## Problem:

- The definition $x = e;$ $(x \notin Vars_e)$ may only be moved to an edge where $e$ is safe ;-)
- The definition must still be available for uses of $x$ ;-)

$$\Longrightarrow$$

We define an analysis which maximally delays computations:

$$
\begin{array}{lcl}
[\![;]\!]^\sharp\, D & = & D \\
[\![x = e;]\!]^\sharp\, D & = & \begin{cases} D \backslash (Use_e \cup Def_x) \cup \{x = e;\} & \text{if} \quad x \notin Vars_e \\ D \backslash (Use_e \cup Def_x) & \text{if} \quad x \in Vars_e \end{cases}
\end{array}
$$

... where:

$$
\begin{aligned}
Use_e &= \{y = e'; \mid y \in Vars_e\} \\
Def_x &= \{y = e'; \mid y \equiv x \vee x \in Vars_{e'}\}
\end{aligned}
$$

... where:

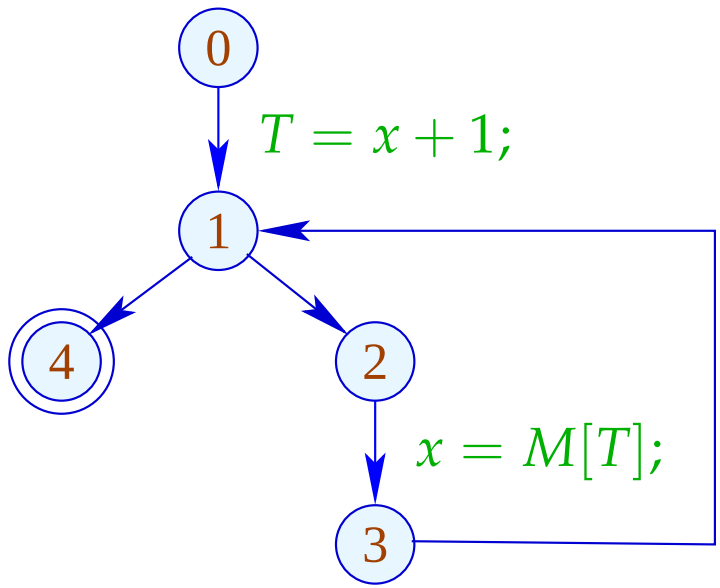
$$
\begin{aligned}
Use_e &= \{y = e'; \mid y \in Vars_e\} \\
Def_x &= \{y = e'; \mid y \equiv x \vee x \in Vars_{e'}\}
\end{aligned}
$$

For the remaining edges, we define:

$$
\begin{aligned}
[\![x = M[e];]\!]^\sharp\, D &= D \backslash (Use_e \cup Def_x) \\
[\![M[e_1] = e_2;]\!]^\sharp\, D &= D \backslash (Use_{e_1} \cup Use_{e_2}) \\
[\![\mathsf{Pos}(e)]\!]^\sharp\, D &= [\![\mathsf{Neg}(e)]\!]^\sharp\, D \quad = \quad D \backslash Use_e
\end{aligned}
$$

## Warning:

We may move $y = e;$ beyond a join only if $y = e;$ can be delayed along all joining edges:



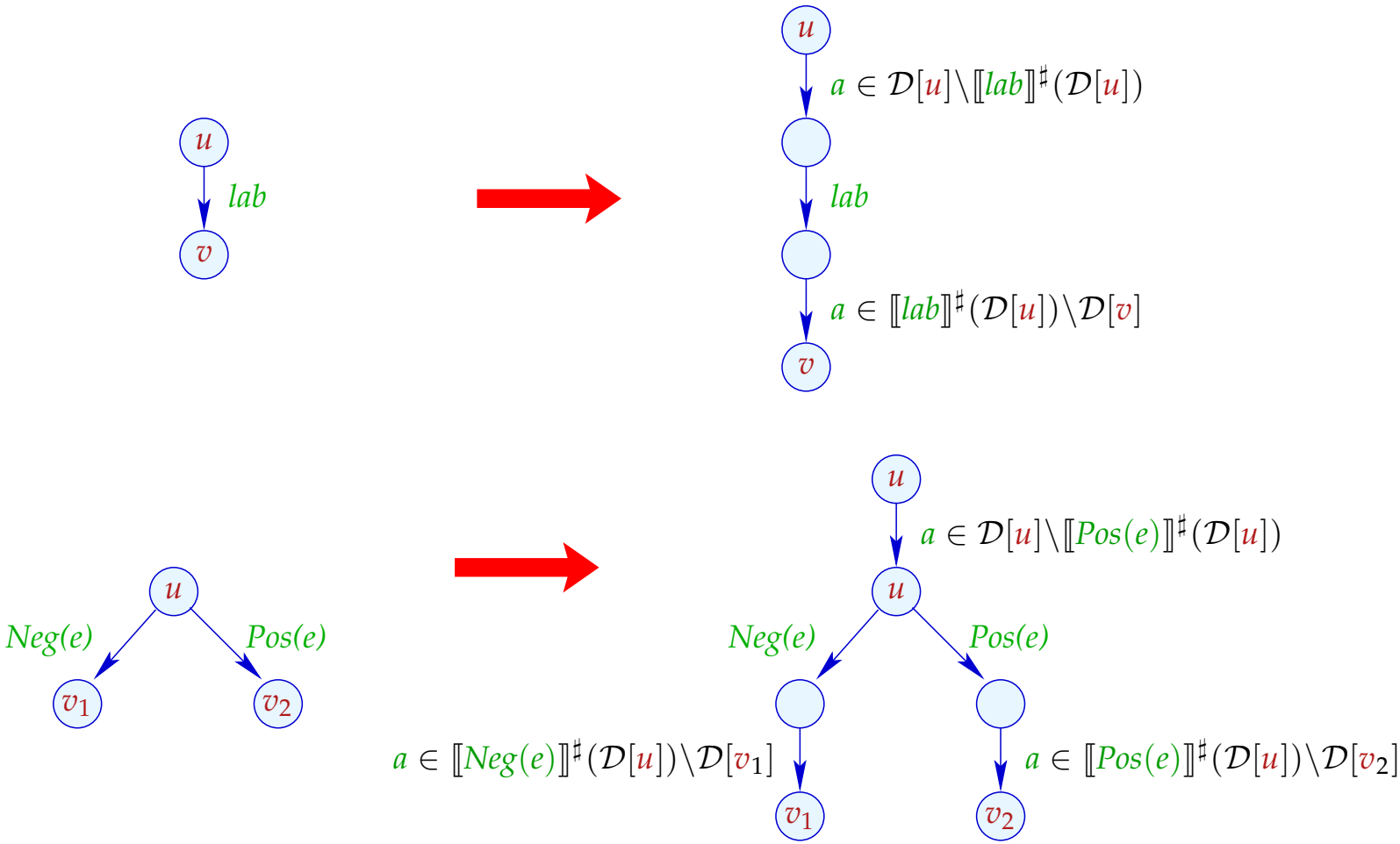

Here, $T = x + 1;$ cannot be moved beyond 1 !!!

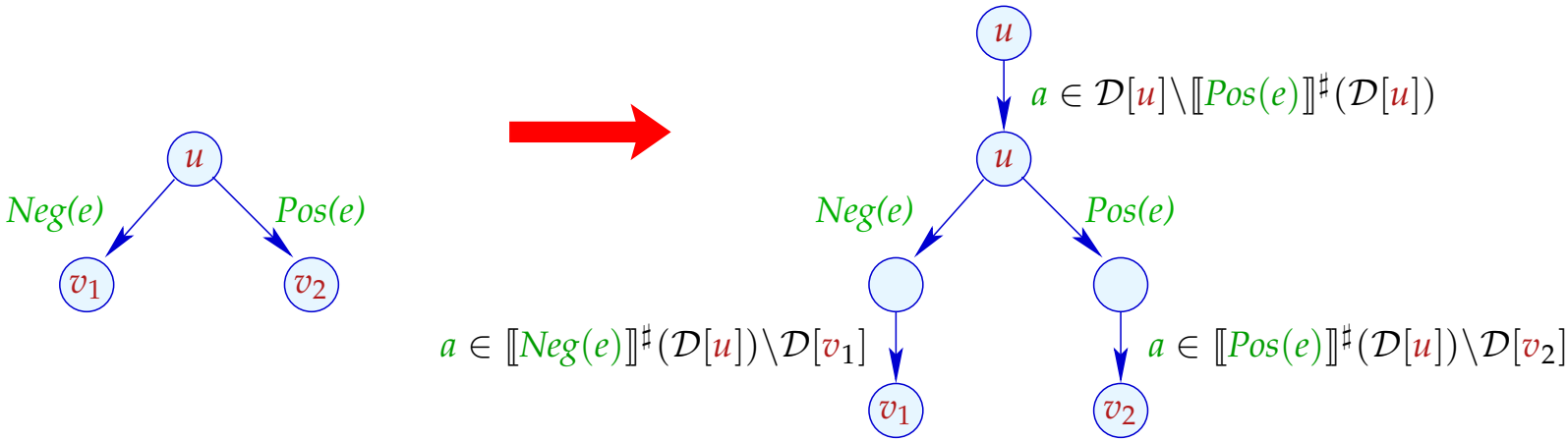
## We conclude:

- The partial ordering of the lattice for delayability is given by "$\supseteq$".
- At program start: $D_0 = \emptyset$.

  Therefore, the sets $\mathcal{D}[u]$ of at $u$ delayable assignments can be computed by solving a system of constraints.
- We delay only assignments $a$ where $a\ a$ has the same effect as $a$ alone.
- The extra insertions render the original assignments as assignments to dead variables ...

# Transformation 7:

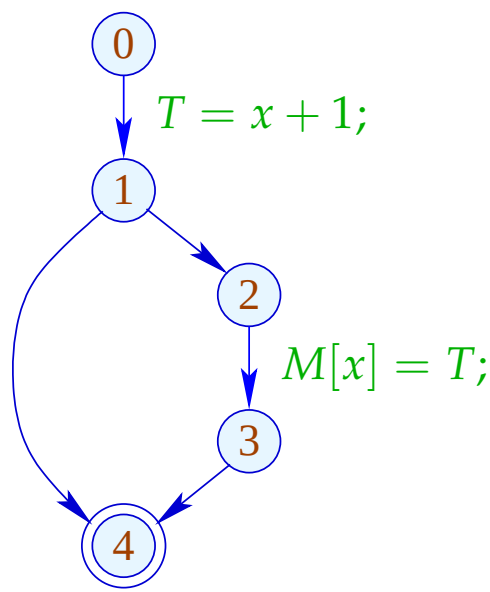




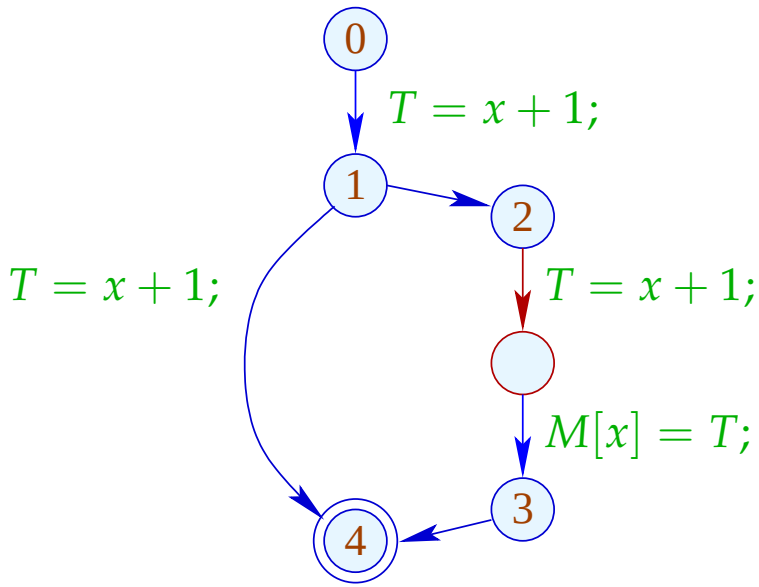

## Note:

Transformation T7 is only meaningful, if we subsequently eliminate assignments to dead variables by means of transformation T2 :-)

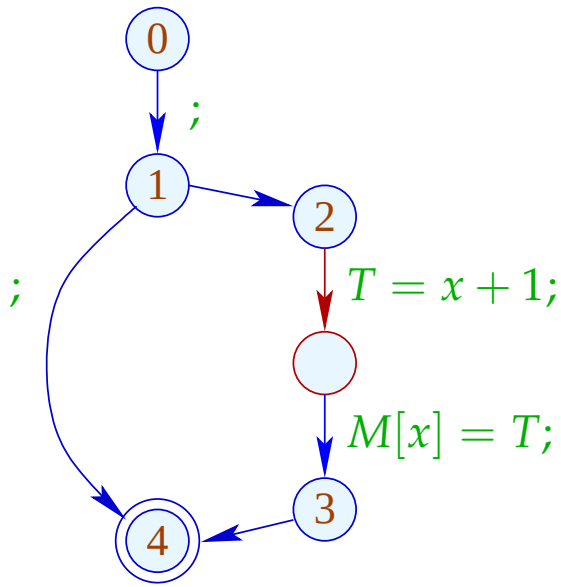
In the example, the partially dead code is eliminated:




| | $\mathcal{D}$ |
|---|---|
| 0 | $\emptyset$ |
| 1 | $\{T = x + 1;\}$ |
| 2 | $\{T = x + 1;\}$ |
| 3 | $\emptyset$ |
| 4 | $\emptyset$ |




| | $\mathcal{D}$ |
|---|---|
| 0 | $\emptyset$ |
| 1 | $\{T = x + 1;\}$ |
| 2 | $\{T = x + 1;\}$ |
| 3 | $\emptyset$ |
| 4 | $\emptyset$ |




| | $\mathcal{L}$ |
|---|---|
| 0 | $\{x\}$ |
| 1 | $\{x\}$ |
| 2 | $\{x\}$ |
| $2'$ | $\{x, T\}$ |
| 3 | $\emptyset$ |
| 4 | $\emptyset$ |

## Remarks:

- After $T7$ , all original assignments $y = e;$ with $y \notin Vars_e$ are assignments to dead variables and thus can always be eliminated :-)
- By this, it can be proven that the transformation is guaranteed to be non-degradating efficiency of the code :-))
- Similar to the elimination of partial redundancies, the transformation can be repeated :-}

## Conclusion:

- → The design of a meaningful optimization is non-trivial.
- → Many transformations are advantageous only in connection with other optimizations :-)
- → The ordering of applied optimizations matters !!
- → Some optimizations can be iterated !!!

... a meaningful ordering:

| | |
|---|---|
| T4 | Constant Propagation<br>Interval Analysis<br>Alias Analysis |
| T6 | Loop Rotation |
| T1, T3, T2 | Available Expressions |
| T2 | Dead Variables |
| T7, T2 | Partially Dead Code |
| T5, T3, T2 | Partially Redundant Code |

# 2 Replacing Expensive Operations by Cheaper Ones

## 2.1 Reduction of Strength

(1) Evaluation of Polynomials

$$f(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_1 \cdot x + a_0$$

| | Multiplications | Additions |
|---|---|---|
| naive | $\frac{1}{2}n(n+1)$ | $n$ |
| re-use | $2n-1$ | $n$ |
| Horner-Scheme | $n$ | $n$ |

Idea:

$$f(x) = (\ldots((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2}) \ldots) \cdot x + a_0$$

(2) Tabulation of a polynomial $f(x)$ of degree $n$ :

→ To recompute $f(x)$ for every argument $x$ is too expensive :-)

→ Luckily, the $n$-th differences are constant !!!

Example: $f(x) = 3x^3 - 5x^2 + 4x + 13$

| $n$ | $f(n)$ | $\Delta$ | $\Delta^2$ | $\Delta^3$ |
|---|---|---|---|---|
| 0 | 13 | 2 | 8 | $\boxed{18}$ |
| 1 | 15 | 10 | $\boxed{26}$ | |
| 2 | 25 | $\boxed{36}$ | | |
| 3 | $\boxed{61}$ | | | |
| 4 | ... | | | |

Here, the $n$-th difference is always

$$\Delta_h^n(f) = n! \cdot a_n \cdot h^n \qquad (h \text{ step width})$$

## Costs:

- $n$ times evaluation of $f$ ;
- $\frac{1}{2} \cdot (n-1) \cdot n$ subtractions to determine the $\Delta^k$ ;
- $2n-2$ multiplications for computing $\Delta_h^n(f)$;
- $n$ additions for every further value :-)

$$\Longrightarrow$$

Number of multiplications only depends on $n$ :-))

## Simple Case: $f(x) = a_1 \cdot x + a_0$

- ... naturally occurs in many numerical loops :-)
- The first differences are already constant:

$$f(x+h) - f(x) = a_1 \cdot h$$

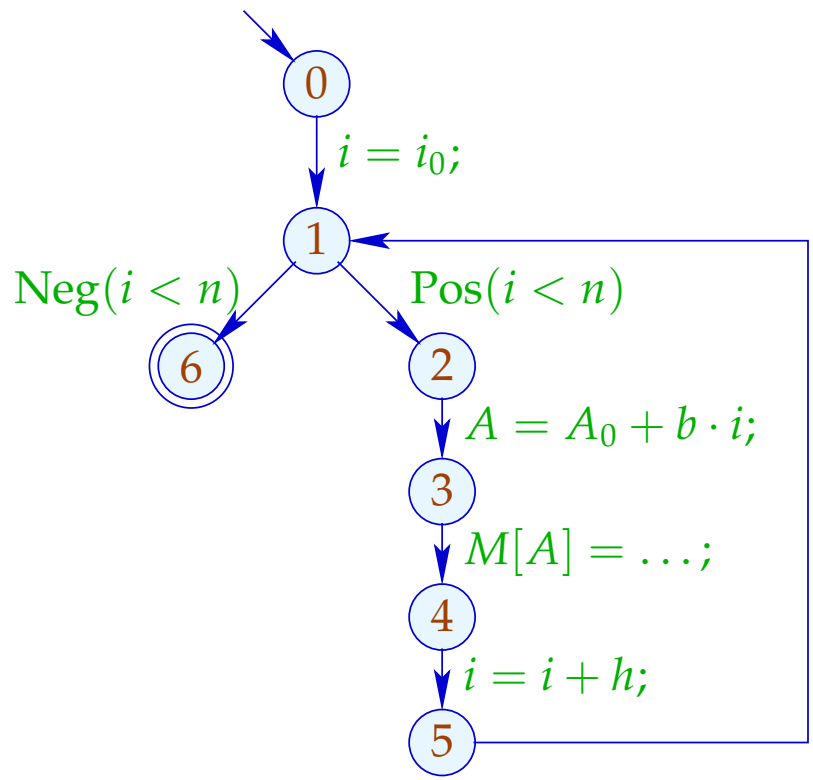
- Instead of the sequence: $y_i = f(x_0 + i \cdot h), \quad i \geq 0$

  we compute: $y_0 = f(x_0), \quad \Delta = a_1 \cdot h$

  $y_i = y_{i-1} + \Delta, \quad i > 0$

# Example:

$$\text{for } (i = i_0; i < n; i = i + h) \ \{$$
$$A = A_0 + b \cdot i;$$
$$M[A] = \ldots;$$
$$\}$$



## ... or, after loop rotation:

$i = i_0;$

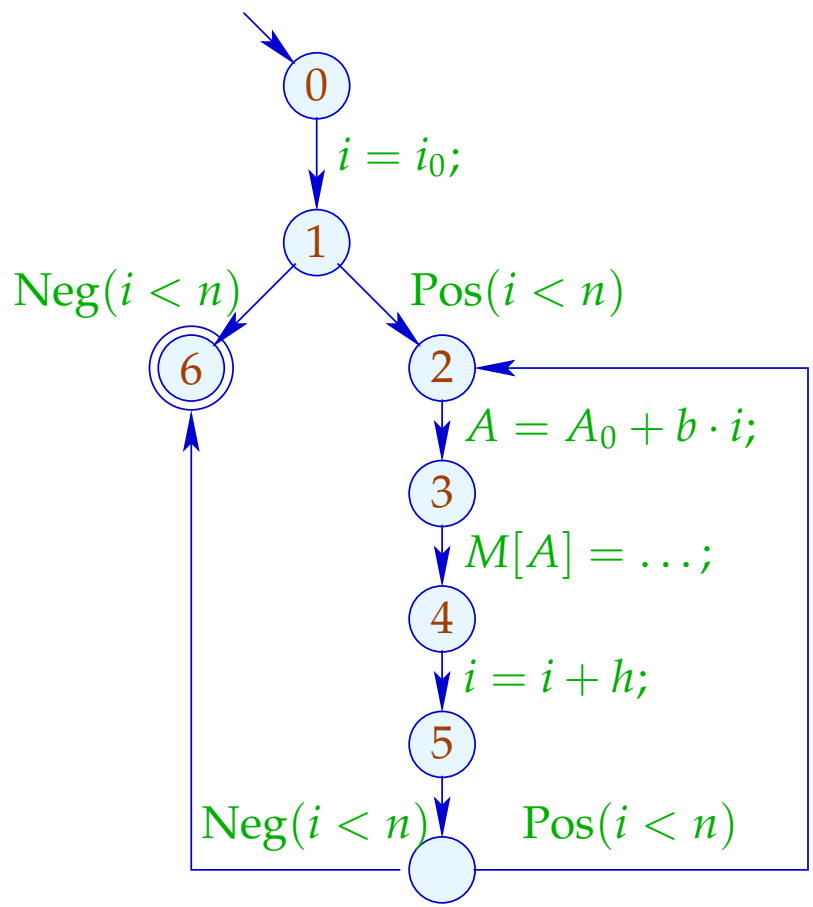
if $(i < n)$ do $\{$

$A = A_0 + b \cdot i;$

$M[A] = \ldots;$

$i = i + h;$

$\}$ while $(i < n);$



# ... and reduction of strength:

$i = i_0;$

if $(i < n)$ {

$\Delta = b \cdot h;$

$A = A_0 + b \cdot i_0;$

do {

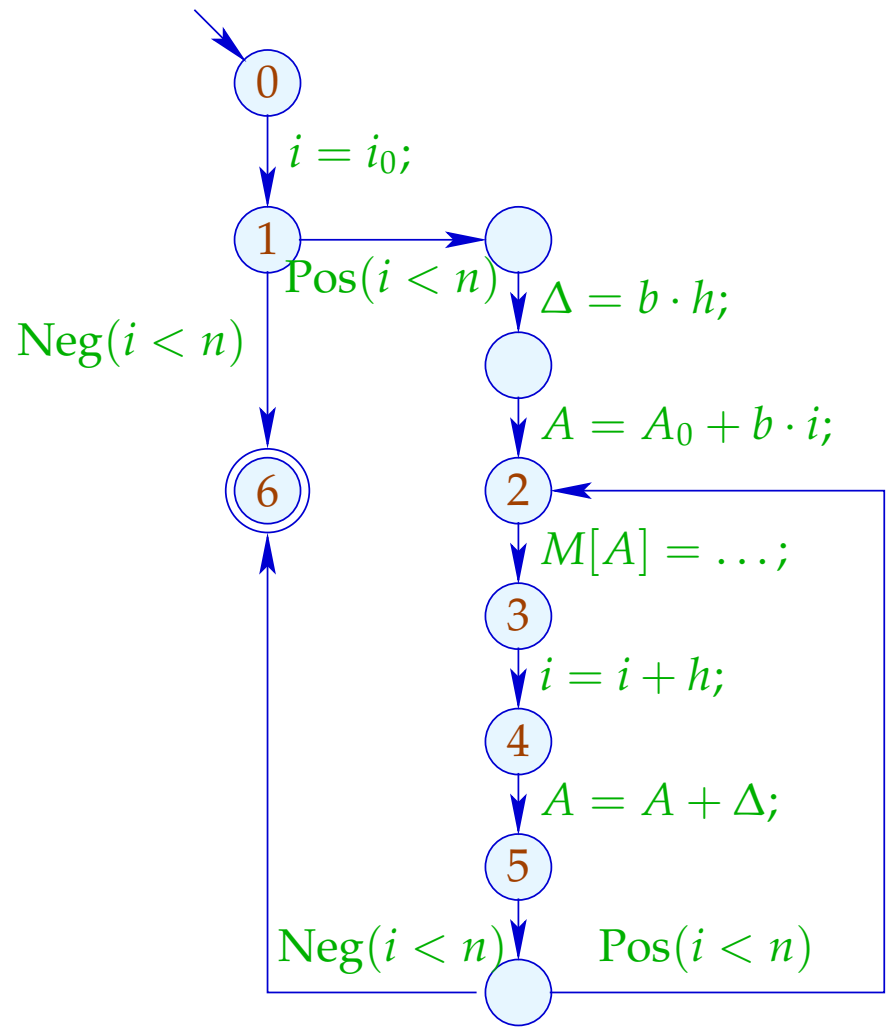
$M[A] = \ldots;$

$i = i + h;$

$A = A + \Delta;$

} while $(i < n);$

}



## Warning:

- The values $b, h, A_0$ must not change their values during the loop.
- $i, A$ may be modified at exactly one position in the loop :-(
- One may try to eliminate the variable $i$ altogether :

  $\rightarrow$ $i$ may not be used else-where.

  $\rightarrow$ The initialization must be transformed into: $A = A_0 + b \cdot i_0$ .

  $\rightarrow$ The loop condition $i < n$ must be transformed into: $A < N$ for $N = A_0 + b \cdot n$ .

  $\rightarrow$ $b$ must always be different from zero !!!

## Approach:

Identify

- ... loops;
- ... iteration variables;
- ... constants;
- ... the matching use structures.

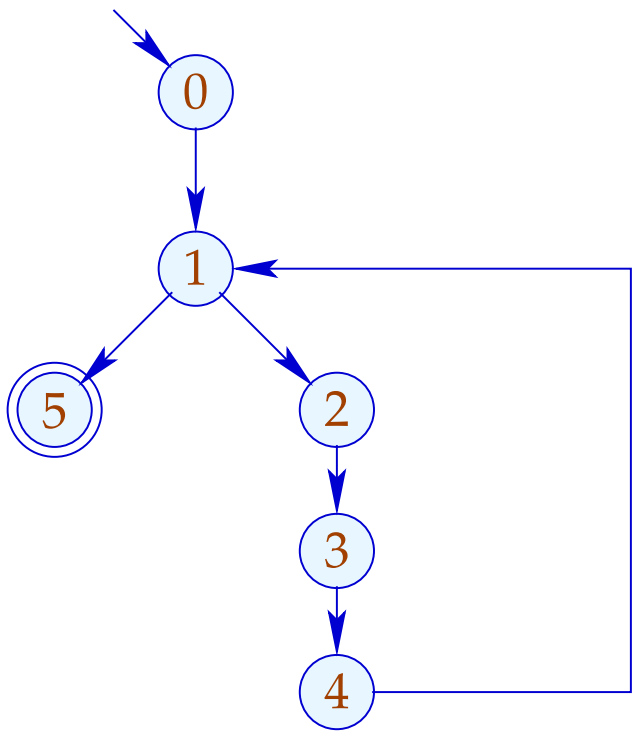
## Loops:

... are identified through the node $v$ with back edge $(_, _, v)$
:-)

For the sub-graph $G_v$ of the cfg on $\{w \mid v \Rightarrow w\}$, we define:

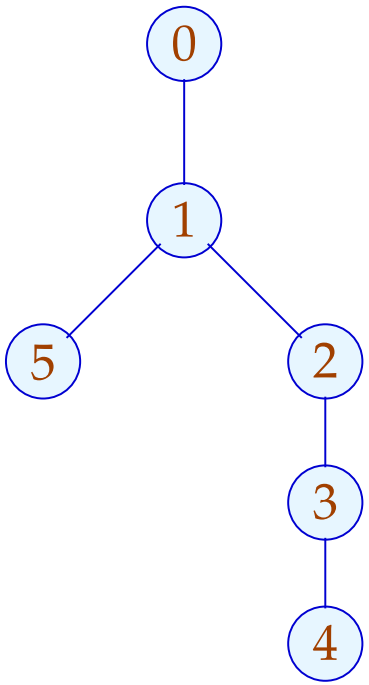
$$\mathsf{Loop}[v] \quad = \quad \{w \mid w \rightarrow^* v \ \text{ in } \ G_v\}$$

## Example:



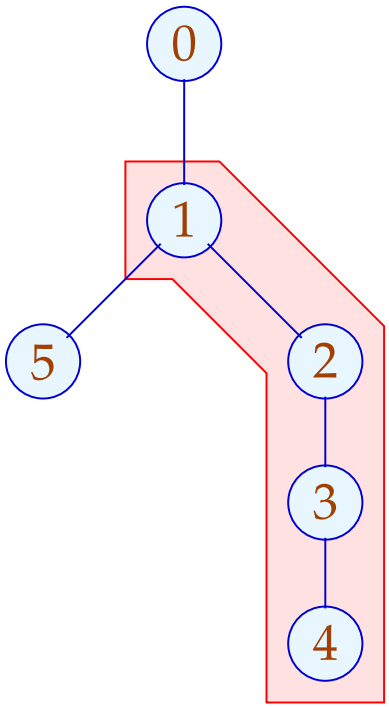

| | $\mathcal{P}$ |
|---|---|
| 0 | $\{0\}$ |
| 1 | $\{0,1\}$ |
| 2 | $\{0,1,2\}$ |
| 3 | $\{0,1,2,3\}$ |
| 4 | $\{0,1,2,3,4\}$ |
| 5 | $\{0,1,5\}$ |

## Example:



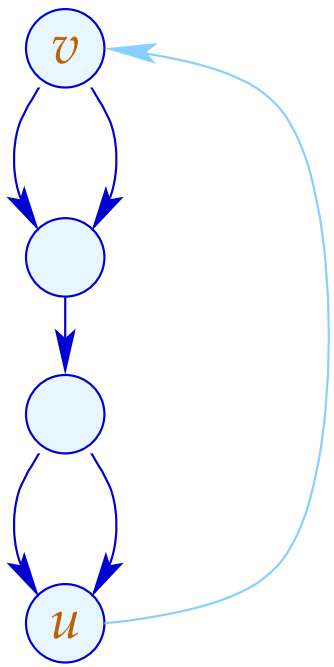

| | $\mathcal{P}$ |
|---|---|
| 0 | $\{0\}$ |
| 1 | $\{0,1\}$ |
| 2 | $\{0,1,2\}$ |
| 3 | $\{0,1,2,3\}$ |
| 4 | $\{0,1,2,3,4\}$ |
| 5 | $\{0,1,5\}$ |

## Example:




| | $\mathcal{P}$ |
|---|---|
| 0 | $\{0\}$ |
| 1 | $\{0,1\}$ |
| 2 | $\{0,1,2\}$ |
| 3 | $\{0,1,2,3\}$ |
| 4 | $\{0,1,2,3,4\}$ |
| 5 | $\{0,1,5\}$ |

We are interested in edges which during each iteration are executed exactly once:




Graph-theoretically, this is not easily expressible :-(

Edges $k$ could be selected such that:

- the sub-graph $G = \mathsf{Loop}[v] \backslash \{(_, _, v)\}$ is connected;
- the graph $G \backslash \{k\}$ is split into two unconnected sub-graphs.

Edges $k$ could be selected such that:

- the sub-graph $G = \mathsf{Loop}[v] \backslash \{(_, _, v)\}$ is connected;
- the graph $G \backslash \{k\}$ is split into two unconnected sub-graphs.

On the level of source programs, this is trivial:

$$\begin{array}{l} \mathsf{do}\ \{\ s_1 \ldots s_k \\ \quad \}\ \mathsf{while}\ (e); \end{array}$$

The desired assignments must be among the $s_i$ :-)

## Iteration Variable:

$i$ is an iteration variable if the only definition of $i$ inside the loop occurs at an edge which separates the body and is of the form:

$$i = i + h;$$

for some loop constant $h$ .

A loop constant is simply a constant (e.g., 42), or slightly more libaral, an expression which only depends on variables which are not modified during the loop :-)

## (3) Differences for Sets

Consider the fixpoint computation:

$$
\begin{array}{l}
x = \emptyset; \\
\text{for } \; (t = F\,x; t \not\subseteq x; \boxed{t = F\,x;}) \\
\qquad\qquad x = x \cup t;
\end{array}
$$

If $F$ is distributive, it could be replaced by:

$$
\begin{array}{l}
x = \emptyset; \\
\text{for } \; (\Delta = F\,x; \Delta \neq \emptyset; \boxed{\Delta = (F\,\Delta) \setminus x;}) \\
\qquad\qquad x = x \cup \Delta;
\end{array}
$$

The function $F$ must only be computed for the smaller sets $\Delta$

:-) semi-naive iteration

Instead of the sequence: $\emptyset \subseteq F(\emptyset) \subseteq F^2(\emptyset) \subseteq \ldots$

we compute: $\Delta_1 \cup \Delta_2 \cup \ldots$

where:

$$\begin{aligned} \Delta_{i+1} &= F(F^i(\emptyset)) \backslash F^i(\emptyset) \\ &= F(\Delta_i) \backslash (\Delta_1 \cup \ldots \cup \Delta_i) \quad \text{with } \Delta_0 = \emptyset \end{aligned}$$

Assume that the costs of $F\ x$ is $1 + \#x$ .

Then the costs may sum up to:

| | | | |
|---|---|---|---|
| naive | $1 + 2 + \ldots + n + n$ | $=$ | $\frac{1}{2}n(n+3)$ |
| semi-naive | | | $2n$ |

where $n$ is the cardinality of the result.

$\Longrightarrow$ A linear factor is saved :-)

## 2.2 Peephole Optimization

Idea:

- Slide a small window over the program.
- Optimize agressively inside the window, i.e.,

  → Eliminate redundancies!

  → Replace expensive operations inside the window by cheaper ones!

## Examples:

$x = x + 1; \quad \Longrightarrow \quad x{+}{+};$

// given that there is a specific increment instruction :-)

$z = y - a + a; \quad \Longrightarrow \quad z = y;$

// algebraic simplifications :-)

$x = x; \quad \Longrightarrow \quad ;$

$x = 0; \quad \Longrightarrow \quad x = x \oplus x;$

$x = 2 \cdot x; \quad \Longrightarrow \quad x = x + x;$