6.035

Fall 2000

Lectures 11 & 12: Program Analysis

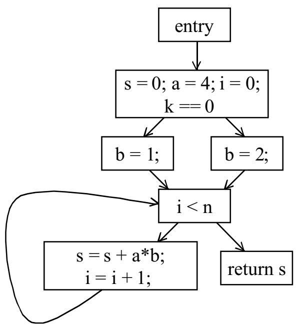
Analysis and Transformations

Analysis and Transformations

- Program Analysis
  - Discover properties of program
- Transformations
  - Use analysis results to transform program
  - Goal: improve some aspect of program
    - number of executed instructions, number of cycles
    - cache hit rate
    - memory space (code or data)
    - power

Control Flow Graph

```
into add(n, k) {
   s = 0; a = 4; i = 0;
   if (k == 0) b = 1;
   else b = 2;
   while (i < n) {
        s = s + a*b;
        i = i + 1;
   }
   return s;
}
```



Control Flow Graph

- Nodes Represent Computation
  - Each Node is a Basic Block
  - Basic Block is a Sequence of Instructions with
    - No Branches Out Of Middle of Basic Block
    - No Branches Into Middle of Basic Block
    - Basic Blocks should be maximal
  - Execution of basic block starts with first instruction
  - Includes all instructions in basic block
- Edges Represent Control Flow

Two Kinds of Variables

- Temporaries Introduced By Compiler
  - Transfer values only within basic block
  - Introduced as part of instruction flattening
  - Introduced by optimizations/transformations
- Program Variables
  - Declared in original program
  - May transfer values between basic blocks

Basic Block Optimizations

- Common Sub-Expression Elimination
  - a = (x+y)+z; b = x+y;
  - t = x+y; a = t+z; b = t;
- Constant Propagation
  - x = 5; b = x+y;
  - b = 5+y;
- Algebraic Identities
  - a = x * 1;
  - a = x;
- Copy Propagation
  - a = x+y; b = a; c = b+z;
  - a = x+y; b = a; c = a+z;
- Dead Code Elimination
  - a = x+y; b = a; c = a+z;
  - a = x+y; c = a+z
- Strength Reduction
  - t = i * 4;
  - t = i << 2;

Value Numbering

- Normalize basic block so that all statements are of the form
  - var = var op var (where op is a binary operator)
  - var = op var (where op is a unary operator)
  - var = var
- Simulate execution of basic block
  - Assign a virtual value to each variable
  - Assign a virtual value to each expression
  - Assign a temporary variable to hold value of each computed expression

Value Numbering for CSE

- As simulate execution of program
- Generate a new version of program
  - Each new value assigned to temporary
    - a = x+y; becomes a = x+y; t = a;
  - Temporary preserves value for use later in program even if original variable rewritten
    - a = x+y; a = a+z; b = x+y becomes
    - a = x+y; t = a; a = a+z; b = t;

CSE Example

• Original	• After CSE
a = x+y	a = x+y
b = a+z	b = a+z
b = b+y	t = b
c = a+z	b = b+y
	c = t

- Issues
  - Temporaries store values for use later
  - CSE with different names
    - a = x; b = x+y; c = a+y;
  - Excessive Temp Generation and Use

Original Basic Block

a = x+y
b = a+z
b = b+y
c = a+z

New Basic Block

a = x+y
t1 = a
b = a+z
t2 = b
b = b+y
t3 = b
c = t2

Var to Val

x→v1
y → v2
a → v3
z → v4
$\boxed{b \to v6}$
c → v5

Exp to Val

v1+v2 → v3
v3+v4 → v5
v5+v2 → v6

Exp to Tmp

v1+v2 → t1
v3+v4 → t2
v5+v2 → t3

Problems

- Algorithm has a temporary for each new value
  - a = x+y; t1 = a
- Introduces
  - lots of temporaries
  - lots of copy statements to temporaries
- In many cases, temporaries and copy statements are unnecessary
- So we eliminate them with copy propagation and dead code elimination

Copy Propagation

- Once again, simulate execution of program
- If can, use original variable instead of temporary
  - a = x+y; b = x+y;
  - After CSE becomes a = x+y; t = a; b = t;
  - After CP becomes a = x+y; b = a;
- Key idea: determine when original variables are NOT overwritten between computation of stored value and use of stored value

Copy Propagation Maps

- Maintain two maps
  - tmp to var: tells which variable to use instead of a given temporary variable
  - var to set: inverse of tmp to var. tells which temps are mapped to a given variable by tmp to var

Copy Propagation Example

- Original
  - a = x+y
  - b = a+z
  - c = x+y
  - a = b
- After CSE
  - a = x+y
  - t1 = a
  - b = a+z
  - t2 = b
  - c = t1
  - a = b
- After CSE and Copy Propagation
  - a = x+y
  - t1 = a
  - b = a+z
  - t2 = b
  - c = a
  - a = b

Copy Propagation Example

Basic Block After CSE	Basic Block After CSE and Copy Prop
a = x+y	a = x+y
t1 = a	t1 = a

tmp to var	var to set
t1Æa	aÆ{t1}

Copy Propagation Example

Basic Block After CSE	Basic Block After CSE and Copy Prop
a = x+y	a = x+y
t1 = a	t1 = a
b = a+z	b = a+z
t2 = b	t2 = b

tmp to var	var to set
t1Æa	aÆ{t1}
t2Æb	bÆ{t2}

Copy Propagation Example

Basic Block After CSE	Basic Block After CSE and Copy Prop
a = x+y	a = x+y
t1 = a	t1 = a
b = a+z	b = a+z
t2 = b	t2 = b
c = t1	

tmp to var	var to set
t1Æa	aÆ{t1}
t2Æb	bÆ{t2}

Copy Propagation Example

Basic Block After CSE	Basic Block After CSE and Copy Prop
a = x+y	a = x+y
t1 = a	t1 = a
b = a+z	b = a+z
t2 = b	t2 = b
c = t1	**c = a**

tmp to var	var to set
t1Æa	aÆ{t1}
t2Æb	bÆ{t2}

Copy Propagation Example

Basic Block After CSE	Basic Block After CSE and Copy Prop
a = x+y	a = x+y
t1 = a	t1 = a
b = a+z	b = a+z
t2 = b	t2 = b
c = t1	c = a
a = b	a = b

tmp to var	var to set
t1Æa	aÆ{t1}
t2Æb	bÆ{t2}

Copy Propagation Example

Basic Block After CSE	Basic Block After CSE and Copy Prop
a = x+y	a = x+y
t1 = a	t1 = a
b = a+z	b = a+z
t2 = b	t2 = b
c = t1	c = a
a = b	a = b

tmp to var	var to set
t1Æt1	aÆ{}
t2Æb	bÆ{t2}

Dead Code Elimination

- Copy propagation keeps all temps around
- May be temps that are never read
- Dead Code Elimination removes them

Basic Block After CSE and Copy Prop	Basic Block After CSE and Copy Prop
a = x+y	a = x+y
t1 = a	b = a+z
b = a+z	c = a
t2 = b	a = b
c = a	
a = b	

Dead Code Elimination

- Basic Idea
  - Process Code In Reverse Execution Order
  - Maintain a set of variables that are needed later in computation
  - If encounter an assignment to a temporary that is not needed, remove assignment

Basic Block After
CSE and Copy Prop

```
   a = x+y
   t1 = a
   b = a+z
   t2 = b
   c = a
⟹ a = b
```

Needed Set
{b}

Basic Block After
CSE and Copy Prop

```
   a = x+y
   t1 = a
   b = a+z
   t2 = b
⟹ c = a
   a = b
```

Needed Set
{a, b}

Basic Block After CSE and Copy Prop

```
   a = x+y
   t1 = a
   b = a+z
⟹ t2 = b
   c = a
   a = b
```

Needed Set
{a, b}

Basic Block After CSE and Copy Prop

```
   a = x+y
   t1 = a
   b = a+z
⟹
   c = a
   a = b
```

Needed Set
{a, b}

Basic Block After CSE and Copy Prop

```
   a = x+y
   t1 = a
⟹ b = a+z

   c = a
   a = b
```

Needed Set
{a, b, z}

Basic Block After CSE and Copy Prop

```
   a = x+y
⟹ t1 = a
   b = a+z

   c = a
   a = b
```

Needed Set
{a, b, z}

Basic Block After CSE and Copy Prop

```
   a = x+y
⟹
   b = a+z

   c = a
   a = b
```

Needed Set
{a, b, z}

Basic Block After , CSE Copy Propagation, and Dead Code Elimination

```
⟹ a = x+y

   b = a+z

   c = a
   a = b
```

Needed Set
{a, b, z}

Interesting Properties

- Analysis and Transformation Algorithms Simulate Execution of Program
  - CSE and Copy Propagation go forward
  - Dead Code Elimination goes backwards
- Transformations stacked
  - Group of basic transformations
  - Work together to get good result
  - Often, one transformation creates inefficient code that is cleaned up by following transformations

Other Basic Block Transformations

- Constant Propagation
- Strength Reduction
  - a << 2 = a * 4; a + a + a = 3 * a;
- Algebraic Simplification
  - a = a * 1; b = b + 0;
- Do these in unified transformation framework, not in earlier or later phases

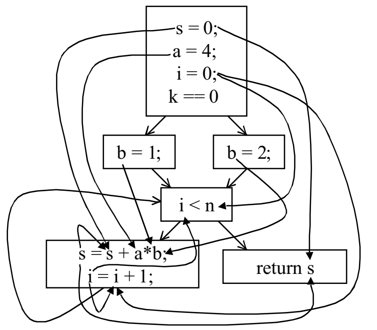
Dataflow Analysis

- Used to determine properties of program that involve multiple basic blocks
- Typically used to enable transformations
  - common sub-expression elimination
  - constant and copy propagation
  - dead code elimination
- Analysis and transformation often come in pairs

Reaching Definitions

- Concept of definition and use
  - a = x+y
  - is a definition of a
  - is a use of x and y
- A definition reaches a use if
  - value written by definition
  - may be read by use

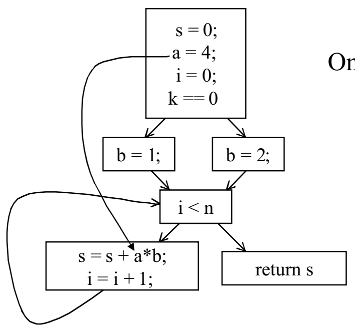
Reaching Definitions



Reaching Definitions and Constant Propagation

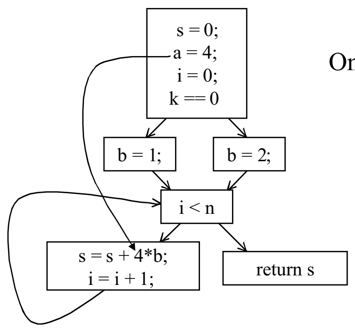
- Is a use of a variable a constant?
  - Check all reaching definitions
  - If all assign variable to same constant
  - Then use is in fact a constant
- Can replace variable with constant

Is a Constant in s = s+a*b?



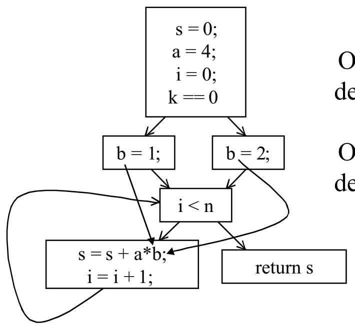
Yes!
On all reaching definitions
a = 4

Constant Propagation Transform



Yes!
On all reaching definitions
a = 4

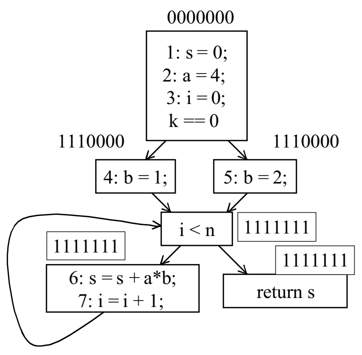
Is b Constant in s = s+a*b?



No!
One reaching definition with
b = 1
One reaching definition with
b = 2

Computing Reaching Definitions

- Compute with sets of definitions
  - represent sets using bit vectors
  - each definition has a position in bit vector
- At each basic block, compute
  - definitions that reach start of block
  - definitions that reach end of block
- Do computation by simulating execution of program until reach fixed point



Formalizing Analysis

- Each basic block has
  - IN - set of definitions that reach beginning of block
  - OUT - set of definitions that reach end of block
  - GEN - set of definitions generated in block
  - KILL - set of definitions killed in in block
- GEN[s = s + a*b; i = i + 1;] = 0000011
- KILL[s = s + a*b; i = i + 1;] = 1010000
- Compiler scans each basic block to derive GEN and KILL sets

Dataflow Equations

- IN[b] = OUT[b1] U ... U OUT[bn]
  - where b1, ..., bn are predecessors of b in CFG
- OUT[b] = (IN[b] - KILL[b]) U GEN[b]
- IN[entry] = 0000000
- Result: system of equations

Solving Equations

- Use fixed point algorithm
- Initialize with solution of OUT[b] = 0000000
- Repeatedly apply equations
  - IN[b] = OUT[b1] U ... U OUT[bn]
  - OUT[b] = (IN[b] - KILL[b]) U GEN[b]
- Until reach fixed point
- Until equation application has no further effect
- Use a worklist to track which equation applications may have a further effect

Reaching Definitions Algorithm

```
for all nodes n in N OUT[n] = emptyset; // OUT[n] = GEN[n];
Changed = D; // D = all definitions in graph
while (Changed != emptyset)
  choose a node n in Changed;
  Changed = Changed - { n };
  IN[n] = emptyset;
  for all nodes p in predecessors(n) IN[n] = IN[n] U OUT[p];
  OUT[n] = GEN[n] U (IN[n] - KILL[n]);
  if (OUT[n] changed)
    for all nodes s in successors(n) Changed = Changed U { s };
```

Questions

- Does the algorithm halt?
  - yes, because transfer function is monotonic
  - if increase IN, increase OUT
  - in limit, all bits are 1
- If bit is 1, is there always an execution in which corresponding definition reaches basic block?
- If bit is 0, does the corresponding definition ever reach basic block?
- Concept of conservative analysis

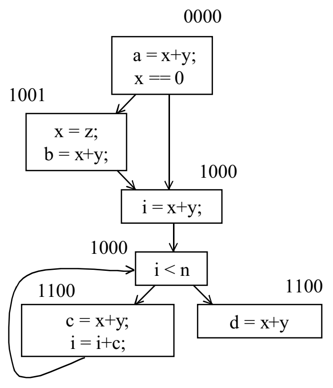
Available Expressions

- An expression x+y is available at a point p if
  - every path from the initial node to p evaluates x+y before reaching p,
  - and there are no assignments to x or y after the evaluation but before p.
- Available Expression information can be used to do global (across basic blocks) CSE
- If expression is available at use, no need to reevaluate it

Computing Available Expressions

- Represent sets of expressions using bit vectors
- Each expression corresponds to a bit
- Run dataflow algorithm similar to reaching definitions
- Big difference
  - definition reaches a basic block if it comes from ANY predecessor in CFG
  - expression is available at a basic block only if it is available from ALL predecessors in CFG

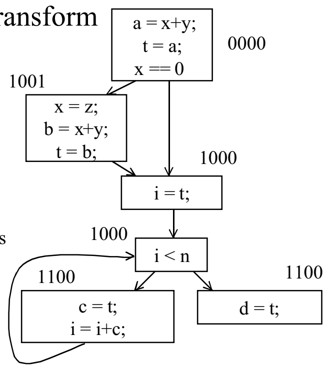
Expressions
1: x+y
2: i<n
3: i+c
4: x==0



Global CSE Transform

Expressions
1: x+y
2: i<n
3: i+c
4: x==0

must use same temp
for CSE in all blocks



Formalizing Analysis

- Each basic block has
  - IN - set of expressions available at start of block
  - OUT - set of expressions available at end of block
  - GEN - set of expressions computed in block
  - KILL - set of expressions killed in in block
- GEN[x = z; b = x+y] = 1000
- KILL[x = z; b = x+y] = 1001
- Compiler scans each basic block to derive GEN and KILL sets

Dataflow Equations

- IN[b] = OUT[b1] intersect ... intersect OUT[bn]
  - where b1, ..., bn are predecessors of b in CFG
- OUT[b] = (IN[b] - KILL[b]) U GEN[b]
- IN[entry] = 0000
- Result: system of equations

Solving Equations

- Use fixed point algorithm
- IN[entry] = 0000
- Initialize OUT[b] = 1111
- Repeatedly apply equations
  - IN[b] = OUT[b1] intersect ... intersect OUT[bn]
  - OUT[b] = (IN[b] - KILL[b]) U GEN[b]
- Use a worklist algorithm to reach fixed point

Available Expressions Algorithm

```
for all nodes n in N OUT[n] = E;  // OUT[n] = E - KILL[n];
 IN[Entry] = emptyset; OUT[Entry] = GEN[Entry];
Changed = N - { Entry }; // N = all nodes in graph
 while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };
    IN[n] = E; // E is set of all expressions
    for all nodes p in predecessors(n)
         IN[n] = IN[n] intersect OUT[p];
    OUT[n] = GEN[n] U (IN[n] - KILL[n]);
    if (OUT[n] changed)
      for all nodes s in successors(n) Changed = Changed U { s };
```

Questions

- Does algorithm always halt?
- If expression is available in some execution, is it always marked as available in analysis?
- If expression is not available in some execution, can it be marked as available in analysis?
- In what sense is algorithm conservative?

Duality In Two Algorithms

- Reaching definitions
  - Confluence operation is set union
  - OUT[b] initialized to empty set
- Available expressions
  - Confluence operation is set intersection
  - OUT[b] initialized to set of available expressions
- General framework for dataflow algorithms.
- Build parameterized dataflow analyzer once, use for all dataflow problems

Liveness Analysis

- A variable v is live at point p if
  - v is used along some path starting at p, and
  - no definition of v along the path before the use.
- When is a variable v dead at point p?
  - No use of v on any path from p to exit node, or
  - If all paths from p redefine v before using v.

What Use is Liveness Information?

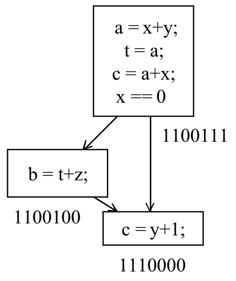
- Register allocation.
  - If a variable is dead, can reassign its register
- Dead code elimination.
  - Eliminate assignments to variables not read later.
  - But must not eliminate last assignment to variable (such as instance variable) visible outside CFG.
  - Can eliminate other dead assignments.
  - Handle by making all externally visible variables live on exit from CFG

Conceptual Idea of Analysis

- Simulate execution
- But start from exit and go backwards in CFG
- Compute liveness information from end to beginning of basic blocks

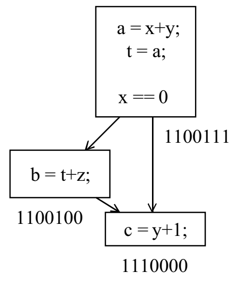
Liveness Example

- Assume a,b,c visible outside method
- So are live on exit
- Assume x,y,z,t not visible
- Represent Liveness Using Bit Vector
  - order is abcxyzt



Dead Code Elimination

- Assume a,b,c visible outside method
- So are live on exit
- Assume x,y,z,t not visible
- Represent Liveness Using Bit Vector
  - order is abcxyzt



Formalizing Analysis

- Each basic block has
  - IN - set of variables live at start of block
  - OUT - set of variables live at end of block
  - USE - set of variables with upwards exposed uses in block
  - DEF - set of variables defined in block
- USE[x = z; x = x+1;] = { z } (x not in USE)
- DEF[x = z; x = x+1;y = 1;] = {x, y}
- Compiler scans each basic block to derive GEN and KILL sets

Algorithm

```
out[Exit] = emptyset; in[Exit] = use[n];
for all nodes n in N - { Exit } in[n] = emptyset;
Changed = N - { Exit };
while (Changed != emptyset)
   choose a node n in Changed;
   Changed = Changed - { n };
   out[n] = emptyset;
   for all nodes s in successors(n) out[n] = out[n] U in[p];
   in[n] = use[n] U (out[n] - def[n]);
   if (in[n] changed)
     for all nodes p in predecessors(n)
       Changed = Changed U { p };
```

Similar to Other Dataflow Algorithms

- Backwards analysis, not forwards
- Still have transfer functions
- Still have confluence operators
- Can generalize framework to work for both forwards and backwards analyses

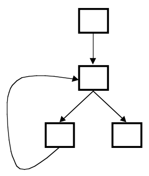
Analysis Information Inside Basic Blocks

- One detail:
  - Given dataflow information at IN and OUT of node
  - Also need to compute information at each statement of basic block
  - Simple propagation algorithm usually works fine
  - Can be viewed as restricted case of dataflow analysis

Loop Optimizations

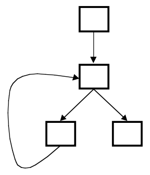
- Important because lots of computation occurs in loops
- We will study two optimizations
  - Loop-invariant code motion
  - Induction variable elimination

What is a Loop?



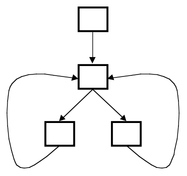
What is a Loop?

- Set of nodes
- Loop header
  - Single node
  - All iterations of loop go through header
- Back edge

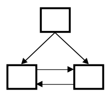


Anamalous Situations

- Two back edges, two loops, one header
- Compiler merges loops



- No loop header, no loop



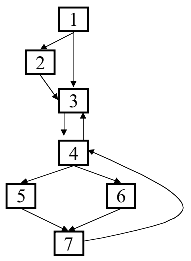
Defining Loops With Dominators

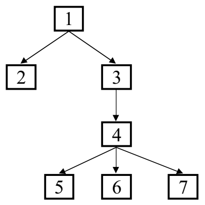
- Concept of dominator
  - Node n dominates a node m if all paths from start node to m go through n
  - "The road to the Super Bowl goes through St. Louis" Conclusion? St. Louis dominates the Super Bowl!
- If d_1 and d_2 both dominate n, then either
  - d_1 dominates d_2, or
  - d_2 dominates d_1 (but not both – look at path from start)
- Immediate dominator n – last dominator of n on any path from start node

Dominator Tree

- Nodes are nodes of control flow graph
- Edge from d to n if d immediate dominator of n
- This structure is a tree
- Rooted at start node

Example Dominator Tree

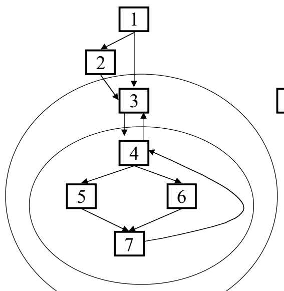


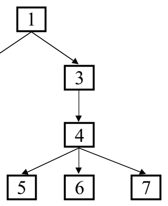


Defining Loops

- Unique entry point – header
- At least one path back to header
- Find edges whose heads dominate tails
  - These edges are back edges of loops
  - Given back edge d→n
  - Loop consists of d plus all nodes that can reach n without going through d
    (all nodes "between" d and n)
  - d is loop header

Two Loops In Example



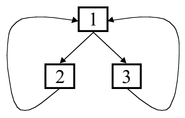


Loop Construction Algorithm

```
insert(m)
  if m ∉ loop then
     loop = loop ∪{M}
     push m onto stack
loop(d,n)
  loop = ∅; stack = ∅; insert(n);
  while stack not empty do
     m = pop stack;
     for all p ∈pred(m) do insert(p)
```

Nested Loops

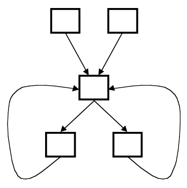
- If two loops do not have same header then
  - Either one loop (inner loop) contained in other (outer loop)
  - Or two loops are disjoint
- If two loops have same header, typically unioned and treated as one loop

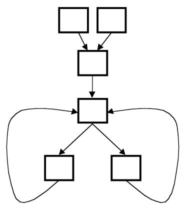


Two loops:
{1,2} and {1, 3}
Unioned: {1,2,3}

Loop Preheader

- Many optimizations stick code before loop
- Put a special node (loop preheader) before loop to hold this code

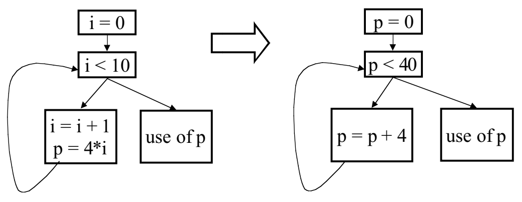




Loop Optimizations

- Now that we have the loop, can optimize it!
- Loop invariant code motion
  - Stick loop invariant code in the header

Induction Variable Elimination



Detecting Loop Invariant Code

- A statement is invariant if operands are
  - Constant,
  - Have all reaching definitions outside loop, or
  - Have exactly one reaching definition, and that definition comes from an invariant statement
- Concept of exit node of loop
  - node with successors outside loop

Loop Invariant Code Detection Algorithm

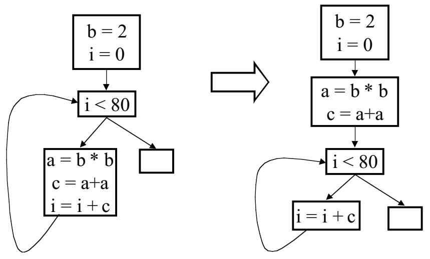
```
for all statements in loop
    if operands are constant or have all reaching definitions
    outside loop, mark statement as invariant
do
    for all statements in loop not already marked invariant
        if operands are constant, have all reaching
        definitions outside loop, or have exactly one reaching
        definition from invariant statement then
            mark statement as invariant
until find no more invariant statements
```

Loop Invariant Code Motion

- Conditions for moving a statement s:x:=y+z into loop header:
  - s dominates all exit nodes of loop
    - If it doesn't, some use after loop might get wrong value
    - Alternate condition: definition of x from s reaches no use outside loop (but moving s may increase run time)
  - No other statement in loop assigns to x
    - If one does, assignments might get reordered
  - No use of x in loop is reached by definition other than s
    - If one is, movement may change value read by use

Order of Statements in Preheader

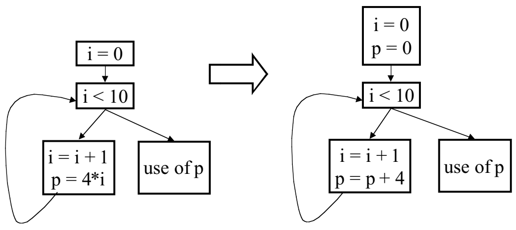
Preserve data dependences from original program
(can use order in which discovered by algorithm)



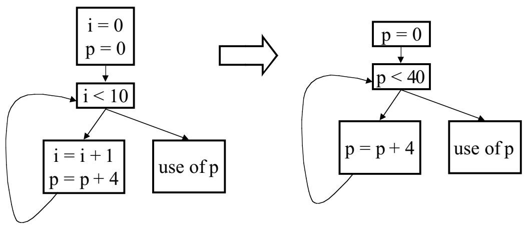
What is an Induction Variable?

- Base induction variable
  - Only assignments in loop are of form $i = i \pm c$
- Derived induction variables
  - Value is a linear function of a base induction variable
  - Within loop, $j = c*i + d$, where i is a base induction variable
  - Very common in array index expressions – an access to a[i] produces code like $p = a + 4*i$

Strength Reduction for Derived Induction Variables



Elimination of Superfluous Induction Variables



Three Algorithms

- Detection of induction variables
  - Find base induction variables
  - Each base induction variable has a family of derived induction variables, each of which is a linear function of base induction variable
- Strength reduction for derived induction variables
- Elimination of superfluous induction variables

Output of Induction Variable Detection Algorithm

- Set of induction variables
  - base induction variables
  - derived induction variables
- For each induction variable j, a triple <i,c,d>
  - i is a base induction variable
  - value of j is i*c+d
  - j belongs to family of i

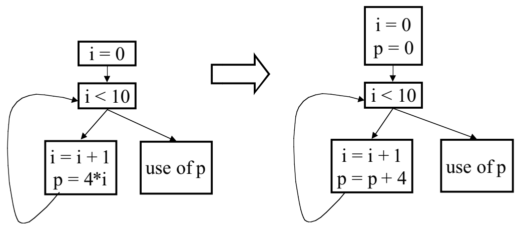
Induction Variable Detection Algorithm

```
Scan loop to find all base induction variables
do
   Scan loop to find all variables k with one assignment of
     form k = j*b where j is an induction variable with
     triple <i,c,d>
     make k an induction variable with triple <i,c*b,d>
   Scan loop to find all variables k with one assignment of
     form k = j±b where j is an induction variable with
     triple <i,c,d>
     make k an induction variable with triple <i,c,b±d>
until no more induction variables found
```

Strength Reduction Algorithm

```
for all derived induction variables j with triple
  <i,c,d>
  Create a new variable s
  Replace assignment j = i*c+d with j = s
  Immediately after each assignment i = i + e,
  insert statement s = s + c*e (c*e is constant)
  place s in family of i with triple <i,c,d>
  Insert s = c*i+d into preheader
```

Strength Reduction for Derived Induction Variables



Induction Variable Elimination

Choose a base induction variable i such that
  only uses of i are in
    termination condition of the form i < n
    assignment of the form i = i + m
Choose a derived induction variable k with <i,c,d>
  Replace termination condition with k < c*n+d

Induction Variable Wrapup

- There is lots more to induction variables
  - more general classes of induction variables
  - more general transformations involving induction variables

Summary

- Wide range of analyses and optimizations
- Dataflow Analyses and Corresponding Optimizations
  - reaching definitions, constant propagation
  - live variable analysis, dead code elimination
- Induction variable analyses and optimizations
  - Strength Reduction
  - Induction variable elimination
  - Important because of time spent in loops