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

Complexity:
we have:

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

\[ \implies \quad \text{We require efficient Union-Find data-structure :) } \]
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 ...
\[ \text{→ find } (\pi, u) \text{ follows the father references } \ :-) \]

\[ \text{→ union } (\pi, u_1, u_2) \text{ re-directs the father reference of one } u_i \ldots \]
The Costs:

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

Strategy to Avoid Deep Trees:

- Put the smaller tree below the bigger!
- Use \textit{find} to compress paths...
Robert Endre Tarjan, Princeton
Note:

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

  // \( \alpha \) 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: \[ x_i \supseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \]

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 :-(

404
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 $\text{Dep } f_i$ of unknowns which are accessed during evaluation of $f_i$. From that, we compute the lists:

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

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 $D[x_i] = \perp$;

→ a list $W$ of all unknowns whose value must be recomputed ...
The Algorithm:

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

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

where:  \( eval \ x_j = D[x_j] \)
Example:

\[ x_1 \supseteq \{a\} \cup x_3 \]
\[ x_2 \supseteq x_3 \cap \{a, b\} \]
\[ x_3 \supseteq x_1 \cup \{c\} \]
Example:

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

<table>
<thead>
<tr>
<th></th>
<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></td>
<td>(x_1, x_2, x_3)</td>
</tr>
<tr>
<td>({a})</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td></td>
<td>(x_2, x_3)</td>
</tr>
<tr>
<td>({a})</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td></td>
<td>(x_3)</td>
</tr>
<tr>
<td>({a})</td>
<td>(\emptyset)</td>
<td>({a, c})</td>
<td></td>
<td>(x_1, x_2)</td>
</tr>
<tr>
<td>({a, c})</td>
<td>(\emptyset)</td>
<td>({a, c})</td>
<td></td>
<td>(x_3, x_2)</td>
</tr>
<tr>
<td>({a, c})</td>
<td>(\emptyset)</td>
<td>({a, c})</td>
<td></td>
<td>(x_2)</td>
</tr>
<tr>
<td>({a, c})</td>
<td>({a})</td>
<td>({a, c})</td>
<td></td>
<td>([])</td>
</tr>
</tbody>
</table>
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 height \( h > 0 \).

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

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

(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:

\[
\leq n + \sum_{i=1}^{n} (h \cdot \#(I[x_i])) \\
= n + h \cdot \sum_{i=1}^{n} \#(I[x_i]) \\
= n + h \cdot \sum_{i=1}^{n} \#(\text{Dep } f_i) \\
\leq h \cdot \sum_{i=1}^{n} (1 + \#(\text{Dep } f_i)) \\
= h \cdot N
\]
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)
- $D[x_i] \not\supseteq f_i \text{eval} \implies x_i \in W$ (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 ... :-)
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 $Stable$ of unknown is maintained for which $\text{solve}$ just looks up their values

Initially, $Stable = \emptyset$ ...
The Function \texttt{solve}: \[
\texttt{solve } x_i \ = \ \text{if } (x_i \notin \texttt{Stable}) \{ \\
\texttt{Stable} = \texttt{Stable} \cup \{x_i\}; \\
t = f_i(\texttt{eval } x_i); \\
t = D[x_i] \sqcup t; \\
\text{if } (t \neq D[x_i]) \{ \\
W = I[x_i]; \quad I[x_i] = \emptyset; \\
D[x_i] = t; \\
\texttt{Stable} = \texttt{Stable} \setminus W; \\
\texttt{app } \texttt{solve } W; \\
\} \\
\} \]
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:
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 $x_3$  

\[ I[x_3] = \{x_1\} \]
\[ \Rightarrow \emptyset \]

\[ D[x_1] = \{a\} \]

\[ I[x_1] = \{x_3\} \]
\[ \Rightarrow \{a\} \]

\[ D[x_3] = \{a, c\} \]

\[ I[x_3] = \emptyset \]

solve $x_1$  
 eval $x_1$ $x_3$  
 solve $x_3$  

\[ I[x_3] = \{x_1\} \]
\[ \Rightarrow \{a, c\} \]

\[ D[x_1] = \{a, c\} \]

\[ I[x_1] = \emptyset \]

solve $x_3$  
 eval $x_3$ $x_1$  
 solve $x_1$  

\[ I[x_1] = \{x_3\} \]
\[ \Rightarrow \{a, c\} \]

\[ D[x_1] = \{a, c\} \]

\[ I[x_3] = \{x_1, x_2\} \]
\[ \Rightarrow \{a, c\} \]

\[ D[x_2] = \{a\} \]
Evaluation starts with an interesting unknown $x_i$ (e.g., the value at $stop$)

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 !!!

$\implies$ interprocedural analysis