Script generated by TTT

Title: Seidl: Virtual_Machines (17.07.2012)
Date: Tue Jul 17 14:01:41 CEST 2012
Duration: 75:57 min
Pages: 38

34 Last Call Optimization

Consider the app predicate from the beginnning:

$$\begin{array}{lll} \mathrm{app}(X,Y,Z) & \leftarrow & X = [\,],\ Y = Z \\ \mathrm{app}(X,Y,Z) & \leftarrow & X = [H|X'],\ Z = [H|Z'],\ \mathrm{app}(X',Y,Z') \end{array}$$

We observe:

- The recursive call occurs in the last goal of the clause.
- Such a goal is called last call.

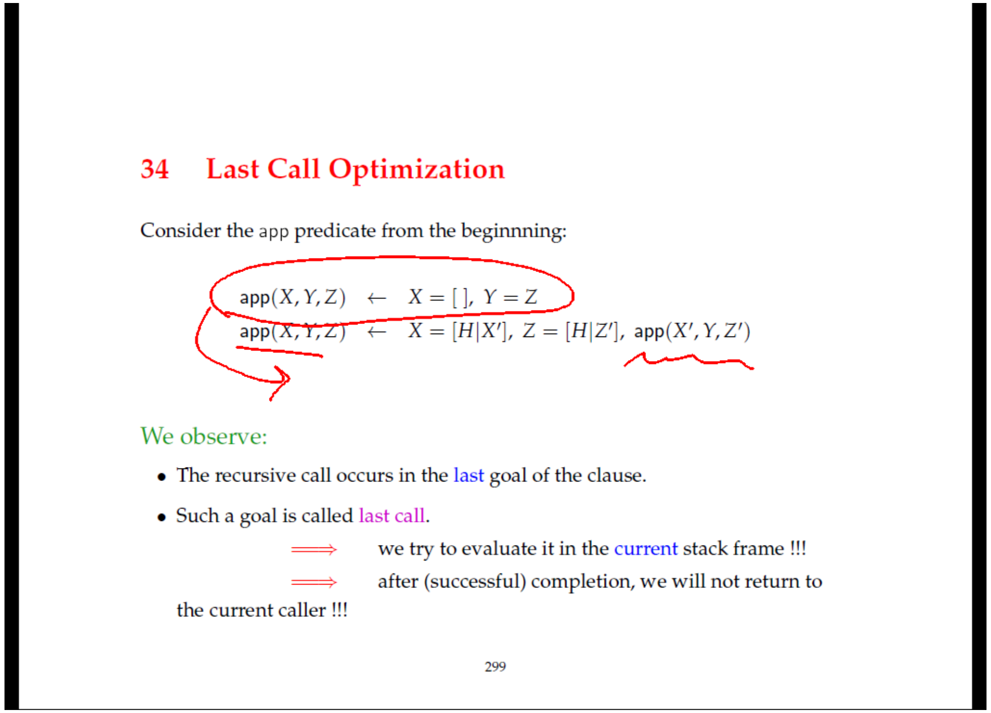
  $\Longrightarrow$ we try to evaluate it in the current stack frame !!!

  $\Longrightarrow$ after (successful) completion, we will not return to the current caller !!!

o Programmiersprachen

o Program Opt.

IDP with
Frau Vogel-Heuser



34 Last Call Optimization

Consider the app predicate from the beginnning:

$$\begin{array}{lll} \mathrm{app}(X,Y,Z) & \leftarrow & X = [\,],\ Y = Z \\ \mathrm{app}(X,Y,Z) & \leftarrow & X = [H|X'],\ Z = [H|Z'],\ \mathrm{app}(X',Y,Z') \end{array}$$

We observe:

- The recursive call occurs in the last goal of the clause.
- Such a goal is called last call.

  $\Longrightarrow$ we try to evaluate it in the current stack frame !!!

  $\Longrightarrow$ after (successful) completion, we will not return to the current caller !!!

Consider a clause r: $p(X_1,\ldots,X_k) \leftarrow g_1,\ldots,g_n$ with m locals where $g_n \equiv q(t_1,\ldots,t_h)$. The interplay between code_C and code_G:

$$\begin{array}{rcrl} \text{code}_C\ r & = & & \text{pushenv m} \\ & & & \text{code}_G\ g_1\ \rho \\ & & & \ldots \\ & & & \text{code}_G\ g_{n-1}\ \rho \\ & & & \text{mark B} \\ & & & \text{code}_A\ t_1\ \rho \\ & & & \ldots \\ & & & \text{code}_A\ t_h\ \rho \\ & & & \text{call q/h} \\ & & \text{B}: & \text{popenv} \end{array}$$

Replacement:

mark B ⟹ lastmark

call q/h; popenv ⟹ lastcall q/h m

Consider a clause r: $p(X_1,\ldots,X_k) \leftarrow g_1,\ldots,g_n$ with m locals where $g_n \equiv q(t_1,\ldots,t_h)$. The interplay between code_C and code_G:

$$\begin{array}{rcl} \text{code}_C\ r & = & \text{pushenv m} \\ & & \text{code}_G\ g_1\ \rho \\ & & \ldots \\ & & \text{code}_G\ g_{n-1}\ \rho \\ & & \text{lastmark} \\ & & \text{code}_A\ t_1\ \rho \\ & & \ldots \\ & & \text{code}_A\ t_h\ \rho \\ & & \text{lastcall q/h m} \end{array}$$

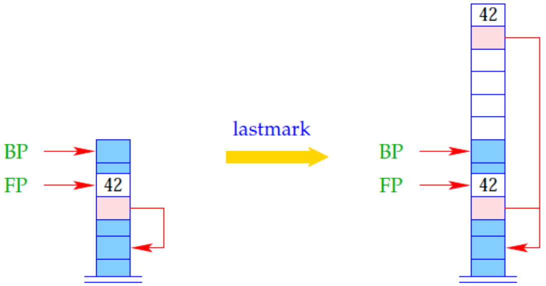
Replacement:

mark B ⟹ lastmark

call q/h; popenv ⟹ lastcall q/h m

If the current clause is not last or the $g_1,\ldots,g_{n-1}$ have created backtrack points, then FP ≤ BP :-)

Then lastmark creates a new frame but stores a reference to the predecessor:



```
if (FP ≤ BP) {
    SP = SP + 6;
    S[SP] = posCont; S[SP-1] = FPold;
}
```

If FP > BP then lastmark does nothing :-)

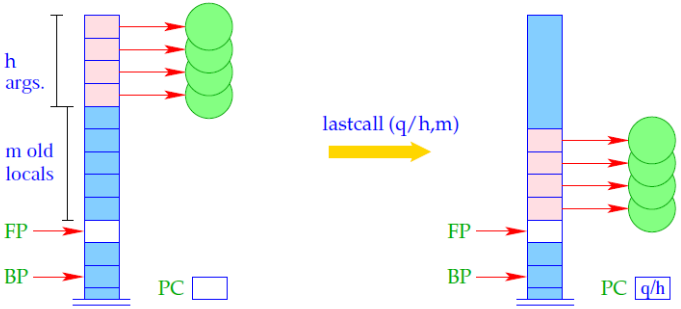
If FP ≤ BP, then lastcall q/h m behaves like a normal call q/h.

Otherwise, the current stack frame is re-used. This means that:

- the cells S[FP+1], S[FP+2], . . ., S[FP+h] receive the new values and
- q/h can be jumped to :-)

```
lastcall q/h m   =   if (FP ≤ BP)  call q/h;
                     else {
                        move m h;
                        jump q/h;
                     }
```

The difference between the old and the new addresses of the parameters m just equals the number of the local variables of the current clause :-))



Example:

Consider the clause:

$$a(X, Y) \leftarrow f(\bar{X}, X_1), a(\bar{X}_1, \bar{Y})$$

The last-call optimization for $\text{code}_C\ r$ yields:

```
                 mark A         A:   lastmark
   pushenv 3     putref 1            putref 3
                 putvar 3            putref 2
                 call f/2            lastcall a/2 3
```

Example:

Consider the clause:

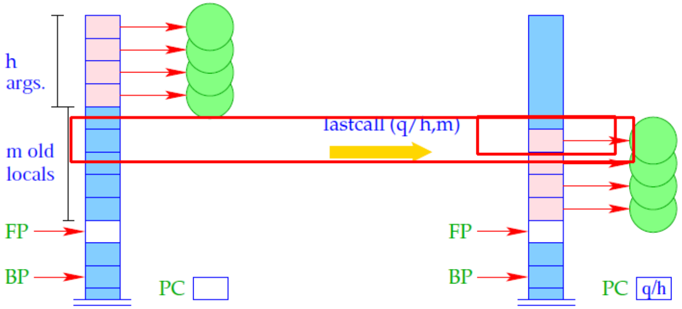
$$a(X, Y) \leftarrow f(\bar{X}, X_1), a(\bar{X}_1, \bar{Y})$$

The last-call optimization for $\text{code}_C\ r$ yields:

```
                 mark A         A:   lastmark
   pushenv 3     putref 1            putref 3
                 putvar 3            putref 2
                 call f/2            lastcall a/2 3
```

Note:

If the clause is last and the last literal is the only one, we can skip lastmark and can replace lastcall q/h m with the sequence move m n; jump p/n :-))



Consider a clause r: $p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_n$ with m locals where $g_n \equiv q(t_1, \ldots, t_h)$. The interplay between code_C and code_G:

$$\begin{array}{rcl} \text{code}_C\ r & = & \text{pushenv m} \\ & & \text{code}_G\ g_1\ \rho \\ & & \ldots \\ & & \text{code}_G\ g_{n-1}\ \rho \\ & & \text{mark B} \\ & & \text{code}_A\ t_1\ \rho \\ & & \ldots \\ & & \text{code}_A\ t_h\ \rho \\ & & \text{call q/h} \\ & \text{B :} & \text{popenv} \end{array}$$

Replacement:	mark B	$\Longrightarrow$	lastmark
	call q/h; popenv	$\Longrightarrow$	lastcall q/h m

---

35 Trimming of Stack Frames

Idea:

- Order local variables according to their life times;
- Pop the dead variables — if possible :-}

---

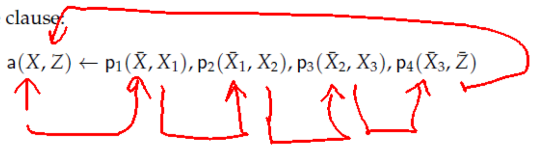
35 Trimming of Stack Frames

Idea:

- Order local variables according to their life times;
- Pop the dead variables — if possible :-}

Example:

Consider the clause:



$$a(X, Z) \leftarrow p_1(\bar{X}, X_1), p_2(\bar{X}_1, X_2), p_3(\bar{X}_2, X_3), p_4(\bar{X}_3, \bar{Z})$$

---

35 Trimming of Stack Frames

Idea:

- Order local variables according to their life times;
- Pop the dead variables — if possible :-}

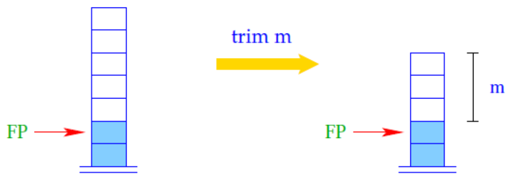
Example:

Consider the clause:

$$a(X, Z) \leftarrow p_1(\bar{X}, X_1), p_2(\bar{X}_1, X_2), p_3(\bar{X}_2, X_3), p_4(\bar{X}_3, \bar{Z})$$

After the query $p_2(\bar{X}_1, X_2)$, variable X_1 is dead.
After the query $p_3(\bar{X}_2, X_3)$, variable X_2 is dead :-)

After every non-last goal with dead variables, we insert the instruction trim :



```
if (FP ≥ BP)
    SP = FP + m;
```

Example (continued):

$$\mathsf{a}(X, Z) \leftarrow \mathsf{p}_1(\bar{X}, X_1), \mathsf{p}_2(\bar{X}_1, X_2), \mathsf{p}_3(\bar{X}_2, X_3), \mathsf{p}_4(\bar{X}_3, \bar{Z})$$

Ordering of the variables:

$$\rho = \{X \mapsto 1, Z \mapsto 2, X_3 \mapsto 3, X_2 \mapsto 4, X_1 \mapsto 5\}$$

The resulting code:

```
pushenv 5    A:  mark B          mark C      lastmark
mark A           putref 5        putref 4    putref 3
putref 1         putvar 4        putvar 3    putref 2
putvar 5         call p2/2       call p3/2   lastcall p4/2 3
call p1/2    B:  trim 4      C:  trim 3
```

---

36 Clause Indexing

Observation:

Often, predicates are implemented by case distinction on the first argument.

⟹ Inspecting the first argument, many alternatives can be excluded :-)

⟹ Failure is earlier detected :-)

⟹ Backtrack points are earlier removed. :-))

⟹ Stack frames are earlier popped :-)))

---

app([], Y, Y).

Example: The app-predicate:

$$\mathsf{app}(\boxed{X}, Y, Z) \leftarrow X = [\,],\ Y = Z$$
$$\mathsf{app}(\boxed{X}, Y, Z) \leftarrow X = [H|X'],\ Z = [H|Z'],\ \mathsf{app}(X', Y, Z')$$

- If the root constructor is [ ], only the first clause is applicable.
- If the root constructor is [|], only the second clause is applicable.
- Every other root constructor should fail !!
- Only if the first argument equals an unbound variable, both alternatives must be tried ;-)

Idea:

- Introduce separate try chains for every possible constructor.
- Inspect the root node of the first argument.
- Depending on the result, perform an indexed jump to the appropriate try chain.

Assume that the predicate p/k is defined by the sequence rr of clauses $r_1 \ldots r_m$.

Let tchains rr denote the sequence of try chains as built up for the root constructors occurring in unifications $X_1 = t$.

Example:

Consider again the app-predicate, and assume that the code for the two clauses start at addresses A_1 and A_2, respectively.

Then we obtain the following four try chains:

```
VAR:   setbtp      // variables     NIL:    jump $A_1$    // atom [ ]
       try $A_1$
       delbtp                       CONS:   jump $A_2$    // constructor [|]
       jump $A_2$

                                    ELSE:   fail          // default
```

Example:

Consider again the app-predicate, and assume that the code for the two clauses start at addresses A_1 and A_2, respectively.

Then we obtain the following four try chains:

```
VAR:   setbtp      // variables     NIL:    jump $A_1$    // atom [ ]
       try $A_1$
       delbtp                       CONS:   jump $A_2$    // constructor [|]
       jump $A_2$

                                    ELSE:   fail          // default
```

The new instruction fail takes care of any constructor besides [ ] and [|] ...

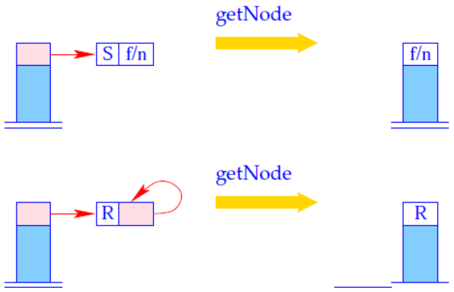
```
fail   =   backtrack()
```

It directly triggers backtracking :-)

Then we generate for a predicate p/k:

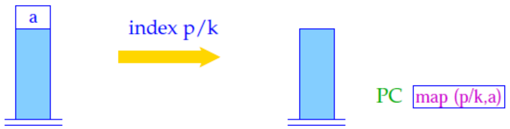
```
$\text{code}_P\ rr$   =            putref 1
                                   getNode      // extracts the root label
                                   index p/k    // jumps to the try block
                                   tchains $rr$
                       $A_1$ :     $\text{code}_C\ r_1$
                                   ...
                       $A_m$ :     $\text{code}_C\ r_m$
```

The instruction getNode returns “R” if the pointer on top of the stack points to an unbound variable. Otherwise, it returns the content of the heap object:



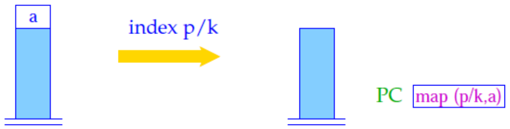
```
switch (H[S[SP]]) {
case (S, f/n):    S[SP] = f/n; break;
case (A,a):       S[SP] = a; break;
case (R,_) :      S[SP] = R;
}
```

The instruction index p/k performs an indexed jump to the appropriate try chain:



PC = map (p/k,S[SP]);
SP−−;

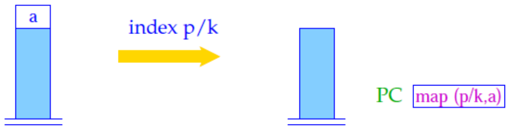
The instruction index p/k performs an indexed jump to the appropriate try chain:



PC = map (p/k,S[SP]);
SP−−;

The function map() returns, for a given predicate and node content, the start address of the appropriate try chain :-)

It typically is defined through some hash table :-))


The instruction index p/k performs an indexed jump to the appropriate try chain:

PC = map (p/k,S[SP]);
SP−−;

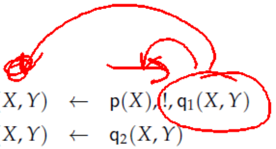
The function map() returns, for a given predicate and node content, the start address of the appropriate try chain :-)

It typically is defined through some hash table :-))

37 Extension: The Cut Operator

Realistic Prolog additionally provides an operator "!" (cut) which explicitly allows to prune the search space of backtracking.

Example:



$$\begin{array}{lcl} \mathsf{branch}(X,Y) & \leftarrow & \mathsf{p}(X), !, \mathsf{q}_1(X,Y) \\ \mathsf{branch}(X,Y) & \leftarrow & \mathsf{q}_2(X,Y) \end{array}$$

Once the queries before the cut have succeeded, the choice is committed:

Backtracking will return only to backtrack points preceding the call to the left-hand side ...

The Basic Idea:

- We restore the oldBP from our current stack frame;
- We pop all stack frames on top of the local variables.

Accordingly, we translate the cut into the sequence:

```
prune
pushenv m
```

where m is the number of (still used) local variables of the clause.

Example:

Consider our example:

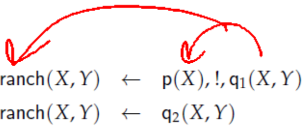
$$\begin{array}{lcl} \mathsf{branch}(X,Y) & \leftarrow & \mathsf{p}(X), !, \mathsf{q}_1(X,Y) \\ \mathsf{branch}(X,Y) & \leftarrow & \mathsf{q}_2(X,Y) \end{array}$$

We obtain:

```
setbtp     A:  pushenv 2   C:  prune       lastmark        B:  pushenv 2
try A          mark C          pushenv 2   putref 1            putref 2
delbtp         putref 1                    putref 2            putref 2
jump B         call p/1                    lastcall q1/2 2     move 2 2
                                                               jump q2/2
```

Example:

Consider our example:

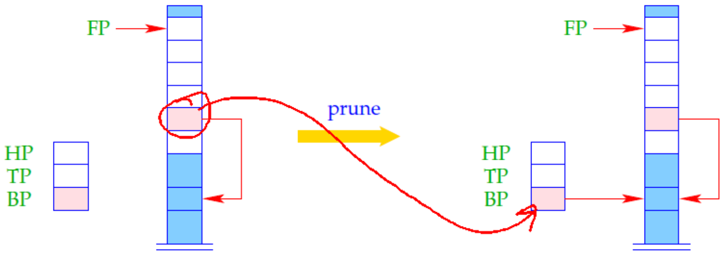


$$\begin{array}{lcl} \mathsf{branch}(X,Y) & \leftarrow & \mathsf{p}(X), !, \mathsf{q}_1(X,Y) \\ \mathsf{branch}(X,Y) & \leftarrow & \mathsf{q}_2(X,Y) \end{array}$$

We obtain:

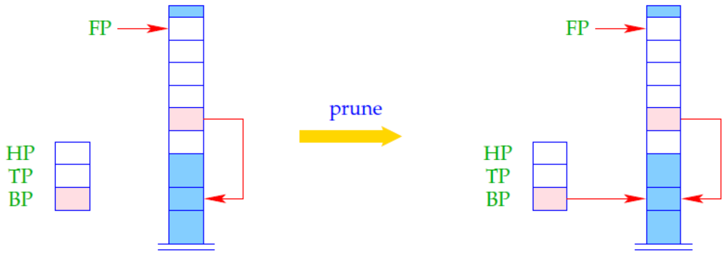
```
setbtp     A:  pushenv 2   C:  prune       lastmark        B:  pushenv 2
try A          mark C          pushenv 2   putref 1            putref 2
delbtp         putref 1                    putref 2            putref 2
jump B         call p/1                    lastcall q1/2 2     move 2 2
                                                               jump q2/2
```

The new instruction prune simply restores the backtrack pointer:



BP = BPold;

The new instruction prune simply restores the backtrack pointer:



BP = BPold;

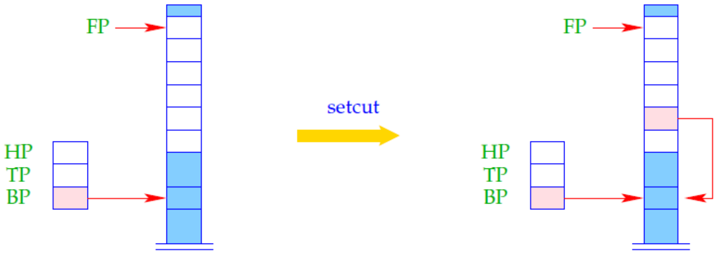
Problem:

If a clause is single, then (at least so far ;-) we have not stored the old BP inside the stack frame :-(

$\Longrightarrow$

For the cut to work also with single-clause predicates or try chains of length 1, we insert an extra instruction setcut before the clausal code (or the jump):

The instruction setcut just stores the current value of BP:



BPold = BP;

The Final Example: Negation by Failure

The predicate notP should succeed whenever p fails (and vice versa :-)

$$\begin{array}{lll} \text{notP}(X) & \leftarrow & \text{p}(X), !, \text{fail} \\ \text{notP}(X) & \leftarrow & \end{array}$$

where the goal fail never succeeds. Then we obtain for notP :

```
setbtp      A:  pushenv 1    C:  prune        B:  pushenv 1
try A           mark C           pushenv 1        popenv
delbtp          putref 1         fail
jump B          call p/1         popenv
```

The Final Example: Negation by Failure

The predicate notP should succeed whenever p fails (and vice versa :-)

$$\begin{array}{lll} \text{notP}(X) & \leftarrow & \text{p}(X), !, \text{fail} \\ \text{notP}(X) & \leftarrow & \end{array}$$

where the goal fail never succeeds. Then we obtain for notP :

```
setbtp      A:  pushenv 1    C:  prune        B:  pushenv 1
try A           mark C           pushenv 1        popenv
delbtp          putref 1         fail
jump B          call p/1         popenv
```

The Final Example: Negation by Failure

The predicate notP should succeed whenever p fails (and vice versa :-)

$$\begin{array}{lll} \text{notP}(X) & \leftarrow & \text{p}(X), !, \text{fail} \\ \text{notP}(X) & \leftarrow & \end{array}$$

where the goal fail never succeeds. Then we obtain for notP :

```
setbtp      A:  pushenv 1    C:  prune        B:  pushenv 1
try A           mark C           pushenv 1        popenv
delbtp          putref 1         fail
jump B          call p/1         popenv
```