... in the Example:
The function \([ . \) \] is extended to computation forests: \( w \):

\[
[w] : (\text{Vars} \rightarrow \mathbb{Z}) \times (\mathbb{N} \rightarrow \mathbb{Z}) \rightarrow (\text{Vars} \rightarrow \mathbb{Z}) \times (\mathbb{N} \rightarrow \mathbb{Z})
\]

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

- determine the initial values for the locals:
  \[
  \text{enter } \rho = \{x \mapsto 0 \mid x \in \text{Locals}\} \oplus (\rho|_{\text{Globals}})
  \]

- ... combine the new values for the globals with the old values for the locals:
  \[
  \text{combine } (\rho_1, \rho_2) = (\rho_1|_{\text{Locals}}) \oplus (\rho_2|_{\text{Globals}})
  \]

- ... evaluate the computation forest inbetween:
  \[
  [k \langle w \rangle] (\rho, \mu) = \text{let } (\rho_1, \mu_1) = [w] (\text{enter } \rho, \mu) \text{ in } (\text{combine } (\rho, \rho_1), \mu_1)
  \]
Warning:

- In general, $[w]$ is only partially defined :-)
- Dedicated global/local variables $a_i, b_i, \text{ret}$ 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:

\[
\begin{align*}
configuration &= stack \times store \\
store &= globals \times (\mathbb{N} \rightarrow \mathbb{Z}) \\
globals &= (\text{Globals} \rightarrow \mathbb{Z}) \\
stack &= frame \cdot frame^* \\
frame &= point \times locals \\
locals &= (\text{Locals} \rightarrow \mathbb{Z})
\end{align*}
\]

A \textit{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:

\textbf{call} \quad k = (u, f() ; v) : \\
\left( (u, \rho) \cdot \sigma, \langle \gamma, \mu \rangle \right) \implies \left( (u_f, \{x \rightarrow 0 \mid x \in Locals\}) \cdot (v, \rho) \cdot \sigma, \langle \gamma, \mu \rangle \right)

\text{\textit{u}}_f \quad \text{entry point of } f

\textbf{return:} \\
\left( (r_f, \_ \_ ) \cdot \sigma, \langle \gamma, \mu \rangle \right) \implies \left( \sigma, \langle \gamma, \mu \rangle \right)

\text{\textit{r}}_f \quad \text{return point of } f
The call stack explicitly implements the DFS traversal through the computation forest :-)
The call stack explicitly implements the DFS traversal through the computation forest  :-) 

... in the Example: 

<table>
<thead>
<tr>
<th>5</th>
<th>b ↦ 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:

<table>
<thead>
<tr>
<th>5</th>
<th>( b \mapsto 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>( b \mapsto 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:

<table>
<thead>
<tr>
<th>7</th>
<th>b \rightarrow 2</th>
</tr>
</thead>
<tbody>
<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:

<table>
<thead>
<tr>
<th>5</th>
<th>( b \mapsto 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>( b \mapsto 2 )</td>
</tr>
<tr>
<td>9</td>
<td>( b \mapsto 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:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>b ↦ 2</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:

```
11 b \rightarrow 2

9 b \rightarrow 3

2
```
The call stack explicitly implements the DFS traversal through the computation forest  :-)

... in the Example:

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

... in the Example:

```
11   b ↦ 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.

⇒ ... quite expensive !!!
1. Idea: Inlining

Copy the procedure body at every call site !!!

Example:

```c
abs () { 
    a2 = -a1; 
    max (); 
} 
max () { 
    if (a1 < a2) { ret = a2; goto _exit; } 
    ret = a1; 
} 
_exit : 
}```
... yields:

```c
abs () {
    a2 = \neg a1;
    if (a1 < a2) {
        ret = a2; goto _exit;
    }
    ret = a1;
    _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:

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

... here, we consider just leaf-procedures ;-)
Transformation 9:

\[ f(); \quad \text{copy of } f \]

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

```c
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:

```c
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 11:

$f() : v$  

$f() : x = 0; (x \in \text{Locals})$
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

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

2
\[ M[17] = 3; \]

3
\[ a_1 = t; \]

4
\[ \text{work}(); \]

5
\[ \text{work}(); \]

6
\[ \text{ret} = 1 - \text{ret}; \]

work()

7
\[ \text{Neg} (a_1) \]

8
\[ \text{Pos} (a_1) \]

9
\[ \text{work}(); \]

10
\[ \text{ret} = a_1; \]
Example: Constant Propagation

main()

\( t = 0; \)

\( a_1 = 0; \)

\( \text{work}_0(); \)

\( \text{ret} = 1; \)

work\(_0\) ()

\( \text{ret} = 0; \)
(1) **Functional Approach:**

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

**Idea:**

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

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

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

Then we define:

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

\[ D = (\text{Vars} \rightarrow \mathbb{Z}^\top)_\bot \]

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

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

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

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