2. Subproblem: Linearization

After optimization, the CFG must again be brought into a linearly arranged arrangement of instructions :-)

Warning:

Not every linearization is equally efficient !!!

Example:

Bad: The loop body is jumped into :-(

Example:

$\begin{align*}
0: & \\
1: & \text{ if } (c_1) \text{ goto } 2; \\
4: & \text{ halt} \\
2: & \text{ Rumpf} \\
3: & \text{ if } (c_2) \text{ goto } 4; \\
& \text{ goto } 1;
\end{align*}$

$\begin{align*}
0: & \\
1: & \text{ if } (c_2) \text{ goto } 4; \\
2: & \text{ Rumpf} \\
3: & \text{ if } (c_2) \text{ goto } 1; \\
4: & \text{ halt}
\end{align*}$
Idea:

- Assign to each node a `temperature`!
- Always jumps to
  1. nodes which have already been handled;
  2. colder nodes.
- `Temperature` $\approx$ nesting depth

For the computation, we use the pre-dominator tree and strongly connected components ...

Example:

```
0:  if (v1) goto 4;
1:   Rumpf
2:   if (v2) goto 1;
3:   halt
```

... in the Example:
More Complicated Example:

Our definition of \textit{Loop} implies that (detected) loops are necessarily nested \( \Rightarrow \)

Is is also meaningful for do-while-loops with breaks ...

More Complicated Example:

Our definition of \textit{Loop} implies that (detected) loops are necessarily nested \( \Rightarrow \)

Is is also meaningful for do-while-loops with breaks ...
Our definition of loop implies that (detected) loops are necessarily nested :-)
Is is also meaningful for do-while-loops with breaks ...

Summary: The Approach

1. For every node, determine a temperature:
2. Pre-order-DFS over the CFG:
   -> If an edge leads to a node we already have generated code for, then we insert a jump.
   -> If a node has two successors with different temperature, then we insert a jump to the colder of the two.
   -> If both successors are equally warm, then it does not matter :-)
Example:
\[ f(e_1, e_2) \]

```
int a, ret;
main () {
    a = 3;
f();
    M[17] = ret;
    ret = 0;
}
```

```
f () {
    int b;
    if (a \leq 1) {ret = 1; goto exit; }
    b = a;
f();
    a = a - 1;
    ret = b \times ret;
}
```

exit:
```
}
```

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

... in the Example:
```
main()
0
1
f();
M[17] = ret;
ret = 0;
```
```
f()
5
6
Neg(a \leq 1)
7
8
Pos(a \leq 1)
```
```

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

Program executions are no longer paths, but forests:
The function \([\cdot]\) 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:
  \[
  [k \langle w \rangle] (\rho, \mu) = \begin{cases} 
    \text{let } (\rho_1, \mu_1) = [w] \langle \text{enter } \rho, \mu \rangle 
    & \text{in } \langle \text{combine } (\rho, \rho_1), \mu_1 \rangle 
  \end{cases}
  \]

The function \([\cdot]\) 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:
  \[
  [k \langle w \rangle] (\rho, \mu) = \begin{cases} 
    \text{let } (\rho_1, \mu_1) = [w] \langle \text{enter } \rho, \mu \rangle 
    & \text{in } \langle \text{combine } (\rho, \rho_1), \mu_1 \rangle 
  \end{cases}
  \]
The function \( [] \) is extended to computation forests: \( w \):

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

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

- determine the initial values for the locals:

\[
\text{enter } \rho = \{ \sigma \mapsto 0 \mid x \in \text{Locals} \} \oplus (\rho|_{\text{Globals}})
\]

- ... 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:

\[
[k \, \langle w \rangle] (\rho, \mu) = \begin{cases} (\rho_1, \mu_1) = [w] (\text{enter } \rho, \mu) \\
\text{in } \langle \text{combine } (\rho, \rho_1), \mu_1 \rangle
\end{cases}
\]

**Warning:**

- In general, \( [w] \) is only partially defined \( :-) \)
- Dedicated global/local variables \( a_0, b_1, \text{ret} \) 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:**

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

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:

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

\[
\begin{array}{c}
\text{return: } \quad \{ (u, \rho) \cdot \sigma, (\gamma, \mu) \} \Rightarrow (\sigma, (\gamma, \mu)) \quad \text{return point of } f
\end{array}
\]
The call stack explicitly implements the DFS traversal through the computation forest.

... in the Example:

```
1
```

```
5 b → 0
2
```

The call stack explicitly implements the DFS traversal through the computation forest.

... in the Example:

```
1
```

```
7 b → 3
2
```

The call stack explicitly implements the DFS traversal through the computation forest.

... in the Example:

```
1
```

```
5 b → 0
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:

In order to optimize such programs, we require an extended operational semantics.

Program executions are no longer paths, but forests.

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 !!!

The call stack explicitly implements the DFS traversal through the computation forest 😊

... in the Example:

1. **Idea:** Inlining

Copy the procedure body at every call site !!!

Example:

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

_exit : 
}
```
... yields:

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

1. Idea: **Inlining**

Copy the procedure body at every call site !!!

Example:

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

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

... yields:

```c
abs () {
  a2 = -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:
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

```java
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.