Script **generated by TTT**

Title: Seidl: Programmoptimierung (11.12.2013)

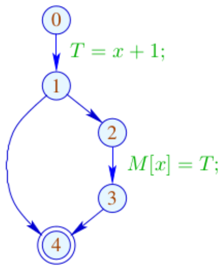
Date: Wed Dec 11 08:32:32 CET 2013

Duration: 87:21 min

Pages: 50

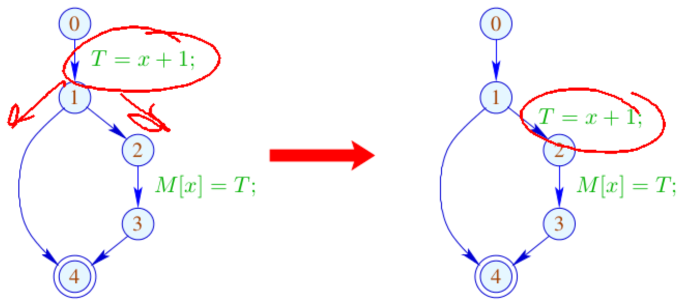
1.9 Eliminating Partially Dead Code

Example:



$x+1$ need only be computed along one path ;-(

Idea:



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:

$$[\![;]\!]^\sharp\, D = D$$

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

y + z

... where:

$$\begin{array}{lll} \mathit{Use}_e & = & \{y = e'; \mid y \in \mathit{Vars}_e\} \\ \mathit{Def}_x & = & \{y = e'; \mid y \equiv x \vee x \in \mathit{Vars}_{e'}\} \end{array}$$

Problem:

- The definition $x = e;$ $(x \notin \mathit{Vars}_e)$ 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:

$$\begin{array}{lll} [\![;]\!]^\sharp\, D & = & D \\ [\![x = e;]\!]^\sharp\, D & = & \begin{cases} D \backslash (\mathit{Use}_e \cup \mathit{Def}_x) \cup \{x = e;\} & \text{if} \quad x \notin \mathit{Vars}_e \\ D \backslash (\mathit{Use}_e \cup \mathit{Def}_x) & \text{if} \quad x \in \mathit{Vars}_e \end{cases} \end{array}$$

$\cup$

$y = x+1$

... where:

$$\begin{array}{lll} \mathit{Use}_e & = & \{y = e'; \mid y \in \mathit{Vars}_e\} \\ \mathit{Def}_x & = & \{y = e'; \mid y \equiv x \vee x \in \mathit{Vars}_{e'}\} \end{array}$$

... where:

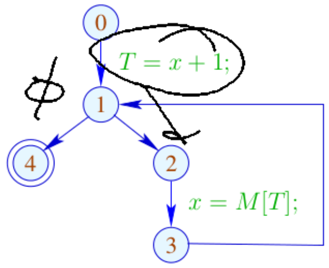
$$\begin{array}{lll} \mathit{Use}_e & = & \{y = e'; \mid y \in \mathit{Vars}_e\} \\ \mathit{Def}_x & = & \{y = e'; \mid y \equiv x \vee x \in \mathit{Vars}_{e'}\} \end{array}$$

For the remaining edges, we define:

$$\begin{array}{lll} [\![x = M[e];]\!]^\sharp\, D & = & D \backslash (\mathit{Use}_e \cup \mathit{Def}_x) \\ [\![M[e_1] = e_2;]\!]^\sharp\, D & = & D \backslash (\mathit{Use}_{e_1} \cup \mathit{Use}_{e_2}) \\ [\![\mathsf{Pos}(e)]\!]^\sharp\, D & = & [\![\mathsf{Neg}(e)]\!]^\sharp\, D \quad = \quad D \backslash \mathit{Use}_e \end{array}$$

Warning:

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

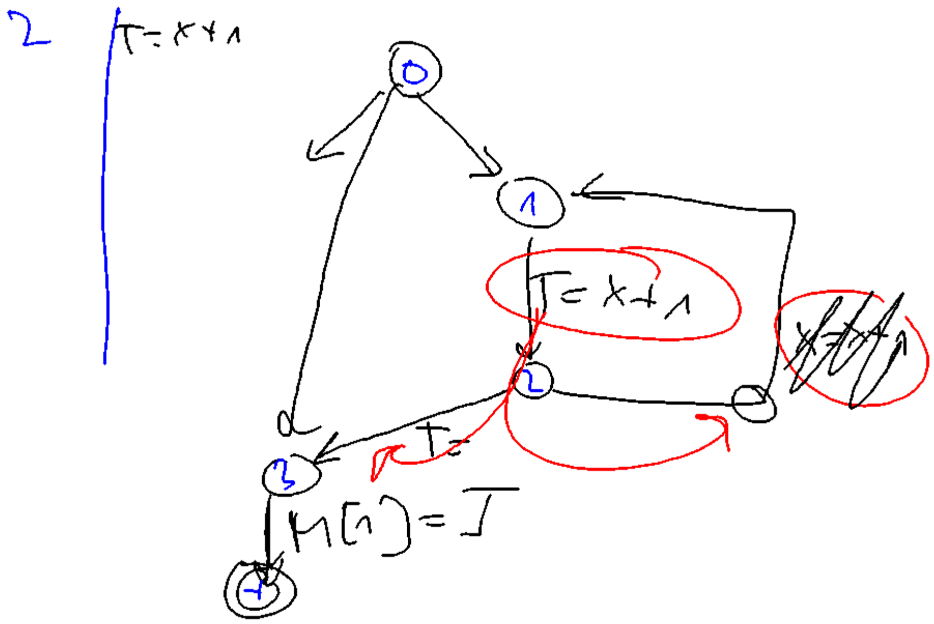


Here, $T = x + 1;$ cannot be moved beyond 1 !!!

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 ...



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

2 T=x+1

We conclude:

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

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

T=x+1

M[1]=T

... where:

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

For the remaining edges, we define:

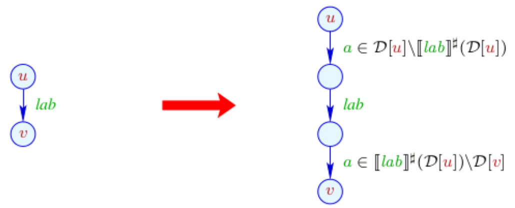
$$\begin{array}{lcl} [\![x = M[e];]\!]^\sharp\, D & = & D \backslash (Use_e \cup Def_x) \\ [\![M[e_1] = e_2;]\!]^\sharp\, D & = & D \backslash (Use_{e_1} \cup Use_{e_2}) \\ [\![\mathsf{Pos}(e)]\!]^\sharp\, D & = & [\![\mathsf{Neg}(e)]\!]^\sharp\, D \quad = \quad D \backslash Use_e \end{array}$$

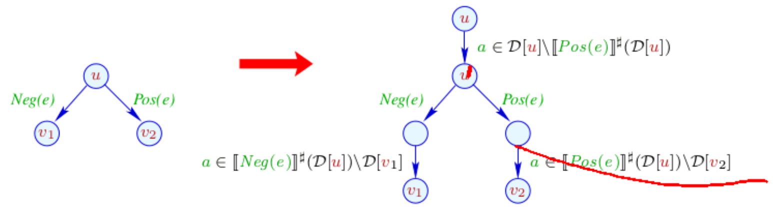
We conclude:

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

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

Transformation 7:



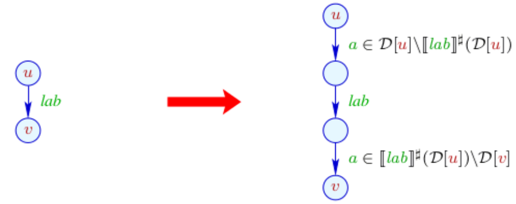


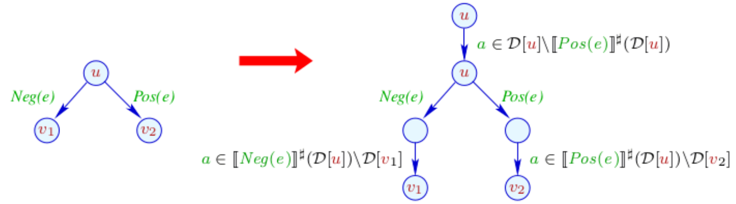
Note:

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:

Transformation 7:

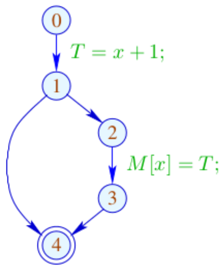




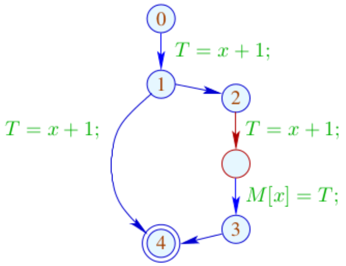
Note:

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

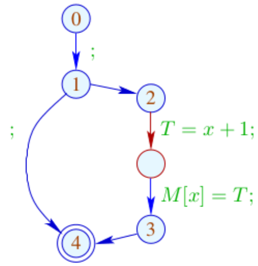
In the example, the partially dead code is eliminated:



	$\mathcal{D}$
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	$\emptyset$

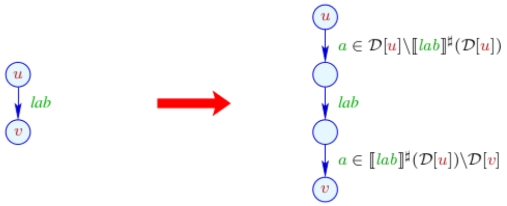


	$\mathcal{L}$
0	$\{x\}$
1	$\{x\}$
2	$\{x\}$
$2'$	$\{x, T\}$
3	$\emptyset$
4	$\emptyset$

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 :-}

Transformation 7:



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 Interval Analysis Alias Analysis
T6	Loop Rotation
T1, T3, T2	Available Expressions
T2	Dead Variables
T7, T2	Partially Dead Code
T5, T3, T2	Partially Redundant Code

... a meaningful ordering:

T4	Constant Propagation Interval Analysis 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

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

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

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 !!!

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 !!!

$a\,(x+1)^n - a x^n = a \sum_i \binom{n}{i} x^i - a x^n$

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

n	$f(n)$	Δ	Δ^2	Δ^3
0	13	2	8	18
1	15	10	26	18
2	25	36	44	
3	61	80		
4	141			

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 Δ^k ;
- n additions for every further value :-)

$\Longrightarrow$

Number of multiplications only depends on n :-))

8 A_0

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

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

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

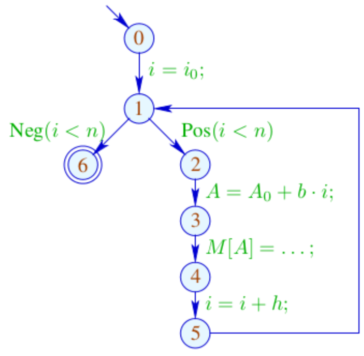
- Instead of the sequence: $y_i = f\,(x_0 + i \cdot h)\,, \quad i \geq 0$

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

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

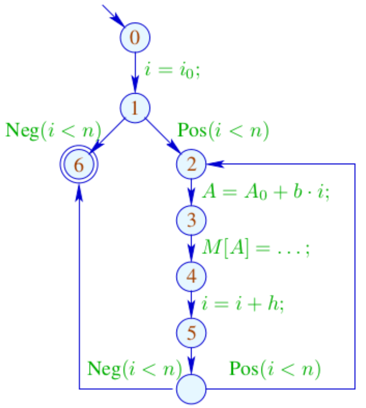
Example:

```
for (i = i0; i < n; i = i + h) {
        A = A0 + b · i;
        M[A] = ...;
}
```



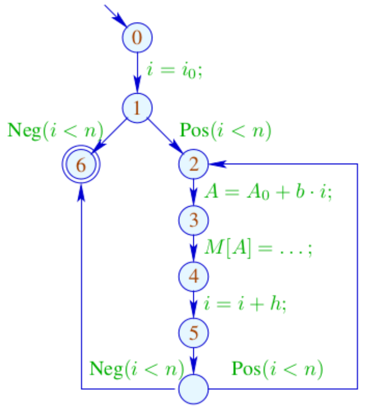
... or, after loop rotation:

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



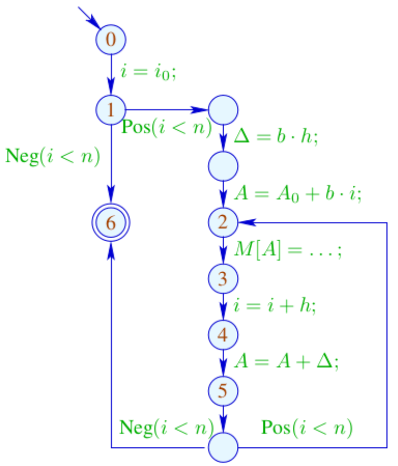
... or, after loop rotation:

$i = i_0;$
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 :

  → i may not be used else-where.

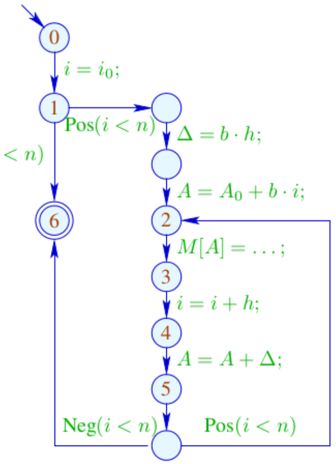
  → The initialization must be transformed into:
  $A = A_0 + b \cdot i_0$.

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

  → b must always be different from zero !!!

... 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 :

→ i may not be used else-where.

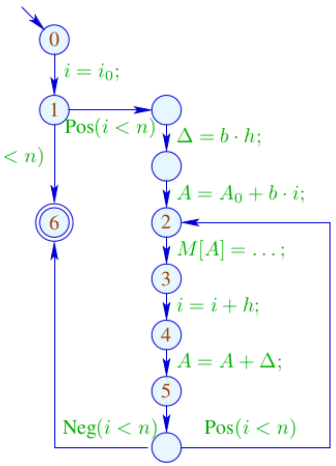
→ The initialization must be transformed into:
$A = A_0 + b \cdot i_0$.

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

→ b must always be different from zero !!!

... 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 :

→ i may not be used else-where.

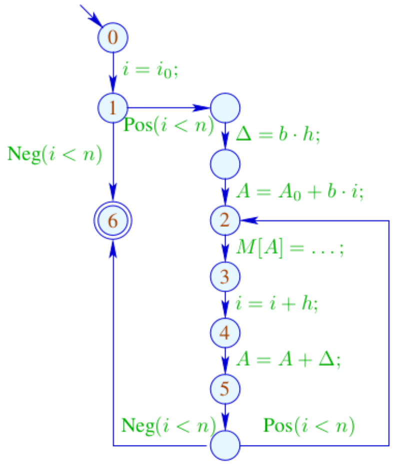
→ The initialization must be transformed into:
$A = A_0 + b \cdot i_0$.

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

→ b must always be different from zero !!!

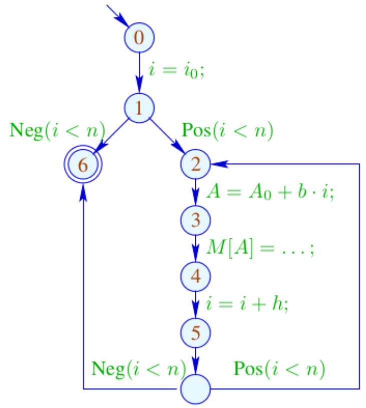
... and reduction of strength:

$i = i_0;$
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 $(i < n);$
}



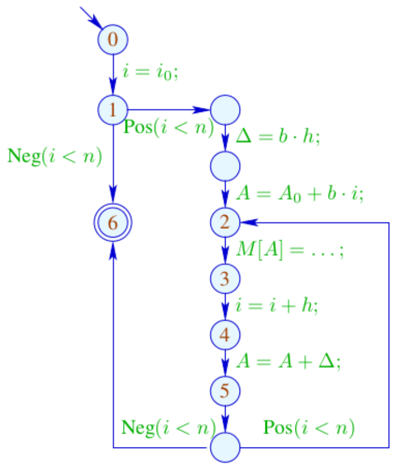
... or, after loop rotation:

$i = i_0;$
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 $(i < n);$
}



Approach:

Identify

... loops;

... iteration variables;

... constants;

... the matching use structures.

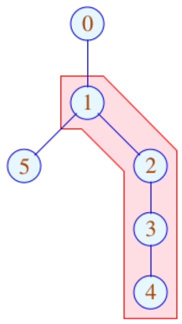
Loops:

... 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:

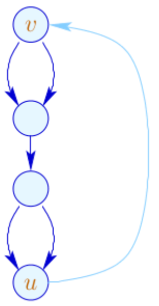
$$\mathsf{Loop}[v] \quad = \quad \{w \mid w \rightarrow^* v \;\text{ in }\; G_v\}$$

Example:



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

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



This property can be expressed by means of the pre-dominator relation ...