Script generated by TTT

Title: Seidl: Programmoptimierung (05.12.2012)

Date: Wed Dec 05 09:31:53 CET 2012

Duration: 89:46 min

Pages: 49

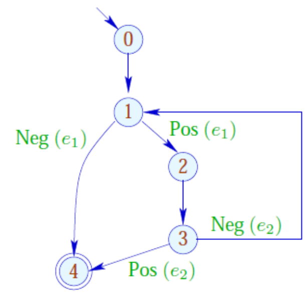
2. Subproblem: Linearization

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

Warning:

Not every linearization is equally efficient !!!

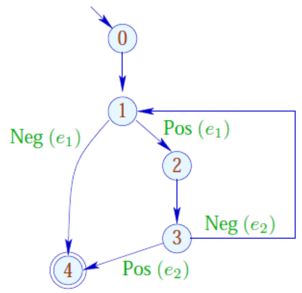
Example:



0:
1: if (e_1) goto 2;
4: halt
2: Rumpf
3: if (e_2) goto 4;
goto 1;

Bad: The loop body is jumped into :-(

Example:



0:
1: if $(!e_1)$ goto 4;
2: Rumpf
3: if $(!e_2)$ goto 1;
4: halt

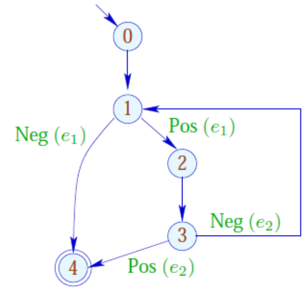
// better cache behavior :-)

Idea:

- Assign to each node a temperature!
- always jumps to
  - (1) nodes which have already been handled;
  - (2) colder nodes.
- Temperature ≈ nesting-depth

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

Example:



0:
1: if (!e_1) goto 4;
2: Rumpf
3: if (!e_2) goto 1;
4: halt

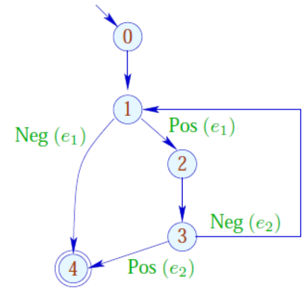
// better cache behavior :-)

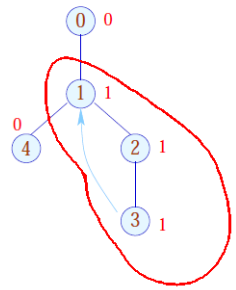
Idea:

- Assign to each node a temperature!
- always jumps to
  - (1) nodes which have already been handled;
  - (2) colder nodes.
- Temperature ≈ nesting-depth

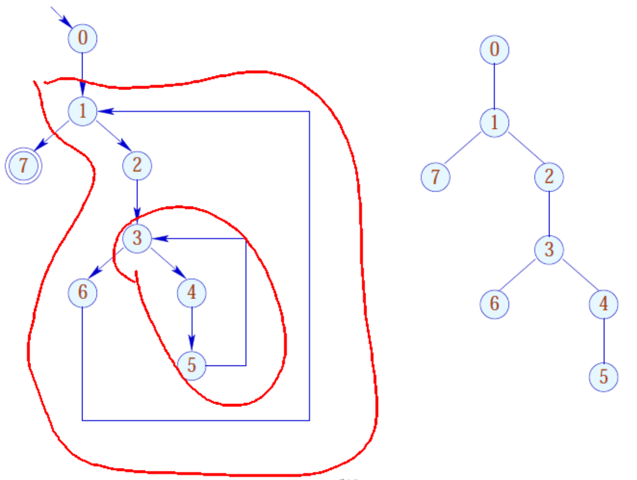
For the computation, we use the pre-dominator tree and strongly connected components ...

... in the Example:

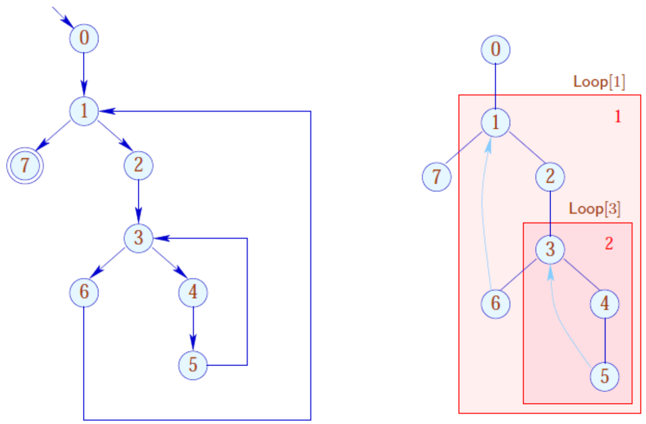




More Complicated Example:

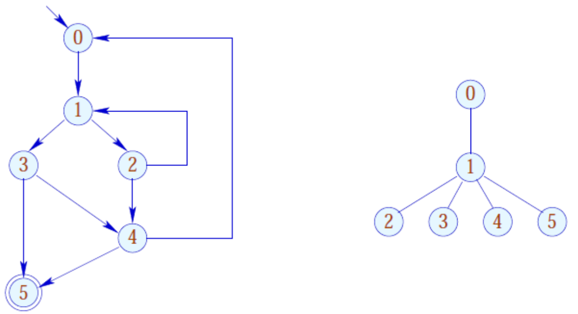


More Complicated Example:



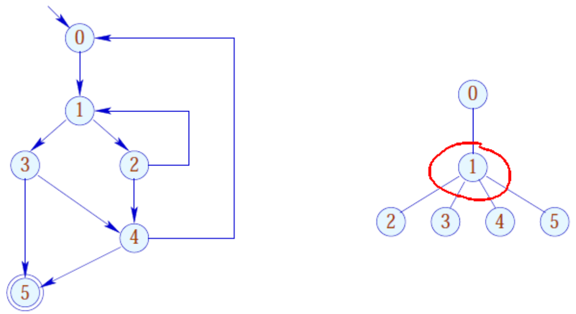
Our definition of Loop implies that (detected) loops are necessarily nested :-)

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



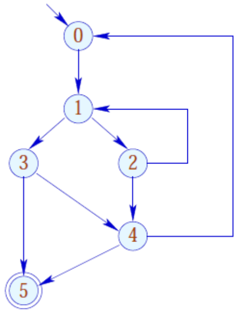
Our definition of Loop implies that (detected) loops are necessarily nested :-)

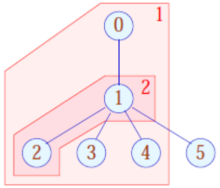
Is is also meaningful for do-while-loops with breaks ...



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 ;-)

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 ;-)

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 main () .

Example:

$f(e_1, e_2)$

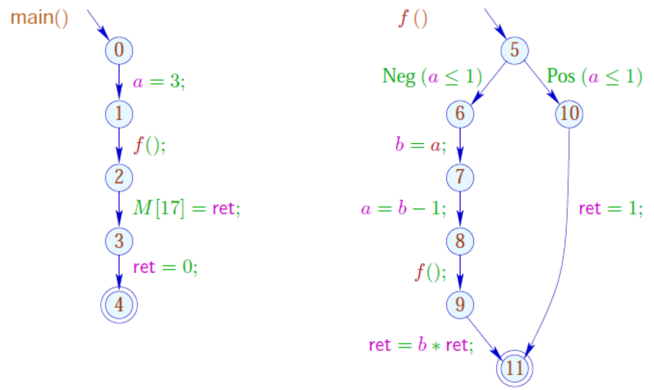
$x_1 \quad x_2$

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

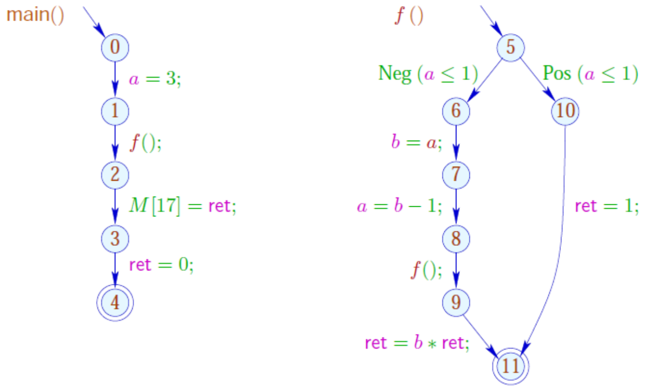
```
f () {
    int b;
    if (a ≤ 1) {ret = 1; goto exit; }
    b = a;
    a = b − 1;
    f();
    ret = b ∗ ret;
exit :
}
```

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

... in the Example:

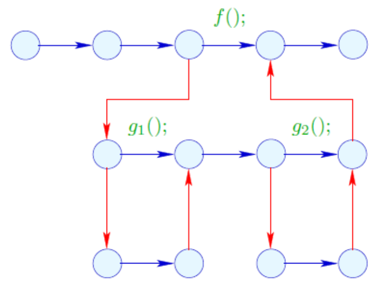


... in the Example:

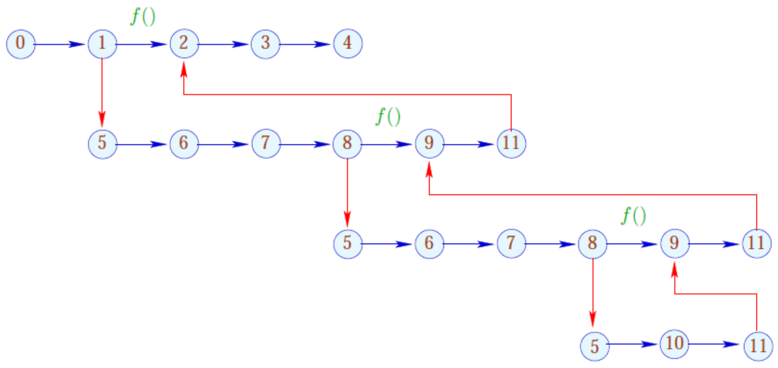


In order to optimize such programs, we require an extended operational semantics ;-)

Program executions are no longer paths, but forests:



... in the Example:



The function $[\![.]\!]$ is extended to computation forests: w :

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

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

- determine the initial values for the locals:

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

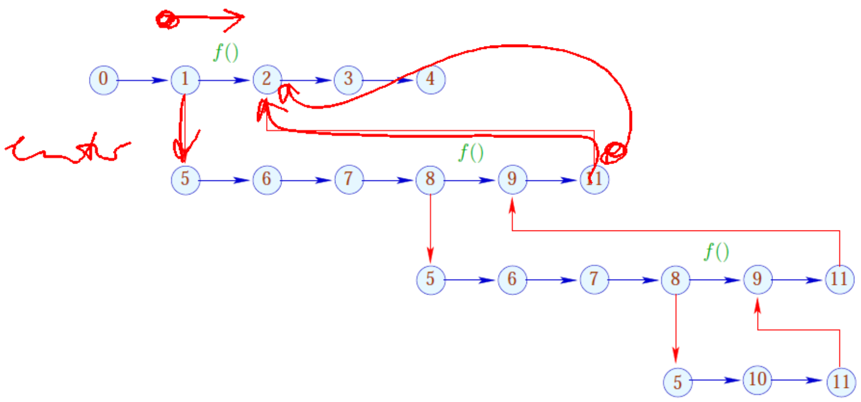
- ... combine the new values for the globals with the old values for the locals:

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

- ... evaluate the computation forest inbetween:

$$[\![k\ \langle w \rangle]\!]\ (\rho, \mu) \quad = \quad \text{let } (\rho_1, \mu_1) = [\![w]\!]\ (\text{enter } \rho, \mu) \quad \text{in } (\text{combine } (\rho, \rho_1), \mu_1)$$

... in the Example:



The function $[\![.]\!]$ is extended to computation forests: w :

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

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

POISON

- determine the initial values for the locals:

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

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

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

- ... evaluate the computation forest inbetween:

$$[\![k\ \langle w \rangle]\!]\ (\rho, \mu) \quad = \quad \text{let } (\rho_1, \mu_1) = [\![w]\!]\ (\text{enter } \rho, \mu) \quad \text{in } (\text{combine } (\rho, \rho_1), \mu_1)$$

The function $[\![.]\!]$ is extended to computation forests: w :

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

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

- determine the initial values for the locals:

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

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

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

- ... evaluate the computation forest inbetween:

$$[\![k\ \langle w \rangle]\!]\ (\rho, \mu) \quad = \quad \text{let}\ (\rho_1, \mu_1) = [\![w]\!]\ (\text{enter}\ \rho, \mu)\ \ \text{in}\ \ (\text{combine}\ (\rho, \rho_1), \mu_1)$$

Warning:

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

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

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

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

u_f entry point of f

return:

$$((r_f, \_) \cdot \sigma, \langle \gamma, \mu \rangle) \Longrightarrow (\sigma, \langle \gamma, \mu \rangle)$$

r_f return point of f

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

... in the Example:

1	

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

... in the Example:

1	

525

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	

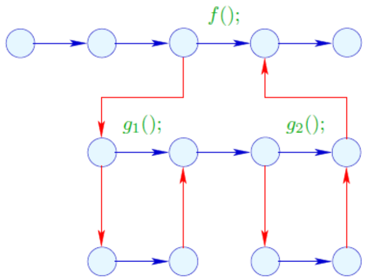
The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:

5	$b \mapsto 0$
9	$b \mapsto 3$
2	

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:

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:

5	$b \mapsto 0$
9	$b \mapsto 2$
9	$b \mapsto 3$
2	

530

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:

2	

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

... yields:

```
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:

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


... yields:

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

539

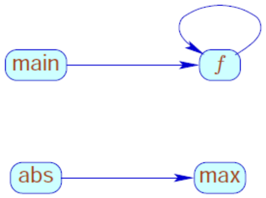
Problems:

- 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:

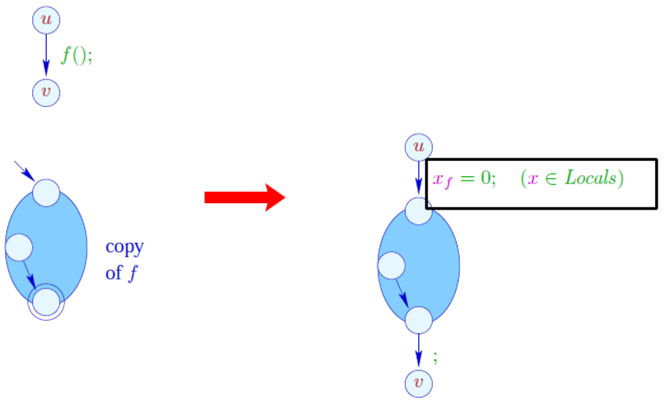
- The nodes are the procedures.
- An edge connexts g with h, whenever the body of g contains a call of h.

Strategies for Inlining:

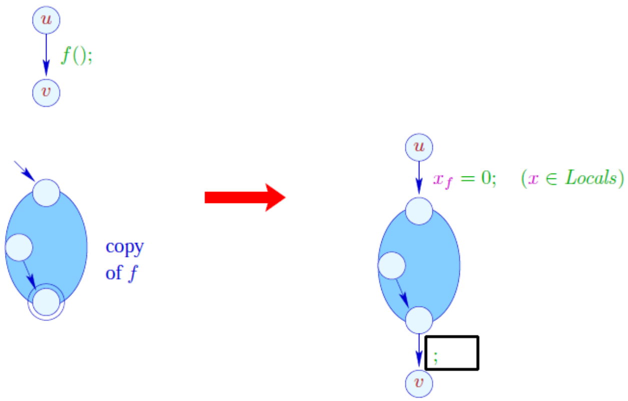
- Just copy nur leaf-procedures, i.e., procedures without further calls :-)
- Copy all non-recursive procedures!

... here, we consider just leaf-procedures ;-)

Transformation 9:



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

$f\,()\ \{\ \ \mathsf{int}\ \ b;$

$\quad \mathsf{if}\ \ (a_2 \leq 1)\ \ \{\ \ \mathsf{ret} = a_1;\ \ \mathsf{goto}\ \_exit;\ \ \}$

$\quad b = a_1 \cdot a_2;$

$\quad a_2 = a_2 - 1;$

$\quad a_1 = b;$

$\quad f\,();$

$\_exit\ :$

$\}$

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.