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:
The app-predicate:

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

- 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 \( \text{chains} rr \) denote the sequence of try chains as built up for the root constructors occurring in unifications \( X_0 = t \).
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: setbip // variables NIL: jump $A_1$ // atom []
try $A_1$
delbip CONS: jump $A_2$ // constructor [[]]
jump $A_2$
ELSE: fail // default
```

The new instruction `fail` takes care of any constructor besides [ ] and [[]]...

```
fail = backtrack()
```

It directly triggers backtracking :-)

---

Then we generate for a predicate $p/k$:

```
code$p$ rr = putref 1
getNode // extracts the root label
index p/k // jumps to the try block
tchains rr
$A_1$ : code$r_1$
... $A_n$ : code$r_n$
```

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 (HS[SP]) {
case (S, f/n):
    S[SP] = f/n; break;
case (A,a):
    S[SP] = a; break;
case (R,):    S[SP] = R;
}
```
The instruction `index p/k` performs an indexed jump to the appropriate try chain:

PC = map [p/k,S[SP]];
SP−−;

The function `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 Extension: The Cut Operator

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

**Example:**

\[
\text{branch}(X, Y) \leftarrow p(X), l_1(X, Y) \\
\text{branch}(X, Y) \leftarrow q_1(X, Y)
\]

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 ...
38 Extension: The Cut Operator

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

Example:

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

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:

\[
\text{prune} \\
\text{pushenv } m
\]

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

38 Extension: The Cut Operator

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

Example:

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

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 ...

38 Extension: The Cut Operator

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

Example:

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

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.

---

38 Extension: The Cut Operator

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

Example:

```
branch(X, Y) ← p(X), q(X, Y)
branch(X, Y) ← q(X, Y)
```

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 ...
Example:

Consider our example:

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

We obtain:

\[
\begin{align*}
\text{setbtp} & \quad \text{A: pushenv 2} & \quad \text{C: prune} & \quad \text{B: pushenv 2} \\
\text{try A} & \quad \text{mark C} & \quad \text{putref 1} & \quad \text{putref 2} \\
\text{delbtp} & \quad \text{putref 2} & \quad \text{lastcall q_1/2 2} & \quad \text{putref 2} \\
\text{jump B} & \quad \text{call p/1} & \quad \text{move 2 2} & \quad \text{jump q_2/2}
\end{align*}
\]

In fact, an \textbf{optimized} translation even yields here:

\[
\begin{align*}
\text{setbtp} & \quad \text{A: pushenv 2} & \quad \text{C: prune} & \quad \text{putref 1} & \quad \text{B: pushenv 2} \\
\text{try A} & \quad \text{mark C} & \quad \text{putref 2} & \quad \text{putref 2} & \quad \text{putref 1} \\
\text{delbtp} & \quad \text{putref 2} & \quad \text{move 2 2} & \quad \text{jump q_2/2} \\
\text{jump B} & \quad \text{call p/1} & \quad \text{jump q_2/2} & \quad \text{move 2 2}
\end{align*}
\]

The new instruction \textbf{prune} simply restores the backtrack pointer:

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

Problem:

If a clause is \textbf{single}, then (at least so far >) we have not stored the old BF inside the stack frame >

For the cut to work also with \textbf{single-clause} predicates or \textbf{try} chains of length 1, we insert an extra instruction \textbf{setcut} before the clausal code (or the jump):
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 \texttt{setcut} before the clausal code (or the jump):

The instruction \texttt{setcut} just stores the current value of BP:

\[
\text{BPold} = \text{BP};
\]

The Final Example: \hspace{1cm} Negation by Failure

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

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

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

- \texttt{setbtp}
- \texttt{try A}
- \texttt{delbop}
- \texttt{jump B}
- \texttt{A: pushenv 1}
- \texttt{mark C}
- \texttt{putref 1}
- \texttt{call p/1}
- \texttt{C: prune}
- \texttt{pushenv 1}
- \texttt{fail}
- \texttt{popenv}

39 \hspace{1cm} Garbage Collection

- Both during execution of a \texttt{MaMa}- as well as a \texttt{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 \texttt{WiM} provides some kind of heap de-allocation. This, however, only frees the storage of failed alternatives 😢

344
39 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.

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.
(3) Traversing of the to-space in order to correct the references.
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:

```java
int count = 0;
class List {
    int info;
    List list = next;
    List (int x) {
        info = x; count++; next = null;
    }
    virtual int last () {
        if (next == null) return info;
        else return next -> last();
    }
}
```