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:

\[
\begin{align*}
\text{prune} \\
\text{pushenv } m
\end{align*}
\]

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

38 Extension: The Cut Operator

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

Example

\[
\begin{align*}
\text{branch}(X,Y) & \leftarrow \text{p}(X), \text{l}_1(X,Y) \\
\text{branch}(X,Y) & \leftarrow \text{l}_2(X,Y)
\end{align*}
\]

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 ...

Example

Consider our example:

\[
\begin{align*}
\text{branch}(X,Y) & \leftarrow \text{p}(X), \text{l}_1(X,Y) \\
\text{branch}(X,Y) & \leftarrow \text{l}_2(X,Y)
\end{align*}
\]

We obtain:

\[
\begin{align*}
\text{setbp} & \quad \text{A}: \quad \text{pushenv} 2 \\
\text{try} A & \quad \text{mark} C \\n\text{delbp} & \quad \text{putref} 1 \\
\text{jump} B & \quad \text{call} p/1 \\
\text{putref} 1 & \quad \text{lastcall} \text{l}_2/2 \\
\text{putref} 2 & \quad \text{move} 2 \\
\text{putref} 2 & \quad \text{jump} \text{l}_2/2
\end{align*}
\]
Example

Consider our example:

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

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

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>delete</td>
<td>putref 1</td>
<td>move 2</td>
<td>putref 2</td>
<td>move 2</td>
</tr>
<tr>
<td>jump B</td>
<td>call p/1</td>
<td>jump q_1/2</td>
<td>move 2</td>
<td>jump q_1/2</td>
</tr>
</tbody>
</table>

Problem

\[
\overrightarrow{p(X)} \leftarrow q_1(X), q_2(X)
\]

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

\[\Rightarrow\]

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 Final Example

Negation by Failure

The predicate 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 notP:

\[
\text{setbp} \quad \text{A: pushenv} \quad \text{C: prune} \quad \text{B: pushenv}
\]

\[
\text{try A} \quad \text{mark C} \quad \text{pushenv} \quad \text{popenv}
\]

\[
\text{delbp} \quad \text{putref} \quad \text{fail}
\]

\[
\text{jump B} \quad \text{call p/1} \quad \text{popenv}
\]

346

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.

Caveats

The WiM provides some kind of heap de-allocation. This, however, only frees the storage of failed alternatives. !!!

347

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.

348
(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.

\[ \implies \]

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.
Caveat

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.

Remarks

- Marking, copying and placing a forward reference can be squeezed into a single pass.
  A second pass then is only required to correct the references.
- If the heap objects are traversed in post-order, most of the references can be corrected in the same pass.
  Only references to not yet copied objects must be patched later-on.
- Overall, the run-time of gc is proportional only to the number of live objects.

I think

- Concurrent
- Quadruple collector
Remarks

- Marking, copying and placing a forward reference can be squeezed into a single pass.
  A second pass then is only required to correct the references.
- If the heap objects are traversed in post-order, most of the references can be corrected in the same pass.
  Only references to not yet copied objects must be patched later-on.
- Overall, the run-time of gc is proportional only to the number of live objects.

Remarks

- While marking still visits only live objects, copying requires a separate sequential pass over the from-space.
- Therefore, the run-time of copying is proportional to the total amount of from-space.
Classes and Objects

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.

Example

```c
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();
    }
}
```

40 Object Layout

Idea

- Only attributes and virtual member functions are stored inside the class II.
- 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:

```
    info
    next
    last
```
Idea (cont.)

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

Example

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

... results in:

```
info
next
last
moreInfo
```

374

Idea (cont.)

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

Example

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

... results in:

```
info
next
last
moreInfo
```