Example:

For our example term $f(g(X, Y), s, Z)$ and
$$\rho = \{ X \mapsto 1, Y \mapsto 2, Z \mapsto 3 \}$$ we obtain:

\[
\begin{array}{cccccc}
\text{ustruct f/3 A}_1 & \text{up } B_1 & \text{E}_2 & \text{son } 2 & \text{putvar } 2 \\
\text{son } 1 & \text{uatom } a & \text{putstruct } g/2 \\
\text{ustruct g/2 A}_2 & \text{A}_2 & \text{check } 1 & \text{son } 3 & \text{putatom } a \\
\text{son } 1 & \text{putref } 1 & \text{uvar } 3 & \text{putvar } 3 \\
\text{uref } 1 & \text{putvar } 2 & \text{up } B_1 & \text{putstruct } f/3 \\
\text{son } 2 & \text{putstruct } g/2 & \text{A}_1 & \text{check } 1 & \text{bind} \\
\text{uvar } 2 & \text{bind} & \text{putref } 1 & \text{B}_1 & \ldots \\
\end{array}
\]

Code size can grow quite considerably — for deep terms. In practice, though, deep terms are “rare” :-)

31 Clauses

Clausal code must

- allocate stack space for locals;
- evaluate the body;
- free the stack frame (whenever possible :-)

Let $r$ denote the clause: $p(X_1, \ldots, X_i) \leftarrow X_1, \ldots, X_k$.

Let $\{X_1, \ldots, X_n\}$ denote the set of locals of $r$ and $\rho$ the address environment:

$$\rho X_i = i$$

Remark: The first $k$ locals are always the formals :-)}
Then we translate:

```
codeC r = pushenv m // allocates space for locals
codeC g1 ρ
    ...
codeC gₙ ρ
popenv
```

The instruction `popenv` restores FP and PC and tries to pop the current stack frame.

It should succeed whenever program execution will never return to this stack frame ☺️

---

Example:

Consider the clause \( r \):

\[
a(X, Y) \leftarrow f(X, X_1), a(X_1, Y)
\]

Then `codeC r` yields:

```
pushenv 3
man1 A
putref 1
putvar 3
putref 2
call1/2
```

---

32 Predicates

A predicate \( q/k \) is defined through a sequence of clauses \( rr \equiv r_1 \ldots r_f \).

The translation of \( q/k \) provides the translations of the individual clauses \( r_i \).

In particular, we have for \( f = 1 \):

```
codeC rr = codeC r_1
```

If \( q/k \) is defined through several clauses, the first alternative must be tried.

On failure, the next alternative must be tried.

```
→ backtracking ☺️
```
32.1 Backtracking

- Whenever unification fails, we call the run-time function `backtrack()`.
- The goal is to roll back the whole computation to the (dynamically) latest goal where another clause can be chosen — the last backtrack point.
- In order to undo intermediate variable bindings, we always have recorded new bindings with the run-time function `trail()`.
- The run-time function `trail()` stores variables in the data-structure `trail`:

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:

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

for the corresponding addresses.

Remark:

- Occurrence on the left — saving the register
- Occurrence on the right — restoring the register

32.2 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 manually :-(
- These are therefore recorded in the trail.

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

```c
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.

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]) {
        TP = TP+1;
        H(TP) = H(TP);
        T[TP] = u;
    }
}
```

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

Here, S[BP-2] represents the heap pointer when creating the last backtrack point.
32.3 Wrapping it Up

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

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

This means:

\[ \text{code}_{r_1} \cdot \text{setbtp} \]
\[ \text{try } A_1 \]
\[ \ldots \]
\[ \text{try } A_{f-1} \]
\[ \text{deltbtp} \]
\[ \text{jump } A_f \]
\[ A_1 : \text{code}\_r_1 \]
\[ \ldots \]
\[ A_f : \text{code}\_r_f \]

**Note:**
- We delete the backtrack point **before** the last alternative \( \rightarrow \)
- We **jump** to the last alternative — never to return to the present frame \( \rightarrow \)\)

**Example:**

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

The translation of the predicate $s$ yields:

\[ s/1: \text{setbtp} \quad \text{A: pushenv 1} \quad \text{B: pushenv 1} \]
\[ \text{try } A \quad \text{mark C} \quad \text{putref 1} \quad \text{putref 1} \]
\[ \text{deltbtp} \quad \text{uatom a} \quad \text{uatom a} \]
\[ \text{jump } B \quad \text{call } t/1 \quad \text{call } t/1 \]
\[ \text{C: popenv} \]

The instruction **setbtp** saves the registers HP, TP, BP:
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 `delbp` restores the old backtrack pointer:

```
BP = BPold;
```

---

### 32.4 Popping of Stack Frames

Recall the translation scheme for clauses:

```
code \{ r \} = pushenv m
code \{ \lambda_1 \rho \}
... 
code \{ \lambda_2 \rho \}
popenv
```

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;
```

Warning: `popenv` may fail to de-allocate the frame !!!
33 Queries and Programs

The translation of a program: \( p = r_1 \ldots r_n \) consists of:
- an instruction \( \text{no} \) for failure;
- code for evaluating the query \( g \);
- code for the predicate definitions \( r_i \).

**Preceding** query evaluation:
- initialization of registers
- allocation of space for the globals

**Succeeding** query evaluation:
- returning the values of globals

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

\[
\begin{align*}
\text{code } p &= \quad \text{init } A \\
&\quad \text{pushenv } d \\
&\quad \text{code}_{\text{g } p} \\
&\quad \text{halt } d \\
\text{A: } &\quad \text{no} \\
&\quad \text{code}_{\text{g } r_1} \\
&\quad \ldots \\
&\quad \text{code}_{\text{g } r_n}
\end{align*}
\]

where \( \text{free}(g) = \{ X_1, \ldots, X_d \} \) and \( p \) is given by \( p : X_i = i \).

The instruction \( \text{halt } d \) ... 
- ... terminates the program execution;
- ... returns the bindings of the \( d \) globals;
- ... causes backtracking — if demanded by the user.

At address "A" for a failing goal we have placed the instruction \( \text{no} \) for printing \( \text{no} \) to the standard output and halt.
The instruction \( \text{init A} \) is defined by:

\[
\begin{align*}
\text{FP} & \quad \text{HP} \\
0 & \quad 0 \\
\text{TP} & \quad \text{BP} \\
-1 & \quad -1
\end{align*}
\]

\[\text{BP} = \text{FP} - \text{SP} = 5;\]  
\[\text{S}[0] = \text{A};\]  
\[\text{S}[1] = \text{S}[2] = -1;\]  
\[\text{S}[3] = 0;\]  
\[\text{BP} = \text{FP};\]

At address "A" for a failing goal we have placed the instruction \( \text{no} \) for printing \( \text{no} \) to the standard output and halt \( \triangleright \rightarrow \).

---

The Final Example:

\[t(X) \leftarrow X = b\]  
\[q(X) \leftarrow s(X)\]  
\[s(X) \leftarrow X = a\]  
\[p \leftarrow q(X), t(X)\]  
\[s(X) \leftarrow t(X)\]  
\[? \leftarrow p\]

The translation yields:

\[
\begin{align*}
\text{init N} & \quad \text{popenv} & \text{popenv} & \text{pushenv 1} & \text{pushenv 1} & \text{E: pushenv 1} & \text{pushenv 1} \\
\text{pushenv 0} & \quad \text{p/o} & \text{pushenv 1} & \text{mark D} & \text{mark G} \\
\text{mark A} & \quad \text{mark B} \\
\text{call p/0} & \quad \text{putvar 1} & \text{putref 1} & \text{putref 1} \\
\text{A:} & \quad \text{halt 0} \\
\text{D:} & \quad \text{call q/1} \\
\text{B:} & \quad \text{popenv} \\
\text{C:} & \quad \text{setbtp} & \text{F: pushenv 1} \\
\text{t/1:} & \quad \text{putenv 1} \\
\text{putref 1} & \quad \text{call t/1} \\
\text{utam b} & \quad \text{setbtp} & \text{F: pushenv 1} \\
\expandafter\text{C:} & \quad \text{popenv} & \text{jump F} & \text{popenv}
\end{align*}
\]

---

34 Last Call Optimization

Consider the \text{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 last call.
  - We try to evaluate it in the current stack frame !!!
  - After (successful) completion, we will not return to the current caller !!!
Consider a clause \( p(X_1, \ldots, X_k) \rightarrow g_1, \ldots, g_\ell \) with \( m \) locals where \( g_\ell \equiv q(t_1, \ldots, t_k) \). The interplay between code \( c \) and code \( c' \):

\[
\begin{array}{l}
\text{code}_c \; r = \text{pushenv} \; m \\
\text{code}_c \; g_1 \; \rho \\
\vdots \\
\text{code}_c \; g_{\ell-1} \; \rho \\
\text{code}_c \; g_\ell \; \rho \\
\text{mark} \; B \\
\text{code}_A \; t_1 \; \rho \\
\vdots \\
\text{code}_A \; t_k \; \rho \\
\text{call} \; q/h \\
B \oplus \text{popenv}
\end{array}
\]

**Replacement:**

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

If the current clause is not last or the \( g_1, \ldots, g_{\ell-1} \) have created backtrack points, then \( FP \leq BP \implies \)

Then \text{lastmark} creates a new frame but stores a reference to the predecessor:

\[
\begin{array}{l}
\text{BP} \quad \text{FP} \quad 42 \\
\text{lastmark} \\
\text{BP} \quad \text{FP} \quad 42
\end{array}
\]

if \( FP \leq BP \) 
\[
\begin{array}{l}
SP = SP + 6 \\
S[SP] = \text{posCont}; S[SP-1] = FPopd;
\end{array}
\]

If \( FP > BP \) then \text{lastmark} does nothing \( \implies \)

If \( FP \leq BP \), then \text{lastcall} \; q/h \; m \; \text{behaves like a normal \; call} \; q/h.

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

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

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

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 \( \implies \)
Example:

Consider the clause:

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

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

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

$$\text{putref 3}$$ $$\text{putref 2}$$ $$\text{lastcall a/2 3}$$

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 $\text{move m n; jump p/n}$.

Example:

Consider the last clause of the app predicate:

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

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

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

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

Example:
Consider the clause:
\[ \text{app}(X, Y, Z) \leftarrow X = [H|X'], Z = [R|Z'], \text{app}(X', Y, Z') \]

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

A: pushv 6
putref 1 B: putvar 4
tvar 5
uvar 1
D: putref 4
up E: putvar 6
uvar 4
B: putref 5
uvar 6
uvar 4
D: putref 4
up E: putvar 6
uvar 4

35 Trimming of Stack Frames

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

Example:
Consider the clause:
\[ \text{app}(X, Y, Z) \leftarrow p_1(\bar{X}, X_1), p_2(\bar{X}, X_2), p_3(\bar{X}, X_3), p_4(\bar{X}, \bar{Z}) \]

After every non-last goal with dead variables, we insert the instruction `trim`:
Example (continued):

$$a(X, Z) \leftarrow p_1(\overline{X}, X_1), p_1(\overline{X}, X_2), p_3(\overline{X}, X_3), p_4(\overline{X}, Z)$$

Ordering of the variables:

$$\rho = \{ X \rightarrow 1, Z \rightarrow 2, X_0 \rightarrow 3, X_2 \rightarrow 4, X_1 \rightarrow 5 \}$$

The resulting code:

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

36 Clause Indexing

Observation:

- Inspecting the first argument, many alternatives can be excluded :>
- Failure is earlier detected :)
- Backtrack points are earlier removed :>))
- Stack frames are earlier popped :>)))

Example: The app-predicate:

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

- 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 :>