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: \( w : \)

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

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

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

- ... 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:
  
  \[
  \llbracket k \langle w \rangle \rrbracket (\rho, \mu) = \text{let } (\rho_1, \mu_1) = \llbracket w \rrbracket (\text{enter } \rho, \mu) \text{ in } \text{combine } (\rho, \rho_1), \mu_1)
  \]
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:

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

A \text{ frame} specifies the local state of computation inside a procedure call :-)

The \text{leftmost} frame corresponds to the current call.
Computation steps refer to the current call  :-) 

The novel kinds of steps:

call  \[k = (u, f(); v) :\]
\[
(u, \rho) \cdot \sigma, \langle \gamma, \mu \rangle \rightarrow (u_f, \{x \rightarrow 0 \mid x \in \text{Locals}\}) \cdot (v, \rho) \cdot \sigma, \langle \gamma, \mu \rangle
\]
\[u_f \quad \text{entry point of } f\]

return:
\[
(r_f, _) \cdot \sigma, \langle \gamma, \mu \rangle \rightarrow \sigma, \langle \gamma, \mu \rangle
\]
\[r_f \quad \text{return point of } f\]
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:

<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:
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 ↦ 2</th>
</tr>
</thead>
<tbody>
<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:

<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:

<table>
<thead>
<tr>
<th>11</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:

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

... in the Example:

```
9  b ↦ 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:
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:

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

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

```
_exit :
}
```
... yields:

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

```
539
```
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:

\[ f(); \]

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

\[
\begin{align*}
f() & \quad \{ \text{ int } b; \\
& \quad \text{ if } (a_2 \leq 1) \quad \{ \text{ ret } = a_1; \ \text{ goto } \_\text{exit}; \ \} \\
& \quad b = a_1 \cdot a_2; \\
& \quad a_2 = a_2 - 1; \\
& \quad a_1 = b; \\
& \quad f(); \\
& \_\text{exit} : \\
& \}\end{align*}
\]

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

\[=\Rightarrow \text{ We may directly jump to the beginning } \] \(\Rightarrow\)

... after having reset the locals to 0.
... this yields in the Example:

```c
f () {  int b;
    _f :  if (a_2 ≤ 1) { ret = a_1; goto _exit; }
        b = a_1 · a_2;
        a_2 = a_2 − 1;
        a_1 = b;
        b = 0;  goto _f;
    _exit :
}
```

// It works, since we have ruled out references to variables!
Transformation 11:

\[ f() : v \Rightarrow f() : \]

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

Pos (t)

2

M[17] = 3;

Neg (t)

3

a₁ = t;

4

work();

5

ret = 1 - ret;

6

work ()

7

Neg (a₁)

8

Pos (a₁)

9

work();

10

ret = a₁;
Example: Constant Propagation

main()

0
\[ t = 0; \]

1

2
\[ a_1 = 0; \]

3

4
\[ \text{work}_0(); \]

5
\[ \text{ret} = 1; \]

6

\[ \text{ret} = 0; \]

work_0 ()

7

8

9

10
(1) **Functional Approach:**

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

**Idea:**

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

$$\left[ f \right]^{\#} : \mathbb{D} \rightarrow \mathbb{D}$$
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:

\[
\text{enter}^\# : \mathbb{D} \to \mathbb{D} \\
\text{combine}^\# : \mathbb{D}^2 \to \mathbb{D}
\]

Then we define:

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

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

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

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

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

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

- How can we represent functions \( f : \mathbb{D} \to \mathbb{D} \)??
- If \( \#\mathbb{D} = \infty \), then \( \mathbb{D} \to \mathbb{D} \) has infinite strictly increasing chains  

Simplification:  

Copy-Constants

\[ \rightarrow \quad \text{Conditions are interpreted as } ; \quad :-) \]
\[ \rightarrow \quad \text{Only assignments } x = e; \quad \text{with } e \in Vars \cup \mathbb{Z} \quad \text{are treated exactly} \quad :-) \]
Observation:

→ The effects of assignments are:

\[ [x = e;] D' = \begin{cases} 
D \oplus \{x \mapsto c\} & \text{if } e = c \in \mathbb{Z} \\
D \oplus \{x \mapsto (D y)\} & \text{if } e = y \in \text{Vars} \\
D \oplus \{x \mapsto \top\} & \text{otherwise}
\end{cases} \]

→ 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

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

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