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:

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

Copy of \( f \)
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 :-}

543
2. Idea: Elimination of Tail Recursion

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

After the procedure call, nothing in the body remains to be done.\[\Rightarrow \text{We may \textbf{directly} jump to the beginning} \quad \therefore\]

... 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:

\[ f() : \]

\[ f() : \]

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

2
  M[17] = 3;

3
  a1 = t;

4
  work();

5
  ret = 1 - ret;

6

work()
7
  Neg(a1)
  Pos(a1)

8

9
  work();

10
  ret = a1;
```
Example: Constant Propagation

main()

0
  t = 0;

1

2

3
  a1 = 0;

4
  work0();

5
  ret = 1;

6

work0()

7

8
  ret = 0;

9

10
(1) Functional Approach:

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

Idea:

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

\[
[f]^\#: \mathcal{D} \rightarrow \mathcal{D}
\]
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:

- \( \text{enter}^\# : D \rightarrow D \)
- \( \text{combine}^\# : D^2 \rightarrow D \)

Then we define:

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

\[
\begin{align*}
\mathcal{D} & = (\text{Vars} \rightarrow \mathbb{Z}^+)_{\perp} \\
\text{enter}^\# D & = \begin{cases} 
\perp & \text{if } D = \perp \\
D|_{\text{Globals}} \oplus \{ x \mapsto 0 \mid x \in \text{Locals} \} & \text{otherwise}
\end{cases}
\end{align*}
\]

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

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

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

556
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

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

→ The effects of assignments are:

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

→ Let \( V \) denote the (finite !!!) set of constant right-hand sides. Then variables may only take values from \( V^\top :)) \)

→ 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 !!!
Improvement:

→ Not all functions from \( D_f \rightarrow D_f \) will occur \( \text{:-)} \)
→ All occurring functions \( \lambda D. \bot \neq M \) are of the form:

\[
M = \{ x \mapsto (b_x \sqcup \bigsqcup_{y \in I_x} y) \mid x \in Vars \} \quad \text{where:}
\]
\[
M D = \{ x \mapsto (b_x \sqcup \bigsqcup_{y \in I_x} D y) \mid x \in Vars \} \quad \text{für} \quad D \neq \bot
\]

→ Let \( M \) denote the set of all these functions. Then for \( M_1, M_2 \in M \) \( (M_1 \neq \lambda D. \bot \neq M_2) \):

\[
(M_1 \sqcup M_2) x = (M_1 x) \sqcup (M_2 x)
\]

→ For \( k = \#Vars \), \( M \) has height \( \mathcal{O}(k^2) \) \( \text{:-)} \)