Improvement:

→ Not all functions from $\mathbb{D}_f \to \mathbb{D}_f$ will occur \:-)

→ All occurring functions $\lambda D. \bot \neq M$ are of the form:

\[
M = \{ x \mapsto (b_x \sqcup \bigsqcup_{y \in I_x} y) \mid x \in \text{Vars} \}
\]

where:

\[
M D = \{ x \mapsto (b_x \sqcup \bigsqcup_{y \in I_x} D y) \mid x \in \text{Vars} \}
\]

für $D \neq \bot$

→ Let $\mathcal{M}$ denote the set of all these functions. Then for $M_1, M_2 \in \mathcal{M}$ ($M_1 \neq \lambda D. \bot \neq M_2$):

\[
(M_1 \sqcup M_2) x = (M_1 x) \sqcup (M_2 x)
\]

→ For $k = \# \text{Vars}$, $\mathcal{M}$ has height $\mathcal{O}(k^2)$ \:-)
Improvement (Cont.):

→ Also, composition can be directly implemented:

\[(M_1 \circ M_2) \ x \ = \ b' \sqcup \bigcup_{y \in I'} y\]

with

\[b' = b \sqcup \bigcup_{z \in I} b_z\]

\[I' = \bigcup_{z \in I} I_z\]

where

\[M_1 \ x \ = \ b \sqcup \bigcup_{y \in I} y\]

\[M_2 \ z \ = \ b_z \sqcup \bigcup_{y \in I_z} y\]

→ The effects of assignments then are:

\[[x = e;]^# = \begin{cases} 
\text{Id}_{\text{Vars}} \oplus \{x \mapsto c\} & \text{if } e = c \in \mathbb{Z} \\
\text{Id}_{\text{Vars}} \oplus \{x \mapsto y\} & \text{if } e = y \in \text{Vars} \\
\text{Id}_{\text{Vars}} \oplus \{x \mapsto \top\} & \text{otherwise}
\end{cases}\]
... in the Example:

\[
\begin{align*}
[t = 0;] & = \{ a_1 \mapsto a_1, \text{ret} \mapsto \text{ret}, t \mapsto 0 \} \\
[a_1 = t;] & = \{ a_1 \mapsto t, \text{ret} \mapsto \text{ret}, t \mapsto t \}
\end{align*}
\]

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

\[
\begin{align*}
[k] & = H ([f]) \\
H (M) & = \text{Id}|_{\text{Locals} \oplus (M \circ \text{enter}^\#)}|_{\text{Globals}} \\
\text{enter}^\# x & = \begin{cases} 
  x & \text{if } x \in \text{Globals} \\
  0 & \text{otherwise}
\end{cases}
\end{align*}
\]
... in the Example:

$$\begin{align*}
\text{If} & \quad [\text{work}]^\# = \{a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t\} \\
\text{then} & \quad H[\text{work}]^\# = \text{id}_{\{t\}} \oplus \{a_1 \mapsto a_1, \text{ret} \mapsto a_1\} \\
& \quad = \{a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t\}
\end{align*}$$

Now we can perform fixpoint iteration  :-)

563
\[ 7 \quad \text{work}() \]

\[ 8 \quad \text{Neg}(a_1) \quad \text{Pos}(a_1) \quad \text{work}(); \]

\[ 9 \quad \text{ret} = a_1; \]

\[ 10 \]

\[ [(8, \ldots, 9)]^\# \circ [8]^\# = \{a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t\} \circ \{a_1 \mapsto a_1, \text{ret} \mapsto \text{ret}, t \mapsto t\} \]

\[ = \{a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t\} \]
work()();

Neg(a_1) → Pos(a_1)

work(); ret = a_1;

\[
[(8, \ldots, 9)]^\# \circ [8]^\# = \{a_1 \mapsto a_1, \text{ret} \mapsto \text{ret}, t \mapsto t\} \circ \\
\{a_1 \mapsto a_1, \text{ret} \mapsto \text{ret}, t \mapsto t\} \\
= \{a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t\}
\]
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:

\[
\begin{align*}
\mathcal{R}[\text{main}] & \sqsubseteq \text{enter}^\# d_0 \\
\mathcal{R}[f] & \sqsubseteq \text{enter}^\# (\mathcal{R}[u]) \quad k = (u, f();, _) \quad \text{call} \\
\mathcal{R}[v] & \sqsubseteq \mathcal{R}[f] \quad v \quad \text{entry point of } f \\
\mathcal{R}[v] & \sqsubseteq \langle k \rangle^\# (\mathcal{R}[u]) \quad k = (u, _, v) \quad \text{edge}
\end{align*}
\]
... in the Example:

```
main()

0
  \[ t = 0; \]

1
  \[ \text{Neg}(t) \] \quad \text{Pos}(t) \]

2
  \[ M[17] = 3; \]

3
  \[ a_1 = t; \]

4
  \[ \text{work}(); \]

5
  \[ \text{ret} = 1 - \text{ret}; \]

6
```

<table>
<thead>
<tr>
<th></th>
<th>(a_1)</th>
<th>\text{ret}</th>
<th>(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\top)</td>
<td>(\top)</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>(\top)</td>
<td>(\top)</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>(\top)</td>
<td>(\top)</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>(\top)</td>
<td>(\top)</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>(\top)</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>(\top)</td>
<td>0</td>
</tr>
</tbody>
</table>
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 \( M \subseteq D \rightarrow D \) must be finite;
  
  (2) The functions \( M \in M \) must be efficiently implementable :-)

- The second condition can, sometimes, be abandoned ...
Observation: Sharir/Pnueli, Cousot

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

\[
\begin{align*}
[v, a] & \supseteq a & v & \text{entry point} \\
[v, a] & \supseteq \text{combine} ([u, a], [f, \text{enter} [u, a]]) \\
(u, f();, v) & \text{call} \\
[v, a] & \supseteq [lab] [u, a] & k = (u, lab, v) & \text{edge} \\
[f, a] & \supseteq [stop_f, a] & \text{stop}_f & \text{end point of } f \\
//& & [v, a] & = & \text{value for the argument } a.
\end{align*}
\]
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 $[\text{main()}, a_0]^\sharp \implies$ We apply our local fixpoint algorithm :-(
- The fixpoint algo provides us also with the set of actual parameters $a \in \mathcal{D}$ for which procedures are (possibly) called and all abstract values at their program points for each of these calls :-)

570
... in the Example:

Let us try a **full** constant propagation ...

![Graph Diagram]

<table>
<thead>
<tr>
<th></th>
<th>$a_1$</th>
<th>ret</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>⊥</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
| ret| T | T | 0 | 1

main()
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  :-)
(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: $d = 0$.

⇒⇒ Just track the current stack frame ...
... in the Example:

main()

0

$t = 0$;

1

Neg(t)  Pos(t)

2

$M[17] = 3$;

3

$a_1 = t$;

4

work();

5

ret = 1 - ret;

6

work();

7

Neg($a_1$)

8

Pos($a_1$)

9

work();

10

ret = $a_1$;
... in the Example:

main()

0

\( t = 0; \)

1

Neg (\( t \))

Pos (\( t \))

2

\( M[17] = 3; \)

3

4

a_1 = t;

5

\( \text{ret} = 1 - \text{ret}; \)

6

7

work ()

enter

8

9

Neg (a_1)

Pos (a_1)

10

\( \text{ret} = a_1; \)

combine

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

\[
\mathcal{R}[5] \supseteq \text{combine}^\# (\mathcal{R}[4], \mathcal{R}[10])
\]

\[
\mathcal{R}[7] \supseteq \text{enter}^\# (\mathcal{R}[4])
\]

\[
\mathcal{R}[7] \supseteq \text{enter}^\# (\mathcal{R}[8])
\]

\[
\mathcal{R}[9] \supseteq \text{combine}^\# (\mathcal{R}[8], \mathcal{R}[10])
\]

**Warning:**

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

```
main() {
    t = 0;
    Pos(t) = M[17] = 3;
    a1 = t;
    combine

    work() {
        Neg(a1) = Pos(a1) = ret = a1;
        combine
    }

    combine

    ret = 1 - ret;
}
```
... in the Example this is:

main()

0

t = 0;

1

Neg (t)

Pos (t)

2

M[17] = 3;

3

a_1 = t;

4

5

ret = 1 - ret;

6

7

enter

work ()

8

9

Pos (a_1)

Neg (a_1)

10

ret = a_1;

combine

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 :-)}
3 Exploiting Hardware Features

Question: How can we optimally use:

... Registers
... Pipelines
... Caches
... Processors ??