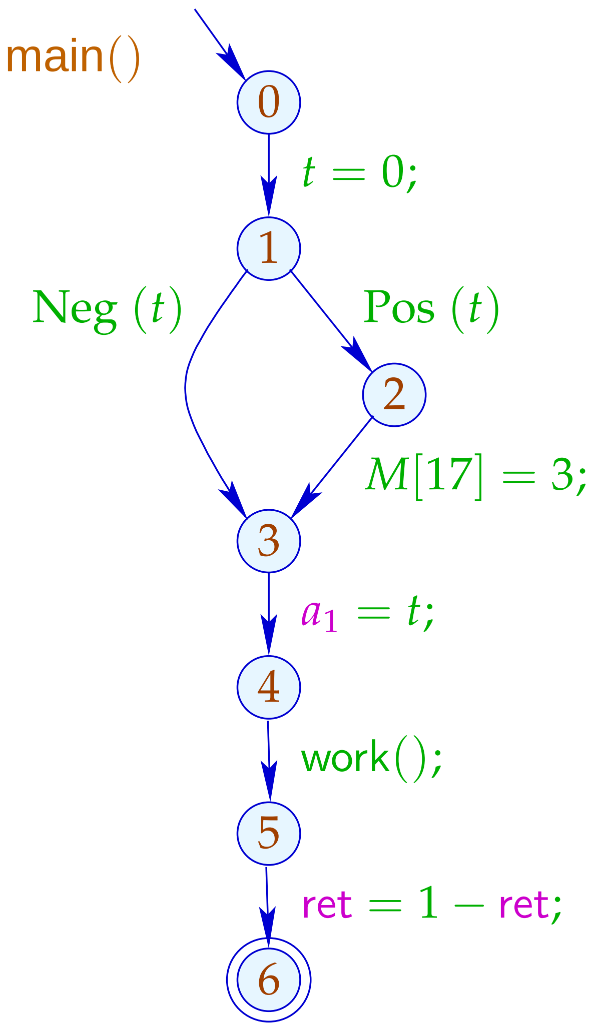
If we know the effects of procedure calls, we can put up a constraint system for determining the abstract state when reaching a program point:

$$
\begin{array}{llll}
\mathcal{R}[\mathsf{main}] & \sqsupseteq & \mathsf{enter}^\sharp\; d_0 & \\
\mathcal{R}[f] & \sqsupseteq & \mathsf{enter}^\sharp\; (\mathcal{R}[u]) & k = (u, f\,();, \_) \quad \text{call} \\
\mathcal{R}[v] & \sqsupseteq & \mathcal{R}[f] & v \quad \text{entry point of} \quad f \\
\mathcal{R}[v] & \sqsupseteq & [\![k]\!]^\sharp\; (\mathcal{R}[u]) & k = (u, \_, v) \quad \text{edge}
\end{array}
$$

... in the Example:



0	$\{a_1 \mapsto \top, \text{ret} \mapsto \top, t \mapsto 0\}$
1	$\{a_1 \mapsto \top, \text{ret} \mapsto \top, t \mapsto 0\}$
2	$\{a_1 \mapsto \top, \text{ret} \mapsto \top, t \mapsto 0\}$
3	$\{a_1 \mapsto \top, \text{ret} \mapsto \top, t \mapsto 0\}$
4	$\{a_1 \mapsto 0, \text{ret} \mapsto \top, t \mapsto 0\}$
5	$\{a_1 \mapsto 0, \text{ret} \mapsto 0, t \mapsto 0\}$
6	$\{a_1 \mapsto 0, \text{ret} \mapsto \top, t \mapsto 0\}$

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 $\mathbb{M} \subseteq \mathbb{D} \to \mathbb{D}$ must be finite;

  (2) The functions $M \in \mathbb{M}$ must be efficiently implementable :-)

- The second condition can, sometimes, be abandoned ...

Observation: Sharir/Pnueli, Cousot

$\longrightarrow$ Often, procedures are only called for few distinct abstract arguments.

$\longrightarrow$ Each procedure need only to be analyzed for these :-)

$\longrightarrow$ Put up a constraint system:

$$
\begin{array}{lll}
[\![v, a]\!]^\sharp & \sqsupseteq & a \qquad\qquad\qquad v \quad \text{entry point} \\
[\![v, a]\!]^\sharp & \sqsupseteq & \mathsf{combine}^\sharp\,([\![u, a]\!], [\![f, \mathsf{enter}^\sharp\,[\![u, a]\!]^\sharp]\!]^\sharp) \\
 & & \qquad\qquad (u, f();, v) \quad \text{call} \\
[\![v, a]\!]^\sharp & \sqsupseteq & [\![lab]\!]^\sharp\,[\![u, a]\!]^\sharp \quad k = (u, lab, v) \quad \text{edge} \\
[\![f, a]\!]^\sharp & \sqsupseteq & [\![stop_f, a]\!]^\sharp \qquad stop_f \quad \text{end point of} \quad f
\end{array}
$$

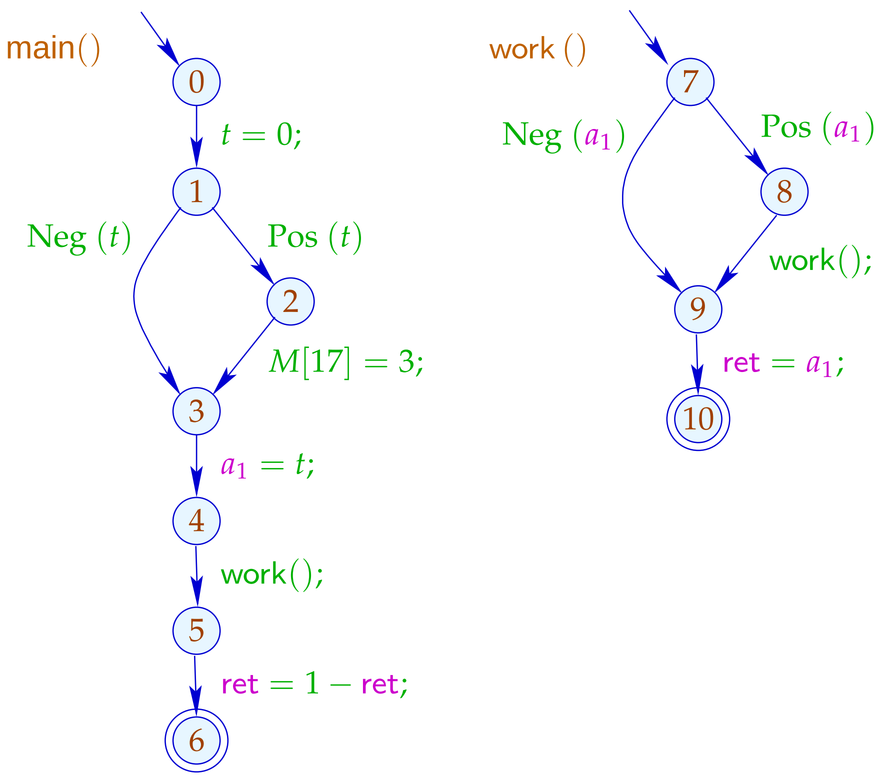
// $[\![v, a]\!]^\sharp$ $==$ value for the argument a .

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 $[\![\text{main}(), a_0]\!]^\sharp$ $\Longrightarrow$ 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 ...



	a_1	ret	a_1	ret
0	$\top$	$\top$	$\top$	$\top$
1	$\top$	$\top$	$\top$	$\top$
2	$\top$	$\top$	$\bot$	
3	$\top$	$\top$	$\top$	$\top$
4	$\top$	$\top$	0	$\top$
7	0	$\top$	0	$\top$
8	0	$\top$	$\bot$	
9	0	$\top$	0	$\top$
10	0	$\top$	0	0
5	$\top$	$\top$	0	0
main()	$\top$	$\top$	0	1

Discussion:

- In the Example, the analysis terminates quickly :-)
- If $\mathbb{D}$ 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!

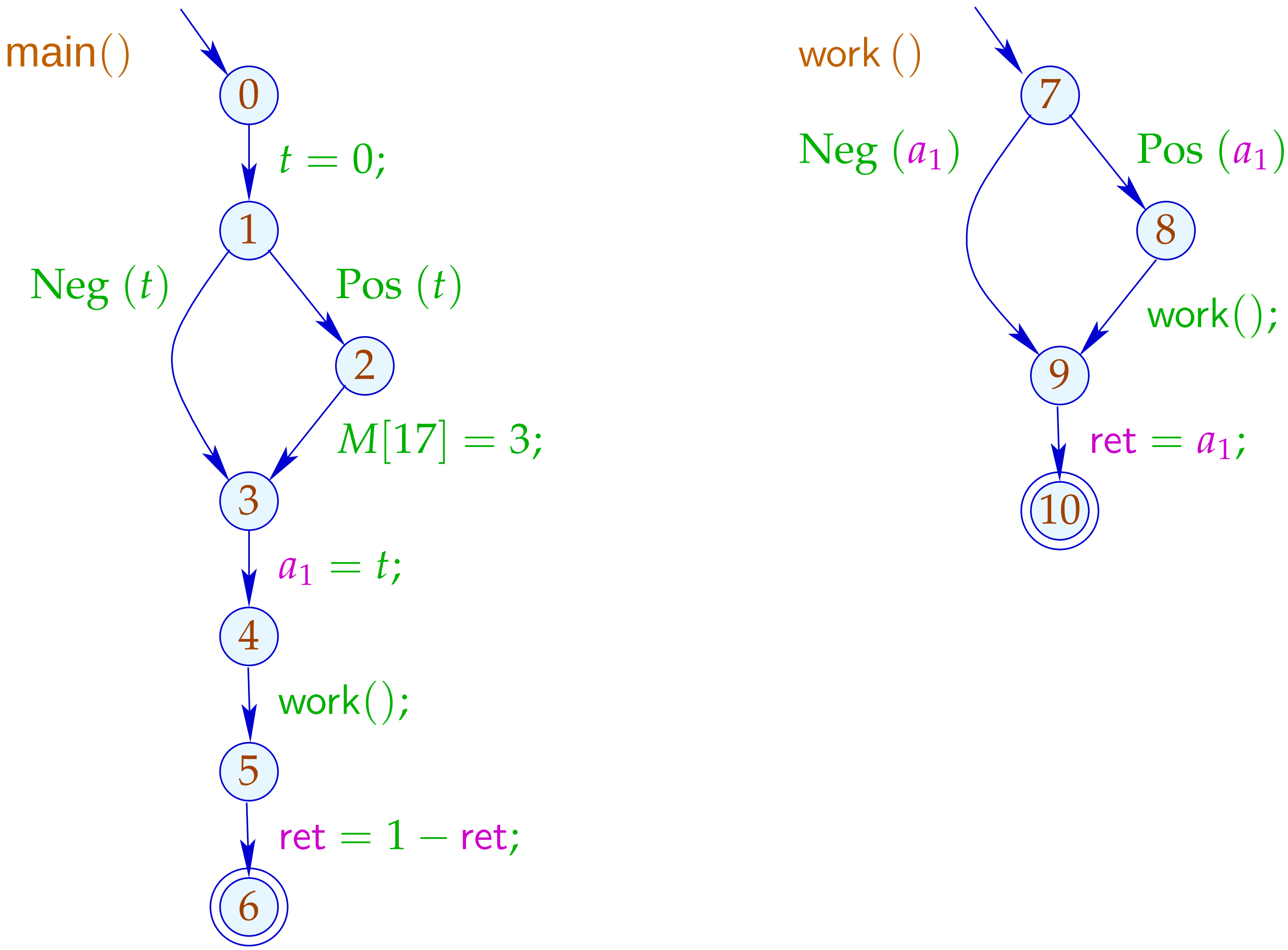
→ In general, this is infinite :-(

→ Only treat stacks up to a fixed depth d precisely! From longer stacks, we only keep the upper prefix of length d :-)

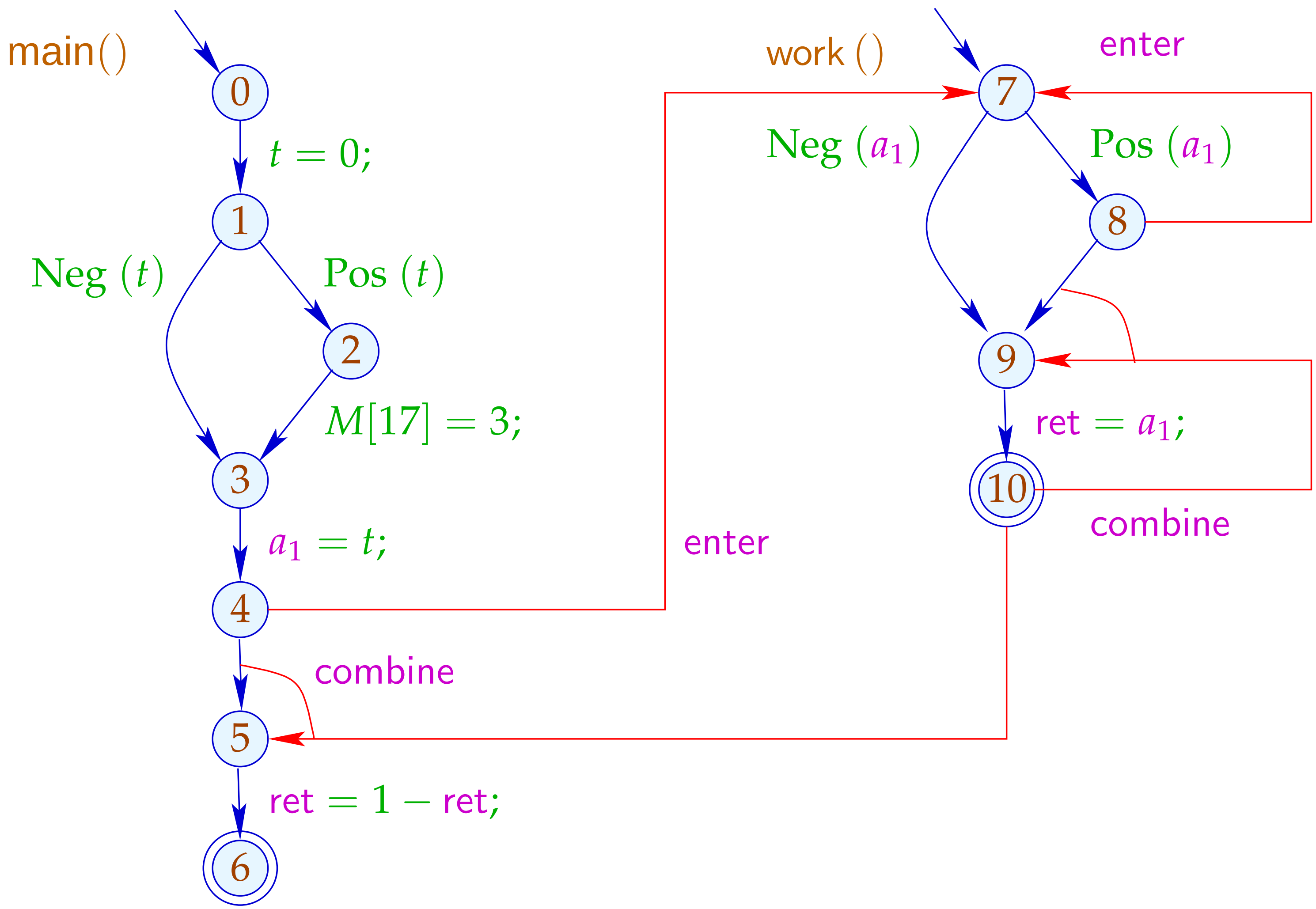
→ Important special case: $d = 0$.

$\Longrightarrow$ Just track the current stack frame ...

... in the Example:



... in the Example:



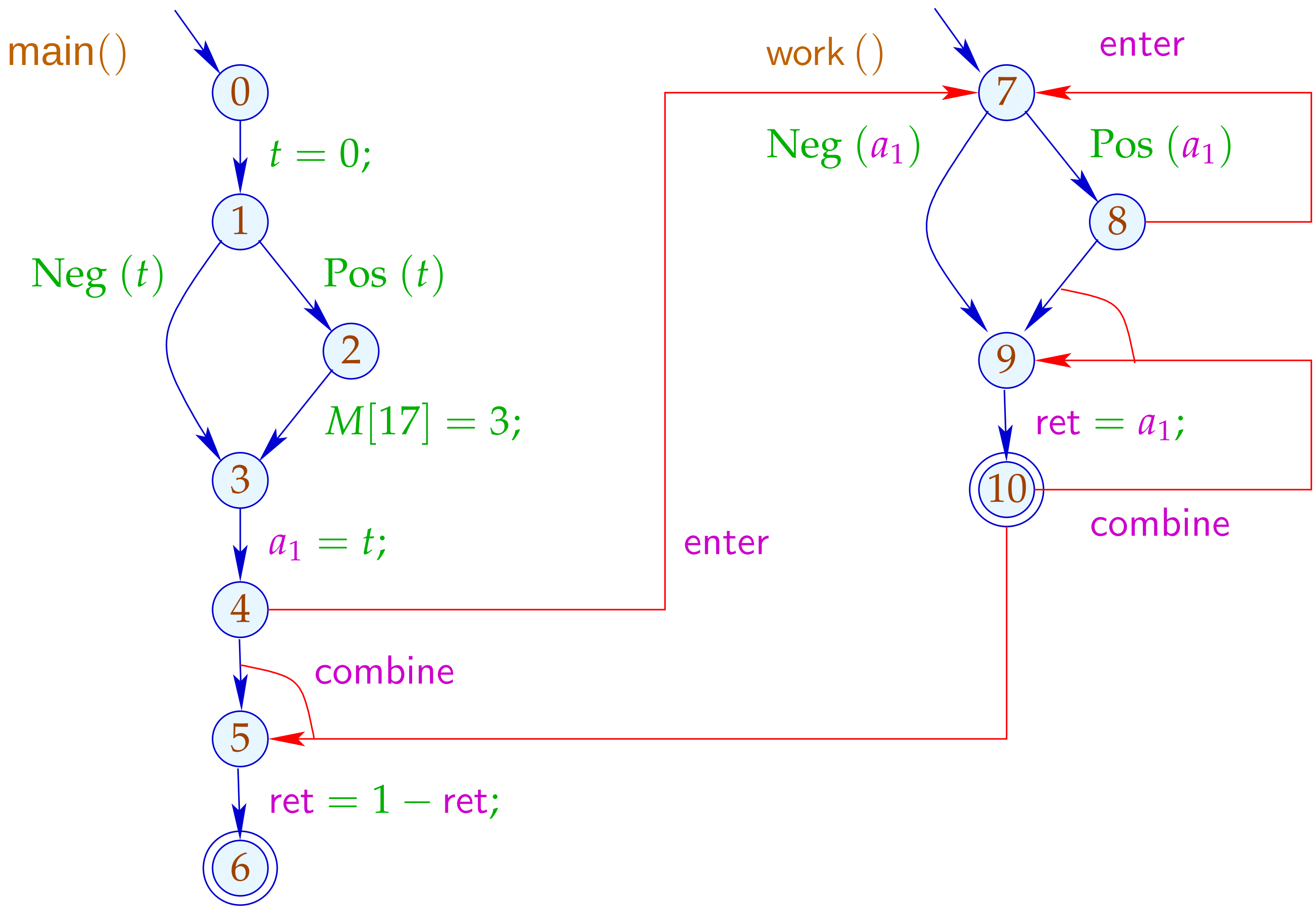
The conditions for $5, 7, 10$, e.g., are:

$$\begin{array}{lll} \mathcal{R}[5] & \sqsupseteq & \textsf{combine}^\sharp\,(\mathcal{R}[4], \mathcal{R}[10]) \\ \mathcal{R}[7] & \sqsupseteq & \textsf{enter}^\sharp\,(\mathcal{R}[4]) \\ \mathcal{R}[7] & \sqsupseteq & \textsf{enter}^\sharp\,(\mathcal{R}[8]) \\ \mathcal{R}[9] & \sqsupseteq & \textsf{combine}^\sharp\,(\mathcal{R}[8], \mathcal{R}[10]) \end{array}$$

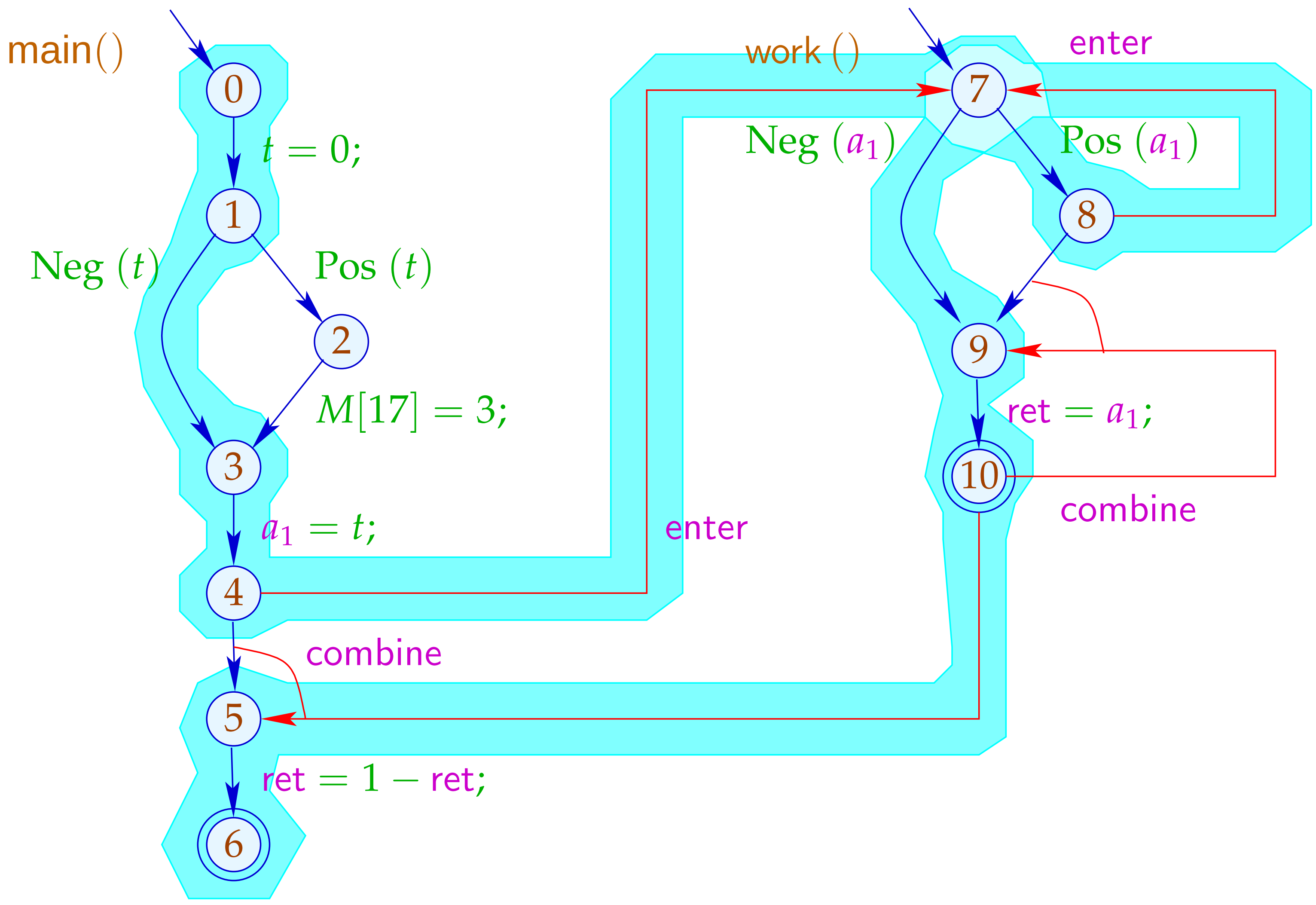
Warning:

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

... in the Example this is:



... 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 $\mathbb{D}$ 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

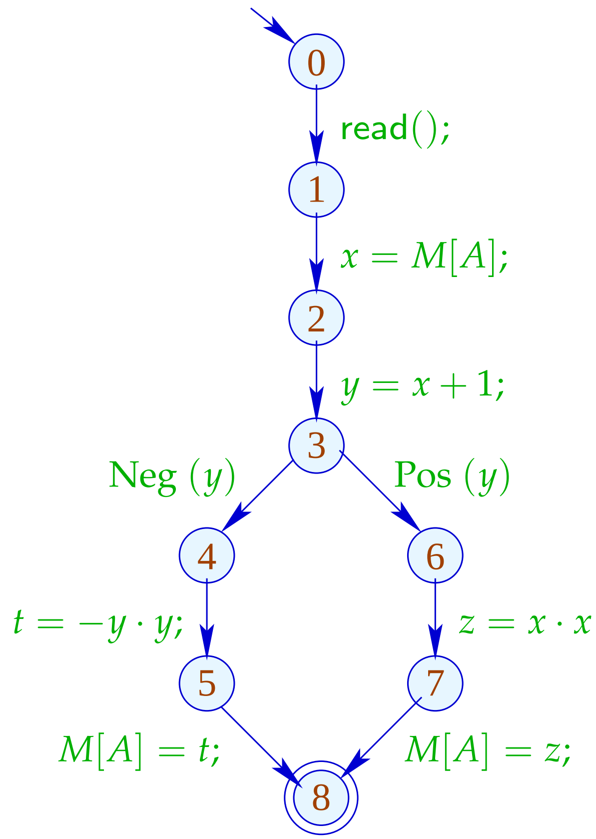
Question: How can we optimally use:

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

3.1 Registers

Example:

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



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 ...

$\text{read}();$

$x = M[A];$

$y = x + 1;$

$\text{if } (y) \ \{$

$\qquad z = x \cdot x;$

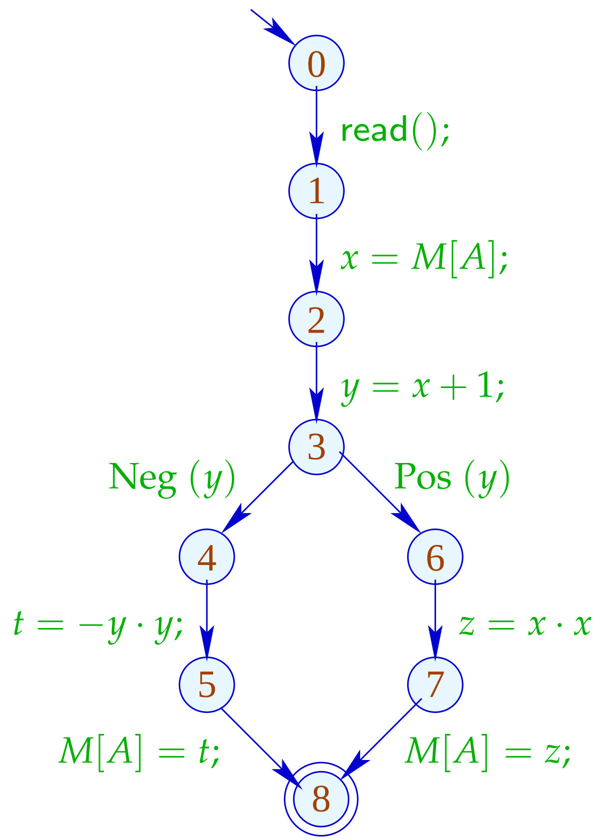
$\qquad M[A] = z;$

$\} \text{ else } \{$

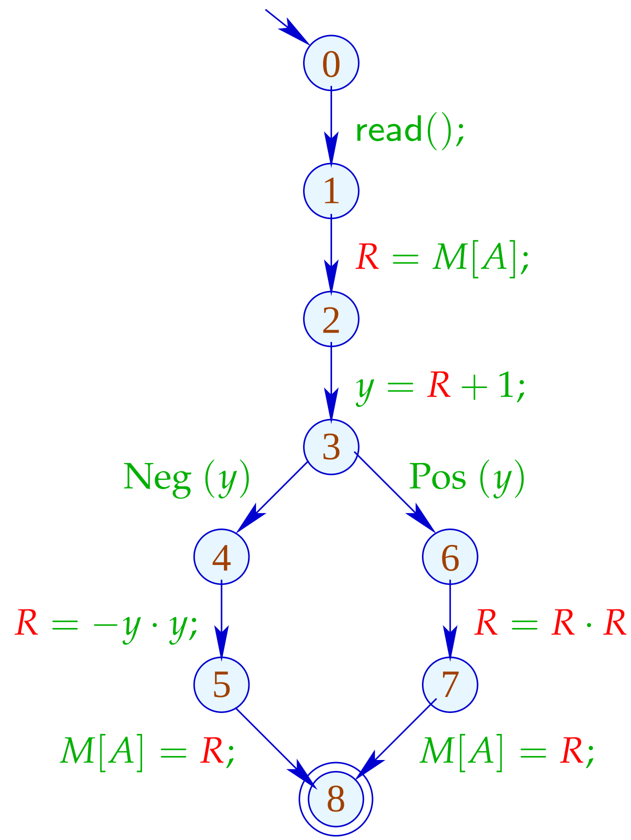
$\qquad t = -y \cdot y;$

$\qquad M[A] = t;$

$\}$



```
read();
R = M[A];
y = R + 1;
if  (y)  {
        R = R · R;
        M[A] = R;
}  else  {
        R = −y · y;
        M[A] = R;
}
```



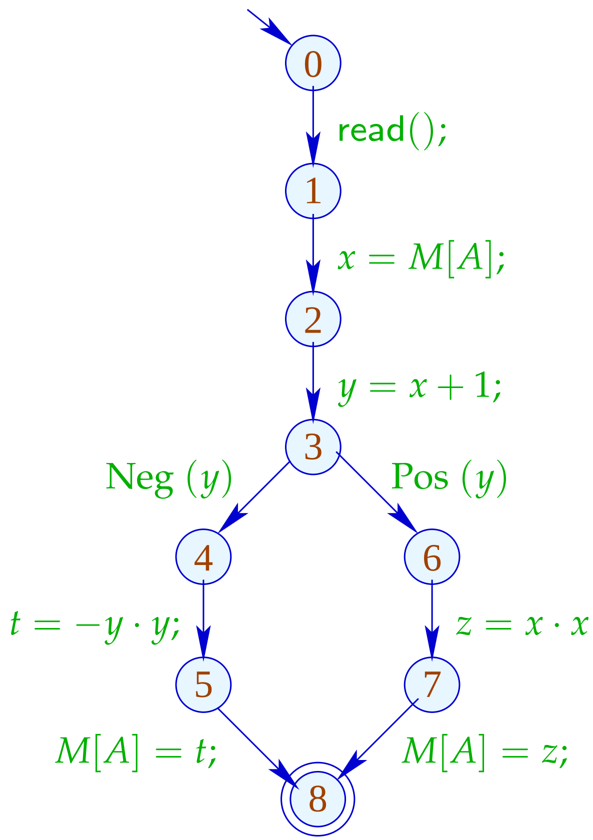
Warning:

This is only possible if the live ranges do not overlap :-)

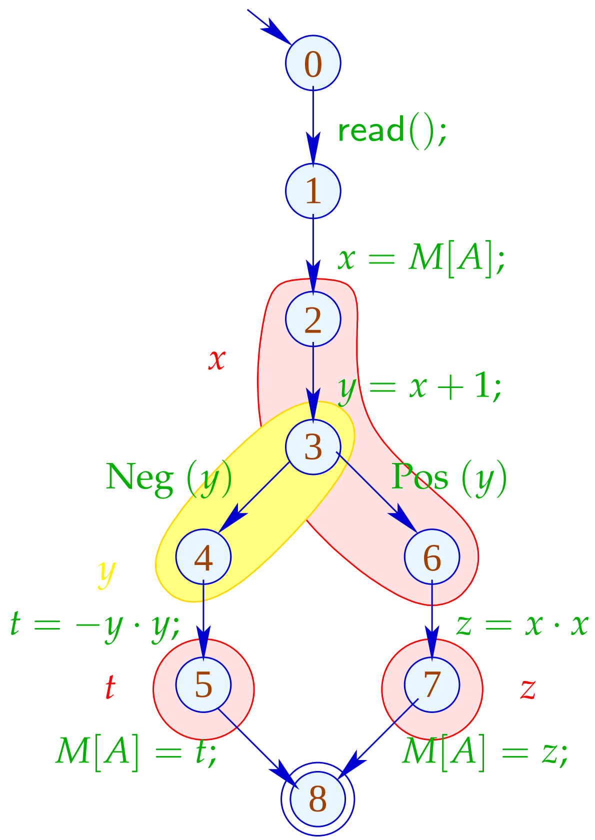
The (true) live range of x is defined by:

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

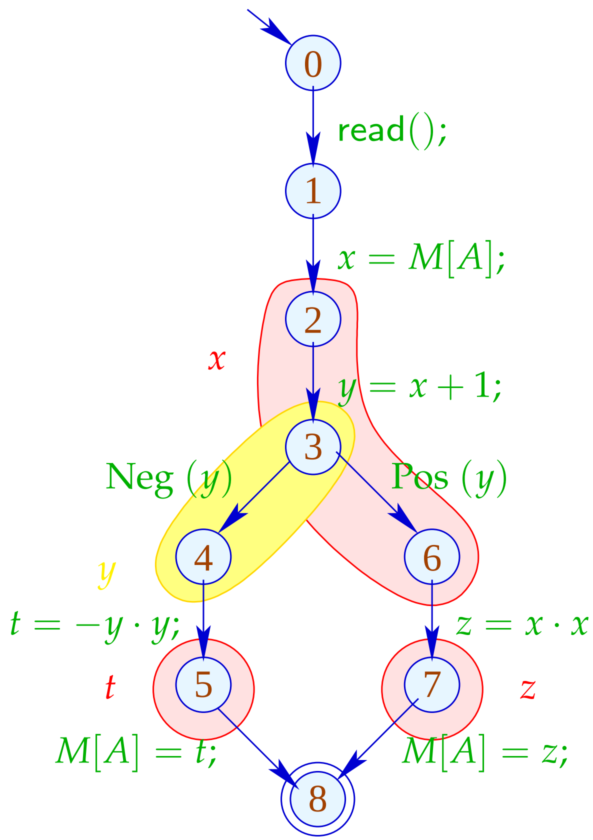
... in the Example:



	$\mathcal{L}$
8	$\emptyset$
7	$\{A, z\}$
6	$\{A, x\}$
5	$\{A, t\}$
4	$\{A, y\}$
3	$\{A, x, y\}$
2	$\{A, x\}$
1	$\{A\}$
0	$\emptyset$



	$\mathcal{L}$
8	$\emptyset$
7	$\{A, z\}$
6	$\{A, x\}$
5	$\{A, t\}$
4	$\{A, y\}$
3	$\{A, x, y\}$
2	$\{A, x\}$
1	$\{A\}$
0	$\emptyset$



Live Ranges:

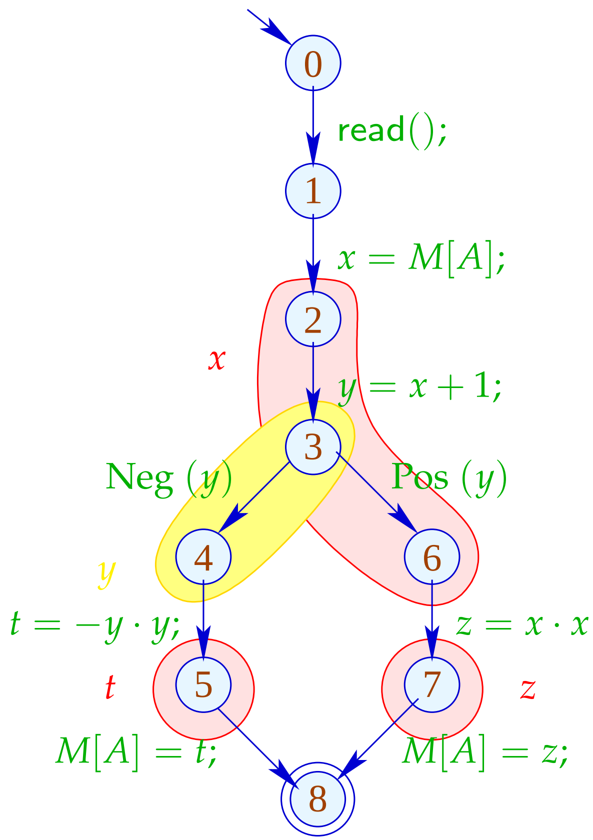
A	$\{1, \ldots, 7\}$
x	$\{2, 3, 6\}$
y	$\{2, 4\}$
t	$\{5\}$
z	$\{7\}$

In order to determine sets of compatible variables, we construct the Interference Graph $I = (Vars, E_I)$ where:

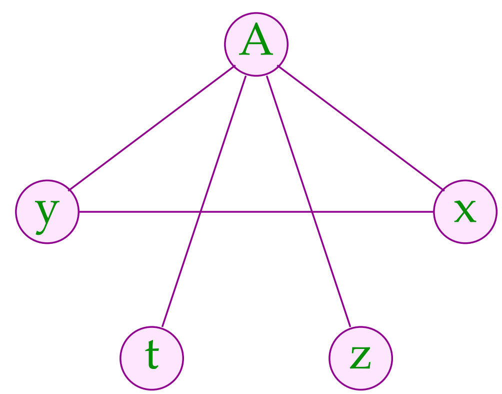
$$E_I = \{\{x, y\} \mid x \neq y, \mathcal{L}[x] \cap \mathcal{L}[y] \neq \emptyset\}$$

E_I has an edge for $x \neq y$ iff x, y are jointly live at some program point :-)

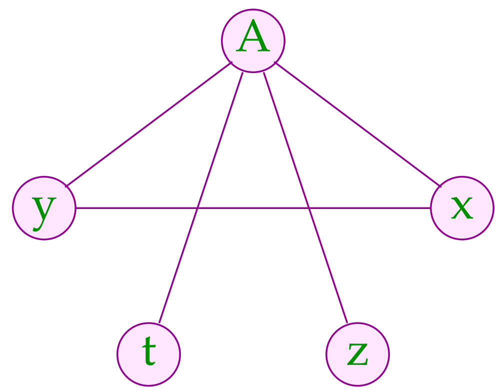
... in the Example:



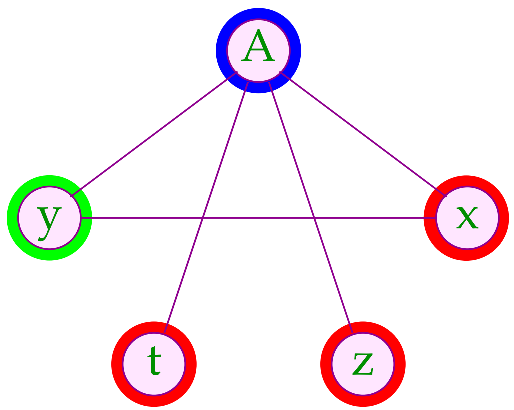
Interference Graph:



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



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



Color $==$ Register