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:

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

$$= n + h \cdot \sum_{i=1}^{n} \#(I[x_i])$$

$$= n + h \cdot \sum_{i=1}^{n} \#(Dep f_i)$$

$$\leq h \cdot \sum_{i=1}^{n} (1 + \#(Dep f_i))$$

$$= h \cdot N$$
Ad (2):

We only consider the assertion for monotonic $f_i$.

Let $D_0$ denote the least solution. We show:

- $D_0[x_i] \supseteq D[x_i]$ (all the time)
- $D[x_i] \not\supseteq f_i\text{eval} \implies x_i \in W$ (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:
  
  $$t = D[x_i] \sqcup t; \quad \Rightarrow \quad ;$$

- In presence of widening, we replace:

  $$t = D[x_i] \sqcup t; \quad \Rightarrow \quad t = D[x_i] \sqcup t;$$

- In presence of Narrowing, we replace:

  $$t = D[x_i] \sqcup t; \quad \Rightarrow \quad t = D[x_i] \sqcap t;$$
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 ... :-)

    ⇒ 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 $I[x_j]$ :-)

\[
\text{eval } x_i \ x_j = \text{solve } x_j;
\]
\[
I[x_j] = I[x_j] \cup \{x_i\};
\]
\[
D[x_j];
\]

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

Initially, $\text{Stable} = \emptyset$ ...
The Function \texttt{solve}:

\[
\text{solve } x_i = \begin{cases} 
(x_i \not\in \textit{Stable}) & \{ \\
\textit{Stable} = \textit{Stable} \cup \{x_i\}; \\
t = f_i(\text{eval } x_i); \\
t = D[x_i] \sqcup t; \\
\text{if } (t \neq D[x_i]) & \{ \\
W = I[x_i]; \quad I[x_i] = \emptyset; \\
D[x_i] = t; \\
\textit{Stable} = \textit{Stable} \setminus W; \\
\text{app } \texttt{solve } W;
\end{cases}
\]
Helmut Seidl, TU München ;-)
Example:

Consider our standard example:

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

A trace of the fixpoint algorithm then looks as follows:
\begin{align*}
\text{solve } x_2 & \quad \text{eval } x_2 x_3 & \quad \text{solve } x_3 \\
\text{eval } x_3 x_1 & \quad \text{solve } x_1 & \quad \text{eval } x_1 x_3 & \quad \text{solve } x_3 \\
\text{stable!} & \quad I[x_3] = \{x_1\} & \Rightarrow & \emptyset \\
D[x_1] = \{a\} \\
I[x_1] = \{x_3\} & \Rightarrow & \{a\} \\
D[x_3] = \{a, c\} & \quad I[x_3] = \emptyset \\
\text{solve } x_1 & \quad \text{eval } x_1 x_3 & \quad \text{solve } x_3 \\
\text{stable!} & \quad I[x_3] = \{x_1\} & \Rightarrow & \{a, c\} \\
D[x_1] = \{a, c\} & \quad I[x_1] = \emptyset \\
\text{solve } x_3 & \quad \text{eval } x_3 x_1 & \quad \text{solve } x_1 \\
\text{stable!} & \quad I[x_1] = \{x_3\} & \Rightarrow & \{a, c\} \\
\text{ok} & \quad I[x_3] = \{x_1, x_2\} & \Rightarrow & \{a, c\} \\
D[x_2] = \{a\}
\end{align*}
Evaluation starts with an interesting unknown \( x_i \) (e.g., the value at \( \text{stop} \) )

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 !!!

\[ \text{====> interprocedural analysis} \]
1.7 Eliminating Partial Redundancies

Example:

\[ x = M[a]; \]

\[ y_1 = x + 1; \]

\[ y_2 = x + 1; \]

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

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

421
Goal:

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

\[ T = x + 1; \]
\[ y_1 = T; \]
\[ M[x] = y_1 + T; \]
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 $\pi$, 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 $\pi$, 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:

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

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

$$[[\pi]]^{\#} = [[k_1]]^{\#} \circ \ldots \circ [[k_m]]^{\#}$$
Our complete lattice is given by:

\[ \mathcal{B} = 2^{\text{Expr} \setminus \text{Vars}} \quad \text{with} \quad \subseteq = \supseteq \]

The effect \([ k ]^\#\) of an edge \( k = (u, \text{lab}, v) \) only depends on \( \text{lab} \), i.e., \([ k ]^\# = [\text{lab}]^\#\) where:

\[
\begin{align*}
[;]^\# B &= B \\
[\text{Pos}(e)]^\# B &= [\text{Neg}(e)]^\# B &= B \cup \{e\} \\
[x = e;]^\# B &= (B \setminus \text{Expr}_x) \cup \{e\} \\
[x = M[e];]^\# B &= (B \setminus \text{Expr}_x) \cup \{e\} \\
[M[e_1] = e_2;]^\# B &= B \cup \{e_1, e_2\}
\end{align*}
\]
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:**

\[ x = M[a]; \]
\[ y_1 = x + 1; \]
\[ y_2 = x + 1; \]
\[ M[x] = y_1 + y_2; \]
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 
  
  $$e \in \mathcal{B}[v] \setminus \llbracket lab \rrbracket_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u])$$
Transformation 5.1:

\[ T_e = e; \quad (e \in B[v] \setminus \llbracket lab \rrbracket_A (A[u] \cup B[u])) \]

\[ T_e = e; \quad (e \in B[v]) \]
Transformation 5.2:

\[ u \quad x = e; \quad \rightarrow \quad u \quad x = T_e; \]

// analogously for the other uses of \( e \)
// at old edges of the program.
Bernhard Steffen, Dortmund

Jens Knoop, Wien
In the Example:

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

<table>
<thead>
<tr>
<th></th>
<th>( A )</th>
<th>( B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>1</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>2</td>
<td>( \emptyset )</td>
<td>( { x + 1 } )</td>
</tr>
<tr>
<td>3</td>
<td>( \emptyset )</td>
<td>( { x + 1 } )</td>
</tr>
<tr>
<td>4</td>
<td>( { x + 1 } )</td>
<td>( { x + 1 } )</td>
</tr>
<tr>
<td>5</td>
<td>( \emptyset )</td>
<td>( { x + 1 } )</td>
</tr>
<tr>
<td>6</td>
<td>( { x + 1 } )</td>
<td>( { y_1 + y_2 } )</td>
</tr>
<tr>
<td>7</td>
<td>( { x + 1, y_1 + y_2 } )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
In the Example:

\(x = M[a];\)

\(y_1 = x + 1;\)

\(y_2 = x + 1;\)

\(M[x] = y_1 + y_2;\)

<table>
<thead>
<tr>
<th></th>
<th>(A)</th>
<th>(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>1</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>2</td>
<td>(\emptyset)</td>
<td>({x + 1})</td>
</tr>
<tr>
<td>3</td>
<td>(\emptyset)</td>
<td>({x + 1})</td>
</tr>
<tr>
<td>4</td>
<td>({x + 1})</td>
<td>({x + 1})</td>
</tr>
<tr>
<td>5</td>
<td>(\emptyset)</td>
<td>({x + 1})</td>
</tr>
<tr>
<td>6</td>
<td>({x + 1})</td>
<td>({y_1 + y_2})</td>
</tr>
<tr>
<td>7</td>
<td>({x + 1})</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>
Im Example:

\[ T = x + 1; \]

\[ y_1 = T; \]

\[ y_2 = T; \]

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

<table>
<thead>
<tr>
<th></th>
<th>( A )</th>
<th>( B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>1</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>2</td>
<td>( \emptyset )</td>
<td>( { x + 1 } )</td>
</tr>
<tr>
<td>3</td>
<td>( \emptyset )</td>
<td>( { x + 1 } )</td>
</tr>
<tr>
<td>4</td>
<td>( { x + 1 } )</td>
<td>( { x + 1 } )</td>
</tr>
<tr>
<td>5</td>
<td>( \emptyset )</td>
<td>( { x + 1 } )</td>
</tr>
<tr>
<td>6</td>
<td>( { x + 1 } )</td>
<td>( { y_1 + y_2 } )</td>
</tr>
<tr>
<td>7</td>
<td>( { x + 1 } )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
Correctness:

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

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

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

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

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

$$e \in [\beta]_{i,\Delta}(A[u] \cup B[u])$$

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 $:-)\Rightarrow$ correctness of the transformation

- Every $T = e$; which is inserted into a path corresponds to an $e$ which is replaced with $T$. $:-))\Rightarrow$ non-degradation of the efficiency