3. Idea:

Determine one equivalence relation \( \equiv \) on variables \( x \) and memory accesses \( y[] \) 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:

\[
\begin{align*}
0 & : x = \text{new}(); \\
1 & : y = \text{new}(); \\
2 & : x[0] = y; \\
3 & : y[1] = 7; \\
\end{align*}
\]

\[
\equiv = \{\{x\}\}, \{y, x[]\}, \{y[]\}\}
\]
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:

\[
\text{find}(\pi, p) = p_i \quad \text{if} \quad p \in P_i \\
\hspace{1cm} // \quad \text{return the representative}
\]
\[
\text{union}(\pi, p_{i_1}, p_{i_2}) = \{P_{i_1} \cup P_{i_2}\} \cup \{P_j \mid i_1 \neq j \neq i_2\} \\
\hspace{1cm} // \quad \text{unions the represented classes}
\]
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:

$$\text{union}^* (\pi, q_1, q_2) = \text{let } p_{i_1} = \text{find} (\pi, q_1)$$
$$\quad p_{i_2} = \text{find} (\pi, q_2)$$
$$\text{in } \text{if } p_{i_1} == p_{i_2} \text{ then } \pi$$
$$\quad \text{else let } \pi = \text{union} (\pi, p_{i_1}, p_{i_2})$$
$$\text{in } \text{if } p_{i_1}, p_{i_2} \in Vars \text{ then}$$
$$\quad \text{union}^* (\pi, p_{i_1}[], p_{i_2}[])$$
The analysis iterates over all edges once:

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

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

where:

\[ [x = y;]^\# \pi = \text{union}^* (\pi, x, y) \]
\[ [x = y[e];]^\# \pi = \text{union}^* (\pi, x, y[\]) \]
\[ [y[e] = x;]^\# \pi = \text{union}^* (\pi, x, y[\]) \]
\[ [lab]^\# \pi = \pi \quad \text{otherwise} \]
... in the Simple Example:

\[
\begin{array}{c|c}
\text{Node} & \text{State} \\
0 & \{\{x\}, \{y\}, \{x[\]\}, \{y[\]\}\} \\
1 & \{\{x\}, \{y\}, \{x[\]\}, \{y[\]\}\} \\
2 & \{\{x\}, \{y\}, \{x[\]\}, \{y[\]\}\} \\
3 & \{\{x\}, \{y, x[\]\}, \{y[\]\}\} \\
4 & \{\{x\}, \{y, x[\]\}, \{y[\]\}\} \\
\end{array}
\]
... in the More Complex Example:

```
r = Null;
```

```
Neg(t ≠ Null)
```

```
Pos(t ≠ Null)
```

```
0
```

```
r = Null;
```

```
1
```

```
h = t;
```

```
2
```

```
t = t[0];
```

```
3
```

```
h[0] = r;
```

```
4
```

```
r = h;
```

```
5
```

```
6
```

```
7
```

```
{\{h\}, \{r\}, \{t\}, \{h[],\}, \{t[]\}}
```

<table>
<thead>
<tr>
<th>(2, 3)</th>
<th>{{h,t}, {r}, {h[],t[]}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3, 4)</td>
<td>{{h, t, h[], t[]} {r}}</td>
</tr>
<tr>
<td>(4, 5)</td>
<td>{{h, t, r, h[], t[]}}</td>
</tr>
<tr>
<td>(5, 6)</td>
<td>{{h, t, r, h[], t[]}}</td>
</tr>
</tbody>
</table>
Caveat:
In order to find something, we must assume that variables / addresses always receive a value before they are accessed.

Complexity:
we have:

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

⇒ 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 ...
$\rightarrow \text{find } (\pi, u) \text{ follows the father references} \quad :-)\quad$

$\rightarrow \text{union } (\pi, u_1, u_2) \text{ re-directs the father reference of one } u_i \ldots$
0 1 2 3 4 5 6 7

1 1 3 1 4 7 5 7
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 \textbf{find} to compress paths...
Robert Endre Tarjan, Princeton
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: \[ x_i \supseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \]

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 :-(
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: \(\text{eval } x_j = D[x_j]\)
Example:

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

<table>
<thead>
<tr>
<th></th>
<th>(I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_1)</td>
<td>{(x_3)}</td>
</tr>
<tr>
<td>(x_2)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(x_3)</td>
<td>{(x_1, x_2)}</td>
</tr>
</tbody>
</table>
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>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>({a})</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

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

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