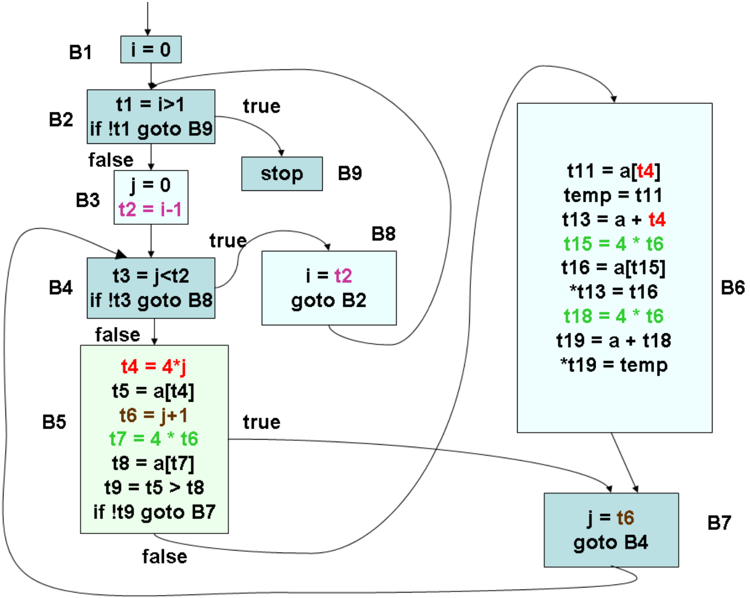
x in s7 cannot be replaced by z in s4 or s5 (two different copies of z)

p in s8 can be replaced by q in s2 (s2 reaches B5 thro' both the paths)

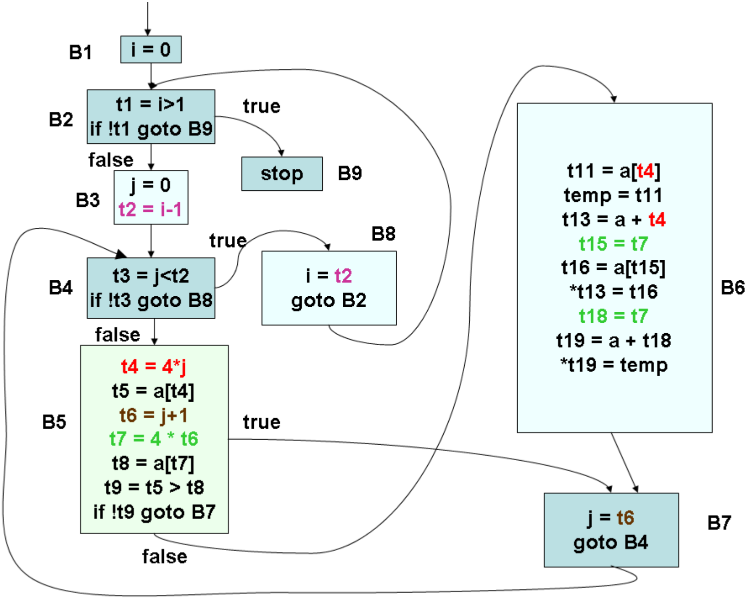
Adapted from "The Dragon Book" A-W 1986

Copy Propagation on Running Example 1.1

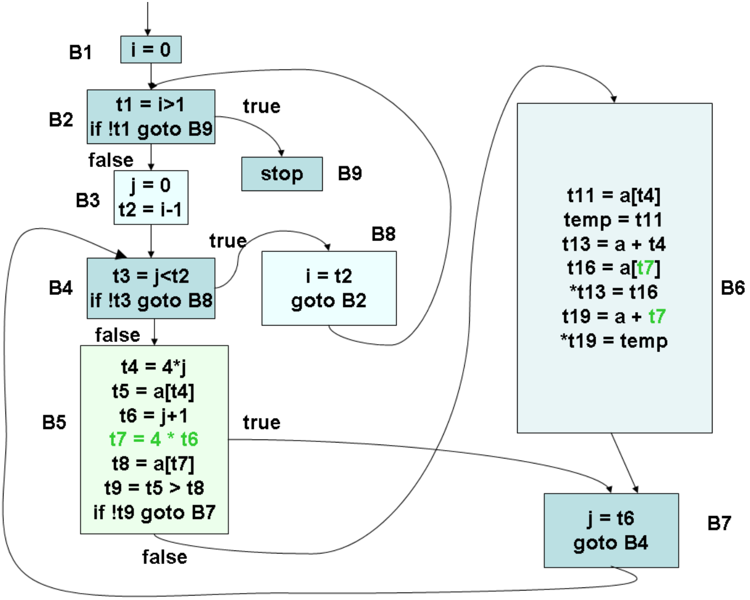



B1
i = 0
B2
t1 = i>1
if !t1 goto B9
true
stop
B9
false
B3
j = 0
t2 = i-1
true
B8
i = t2
goto B2
B4
t3 = j<t2
if !t3 goto B8
false
B5
t4 = 4*j
t5 = a[t4]
t6 = j+1
t7 = 4 * t6
t8 = a[t7]
t9 = t5 > t8
if !t9 goto B7
true
false
t11 = a[t4]
temp = t11
t13 = a + t4
t15 = 4 * t6
t16 = a[t15]
*t13 = t16
t18 = 4 * t6
t19 = a + t18
*t19 = temp
B6
j = t6
goto B4
B7

GCSE and Copy Propagation on Running Example 1.1



GCSE and Copy Propagation on Running Example 1.2



Detection of Loop-invariant Computations

Given a loop L, and the $u - d$ and $d - u$ chains

Mark as "invariant", those statements whose operands are all either constant or have all their reaching definitions outside L

Repeat {
  Mark as "invariant" all those statements not previously so marked all of whose operands are constants, or have all their reaching definitions outside L, or have exactly one reaching definition, and that definition is a statement in L marked "invariant"
} until no new statements are marked "invariant"

$u - d$ chains are useful in marking statements as "invariant"
$d - u$ chains are useful in examining all uses of a definition marked "invariant"

Loop Invariant Code motion Example

```
      t1 = 202
      i = 1
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t3 = addr(a)
      t4 = t3 - 4
      t5 = 4*i
      t6 = t4+t5
      *t6 = t1
      i = i+1
      goto L1
L2:
```

Before LIV code motion

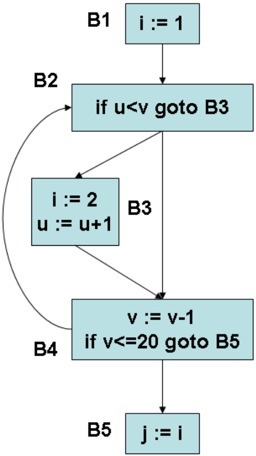
```
      t1 = 202
      i = 1
      t3 = addr(a)
      t4 = t3 - 4
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t5 = 4*i
      t6 = t4+t5
      *t6 = t1
      i = i+1
      goto L1
L2:
```

After LIV code motion

Loop-Invariant Code Motion Algorithm

1. Find loop-invariant statements
2. For each statement s defining x found in step (1), check that
   (a) it is in a block that dominates all exits of L
   (b) x is not defined elsewhere in L
   (c) all uses in L of x can only be reached by the definition of x in s
3. Move each statement s found in step (1) and satisfying conditions of step (2) to a newly created preheader
   - provided any operands of s that are defined in loop L have previously had their definition statements moved to the preheader
4. Update all the $u - d$ and $d - u$ chains appropriately

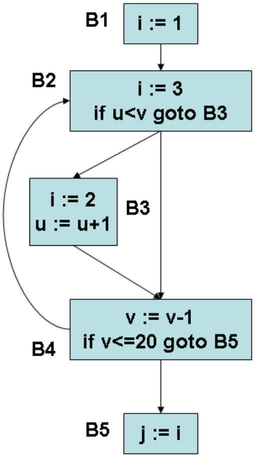
Code Motion - Violation of condition 2(a)



The statement i:=2 from B3 cannot be moved to a preheader since condition 2(a) is violated (B3 does not dominate B4)
The computation gets altered due to code movement
i always gets value 2, and never 1, and hence j always gets value 2

Condition 2(a):
s dominates all exits of *L*

Code Motion - Violation of condition 2(b)



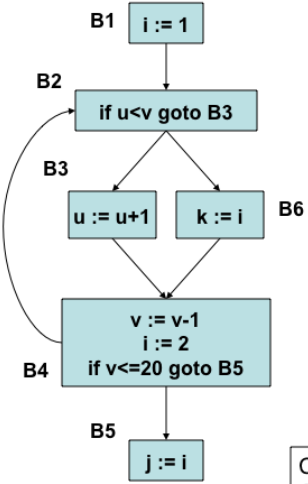
B2 dominates B4 and hence condition 2(a) is satisfied for i:=3 in B2. However statement i:=3 from B2 cannot be moved to a preheader since condition 2(b) is violated (i is defined in B3)

The computation gets altered due to code movement
If the loop is executed twice, i may pass its value of 3 from B2 to j in the original loop.
In the revised loop, i gets the value 2 in the second iteration and retains it forever

Condition 2(a):
s dominates all exits of *L*

Condition 2(b):
x is not defined elsewhere in *L*

Code Motion - Violation of condition 2(c)



Conditions 2(a) and 2(b) are satisfied. However statement i:=2 from B4 cannot be moved to a preheader since condition 2(c) is violated (use of i in B6 is reached by defs of i in B1 and B4)

The computation gets altered due to code movement
In the revised loop, i gets the value 2 from the def in the preheader and k becomes 2.
However, k could have received the value of either 1 (from B1) or 2 (from B4) in the original loop

Condition 2(a): *s* dominates all exits of *L*
Condition 2(b): *x* is not defined elsewhere in *L*
Condition 2(c): All uses of *x* in *L* can only be reached by the definition of *x* in *s*

Induction Variables

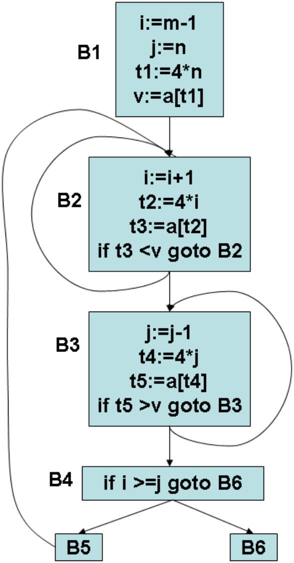
- An **induction variable** x of a loop L changes its value only through an increment or decrement operation by a constant amount
- **Basic induction variables**: variables i whose only assignments within a loop L are of the form $i := i \pm n$, where n is a constant
- Another variable j which is *defined only once* within L, and whose value is $c * i + d$ (linear function of i) is an $i.v.$ in the **family** of i
- We associate a triple (i, c, d) with j (c and d are constants), and i belongs to its own family with a triple $(i, 1, 0)$

Induction Variables - Example 1

```
      t1 = 202
      i = 1
      t3 = addr(a)
      t4 = t3 - 4
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t5 = 4*i
      t6 = t4+t5
      *t6 = t1
      i = i+1
      goto L1
L2:
```

i is a basic i.v. and
t5 is a derived i.v.
in the family of i

Induction Variables - Example 2



i and j are both basic i.v. in both inner and outer loops

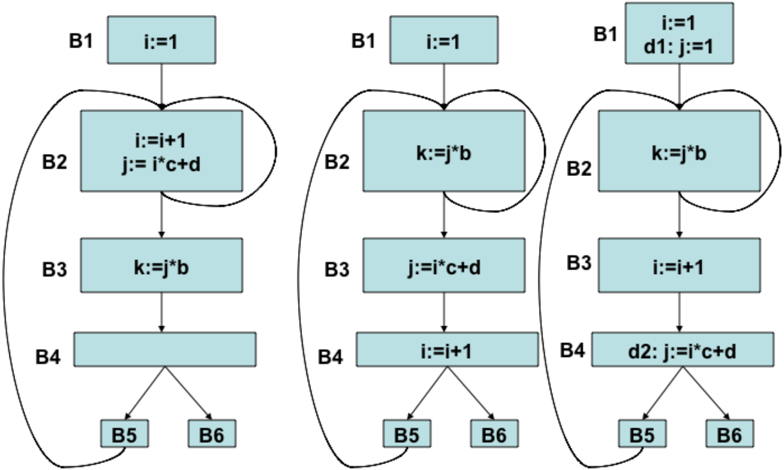
t2 (in the family of i) and t4 (in the family of j) are both derived i.v. in both inner and outer loops

Detection of Induction Variables

We need a loop L, reaching definitions, and loop-invariant computation information

1. Find all the basic *i.v.*, by scanning the statements of L
2. Search for variables k, with a single assignment to k within L, having one of the following forms:
   $k := j * b$, $k := b * j$, $k := j/b$, $k := j \pm b$, $k := b \pm j$, $k := j * b \pm a$, $k := a \pm j * b$, where b is a constant and j is an i.v., basic or otherwise
   (a) If j is basic, then for $k := j * b$, the triple for k is $(j, b, 0)$ (similarly for other forms)
   (b) If j is not basic, then let its triple be (i, c, d). We need to check two more conditions
      (i) there is no assignment to i between the lone point of assignment to j in L and the assignment to k
      (ii) no definition of j outside L reaches k

Induction Variables - Conditions



Conditions 2.b.i and 2.b.ii are both satisfied

Condition 2.b.i is not satisfied; value of j in B2 is not up-to-date

Condition 2.b.i is satisfied, but 2.b.ii is not satisfied. Both d1 and d2 reach k in B2

Detection of Induction Variables (2)

- If both j and k are temporaries in the same block, then checking the conditions (i) and (ii) above is easy
- Otherwise, we need to find all the basic blocks on the paths from the point of assignment to j, to the point of assignment to k, and check condition (i)
- Condition (ii) can be checked using u-d chain of j in the assignment to k
- Triple for k can be computed from (i, c, d) and the form of assignment to k
  - If $k := j * b$ and j is $i * c + d$,
    $k = (i * c + d) * b = (i * b * c) + (d * b)$
  - Hence the triple for k is $(i, b * c, d * b)$
  - Note that $b * c$ and $d * b$ are constants and can be evaluated by the compiler

Strength Reduction

Consider each basic IV, i in turn. For each IV j in the family of i, with triple (i, c, d) do the following

1. Create a new variable s and replace the assignment to j by $j := s$ (for two IVs, j_1 and j_2, with the same triples, create a single variable)
2. Immediately after each assignment $i := i + n$ in L, where n is a constant, append $s := s + c * n$ (note that $c * n$ is a constant)
3. Place s in the family of i with the triple (i, c, d). We have replaced a costly * operation by a cheaper + operation
4. Place the code to initialize s to $c * i + d$ at the end of the preheader

Induction Variables - Strength Reduction Ex 1

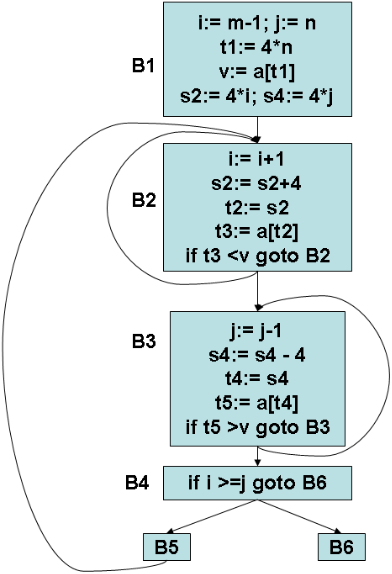
```
      t1 = 202
      i = 1
      t3 = addr(a)
      t4 = t3 - 4
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t5 = 4*i
      t6 = t4+t5
      *t6 = t1
      i = i+1
      goto L1
L2:
```

Before strength reduction for t5

```
      t1 = 202
      i = 1
      t3 = addr(a)
      t4 = t3 – 4
      t7 = 4
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t5 = t7
      t6 = t4+t5
      *t6 = t1
      i = i+1
      t7 = t7 + 4
      goto L1
L2:
```

After strength reduction for t5

Induction Variables - Strength Reduction Ex 2



Elimination of Induction Variables

- Consider each basic IV *i* whose only uses are to compute other IV in its family and in conditional branches
- Consider *j* in *i*'s family with the triple (i, c, d)
- Replace *if i relop x goto B* by the code sequence $\{r := c * x;\ r := r + d;\ if\ j\ relop\ r\ goto\ B\}$
- If *c* is negative, then we use $\overline{relop}$ in place of *relop* in the above code sequence
  - For example, if *c* is -4, then $if\ i \geq x\ goto\ B$ is replaced by the code sequence, $\{r := -4 * x;\ r := r + d;\ if\ j \leq r\ goto\ B\}$
- Delete all assignments to the eliminated IV in loop *L*
- Apply copy propagation (to eliminate statements $j := s$)

Induction Variable Elimination

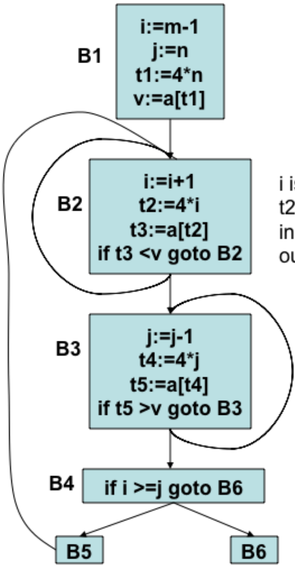
```
       t1 = 202
       i = 1
       t3 = addr(a)
       t4 = t3 – 4
       t7 = 4
L1:    t2 = i>100
       if t2 goto L2
       t1 = t1-2
       t6 = t4+t7
       *t6 = t1
       i = i+1
       t7 = t7 + 4
       goto L1
L2:
```

Before induction variable elimination (i)

```
       t1 = 202
       t3 = addr(a)
       t4 = t3 – 4
       t7 = 4
L1:    t2 = t7 >400
       if t2 goto L2
       t1 = t1-2
       t6 = t4+t7
       *t6 = t1
       t7 = t7 + 4
       goto L1
L2:
```

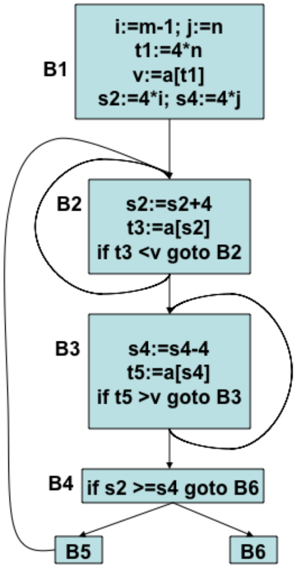
After eliminating i and replacing it with t7

Induction Variable Elimination and Strength Reduction

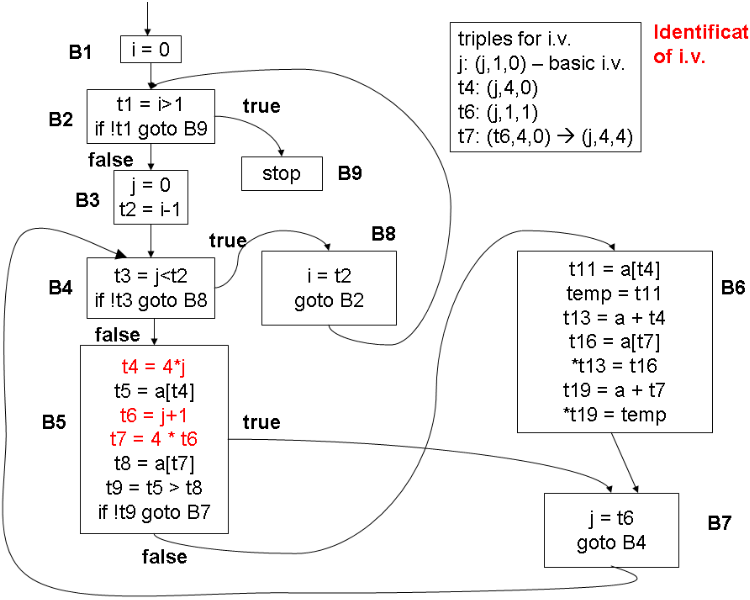


i is basic i.v
t2 is derived i.v
in both inner and
outer loops

j is basic i.v
t4 is derived i.v
in both inner
and outer loops



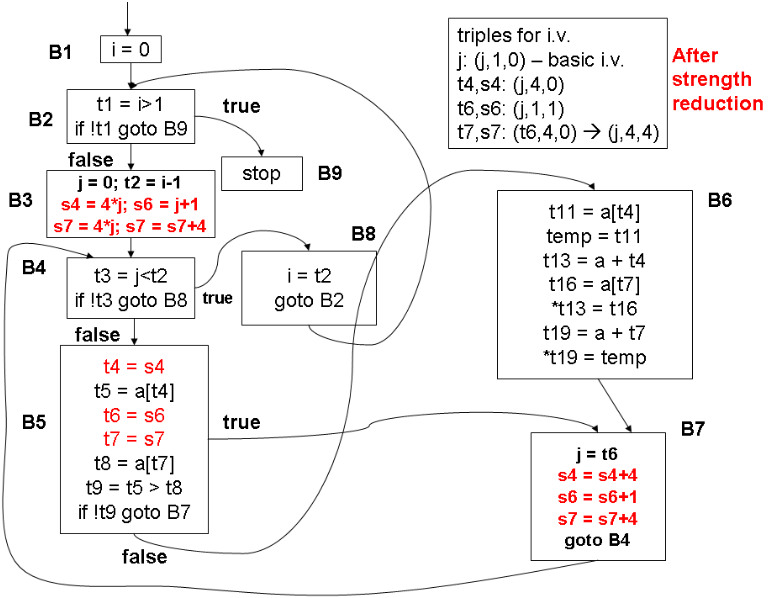
I.V. Detection - Running Example

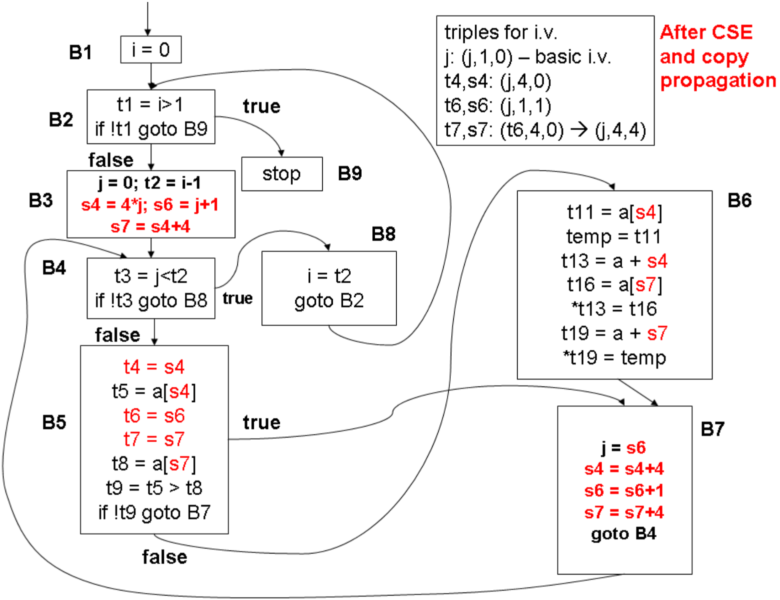


triples for i.v.
j: (j,1,0) – basic i.v.
t4: (j,4,0)
t6: (j,1,1)
t7: (t6,4,0) → (j,4,4)

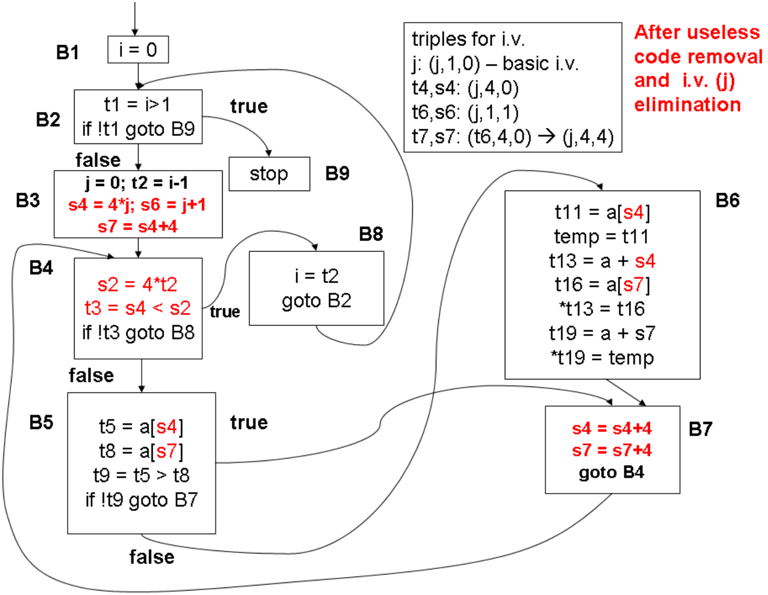
Identification of i.v.

I.V. Strength Reduction - Running Example




B1
i = 0
B2
t1 = i>1
if !t1 goto B9
true
false
stop
B9
B3
j = 0; t2 = i-1
s4 = 4*j; s6 = j+1
s7 = s4+4
B8
i = t2
goto B2
B4
t3 = j<t2
if !t3 goto B8
true
false
B5
t4 = s4
t5 = a[s4]
t6 = s6
t7 = s7
t8 = a[s7]
t9 = t5 > t8
if !t9 goto B7
true
false
B6
t11 = a[s4]
temp = t11
t13 = a + s4
t16 = a[s7]
*t13 = t16
t19 = a + s7
*t19 = temp
B7
j = s6
s4 = s4+4
s6 = s6+1
s7 = s7+4
goto B4
triples for i.v.
j: (j,1,0) – basic i.v.
t4,s4: (j,4,0)
t6,s6: (j,1,1)
t7,s7: (t6,4,0) → (j,4,4)
After CSE and copy propagation

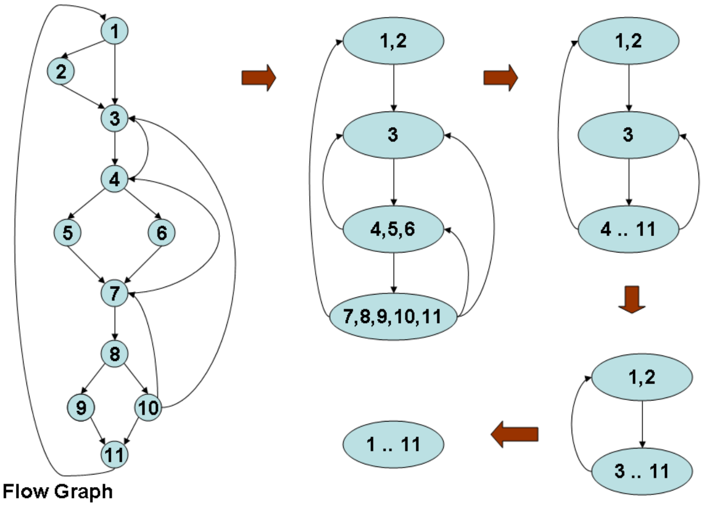
I.V. Elimination - Running Example



Region Based Data-flow Analysis

- **Region**: A set of nodes N that includes a header, which dominates all other nodes in the region
- All edges between nodes in N are in the region, except (possibly) for some of those that enter the header
- All intervals are regions but there are regions that are not intervals
  - A region may omit some nodes that an interval would include or they may omit some edges back to the header
  - For example, $I(7) = \{7, 8, 9, 10, 11\}$, but $\{8, 9, 10\}$ could be a region (see next slide)
- A region may have multiple exits
- We shall compute $gen_{R,B}$ and $kill_{R,B}$ of definitions generated and killed (resp.), along paths within the region R, from the header to the end of the block B

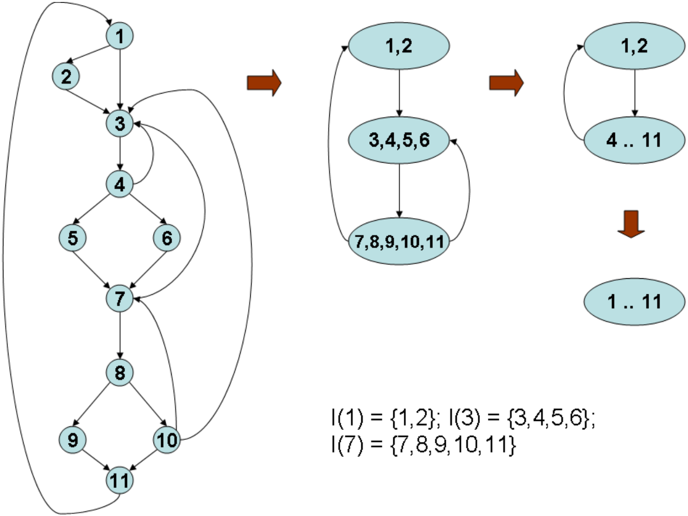
Intervals and Regions



I(1) = {1,2}; I(3) = {3}
I(4) = {4,5,6}; I(7) = {7,8,9,10,11}

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

Intervals and Regions



I(1) = {1,2}; I(3) = {3,4,5,6};
I(7) = {7,8,9,10,11}

Flow Graph

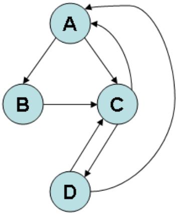
Region Based Data-flow Analysis (2)

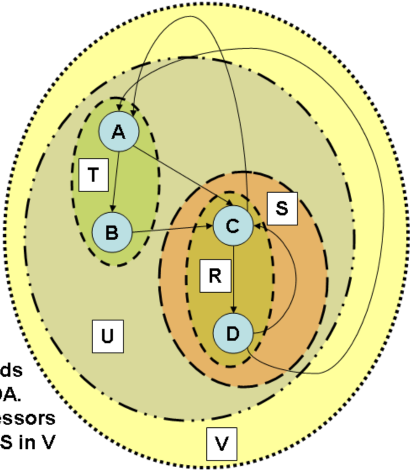
- These will be used to define a transfer function $trans_{R,B}(S)$, that tells for any set S of definitions, what subset of definitions reach the end of B by travelling along paths wholly within R, assuming that all and only the definitions in S reach the header of R
- $trans_{R,B}(S) = gen_{R,B} \bigcup (S - kill_{R,B})$
- $trans_{U,B}(\phi) = OUT[B] = gen_{U,B}$, where U is the region consisting of the entire flow graph
- We need to provide a method to compute the transfer functions $trans_{R,B}$, for progressively larger regions defined by some $(T_1 - T_2)$ transformation of a CFG
- Since $OUT[B] = gen_{U,B}$, we need to compute only $gen_{R,B}$ and $kill_{R,B}$, for each basic block, for progressively larger regions
- Interestingly, this approach does not compute $IN[B]$ at all

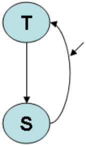
Region Based Data-flow Analysis (3)

- As we reduce a flow graph G by T_1 and T_2 transformations, at all times, the following conditions are true
  1. A node represents a region of G
  2. An edge from a to b in a reduced graph represents a set of edges
  3. Each node and edge of G is represented by exactly one node or edge of the current graph
- Region based DFA can be compared to *syntax-directed translation*, with the structure being provided by the hierarchy of regions
- We consider data-flow analysis for *reaching definitions*
- It should be emphasized that all data-flow values which reach the header of a region will surely flow to all the constituent regions and basic blocks, since all basic blocks are reacheable from the header of the enclosing region

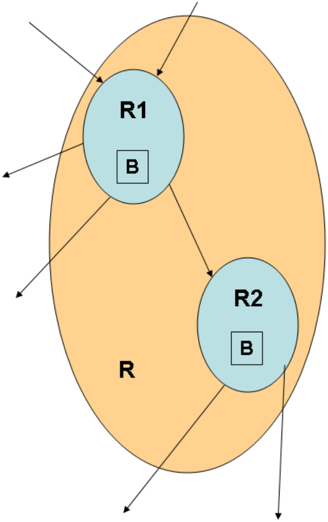
Region Example







This arc corresponds to 2 arcs, CA and DA. Hence, the predecessors of T, the header of S in V are C and D



For reaching definitions problem

Basic regions

$gen_{B,B} = gen[B]$
$kill_{B,B} = kill[B]$

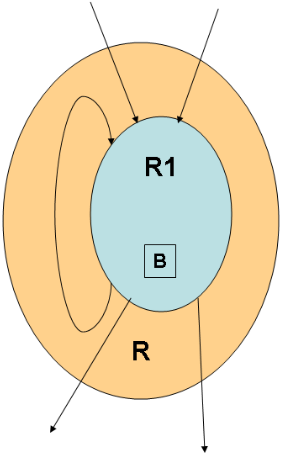
Region building by T2

For basic blocks B within R1,
$gen_{R,B} = gen_{R1,B}$
$kill_{R,B} = kill_{R1,B}$

Edges from R2 to header of R1 are not part of R

For basic blocks B within R2,
$gen_{R,B} = gen_{R2,B} \cup (G - kill_{R2,B})$
$kill_{R,B} = kill_{R2,B} \cup (K - gen_{R2,B})$
where, $G = \bigcup gen_{R1,P}$, and
$K = \bigcap kill_{R1,P}$
for all predecessors P of the header of R2 in R1

Region Building by T1 Trans. - Reaching Def



For reaching definitions problem

<u>Region building by T1</u>

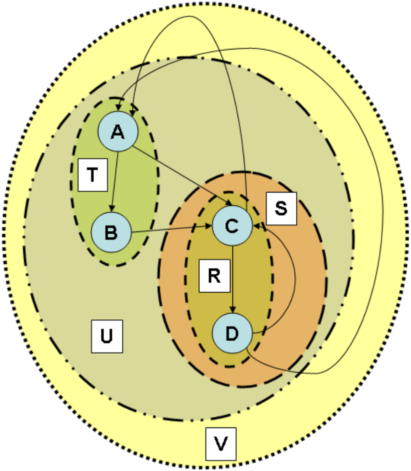
$gen_{R,B} = gen_{R1,B}$ **U** $(G - kill_{R1,B})$
$kill_{R,B} = kill_{R1,B}$

where, G = **U** $gen_{R1,P}$, for all predecessors P of the header of R1 in R

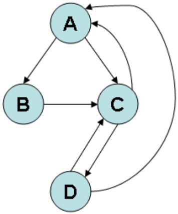
It is not necessary to compute $kill_{R,B}$ as in the previous case (T2).

A definition gets killed going from the header to B iff it is killed along all acyclic paths, and hence back edges incorporated into R will not cause more definitions to be killed

Region Based RD Analysis - An Example (1)

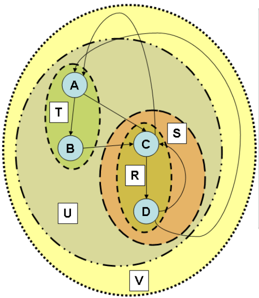


Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000



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

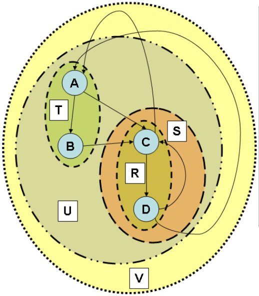
Region Based RD Analysis - An Example (2)



Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

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

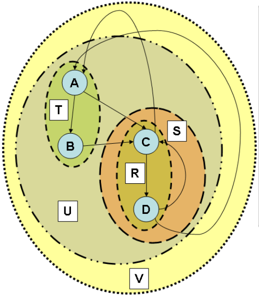
- Building region R from regions C and D by T2 transf.
- $gen_{R,C} = gen_{C,C} = 000$; $kill_{R,C} = kill_{C,C} = 010$
- Header of D is D and pred. of D in C is C
- $G = gen_{C,C} = 000$ and $K = kill_{C,C} = 010$
- $gen_{R,D} = gen_{D,D} \cup (G - kill_{D,D}) = 001 + (000 - 000) = 001$
  $kill_{R,D} = kill_{D,D} \cup (K - gen_{D,D}) = 000 + (010 - 001) = 010$



Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

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

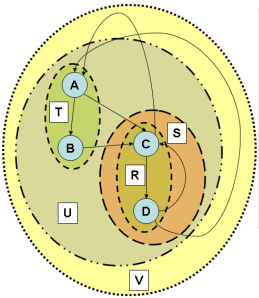
- Building region S from region R by T1 transformation
- The only predecessor of the header C, within S is D
- Therefore, $G = gen_{R,D} = 001$
- $kill_{S,C} = kill_{R,C} = 010$; $kill_{S,D} = kill_{R,D} = 010$
- $gen_{S,C} = gen_{R,C} \cup (G - kill_{R,C}) = 000 + (001 - 010) = 001$
  $gen_{S,D} = gen_{R,D} \cup (G - kill_{R,D}) = 001 + (001 - 010) = 001$



Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

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

- Building region T from regions A and B by T2 transf.
- $gen_{T,A} = gen_{A,A} = 100$; $kill_{T,A} = kill_{A,A} = 010$
- Header of B is B and pred. of B in A is A
- $G = gen_{A,A} = 100$ and $K = kill_{A,A} = 010$
- $gen_{T,B} = gen_{B,B} \cup (G - kill_{B,B}) = 010 + (100 - 101) = 010$
  $kill_{T,B} = kill_{B,B} \cup (K - gen_{B,B}) = 101 + (010 - 010) = 101$

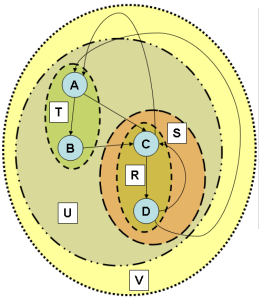


Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

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

- Building region U from regions T and S by T2 transf.
- $gen_{U,A} = gen_{T,A} = 100$; $kill_{U,A} = kill_{T,A} = 010$
- $gen_{U,B} = gen_{T,B} = 010$; $kill_{U,B} = kill_{T,B} = 101$

Region Based RD Analysis - An Example (6)

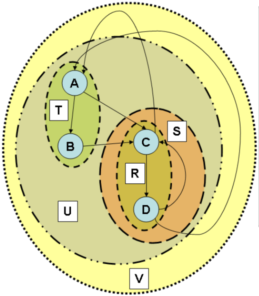


Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

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

- Building region U from regions T and S by T2 transf.
- Header of S is C and pred. of C in T are A and B
- $G = gen_{T,A} \cup gen_{T,B} = 110$ and
  $K = kill_{T,A} \cap kill_{T,B} = 000$
- $gen_{U,C} = gen_{S,C} \cup (G - kill_{S,C}) = 001 + (110 - 010) = 101$
  $kill_{U,C} = kill_{S,C} \cup (K - gen_{S,C}) = 010 + (000 - 001) = 010$
  $gen_{U,D} = gen_{S,D} \cup (G - kill_{S,D}) = 001 + (110 - 010) = 101$
  $kill_{U,D} = kill_{S,D} \cup (K - gen_{S,D}) = 010 + (000 - 001) = 010$

Region Based RD Analysis - An Example (7)

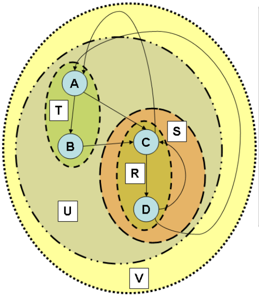


Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

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

- Building region V from region V by T1 transf.
- Header of U is A and pred. of A in U are C and D
- $G = gen_{U,C} \cup gen_{U,D} = 101$
- $gen_{V,C} = gen_{U,C} \cup (G - kill_{U,C}) = 101 + (101 - 010) = 101$
  $gen_{V,D} = gen_{U,D} \cup (G - kill_{U,D}) = 101 + (101 - 010) = 101$
  $kill_{V,C} = kill_{U,C} = 010; kill_{V,D} = kill_{U,D} = 010$

Region Based RD Analysis - An Example (8)



Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

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

- Building region V from region V by T1 transf.
- Header of U is A and pred. of A in U are C and D
- $G = gen_{U,C} \cup gen_{U,D} = 101$
- $gen_{V,A} = gen_{U,A} \cup (G - kill_{U,A}) = 100 + (101 - 010) = 101$
  $gen_{V,B} = gen_{U,B} \cup (G - kill_{U,B}) = 010 + (101 - 101) = 010$
  $kill_{V,A} = kill_{U,A} = 010; kill_{V,B} = kill_{U,B} = 101$

Results from Iterative RD DFA for the same example

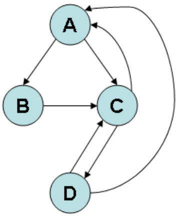
	gen	kill	OUT_1	IN_1	OUT_2	IN_2	OUT_3	IN_3	OUT_4	IN_4	RA
A	100	010	100	001	101	101	101	101	101	101	101
B	010	101	010	100	011	101	010	101	010	101	010
C	000	010	000	111	101	111	101	111	101	111	101
D	001	000	001	000	001	101	101	101	101	101	101

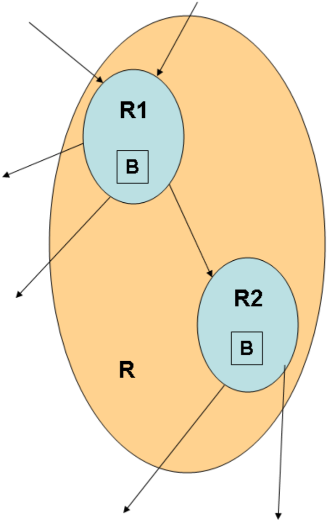
$$OUT[B] = gen[B] \bigcup (IN[B] - kill[B])$$

$$IN[B] = \bigcup_{P,\ \text{a predecessor of B}} OUT[P]$$

$$IN[B] = \varnothing \ (initialization)$$

Reaching Definitions Problem





For available expressions problem

Basic regions

$gen_{B,B} = gen[B]$

$kill_{B,B} = kill[B]$

Region building by T2

For basic blocks B within R1,

$gen_{R,B} = gen_{R1,B}$

$kill_{R,B} = kill_{R1,B}$

Edges from R2 to header of R1 are not part of R

For basic blocks B within R2,

$gen_{R,B} = gen_{R2,B}$ **U** $(G - kill_{R2,B})$

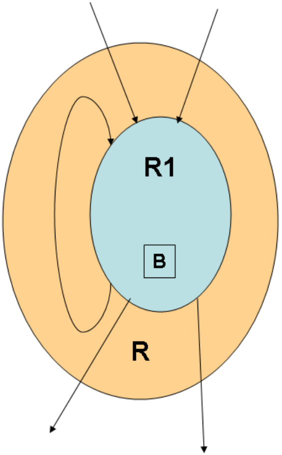
$kill_{R,B} = kill_{R2,B}$ **U** $(K - gen_{R2,B})$

where, $G = \bigcap gen_{R1,P}$, and

$K = \bigcup kill_{R1,P}$

for all predecessors P of the header of R2 in R1

Region Building by T1 Trans. - Available Exp



Region building by T1

$gen_{R,B} = gen_{R1,B}$
$kill_{R,B} = kill_{R1,B} \cup (K - gen_{R1,B})$

where, $K = \cup\ kill_{R1,P}$, for all predecessors P of the header of R1 in R

It is not necessary to compute $gen_{R,B}$ as in the previous case (T2).

An expression gets generated going from the header to B iff it is generated along all acyclic paths, and hence back edges incorporated into R will not cause more expressions to be generated

For available expressions problem

Results from Iterative AE DFA for the same example

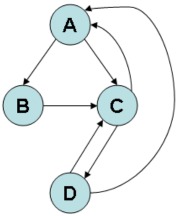
	gen	kill	OUT_1	IN_1	OUT_2	IN_2	OUT_3	IN_3	OUT_4	IN_4	RA
A	100	010	100	101	101	000	100	000	100	000	100
B	010	101	010	100	010	101	010	100	010	100	010
C	000	010	101	000	101	000	000	000	000	000	000
D	001	000	111	101	101	101	101	000	001	000	001

$$OUT[B] = gen[B] \bigcup (IN[B] - kill[B])$$

$$IN[B] = \bigcap_{P,\ a\ predecessor\ of\ B} OUT[P]$$

$$IN[B] = U \ \ (initialization)$$

Available Expressions Problem



Handling Irreducible Flow-Graphs

- At some point of reduction in $T_1 - T_2$ analysis, no further reduction is possible if the graph is irreducible
- At this point, we split nodes (regions are now nodes) and duplicate them as explained earlier
- We then continue our analysis
- If we wish to retain the original graph with no splitting, then after analyzing the split graph, we compute $IN[B] = IN[B_1] \wedge IN[B_2] \wedge ... \wedge IN[B_k]$, where, $B_i,\ 1 \leq i \leq k$ are the siblings of the split node B
- Splitting regions may be some times beneficial to optimizations since data-flow information may become more precise after splitting
    - For example, fewer definitions may reach each of the duplicated blocks than that reach the original block