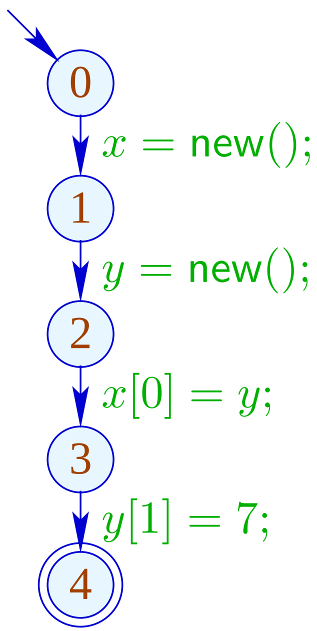
The analysis iterates over all edges once:

$$\pi = \{\{x\}, \{x[\,]\} \mid x \in \mathit{Vars}\};$$
$$\textsf{forall} \quad k = (\_, \mathit{lab}, \_) \quad \textsf{do} \quad \pi = [\![\mathit{lab}]\!]^\sharp\,\pi;$$

where:

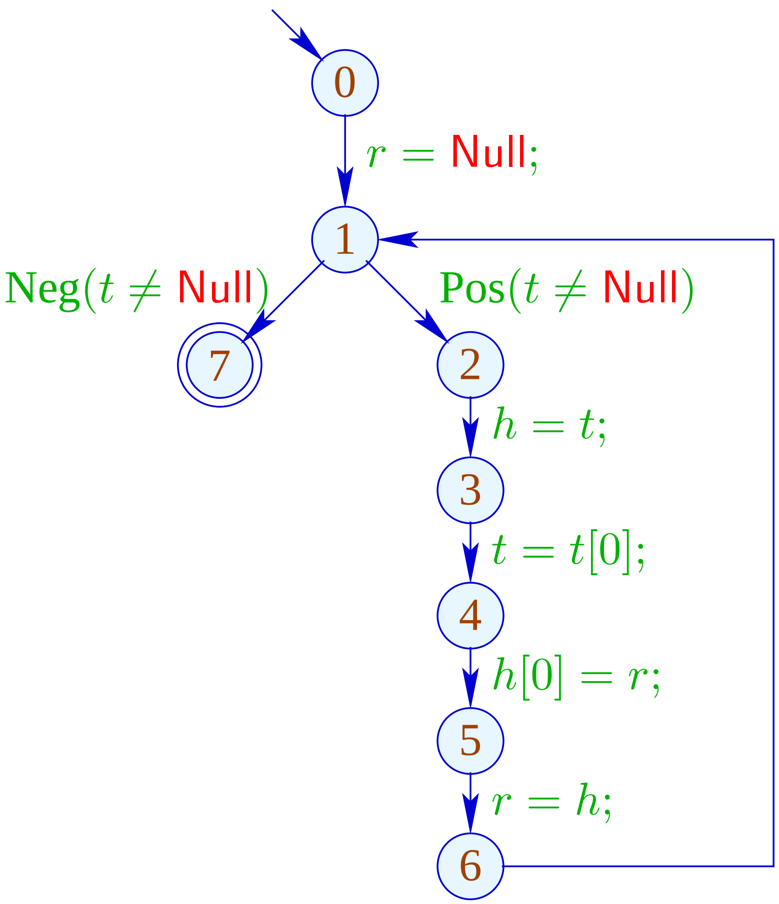
$$\begin{array}{lll}
[\![x = y;]\!]^\sharp\,\pi & = & \textsf{union}^*\,(\pi, x, y) \\
[\![x = y[e];]\!]^\sharp\,\pi & = & \textsf{union}^*\,(\pi, x, y[\,]) \\
[\![y[e] = x;]\!]^\sharp\,\pi & = & \textsf{union}^*\,(\pi, x, y[\,]) \\
[\![\mathit{lab}]\!]^\sharp\,\pi & = & \pi \qquad\qquad \text{otherwise}
\end{array}$$

... in the Simple Example:



	$\{\{x\},\{y\},\{x[\,]\},\{y[\,]\}\}$
$(0,1)$	$\{\{x\},\{y\},\{x[\,]\},\{y[\,]\}\}$
$(1,2)$	$\{\{x\},\{y\},\{x[\,]\},\{y[\,]\}\}$
$(2,3)$	$\{\{x\},\boxed{\{y,x[\,]\}},\{y[\,]\}\}$
$(3,4)$	$\{\{x\},\{y,x[\,]\},\{y[\,]\}\}$

... in the More Complex Example:



	$\{\{h\}, \{r\}, \{t\}, \{h[\,]\}, \{t[\,]\}\}$
$(2, 3)$	$\{\{h, t\}, \{r\}, \{h[\,], t[\,]\}\}$
$(3, 4)$	$\{\{h, t, h[\,], t[\,]\}, \{r\}\}$
$(4, 5)$	$\{\{h, t, r, h[\,], t[\,]\}\}$
$(5, 6)$	$\{\{h, t, r, h[\,], t[\,]\}\}$

Caveat:

In order to find something, we must assume that variables / addresses always receive a value before they are accessed.

Complexity:

we have:

$\mathcal{O}(\#\, edges + \#\, Vars)$	calls of	union*
$\mathcal{O}(\#\, edges + \#\, Vars)$	calls of	find
$\mathcal{O}(\#\, Vars)$	calls of	union

$\Longrightarrow$ We require efficient Union-Find data-structure :-)

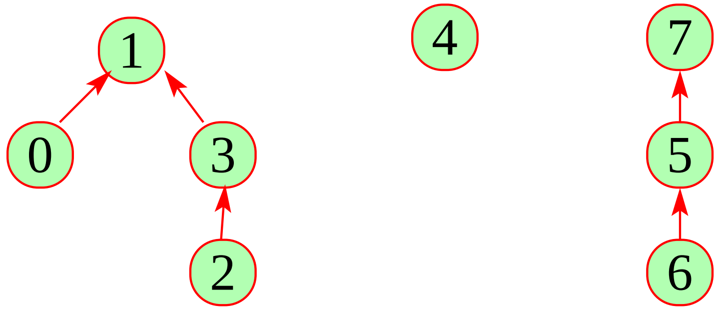
Idea:

Represent partition of U as directed forest:

- For $u \in U$ a reference $F[u]$ to the father is maintained;
- Roots are elements u with $F[u] = u$.

Single trees represent equivalence classes.

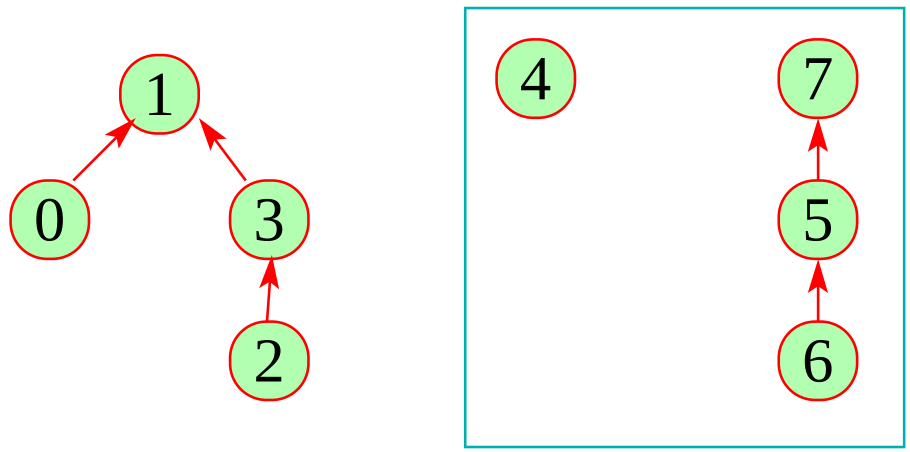
Their roots are their representatives ...



0	1	2	3	4	5	6	7
1	1	3	1	4	7	5	7

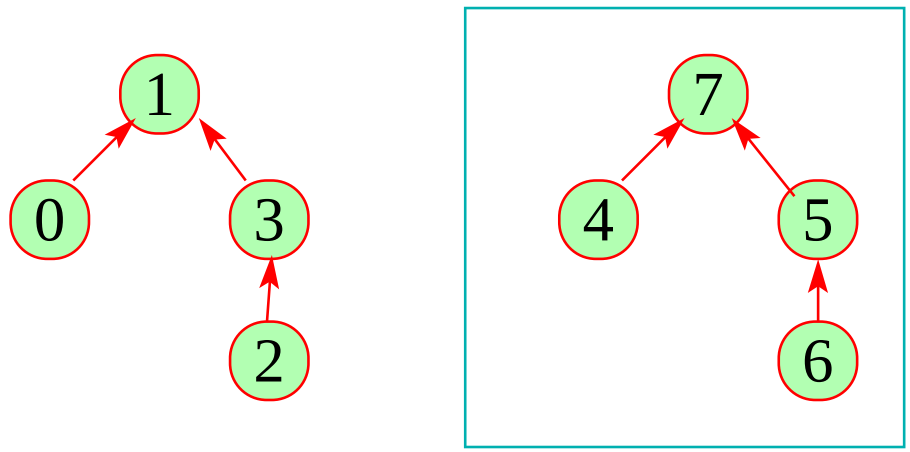
$\rightarrow$ find (π, u) follows the father references :-)

$\rightarrow$ union (π, u_1, u_2) re-directs the father reference of one u_i ...



0	1	2	3	4	5	6	7

1	1	3	1	4	7	5	7



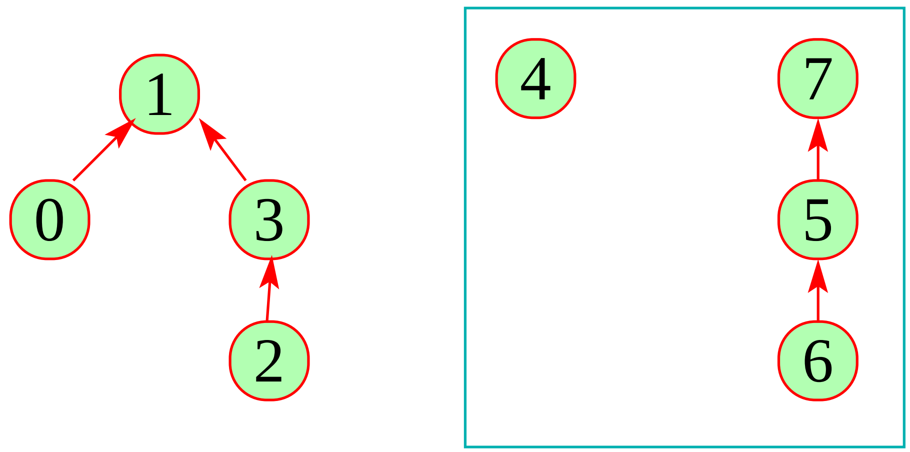
0	1	2	3	4	5	6	7
1	1	3	1	7	7	5	7

The Costs:

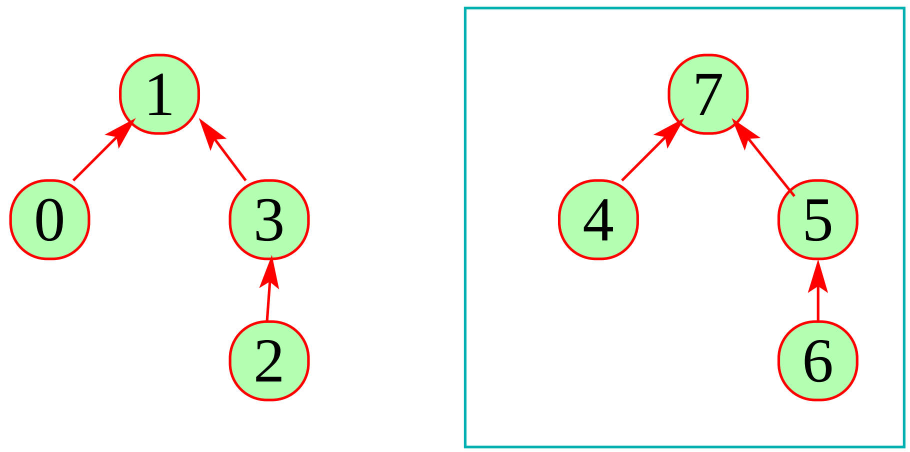
union	:	$\mathcal{O}(1)$	:-)
find	:	$\mathcal{O}(depth(\pi))$	:-(

Strategy to Avoid Deep Trees:

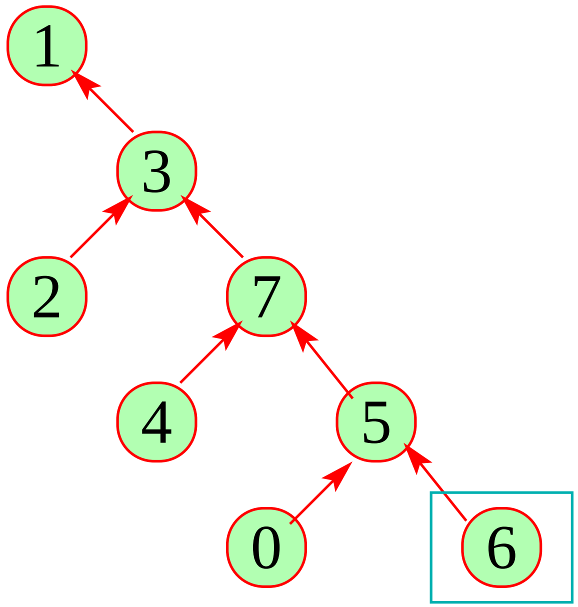
- Put the smaller tree below the bigger !
- Use find to compress paths ...



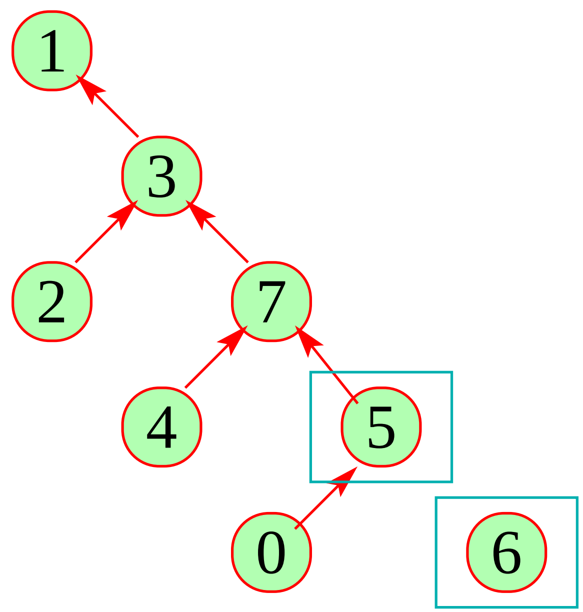
0	1	2	3	4	5	6	7
1	1	3	1	4	7	5	7



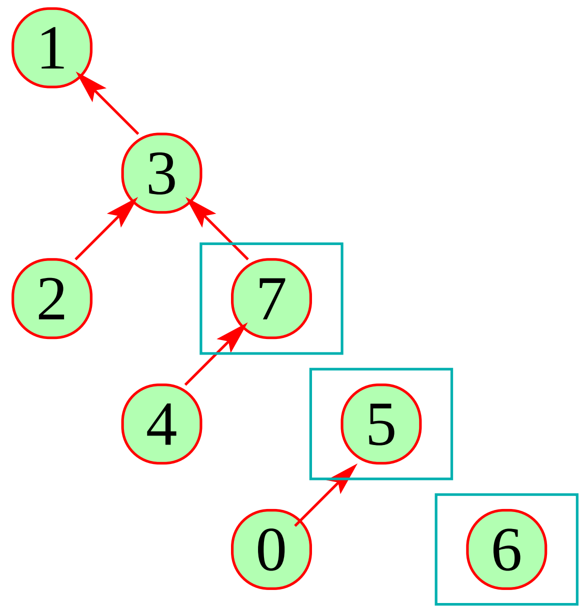
0	1	2	3	4	5	6	7
1	1	3	1	7	7	5	7



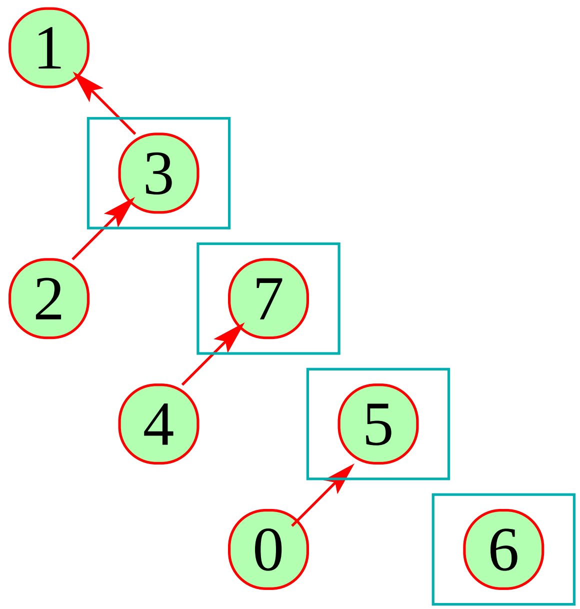
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



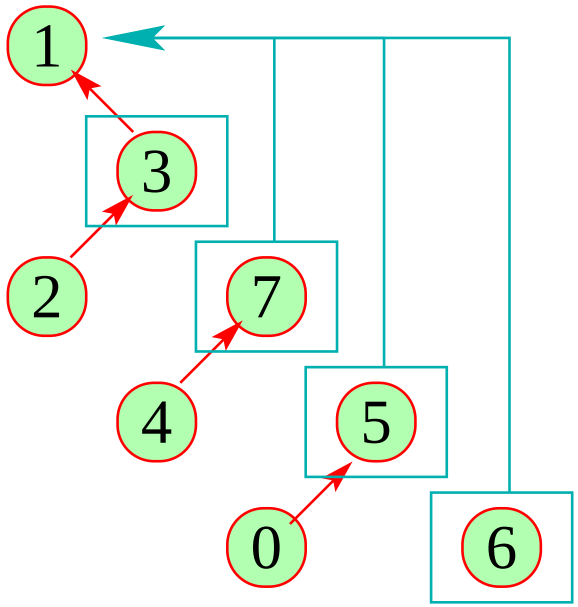
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



0	1	2	3	4	5	6	7
5	1	3	1	1	7	1	1

Robert Endre Tarjan, Princeton

Note:

- By this data-structure, n union- und m find operations require time $\mathcal{O}(n + m \cdot \alpha(n, n))$

  // α the inverse Ackermann-function :-)

- For our application, we only must modify union such that roots are from *Vars* whenever possible.

- This modification does not increase the asymptotic run-time. :-)

Summary:

The analysis is extremely fast — but may not find very much.

Background 3: Fixpoint Algorithms

Consider: $x_i \quad \sqsupseteq \quad f_i\,(x_1, \ldots, x_n)\,, \quad i = 1, \ldots, n$

Observation:

RR-Iteration is inefficient:

→ We require a complete round in order to detect termination :-(

→ If in some round, the value of just one unknown is changed, then we still re-compute all :-(

→ The practical run-time depends on the ordering on the variables :-(

Idea: Worklist Iteration

If an unknown x_i changes its value, we re-compute all unknowns which depend on x_i . Technically, we require:

- $\to$ the lists $Dep\ f_i$ of unknowns which are accessed during evaluation of f_i. From that, we compute the lists:

$$I[x_i] = \{x_j \mid x_i \in Dep\ f_j\}$$

  i.e., a list of all x_j which depend on the value of x_i ;

- $\to$ the values $D[x_i]$ of the x_i where initially $D[x_i] = \bot$;
- $\to$ a list W of all unknowns whose value must be recomputed ...

The Algorithm:

$$
\begin{array}{l}
W = [x_1, \ldots, x_n]; \\
\textsf{while } (W \neq [\,]) \; \{ \\
\qquad \begin{array}{lcl} x_i & = & \textsf{extract } W; \\ t & = & f_i \textsf{ eval}; \\ t & = & D[x_i] \sqcup t; \end{array} \\
\qquad \textsf{if } (t \neq D[x_i]) \; \{ \\
\qquad\qquad \begin{array}{lcl} D[x_i] & = & t; \\ W & = & \textsf{append } I[x_i] \; W; \end{array} \\
\qquad \} \\
\}
\end{array}
$$

$$
\text{where}: \quad \mathit{eval} \;\; x_j \quad = \quad D[x_j]
$$

Example:

$$x_1 \supseteq \{a\} \cup x_3$$
$$x_2 \supseteq x_3 \cap \{a, b\}$$
$$x_3 \supseteq x_1 \cup \{c\}$$

	I
x_1	$\{x_3\}$
x_2	$\emptyset$
x_3	$\{x_1, x_2\}$

Example:

$$x_1 \supseteq \{a\} \cup x_3$$
$$x_2 \supseteq x_3 \cap \{a, b\}$$
$$x_3 \supseteq x_1 \cup \{c\}$$

	I
x_1	$\{x_3\}$
x_2	$\emptyset$
x_3	$\{x_1, x_2\}$

$D[x_1]$	$D[x_2]$	$D[x_3]$	W
$\emptyset$	$\emptyset$	$\emptyset$	$\boxed{x_1}, x_2, x_3$
$\{a\}$	$\emptyset$	$\emptyset$	$\boxed{x_2}, x_3$
$\{a\}$	$\emptyset$	$\emptyset$	$\boxed{x_3}$
$\{a\}$	$\emptyset$	$\{a, c\}$	$\boxed{x_1}, x_2$
$\{a, c\}$	$\emptyset$	$\{a, c\}$	$\boxed{x_3}, x_2$
$\{a, c\}$	$\emptyset$	$\{a, c\}$	$\boxed{x_2}$
$\{a, c\}$	$\{a\}$	$\{a, c\}$	[ ]

Theorem

Let $x_i \sqsupseteq f_i\,(x_1, \ldots, x_n)$, $i = 1, \ldots, n$ denote a constraint system over the complete lattice $\mathbb{D}$ of hight $h > 0$.

(1) The algorithm terminates after at most $h \cdot N$ evaluations of right-hand sides where

$$N = \sum_{i=1}^{n} (1 + \#\,(Dep\ f_i)) \qquad // \quad \text{size of the system} \quad \text{:-)}$$

(2) The algorithm returns a solution.
If all f_i are monotonic, it returns the least one.

Proof:

Ad (1):

Every unknown x_i may change its value at most h times :-)

Each time, the list $I[x_i]$ is added to W .

Thus, the total number of evaluations is:

$$
\begin{aligned}
&\leq && n + \sum_{i=1}^{n} (h \cdot \#\,(I[x_i])) \\
&= && n + h \cdot \sum_{i=1}^{n} \#\,(I[x_i]) \\
&= && n + h \cdot \sum_{i=1}^{n} \#\,(\mathit{Dep}\ f_i) \\
&\leq && h \cdot \sum_{i=1}^{n} (1 + \#\,(\mathit{Dep}\ f_i)) \\
&= && h \cdot N
\end{aligned}
$$

Ad (2):

We only consider the assertion for monotonic f_i .

Let D_0 denote the least solution. We show:

- $D_0[x_i] \sqsupseteq D[x_i]$ (all the time)
- $D[x_i] \not\sqsupseteq f_i \,\textsf{eval} \implies x_i \in W$ (at exit of the loop body)
- On termination, the algo returns a solution :-))

Discussion:

- In the example, fewer evaluations of right-hand sides are required than for RR-iteration :-)
- The algo also works for non-monotonic f_i :-)
- For monotonic f_i, the algo can be simplified:

$$\boxed{t = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{;}$$

- In presence of widening, we replace:

$$\boxed{t = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{t = D[x_i] \sqcup\!\!\!\!\lrcorner\, t;}$$

- In presence of Narrowing, we replace:

$$\boxed{t = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{t = D[x_i] \sqcap\!\!\!\!\urcorner\, t;}$$

Warning:

- The algorithm relies on explicit dependencies among the unknowns.

  So far in our applications, these were obvious. This need not always be the case :-(

- We need some strategy for extract which determines the next unknown to be evaluated.

- It would be ingenious if we always evaluated first and then accessed the result ... :-)

$\Longrightarrow$ recursive evaluation ...

Idea:

→ If during evaluation of f_i , an unknown x_j is accessed, x_j is first solved recursively. Then x_i is added to $I[x_j]$:-)

$$\begin{array}{lll} \textsf{eval}\ x_i\ x_j & = & \textsf{solve}\,x_j; \\ & & I[x_j] = I[x_j] \cup \{x_i\}; \\ & & D[x_j]; \end{array}$$

→ In order to prevent recursion to descend infinitely, a set *Stable* of unknown is maintained for which solve just looks up their values :-)

Initially, $Stable = \emptyset$ ...

The Function solve :

$$
\begin{array}{lll}
\textsf{solve}\ x_i & = & \textsf{if}\ (x_i \notin \mathit{Stable})\ \{ \\
 & & \quad \mathit{Stable} = \mathit{Stable} \cup \{x_i\}; \\
 & & \quad t = f_i\,(\textsf{eval}\ \ x_i); \\
 & & \quad t = D[x_i] \sqcup t; \\
 & & \quad \textsf{if}\ (t \neq D[x_i])\ \{ \\
 & & \qquad W = I[x_i]; \quad I[x_i] = \emptyset; \\
 & & \qquad D[x_i] = t; \\
 & & \qquad \mathit{Stable} = \mathit{Stable} \backslash W; \\
 & & \qquad \textsf{app}\ \ \textsf{solve}\ \ W; \\
 & & \quad \} \\
 & & \}
\end{array}
$$

Helmut Seidl, TU München ;-)

Example:

Consider our standard example:

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

A trace of the fixpoint algorithm then looks as follows:

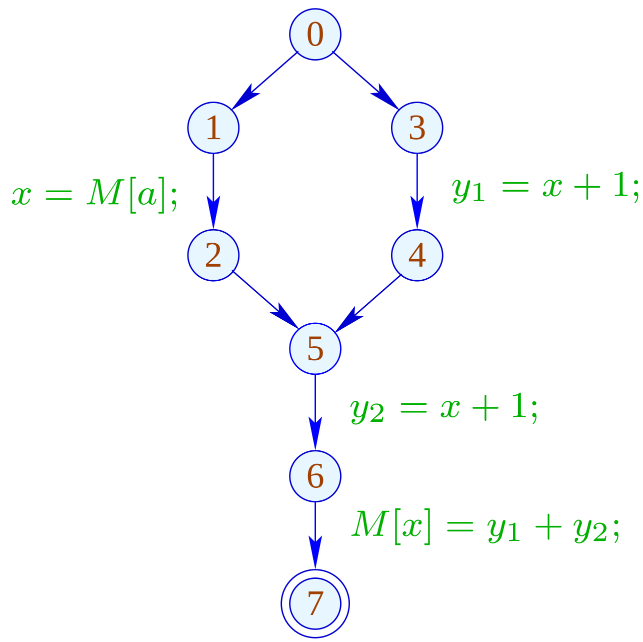
solve x_2	eval x_2 x_3	solve x_3	eval x_3 x_1	solve x_1	eval x_1 x_3	solve x_3
						stable!
					$I[x_3] = \{x_1\}$	
					$\Rightarrow \quad \emptyset$	
				$D[x_1] = \{a\}$		
			$I[x_1] = \{x_3\}$			
			$\Rightarrow \quad \{a\}$			
		$D[x_3] = \{a, c\}$				
		$I[x_3] = \emptyset$				
		solve x_1	eval x_1 x_3	solve x_3		
				stable!		
			$I[x_3] = \{x_1\}$			
			$\Rightarrow \quad \{a, c\}$			
		$D[x_1] = \{a, c\}$				
		$I[x_1] = \emptyset$				
		solve x_3	eval x_3 x_1	solve x_1		
				stable!		
			$I[x_1] = \{x_3\}$			
			$\Rightarrow \quad \{a, c\}$			
		ok				
	$I[x_3] = \{x_1, x_2\}$					
	$\Rightarrow \quad \{a, c\}$					
$D[x_2] = \{a\}$						

- $\rightarrow$ Evaluation starts with an interesting unknown x_i (e.g., the value at *stop*)
- $\rightarrow$ Then automatically all unknowns are evaluated which influence x_i :-)
- $\rightarrow$ The number of evaluations is often smaller than during worklist iteration ;-)
- $\rightarrow$ The algorithm is more complex but does not rely on pre-computation of variable dependencies :-))
- $\rightarrow$ It also works if variable dependencies during iteration change !!!

$\Longrightarrow$ interprocedural analysis

1.7 Eliminating Partial Redundancies

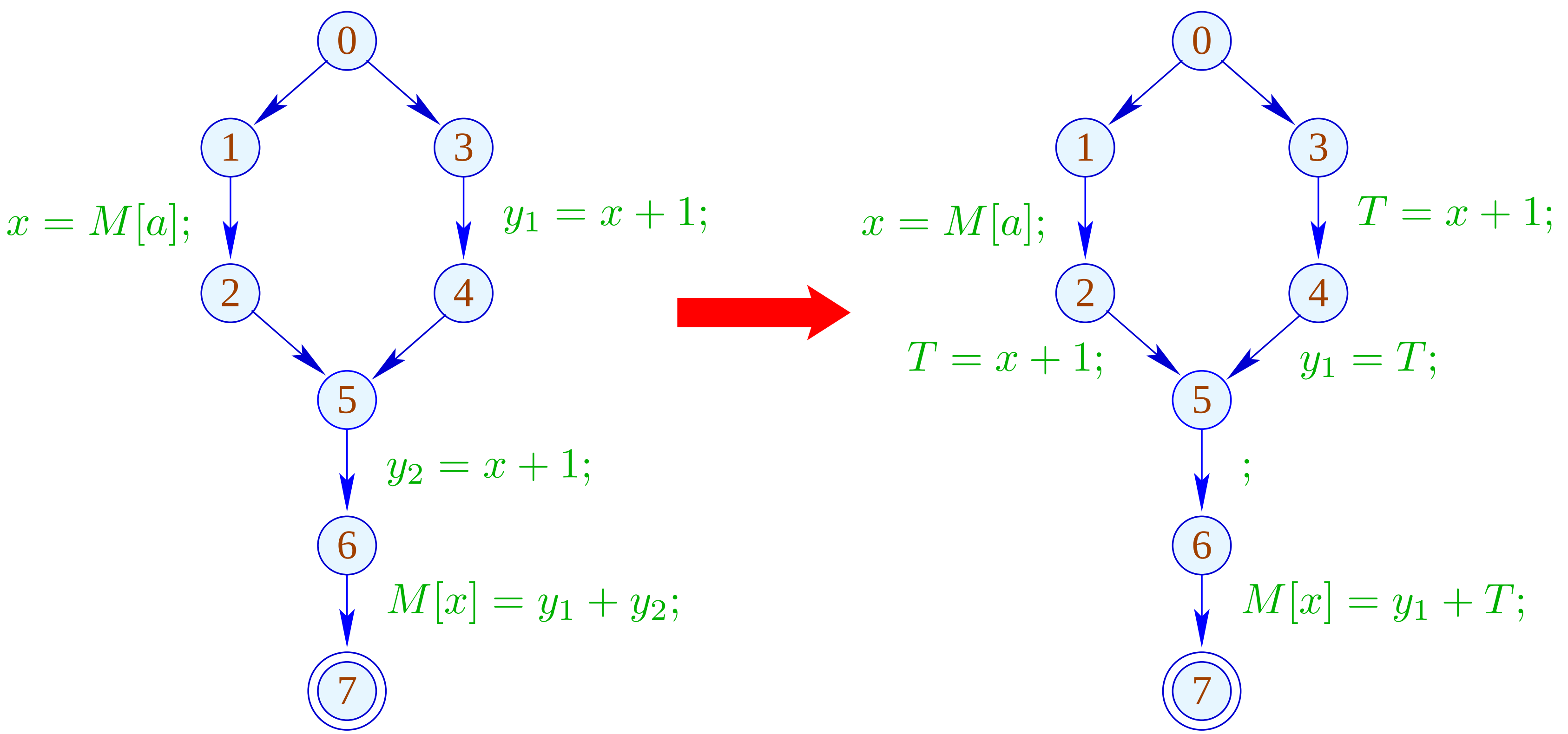
Example:



// $x + 1$ is evaluated on every path ...

// on one path, however, even twice :-(

Goal:


0
1
3
$x = M[a];$
$y_1 = x + 1;$
2
4
5
$y_2 = x + 1;$
6
$M[x] = y_1 + y_2;$
7
0
1
3
$x = M[a];$
$T = x + 1;$
2
4
$T = x + 1;$
$y_1 = T;$
5
;
6
$M[x] = y_1 + T;$
7

Idea:

(1) Insert assignments $T_e = e;$ such that e is available at all points where the value of e is required.

(2) Thereby spare program points where e either is already available or will definitely be computed in future.

Expressions with the latter property are called very busy.

(3) Replace the original evaluations of e by accesses to the variable T_e.

$\Longrightarrow$ we require a novel analysis :-))

An expression e is called busy along a path π , if the expression e is evaluated before any of the variables $x \in Vars(e)$ is overwritten.

$//$ backward analysis!

e is called very busy at u , if e is busy along every path $\pi : u \to^* stop$.

An expression e is called busy along a path π , if the expression e is evaluated before any of the variables $x \in \mathit{Vars}(e)$ is overwriten.

$//$ backward analysis!

e is called very busy at u , if e is busy along every path $\pi : u \to^* \mathit{stop}$.

Accordingly, we require:

$$\mathcal{B}[u] \;=\; \bigcap\{[\![\pi]\!]^\sharp\, \emptyset \mid \pi : u \to^* \mathit{stop}\}$$

where for $\pi = k_1 \ldots k_m$:

$$[\![\pi]\!]^\sharp \;=\; [\![k_1]\!]^\sharp \circ \ldots \circ [\![k_m]\!]^\sharp$$

Our complete lattice is given by:

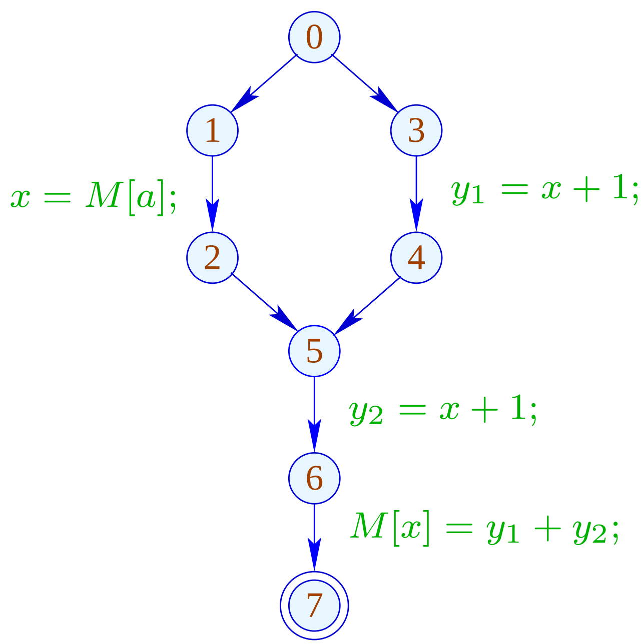
$$\mathbb{B} = 2^{Expr \setminus Vars} \qquad \text{with} \quad \sqsubseteq \; = \; \supseteq$$

The effect $[\![k]\!]^\sharp$ of an edge $k = (u, lab, v)$ only depends on lab , i.e., $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ where:

$$\begin{array}{lll} [\![;]\!]^\sharp \, B & = & B \\ [\![Pos(e)]\!]^\sharp \, B & = & [\![Neg(e)]\!]^\sharp \, B \qquad = \quad B \cup \{e\} \\ [\![x = e;]\!]^\sharp \, B & = & (B \setminus Expr_x) \cup \{e\} \\ [\![x = M[e];]\!]^\sharp \, B & = & (B \setminus Expr_x) \cup \{e\} \\ [\![M[e_1] = e_2;]\!]^\sharp \, B & = & B \cup \{e_1, e_2\} \end{array}$$

These effects are all distributive. Thus, the least solution of the constraint system yields precisely the MOP — given that *stop* is reachable from every program point :-)

Example:



7	$\emptyset$
6	$\{y_1 + y_2\}$
5	$\{x + 1\}$
4	$\{x + 1\}$
3	$\{x + 1\}$
2	$\{x + 1\}$
1	$\emptyset$
0	$\emptyset$

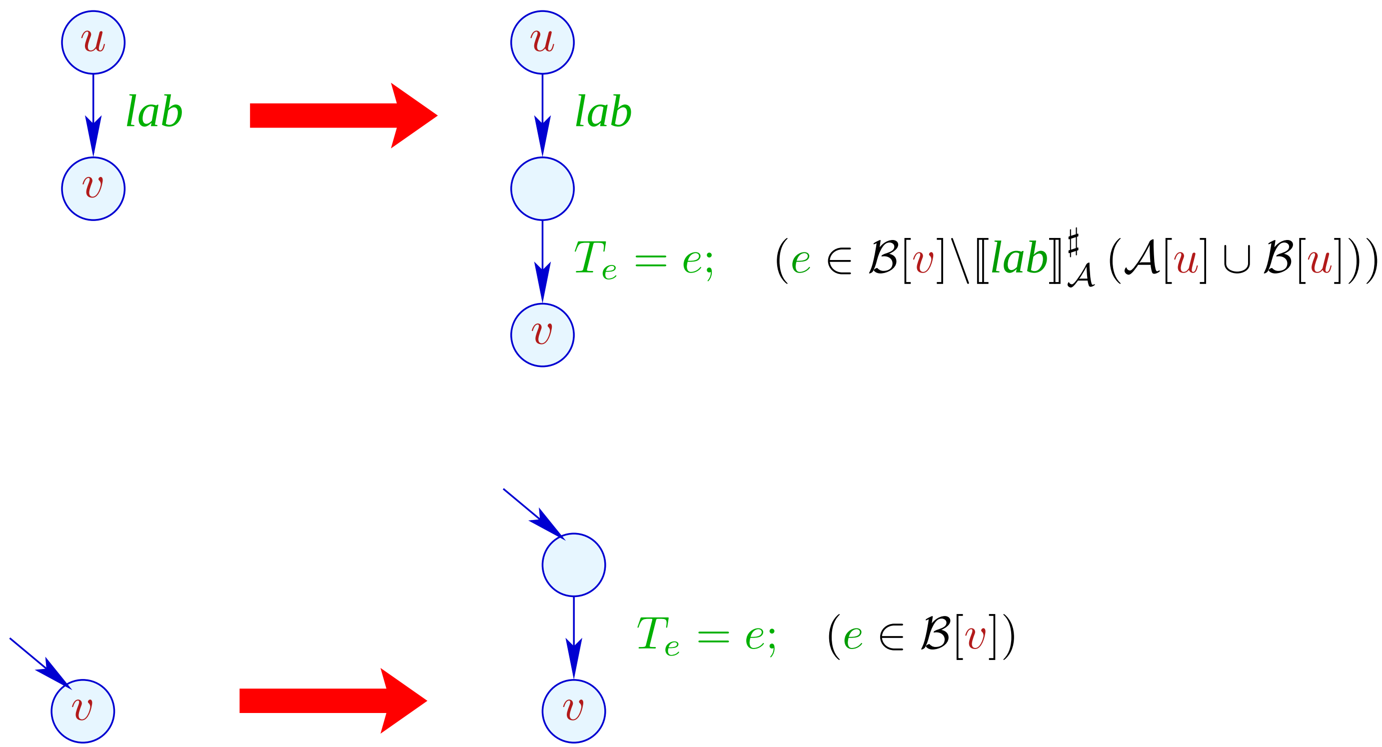
A point u is called safe for e, if $e \in \mathcal{A}[u] \cup \mathcal{B}[u]$, i.e., e is either available or very busy.

Idea:

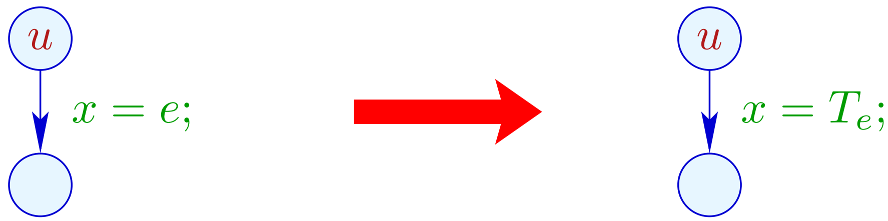
- We insert computations of e such that e becomes available at all safe program points :-)
- We insert $T_e = e;$ after every edge (u, lab, v) with

$$e \in \mathcal{B}[v] \backslash [\![lab]\!]^\sharp_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u])$$

Transformation 5.1:



Transformation 5.2:



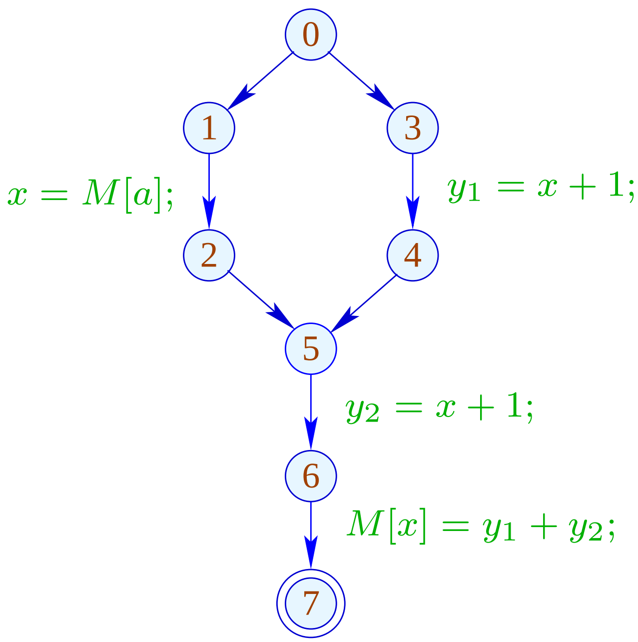
// analogously for the other uses of e

// at old edges of the program.

Bernhard Steffen, Dortmund

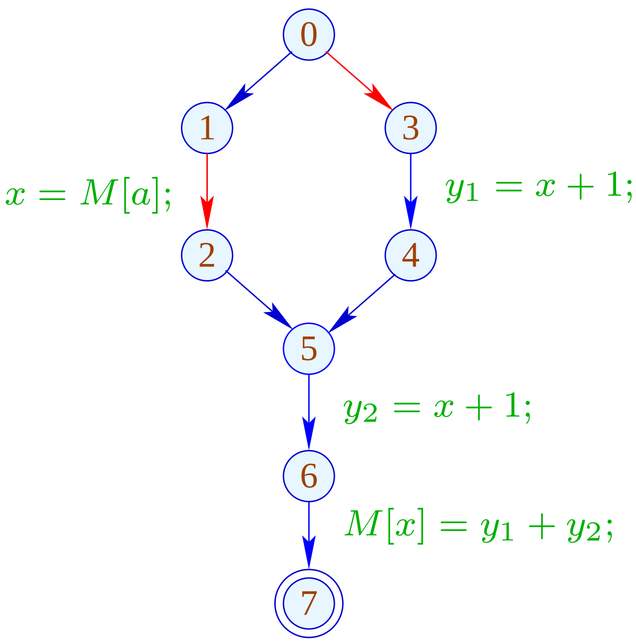
Jens Knoop, Wien

In the Example:



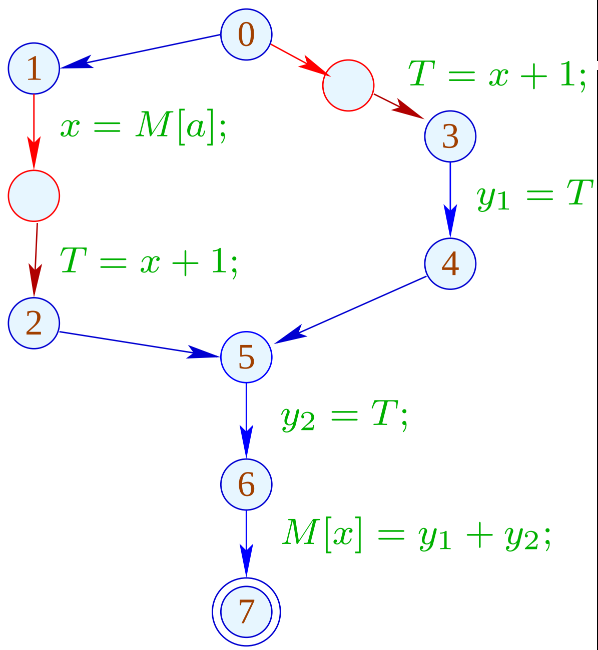
	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x+1\}$
3	$\emptyset$	$\{x+1\}$
4	$\{x+1\}$	$\{x+1\}$
5	$\emptyset$	$\{x+1\}$
6	$\{x+1\}$	$\{y_1+y_2\}$
7	$\{x+1, y_1+y_2\}$	$\emptyset$

In the Example:



	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x+1\}$
3	$\emptyset$	$\{x+1\}$
4	$\{x+1\}$	$\{x+1\}$
5	$\emptyset$	$\{x+1\}$
6	$\{x+1\}$	$\{y_1+y_2\}$
7	$\{x+1, y_1+y_2\}$	$\emptyset$

Im Example:



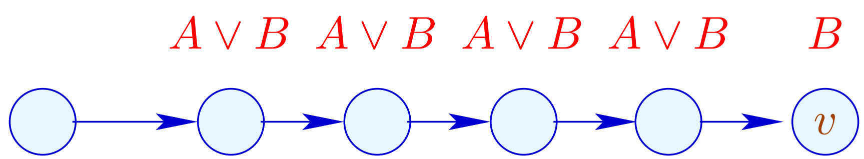
	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x + 1\}$
3	$\emptyset$	$\{x + 1\}$
4	$\{x + 1\}$	$\{x + 1\}$
5	$\emptyset$	$\{x + 1\}$
6	$\{x + 1\}$	$\{y_1 + y_2\}$
7	$\{x + 1, y_1 + y_2\}$	$\emptyset$

Correctness:

Let π denote a path reaching v after which a computation of an edge with e follows.

Then there is a maximal suffix of π such that for every edge $k = (u, lab, u')$ in the suffix:

$$e \in [\![lab]\!]^\sharp_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u])$$



Correctness:

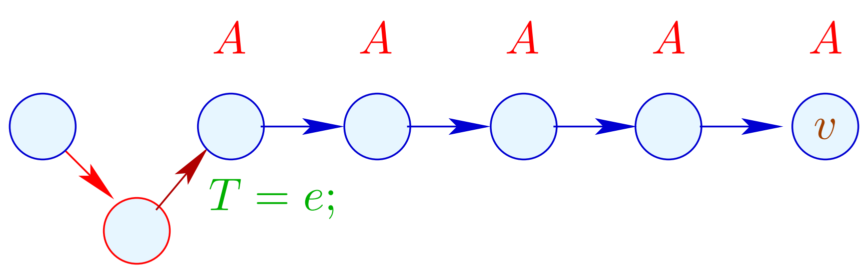
Let π denote a path reaching v after which a computation of an edge with e follows.

Then there is a maximal suffix of π such that for every edge $k = (u, lab, u')$ in the suffix:

$$e \in [\![lab]\!]^\sharp_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u])$$

In particular, no variable in e receives a new value **:-)**

Then $T_e = e;$ is inserted before the suffix **:-))**



We conclude:

- Whenever the value of e is required, e is available :-)

  $\Longrightarrow$ correctness of the transformation

- Every $T = e;$ which is inserted into a path corresponds to an e which is replaced with T :-))

  $\Longrightarrow$ non-degradation of the efficiency