Idea: Worklist Iteration

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

→ 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 ;

→ the values $D[x_i]$ of the x_i where initially $D[x_i] = \bot$;

→ a list W of all unknowns whose value must be recomputed ...

The Algorithm:

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

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

Example:

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

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

Example:

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

	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

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

(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:

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

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:

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

- In presence of Narrowing, we replace:

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

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:

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

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

$\rightarrow$ 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 :

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

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:

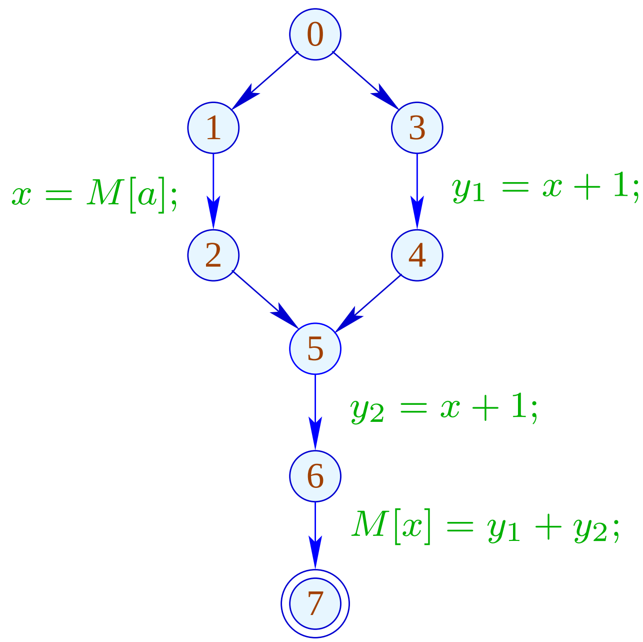
solve x_2	eval x_2 x_3	solve x_3	eval x_3 x_1	solve x_1	eval x_1 x_3	solve x_3
						stable!
					$I[x_3] = \{x_1\}$	
					$\Rightarrow \quad \emptyset$	
				$\boxed{D[x_1] = \{a\}}$		
			$I[x_1] = \{x_3\}$			
			$\Rightarrow \quad \{a\}$			
		$\boxed{D[x_3] = \{a, c\}}$				
		$I[x_3] = \emptyset$		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

- $\rightarrow$ 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

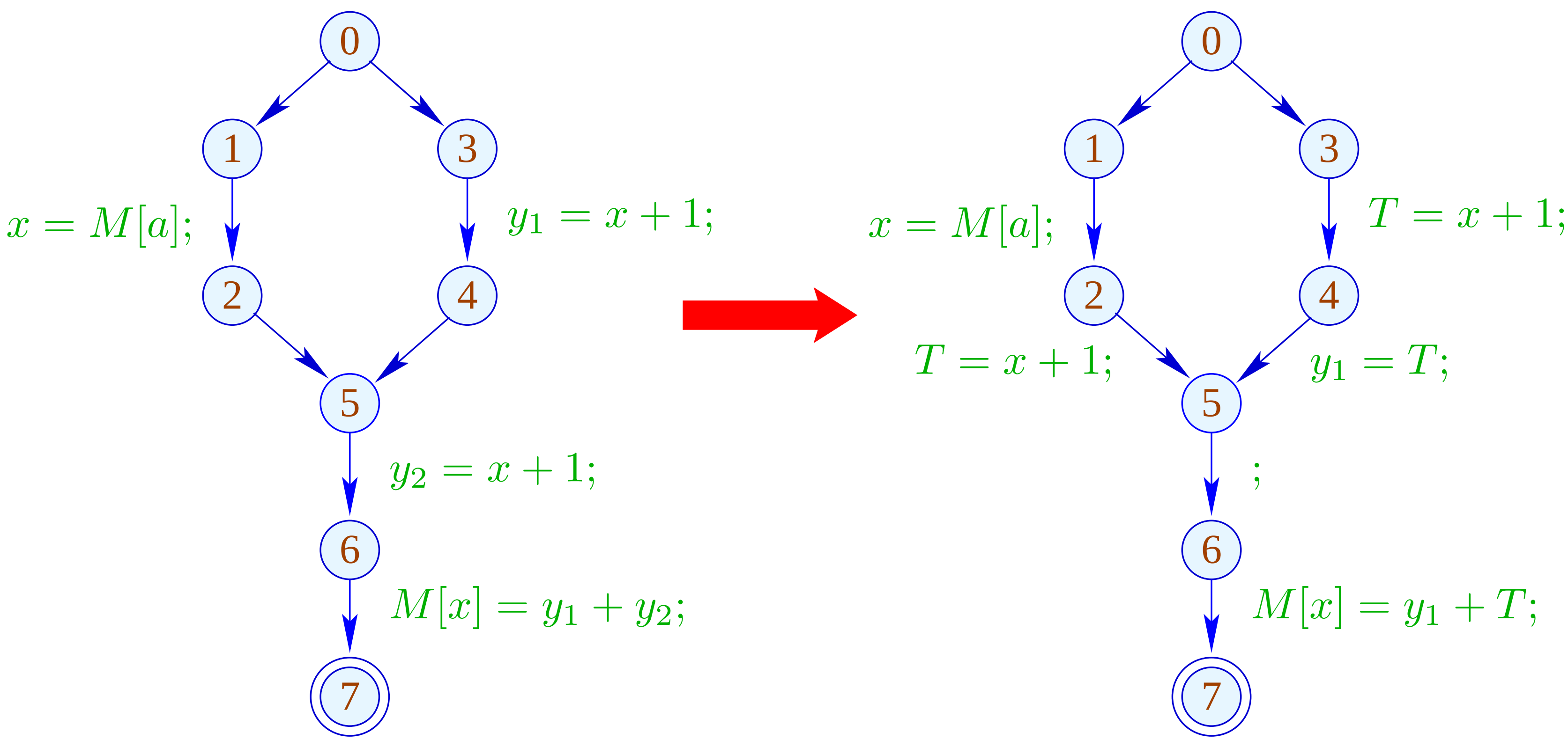
Example:



// $x + 1$ 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 $\pi : u \rightarrow^* stop$.

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 $\pi = k_1 \ldots k_m$:

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