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:

```
| FP | posCont | 0 |
| FPold | -1 |
| HPold | -2 |
| TPold | -3 |
| BPold | -4 |
| negCont | -5 |
```

For more comprehensible notation, we thus introduce the macros:

```
posCont = S[FP]
FPold = S[FP - 1]
HPold = S[FP - 2]
TPold = S[FP - 3]
BPold = S[FP - 4]
negCont = S[FP - 5]
```

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, TPc);
    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 \( \text{void trail}(\text{ref } u) \) and \( \text{void reset}(\text{ref } y, \text{ref } x) \) can thus be implemented as:

\[
\begin{align*}
\text{void trail}(\text{ref } u) \{ \\
\quad \text{if } (u < S[BP-2]) \{ \\
\quad\quad \text{TP} = TP+1; \\
\quad\quad T[\text{TP}] = u; \\
\quad \} \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{void reset}(\text{ref } x, \text{ref } y) \{ \\
\quad \text{for } (\text{ref } w=y; x<u; u--) \\
\quad\quad \text{H}[T[u]] = (\text{R}, T[u]); \\
\quad \} \\
\}
\end{align*}
\]

Here, \( S[BP-2] \) represents the heap pointer when creating the last 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:

```c
void trail (ref u) {
    if (u < S[BP-2]) {
        for (ref u-y, x; u--;)
            TP = TP+1;
            H[T[u]] = (B, T[u]);
            T[TP] = 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

\[ r_r \equiv r_1, \ldots, r_f \]

(\( f > 1 \)).

We provide code for:

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

This means:

\[
\text{code}_{r_r} = q/k: \text{setbp} \\ \text{try} \ A_1 \\ \cdots \\ \text{try} \ A_{f-1} \\ \text{delbp} \\ \text{jump} \ A_f
\]

\[
A_1: \ \text{code}_{r_1} \\ \cdots \\ A_f: \ \text{code}_{r_f}
\]

Note

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

\[ s(\bar{x}) \leftarrow t(\bar{x}) \]
\[ s(\bar{x}) \leftarrow \bar{x} = a \]

The translation of the predicate \( s \) yields:

\( s/1: \)

\[ \text{setbtp} \quad \text{try} \ A \quad \text{putref} \ 1 \quad \text{mark} \ C \quad \text{putenv} \ 1 \]

\[ \text{delbtp} \quad \text{putref} \ 1 \quad \text{putenv} \ a \]

\[ \text{jump} \ B \quad \text{call} \ t/1 \quad \text{popenv} \]

\[ \text{C: popenv} \]

The instruction \( \text{setbtp} \) saves the registers \( HP, TP, BP \):

\[ \text{HPold} = HP; \]
\[ \text{TPold} = TP; \]
\[ \text{BPold} = BP; \]
\[ \text{BP} = FP; \]

The instruction \( \text{try} \ A \) tries the alternative at address \( A \) and updates the negative continuation address to the current \( PC \):

\[ \text{code}_{rr} = q/k: \]

\[ \text{setbtp} \]

\[ \text{try} \ A_1 \]

\[ \ldots \]

\[ \text{try} \ A_{f-1} \]

\[ \text{delbtp} \]

\[ \text{jump} \ A_f \]

\[ A_1 : \]

\[ \text{code}_{rr}, r_1 \]

\[ \ldots \]

\[ A_f : \]

\[ \text{code}_{rr}, r_f \]

Note

- We delete the backtrack point before the last alternative \( \Rightarrow \)
- We jump to the last alternative — never to return to the present frame \( \Rightarrow \)
The instruction **try A** tries the alternative at address A and updates the negative continuation address to the current PC:

\[ \text{negConts} = \text{PC}; \]
\[ \text{PC} = A; \]

### 33.4 Popping of Stack Frames

Recall the translation scheme for clauses:

\[
\text{code}_c \cdot r = \begin{cases} 
\text{pushenv} & \text{in} \\
\text{code}_c \cdot g_1 \cdot \rho & \\
\vdots & \\
\text{code}_c \cdot g_n \cdot \rho & \\
\text{popenv} & 
\end{cases}
\]

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**.

\[ \implies \text{the backtrack point is older} \implies \]
\[ \implies \text{FP} > \text{BP} \]

The instruction **delbp** restores the old backtrack pointer:
\[ \text{BP} = \text{BP}_{\text{old}}; \]

The instruction **popenv** restores the registers FP and PC and possibly pops the stack frame:
\[ \text{if (FP > BP) SP = FP - 6;} \]
\[ \text{PC = posCont;} \]
\[ \text{FP = FP}_{\text{old}}; \]

**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 !!!

34 Queries and Programs

The translation of a program: 

```
p ≡ r₁ . . . rₙ|g
```

consists of:

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

**Preceding** query evaluation:

```
⇒ initialization of registers
⇒ allocation of space for the globals
```

**Succeeding** query evaluation:

```
⇒ returning the values of globals
```

```
code p =

pushenv d

code; g p
halt d

A: no

code p r₁
...

code p rₙ
```

where `free(g) = {X₁, . . . , Xₜ}` and `ρ` is given by 

```
ρ Xᵢ = i
```

The instruction `halt d`...

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

301
\[
\begin{align*}
\text{code } p &= \begin{aligned}
&\text{init } A \\
pushenv \ d \\
\text{code: } p &\ a \\
\text{halt } d \\
\text{no} \\
\text{code: } r r_1 \\
&\ldots \\
\text{code: } r r_4 
\end{aligned} \\
\text{where } f r e e ( g ) &= \{ X_1, \ldots , X_d \} \quad \text{and } \rho \text{ is given by } \rho X_i = i .
\end{align*}
\]

The instruction \text{halt } d \ldots

\begin{itemize}
\item \ldots terminates program execution;
\item \ldots returns the bindings of the \( d \) globals;
\item \ldots causes backtracking — if demanded by the user.
\end{itemize}

\[312\]

The Final Example

\[
\begin{align*}
t(X) & \leftarrow X = b \\
q(X) & \leftarrow s(X) \\
s(X) & \leftarrow t(X) \\
p & \leftarrow q(X), t(X)
\end{align*}
\]

The translation yields:

\[
\begin{array}{cccccccc}
\text{init } N & \text{popenv} & q/1: & \text{pushenv} & 1 & E: & \text{pushenv} & 1 \\
pushenv 0 & p/0: & \text{pushenv} 1 & \text{mark } D & \text{putref} 1 & \text{call } t/1 \\
\text{mark } A & \text{mark } B & \text{putvar} 1 & \text{call } q/1 \\
call p/0 & \text{popenv} & G: & \text{popenv} \\
A: & \text{halt } 0 & \text{popenv} & F: & \text{pushenv} 1 & \text{try } E & \text{putref} 1 & \text{deltp } \\
N: & \text{no} & B: & \text{mark } C & \text{call } t/1 & \text{uatom } b & \text{popenv} \\
t/1: & \text{pushenv } 1 & \text{putref } 1 & \text{jump } F & \text{popenv}
\end{array}
\]

\[314\]

The Final Example

\[
\begin{align*}
t(X) & \leftarrow X = b \\
q(X) & \leftarrow s(X) \\
s(X) & \leftarrow t(X) \\
p & \leftarrow q(X), t(X)
\end{align*}
\]

The translation yields:

\[
\begin{array}{cccccccc}
\text{init } N & \text{popenv} & q/1: & \text{pushenv} & 1 & E: & \text{pushenv} & 1 \\
pushenv 0 & p/0: & \text{pushenv} 1 & \text{mark } D & \text{putref} 1 & \text{call } t/1 \\
\text{mark } A & \text{mark } B & \text{putvar} 1 & \text{call } q/1 \\
call p/0 & \text{popenv} & G: & \text{popenv} \\
A: & \text{halt } 0 & \text{call } q/1 & D: & \text{popenv} & \text{popenv} & \text{popenv} \\
N: & \text{no} & B: & \text{mark } C & \text{call } t/1 & \text{uatom } a & \text{popenv} \\
t/1: & \text{pushenv } 1 & \text{putref } 1 & \text{try } E & \text{putref } 1 & \text{deltp } & \text{uatom } a \\
\text{uatom } b & \text{popenv} & \text{jump } F & \text{popenv} & \text{popenv}
\end{array}
\]

\[315\]

The instruction \text{init } A \ldots is defined by:

\[
\begin{align*}
FP &= -1 \\
HP &= 0 \\
TP &= -1 \\
BP &= -1 \\
BP &= FP = SP = 5; \\
S[0] &= A_i \\
S[3] &= 0; \\
BP &= FP;
\end{align*}
\]

At address “A” for a failing goal we have placed the instruction \text{no} for printing \text{no} to the standard output and halt \ldots

\[316\]
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.

\[\Rightarrow\] we try to evaluate it in the current stack frame !!!
\[\Rightarrow\] after (successful) completion, we will not return to
the current caller !!!

Consider a clause \( r \):

\[
p(X_1, \ldots, X_n) \leftarrow g_1, \ldots, g_n
\]
\[
g_n \equiv q(t_1, \ldots, t_k).
\]

The interplay between code\(_C\) and code\(_Q\):

\[
\text{code}_C \ r = \begin{cases} 
\text{pushenv } m \\
\text{code}_C \ g_1 \rho \\
\ldots \\
\text{code}_C \ s_{n-1} \rho \\
\text{lastmark} \\
\text{code}_Q \ t_1 \rho \\
\ldots \\
\text{lastcall } q/h \ m
\end{cases}
\]

Replacement:

\[\text{mark } B \implies \text{lastmark}
\]
\[\text{call } q/h; \text{popenv} \implies \text{lastcall } q/h \ m\]

If the current clause is not last or the \( s_{n-1} \) have created backtrack points, then \( FP \leq BP \implies \)
Then lastmark creates a new frame and stores a reference to the predecessor:

\[
\text{if } (FP \leq BP) \{
SP = SP + 6;
S[SP] = \text{posCont}; S[SP-1] = \text{FPold};
\}
\]
If \( FP > BP \) then lastmark does nothing \( \implies \)
If \( FP \leq BP \), then \( \text{lastcall q/h m} \) behaves like a normal \( \text{call q/h} \).
Otherwise, the current stack frame is re-used. This means that:

- the cells \( S[FP+1], S[FP+2], \ldots, S[FP+\text{m}] \) receive the new values and
- \( q/h \) can be jumped to.

\[
\text{lastcall q/h m} = \begin{cases} 
\text{if (FP} \leq \text{BP) call q/h;} \\
\quad \text{else} \begin{cases} 
\quad \text{move m h;} \\
\quad \text{jump q/h;}
\end{cases}
\end{cases}
\]

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:

\[
\begin{align*}
\text{app}(X, Y, Z) & \leftarrow X = [], Y = Z, \\
\text{app}(X, Y, Z) & \leftarrow X = [H[X']], Z = [H[Z']], \text{app}(X', Y, Z')
\end{align*}
\]

We observe:

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

\[\Rightarrow\] we try to evaluate it in the current stack frame!!!

\[\Rightarrow\Rightarrow\] after (successful) completion, we will not return to the current caller !!!

Example

Consider the clause:

\[
a(X, Y) \leftarrow f(X, X), a(X, Y)
\]

The last-call optimization for \( \text{code: r} \) yields:

- \( \text{pushenv 3} \)
- \( \text{putref 1} \)  \( \text{putref 3} \)
- \( \text{putvar 3} \)  \( \text{putref 2} \)
- \( \text{call f/2} \)  \( \text{lastcall a/2 3} \)

\[\text{mark A} \quad A: \text{lastmark}\]
Example

Consider the clause:

\[ a(X, Y) \leftarrow t(X, X), a(X, Y) \]

The last-call optimization for code \( r \) yields:

\[
\begin{align*}
\text{mark} A & \quad \text{A: lastmark} \\
\text{pushenv} 3 & \quad \text{putref 1} \\
\text{putvar} 3 & \quad \text{putref 2} \\
\text{call} f/2 & \quad \text{lastcall a/2 3}
\end{align*}
\]

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 (a jump/3, \( r \))

Example

Consider the last clause of the app predicate:

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

Here, the last call is the only one \( r \) Consequently, we obtain:

\[
\begin{align*}
\text{A: pushenv} 6 & \quad \text{B: putvar 4} \\
\text{putref 1} & \quad \text{putvar 5} \\
\text{ustruct} []/2 B & \quad \text{uvar 6} \\
\text{son 1} & \quad \text{putstruct}[]/2 \\
\text{uvar 4} & \quad \text{bind} \\
\text{son 2} & \quad \text{C: putref 3} \\
\text{uvar 5} & \quad \text{ustruct}[]/2 D \\
\text{up} C & \quad \text{son 1} \\
\end{align*}
\]

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:

\[ a(X, Z) \leftarrow p_1(\hat{X}, X_1), p_2(\hat{X}, X_2), p_3(\hat{X}, X_3), p_4(\hat{X}, \hat{Z}) \]

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

\[ \text{FP} \rightarrow \text{trim } m \]

if \((\text{FP} \geq \text{BP})\)

\[ SF = FP + m \]

Example (continued)

\[ a(X, Z) \leftarrow p_1(\hat{X}, X_1), p_2(\hat{X}, X_2), p_3(\hat{X}, X_3), p_4(\hat{X}, Z) \]

Ordering of the variables:

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

The resulting code:

- pushenv 5
- mark A
- putref 5
- mark C
- lastmark
- mark A
- putref 4
- putref 3
- putvar 4
- putvar 2
- call p2/2
- call p3/2
- lastcall p4/2
- call p1/2
- B: trim 4
- C: trim 3
Example (continued)

\[ a(X, Z) \leftarrow p_1(X, X_1), p_2(X, X_2), p_3(X, X_3), p_4(X, Z) \]

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
putvar 1  putvar 4  putvar 3  putvar 2
putvar 5  call p2/2  call p3/2  lastcall p4/2 3
call p1/2  B: trim 4  C: trim 3
```