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{itemize}
  \item $x = \text{new}();$
  \item $y = \text{new}();$
  \item $x[0] = y;$
  \item $y[1] = 7;$
\end{itemize}

$$= \{x[1]\} \cup \{y[1]\}$$

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 $s^t \in \text{States}^t$ we show:

→ We compute a single information to 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:

\begin{itemize}
  \item $\text{find}(\pi, p) = p_i$ if $p \in P_i$
  \item $\text{union}(\pi, p_i, p_j) = \{P_i \cup P_j\} \cup \{P_j \mid i \neq j \neq i_2\}$
\end{itemize}

// returns the representative

// unions the represented classes
The analysis iterates over all edges once:

$$\pi = \{\{x\}, \{x[\_]\} | x \in Vars\};$$

forall \(k = (u, lab, v)\) do \(\pi = [lab]^2 \pi\);

where:

\[
\begin{align*}
[x = y]^2 \pi &= \text{union}^* (\pi, x, y) \\
[x = y[e]]^2 \pi &= \text{union}^* (\pi, x, y[\_]) \\
[y[c] = x]^2 \pi &= \text{union}^* (\pi, x, y[\_]) \\
[lab]^2 \pi &= \pi & \text{otherwise}
\end{align*}
\]

... in the Simple Example

9  \(x = \text{new}();\)  | \{\{x\}, \{y\}, \{x[\_]\}, \{y[\_]\}\}
10 \(y = \text{new}();\)  | \{\{x\}, \{y\}, \{x[\_]\}, \{y[\_]\}\}
11 \(z[0] = y;\)       | \{\{x\}, \{y, x[\_]\}, \{y[\_]\}\}
12 \(y[1] = 7;\)       | \{\{x\}, \{y, x[\_]\}, \{y[\_]\}\}

... in the More Complex Example

Neg(\(t \neq \text{null}\))

Pos(\(t \neq \text{null}\))

\[
\begin{align*}
(2, 3) &= \{\{h, t\}, \{x\}, \{h[\_], t[\_]\}\} \\
(3, 4) &= \{\{h, t, h[\_], t[\_]\}, \{r\}\} \\
(4, 5) &= \{\{h, t, r, h[\_], t[\_]\}\}
\end{align*}
\]
... 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:

- $O(#\ edges + #\ Vars)$ calls of union$^*$
- $O(#\ edges + #\ Vars)$ calls of find
- $O(#\ Vars)$ calls of union

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 ...

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

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

Strategy to Avoid Deep Trees

- Put the smaller tree below the bigger!
- Use find to compress paths...
Remark

- By this data-structure, \( n \text{ union} \) and \( m \text{ find} \) operations require time \( O(n + m \cdot \alpha(n, n)) \)
  \[ \text{// } \alpha \text{ the inverse Ackermann-function.} \]
- 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.

Remark

- By this data-structure, \( n \text{ union} \) and \( m \text{ find} \) operations require time \( O(n + m \cdot \alpha(n, n)) \)
  \[ \text{// } \alpha \text{ the inverse Ackermann-function.} \]
- 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.
Background 3: Fixpoint Algorithms

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

Observation

RR-Iteration is inefficient:

\[ \rightarrow \] We require a complete round in order to detect termination.

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

\[ \rightarrow \] 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:

\[ \rightarrow \] the lists \( \text{Dep } f_i \) of unknowns which are accessed during evaluation of \( f_i \). From that, we compute the lists:

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

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

\[ \rightarrow \] the values \( D[x_i] \) of the \( x_i \) where initially \( D[x_i] = \bot \);

\[ \rightarrow \] a list \( W \) of all unknowns whose value must be recomputed ...

The Algorithm

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

while \((W \neq \{\})\) {

\[ x_i = \text{extract } W; \]

\[ t = f_i \text{eval}; \]

if \((t \notin D[x_i])\) {

\[ D[x_i] = D[x_i] \cup t; \]

\[ W = \text{append } I[x_i] \cup W; \]

\}

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

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

<table>
<thead>
<tr>
<th>(I)</th>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(x_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\emptyset)</td>
<td>({a})</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_3, 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>

Example

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

<table>
<thead>
<tr>
<th>(I)</th>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(x_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\emptyset)</td>
<td>({a})</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, c})</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_3, 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>

Example

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

<table>
<thead>
<tr>
<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>(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_3, 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 \(k > 0\).

1. The algorithm terminates after at most \(k \cdot N\) 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:

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

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