36 Clause Indexing

Observation:

Often, predicates are implemented by case distinction on the first argument.

\[\Rightarrow\] Inspecting the first argument, many alternatives can be excluded

\[\Rightarrow\] Failure is earlier detected

\[\Rightarrow\] Backtrack points are earlier removed.

\[\Rightarrow\] 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 ;-)
Idea:

- Introduce separate try chains for every possible constructor.
- Inspect the root node of the first argument.
- Depending on the result, perform an indexed jump to the appropriate try chain.

Assume that the predicate \( p/k \) is defined by the sequence \( rr \) of clauses \( r_1 \ldots r_m \).

Let \( \text{tchains} \ rr \) denote the sequence of try chains as built up for the root constructors occurring in unifications \( X_1 = t \).
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
- jump $A_2$
- CONS: jump $A_2$ // constructor [ ]
- ELSE: fail // default

It directly triggers backtracking :-(
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:

\[
\begin{align*}
\text{VAR:} & \quad \text{setbtp} \quad \text{\(\rightarrow\)} \quad \text{variables} \\
& \quad \text{try} \ A_1 \\
& \quad \text{delbtp} \\
& \quad \text{jump} \ A_2 \\
\text{NIL:} & \quad \text{jump} \ A_1 \\
\text{CONS:} & \quad \text{jump} \ A_2 \\
\text{ELSE:} & \quad \text{fail} \\
\end{align*}
\]

The new instruction fail takes care of any constructor besides \([\ ]\) and \([|]\) ...

\[
\text{fail} \quad = \quad \text{backtrack()}
\]

It directly triggers backtracking :-)
Then we generate for a predicate $p/k$:

\[
\begin{align*}
\text{code}_p \ rr & = \ \text{putref} \ 1 \\
\text{getNode} & \quad \text{// extracts the root label} \\
\text{index p/k} & \quad \text{// jumps to the try block} \\
\text{tchains} \ rr & \\
A_1 : \ & \text{code}_c \ r_1 \\
... & \\
A_m : \ & \text{code}_c \ r_m
\end{align*}
\]
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 \texttt{index p/k} performs an indexed jump to the appropriate try chain:

\begin{figure}
\centering
\includegraphics[width=\textwidth]{index_p_k_diagram.png}
\caption{Diagram illustrating the indexing process.}
\end{figure}

\begin{verbatim}
PC = map (p/k, S[SP]);
SP--;  
\end{verbatim}
The instruction \textbf{index p/k} performs an indexed jump to the appropriate try chain:

\[
\text{index p/k}
\]

PC = \text{map (p/k,S[SP])};
SP– –;

The function \textbf{map()} returns, for a given predicate and node content, the start address of the appropriate try chain \textbf{:-)}.

It typically is defined through some hash table \textbf{:-))}. 

326
37 Extension: The Cut Operator

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

Example:

\[
\begin{align*}
\text{branch}(X, Y) & \leftarrow p(X), !, q_1(X, Y) \\
\text{branch}(X, Y) & \leftarrow q_2(X, Y)
\end{align*}
\]

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 ...
The Basic Idea:

- We restore the oldBP from our current stack frame;
- We pop all stack frames on top of the local variables.

Accordingly, we translate the cut into the sequence:

```
prune
pushenv m
```

where \( m \) is the number of (still used) local variables of the clause.
Example:

Consider our example:

\[
\begin{align*}
\text{branch}(X, Y) & \leftarrow \text{p}(X), !, q_1(X, Y) \\
\text{branch}(X, Y) & \leftarrow q_2(X, Y)
\end{align*}
\]

We obtain:

\[
\begin{align*}
\text{setbtp} & \quad \text{A: pushenv 2} \quad \text{C: prune} \quad \text{lastmark} \\
\text{try A} & \quad \text{mark C} \quad \text{pushenv 2} \quad \text{putref 1} \\
\text{delbtp} & \quad \text{putref 1} \quad \text{putref 2} \\
\text{jump B} & \quad \text{call p}/1 \quad \text{lastcall q}_1/2 \; 2 \\
\text{B: pushenv 2} & \quad \text{putref 2} \\
\text{jump q}_2/2
\end{align*}
\]
Example:

Consider our example:

\[
\text{branch}(X, Y) \leftarrow p(X), !, q_1(X, Y)
\]
\[
\text{branch}(X, Y) \leftarrow q_2(X, Y)
\]

In fact, an optimized translation even yields here:

<table>
<thead>
<tr>
<th>setbtp</th>
<th>A: pushenv 2</th>
<th>C: prune</th>
<th>putref 1</th>
<th>B: pushenv 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>try A</td>
<td>mark C</td>
<td>pushenv 2</td>
<td>putref 2</td>
<td>putref 1</td>
</tr>
<tr>
<td>delbtp</td>
<td>putref 1</td>
<td></td>
<td></td>
<td>putref 2</td>
</tr>
<tr>
<td>jump B</td>
<td>call p/1</td>
<td></td>
<td>jump q_1/2</td>
<td>move 2 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>move 2 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>jump q_2/2</td>
</tr>
</tbody>
</table>
The new instruction \textit{prune} simply restores the backtrack pointer:

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure}
\end{figure}

\[ BP = B_{\text{Pold}}; \]
Problem:

If a clause is single, then (at least so far ;-) we have not stored the old BP inside the stack frame :-(

⇒

For the cut to work also with single-clause predicates or try chains of length 1, we insert an extra instruction setcut before the clausal code (or the jump):
The instruction `setcut` just stores the current value of `BP`:

```
BPold = BP;
```
The Final Example: Negation by Failure

The predicate \( \text{notP} \) should succeed whenever \( p \) fails (and vice versa : -)

\[
\text{notP}(X) \leftarrow p(X), !, \text{fail}
\]

where the goal fail never succeeds. Then we obtain for \( \text{notP} \):

```
setbtp A: pushenv 1 C: prune B: pushenv 1
try A mark C pushenv 1 popenv
delbtp putref 1 fail
jump B call p/1 popenv
```
38 Garbage Collection

- Both during execution of a MaMa- as well as a WiM-programs, it may happen that some objects can no longer be reached through references.

- Obviously, they cannot affect the further program execution. Therefore, these objects are called garbage.

- Their storage space should be freed and reused for the creation of other objects.

**Warning:**

The WiM provides some kind of heap de-allocation. This, however, only frees the storage of failed alternatives !!!
Operation of a stop-and-copy-Collector:

- Division of the heap into two parts, the to-space and the from-space — which, after each collection flip their roles.
- Allocation with new in the current from-space.
- In case of memory exhaustion, call of the collector.

The Phases of the Collection:

1. Marking of all reachable objects in the from-space.
2. Copying of all marked objects into the to-space.
3. Correction of references.
4. Exchange of from-space and to-space.
(1) **Mark**: Detection of live objects:

- all references in the stack point to live objects;
- every reference of a live object points to a live object.

\[\rightarrow\]

**Graph Reachability**
(2) **Copy:** Copying of all live objects from the current *from-space* into the current *to-space*. This means for every detected object:

- Copying the object;
- Storing a forward reference to the new place at the old place  :-)

⇒

all references of the copied objects point to the forward references in the *from-space*.
Traversing of the to-space in order to correct the references.
(4) Exchange of to-space and from-space.
Warning:

The garbage collection of the WiM must harmonize with backtracking.

This means:

- The relative position of heap objects must not change during copying.
- The heap references in the trail must be updated to the new positions.
- If heap objects are collected which have been created before the last backtrack point, then also the heap pointers in the stack must be updated.
Classes and Objects
Example:

```cpp
int count = 0;

class list {

    int info;
    class list * next;

    list (int x) {
        info = x; count++; next = null;
    }

    virtual int last () {
        if (next == null) return info;
        else return next -> last ();
    }
}
```
Discussion:

- We adopt the C++ perspective on classes and objects.
- We extend our implementation of C. In particular ...
- Classes are considered as extensions of structs. They may comprise:
  - attributes, i.e., data fields;
  - constructors;
  - member functions which either are virtual, i.e., are called depending on the run-time type or non-virtual, i.e., called according to the static type of an object :-)
  - static member functions which are like ordinary functions :-))
- We ignore visibility restrictions such as public, protected or private but simply assume general visibility.
- We ignore multiple inheritance :-)

358
39 Object Layout

Idea:

- Only attributes and virtual member functions are stored inside the class !!
- The addresses of non-virtual or static member functions as well as of constructors can be resolved at compile-time :-) 
- The fields of a sub-class are appended to the corresponding fields of the super-class ...

... in our Example:

```
<table>
<thead>
<tr>
<th>info</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td></td>
</tr>
<tr>
<td>last</td>
<td></td>
</tr>
</tbody>
</table>
```
Idea (cont.):

- The fields of a sub-class are appended to the corresponding fields of the super-class :-)

Example:

```python
class mylist : list {
    int moreInfo;
}
```

... results in:

```
info
next
last
moreInfo
```