2 Inter-procedural analysis

We extend our mini-programming language by procedures without parameters and procedure calls.

For that, we introduce a new statement:

\[ f(); \]

Every procedure \( f \) has a definition:

\[
f() \{ \text{stmt}^* \}
\]

Additionally, we distinguish between global and local variables.

Program execution starts with the call of a procedure \( \text{main}() \).

Dedicated global/local variables \( a_i, b_i, \text{ret} \) can be used for parameter transfer.
Example:

```c
int a, ret;
main () {
  a = 3;
  f();
  M[17] = ret;
  ret = 0;
}

f () {
  int b;
  if (a ≤ 1) {ret = 1; goto exit;}
  b = a;
  a = b − 1;
  f();
  ret = b · ret;
}

exit :
}
```

Such programs can be represented by a set of CFGs: one for each procedure ...
... in the example:

```
main()

0
  a = 3;

1
  f();

2
  M[17] = ret;

3
  ret = 0;

4

5
  f()

6
  Neg (a ≤ 1)

7
  b = a;

8
  Pos (a ≤ 1)

9
  a = b - 1;

10

11
  ret = b * ret;

12
  ret = 1;
```

4
In order to optimize such programs, we require an extended operational semantics.

Program executions are no longer *paths*, but *forests*:
... in the example:
Configurations:

The operational semantics is defined by a one-step computation relation between configurations.

\[
\text{configuration} \quad = \quad \text{stack} \times \text{globals} \times \text{store} \\
\text{globals} \quad = \quad \text{Glob} \to \mathbb{Z} \\
\text{store} \quad = \quad \mathbb{N} \to \mathbb{Z} \\
\text{stack} \quad = \quad \text{frame} \cdot \text{frame}^* \\
\text{frame} \quad = \quad \text{point} \times \text{locals} \\
\text{locals} \quad = \quad \text{Loc} \to \mathbb{Z}
\]

\text{Glob} and \text{Loc} the sets of global and local variables. \\
\text{point} the set of program points. \\

The call stack, \text{stack} contains one \text{frame} for each procedure called, but not yet left.

A \text{frame} keeps the local state of computation inside a procedure call.
Computation steps refer to the current call.

The novel kinds of steps:

\[ \text{call } k = (u, f(), v) : \]
\[ (\sigma \cdot (u, \rho_{Loc}), \rho_{Glob}, \mu) \vdash (\sigma \cdot (v, \rho_{Loc}) \cdot (u_f, \rho_f), \rho_{Glob}, \mu) \]
\[ u_f \text{ entry point of } f \]

\[ \text{return from a call :} \]
\[ (\sigma \cdot (v, \rho_{Loc}) \cdot (r_f, \_), \rho_{Glob}, \mu) \vdash (\sigma \cdot (v, \rho_{Loc}), \rho_{Glob}, \mu) \]
\[ r_f \text{ exit point of } f \]

Local variables are initialized to 0 at procedure entry, i.e., \( \rho_f = \{ x \mapsto 0 \mid x \in Loc \} \).
Computation is a sequence of computation steps.

Two new labels,

- $\langle f \rangle$ for a call to procedure $f$ and
- $\langle /f \rangle$ for the exit from this procedure.

A computation $\pi$ that leads from a configuration $(\sigma \cdot (u, \rho_{Loc}), \rho_{Glob}, \mu)$ to a configuration $(\sigma \cdot (v, \rho'_{Loc}), \rho'_{Glob}, \mu')$ is called same-level.

Same-level computations

- contain exits for all procedures called,
- leave the stack at the height it had upon procedure entry.
The function \([.]\) is extended to nested paths \(w:\)

\[[w]: stack \times globals \times store \to stack \times globals \times store\]

For a call \(k = (u, f(), v)\)

\[
\left[\pi_1 \langle f\rangle \pi_2 \langle /f\rangle\right] = H([\pi_2]) \circ [\pi_1]
\]

the state transformation caused by a procedure’s body is translated into one for the caller of that procedure by the operator \(H(\cdots)\):

\[
H(g) (\rho_{Loc}, \rho_{Glob}, \mu) = \textbf{let} \ (\rho'_{Loc}, \rho'_{Glob}, \mu') = g (\emptyset, \rho_{Glob}, \mu) \textbf{ in} \ (\rho_{Loc}, \rho'_{Glob}, \mu')
\]

In general, \([w]\) is only partially defined.
Alternatively:

• determine the initial values for the locals:

\[
\text{enter } \rho = \{ x \mapsto 0 \mid x \in \text{Loc} \} \uplus (\rho|_{\text{Glob}})
\]

• ... combine the new values for the globals with the old values for the locals of the caller:

\[
\text{combine } (\rho_1, \rho_2) = (\rho_1|_{\text{Loc}}) \uplus (\rho_2|_{\text{Glob}})
\]

• ... evaluate the same-level computation inbetween:

\[
[k \langle w \rangle] (\rho, \mu) = \text{let } (\rho_1, \mu_1) = \llbracket w \rrbracket \text{ (enter } \rho, \mu) \text{ in } \text{combine } (\rho, \rho_1), \mu_1)
\]
The call stack explicitly implements the DFS traversal through the computation forest

... in the example:
The call stack explicitly implements the DFS traversal through the computation forest

... in the example:

<table>
<thead>
<tr>
<th>5</th>
<th>$b \mapsto 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
The call stack explicitly implements the DFS traversal through the computation forest

... in the example:

```
<table>
<thead>
<tr>
<th>7</th>
<th>b ↔ 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
```
The call stack explicitly implements the DFS traversal through the computation forest

... in the example:

```
5 | b ↦ 0
9 | b ↦ 3
2 |
```
The call stack explicitly implements the DFS traversal through the computation forest

... in the example:
The call stack explicitly implements the DFS traversal through the computation forest

... in the example:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>b \rightarrow 0</td>
</tr>
<tr>
<td>9</td>
<td>b \rightarrow 2</td>
</tr>
<tr>
<td>9</td>
<td>b \rightarrow 3</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
The call stack explicitly implements the DFS traversal through the computation forest

... in the example:

\[
\begin{array}{|c|c|}
\hline
11 & b \leftrightarrow 0 \\
\hline
9 & b \leftrightarrow 2 \\
\hline
9 & b \leftrightarrow 3 \\
\hline
2 & \\
\hline
\end{array}
\]
The call stack explicitly implements the DFS traversal through the computation forest

... in the example:

<table>
<thead>
<tr>
<th></th>
<th>b ↦ 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>b ↦ 3</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
The call stack explicitly implements the DFS traversal through the computation forest

... in the example:
The call stack explicitly implements the DFS traversal through the computation forest

... in the example:

\[
\begin{array}{c|c}
9 & b \mapsto 3 \\
2 & \\
\end{array}
\]
The call stack explicitly implements the DFS traversal through the computation forest

... in the example:
The call stack explicitly implements the DFS traversal through the computation forest

... in the example:
The factorial program and one of its computations:

main()

0
  a = 3;
  f();

1
  f();

2
  M[17] = ret;

3
  ret = 0;

4

f()

5
  Neg (a \leq 1)

6
  b = a;

7
  Pos (a \leq 1)

8
  a = b - 1;

9
  f();

10
  ret = 1;

11
  ret = b * ret;

\langle \text{main} \rangle 0, 1 \langle f \rangle 5, 6, 7
\langle f \rangle 5, 6, 7, 8
\langle f \rangle 5, 6, 7, 8
\langle f \rangle 5, 10, 11 \langle /f \rangle
\langle f \rangle 9, 11 \langle /f \rangle
\langle f \rangle 9, 11 \langle /f \rangle
\langle f \rangle 9, 11 \langle /f \rangle
2, 3, 4 \langle /main \rangle
This operational semantics is quite realistic

Costs for a Procedure Call:

**Before entering the body:**  
- Creating a stack frame;
- assigning of the parameters;
- Saving the registers;
- Saving the return address;
- Jump to the body.

**At procedure exit:**  
- Freeing the stack frame.
- Restoring the registers.
- Passing of the result.
- Return behind the call.

⇒ ... quite expensive !!!
2.1 Inlining

Copy the procedure body at every call site

Example:

```c
abs () {
    a2 = -a1;
    max ();
}
```

```c
max () {
    if (a1 < a2) {
        ret = a2;
        goto _exit;
    }
    ret = a1;
}
```

```c
_exit :
}
```
... yields:

```c
abs () {
    a_2 = -a_1;

    if (a_1 < a_2) {
        ret = a_2;  goto _exit;
    }
    ret = a_1;

    _exit:
}
```
Problems:

- The copied block may modify the locals of the calling procedure
- More general: Multiple use of local variable names may lead to errors.
- Multiple calls of a procedure may lead to code duplication
- How can we handle recursion ???
Detection of Recursion:

We construct the call-graph of the program.

In the examples:
Call-Graph:

- The nodes are the procedures.
- An edge connects $g$ with $h$, whenever the body of $g$ contains a call of $h$.

Strategies for Inlining:

- Copy only leaf-procedures, i.e., procedures without further calls.
- Copy all non-recursive procedures.

... here, we consider just leaf-procedures.
Transformation PI:

\[ f() ; \]

\[ u \]

\[ v \]

\[ \text{copy of } f \]

\[ x_f = 0; \quad (x \in \text{Locals}) \]

\[ u \]

\[ v \]
Note:

- The Nop-edge can be eliminated if the stop-node of $f$ has no out-going edges ...
- The $x_f$ are the copies of the locals of the procedure $f$.
- According to our semantics of procedure calls, these must be initialized with 0
2.2 Elimination of Tail Recursion

\[
\begin{align*}
f() & \{ \quad \text{int } b; \\
& \quad \text{if } (a_2 \leq 1) \{ \text{ ret } = a_1; \quad \text{goto } _\text{exit}; \} \\
& \quad b = a_1 \cdot a_2; \\
& \quad a_2 = a_2 - 1; \\
& \quad a_1 = b; \\
& \quad f(); \\
& \quad _\text{exit} : \\
& \}
\end{align*}
\]

After the procedure call, nothing in the body remains to be done.

We may directly jump to the beginning after having reset the locals to 0.
... this yields in the example:

```c
f () { int b;
    _f : if (a_2 \leq 1) { ret = a_1; goto _exit; }
        b = a_1 \cdot a_2;
        a_2 = a_2 - 1;
        a_1 = b;
        b = 0; goto _f;
    _exit :
}

// It works, since we have ruled out references to variables!
```
Transformation LC:

\[
\begin{align*}
  f() : & \\
  u \quad f(); \\
  v \\
\end{align*}
\]

\[
\begin{align*}
  f() : & \\
  x = 0; \quad (x \in \text{Locals}) \\
  u \quad v \\
\end{align*}
\]
Warning:

→ This optimization is crucial for programming languages without iteration constructs.
→ No duplication of code,
→ No variable renaming is necessary,
→ The optimization may also be profitable for non-recursive tail calls,
→ The generated code contains jumps from the body of one procedure into the body of another.
2.3 Interprocedural Analysis

So far, we can analyze each procedure separately.

→ The costs are moderate
→ The methods also work in presence of separate compilation
→ At procedure calls, we must assume the worst case
→ Constant propagation only works for local constants

Question:

How can recursive programs be analyzed?
Example: Constant Propagation

```c
main() { int t;
    t = 0;
    if (t) M[17] = 3;
    a1 = t;
    work();
    ret = 1 - ret;
}

work() { if (a1) work();
    ret = a1;
    }
```
Example: Constant Propagation

main()

0

\[ t = 0; \]

1

\[ \text{Neg}(t) \]

\[ \text{Pos}(t) \]

2

\[ M[17] = 3; \]

3

\[ a_1 = t; \]

4

work();

5

6

ret = 1 - ret;

7

work();

8

\[ \text{Neg}(a_1) \]

\[ \text{Pos}(a_1) \]

9

10

ret = a_1;
Example: Constant Propagation

main()

0

1

2

3

4

5

6

t = 0;

a1 = 0;

work0();

ret = 1;

work0()

7

8

9

10

ret = 0;
(1) Functional Approach:

Let $D$ denote a complete lattice of (abstract) states.

Idea:

Represent the effect of $f()$ by a function:

$$[f]^\sharp : D \rightarrow D$$
In order to determine the effect of a call edge \[ k = (u, f();, v) \] we require abstract functions:

\[
\begin{align*}
\text{enter} & : D \rightarrow D \\
\text{combine} & : D^2 \rightarrow D
\end{align*}
\]

Then we define:

\[
[[k]]^\# D = \text{combine}^\# (D, [[[f]]^\# (\text{enter}^\# D))
\]
... for Constant Propagation:

\[
\mathcal{D} = (\text{Vars} \rightarrow \mathbb{Z}_\uparrow)_\perp
\]

\[
\text{enter}^\# D = \begin{cases} 
\bot & \text{if } D = \bot \\
D|_{\text{Glob}} \uplus \{x \mapsto 0 \mid x \in \text{Loc}\} & \text{otherwise}
\end{cases}
\]

\[
\text{combine}^\# (D_1, D_2) = \begin{cases} 
\bot & \text{if } D_1 = \bot \lor D_2 = \bot \\
D_1|_{\text{Loc}} \uplus D_2|_{\text{Glob}} & \text{otherwise}
\end{cases}
\]
The effects \([f]\) then can be determined by a system of constraints over the complete lattice \(\mathbb{D} \rightarrow \mathbb{D}\):

\[
\begin{align*}
[v]^\# & \supseteq \text{Id} & v & \text{entry point} \\
[v]^\# & \supseteq [k]^\# \circ [u]^\# & k = (u, _, v) & \text{edge} \\
[f]^\# & \supseteq [stop_f]^\# & stop_f & \text{end point of } f
\end{align*}
\]

\([v]^\# : \mathbb{D} \rightarrow \mathbb{D}\) describes the effect of all prefixes of computations \(w\) of a procedure that lead from the entry point to \(v\). (called \(v\)-reaching computations in the book.)
Problems:

- How can we represent functions $f : \mathbb{D} \to \mathbb{D}$?
- If $\#\mathbb{D} = \infty$, then $\mathbb{D} \to \mathbb{D}$ has infinite strictly increasing chains

Simplification: Copy-Constants

$\to$ Conditions are interpreted as $;$
$\to$ Only assignments $x = e$; with $e \in Vars \cup \mathbb{Z}$ are treated exactly
Observation:

→ The effects of assignments are:

\[
[x = e;] D = \begin{cases} 
D \oplus \{ x \mapsto c \} & \text{if } e = c \in \mathbb{Z} \\
D \oplus \{ x \mapsto (D y) \} & \text{if } e = y \in \text{Vars} \\
D \oplus \{ x \mapsto \top \} & \text{otherwise}
\end{cases}
\]

→ Let \( \mathbb{V} \) denote the (finite) set of constant right-hand sides. Then variables may only take values from \( \mathbb{V}^\top \).

→ The occurring effects can be taken from

\[
\mathbb{D}_f \to \mathbb{D}_f \quad \text{ with } \quad \mathbb{D}_f = (\text{Vars} \to \mathbb{V}^\top)_\perp
\]

→ The complete lattice is huge, but finite!
Improvement:

→ Not all functions from $\mathbb{D}_f \rightarrow \mathbb{D}_f$ will occur

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

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

where:

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

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

→ Also, composition can be directly implemented:

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

→ The effects of assignments then are:

\[
[x = e;] = \begin{cases} 
\text{Id}_{V} \oplus \{x \mapsto c\} & \text{if } e = c \in \mathbb{Z} \\
\text{Id}_{V} \oplus \{x \mapsto y\} & \text{if } e = y \in V \\
\text{Id}_{V} \oplus \{x \mapsto \top\} & \text{otherwise}
\end{cases}
\]
... 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 \( f \):

\[
[k]' = H([f]')
\]

where:

\[
H(M) = \text{Id}|_{\text{Loc}} \cup (M \circ \text{enter}'')|_{\text{Glob}}
\]

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

\[
\text{If } \lbrack \text{work} \rbrack^\# = \{a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t\}
\]

\[
\text{then } H \lbrack \text{work} \rbrack^\# = \text{ld}_\{t\} \oplus \{a_1 \mapsto a_1, \text{ret} \mapsto a_1\}
\]

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

Now we can perform fixpoint iteration
\[
[(8, \ldots, 9)]^\# \circ [8]^\#
\] = \{a_1 \leftrightarrow a_1, \text{ret} \leftrightarrow a_1, t \leftrightarrow t\} \circ \\
\{a_1 \leftrightarrow a_1, \text{ret} \leftrightarrow \text{ret}, t \leftrightarrow t\}
\]

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

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

```c
main()

0
  t = 0;

1
  Pos (t)

2
  Neg (t)
  M[17] = 3;

3
  a1 = t;

4
  work();

5
  ret = 1 - ret;

6
```

<table>
<thead>
<tr>
<th></th>
<th>a1</th>
<th>ret</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</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>T</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 needs only to be analyzed for these.

→ Construct a constraint system:

\[
\begin{align*}
[v, a] & \subseteq a & \text{v entry point} \\
[v, a] & \subseteq \text{combine}([u, a], [f, \text{enter} [u, a]]) & (u, f(); v) \text{ call} \\
[v, a] & \subseteq [l \text{ab}][u, a] & k = (u, l \text{ab}, v) \text{ edge} \\
[f, a] & \subseteq [s \text{top}\_f, a] & s \text{top}_f \text{ end point of } f \\
\text{// } [v, a] & = = \text{ value for the argument } a
\end{align*}
\]
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 that occur, i.e., which are necessary to determine the value \([\text{main()}, a_0]^\#\).
  \[\implies\] We apply our local fixpoint algorithm.
- The fixpoint algo provides us also with the set of actual parameters \(a \in \mathbb{D}\) 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 ...

```
main()
0
 t = 0;

1
 Pos (t)

2
 M[17] = 3;

3
 a1 = t;

4
 work();

5
 ret = 1 - ret;

6

7
 Neg (a1)

8
 Pos (a1)

work();

9
 ret = a1;

10
```

<table>
<thead>
<tr>
<th></th>
<th>a1</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>T</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>main()</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

...
Discussion:

- In the Example, the analysis terminates quickly.
- If $\mathcal{D}$ 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:

```c
main()
  t = 0;
  M[17] = 3;
  a1 = t;
  work();
  ret = 1 - ret;

work()
  Neg(a1)
  Pos(a1)
  ret = a1;
```
... in the Example:

```
main()

    t = 0;

    Pos(t)  Neg(t)

    M[17] = 3;

    a1 = t;

    combine

    ret = a1;

    combine

    ret = 1 - ret;

work()

    Neg(a1)

    Pos(a1)

    combine
```

```
ret = a1;
```

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

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

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

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

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

**Warning:**

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

```
main()
t = 0;
Pos(t)
Neg(t)

M[17] = 3;
a1 = t;

ret = a1;

ret = 1 - ret;
```

```
work()

work()

Neg(a1)

Pos(a1)

combine

combine

enter

enter
```
... in the Example this is:

main()

t = 0;

Neg (t)

Pos (t)

M[17] = 3;

a1 = t;

ret = a1;

combine

work ()

enter

Pos (a1)

Neg (a1)

enter

combine

ret = 1 - ret;
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 $\mathcal{D}$ 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