

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:

$$f () \{ stmt^* \}$$

Additionally, we distinguish between **global** and **local** variables.

Program execution starts with the call of a procedure `main ()`.

Dedicated global/local variables a_i, b_i, ret can be used for parameter transfer.

Example:

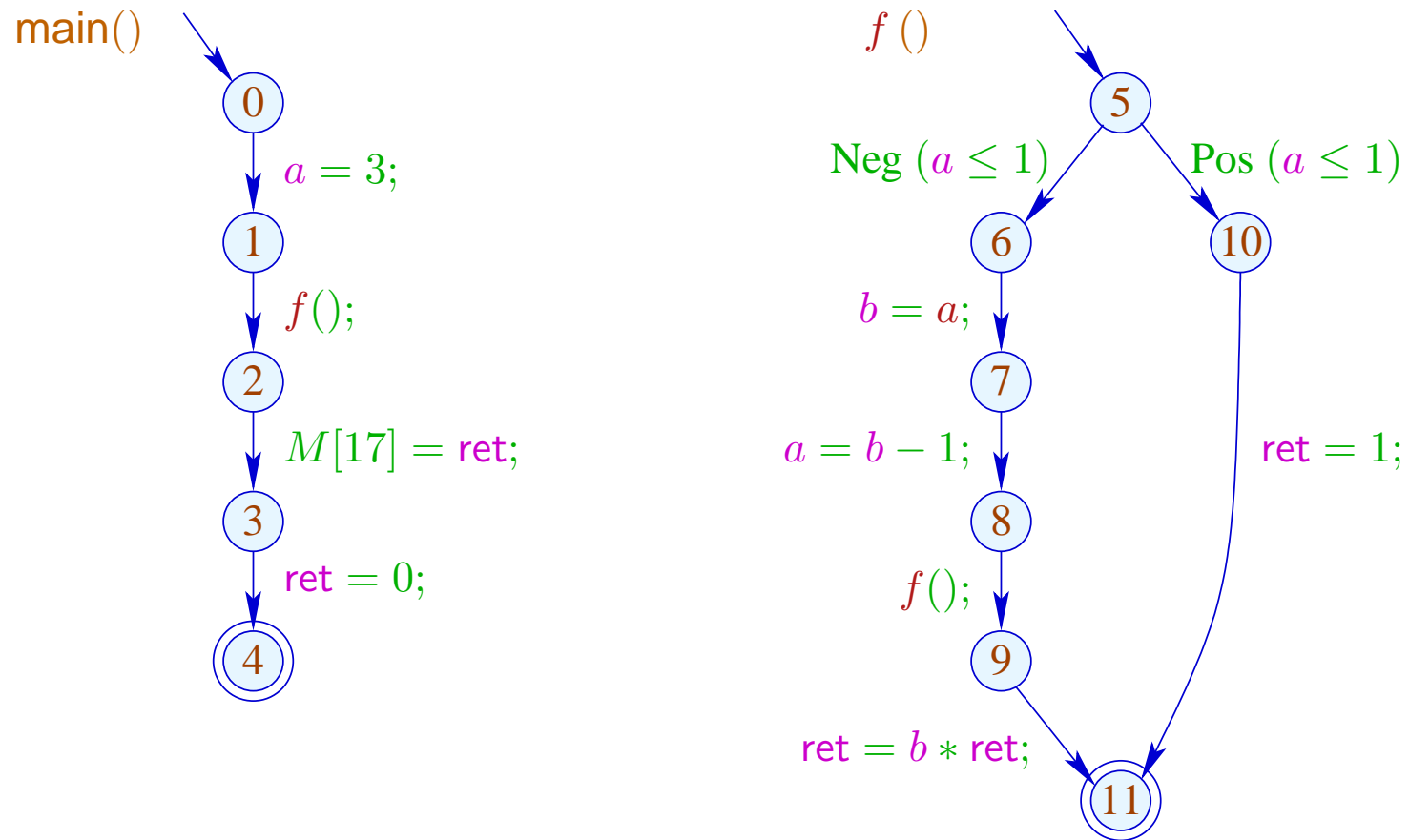
```
int a, ret;
main () {
    a = 3;
    f();
    M[17] = ret;
    ret = 0;
}

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

    exit :
}
```

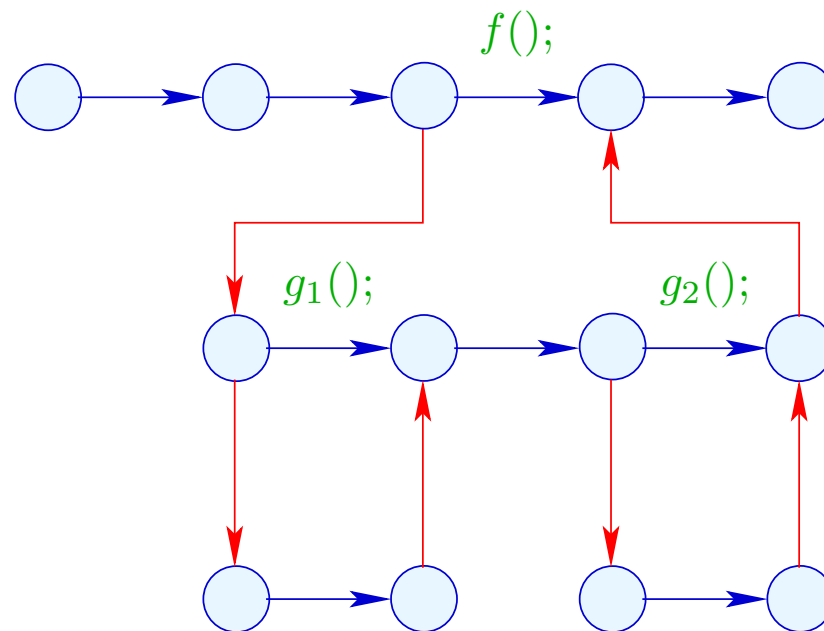
Such programs can be represented by a **set** of CFGs: one for each procedure ...

... in the example:

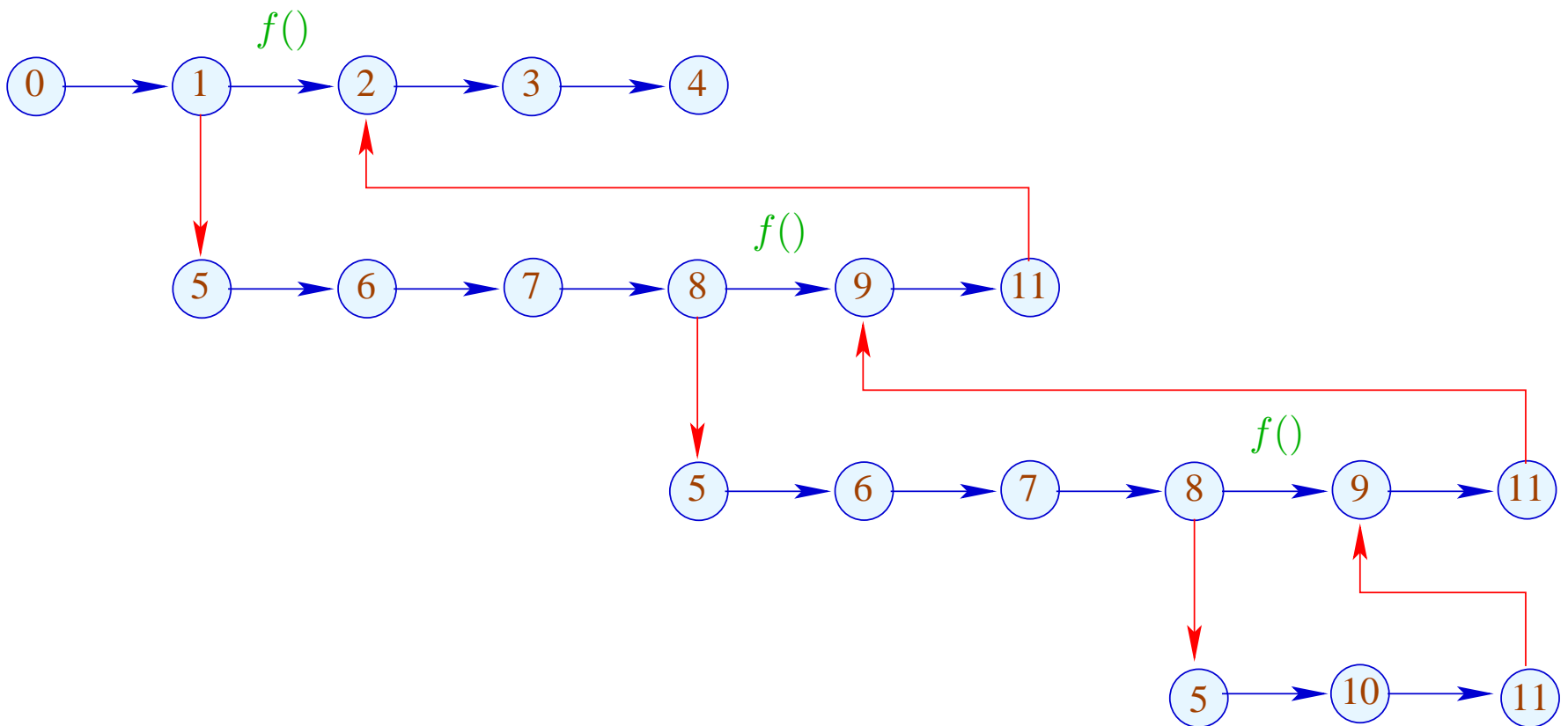


In order to optimize such programs, we require an extended operational semantics

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**.

$$\textit{configuration} \quad \equiv \quad \textit{stack} \times \textit{globals} \times \textit{store}$$

$$\textit{globals} \quad \equiv \quad \textit{Glob} \rightarrow \mathbb{Z}$$

$$\textit{store} \quad \equiv \quad \mathbb{N} \rightarrow \mathbb{Z}$$

$$\textit{stack} \quad \equiv \quad \textit{frame} \cdot \textit{frame}^*$$

$$\textit{frame} \quad \equiv \quad \textit{point} \times \textit{locals}$$

$$\textit{locals} \quad \equiv \quad \textit{Loc} \rightarrow \mathbb{Z}$$

Glob and *Loc* the sets of global and local variables.

point the set of program points.

The **call stack**, *stack* contains one **frame** for each procedure called, but not yet left.

A **frame** keeps the local state of computation inside a procedure call.

Computation steps refer to the current call.

The novel kinds of steps:

call $k = (u, f(), v)$:

$$(\sigma \cdot \boxed{(u, \rho_{Loc})}, \rho_{Glob}, \mu) \quad \vdash \quad (\sigma \cdot \boxed{(v, \rho_{Loc}) \cdot (u_f, \rho_f)}, \rho_{Glob}, \mu)$$

u_f entry point of f

return from a call :

$$(\sigma \cdot \boxed{(v, \rho_{Loc}) \cdot (r_f, -)}, \rho_{Glob}, \mu) \quad \vdash \quad (\sigma \cdot \boxed{(v, \rho_{Loc})}, \rho_{Glob}, \mu)$$

r_f 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 π 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 $\llbracket \cdot \rrbracket$ is extended to nested paths w :

$$\llbracket w \rrbracket : \textit{stack} \times \textit{globals} \times \textit{store} \rightarrow \textit{stack} \times \textit{globals} \times \textit{store}$$

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

$$\llbracket \pi_1 \langle f \rangle \pi_2 \langle /f \rangle \rrbracket = H(\llbracket \pi_2 \rrbracket) \circ \llbracket \pi_1 \rrbracket$$

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

$$H(g) (\rho_{Loc}, \rho_{Glob}, \mu) = \mathbf{let} (\rho'_{Loc}, \rho'_{Glob}, \mu') = g(\underline{Q}, \rho_{Glob}, \mu) \\ \mathbf{in} (\rho_{Loc}, \rho'_{Glob}, \mu')$$

In general, $\llbracket w \rrbracket$ is only partially defined.

Alternatively:

- determine the initial values for the locals:

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

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

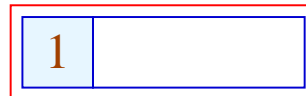
$$\text{combine } (\rho_1, \rho_2) = (\rho_1|_{Loc}) \uplus (\rho_2|_{Glob})$$

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

$$\begin{aligned} \llbracket k \langle w \rangle \rrbracket (\rho, \mu) &= \text{let } (\rho_1, \mu_1) = \llbracket w \rrbracket (\text{enter } \rho, \mu) \\ &\text{in } (\text{combine } (\rho, \rho_1), \mu_1) \end{aligned}$$

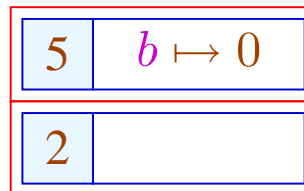
The call stack explicitly implements the DFS traversal through the computation forest

... in the example:



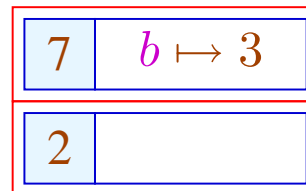
The call stack explicitly implements the DFS traversal through the computation forest

... in the example:



The 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:

5	$b \mapsto 0$
9	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest

... in the example:

7	$b \mapsto 2$
9	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest

... in the example:

5	$b \mapsto 0$
9	$b \mapsto 2$
9	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest

... in the example:

11	$b \mapsto 0$
9	$b \mapsto 2$
9	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest

... in the example:

9	$b \mapsto 2$
9	$b \mapsto 3$
2	

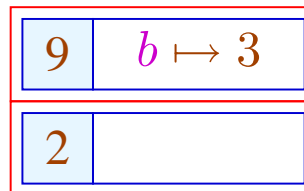
The call stack explicitly implements the DFS traversal through the computation forest

... in the example:

11	$b \mapsto 2$
9	$b \mapsto 3$
2	

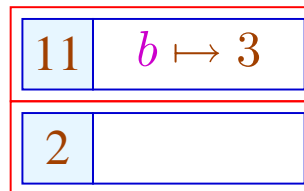
The call stack explicitly implements the DFS traversal through the computation forest

... in the example:



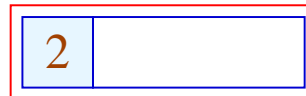
The call stack explicitly implements the DFS traversal through the computation forest

... in the example:

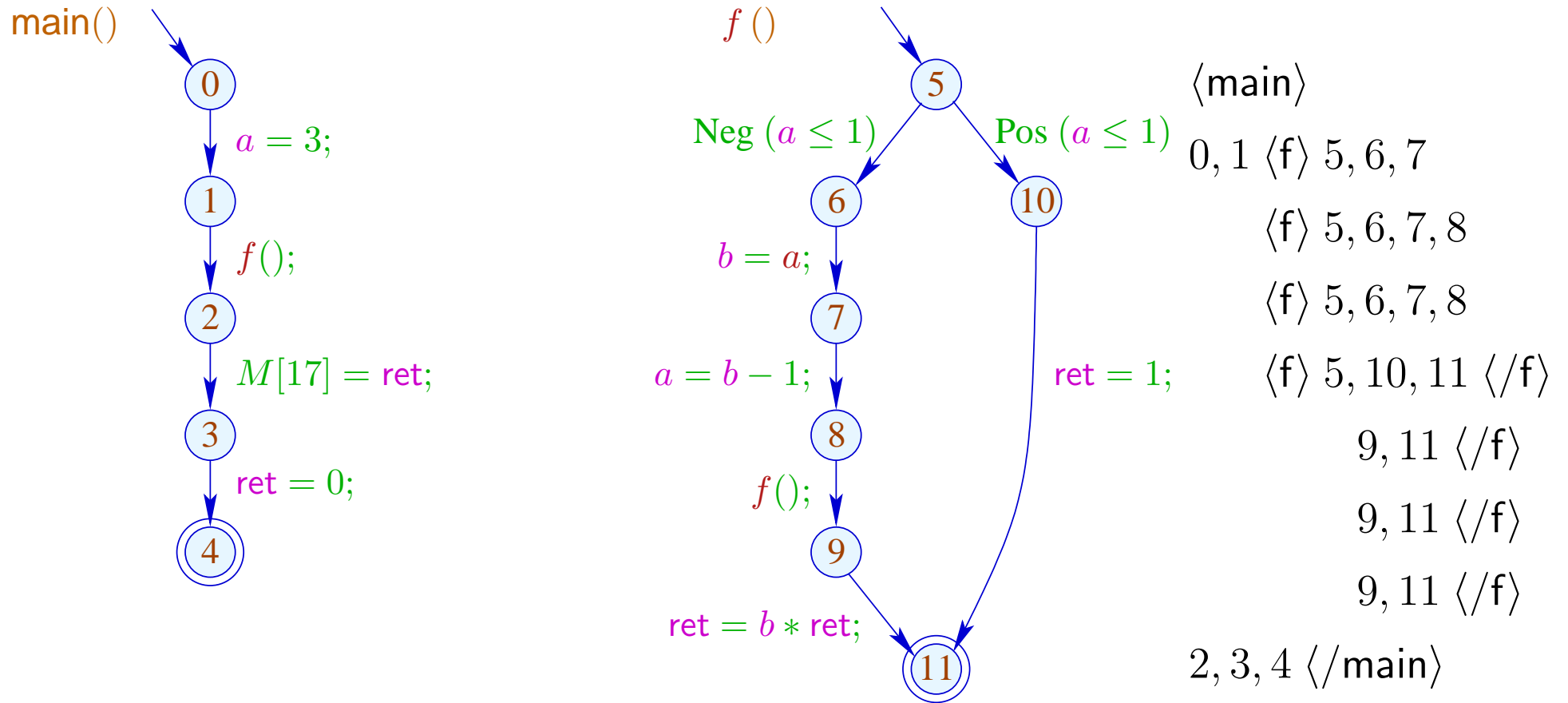


The call stack explicitly implements the DFS traversal through the computation forest

... in the example:



The factorial program and one of its computations:



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:

```
abs () {  
     $a_2 = -a_1$ ;  
    max ();  
}  
  
max () {  
    if ( $a_1 < a_2$ ) {  $ret = a_2$ ; goto _exit; }  
     $ret = a_1$ ;  
    _exit :  
}
```

... yields:

```
abs () {  
   $a_2 = -a_1$ ;  
  if ( $a_1 < a_2$ ) {  $ret = a_2$ ; goto _exit; }  
   $ret = a_1$ ;  
  _exit :  
}
```

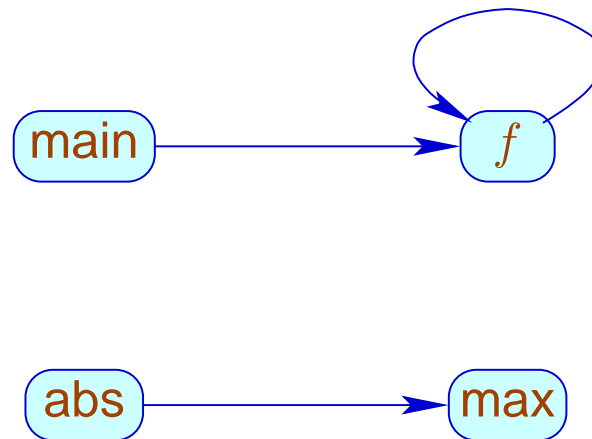
Problems:

- 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:

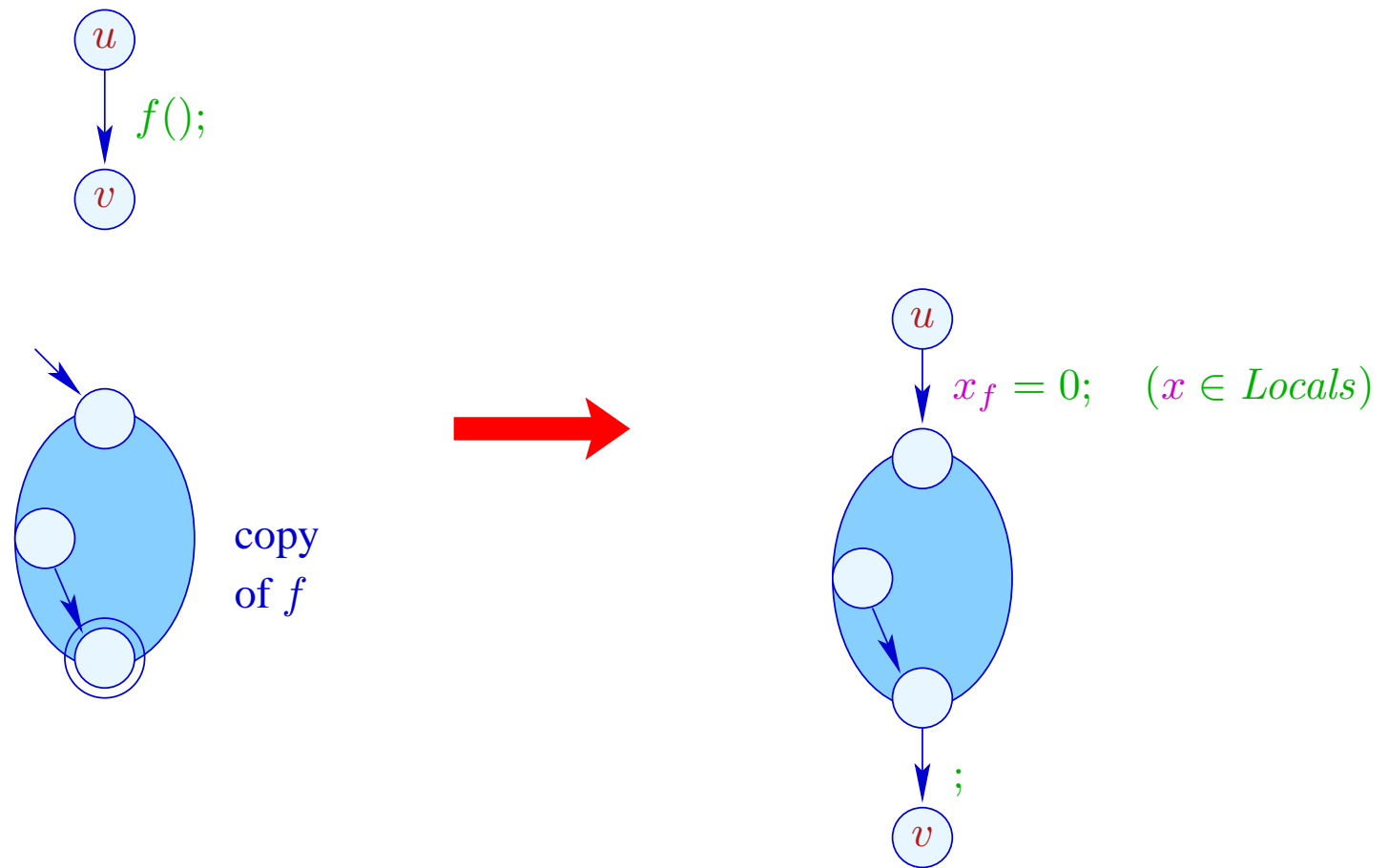
- The nodes are the procedures.
- An edge connects g with h , whenever the body of g contains a call of h .

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:



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

```
f () { int b;  
      if (a2 ≤ 1) { ret = a1; goto _exit; }  
      b = a1 · a2;  
      a2 = a2 - 1;  
      a1 = b;  
      f ();  
      _exit :  
    }
```

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

⇒ We may **directly** jump to the beginning

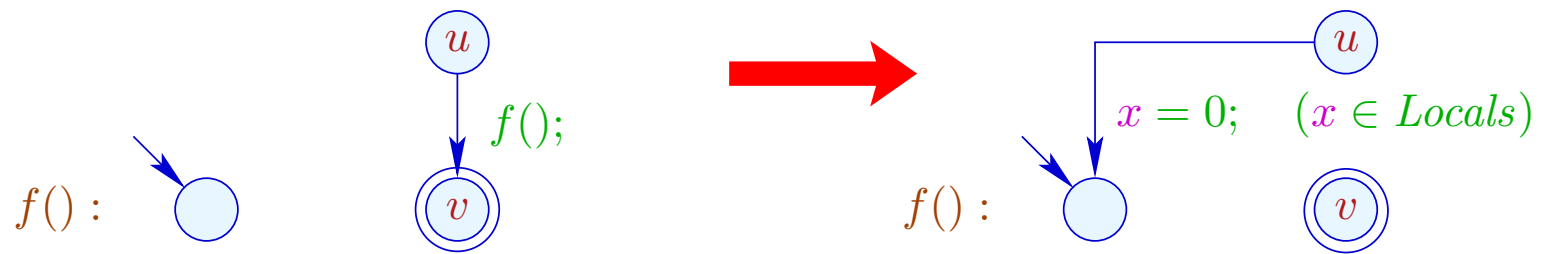
... after having reset the locals to 0.

... this yields in the example:

```
f () { int b;  
  _f :   if (a2 ≤ 1) { ret = a1; goto _exit; }  
        b = a1 · a2;  
        a2 = a2 − 1;  
        a1 = b;  
        b = 0; goto _f;  
  _exit :  
}
```

// It works, since we have ruled out **references to variables!**

Transformation LC:



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?

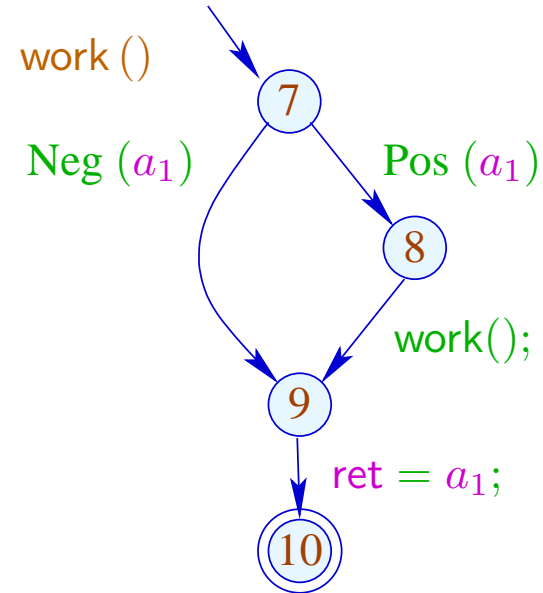
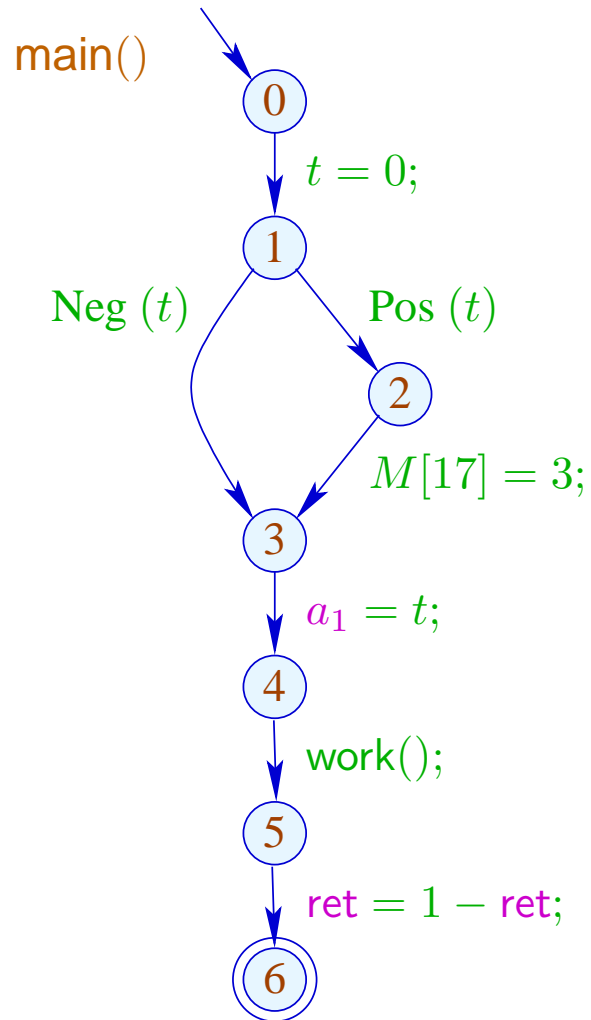
Example: Constant Propagation

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

work() {
        if (a1) work();
        ret = a1;
    }
```

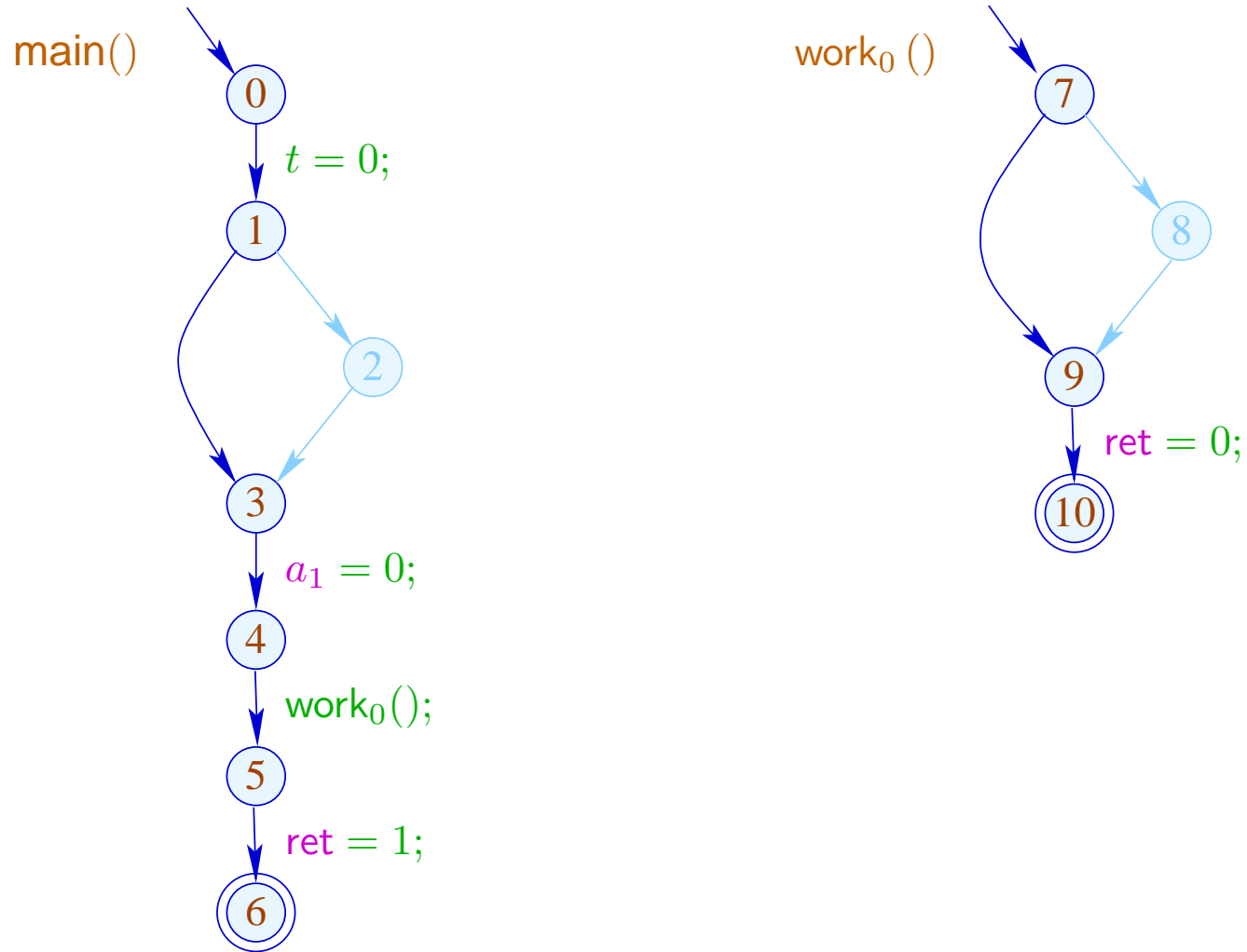
Example:

Constant Propagation



Example:

Constant Propagation



(1) **Functional Approach:**

Let \mathbb{D} denote a complete lattice of (abstract) states.

Idea:

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

$$\llbracket f \rrbracket^\# : \mathbb{D} \rightarrow \mathbb{D}$$

In order to determine the effect of a call edge $k = (u, f();, v)$ we require abstract functions:

$$\begin{aligned} \text{enter}^\# & : \mathbb{D} \rightarrow \mathbb{D} \\ \text{combine}^\# & : \mathbb{D}^2 \rightarrow \mathbb{D} \end{aligned}$$

Then we define:

$$\llbracket k \rrbracket^\# D = \text{combine}^\# (D, \llbracket f \rrbracket^\# (\text{enter}^\# D))$$

... for Constant Propagation:

$$\mathbb{D} = (\mathit{Vars} \rightarrow \mathbb{Z}^{\top})_{\perp}$$

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

$$\mathit{combine}^{\#} (D_1, D_2) = \begin{cases} \perp & \text{if } D_1 = \perp \vee D_2 = \perp \\ D_1|_{\mathit{Loc}} \uplus D_2|_{\mathit{Glob}} & \text{otherwise} \end{cases}$$

The effects $\llbracket f \rrbracket^\sharp$ then can be determined by a system of constraints over the complete lattice $\mathbb{D} \rightarrow \mathbb{D}$:

$$\begin{aligned} \llbracket v \rrbracket^\sharp &\sqsupseteq \text{Id} && v \text{ entry point} \\ \llbracket v \rrbracket^\sharp &\sqsupseteq \llbracket k \rrbracket^\sharp \circ \llbracket u \rrbracket^\sharp && k = (u, _, v) \text{ edge} \\ \llbracket f \rrbracket^\sharp &\sqsupseteq \llbracket stop_f \rrbracket^\sharp && stop_f \text{ end point of } f \end{aligned}$$

$\llbracket v \rrbracket^\sharp : \mathbb{D} \rightarrow \mathbb{D}$ 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 $f : \mathbb{D} \rightarrow \mathbb{D}$?
- If $\#\mathbb{D} = \infty$, then $\mathbb{D} \rightarrow \mathbb{D}$ has **infinite** strictly increasing chains

Simplification: Copy-Constants

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

Observation:

→ The effects of assignments are:

$$\llbracket x = e; \rrbracket^\# D = \begin{cases} D \oplus \{x \mapsto c\} & \text{if } e = c \in \mathbb{Z} \\ D \oplus \{x \mapsto (D \ y)\} & \text{if } e = y \in \mathit{Vars} \\ D \oplus \{x \mapsto \top\} & \text{otherwise} \end{cases}$$

→ 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

$$\mathbb{D}_f \rightarrow \mathbb{D}_f \quad \text{with} \quad \mathbb{D}_f = (\mathit{Vars} \rightarrow \mathbb{V}^\top)_\perp$$

→ The complete lattice is huge, but **finite**!

Improvement:

- Not all functions from $\mathbb{D}_f \rightarrow \mathbb{D}_f$ will occur
- All occurring functions $\lambda D. \perp \neq M$ are of the form:

$$\begin{aligned} M &= \{x \mapsto (b_x \sqcup \bigsqcup_{y \in I_x} y) \mid x \in Vars\} && \text{where:} \\ M D &= \{x \mapsto (b_x \sqcup \bigsqcup_{y \in I_x} D y) \mid x \in Vars\} && \text{für } D \neq \perp \end{aligned}$$

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

$$(M_1 \sqcup M_2) x = (M_1 x) \sqcup (M_2 x)$$

- For $k = \#Vars$, \mathbb{M} has height $\mathcal{O}(k^2)$

Improvement (Cont.):

→ Also, composition can be directly implemented:

$$(M_1 \circ M_2) x = b' \sqcup \bigsqcup_{y \in I'} y \quad \text{with}$$

$$b' = b \sqcup \bigsqcup_{z \in I} b_z$$

$$I' = \bigcup_{z \in I} I_z \quad \text{where}$$

$$M_1 x = b \sqcup \bigsqcup_{y \in I} y$$

$$M_2 z = b_z \sqcup \bigsqcup_{y \in I_z} y$$

→ The effects of assignments then are:

$$\llbracket x = e; \rrbracket^\# = \begin{cases} \text{Id}_{Vars} \oplus \{x \mapsto c\} & \text{if } e = c \in \mathbb{Z} \\ \text{Id}_{Vars} \oplus \{x \mapsto y\} & \text{if } e = y \in Vars \\ \text{Id}_{Vars} \oplus \{x \mapsto \top\} & \text{otherwise} \end{cases}$$

... in the Example:

$$\begin{aligned} \llbracket t = 0; \rrbracket^\# &= \{a_1 \mapsto a_1, \text{ret} \mapsto \text{ret}, t \mapsto 0\} \\ \llbracket a_1 = t; \rrbracket^\# &= \{a_1 \mapsto t, \text{ret} \mapsto \text{ret}, t \mapsto t\} \end{aligned}$$

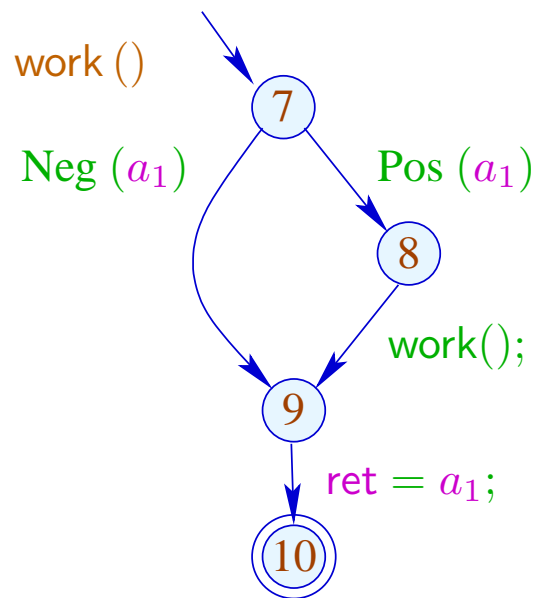
In order to implement the analysis, we additionally must construct the effect of a call $k = (_, f ();, _)$ from the effect of a procedure f :

$$\begin{aligned} \llbracket k \rrbracket^\# &= H(\llbracket f \rrbracket^\#) && \text{where:} \\ H(M) &= \text{Id}|_{Loc} \uplus (M \circ \text{enter}^\#)|_{Glob} \\ \text{enter}^\# x &= \begin{cases} x & \text{if } x \in Glob \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

... in the Example:

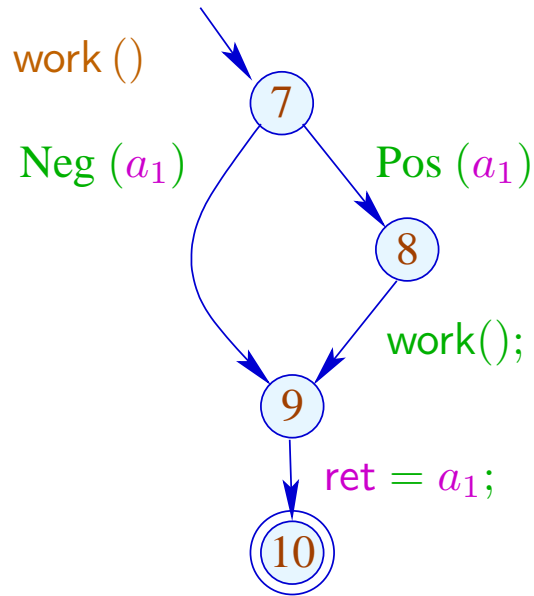
$$\begin{aligned} \text{If } \llbracket \text{work} \rrbracket^\# &= \{a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t\} \\ \text{then } H \llbracket \text{work} \rrbracket^\# &= \text{Id}_{\{t\}} \oplus \{a_1 \mapsto a_1, \text{ret} \mapsto a_1\} \\ &= \{a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t\} \end{aligned}$$

Now we can perform fixpoint iteration



	1
7	$\{a_1 \mapsto a_1, \text{ret} \mapsto \text{ret}, t \mapsto t\}$
9	$\{a_1 \mapsto a_1, \text{ret} \mapsto \text{ret}, t \mapsto t\}$
10	$\{a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t\}$
8	$\{a_1 \mapsto a_1, \text{ret} \mapsto \text{ret}, t \mapsto t\}$

$$\begin{aligned}
 \llbracket (8, \dots, 9) \rrbracket^\# \circ \llbracket 8 \rrbracket^\# &= \{a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t\} \circ \\
 &\quad \{a_1 \mapsto a_1, \text{ret} \mapsto \text{ret}, t \mapsto t\} \\
 &= \{a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t\}
 \end{aligned}$$



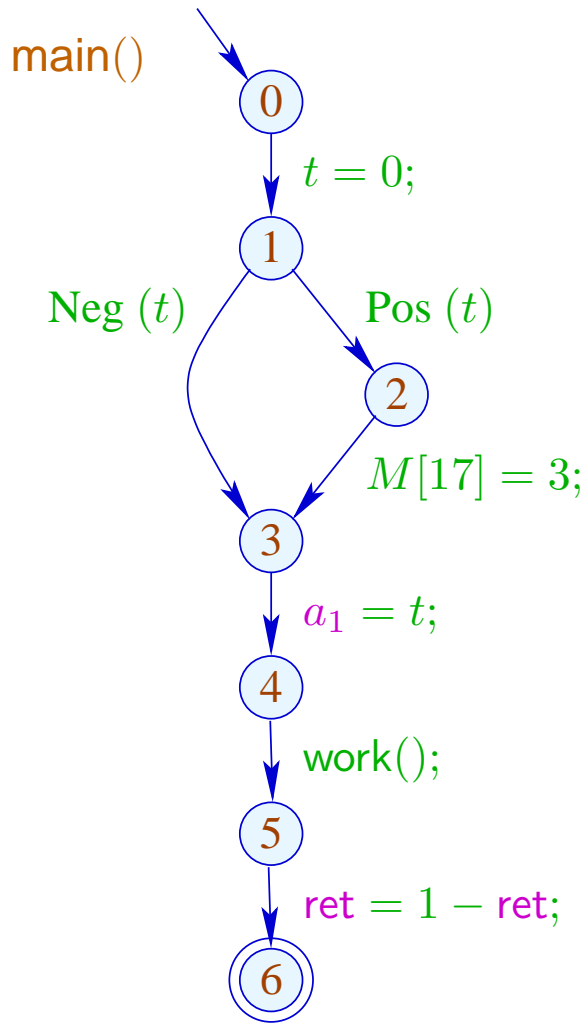
	2
7	$\{a_1 \mapsto a_1, \text{ret} \mapsto \text{ret}, t \mapsto t\}$
9	$\{a_1 \mapsto a_1, \text{ret} \mapsto a_1 \sqcup \text{ret}, t \mapsto t\}$
10	$\{a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t\}$
8	$\{a_1 \mapsto a_1, \text{ret} \mapsto \text{ret}, t \mapsto t\}$

$$\begin{aligned}
 \llbracket (8, \dots, 9) \rrbracket^\# \circ \llbracket 8 \rrbracket^\# &= \{a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t\} \circ \\
 &\quad \{a_1 \mapsto a_1, \text{ret} \mapsto \text{ret}, t \mapsto t\} \\
 &= \{a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t\}
 \end{aligned}$$

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:

$$\begin{array}{lll}
 \mathcal{R}[\text{main}] & \sqsupseteq & \text{enter}^\# d_0 \\
 \mathcal{R}[f] & \sqsupseteq & \text{enter}^\# (\mathcal{R}[u]) \quad k = (u, f(), _) \quad \text{call} \\
 \mathcal{R}[v] & \sqsupseteq & \mathcal{R}[f] \quad v \text{ entry point of } f \\
 \mathcal{R}[v] & \sqsupseteq & \llbracket k \rrbracket^\# (\mathcal{R}[u]) \quad k = (u, _, v) \quad \text{edge}
 \end{array}$$

... in the Example:



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

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 $\mathbb{M} \subseteq \mathbb{D} \rightarrow \mathbb{D}$ 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

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

$\llbracket v, a \rrbracket^\# \sqsupseteq a$ v entry point

$\llbracket v, a \rrbracket^\# \sqsupseteq \text{combine}^\# (\llbracket u, a \rrbracket, \llbracket f, \text{enter}^\# \llbracket u, a \rrbracket^\# \rrbracket^\#)$
 $(u, f ();, v)$ call

$\llbracket v, a \rrbracket^\# \sqsupseteq \llbracket lab \rrbracket^\# \llbracket u, a \rrbracket^\#$ $k = (u, lab, v)$ edge

$\llbracket f, a \rrbracket^\# \sqsupseteq \llbracket stop_f, a \rrbracket^\#$ $stop_f$ end point of f

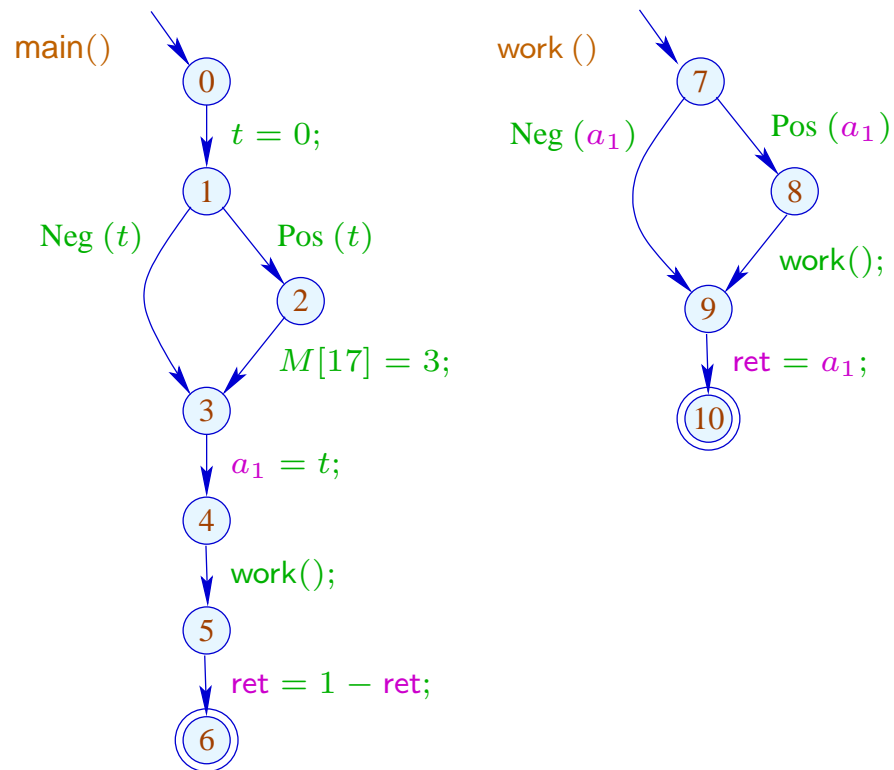
// $\llbracket v, a \rrbracket^\# =$ value for the argument a .

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 that **occur**, i.e., which are necessary to determine the value $\llbracket \text{main}(), a_0 \rrbracket^\sharp$
 \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 ...



	a_1	ret	a_1	ret
0	T	T	T	T
1	T	T	T	T
2	T	T	⊥	
3	T	T	T	T
4	T	T	0	T
7	0	T	0	T
8	0	T	⊥	
9	0	T	0	T
10	0	T	0	0
5	T	T	0	0
main()	T	T	0	1

Discussion:

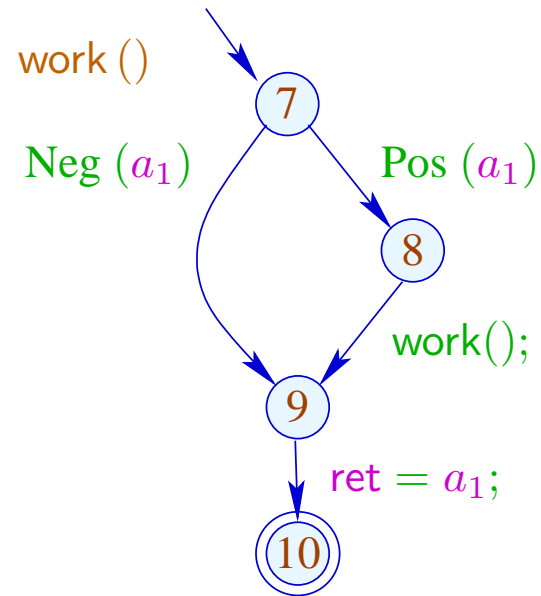
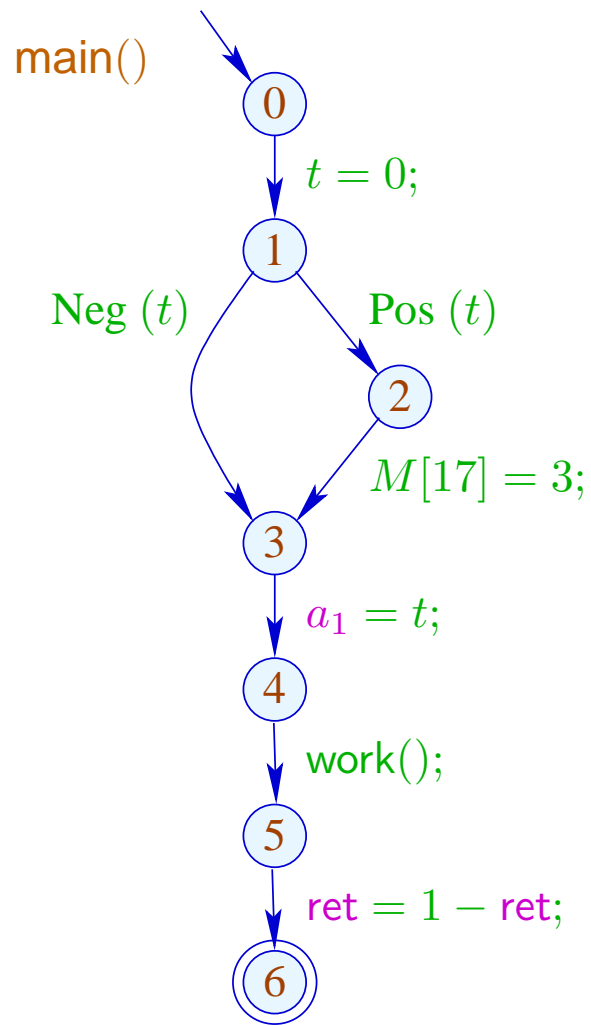
- In the Example, the analysis terminates **quickly**
- If \mathbb{D} has finite height, the analysis terminates if each procedure is only analyzed for **finitely many** arguments
- Analogous analysis algorithms have proved very effective for the analysis of **Prolog**
- Together with a points-to analysis 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:

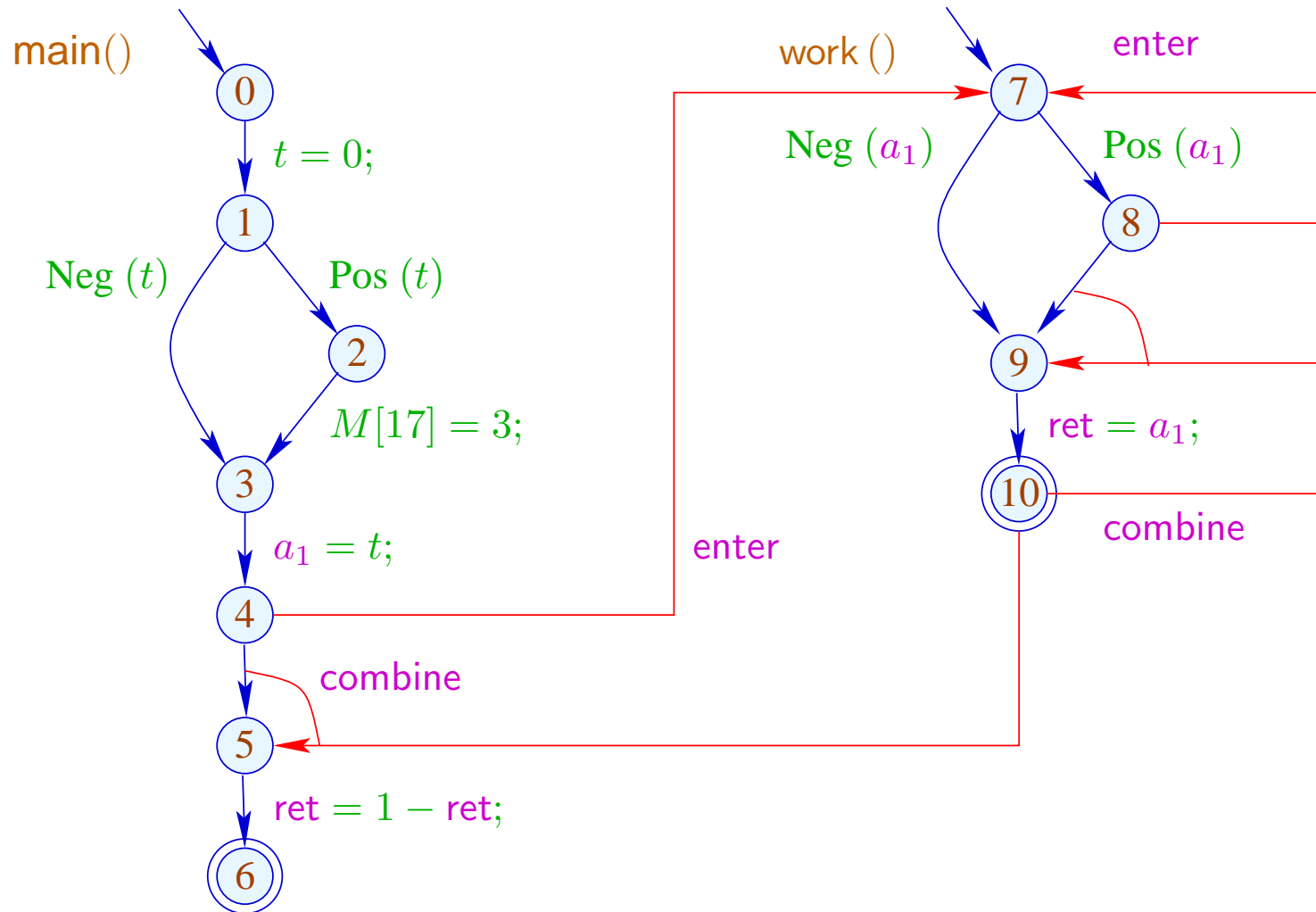
Idea:

- 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: $d = 0$.
 - ⇒ Just track the current stack frame ...

... in the Example:



... in the Example:



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

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

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

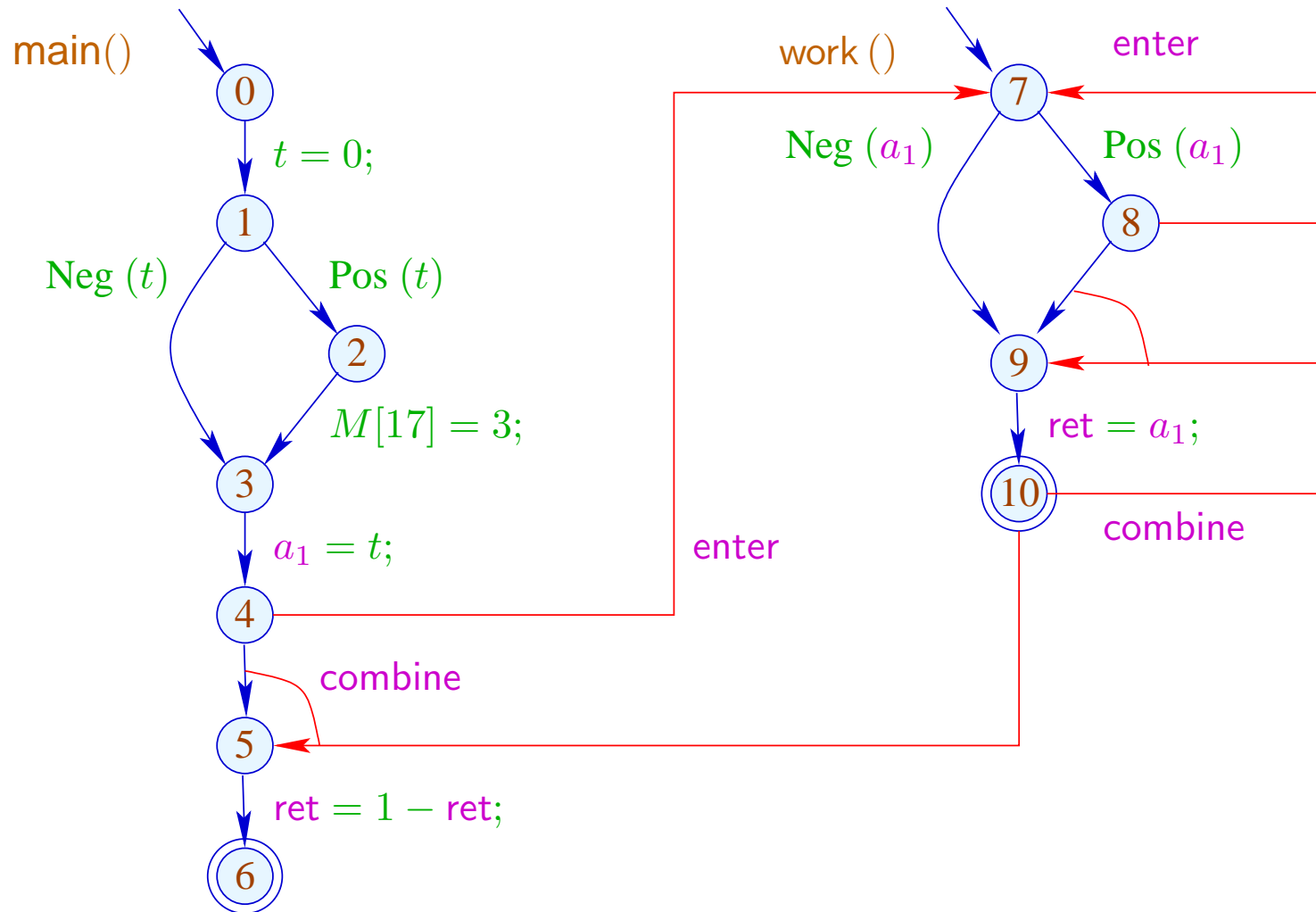
$$\mathcal{R}[7] \sqsupseteq \text{enter}^\# (\mathcal{R}[8])$$

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

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 \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