Loop Optimizations

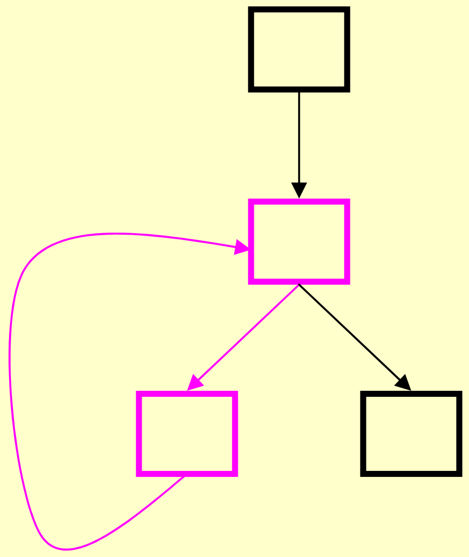
This lecture is primarily based on Konstantinos Sagonas set of slides
(**Advanced Compiler Techniques**, (2AD518)
at Uppsala University, January-February 2004).
Used with kind permission.

Loop Optimizations

- Important because lots of execution time occurs in loops
- First, we will identify loops
- We will study three optimizations
  - Loop-invariant code motion
  - Strength reduction
  - Induction variable elimination

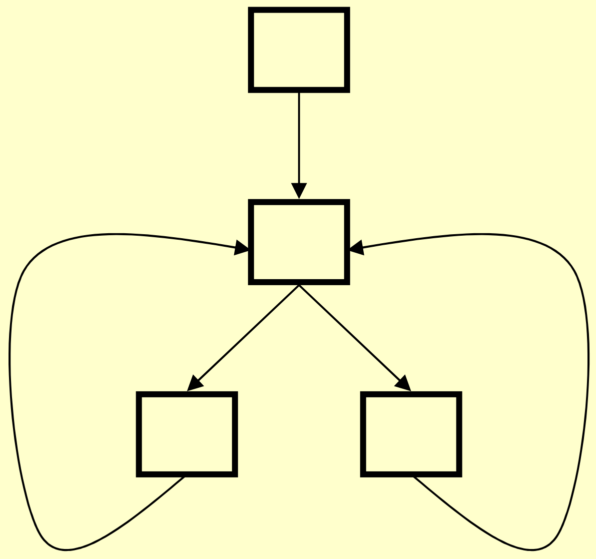
What is a Loop?

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

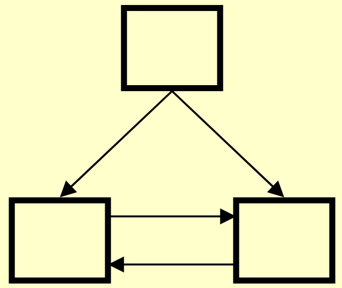


Anomalous Situations

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



- No loop header, no loop



Defining Loops With Dominators

Recall the concept of *dominators:*

- Node n dominates a node m if all paths from the start node to m go through n.
- The *immediate dominator* of m is the last dominator of m on any path from start node.
- A *dominator tree* is a tree rooted at the start node:
  - Nodes are nodes of control flow graph.
  - Edge from d to n if d is the immediate dominator of n.

Identifying Loops

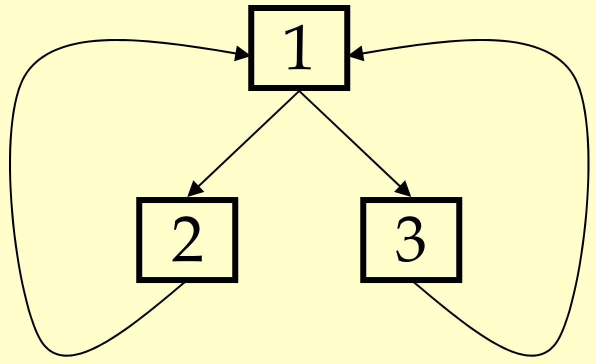
- A loop has a unique entry point – the header.
- At least one path back to header.
- Find edges whose heads (>) dominate tails (-), these edges are back edges of loops.
- Given a back edge $n \rightarrow d$:
  - The node d is the loop header.
  - The loop consists of n plus all nodes that can reach n without going through d (all nodes “between” d and n)

Loop Construction Algorithm

```
*loop*(d,n)
    loop = $\emptyset$; stack = $\emptyset$; *insert*(n);
    while stack not empty do
        m = pop stack;
        for all p $\in$ *pred*(m) do *insert*(p);
*insert*(m)
    if m $\notin$ loop then
        loop = loop $\cup$ {m};
        push m onto stack;
```

Nested Loops

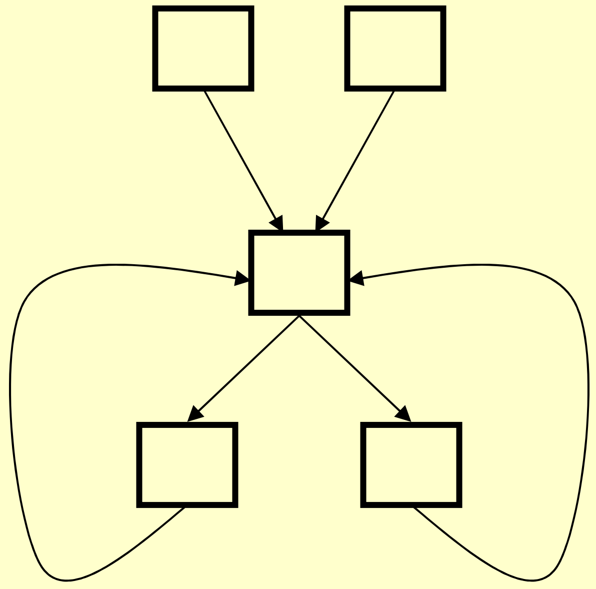
- If two loops do not have same header then
  - Either one loop (inner loop) is contained in the other (outer loop)
  - Or the two loops are disjoint
- If two loops have same header, typically they are unioned and treated as one loop

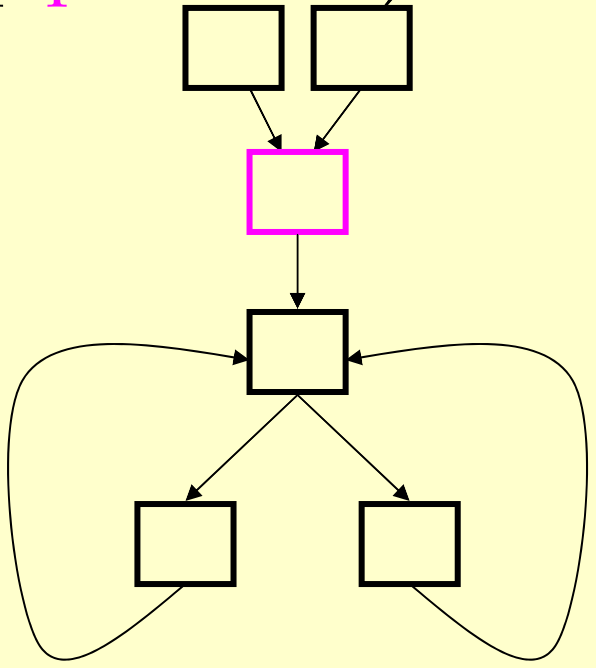


Two loops:
{1,2} and {1, 3}
Unioned: {1,2,3}

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, we can optimize it!
- Loop invariant code motion:
  - Move loop invariant code to the header.

Loop Invariant Code Motion

If a computation produces the same value in every loop iteration, move it out of the loop.

```
for i = 1 to N
  x = x + 1
  for j = 1 to N
    a[i,j] = 100*N + 10*i + j + x
```

$\Longrightarrow$

```
t1 = 100*N
for i = 1 to N
  x = x + 1
  t2 = t1 + 10*i + x
  for j = 1 to N
    a[i,j] = t2 + j
```

Detecting Loop Invariant Code

- A statement is *loop-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

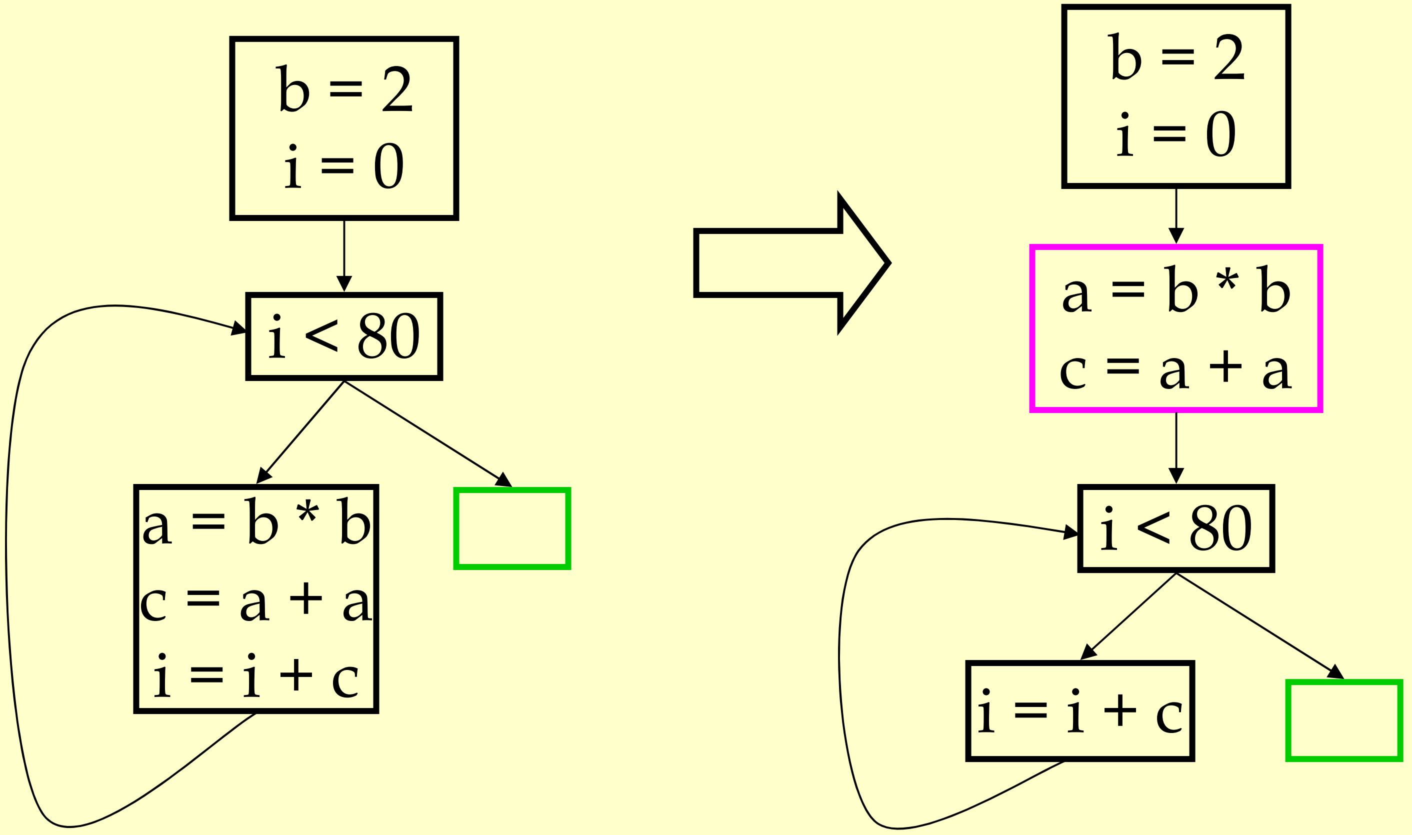
```
for all statements in loop
    if operands are constant or have all reaching definitions
    outside loop, mark statement as invariant
do
    for all statements in loop not already marked invariant
       if operands are constant, have all reaching definitions
    outside loop, or have exactly one reaching definition from
    invariant statement
       then  mark statement as invariant
until there are 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 does not, 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)



Induction Variables

Example:

```
for j = 1 to 100
    *(&A + 4*j) = 202 - 2*j
```

Basic Induction variable:
J = 1, 2, 3, 4, …..

Induction variable &A+4*j:
&A+4*j = &A+4, &A+8, &A+12, &A+16, ….

What are induction variables?

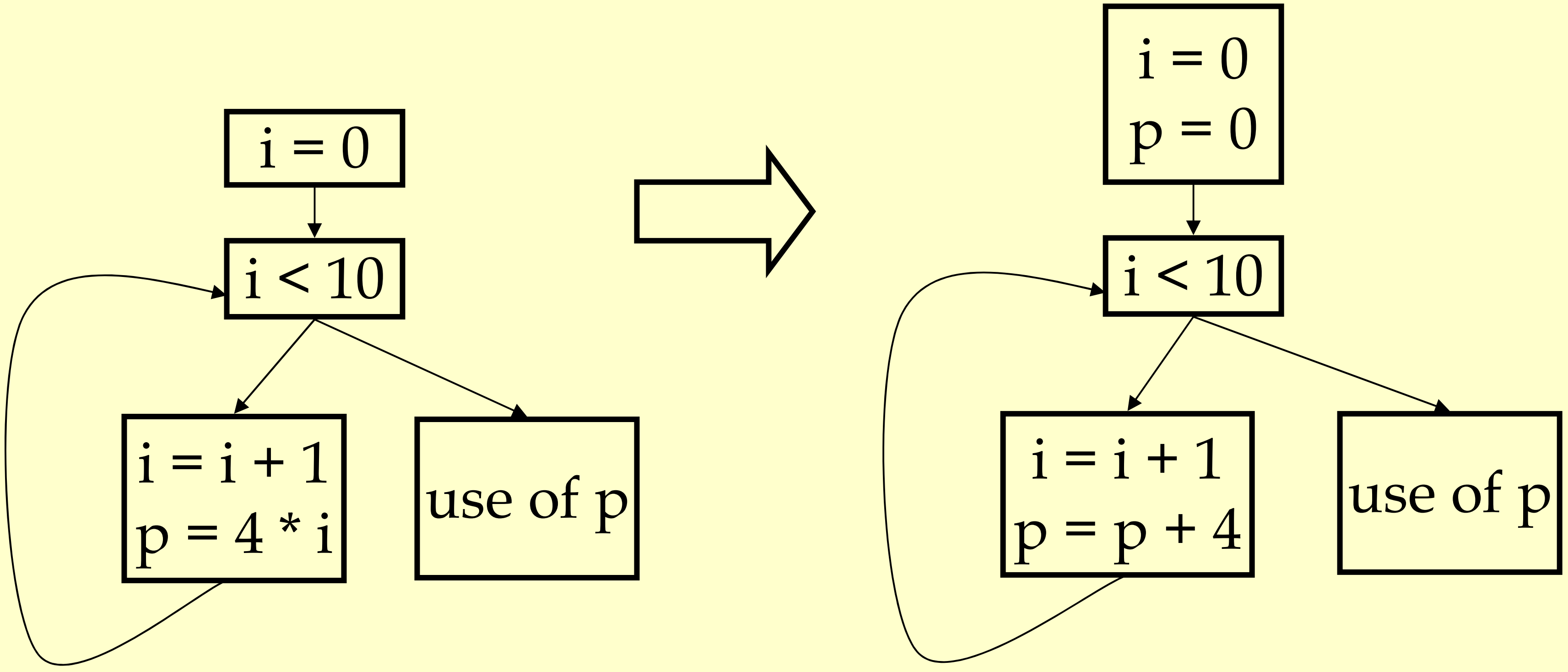
- x is an *induction variable* of a loop L if
  - variable changes its value every iteration of the loop
  - the value is a function of number of iterations of the loop
- In programs, this function is normally a linear function

  Example: for loop index variable j, function d + c*j

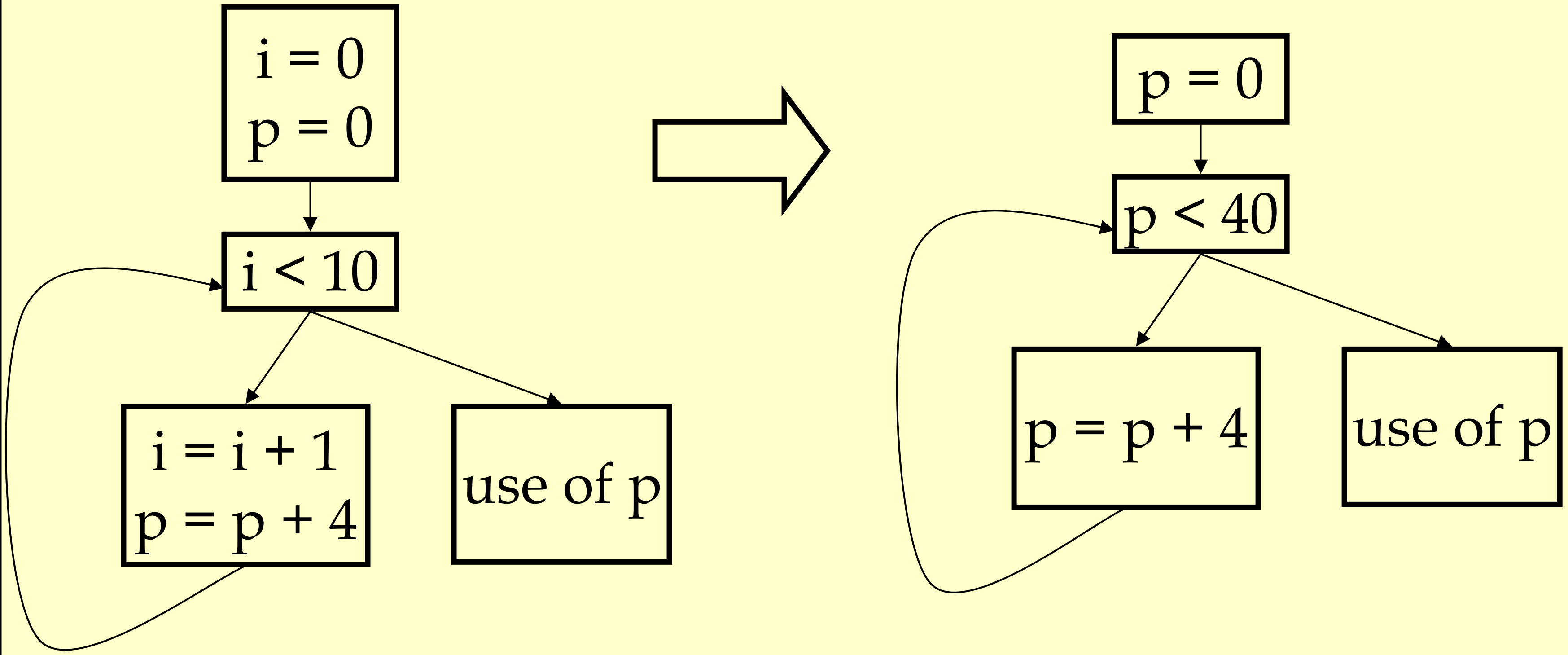
Types of Induction Variables

- Base induction variable:
  - Only assignments in loop are of form i = i ± 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.
  - the 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*b>
    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 are found
```

Strength Reduction

```
t = 202
for j = 1 to 100
  t = t - 2
  *(abase + 4*j) = t
```

Basic Induction variable:

$$\begin{tikzcd} J = 1, \arrow[r, bend right, "1"'] & 2, \arrow[r, bend right, "1"'] & 3, \arrow[r, bend right, "1"'] & 4, \ldots.. \end{tikzcd}$$

Induction variable 202 - 2*j

$$\begin{tikzcd} t = 202, \arrow[r, bend right, "-2"'] & 200, \arrow[r, bend right, "-2"'] & 198, \arrow[r, bend right, "-2"'] & 196, \ldots.. \end{tikzcd}$$

Induction variable abase+4*j:

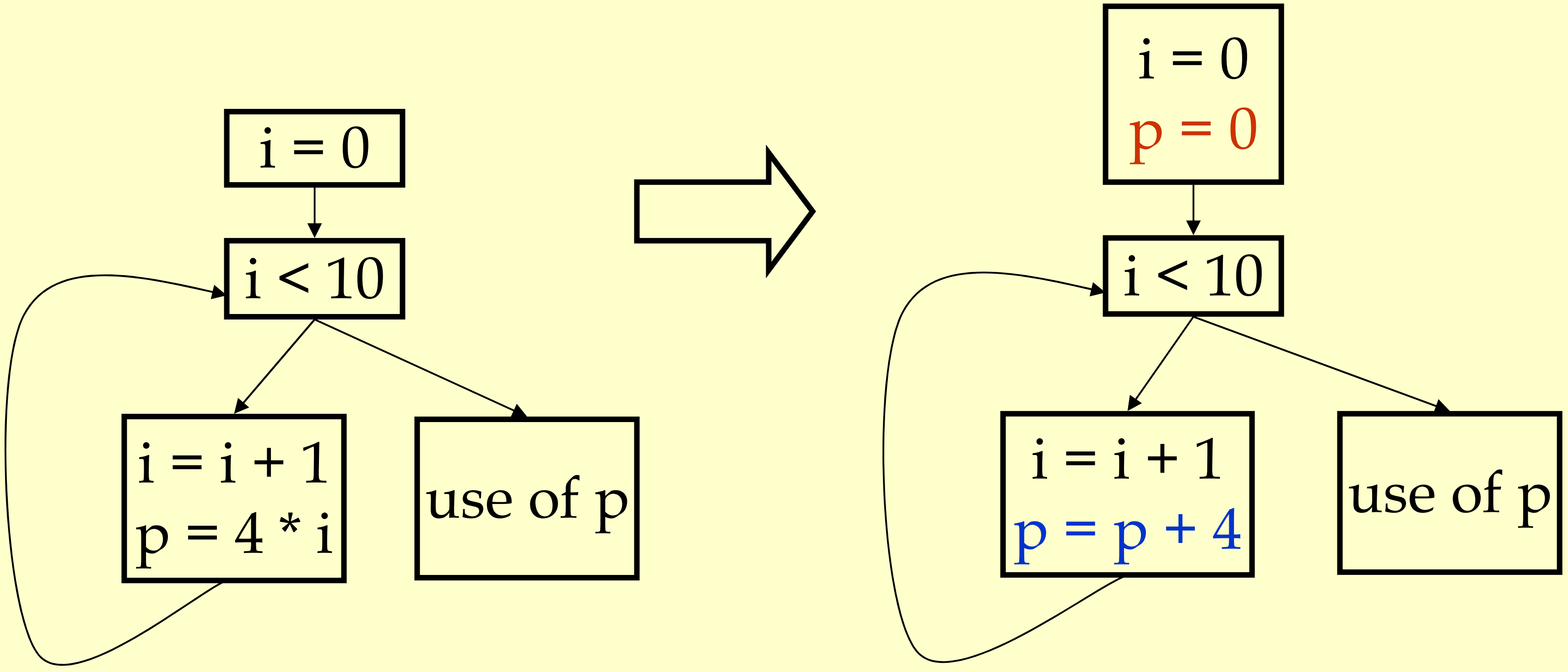
$$\begin{tikzcd} abase+4*j = abase+4, \arrow[r, bend right, "4"'] & abase+8, \arrow[r, bend right, "4"'] & abase+12, \arrow[r, bend right, "4"'] & abase+16, \ldots. \end{tikzcd}$$

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



Example

```
double A[256], B[256][256]
j = 1


while(j<100)
  A[j] = B[j][j]
  j = j + 2
```

```
double A[256], B[256][256]
j = 1
a = &A + 8
b = &B + 2056    // 2048+8
while(j<100)
  *a = *b
  j = j + 2
  a = a + 16
  b = b + 4112  // 4096+16
```

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
- $\langle i,c,d\rangle$
- Replace termination condition with $k < c*n+d$

Summary
Loop Optimization

- Important because lots of time is spent in loops.
- Detecting loops.
- Loop invariant code motion.
- Induction variable analyses and optimizations:
  - Strength reduction.
  - Induction variable elimination.