The analysis iterates over all edges once:

\[ \pi = \{ (x), (x[]) | x \in \text{Vars} \}; \]

forall \( k = (\_ , \text{lab}, \_ ) \) do \( \pi = [\text{lab}]^2 \pi; \)

where:

\[ [x = y]^3 \pi = \text{union}^* (\pi, \pi, y) \]
\[ [x = y[e]]^3 \pi = \text{union}^* (\pi, \pi, y[]) \]
\[ [y[e] = x]^3 \pi = \text{union}^* (\pi, \pi, y[]) \]
\[ [\text{lab}]^2 \pi = \pi \quad \text{otherwise} \]

... in the Simple Example:

\[ 0 \]
\[ x = \text{new}(); \]
\[ 1 \]
\[ y = \text{new}(); \]
\[ 2 \]
\[ x[0] = y; \]
\[ 3 \]
\[ y[1] = 7; \]
\[ 4 \]
\[ \{ \{x\}, \{y\}, \{x[\_]\}, \{y[\_]\} \} \]
\[ (0, 1) \]
\[ \{ \{x\}, \{y\}, \{x[\_]\}, \{y[\_]\} \} \]
\[ (1, 2) \]
\[ \{ \{x\}, \{y\}, \{x[\_]\}, \{y[\_]\} \} \]
\[ (2, 3) \]
\[ \{ \{x\}, \{y, x[\_]\}, \{y[\_]\} \} \]
\[ (3, 4) \]
\[ \{ \{x\}, \{y, x[\_]\}, \{y[\_]\} \} \]
... in the More Complex Example:

Caveat:

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

Complexity:

we have:

\[ \mathcal{O}(\# \text{edges} + \# \text{Vars}) \] calls of union*
\[ \mathcal{O}(\# \text{edges} + \# \text{Vars}) \] calls of find
\[ \mathcal{O}(\# \text{Vars}) \] calls of union

\[ \Rightarrow \] We require efficient Union-Find data-structure :-)

The analysis iterates over all edges once:

\[ \pi = \{ \{x\}, \{x[\_]\} \mid x \in \text{Vars} \}; \]
for all \( k = (\_, \text{lab}, \_) \) do \( \pi = [\text{lab}]^2 \pi; \)

where:

\[ [x = y]^3 \pi = \text{union}^* (\pi, \pi, y) \]
\[ [x = y[\_]]^3 \pi = \text{union}^* (\pi, x, y[\_]) \]
\[ [y[\_] = x]^3 \pi = \text{union}^* (\pi, x, y[\_]) \]
\[ [\text{lab}]^3 \pi = \pi \quad \text{otherwise} \]
Caveat:
In order to find something, we must assume that variables / addresses always receive a value before they are accessed.

Complexity:
we have:
\[
O(\# \text{ edges} + \# \text{ Vars}) \quad \text{calls of union^*} \\
O(\# \text{ edges} + \# \text{ Vars}) \quad \text{calls of find} \\
O(\# \text{ Vars}) \quad \text{calls of union} \\
\]

\[\Rightarrow\] 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 ...

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
1 & 1 & 3 & 1 & 4 & 7 & 5 & 7 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
1 & 1 & 3 & 1 & 4 & 7 & 5 & 7
\end{array}
\]

\[\Rightarrow\] \text{find} \(\pi, u\) follows the father references \(\ : \)
\[\Rightarrow\] \text{union} \(\pi, u_1, u_2\) re-directs the father reference of one \(u_i\) ...
The Costs:

\[
\begin{align*}
\text{union} & : \mathcal{O}(1) \\
\text{find} & : \mathcal{O}(`depth(\pi)`)
\end{align*}
\]

Strategy to Avoid Deep Trees:

- Put the smaller tree below the bigger!
- Use \text{find} to compress paths ...

Note:

- By this data-structure, \text{n union-} und \text{m find} operations require time \(\mathcal{O}(n + m \cdot \alpha(n, n))\)
  
  // \(\alpha\) the inverse Ackermann-function
- For our application, we only must modify \text{union} such that roots are from \text{Vars} whenever possible.
- This modification does not increase the asymptotic run-time.

Summary:

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

- By this data-structure, \( n \) union- and \( m \) find operations require time \( \mathcal{O}(n + m \cdot \alpha(n, n)) \).
  
  // \( \alpha \) 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:

\[
x_i \equiv f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n
\]

Observation:

RR-Iteration is inefficient:

\[
\begin{align*}
&\rightarrow \quad \text{We require a complete round in order to detect termination} \quad \text{:-(} \\
&\rightarrow \quad \text{If in some round, the value of just one unknown is changed, then we still re-compute all} \quad \text{:-(} \\
&\rightarrow \quad \text{The practical run-time depends on the ordering on the variables} \quad \text{:-(}
\end{align*}
\]

Idea: Worklist Iteration

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

\[
\begin{align*}
&\rightarrow \quad \text{the lists} \quad \text{Dep } f_i \text{ of unknowns which are accessed during evaluation of } f_i. \text{ From that, we compute the lists:} \\
&\quad I[x_i] = \{ x_j \mid x_i \in \text{Dep } f_j \}
\end{align*}
\]

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

\[
\begin{align*}
&\rightarrow \quad \text{the values} \quad D[x_i] \text{ of the } x_i \text{ where initially } D[x_i] = \bot; \\
&\rightarrow \quad \text{a list} \quad W \text{ of all unknowns whose value must be recomputed} \ldots
\end{align*}
\]

The Algorithm:

\[
\begin{align*}
&W = [x_1, \ldots, x_n]; \\
&\text{while } (W \neq []) \{ \\
&\quad x_i = \text{extract } W; \\
&\quad t = f_i \text{ eval}; \\
&\quad t = D[x_i] \cup t; \\
&\quad \text{if } (t \neq D[x_i]) \{ \\
&\quad \quad D[x_i] = t; \\
&\quad \quad W = \text{append } I[x_i] \text{ W}; \\
&\quad \}
\}
\end{align*}
\]

where: eval \( x_i = D[x_i] \)

\[407\]
### Example:

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1 \supseteq {x_1}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$x_2 \supseteq \emptyset$</td>
<td>${x_1, x_2}$</td>
</tr>
</tbody>
</table>

### Example:

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</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>$\emptyset$</td>
<td>${a, c}$</td>
<td>$x_2$</td>
</tr>
<tr>
<td>${a, c}$</td>
<td>${a}$</td>
<td>${a, c}$</td>
<td>$|$</td>
</tr>
</tbody>
</table>

### Theorem

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

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

   \[ N = \sum_{i=1}^{n} (1 + \#(\text{Dep } f_i)) \]

   // size of the system

2. The algorithm returns a solution.

   If all $f_i$ are monotonic, it returns the least one.
Theorem

Let $x_i \supseteq f_i(x_1, \ldots, x_n)$, $i = 1, \ldots, n$ 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
   \[ N = \sum_{i=1}^{n} (1 + \#(\text{Dep } f_i)) \]  // size of the system :-)

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

Example:

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$W$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${a}$</td>
<td>${a}$</td>
<td>${a, c}$</td>
<td>$x_1, x_3$</td>
</tr>
<tr>
<td>${a}$</td>
<td>${a, c}$</td>
<td>${a, c}$</td>
<td>$x_1, x_2$</td>
</tr>
<tr>
<td>${a}$</td>
<td>${a}$</td>
<td>${a}$</td>
<td>$x_1$</td>
</tr>
<tr>
<td>${a}$</td>
<td>${a}$</td>
<td>${a}$</td>
<td>$x_2$</td>
</tr>
<tr>
<td>${a}$</td>
<td>${a}$</td>
<td>${a}$</td>
<td>${}$.</td>
</tr>
</tbody>
</table>

Proof:

Ad (I):

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:

\[ \leq n + \sum_{i=1}^{n} (h \cdot \#(I[x_i])) \]
\[ = n + h \cdot \sum_{i=1}^{n} \#(I[x_i]) \]
\[ = n + h \cdot \sum_{i=1}^{n} \#(\text{Dep } f_i) \]
\[ \leq h \cdot \sum_{i=1}^{n} (1 + \#(\text{Dep } f_i)) \]
\[ = h \cdot N \]
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)
- \( D[x_i] \supseteq f_i \text{eval} \implies x_i \in W \) (at exit of the loop body)
- On termination, the algo returns a solution \( \therefore \))

\[ \]

---

The Algorithm:

\[
W = [x_1, \ldots, x_n];
\]

while \( (W \neq []) \) {
    \[
    x_i = \text{extract } W; \\
    t = f_i \text{eval}; \\
    t = D[x_i] \cup t; \\
    \text{if } (t \neq D[x_i]) \{ \\
        D[x_i] = t; \\
        W = \text{append } I[x_i] W; \\
    \}
    \]

where: \( \text{eval } x_i = D[x_i] \)

---

Example:

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

\[
\begin{array}{c|c}
& I \\
\hline
x_1 & \{x_3\} \\
x_2 & \emptyset \\
x_3 & \{x_1, x_2\}
\end{array}
\]

---

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 \( \therefore \)

- We need some strategy for \( \text{extract} \) which determines the next unknown to be evaluated.

- It would be ingenious if we always evaluated \( \text{first} \) and then accessed the result \( \therefore \) 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 \( I[x_j] \) :-)

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

→ 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, \( \text{Stable} = \emptyset \) ...

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:

\[
t = D[x_j] \cup t; \quad \rightarrow \quad ;
\]

→ In presence of \text{widening}, we replace:

\[
t = D[x_j] \cup t; \quad \rightarrow \quad t = D[x_j] \cup t;
\]

→ In presence of \text{Narrowing}, we replace:

\[
t = D[x_j] \cup t; \quad \rightarrow \quad t = D[x_j] \cap t;
\]

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 ... :-)

\[\quad \rightarrow \quad \text{recursive evaluation} \ldots\]

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 \( I[x_j] \) :-)

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

→ 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, \( \text{Stable} = \emptyset \) ...

\[\quad \rightarrow \quad \text{recursive evaluation} \ldots\]
The Function \( \text{solve} \):

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

Example:

Consider our standard example:

- \( x_1 \supseteq \{a\} \cup x_3 \)
- \( x_2 \supseteq x_3 \cap \{a, b\} \)
- \( x_3 \supseteq x_1 \cup \{c\} \)

A trace of the fixpoint algorithm then looks as follows: