Script generated by TTT

Title: Seidl: Virtual_Machines (13.06.2016)

Date: Mon Jun 13 10:23:34 CEST 2016

Duration: 89:50 min

Pages: 33

38 Extension: The Cut Operator

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

Example

$$\begin{array}{lcl} \mathsf{branch}(X,Y) & \leftarrow & \boxed{\mathsf{p}(X)}, \boxed{!}, \boxed{\mathsf{q}_1(X,Y)} \\ \mathsf{branch}(X,Y) & \leftarrow & \boxed{\mathsf{q}_2(X,Y)} \end{array}$$

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:

$$\begin{array}{lcl} \mathsf{branch}(X,Y) & \leftarrow & \mathsf{p}(X), !, \mathsf{q}_1(X,Y) \\ \mathsf{branch}(X,Y) & \leftarrow & \mathsf{q}_2(X,Y) \end{array}$$

We obtain:

setbtp	A:	pushenv 2	C:	prune	lastmark	B:	pushenv 2
try A		mark C		pushenv 2	putref 1		putref 1
delbtp		putref 1			putref 2		putref 2
jump B		call p/1			lastcall q_1/2 2		move 2 2
							jump q_2/2

Example

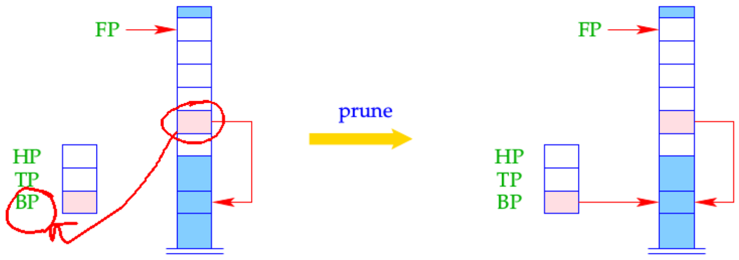
Consider our example:

$$\mathsf{branch}(X,Y) \leftarrow \mathsf{p}(X), !, \mathsf{q}_1(X,Y)$$
$$\mathsf{branch}(X,Y) \leftarrow \mathsf{q}_2(X,Y)$$

In fact, an optimized translation even yields here:

setbtp	A:	pushenv 2	C:	prune	putref 1	B:	pushenv 2
try A		mark C		pushenv 2	putref 2		putref 1
delbtp		putref 1			move 2 2		putref 2
jump B		call p/1			jump $q_1/2$		move 2 2
							jump $q_2/2$

The new instruction prune simply restores the backtrack pointer:



BP = BPold;

Problem

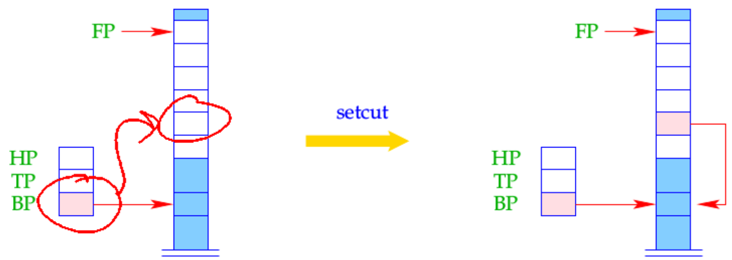
p(X) :- q_1(X), !, q_2(X).

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

$\Longrightarrow$

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 notP should succeed whenever p fails (and vice versa :-)

$$\begin{array}{lcl} \mathrm{notP}(X) & \leftarrow & \mathrm{p}(X), !, \mathrm{fail} \\ \mathrm{notP}(X) & \leftarrow & \end{array}$$

where the goal fail never succeeds. Then we obtain for 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
```


X is Y+3

The Final Example — Negation by Failure

The predicate notP should succeed whenever p fails (and vice versa :-)

$$\begin{array}{lcl} \mathrm{notP}(X) & \leftarrow & \mathrm{p}(X), !, \mathrm{fail} \\ \mathrm{notP}(X) & \leftarrow & \end{array}$$

where the goal fail never succeeds. Then we obtain for 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
```


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.

Caveat

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

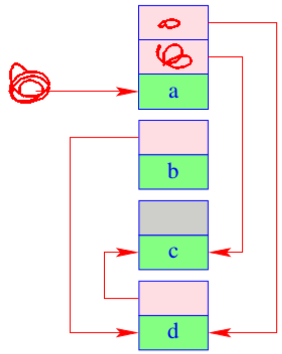
1. Marking of all reachable objects in the from-space.
2. Copying of all marked objects into the to-space.
3. Correction of references.
4. Exchange of from-space and to-space.

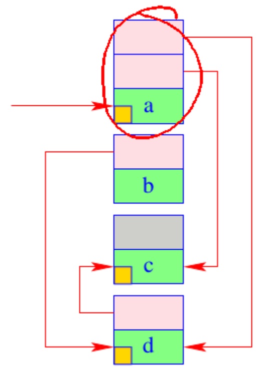
(1) Mark: Detection of live objects:

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

$\Longrightarrow$

Graph Reachability



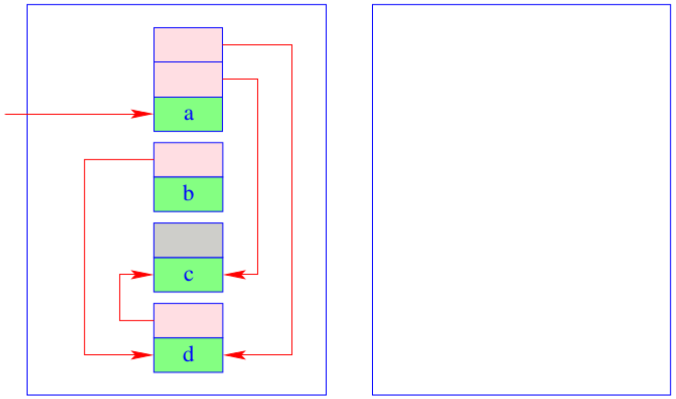


(2) Copy: Copying of all live objects from the current from-space into the current to-space. This means for every detected object:

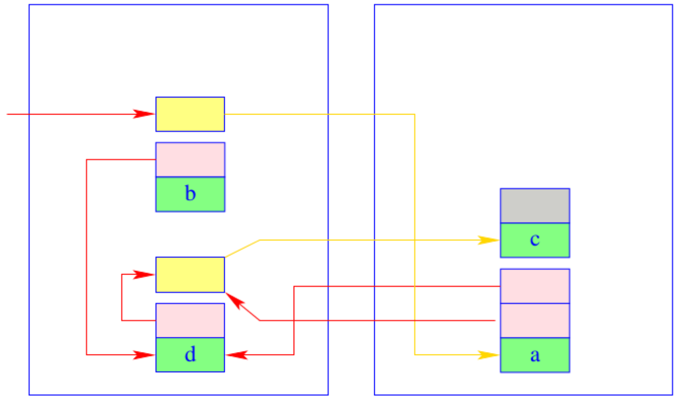
- Copying the object;
- Storing a forward reference to the new place at the old place

$\Longrightarrow$

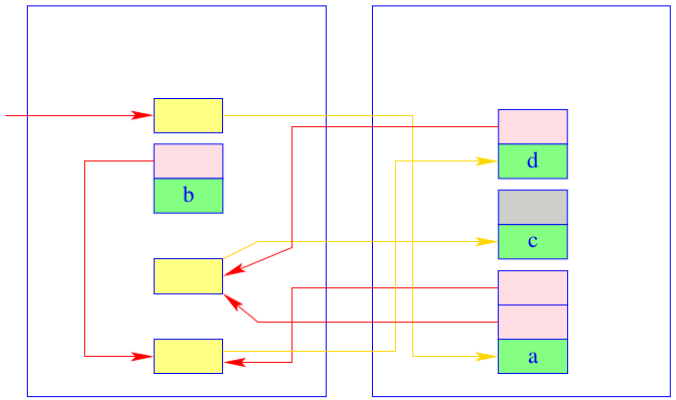
all references of the copied objects point to the forward references in the from-space.



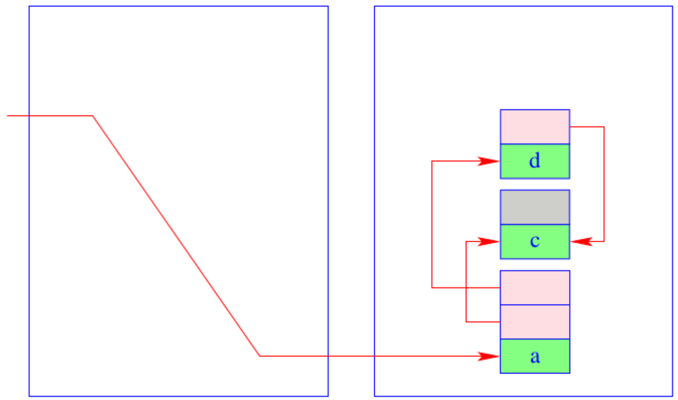
353



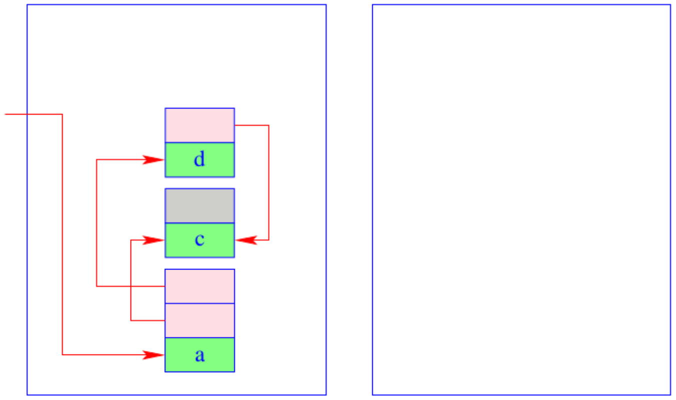
355



356



360



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.

Issues
- Concurrency
- generational collection

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.

Caveat

The garbage collection of the WiM must harmonize with backtracking.

This means:

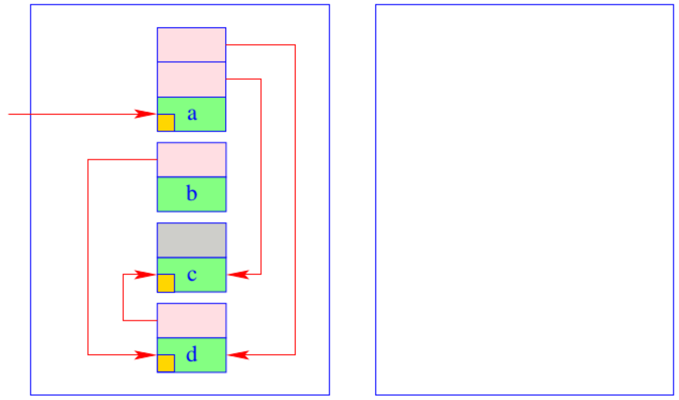
- The relative position of heap objects must not change during copying!
- The heap references in the trail must be updated to the new positions.
- If heap objects are collected which have been created before the last backtrack point, then also the heap pointers in the stack must be updated.

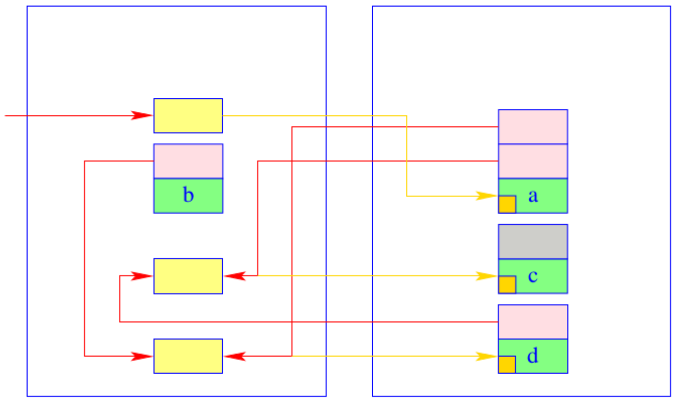
Remarks

- 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.





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

Example

```
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.

40 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:

info	
next	
last	

Idea (cont.)

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

Example

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

... results in:

info	
next	
last	
moreInfo	

Idea (cont.)

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

Example

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

... results in:

info	
next	
last	
moreInfo	

374