Improvement (Cont.):

→ Also, composition can be directly implemented:

\[(M_1 \circ M_2) x = b' \cup \bigcup_{y \in I} y \quad \text{with} \quad b' = b \cup \bigcup_{z \in I} b_z\]
\[I' = \bigcup_{z \in I} I_z \quad \text{where} \quad M_1 x = b \cup \bigcup_{y \in I} y, \quad M_2 z = b_z \cup \bigcup_{y \in I} y\]

→ The effects of assignments then are:

\[ [x = c]^2 = \begin{cases} \text{Id}_{\text{Vars}} \oplus \{x \mapsto c\} & \text{if } c = e \in \mathbb{Z} \\ \text{Id}_{\text{Vars}} \oplus \{x \mapsto y\} & \text{if } e = y \in \text{Vars} \\ \text{Id}_{\text{Vars}} \oplus \{x \mapsto T\} & \text{otherwise} \end{cases} \]
Improvement (Cont.):

→ Also, composition can be directly implemented:

\[
(M_1 \circ M_2) x = b' \cup \bigcup_{y \in \mathcal{Y}} y
\]

with

\[
b' = b \cup \bigcup_{x \in \mathcal{X}} b_x
\]

\[
\mathcal{I}' = \bigcup_{x \in \mathcal{X}} \mathcal{I}_x
\]

where

\[
M_1 x = b \cup \bigcup_{x \in \mathcal{X}} y
\]

\[
M_2 z = b_z \cup \bigcup_{z \in \mathcal{Z}} y
\]

→ The effects of assignments then are:

\[
[x = c] = \begin{cases} 
\text{Id}_\text{Vars} \oplus \{ x \mapsto c \} & \text{if } c = c \in \mathbb{Z} \\
\text{Id}_\text{Vars} \oplus \{ x \mapsto y \} & \text{if } c = y \in \text{Vars} \\
\text{Id}_\text{Vars} \oplus \{ x \mapsto T \} & \text{otherwise}
\end{cases}
\]

... in the Example:

\[
[t = 0] = \{ a_1 \rightarrow a_1, \text{ret} \rightarrow \text{ret}, t \rightarrow 0 \}
\]

\[
[a_1 = t] = \{ a_1 \rightarrow t, \text{ret} \rightarrow \text{ret}, t \rightarrow t \}
\]

In order to implement the analysis, we additionally must construct the effect of a call \( k = (., f (;); _) \) from the effect of a procedure \( f \):

\[
[k] = H \left( \{ \text{ret} \} \right)
\]

where:

\[
H (M) = \text{Id}_\text{Locals} \oplus (M \circ \text{enter}^2) | \text{Globals}
\]

\[
\text{enter}^2 x = \begin{cases} 
x & \text{if } x \in \text{Globals} \\
0 & \text{otherwise}
\end{cases}
\]
... in the Example:

If \([\text{work}]^1 = \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t \}\)
then \(H[\text{work}]^2 = \text{Id}[t] \oplus \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1 \}\)
\(= \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t \}\)

Now we can perform fixpoint iteration \(\vdash\)

\[
M \circ \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto a_1 \in H[8] \}
\]
\(= \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto a_1 \} \in H[8] \)

... in the Example:

If \([\text{work}]^1 = \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t \}\)
then \(H[\text{work}]^2 = \text{Id}[t] \oplus \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1 \}\)
\(= \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t \}\)

Now we can perform fixpoint iteration \(\vdash\)

\[
[(8, \ldots, 9)]^2 \circ [8]^2 = \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t \} \circ \{ a_1 \mapsto a_1, \text{ret} \mapsto t \}
\]
\(= \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t \}\)
If we know the effects of procedure calls, we can put up a constraint system for determining the abstract state when reaching a program point:

\[
\begin{align*}
R[\text{main}] & \equiv \text{enter}^2 d_0 \\
R[f] & \equiv \text{enter}^2 (R[u]) \\
R[v] & \equiv R[f] \\
R[v] & \equiv [k]^2 (R[u]) \quad k = (u, f(); \ldots) \quad \text{call} \\
R[v] & \equiv \text{entry point of } f \\
R[v] & \equiv [k]^2 (R[u]) \quad k = (u, f(v)) \quad \text{edge}
\end{align*}
\]

... in the Example:

```
0 \{a_1 \rightarrow T, \text{ret} \rightarrow T, t \rightarrow 0\}
1 \{a_1 \rightarrow T, \text{ret} \rightarrow T, t \rightarrow 0\}
2 \{a_1 \rightarrow T, \text{ret} \rightarrow T, t \rightarrow 0\}
3 \{a_1 \rightarrow 0, \text{ret} \rightarrow T, t \rightarrow 0\}
4 \{a_1 \rightarrow 0, \text{ret} \rightarrow 0, t \rightarrow 0\}
5 \{a_1 \rightarrow 0, \text{ret} \rightarrow 0, t \rightarrow 0\}
6 \{a_1 \rightarrow 0, \text{ret} \rightarrow T, t \rightarrow 0\}
```

Discussion:

- At least **copy-constants** can be determined interprocedurally.
- For that, we had to ignore conditions and complex assignments :-(
- In the second phase, however, we could have been more precise :-)
- The extra abstractions were necessary for two reasons:
  1. The set of occurring transformers \( \mathcal{M} \subseteq D \rightarrow D \) must be finite;
  2. The functions \( M \in \mathcal{M} \) must be efficiently implementable ;-
- The second condition can, sometimes, be abandoned ...
Observation:

→ Often, procedures are only called for few distinct abstract arguments.
→ Each procedure need only to be analyzed for these :-)
→ Put up a constraint system:

\[
[v, a]^2 \supseteq a \quad \text{entry point}
\]
\[
[v, a]^3 \supseteq \text{combine}_k ([u, a], [f, \text{enter}([u, a]^2]) \quad (u, f(), v) \quad \text{call}
\]
\[
[v, a]^2 \supseteq [\text{lab}]^2 [u, a]^4 \quad k = (u, \text{lab}, v) \quad \text{edge}
\]
\[
[f, a]^2 \supseteq [\text{stop}_f, a]^2 \quad \text{stop}_f \quad \text{end point of } f
\]
// \[v, a]^2 = \text{value for the argument } a.

Discussion:

- This constraint system may be huge :-(
- We do not want to solve it completely!!
- It is sufficient to compute the correct values for all calls which occur, i.e., which are necessary to determine the value \([\text{main}(), a_0]^2\) \implies We apply our local fixpoint algorithm :-)
- The fixpoint algo provides us also with the set of actual parameters \(a \in D\) for which procedures are (possibly) called and all abstract values at their program points for each of these calls :-)

... in the Example:

Let us try a full constant propagation ...

<table>
<thead>
<tr>
<th>(a_1) ret</th>
<th>(a_2) ret</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T T</td>
</tr>
<tr>
<td>1</td>
<td>T T T T</td>
</tr>
<tr>
<td>2</td>
<td>T T</td>
</tr>
<tr>
<td>3</td>
<td>T T T T</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>T T</td>
</tr>
<tr>
<td>6</td>
<td>0 T</td>
</tr>
<tr>
<td>7</td>
<td>0 0 T</td>
</tr>
<tr>
<td>10</td>
<td>0 T</td>
</tr>
<tr>
<td>16</td>
<td>0 0 0</td>
</tr>
<tr>
<td>5</td>
<td>T T T</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Discussion:

- In the Example, the analysis terminates quickly :-)
- If \(D\) has finite height, the analysis terminates if each procedure is only analyzed for finitely many arguments :-))
- Analogous analysis algorithms have proved very effective for the analysis of Prolog :-)
- Together with a points-to analysis and propagation of negative constant information, this algorithm is the heart of a very successful race analyzer for C with Posix threads :-)

573 574 575 576
(2) The Call-String Approach:

Idea:

→ Compute the set of all reachable call stacks!
→ In general, this is infinite :-(
→ Only treat stacks up to a fixed depth $d$ precisely! From longer stacks, we only keep the upper prefix of length $d$ :-(
→ Important special case: $d = 0$.
   $\implies$ Just track the current stack frame ...

... in the Example:

The conditions for $5, 7, 10$, e.g., are:

$\mathcal{R}[5] \equiv \text{combine}^2(\mathcal{R}[4], \mathcal{R}[10])$
$\mathcal{R}[7] \equiv \text{enter}^2(\mathcal{R}[1])$
$\mathcal{R}[7] \equiv \text{enter}^2(\mathcal{R}[8])$
$\mathcal{R}[9] \equiv \text{combine}^2(\mathcal{R}[8], \mathcal{R}[16])$

Warning:

The resulting super-graph contains obviously impossible paths ...
The conditions for 5, 7, 10, e.g., are:

\[ \mathcal{R}[5] \supseteq \text{combine}^2(\mathcal{R}[4], \mathcal{R}[10]) \]
\[ \mathcal{R}[7] \supseteq \text{enter}^4(\mathcal{R}[1]) \]
\[ \mathcal{R}[7] \supseteq \text{enter}^4(\mathcal{R}[8]) \]
\[ \mathcal{R}[9] \supseteq \text{combine}^2(\mathcal{R}[8], \mathcal{R}[10]) \]

Warning:
The resulting super-graph contains obviously impossible paths ...

... in the Example this is:

\[ \text{main}(t) \]
\[ t = 0; \]
\[ \text{Neg}(a_1) \]
\[ \text{Pos}(a_1) \]
\[ M[17] = 3; \]
\[ a_1 = t; \]
\[ \text{combine} \]
\[ \text{ret} = 1 - \text{ret}; \]

The conditions for 5, 7, 10, e.g., are:

\[ \mathcal{R}[5] \supseteq \text{combine}^2(\mathcal{R}[4], \mathcal{R}[10]) \]
\[ \mathcal{R}[7] \supseteq \text{enter}^4(\mathcal{R}[1]) \]
\[ \mathcal{R}[7] \supseteq \text{enter}^4(\mathcal{R}[8]) \]
\[ \mathcal{R}[9] \supseteq \text{combine}^2(\mathcal{R}[8], \mathcal{R}[10]) \]

Warning:
The resulting super-graph contains obviously impossible paths ...
... in the Example this is:

```
main()
1. t = 0;
2. Neg(t)
3. M[17] = 3;
4. combine
5. ret = 1 - ret;
6. combine
```

Note:

- In the example, we find the same results; more paths render the results less precise.
- In particular, we provide for each procedure the result just for one (possibly very boring) argument :-(
- The analysis terminates — whenever \( D \) has no infinite strictly ascending chains :-)
- The correctness is easily shown w.r.t. the operational semantics with call stacks.
- For the correctness of the functional approach, the semantics with computation forests is better suited :-(