Each edge \((u, lab, v)\) gives rise to constraints:

<table>
<thead>
<tr>
<th>(lab)</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = y;)</td>
<td>(P[x] \supseteq P[y])</td>
</tr>
<tr>
<td>(x = \text{new}();)</td>
<td>(P[x] \supseteq {(u, v)})</td>
</tr>
<tr>
<td>(x = y[e];)</td>
<td>(P[x] \supseteq \bigcup{P[f] \mid f \in P[y]})</td>
</tr>
<tr>
<td>(y[e] = x;)</td>
<td>(P[f] \supseteq {f \in P[y] \mid P[f] \cap \Delta = \emptyset}) for all (f \in \text{Addrs})</td>
</tr>
</tbody>
</table>

Other edges have no effect \(\rightarrow\)
Each edge \((u, lab, v)\) gives rise to constraints:

<table>
<thead>
<tr>
<th>lab</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = y);</td>
<td>(\mathcal{P}[x] \supseteq \mathcal{P}[y])</td>
</tr>
<tr>
<td>(x = \text{new}();)</td>
<td>(\mathcal{P}[x] \supseteq {(u, v)})</td>
</tr>
<tr>
<td>(x = y[i];)</td>
<td>(\mathcal{P}[x] \supseteq \bigcup {\mathcal{P}[f] \mid f \in \mathcal{P}[y]})</td>
</tr>
<tr>
<td>(y[i_1] = x;)</td>
<td>(\mathcal{P}[f] \supseteq (f \in \mathcal{P}[y]) \cap \mathcal{P}[x] : \emptyset) for all (f \in \text{Addr}^i)</td>
</tr>
</tbody>
</table>

Other edges have no effect :-(

---

**Discussion:**

- The resulting constraint system has size \(O(k \cdot n)\) for \(k\) abstract addresses and \(n\) edges :-(
- The number of necessary iterations is \(O(k|k + \#Vars|)\) ...
- The computed information is perhaps still too zu precise !?
- In order to prove correctness of a solution \(s^* \in \text{States}^i\) we show:

---

**Alias Analysis**

3. Idea:

Determine one equivalence relation \(\equiv\) on variables \(x\) and memory accesses \(y[i]\) with \(s_1 \equiv s_2\) whenever \(s_1, s_2\) may contain the same address at some \(u_1, u_2\)

... in the Simple Example:

---

**Alias Analysis**

3. Idea:

Determine one equivalence relation \(\equiv\) on variables \(x\) and memory accesses \(y[i]\) with \(s_1 \equiv s_2\) whenever \(s_1, s_2\) may contain the same address at some \(u_1, u_2\)

... in the Simple Example:
Discussion:

- We compute a single information for the whole program.
- The computation of this information maintains partitions \( \pi = \{P_1, \ldots, P_m\} \).
- Individual sets \( P_i \) are identified by means of representatives \( p_i \in P_i \).
- The operations on a partition \( \pi \) are:
  \[
  \text{find} (\pi, p) = p_i \quad \text{if} \quad p \in P_i
  \]
  \[
  \text{union} (\pi, p_1, p_2) = \{P_i \cup P_j\} \cup \{P_j \mid i_j \neq j \neq i_2\}
  \]

- If \( x_1, x_2 \in Vars \) are equivalent, then also \( x_1[] \) and \( x_2[] \) must be equivalent.
- If \( P_i \cap Vars \neq \emptyset \), then we choose \( p_i \in Vars \). Then we can apply \( \text{union} \) recursively:
  \[
  \text{union}^* (\pi, q_1, q_2) = \begin{cases} \text{let} \quad p_{i_1} = \text{find} (\pi, q_1) \\ p_{i_2} = \text{find} (\pi, q_2) \\ \text{in} \quad \text{if} \quad p_{i_1} == p_{i_2} \quad \text{then} \quad \pi \\ \text{else} \quad \text{let} \quad \pi = \text{union} (\pi, p_{i_1}, p_{i_2}) \\ \text{in} \quad \text{if} \quad p_{i_1}, p_{i_2} \in Vars \quad \text{then} \quad \text{union}^* (\pi, p_{i_1}[], p_{i_2}[]) \end{cases}
  \]

The analysis iterates over all edges once:

\[
\pi = \{\{x\}, \{x[]\} \mid x \in Vars\}
\]

for all \( k = (\_ , lab, \_ ) \) do \( \pi = [lab]^2 \pi \)

where:

\[
[x = y] \pi = \text{union}^* (\pi, x, y)
\]

\[
[x = y[z]] \pi = \text{union}^* (\pi, x, y[])
\]

\[
[y[z] = x] \pi = \text{union}^* (\pi, x, y[])
\]

\[
[lab] \pi = \pi \quad \text{otherwise}
\]
... in the Simple Example:

```plaintext
0 x = new();
1 y = new();
2 x[0] = y;
3 y[1] = 7;
```

... in the More Complex Example:

```plaintext
Neg(t ≠ Null)
(0, 1) \{x\}, \{y\}, \{x[\]}, \{y[\]}
(1, 2) \{x\}, \{y\}, \{x[\]}, \{y[\]}
(2, 3) \{x\}, \{y, x[\]}, \{y[\]}
(3, 4) \{x\}, \{y, x[\]}, \{y[\]}
```

```plaintext
Pos(t ≠ Null)
1 h = t;
2 t = t[0];
3 h[0] = r;
4 r = h;
5 r = h;
```

... in the Simple Example:

```plaintext
0 x = new();
1 y = new();
2 x[0] = y;
3 y[1] = 7;
```

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

Complexity:
we have:

- \(O(\# \text{ edges} + \# \text{ Vars})\) calls of union
- \(O(\# \text{ edges} + \# \text{ Vars})\) calls of find
- \(O(\# \text{ Vars})\) calls of union

\(\rightarrow\) We require efficient Union-Find data-structure \(\rightarrow\)
Caveat:
In order to find something, we must assume that variables / addresses always receive a value before they are accessed.

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

$\implies$ 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 ...

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

$\rightarrow$ find $(\pi, u)$ follows the father references
$\rightarrow$ union $(\pi, u_1, u_2)$ re-directs the father reference of one $u_i$ ...
→ find ($\pi, u$) follows the father references
→ union ($\pi, u_1, u_2$) re-directs the father reference of one $u_i$ ...

The Costs:

- union : $\mathcal{O}(1)$
- find : $\mathcal{O}(\text{depth}(\pi))$

Strategy to Avoid Deep Trees:

- Put the smaller tree below the bigger!
- Use find to compress paths ...
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 \( \vdash \)

- 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. \( \vdash \)

Summary:

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

---

Background 3: Fixpoint Algorithms

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

Observation:

RR-Iteration is inefficient:

\[ \rightarrow \] We require a complete round in order to detect termination \( \vdash \)

\[ \rightarrow \] If in some round, the value of just one unknown is changed, then we still re-compute all \( \vdash \)

\[ \rightarrow \] The practical run-time depends on the ordering on the variables \( \vdash \)
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:

$$ I[x_i] = \{ x_j | x_j \in \text{Dep } f_i \} $$

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:

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

```latex
\text{while } (W \neq []) \{ \\
\quad x_i = \text{extract } W; \\
\quad t = f_i \text{ eval}; \\
\quad t = D[x_i] \sqcup 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 \} \\
\}
```

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

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

\[
\begin{array}{|c|c|c|c|c|}
\hline
 & D[x_1] & D[x_2] & D[x_3] & W \\
\hline
\emptyset & \emptyset & \emptyset & x_1 \cup x_2 \\
\{a\} & \emptyset & \emptyset & x_2 \cup x_3 \\
\{a\} & \emptyset & \emptyset & x_3 \\
\{a, c\} & \emptyset & x_1 \cup x_2 \\
\{a, c\} & \emptyset & x_2 \cup x_3 \\
\{a, c\} & \emptyset & x_3 \\
\{a, c\} & \{a\} & \{a, c\} & \emptyset \\
\hline
\end{array}
\]

\[
L
\]

| x_1 | \{x_3\} |
| x_2 | \emptyset |
| x_3 | \{x_1, x_3\} |

\[
\text{Theorem}
\]

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

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

2. The algorithm returns a solution.

If all \( f_i \) are monotonic, it returns the least one.

\[
\text{Proof:}
\]

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:

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

\[
\text{Theorem}
\]

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

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

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:

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