2.3 Procedures

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.

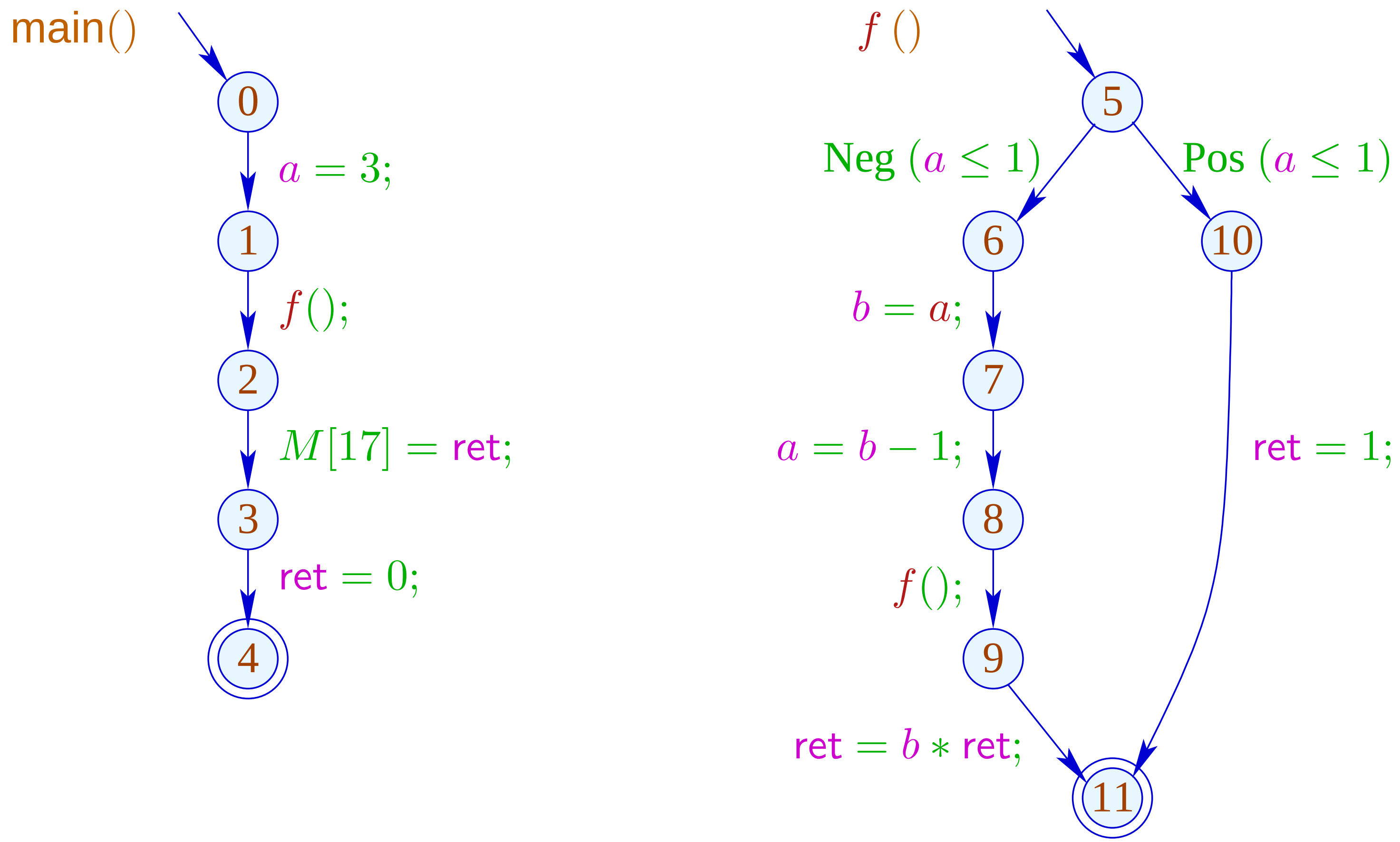
Program execution starts with the call of a procedure $\mathsf{main}\,()$.

Example:

```
int  a, ret;                      f ()  {
main ()  {                                int  b;
        a = 3;                            if (a ≤ 1) {ret = 1; goto exit; }
        f();                              b = a;
        M[17] = ret;                      a = b − 1;
        ret = 0;                          f();
}                                         ret = b · ret;
                                  exit :
                                  }
```

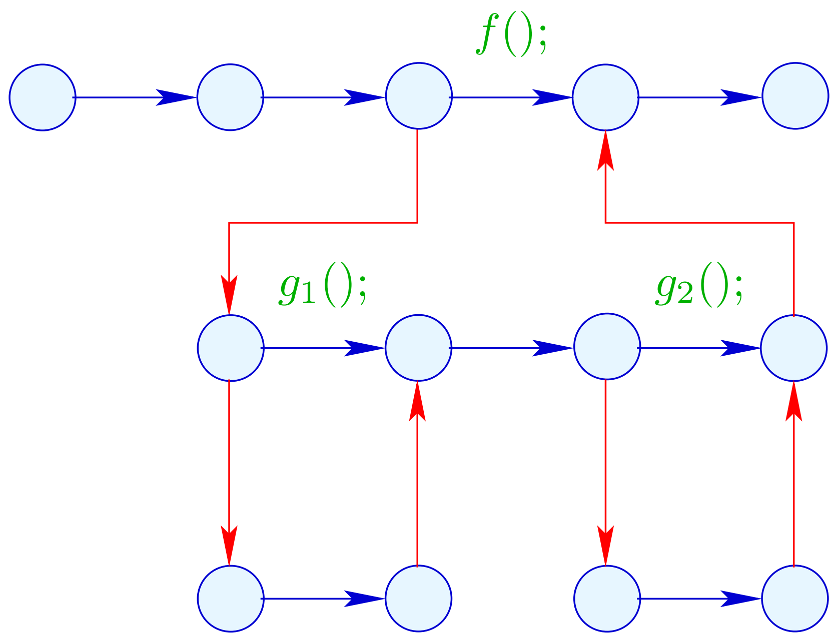
Such programs can be represented by a set of CFGs: one for each procedure ...

... in the Example:

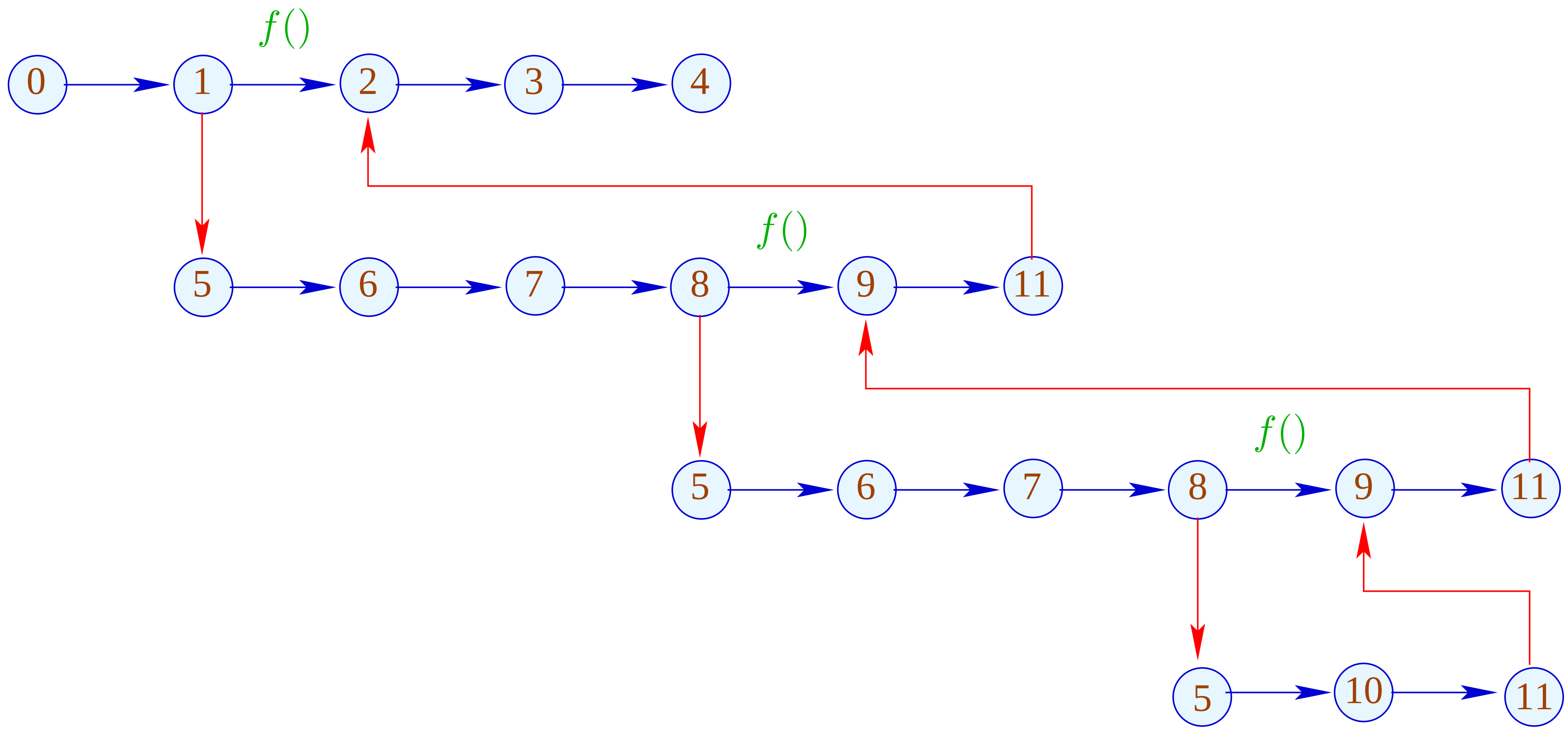


In order to optimize such programs, we require an extended operational semantics ;-)

Program executions are no longer paths, but forests:



... in the Example:



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:

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

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

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

- ... evaluate the computation forest inbetween:

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

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:

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

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

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

u_f entry point of f

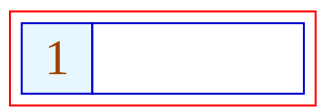
return:

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

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	

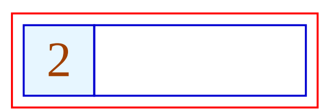
The call stack explicitly implements the DFS traversal through the computation forest :-)

... 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:

$$abs\ () \ \{$$
$$\quad a_2 = -a_1;$$
$$\quad max\ ();$$
$$\}$$

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

... yields:

$$abs\ ()\ \{$$

$$a_2 = -a_1;$$

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

$$\text{ret} = a_1;$$

$$\_exit\ :$$

$$\}$$

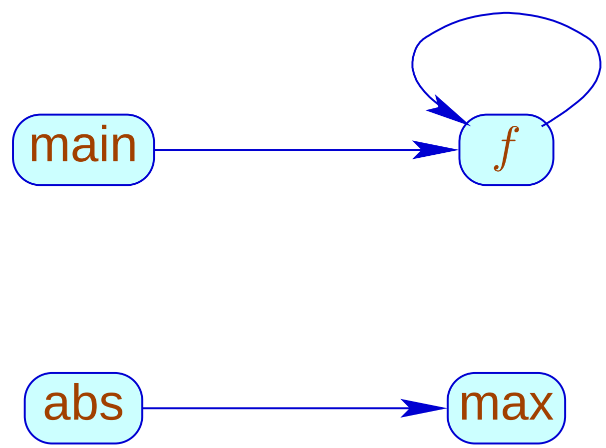
Problems:

- 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:

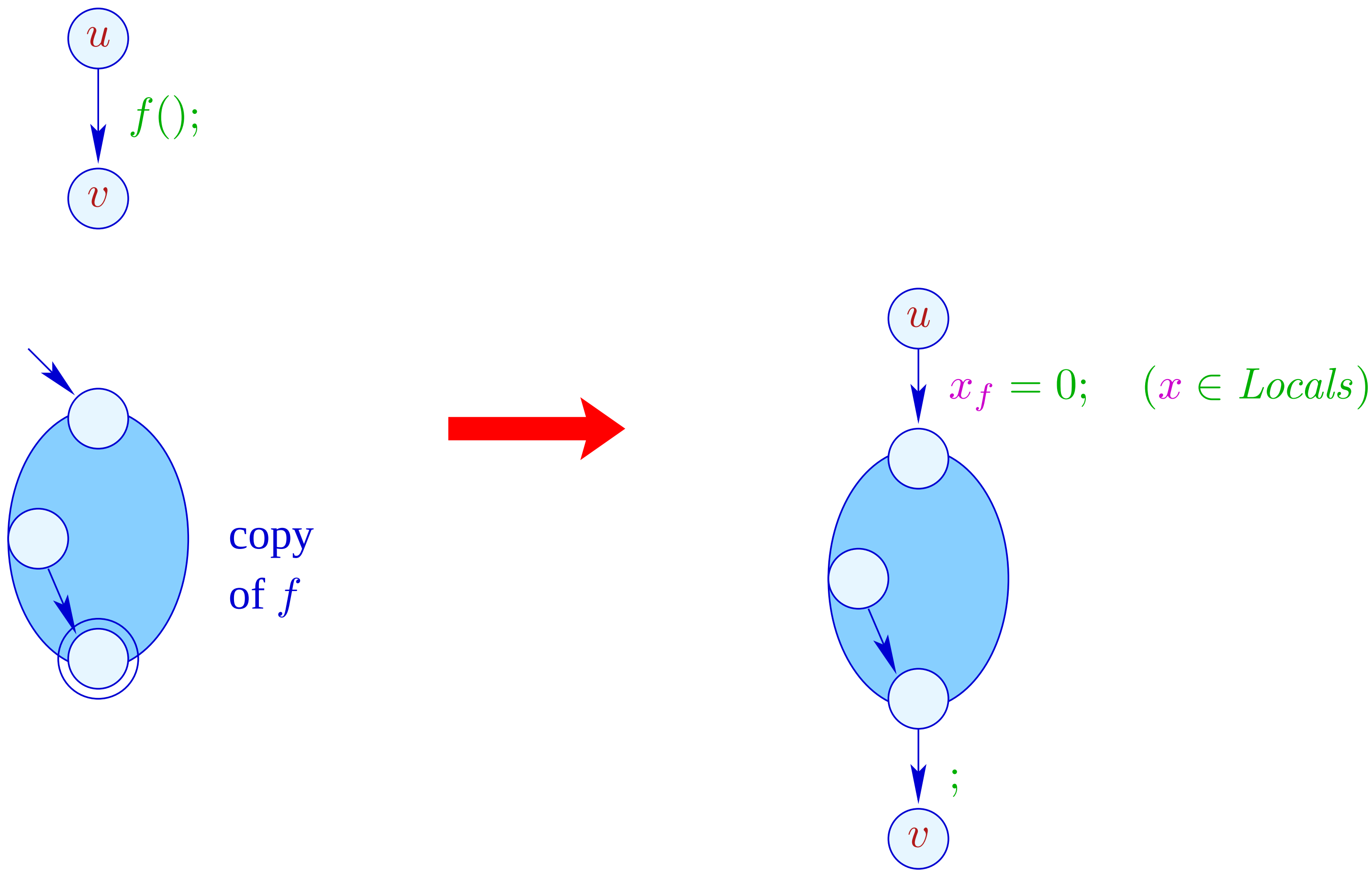
- The nodes are the procedures.
- An edge connexts g with h, whenever the body of g contains a call of h.

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

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

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

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

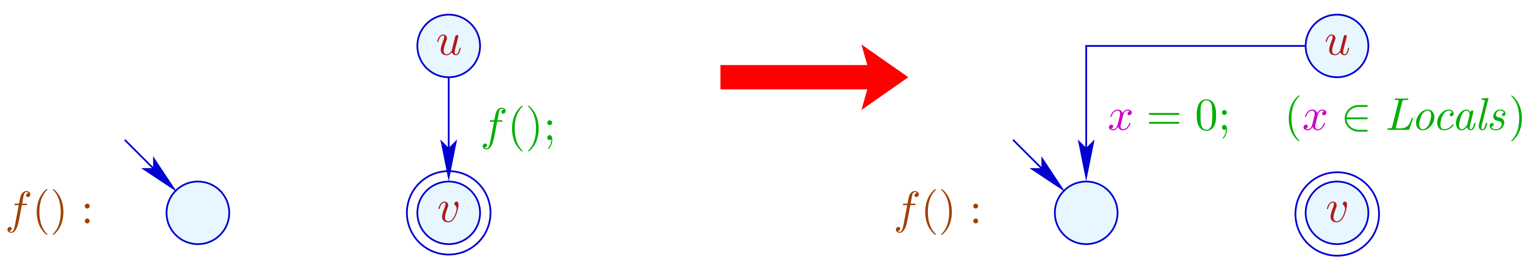
... after having reset the locals to 0.

... this yields in the Example:

$$
\begin{array}{ll}
f\,() \;\{ & \mathsf{int}\;\; b; \\
\_f: & \mathsf{if}\;\; (a_2 \leq 1) \;\; \{ \;\; \mathsf{ret} = a_1; \;\; \mathsf{goto}\; \_exit; \;\; \} \\
 & b = a_1 \cdot a_2; \\
 & a_2 = a_2 - 1; \\
 & a_1 = b; \\
 & b = 0; \;\; \mathsf{goto}\;\; \_f; \\
\_exit: & \\
\} &
\end{array}
$$

// 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 ???