33.1 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 \(\Rightarrow\) 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.

The current stack frame where backtracking should return to is pointed at by the extra register \(BP:\)

![Diagram of stack frame with TP pointing to the topmost occupied Trail cell]
33.2 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.

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;
        T[TP] = u;
        M[T[u]] = (H,T[u]);
    }
}

void reset(ref x, ref y) {
    for (ref u=y; u=x; u--) 
        M[T[u]] = (H,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  \( r_1, \ldots, r_f \)  \( (f > 1) \).  
We provide code for:

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

This means:

```c
proved rr = q/k: setup
            try A_1
            ...
            try A_{f-1}
            delete
            jump A_f

A_1:       codeC r_1
            ...
A_f:       codeC 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  :-))
33.3 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:

\[
\text{code}_{p_{rr}} = \begin{cases} 
\text{setbp} & q/k : \\
\text{try } A_1 & \\
\vdots & \\
\text{try } A_{f-1} & \\
\text{delbp} & \\
\text{jump } A_f & \\
A_1 & \text{code } r_1 \\
\vdots & \\
A_f & \text{code } r_f \\
\end{cases}
\]

**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:**

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

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

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

The instruction **setbp** saves the registers HP, TP, BP:
The instruction \texttt{try \( A \)} tries the alternative at address \( A \) and updates the negative continuation address to the current PC:

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

\[
\text{negForts} = \text{PC};
\]

\[
\text{Note:}
\]

- We delete the backtrack point \texttt{before} the last alternative \( \rightarrow \)
- We jump to the last alternative — never to return to the present frame \( \rightarrow \)

\[
\text{code}\_rr = \begin{array}{l}
q/k : \text{setbp} \\
\text{try } A_1 \\
\cdots \\
\text{try } A_{f-1} \\
\text{delbp} \\
\text{jump } A_f \\
A_1 : \text{code}\_c r_1 \\
\cdots \\
A_f : \text{code}\_c r_f
\end{array}
\]
33.4 Popping of Stack Frames

Recall the translation scheme for clauses:

\[
\begin{align*}
\text{code}_C \cdot \text{r} & = \text{pushenv} \ \text{m} \\
\text{code}_C \cdot \text{g}_1 \cdot \rho & \ldots \\
\text{code}_C \cdot \text{g}_n \cdot \rho & \text{popenv}
\end{align*}
\]

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.

\[\rightarrow \text{the backtrack point is older} \rightarrow\]

\[\rightarrow \text{FP} > \text{BP} \rightarrow\]

---

The instruction \text{popenv} restores the registers FP and PC and possibly pops the stack frame:

\[
\begin{align*}
\text{if} \ (\text{FP} > \text{BP}) \ SP & = \text{FP} - 6; \\
\text{PC} & = \text{posCont}; \\
\text{FP} & = \text{FPold};
\end{align*}
\]

\[\text{Warning: popenv may fail to de-allocate the frame !!!}\]

---

The instruction \text{popenv} restores the registers FP and PC and possibly pops the stack frame:

\[
\begin{align*}
\text{if} \ (\text{FP} > \text{BP}) \ SP & = \text{FP} - 6; \\
\text{PC} & = \text{posCont}; \\
\text{FP} & = \text{FPold};
\end{align*}
\]

\[\text{Warning: popenv may fail to de-allocate the frame !!!}\]
34 Queries and Programs

The translation of a program: \( p \equiv r_1 \ldots r_n ? g \)
consists of:

- an instruction \textbf{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

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

where \( \text{free}(g) = \{X_1, \ldots, X_3\} \) and \( \rho \) is given by \( \rho X_i = i \).

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

The instruction \textbf{init }A is defined by:

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

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

\[
\begin{align*}
\text{BP} & = \text{FP} = \text{SP} = 5; \\
S[0] & = A; \\
S[3] & = 0; \\
\text{BP} & = \text{FP} \\
\end{align*}
\]

The Final Example:

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

The translation yields:

\[
\begin{align*}
\text{init } N \quad & \quad \text{popenv} \quad q/1: \quad \text{pushenv } 1 \\
\text{pushenv } 0 \quad & \quad \text{p/0:} \quad \text{pushenv } 1 \quad \text{mark } D \\
\text{mark } A \quad & \quad \text{mark } B \quad \text{putref } 1 \\
\text{call } p/0 \quad & \quad \text{putvar } 1 \quad \text{call } s/1 \\
\text{BP} & \quad \text{call } q/1 \quad \text{D: popenv} \\
A: & \quad \text{halt } 0 \quad \text{E: pushenv } 1 \\
N: & \quad \text{no} \quad \text{B: mark } C \quad \text{setbtp} \\
t/1: & \quad \text{pushenv } 1 \quad \text{C: popenv} \\
\text{putref } 1 \quad & \quad \text{call } t/1 \quad \text{try } E \\
\text{call } t/1 \quad & \quad \text{delbtp} \\
\text{jump } F \quad & \quad \text{F: pushenv } 1 \\
\end{align*}
\]
The Final Example:

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

The translation yields:

\begin{align*}
\text{init N} & \quad \text{popenv 0} \quad \text{q/1: pushenv 1} \quad \text{E: pushenv 1} \\
\text{pushenv 0} & \quad \text{pushenv 1} \quad \text{mark D} \quad \text{mark G} \\
\text{mark A} & \quad \text{mark B} \quad \text{putref 1} \quad \text{putref 1} \\
\text{call p/0} & \quad \text{putvar 1} \quad \text{call s/1} \quad \text{call t/1} \\
\text{A: halt 0} & \quad \text{call q/1} \quad \text{D: popenv} \quad \text{G: popenv} \\
\text{N: no} & \quad \text{B: mark C} \quad \text{s/1: selbtp} \quad \text{F: pushenv 1} \\
\text{t/1: pushenv 1} & \quad \text{putref 1} \quad \text{try E} \quad \text{putref 1} \\
\text{putref 1} & \quad \text{call t/1} \quad \text{delbtp} \quad \text{autom a} \\
\text{uatom b} & \quad \text{C: popenv} \quad \text{jump F} \quad \text{popenv}
\end{align*}

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**.
  
  \[ \longrightarrow \text{we try to evaluate it in the current stack frame} \]
  \[ \longrightarrow \text{after (successful) completion, we will not return to} \]
  \[ \text{the current caller} \]

Consider a clause \( r \):

\[ p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_s \]
\[ g_s \equiv q(t_1, \ldots, t_k) \]

The interplay between \( \text{code}_C \) and \( \text{code}_C^{\circ} \):

\[ \text{code}_C \ r = \text{pushenv m} \]
\[ \quad \text{code}_C \ g_1 \ \rho \]
\[ \quad \ldots \]
\[ \quad \text{code}_C \ g_{s-1} \ \rho \]
\[ \quad \text{mark B} \]
\[ \quad \text{code}_C \ t_1 \ \rho \]
\[ \quad \ldots \]
\[ \quad \text{code}_C \ t_k \ \rho \]
\[ \quad \text{call q/h} \]
\[ \text{B: popenv} \]

Replacement:

\[ \text{mark B} \longrightarrow \text{lastmark} \]
\[ \text{call q/h, popenv} \longrightarrow \text{lastcall q/h m} \]
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 $FP > BP$ then lastmark does nothing.

If $FP \leq BP$, then lastcall $q/h$ 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.

```
lastcall $q/h$ =
  if ($FP \leq BP$) 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 $FP \leq BP$, then lastcall $q/h$ 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.

```
lastcall $q/h$ =
  if ($FP \leq BP$) 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:
\[ a(X, Y) \leftarrow f(X, X) \cup a(X, Y) \]
The last-call optimization for \( \text{code: } 7 \) yields:

<table>
<thead>
<tr>
<th>pushenv 3</th>
<th>putref 1</th>
<th>putref 3</th>
<th>putref 1</th>
<th>putref 2</th>
<th>lastcall a/2 3</th>
</tr>
</thead>
</table>

Note:
If the clause is last and the last literal is the only one, we can skip \text{lastmark} and can replace \text{lastcall q/h m} with the sequence \text{move m n; jump p/n}.

322

Example:
Consider the last clause of the \text{app} predicate:
\[ \text{app}(X, Y, Z) \leftarrow X = |H|X', Z = |f|Z', \text{app}(X', Y, Z') \]
Here, the last call is the only one \( \Rightarrow \) Consequently, we obtain:

<table>
<thead>
<tr>
<th>pushenv 6</th>
<th>putref 1</th>
<th>putref 4</th>
<th>bind</th>
<th>uvar 6</th>
<th>uvar 5</th>
<th>uvar 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B: putvar 4</td>
<td>push struct</td>
<td></td>
<td>/2 B</td>
<td>son 2</td>
<td>E:</td>
<td>putref 5</td>
</tr>
<tr>
<td>son 1</td>
<td>putvar 5</td>
<td>uvar 6</td>
<td>putref 6</td>
<td>up E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>uvar 4</td>
<td></td>
<td>putref 3</td>
<td>move 6.3</td>
<td>check 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C: putref 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>up C</td>
<td>son 1</td>
<td>putref 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

325

36 Trimming of Stack Frames

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

323
36 Trimming of Stack Frames

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

Example:
Consider the clause:
\[
a(X, Z) \leftarrow p_1(X, X_1), p_2(X_1, X_2), p_3(X_2, X_3), p_4(X_3, Z)
\]

After every non-last goal with dead variables, we insert the instruction \( \text{trim} \):

\[
\text{if (FP} \geq BP) \\
\text{SP} = \text{FP} + m;
\]

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

Example (continued):
\[
a(X, Z) \leftarrow p_1(\tilde{X}, X_1), p_2(\tilde{X}_1, X_2), p_3(\tilde{X}_2, X_3), p_4(\tilde{X}_3, \tilde{Z})
\]

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 |
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.