33.2 Trailing and Resetting Variables

Idea

- The variables which have been created since the last backtracking 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 v, ref x) can thus be implemented as:

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

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

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

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

---

**Example**

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

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

\[
\begin{align*}
\text{s/1:} & \quad \text{setbp} & A: \text{pushenv 1} & B: \text{pushenv 1} \\
& \text{try } A & \text{mark } C & \text{putref 1} \\
& \text{delbp} & \text{putref 1} & \text{utom } a \\
& \text{jump } B & \text{call } t/1 & \text{popenv} \\
& C: & \text{popenv}
\end{align*}
\]

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

\[
\begin{align*}
\text{s}(X) & \leftarrow \text{t}(\bar{X}) \\
\text{s}(X) & \leftarrow \bar{X} = a
\end{align*}
\]

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

\[
\begin{align*}
\text{s/1:} & \quad \text{setbip} \\
\text{try A} & \quad \text{pushenv 1} \\
\text{delbip} & \quad \text{mark C} \\
\text{jump B} & \quad \text{putref 1} \\
\text{C:} & \quad \text{utorn a} \\
\text{call t/1} & \quad \text{popenv}
\end{align*}
\]

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

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

The instruction \( \text{delbip} \) restores the old backtrack pointer:

\[
\begin{align*}
\text{HP} & \quad \text{TP} \\
42 & \quad 17 \\
\text{BP} & \\
&
\end{align*}
\]

\[
\begin{align*}
\text{HPold} & = \text{HP} \\
\text{TPold} & = \text{TP} \\
\text{BPold} & = \text{BP} \\
\text{BP} & = \text{FP}
\end{align*}
\]

\[
\begin{align*}
\text{HP} & \quad \text{TP} \\
42 & \quad 17 \\
\text{BP} & \\
&
\end{align*}
\]

\[
\begin{align*}
\text{HPold} & = \text{HP} \\
\text{TPold} & = \text{TP} \\
\text{BPold} & = \text{BP} \\
\text{BP} & = \text{FP}
\end{align*}
\]

\[
\begin{align*}
\text{HP} & \quad \text{TP} \\
42 & \quad 17 \\
\text{BP} & \\
&
\end{align*}
\]
33.4 Popping of Stack Frames

Recall the translation scheme for clauses:

\[
\begin{align*}
\text{code}_C\ r &= \text{pushenv}\ m \\
\text{code}_C\ g_1\ r_1 \\
&\vdots \\
\text{code}_C\ g_r\ r_r \\
\text{popenv}
\end{align*}
\]

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.

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

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

\[
\begin{align*}
\text{FP} &\rightarrow 42 \\
\text{popenv} &\arrow[r] \text{FP} \\
\text{PC} &\rightarrow 42 \\
\text{BP} &\rightarrow \quad \\
\end{align*}
\]

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

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: \( p \equiv r_1 \ldots r_k \) consists of:
- an instruction \( \text{no} \) for failure;
- code for evaluating the literal \( g \);
- code for the predicate definitions \( r_j \).

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

**Succeeding** query evaluation:
- returning the values of globals
The instruction `halt d`...

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

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:

```
init N
popenv 0
pushenv p/0:
mark A
mark B
call p/0
A: halt 0
B: mark C
N: no
C: popenv
t/1:
pushenv 1
putref 1
call t/1
putref 1
autom b
```

E: `pushenv 1`

F: `pushenv 1`

G: `popenv`

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:

```
init N
popenv 0
pushenv p/0:
mark A
mark B
call p/0
A: halt 0
B: mark C
N: no
C: popenv
t/1:
pushenv 1
putref 1
call t/1
putref 1
autom b
```

E: `pushenv 1`

F: `pushenv 1`

G: `popenv`

At address “A” for a failing goal we have placed the instruction `no` for printing `no` to the standard output and `halt`.
35 Last Call Optimization

Consider the app predicate from the beginning:

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

We observe:
- The recursive call occurs in the last goal of the clause.
- Such a goal is called last call.

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

After (successful) completion, we will not return to the current caller!!!

Consider a clause \( p(X_1, \ldots, X_n) \leftarrow g_1, \ldots, g_n \)
with \( m \) locals where \( g_n \equiv q(t_1, \ldots, t_2) \). The interplay between \( \text{code}_{\text{c}} \) and \( \text{code}_{\text{d}} \):

\[
\text{code}_{\text{c}} r = \begin{align*}
\text{pushenv } m \\
\text{code}_{\text{d}} g_1 \rho \\
\vdots \\
\text{code}_{\text{d}} g_{n-1} \rho \text{ lastmark} \\
\text{code}_{\text{d}} t_1 \rho \\
\vdots \\
\text{code}_{\text{d}} t_2 \rho \\
\text{lastcall } q/h m
\end{align*}
\]

Replacement:
- \( \text{mark } B \) \( \implies \) \( \text{lastmark} \)
- \( \text{call } q/h \) \( \implies \) \( \text{lastcall } q/h m \)

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

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

If \( \text{FP} > \text{BP} \) then \( \text{lastmark} \) 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 $S[FP+1], S[FP+2], \ldots, S[FP+h]$ receive the new values and
- `q/h` can be jumped to.

```plaintext
lastcall q/h m = 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:

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

The last-call optimization for `code: r` yields:

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

Consider the clause:

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

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

mark A  \hspace{1cm} A:  \hspace{1cm} \text{lastmark}
putenv 3  \hspace{1cm} putref 1  \hspace{1cm} putref 3
putvar 3  \hspace{1cm} putref 2
call l/2  \hspace{1cm} 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 a/2 3 with the sequence

move m n, jump p/n.

36  Trimming of Stack Frames

Idea

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

Example

Consider the last clause of the app predicate:

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

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

A:  pushenv 6  \hspace{1cm} uref 4  \hspace{1cm} bind
putref 1  \hspace{1cm} B:  putvar 4  \hspace{1cm} son 2  \hspace{1cm} E:  putref 5
putvar \hspace{1cm} B:  putvar 5  \hspace{1cm} uvar 6
ustruct l/2  \hspace{1cm} putstruct l/2  \hspace{1cm} up E
son 2  \hspace{1cm} C:  putref 3  \hspace{1cm} putref 4
uvar 5  \hspace{1cm} ustruct l/2  \hspace{1cm} D:  check 4
up C  \hspace{1cm} son 1  \hspace{1cm} putstruct l/2
move 6 3
jump app/3

36  Trimming of Stack Frames

Idea

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

Example

Consider the clause:

\[ a(X, Z) \leftarrow p_1(\tilde{X}, X_1), p_2(\tilde{X}_1, X_2), p_3(\tilde{X}_2, X_3), p_4(\tilde{X}_3, \tilde{Z}) \]
After every non-last goal with dead variables, we insert the instruction \texttt{trim}:

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

\[\text{trim } m\]

36  Trimming of Stack Frames

\textbf{Idea}

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

\textbf{Example}

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

After the literal \[p_2(X_1, X_2)\], variable \[X_1\] is dead.
After the literal \[p_3(X_2, X_3)\], variable \[X_2\] is dead.

327

328

\textbf{Example (continued)}

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

Ordering of the variables:
\[\rho = \{ X \mapsto 1, Z \mapsto 2, X_3 \mapsto 3, X_1 \mapsto 4, X_1 \mapsto 5 \}\]

The resulting code:

\begin{verbatim}
pushenv 5  A:  mark B  A:  mark C  A:  lastmark
mark A  A:  putref 5  A:  putref 4  A:  putref 3
putref 1  A:  putvar 4  A:  putvar 3  A:  putref 2
A:  call p_1/2  B:  trim 4  C:  trim 5
\end{verbatim}

327

328
37 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 !!!

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:

```plaintext
VAR:  setbtp  // variables  NIL:  jump A1  // atom []
      try A1
      delbtp
      CONS:  jump A2  // constructor []
      jump A2
ELSE:  fail  // default
```

Example

The app-predicate:

```
app(X, Y, Z) ← X = [], Y = Z
app(X, Y, Z) ← X = [H|X'], Z = [H|Z'], 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 ?

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:

```plaintext
VAR:  setbtp  // variables  NIL:  jump A1  // atom []
      try A1
      delbtp
      CONS:  jump A2  // constructor []
      jump A2
ELSE:  fail  // default
```
Then we generate for the predicate p/k:

```
codep rr = p/k: putref 1
  getNode // extracts the root label
  index p/k // jumps to the try block
  tchains rr
  A1 : codep r1
  ...
  Am : codep rm
```

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 A2 // atom []
  try A3
delbtp   CONS: jump A2 // constructor []
  jump A2
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:

```
codep rr = p/k: putref 1
  getNode // extracts the root label
  index p/k // jumps to the try block
  tchains rr
  A1 : codep r1
  ...
  Am : codep rm
```

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:

```java
switch (H[S(SP)]) {
  case (S, fn): S[SP] = fn; break;
  case (A,a): S[SP] = a; break;
  case (R,_) : S[SP] = R;
  // }
```
The instruction \texttt{index p/k} performs an indexed jump to the appropriate try chain:

\begin{align*}
\text{PC} &= \text{map}(p/k,S[F]) \\
\text{SF} &\leftarrow i
\end{align*}

The function \texttt{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 \hspace{1em} \textbf{Extension: The Cut Operator}

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

Example

\begin{align*}
\text{branch}(X,Y) &\leftarrow p(X), \neg q\neg 1(X,Y) \\
\text{branch}(X,Y) &\leftarrow q(X,Y)
\end{align*}

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 ...