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 $[\mathcal{M} \subseteq \mathcal{D} \to \mathcal{D}]$ must be finite;
  2. The functions $M \in \mathcal{M}$ must be efficiently implementable :) 
- The second condition can, sometimes, be abandoned ...

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 $a \in \mathcal{D}$ for which procedures are (possibly) called and all abstract values at their program points for each of these calls :)

Observation: Sharir/Pnueli, Cousot

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

\[
\begin{align*}
[v, a]^2 & \subseteq a & \text{entry point,} \\
[v, a]^2 & \subseteq \text{combine}([u, a], [f; e; w]^2; [u, a]^2) & \text{call} \\
[v, a]^2 & \subseteq [\text{lab}]^2 [u, a]^2 & k = (u, \text{lab}, v) \text{ edge} \\
[f, a]^2 & \subseteq [\text{stop} f, a]^2 & \text{stop} f \text{ end point of } f \\
// [v, a]^2 & \equiv \text{value for the argument } a.
\end{align*}
\]
... in the Example:

Let us try a full constant propagation ...

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: \(d = 0\).
  \[\text{Just track the current stack frame ...}\]

... in the Example:

\[
\begin{align*}
\text{main()} & \quad 0 \\
& \quad 1 \quad \text{\(t = 0;\)} \\
& \quad 2 \quad \text{\(M[17] = 3;\)} \\
& \quad 3 \quad \text{\(a_1 = t;\)} \\
& \quad 4 \quad \text{\(\text{ret} = 1 - \text{ret};\)} \\
& \quad 5 \quad \text{\(\text{combine}\)} \\
& \quad 6 \quad \text{\(
\begin{align*}
\text{Neg (a_1)} & \quad \text{\(0\)} \\
& \quad 1 \quad \text{\(\text{Pos (a_1)}\)} \\
& \quad 2 \quad \text{\(\text{Pos (a_1)}\)} \\
& \quad 3 \quad \text{\(\text{Neg (a_1)}\)} \\
& \quad 4 \quad \text{\(\text{combine}\)} \\
& \quad 5 \quad \text{\(\text{ret} = 1 - \text{ret;}\)} \\
\end{align*}\)
\end{align*}
\]

... in the Example:

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

\[
\begin{align*}
\mathcal{R}[5] & \sqsupseteq \text{combine}^\phi (\mathcal{R}[4], \mathcal{R}[10]) \\
\mathcal{R}[7] & \sqsupseteq \text{enter}^\phi (\mathcal{R}[4]) \\
\mathcal{R}[7] & \sqsupseteq \text{enter}^\phi (\mathcal{R}[8]) \\
\mathcal{R}[9] & \sqsupseteq \text{combine}^\phi (\mathcal{R}[8], \mathcal{R}[10]) \\
\end{align*}
\]

Warning:

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

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

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

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

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

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

Warning:

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

... in the Example:

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

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

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

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

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

Warning:

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

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 ???

3.1 Registers

Example:

```plaintext
read();
x = M[A];
y = x + 1;
if (y) {
    z = x \cdot x;
    M[A] = z;
} else {
    t = -y \cdot y;
    M[A] = t;
}
```

The program uses 5 variables ...

Problem:

What if the program uses more variables than there are registers :-(

Idea:

Use one register for several variables :-) 
In the example, e.g., one for \( x, t, z \) ...
3.1 Registers

Example:

```
read();

3 = M[A];
y = y + 1;
if (y) {
    x = x - x;
    M[A] = x;
} else {
    y = y - y;
    M[A] = y;
}
```

Warning:

This is only possible if the live ranges do not overlap.

The (true) live range of \( x \) is defined by:

\[
\mathcal{L}[x] = \{ u \mid x \in \mathcal{L}[u] \}
\]

... in the Example:

<table>
<thead>
<tr>
<th>( \mathcal{L} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 ( \emptyset )</td>
</tr>
<tr>
<td>7 {A, z}</td>
</tr>
<tr>
<td>6 {A, x}</td>
</tr>
<tr>
<td>5 {A, t}</td>
</tr>
<tr>
<td>4 {A, y}</td>
</tr>
<tr>
<td>3 {A, x, y}</td>
</tr>
<tr>
<td>2 {A, x}</td>
</tr>
<tr>
<td>1 {A}</td>
</tr>
<tr>
<td>0 {A}</td>
</tr>
</tbody>
</table>
In order to determine sets of compatible variables, we construct the \textbf{Interference Graph} \( I = (\text{Vars}, \mathcal{E}_I) \), where:

\[
\mathcal{E}_I = \{ \{x, y\} \mid x \neq y, \mathcal{L}_x \cap \mathcal{L}_y \neq \emptyset \}
\]

\( \mathcal{E}_I \) has an edge for \( x \neq y \) iff \( x, y \) are jointly live at some program point :-)

... in the Example:

Variables which are not connected with an edge can be assigned to the same register :-)

Color = Register
Abstract Problem:

Given: Undirected Graph \((V, E)\).

Wanted: Minimal coloring, i.e., mapping \(c : V \rightarrow \mathbb{N}\) such that:

1. \(c(u) \neq c(v)\) for \(\{u, v\} \in E\);
2. \(|\{c(u) | u \in V\}|\) is minimal.

- In the example, 3 colors suffice \(\Rightarrow\) But:
- In general, the minimal coloring is not unique \(\Rightarrow\)
- It is NP-complete to determine whether there is a coloring with at most \(k\) colors \(\Rightarrow\)

We must rely on heuristics or special cases \(\Rightarrow\)
Greedy Heuristics:

- Start somewhere with color 1:
- Next choose the smallest color which is different from the colors of all already colored neighbors:
- If a node is colored, color all neighbors which not yet have colors:
- Deal with one component after the other ...

... more concretely:

\[
\begin{align*}
& \text{forall } (v \in V) \quad c[v] = 0; \\
& \text{forall } (v \in V) \quad \text{color} (v); \\
& \text{void color} (v) \{ \\
& \quad \text{if } (c[v] \neq 0) \text{ return; } \\
& \quad \text{neighbors} = \{ u \in V \mid \{u, v\} \in E \}; \\
& \quad c[v] = \prod\{ k > 0 \mid \forall u \in \text{neighbors} : k \neq c(u) \}; \\
& \quad \text{forall } (u \in \text{neighbors}) \\
& \quad \quad \text{if } (c(u) == 0) \quad \text{color} (u); \\
& \} \\
\end{align*}
\]

The new color can be easily determined once the neighbors are sorted according to their colors … :-(

Discussion:

→ Essentially, this is a Pre-order DFS :-(
→ In theory, the result may arbitrarily far from the optimum :-(
→ … in practice, it may not be as bad :-(
→ … Anecdote: different variants have been patented !!!

→ Essentially, this is a Pre-order DFS :-(
→ In theory, the result may arbitrarily far from the optimum :-(
→ … in practice, it may not be as bad :-(
→ … Anecdote: different variants have been patented !!!

The algorithm works the better the smaller life ranges are …

Idea: Life Range Splitting