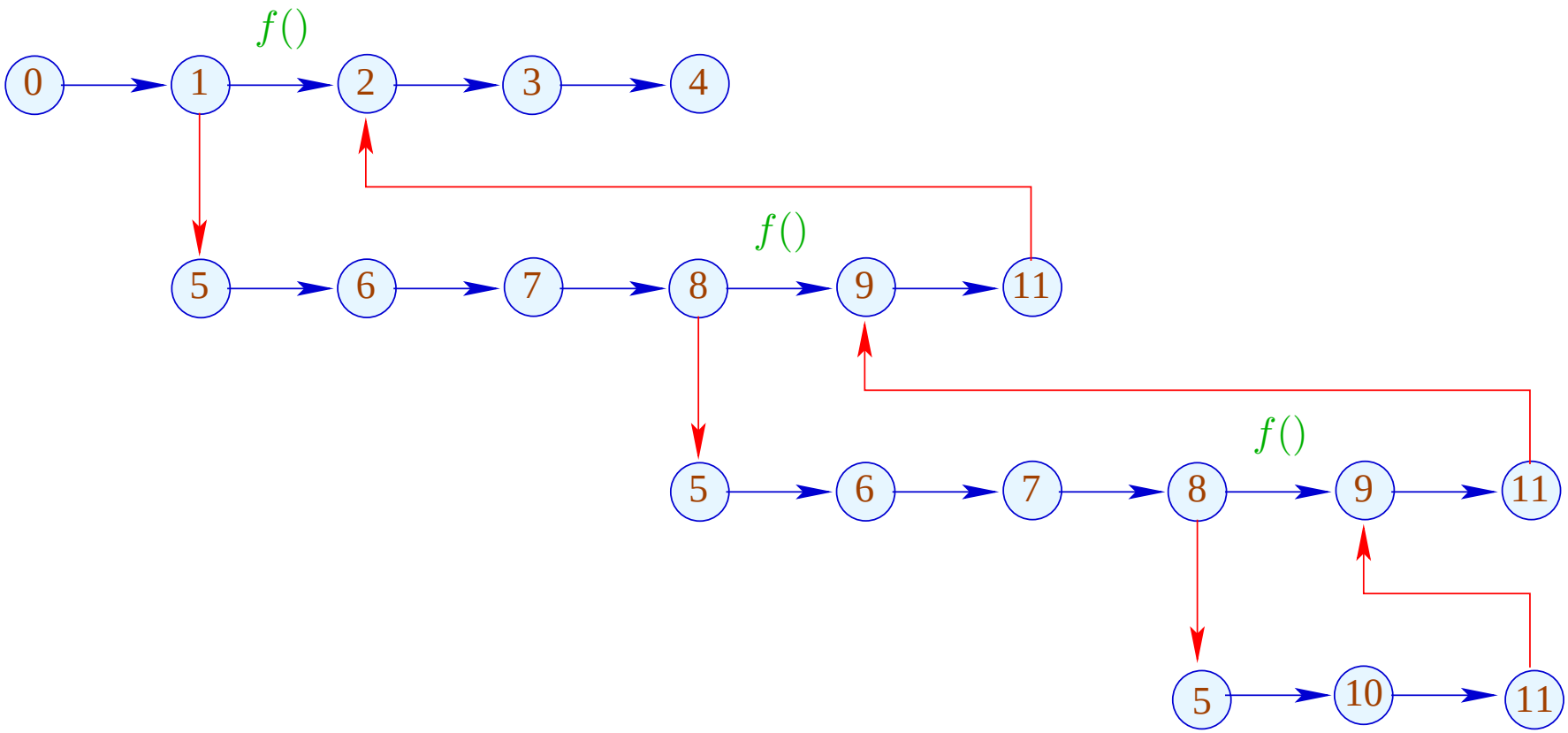
... in the Example:



The function $[\![.]\!]$ is extended to computation forests: w :

$$[\![w]\!] : (\mathit{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z}) \to (\mathit{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z})$$

For a call $k = (u, f();, v)$ we must:

- determine the initial values for the locals:

$$\mathsf{enter}\ \rho = \{x \mapsto 0 \mid x \in \mathit{Locals}\} \oplus (\rho|_{\mathit{Globals}})$$

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

$$\mathsf{combine}\ (\rho_1, \rho_2) = (\rho_1|_{\mathit{Locals}}) \oplus (\rho_2|_{\mathit{Globals}})$$

- ... evaluate the computation forest inbetween:

$$\begin{array}{rcl} [\![k\ \langle w \rangle]\!]\ (\rho, \mu) & = & \mathsf{let}\ \ (\rho_1, \mu_1) = [\![w]\!]\ (\mathsf{enter}\ \rho, \mu) \\ & & \mathsf{in}\ \ \ (\mathsf{combine}\ (\rho, \rho_1), \mu_1) \end{array}$$

Warning:

- In general, $[\![w]\!]$ is only partially defined :-)
- Dedicated global/local variables a_i, b_i, ret can be used to simulate specific calling conventions.
- The standard operational semantics relies on configurations which maintain a call stack.
- Computation forests are better suited for the construction of analyses and correctness proofs :-)
- It is an awkward (but useful) exercise to prove the equivalence of the two approaches ...

Configurations:

$$
\begin{array}{lcl}
\mathit{configuration} & = & \mathit{stack} \times \mathit{store} \\
\mathit{store} & = & \mathit{globals} \times (\mathbb{N} \to \mathbb{Z}) \\
\mathit{globals} & = & (\mathit{Globals} \to \mathbb{Z}) \\
\mathit{stack} & = & \mathit{frame} \cdot \mathit{frame}^* \\
\mathit{frame} & = & \mathit{point} \times \mathit{locals} \\
\mathit{locals} & = & (\mathit{Locals} \to \mathbb{Z})
\end{array}
$$

A *frame* specifies the local state of computation inside a procedure call :-)

The leftmost frame corresponds to the current call.

Computation steps refer to the current call :-)

The novel kinds of steps:

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

$$(\boxed{(u, \rho)} \cdot \sigma, \langle \gamma, \mu \rangle) \implies (\boxed{(u_f, \{x \to 0 \mid x \in \mathit{Locals}\}) \cdot (v, \rho)} \cdot \sigma, \langle \gamma, \mu \rangle)$$

u_f entry point of f

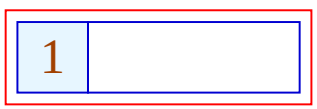
return:

$$(\boxed{(r_f, \_)} \cdot \sigma, \langle \gamma, \mu \rangle) \implies (\sigma, \langle \gamma, \mu \rangle)$$

r_f return point of f

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	$b \mapsto 0$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:

7	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:

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

9	$b \mapsto 3$
2	

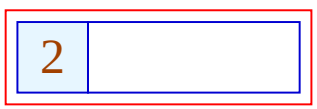
The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:

11	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:



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.

$\Longrightarrow$ ... quite expensive !!!

1. Idea: Inlining

Copy the procedure body at every call site !!!

Example:

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

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

... yields:

$abs\ ()\ \{$

$a_2 = -a_1;$

$\quad$ if $(a_1 < a_2)\ \{\ \mathsf{ret} = a_2;\ \ \mathsf{goto}\ \_exit;\ \ \}$

$\quad$ $\mathsf{ret} = a_1;$

$\_exit:$

$\}$

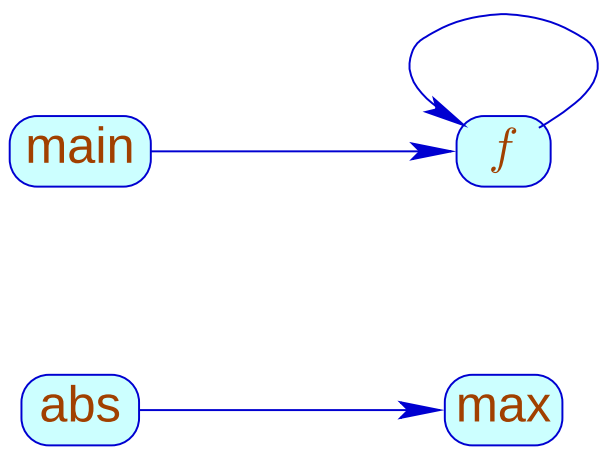
Problems:

- 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:

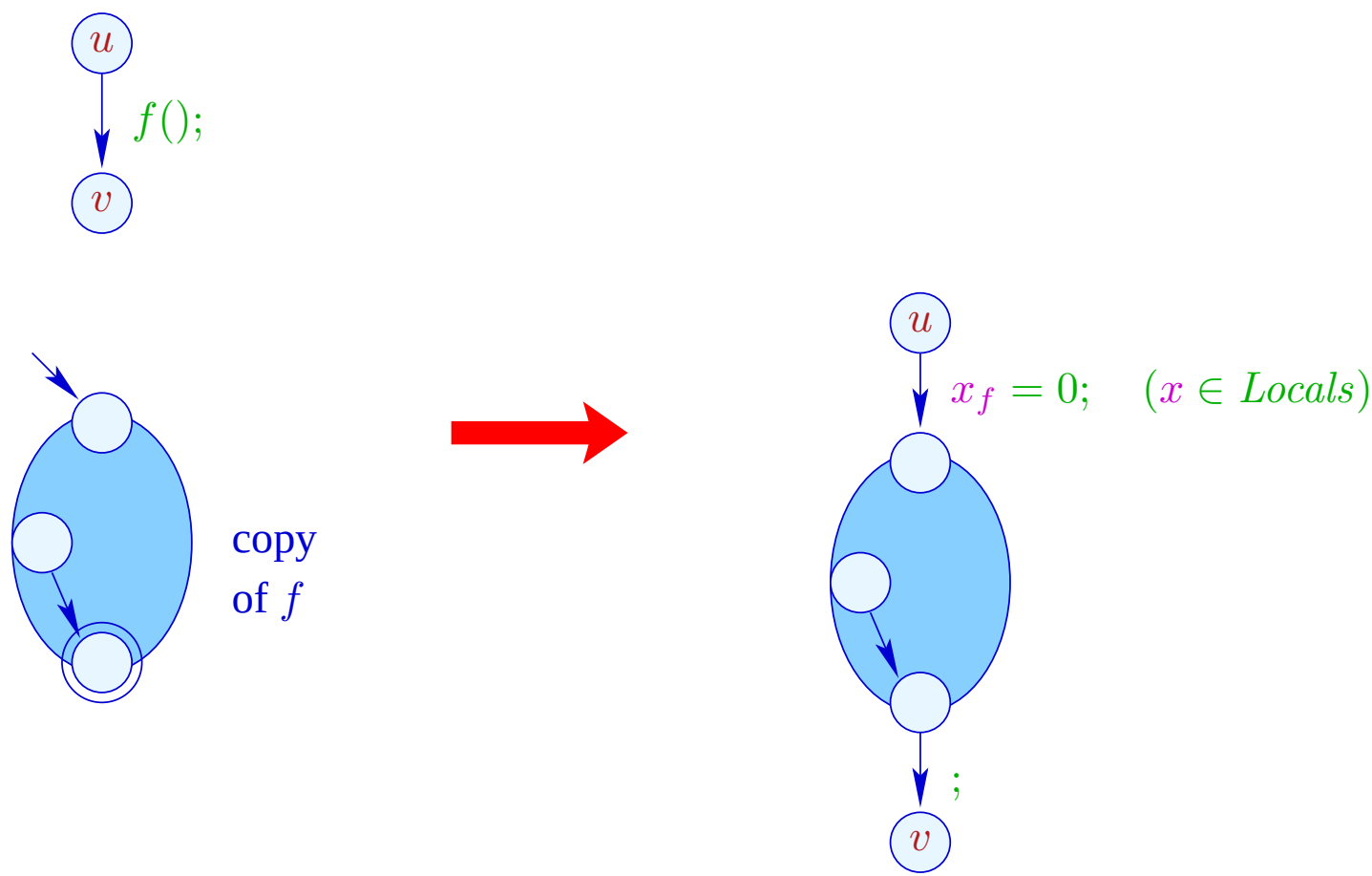
- The nodes are the procedures.
- An edge connexts g with h, whenever the body of g contains a call of h.

Strategies for Inlining:

- Just copy nur leaf-procedures, i.e., procedures without further calls :-)
- Copy all non-recursive procedures!

... here, we consider just leaf-procedures ;-)

Transformation 9:



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. Idea: Elimination of Tail Recursion

$$
\begin{array}{l}
f\,()\ \{\quad \mathsf{int}\ \ b; \\
\qquad \mathsf{if}\ \ (a_2 \leq 1)\ \ \{\ \ \mathsf{ret} = a_1;\ \mathsf{goto}\ \_exit;\ \ \ \} \\
\qquad b = a_1 \cdot a_2; \\
\qquad a_2 = a_2 - 1; \\
\qquad a_1 = b; \\
\qquad f\,(); \\
\_exit\ \ : \\
\}
\end{array}
$$

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

$\Longrightarrow$ We may directly jump to the beginning :-)

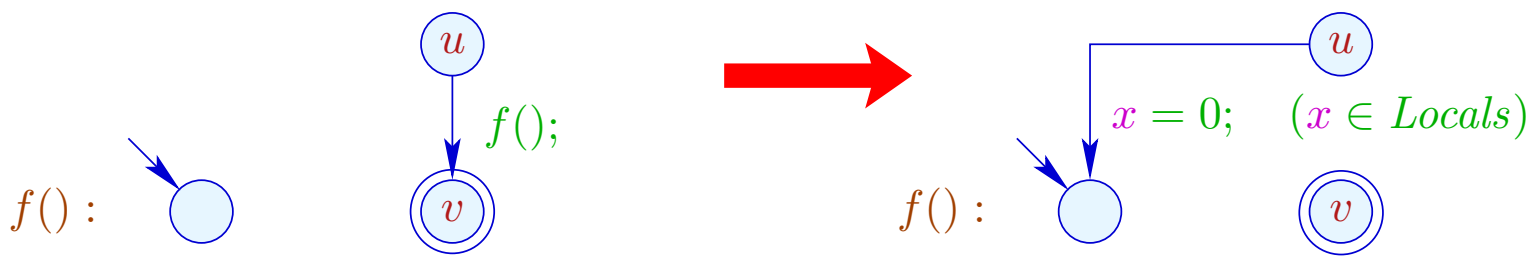
... after having reset the locals to 0.

... this yields in the Example:

```
$f$ () {   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 $\_f$;
 $\_exit$ :
}
```

$\Longrightarrow$ It works, since we have ruled out references to variables!

Transformation 11:



Warning:

- → This optimization is crucial for programming languages without iteration constructs !!!
- → Duplication of code is not necessary :-)
- → No variable renaming is necessary :-)
- → The optimization may also be profitable for non-recursive tail calls :-)
- → The corresponding code may contain jumps from the body of one procedure into the body of another ???

Background 4: 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:

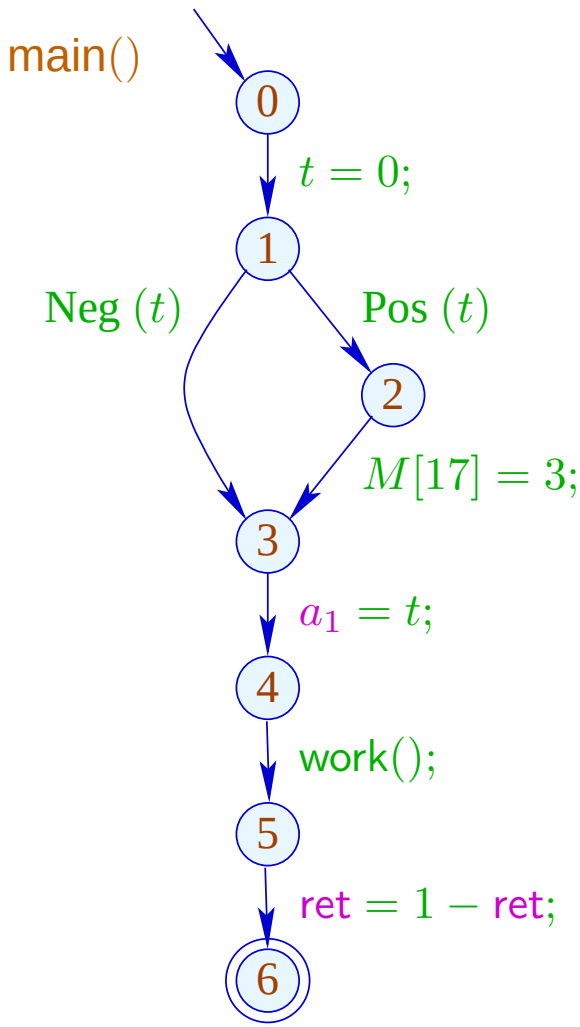
How can recursive programs be analyzed ???

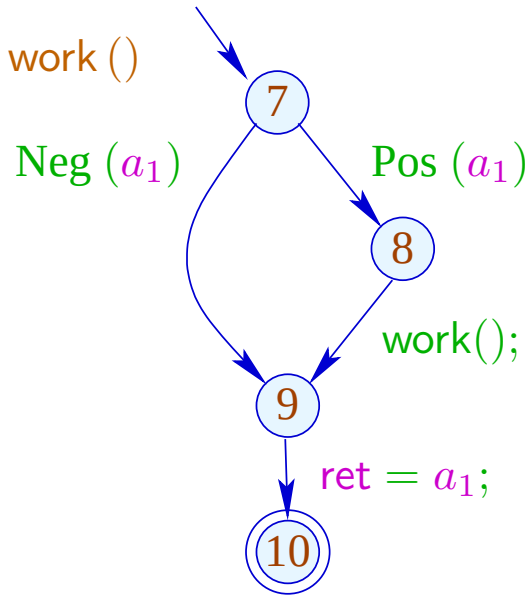
Example: Constant Propagation

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

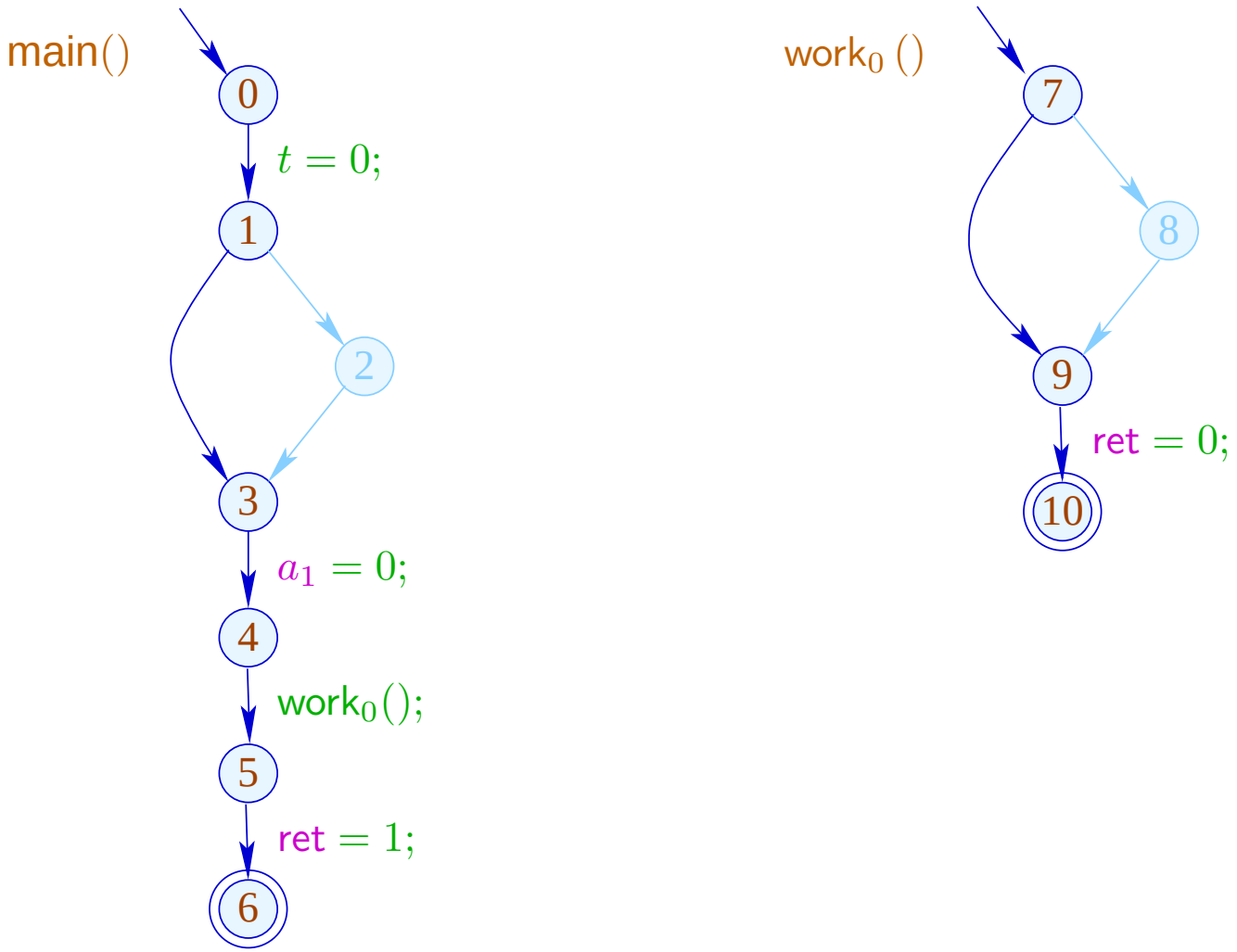
```
work() {
        if (a1) work();
        ret = a1;
}
```

Example: Constant Propagation





Example: Constant Propagation



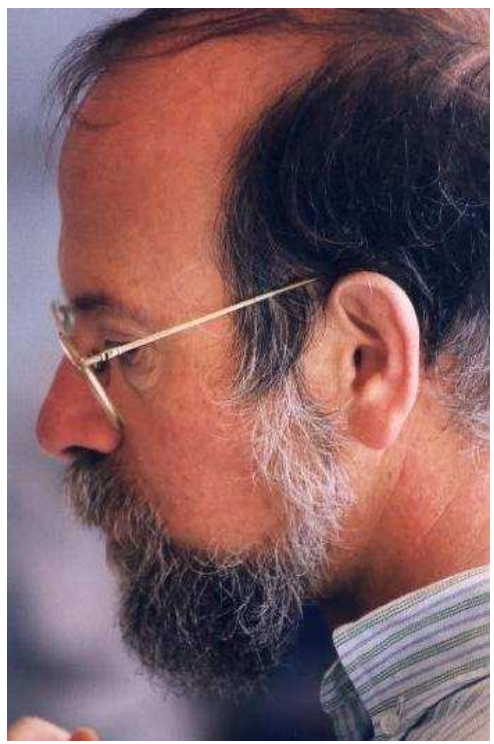
(1) Functional Approach:

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

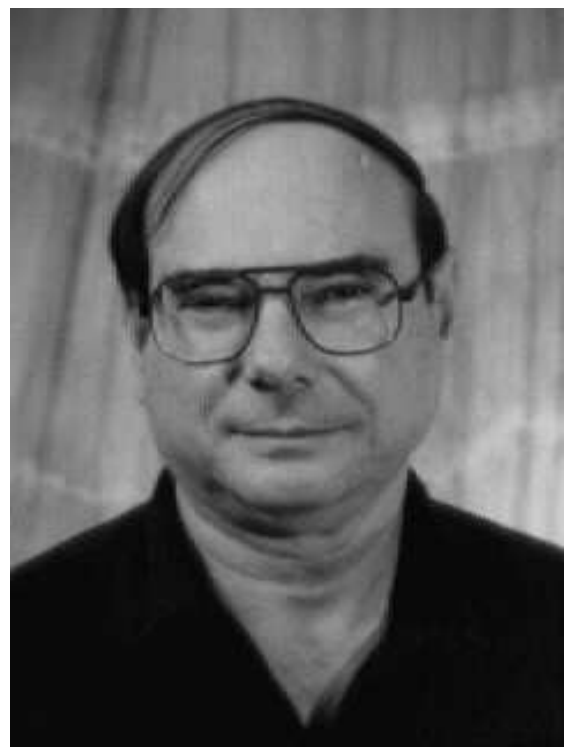
Idea:

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

$$[\![f]\!]^\sharp \;:\; \mathbb{D} \to \mathbb{D}$$



Micha Sharir, Tel Aviv University



Amir Pnueli, Weizmann Institute

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

$$\begin{array}{lcl} \mathsf{enter}^\sharp & : & \mathbb{D} \to \mathbb{D} \\ \mathsf{combine}^\sharp & : & \mathbb{D}^2 \to \mathbb{D} \end{array}$$

Then we define:

$$[\![k]\!]^\sharp\ D \quad = \quad \mathsf{combine}^\sharp\ (D, [\![f]\!]^\sharp\ (\mathsf{enter}^\sharp\ D))$$

... for Constant Propagation:

$$\mathbb{D} = (\mathit{Vars} \to \mathbb{Z}^{\top})_{\bot}$$

$$\mathsf{enter}^{\sharp}\ D = \begin{cases} \bot & \text{if} \quad D = \bot \\ D|_{\mathit{Globals}} \oplus \{x \mapsto 0 \mid x \in \mathit{Locals}\} & \text{otherwise} \end{cases}$$

$$\mathsf{combine}^{\sharp}\ (D_1, D_2) = \begin{cases} \bot & \text{if} \quad D_1 = \bot \vee D_2 = \bot \\ D_1|_{\mathit{Locals}} \oplus D_2|_{\mathit{Globals}} & \text{otherwise} \end{cases}$$

The effects $[\![f]\!]^\sharp$ then can be determined by a system of constraints over the complete lattice $\mathbb{D} \to \mathbb{D}$:

$$
\begin{array}{lll}
[\![v]\!]^\sharp & \sqsupseteq & \mathsf{Id} \qquad\qquad\qquad v \quad \text{entry point} \\
[\![v]\!]^\sharp & \sqsupseteq & [\![k]\!]^\sharp \circ [\![u]\!]^\sharp \qquad k = (u, \_, v) \quad \text{edge} \\
[\![f]\!]^\sharp & \sqsupseteq & [\![stop_f]\!]^\sharp \qquad\qquad stop_f \quad \text{end point of} \quad f
\end{array}
$$

$[\![v]\!]^\sharp$: $\mathbb{D} \to \mathbb{D}$ describes the effect of all prefixes of computation forests w of a procedure which lead from the entry point to v :-)