Script generated by TTT

Title: Seidl: Virtual_Machines (22.06.2015)

Date: Mon Jun 22 10:15:55 CEST 2015

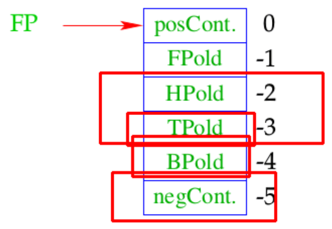
Duration: 89:42 min

Pages: 45

A backtrack point is stack frame to which program execution possibly returns.

- We need the code address for trying the next alternative (negative continuation address);
- We save the old values of the registers HP, TP and BP.
- Note: The new BP will receive the value of the current FP :-)

For this purpose, we use the corresponding four organizational cells:



For more comprehensible notation, we thus introduce the macros:

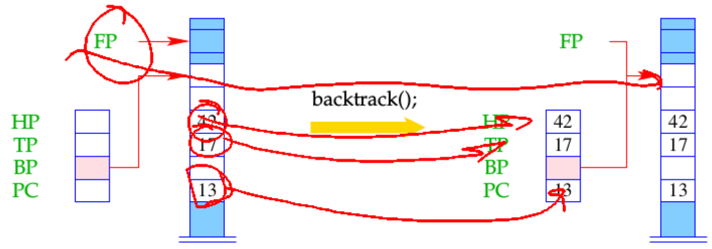
$$
\begin{array}{lcl}
\text{posCont} & \equiv & S[\text{FP}] \\
\text{FPold} & \equiv & S[\text{FP}-1] \\
\text{HPold} & \equiv & S[\text{FP}-2] \\
\text{TPold} & \equiv & S[\text{FP}-3] \\
\text{BPold} & \equiv & S[\text{FP}-4] \\
\text{negCont} & \equiv & S[\text{FP}-5]
\end{array}
$$

for the corresponding addresses.

Remark

Occurrence on the left	==	saving the register
Occurrence on the right	==	restoring the register

Calling the run-time function `void backtrack()` yields:




```
void backtrack() {
    FP = BP; HP = HPold;
    reset (TPold, TP);
    TP = TPold; PC = negCont;
}
```

where the run-time function `reset()` undoes the bindings of variables established since the backtrack point.

33.2 Trailing and Resetting Variables

Idea

- The variables which have been created since the last backtrack point can be removed together with their bindings by popping the heap !!
- This works fine if younger variables always point to older objects.
- Bindings of old variables to younger objects, though, must be reset explicitly :-(
- These are therefore recorded in the trail.

Functions `void trail(ref u)` and `void reset (ref y, ref x)` can thus be implemented as:

```
void trail (ref u) {
   if (u < S[BP-2]) {
      TP = TP+1;
      T[TP] = u;
   }
}
```

```
void reset (ref x, ref y) {
   for (ref u=y; x<u; u--)
      H[T[u]] = (R,T[u]);
}
```

Here, `S[BP-2]` represents the heap pointer when creating the last backtrack point.

Functions `void trail(ref u)` and `void reset (ref y, ref x)` can thus be implemented as:

```
void trail (ref u) {
   if (u < S[BP-2]) {
      TP = TP+1;
      T[TP] = u;
   }
}
```

```
void reset (ref x, ref y) {
   for (ref u=y; x<u; u--)
      H[T[u]] = (R,T[u]);
}
```

Here, `S[BP-2]` represents the heap pointer when creating the last backtrack point.

301

33.2 Trailing and Resetting Variables

Idea

- The variables which have been created since the last backtrack point can be removed together with their bindings by popping the heap !!
- This works fine if younger variables always point to older objects.
- Bindings of old variables to younger objects, though, must be reset explicitly :-(
- These are therefore recorded in the trail.

300

Functions `void trail(ref u)` and `void reset (ref y, ref x)` can thus be implemented as:

```
void trail (ref u) {
   if (u < S[BP-2]) {
      TP = TP+1;
      T[TP] = u;
   }
}
```

```
void reset (ref x, ref y) {
   for (ref u=y; x<u; u--)
      H[T[u]] = (R,T[u]);
}
```

Here, `S[BP-2]` represents the heap pointer when creating the last backtrack point.

33.3 Wrapping it Up

Assume that the predicate q/k is defined by the clauses $rr \equiv r_1, \ldots, r_f$ $(f > 1)$.
We provide code for:

- setting up the backtrack point;
- successively trying the alternatives;
- deleting the backtrack point.

This means:

$$
\begin{array}{lcrl}
\text{code}_P\ rr & = & \text{q/k}: & \text{setbtp} \\
 & & & \text{try}\ A_1 \\
 & & & \ldots \\
 & & & \text{try}\ A_{f-1} \\
 & & & \text{delbtp} \\
 & & & \text{jump}\ A_f \\
 & & A_1: & \text{code}_C\ r_1 \\
 & & & \ldots \\
 & & A_f: & \text{code}_C\ r_f
\end{array}
$$

Note

- We delete the backtrack point before the last alternative :-)
- We jump to the last alternative — never to return to the present frame :-))

$$
\begin{array}{lcrl}
\text{code}_P\ rr & = & \text{q/k}: & \boxed{\text{setbtp}} \\
 & & & \boxed{\text{try}\ A_1} \\
 & & & \ldots \\
 & & & \text{try}\ A_{f-1} \\
 & & & \boxed{\text{delbtp}} \\
 & & & \text{jump}\ A_f \\
 & & A_1: & \text{code}_C\ r_1 \\
 & & & \ldots \\
 & & A_f: & \text{code}_C\ r_f
\end{array}
$$

Note

- We delete the backtrack point before the last alternative :-)
- We jump to the last alternative — never to return to the present frame :-))

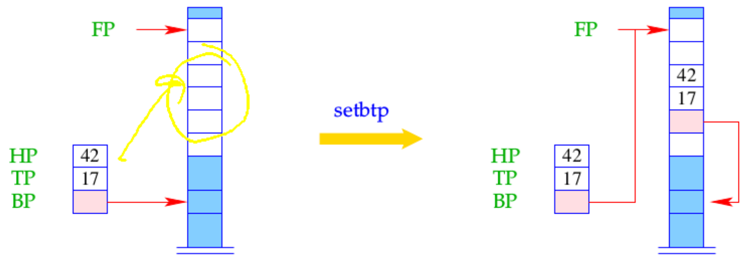
Example

$$\mathrm{s}(X) \leftarrow \mathrm{t}(\bar{X})$$
$$\mathrm{s}(X) \leftarrow \bar{X} = a$$

The translation of the predicate s yields:

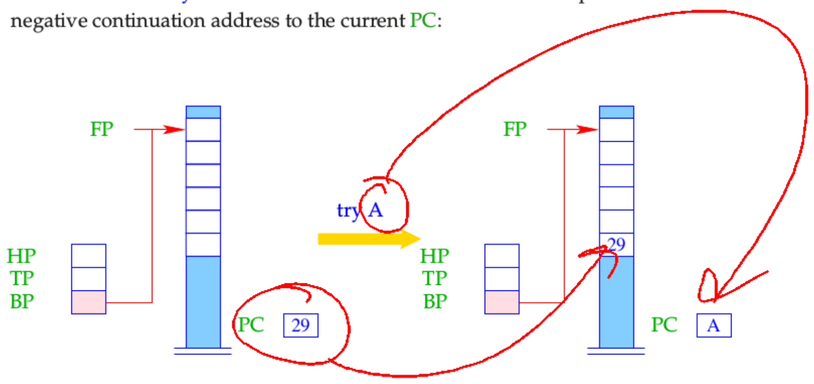
```
s/1:  setbtp     A:  pushenv 1     B:  pushenv 1
      try A          mark C            putref 1
      delbtp         putref 1          uatom a
      jump B         call t/1          popenv
                 C:  popenv
```


The instruction setbtp saves the registers HP, TP, BP:



HPold = HP;
TPold = TP;
BPold = BP;
BP = FP;

The instruction try A tries the alternative at address A and updates the negative continuation address to the current PC:



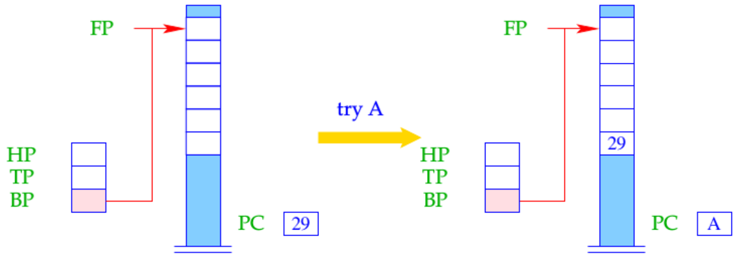
negForts = PC;
PC = A;

$$\text{code}_P\ rr \quad = \quad \text{q/k}: \quad \text{setbtp}$$
$$\text{try } A_1$$
$$\ldots$$
$$\text{try } A_{f-1}$$
$$\text{delbtp}$$
$$\text{jump } A_f$$
$$A_1: \quad \text{code}_C\ r_1$$
$$\ldots$$
$$A_f: \quad \text{code}_C\ r_f$$

Note

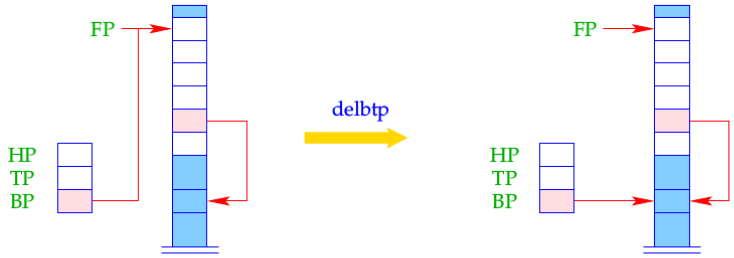
- We delete the backtrack point before the last alternative :-)
- We jump to the last alternative — never to return to the present frame :-))

The instruction try A tries the alternative at address A and updates the negative continuation address to the current PC:



negForts = PC;
PC = A;

The instruction delbtp restores the old backtrack pointer:



BP = BPold;

33.4 Popping of Stack Frames

Recall the translation scheme for clauses:

$$\begin{array}{lll} \text{code}_C\; r & = & \text{pushenv m} \\ & & \text{code}_G\; g_1\; \rho \\ & & \ldots \\ & & \text{code}_G\; g_n\; \rho \\ & & \text{popenv} \end{array}$$

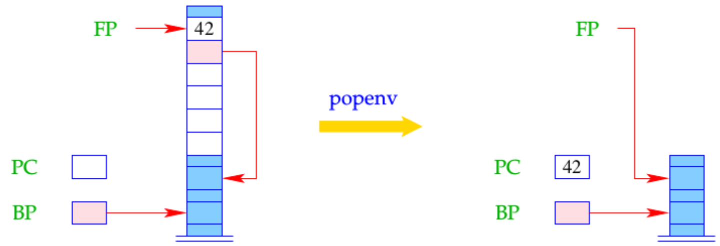
The present stack frame can be popped ...

- if the applied clause was the last (or only); and
- if all goals in the body are definitely finished.

$\Longrightarrow$ the backtrack point is older :-)

$\Longrightarrow$ FP > BP

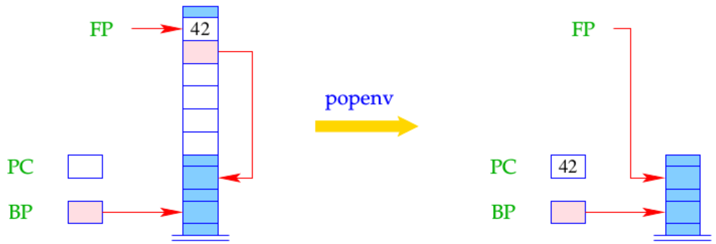
The instruction popenv restores the registers FP and PC and possibly pops the stack frame:



if (FP > BP) SP = FP - 6;
PC = posCont;
FP = FPold;

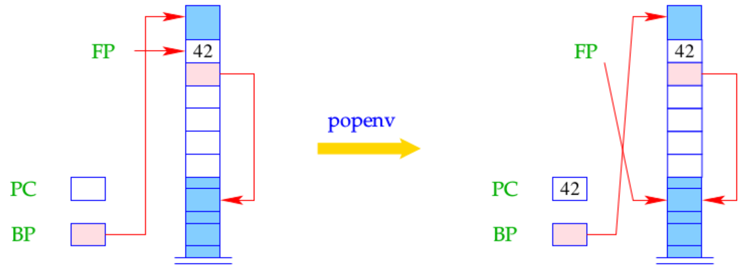
Caveat popenv may fail to de-allocate the frame !!!

The instruction popenv restores the registers FP and PC and possibly pops the stack frame:




```
if (FP > BP) SP = FP - 6;
PC = posCont;
FP = FPold;
```

Caveat popenv may fail to de-allocate the frame !!!




```
if (FP > BP) SP = FP - 6;
PC = posCont;
FP = FPold;
```

If popping the stack frame fails, new data are allocated on top of the stack. When returning to the frame, the locals still can be accessed through the FP :-))

? app(X, Y, [1, 2])

34 Queries and Programs

The translation of a program: $p \equiv rr_1 \ldots rr_h ?g$ consists of:

- an instruction no for failure;
- code for evaluating the literal g;
- code for the predicate definitions rr_i.

Preceding query evaluation:

$\Longrightarrow$ initialization of registers
$\Longrightarrow$ allocation of space for the globals

Succeeding query evaluation:

$\Longrightarrow$ returning the values of globals

code p = init A
pushenv d
code$_G$ g ρ
halt d
A: no
code$_P$ rr_1
...
code$_P$ rr_h

where $free(g) = \{X_1, \ldots, X_d\}$ and ρ is given by $\rho\, X_i = i$.

The instruction halt d ...

- ... terminates program execution;
- ... returns the bindings of the d globals;
- ... causes backtracking — if demanded by the user :-)

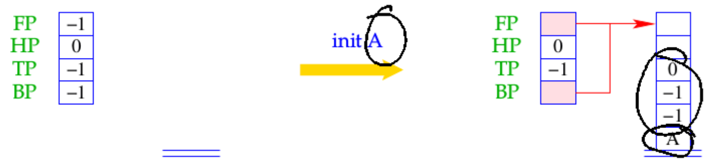
$$
\begin{array}{rll}
\text{code } p & = & \boxed{\text{init A}} \\
 & & \text{pushenv d} \\
 & & \text{code}_G\ g\ \rho \\
 & & \text{halt d} \\
 & \text{A:} & \text{no} \\
 & & \text{code}_P\ rr_1 \\
 & & \ldots \\
 & & \text{code}_P\ rr_h
\end{array}
$$

where $free(g) = \{X_1, \ldots, X_d\}$ and ρ is given by $\rho\ X_i = i$.

The instruction halt d ...

- ... terminates program execution;
- ... returns the bindings of the d globals;
- ... causes backtracking — if demanded by the user :-)

The instruction init A is defined by:



BP = FP = SP = 5;
S[0] = A;
S[1] = S[2] = -1;
S[3] = 0;
BP = FP;

At address "A" for a failing goal we have placed the instruction no for printing no to the standard output and halt :-)

The Final Example

$$
\begin{array}{lll}
t(X) \leftarrow \bar{X} = b & q(X) \leftarrow s(\bar{X}) & s(X) \leftarrow \bar{X} = a \\
p \leftarrow q(X), t(\bar{X}) & s(X) \leftarrow t(\bar{X}) & ?\quad p
\end{array}
$$

The translation yields:

	init N		popenv	q/1:	pushenv 1	E:	pushenv 1
	pushenv 0	p/0:	pushenv 1		mark D		mark G
	mark A		mark B		putref 1		putref 1
	call p/0		putvar 1		call s/1		call t/1
A:	halt 0		call q/1	D:	popenv	G:	popenv
N:	no	B:	mark C	s/1:	setbtp	F:	pushenv 1
t/1:	pushenv 1		putref 1		try E		putref 1
	putref 1		call t/1		delbtp		uatom a
	uatom b	C:	popenv		jump F		popenv

The Final Example

t(X) ← X̄ = b   q(X) ← s(X̄)   s(X) ← X̄ = a
p ← q(X), t(X̄)   s(X) ← t(X̄)   ? p

The translation yields:

init N, popenv, q/1: pushenv 1, E: pushenv 1
pushenv 0, p/0: pushenv 1, mark D, mark G
mark A, mark B, putref 1, putref 1
call p/0, putvar 1, call s/1, call t/1
A: halt 0, call q/1, D: popenv, G: popenv
N: no, B: mark C, s/1: setbtp, F: pushenv 1
t/1: pushenv 1, putref 1, try E, putref 1
putref 1, call t/1, delbtp, uatom a
uatom b, C: popenv, jump F, popenv

35 Last Call Optimization

Consider the app predicate from the beginning:

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

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:

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

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

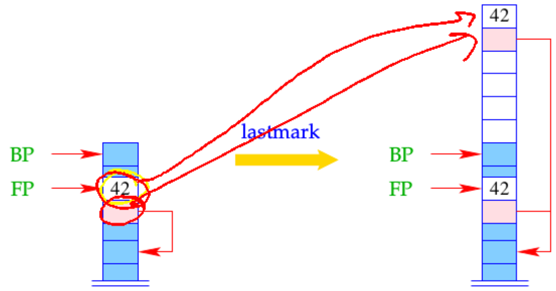
Consider a clause r: $p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_n$ with m locals where $g_n \equiv q(t_1, \ldots, t_h)$. The interplay between code_C and code_G:

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

Replacement:			
	mark B	$\Longrightarrow$	lastmark
	call q/h; popenv	$\Longrightarrow$	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 $\leq$ 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 :-)

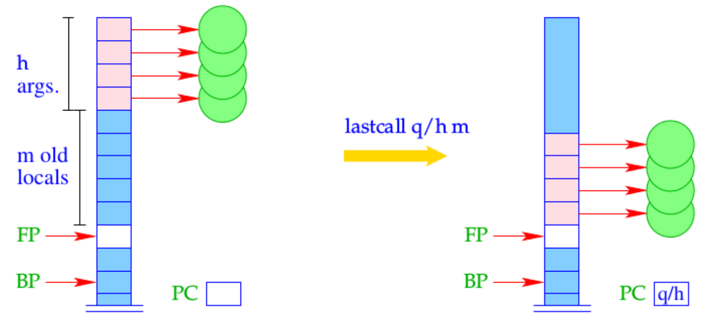
If $FP \leq BP$, then lastcall q/h m behaves like a normal call q/h.

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 :-))



35 Last Call Optimization

Consider the app predicate from the beginning:

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

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 !!!

Example

Consider the clause:

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

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

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

Example

Consider the clause:

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

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 ~~h m~~; jump ~~q/h~~ :-))

Example

Consider the last clause of the app predicate:

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

Here, the last call is the only one :-) Consequently, we obtain:

```
A:  pushenv 6                                   uref 4                 bind
    putref 1           B:  putvar 4             son 2              E:  putref 5
    ustruct [|]/2 B        putvar 5             uvar 6                 putref 2
    son 1                  putstruct [|]/2      up E                   putref 6
    uvar 4                 bind             D:  check 4                move 6 3
    son 2              C:  putref 3             putref 4               jump app/3
    uvar 5                 ustruct [|]/2 D      putvar 6
    up C                   son 1                putstruct [|]/2
```


Example

Consider the last clause of the app predicate:

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

Here, the last call is the only one :-) Consequently, we obtain:

```
A:  pushenv 6                                   uref 4                 bind
    putref 1           B:  putvar 4             son 2              E:  putref 5
    ustruct [|]/2 B        putvar 5             uvar 6                 putref 2
    son 1                  putstruct [|]/2      up E                   putref 6
    uvar 4                 bind             D:  check 4                move 6 3
    son 2              C:  putref 3             putref 4               jump app/3
    uvar 5                 ustruct [|]/2 D      putvar 6
    up C                   son 1                putstruct [|]/2
```

323

36 Trimming of Stack Frames

Idea

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

36 Trimming of Stack Frames

Idea

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

Example

Consider the clause:

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

36 Trimming of Stack Frames

Idea

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

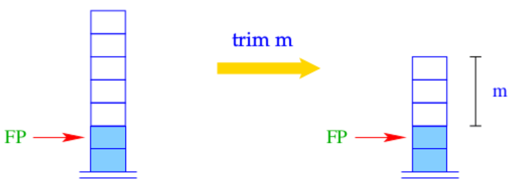
Example

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

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

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 $p_2/2$		call $p_3/2$	lastcall $p_4/2$ 3
call $p_1/2$	B:	trim 4	C:	trim 3	

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 $p_2/2$		call $p_3/2$	lastcall $p_4/2$ 3
call $p_1/2$	B:	trim 4	C:	trim 3	

37 Clause Indexing

Observation

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

$\Longrightarrow$ Inspecting the first argument, many alternatives can be excluded :-)

$\Longrightarrow$ Failure is earlier detected :-)

$\Longrightarrow$ Backtrack points are earlier removed. :-))

$\Longrightarrow$ Stack frames are earlier popped :-)))