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:

\[
\leq n + \sum_{i=1}^{n} (h \cdot \#(I[x_i])) = n + h \cdot \sum_{i=1}^{n} (\#(I[x_i]) \leq n + h \cdot \sum_{i=1}^{n} (\#(Dep f_i)) \leq h \cdot \sum_{i=1}^{n} (1 + \#(Dep f_i)) \leq h \cdot N
\]

Ad (2):

We only consider the assertion for monotonic \( f_i \).
Let \( D_0 \) denote the least solution. We show:

\[
\begin{align*}
& D_0[x_i] \supseteq D[x_i] \quad \text{(all the time)}
\end{align*}
\]

\[
\begin{align*}
& D[x_i] \not\supseteq f_i \text{ eval} \implies x_i \in W \quad \text{(at exit of the loop body)}
\end{align*}
\]

\[
\begin{align*}
& \text{On termination, the algo returns a solution}\quad :\)
\end{align*}
\]

Theorem

Let \( x_i \supseteq f_i (x_1, \ldots, x_n), \quad i = 1, \ldots, n \) denote a constraint system over the complete lattice \( D \) of height \( h > 0 \).

(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)) \quad // \text{size of the system} :\)

(2) The algorithm returns a solution.
If all \( f_i \) are monotonic, it returns the least one.
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:
  \[
  t = D[x_i] \sqcup t; \quad \implies \quad ;
  \]
  
- In presence of widening, we replace:
  \[
  t = D[x_i] \sqcup t; \quad \implies \quad t = D[x_i] \sqcup t;
  \]
  
- In presence of Narrowing, we replace:
  \[
  t = D[x_i] \sqcup t; \quad \implies \quad t = D[x_i] = t;
  \]

Example:

\[
\begin{align*}
x_1 & \supseteq \{a\} \cup x_3 \\
x_2 & \supseteq x_3 \cap \{a, b\} \\
x_3 & \supseteq x_1 \cup \{c\}
\end{align*}
\]

<table>
<thead>
<tr>
<th>( D[x_1] )</th>
<th>( D[x_2] )</th>
<th>( D[x_3] )</th>
<th>( W )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( x_1, x_2, x_3 )</td>
</tr>
<tr>
<td>( {a} )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( x_2, x_3 )</td>
</tr>
<tr>
<td>( {a} )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( x_3 )</td>
</tr>
<tr>
<td>( {a} )</td>
<td>( \emptyset )</td>
<td>( {a, c} )</td>
<td>( x_1, x_2 )</td>
</tr>
<tr>
<td>( {a, c} )</td>
<td>( {a, c} )</td>
<td>( {a, c} )</td>
<td>( x_3, x_2 )</td>
</tr>
<tr>
<td>( {a, c} )</td>
<td>( \emptyset )</td>
<td>( {a, c} )</td>
<td>( x_3 )</td>
</tr>
<tr>
<td>( {a, c} )</td>
<td>( {a, c} )</td>
<td>( {a, c} )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

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. 😃
  \[
  \implies \quad \text{recursive evaluation} ... 
  \]
Idea:

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

$$\text{eval } x_j = \text{solve } x_j;$$

$$I[x_j] = I[x_j] \cup \{x_i\};$$

$$D[x_j];$$

- In order to prevent recursion to descend infinitely, a set $Stable$ of unknown is maintained for which $\text{solve}$ just looks up their values $\rightarrow$

Initially, $Stable = \emptyset$ ...

The Function $\text{solve}$:

$$\text{solve } x_i = \begin{cases} \text{if } (x_i \not\in Stable) \{ \\
Stable = Stable \cup \{x_i\}; \\
t = f_i(\text{eval } x_i); \\
t = D[x_i] \cup t; \\
\text{if } (t \neq D[x_i]) \{ \\
W = I[x_i]; \\
I[x_i] = \emptyset; \\
D[x_i] = t; \\
\text{Stable} = \text{Stable} \setminus W; \\
\text{app solve } W; \\
\} \end{cases}$$

Example:

Consider our standard example:

$$x_1 \geq \{a\} \cup x_3$$

$$x_2 \geq x_3 \cap \{a, b\}$$

$$x_3 \geq x_1 \cup \{c\}$$

A trace of the fixpoint algorithm then looks as follows:
Evaluation starts with an interesting unknown $x_i$ (e.g., the value at \textit{stop}).

Then automatically all unknowns are evaluated which influence $x_i$. 

The number of evaluations is often smaller than during worklist iteration.

The algorithm is more complex but does not rely on pre-computation of variable dependencies.

It also works if variable dependencies during iteration change!!!

\[ \implies \text{interprocedural analysis} \]

\[ \text{Idea (cont.):} \]

Record when evaluation of a variable has started by means of a set $\textit{Called}$.

Whenever during evaluation of a rhs $f_j$, we detect that no longer $x_i \in \textit{Called}$, we abort ...

\[ \text{eval } x_i \ x_j = \text{solve } x_j; \]
\[ \text{if } (x_i \not\in \textit{Called}) \text{ raise } \textit{Abort}; \]
\[ I[x_j] = I[x_j] \cup \{x_i\}; \]
\[ D[x_j]; \]

Initially, $\textit{Called} = \emptyset$ ...
The new Function \( \text{solve} : \)

\[
\text{solve } x_i = \begin{cases} 
\text{if } (x_i \not\in \text{Stable}) \{} \\
\text{Stable} = \text{Stable} \cup \{x_i\}; \text{Called} = \text{Called} \cup \{x_i\}; \\
\text{try} \{} \\
\quad t = f_i(\text{eval } x_i); \quad t = D[x_i] \cup t; \\
\quad \text{Called} = \text{Called} \setminus \{x_i\}; \\
\quad \text{if } (t \neq D[x_i]) \{} \\
\quad\quad W = I[x_i]; \quad I[x_i] = \emptyset; \\
\quad\quad D[x_i] = t; \\
\quad\quad \text{Stable} = \text{Stable} \setminus W; \\
\quad\quad \text{app solve } W; \\
\quad \} \quad \text{with } \text{Abort } \rightarrow (); \\
\end{cases}
\]

1.7 Eliminating Partial Redundancies

Example:

\[
\begin{align*}
\text{\( x = f[a]; \)} \\
\text{\( y_0 = x + 1; \)} \\
\text{\( y_1 = x + 1; \)} \\
\text{\( M[x] = y_0 + y_1; \)} \\
\end{align*}
\]

// \( x + 1 \) is evaluated on every path ...
// on one path, however, even twice :-(
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 $T_e$.

$\Rightarrow$ we require a novel analysis :-(}

An expression $e$ is called busy along a path $u$, 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 $\pi : u \rightarrow \text{stop}$. Accordingly, we require:

$B[u] = \bigcap \{ \{\pi\} \mid \pi : u \rightarrow \text{stop}\}$

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

$\llbracket \pi \rrbracket^B = \llbracket k_1 \rrbracket^B \circ \ldots \circ \llbracket k_m \rrbracket^B$

Our complete lattice is given by:

$B = 2^{\text{Expr}\setminus Vars}$ with $\subseteq = \supseteq$

The effect $\llbracket k \rrbracket^2$ of an edge $k = (u, lab, v)$ only depends on $lab$, i.e., $\llbracket k \rrbracket^2 = [lab]^2$ where:

$\llbracket e \rrbracket^B = B$

$\llbracket Pos(e) \rrbracket^B = [\text{Neg}(e)]^B B = B \cup \{e\}$

$\llbracket x = e_1 \rrbracket^B = (B \setminus \text{Expr}_x) \cup \{e\}$

$\llbracket x = M[e] \rrbracket^B = (B \setminus \text{Expr}_x) \cup \{e\}$

$\llbracket M[e_1] = e_2 \rrbracket^B = B \cup \{e_1, e_2\}$
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:

\[ x = M[0]; \]
\[ y_1 = x + 1; \]
\[ y_2 = x + 1; \]
\[ M[x] = y_1 + y_2; \]

| 7 | ∅ |
| 6 | \{y_1 + y_2\} |
| 5 | \{x + 1\} |
| 4 | \{x + 1\} |
| 3 | \{x + 1\} |
| 2 | \{x + 1\} |
| 1 | ∅ |
| 0 | ∅ |