Script generated by TTT

Title: Seidl: Virtual_Machines (07.06.2016)

Date: Tue Jun 07 10:24:18 CEST 2016

Duration: 90:45 min

Pages: 46

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 !!
- Bindings of variables in the old heap section, 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.

33.3 Wrapping it Up

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

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

This means:

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

Remark

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

Example

$$\begin{array}{lcl} \text{s}(X) & \leftarrow & \text{t}(\bar{X}) \\ \text{s}(X) & \leftarrow & \bar{X} = a \end{array}$$

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
```


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

Remark

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

303

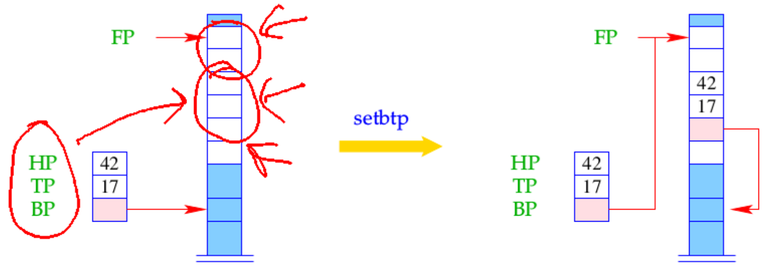
Example

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

The translation of the predicate s yields:

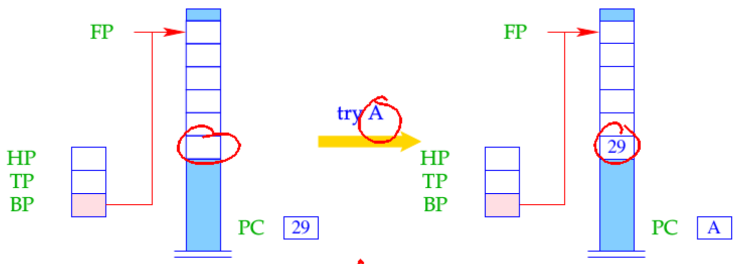
s/1:	setbtp	A:	pushenv 1	B:	pushenv 1
	try A		mark C		putref 1
	delbtp		putref 1		uatom a
	jump B		call t/1		popenv
		C:	popenv		

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



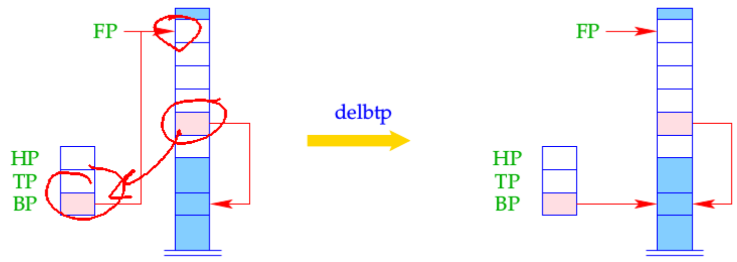
HPold = HP;
TPold = TP;
BPold = BP;
BP = FP;

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



negForts = PC;
PC = A;

The instruction delbtp restores the old backtrack pointer:



BP = BPold;

33.4 Popping of Stack Frames

Recall the translation scheme for clauses:

$$\text{code}_C\ r = \begin{array}{l}\text{pushenv m}\\ \text{code}_G\ g_1\ \rho\\ \dots\\ \text{code}_G\ g_n\ \rho\\ \text{popenv}\end{array}$$

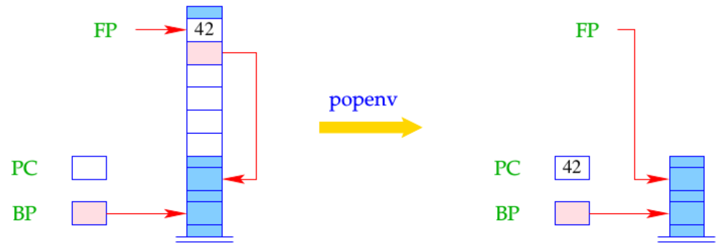
The present stack frame can be popped ...

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

⟹ the backtrack point is older

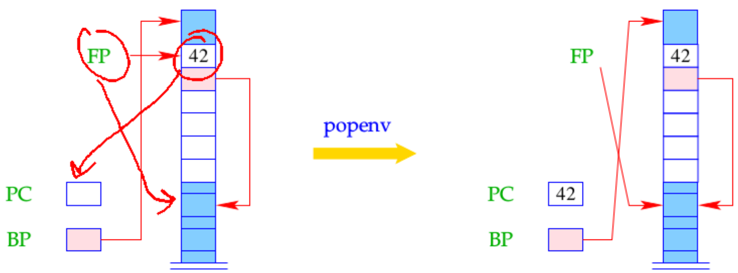
⟹ FP > BP

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 !!!



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!

34 Queries and Programs

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

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

Preceding query evaluation:

⟹ initialization of registers
⟹ allocation of space for the globals

Succeeding query evaluation:

⟹ returning the values of globals

$$\text{code } p \quad = \quad \begin{array}{l} \text{init A} \\ \text{pushenv d} \\ \text{code}_G\ g\ \rho \\ \text{halt d} \end{array}$$

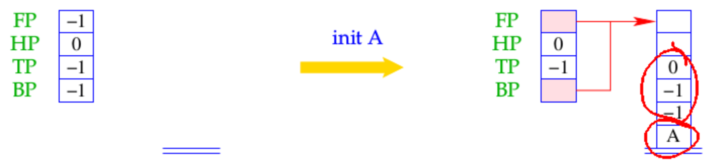
$$\text{A:} \quad \begin{array}{l} \text{no} \\ \text{code}_P\ rr_1 \\ \ldots \\ \text{code}_P\ rr_h \end{array}$$

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

$$t(X) \leftarrow \bar{X} = b \qquad q(X) \leftarrow s(\bar{X}) \qquad s(X) \leftarrow \bar{X} = a$$
$$p \leftarrow q(X), t(\bar{X}) \qquad s(X) \leftarrow t(\bar{X}) \qquad ?\ \ p$$

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

The Final Example

$$t(X) \leftarrow \bar{X} = b \qquad q(X) \leftarrow s(\bar{X}) \qquad s(X) \leftarrow \bar{X} = a$$
$$p \leftarrow q(X), t(\bar{X}) \qquad s(X) \leftarrow t(\bar{X}) \qquad ?\ \ p$$

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

314

35 Last Call Optimization

Consider the app predicate from the beginnning:

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

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:

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

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

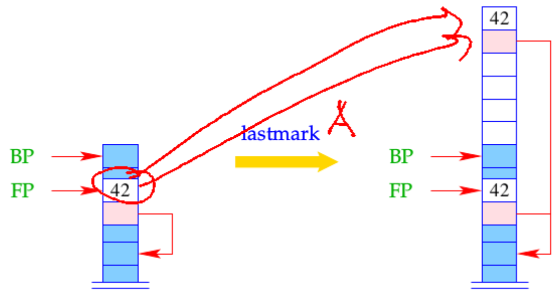
Consider a clause r: $p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_n$
with m locals where $g_n \equiv q(t_1, \ldots, t_h)$. The interplay between code_C and code_G:

$\text{code}_C\ r$ =
pushenv m
$\text{code}_G\ g_1\ \rho$
...
$\text{code}_G\ g_{n-1}\ \rho$
lastmark
$\text{code}_A\ t_1\ \rho$
. . .
$\text{code}_A\ t_h\ \rho$
lastcall q/h m

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 ≤ 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.

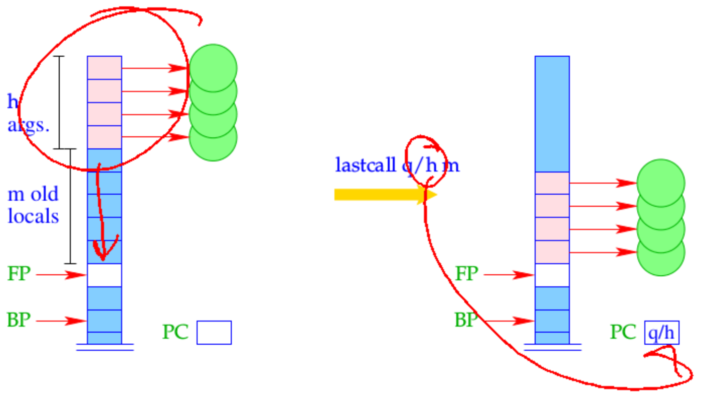
If FP ≤ BP, then lastcall q/h m behaves like a normal call q/h.

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:

```
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:

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

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
```

Remark

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.

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
```


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.

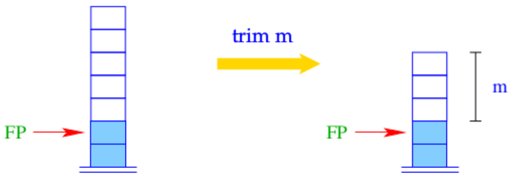
Example

Consider the clause:

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

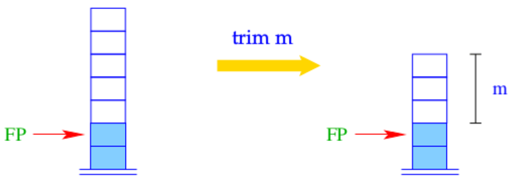
After the literal $p_2(\bar{X}_1, X_2)$, variable X_1 is dead.
After the literal $p_3(\bar{X}_2, X_3)$, variable X_2 is dead.

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




```
if (FP ≥ BP)
   SP = FP + m;
```


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




```
if (FP ≥ BP)
   SP = FP + m;
```

The dead locals can only be popped if no new backtrack point has been allocated.

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 p2/2       call p3/2   lastcall p4/2 3
    call p1/2    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 !!!

Example

The app-predicate:

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

- 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 ;-)

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

333

Then we generate for the predicate p/k:

$\text{code}_P\ rr$ = p/k: 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 : $\text{code}_C\ r_m$

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 [|] ...

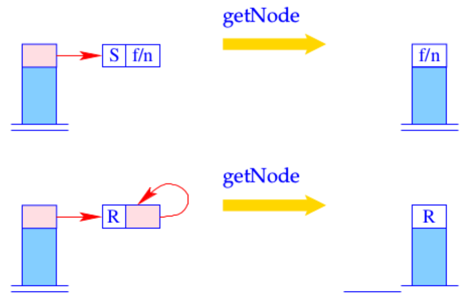
fail = `backtrack()`

It directly triggers backtracking ...

Then we generate for the predicate p/k:

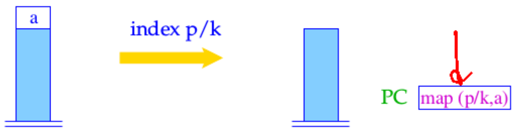
$\text{code}_P\ rr$ = p/k: putref 1
getNode // extracts the root label
index p/k // jumps to the try block
tchains rr
A_1 : $\text{code}_C\ r_1$
...
A_m : $\text{code}_C\ r_m$

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:



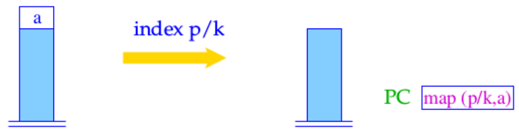

```
switch (H[S[SP]]) {
case (S, f/n):    S[SP] = f/n; break;
case (A,a):       S[SP] = a; break;
case (R,_) :      S[SP] = R;
}
```

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 ...

38 Extension: The Cut Operator

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

Example

$$\text{branch}(X, Y) \leftarrow \text{p}(X), !, \text{q}_1(X, Y)$$
$$\text{branch}(X, Y) \leftarrow \text{q}_2(X, Y)$$

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 ...