Script generated by TTT

Title: Seidl: Virtual_Machines (11.06.2013)

Date: Tue Jun 11 14:02:23 CEST 2013

Duration: 88:29 min

Pages: 46

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 $r_1 \dots r_m$.

Let 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.

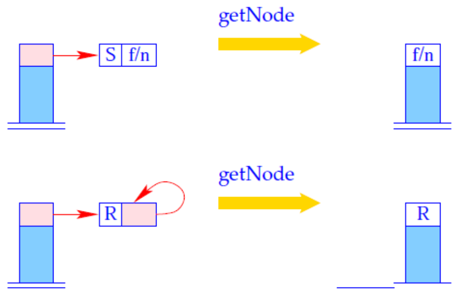
Then we obtain the following four try chains:

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

Then we generate for a predicate p/k:

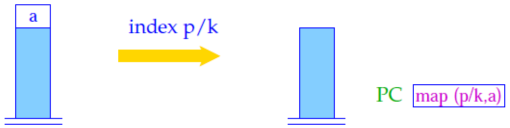
$\text{code}_P\ rr$	=	putref 1	
		getNode	// extracts the root label
		index p/k	// jumps to the try block
		tchains rr	
	A_1 :	$\text{code}_C\ r_1$	
		...	
	A_m :	$\text{code}_C\ r_m$	

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:



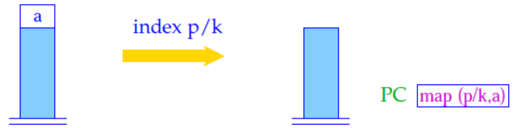

```
switch (H[S[SP]]) {
case (S, f/n):    S[SP] = f/n; break;
case (A,a):       S[SP] = a; break;
case (R,_) :      S[SP] = R;
}
```


The instruction index p/k performs an indexed jump to the appropriate try chain:




```
PC = map (p/k,S[SP]);
SP−−;
```


The instruction index p/k performs an indexed jump to the appropriate try chain:

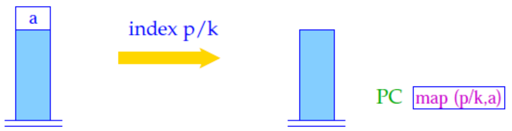



```
PC = map (p/k,S[SP]);
SP−−;
```


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 :-))

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 :-))

37 Extension: The Cut Operator

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

Example:

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

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 ...

37 Extension: The Cut Operator

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

Example:

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

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.

37 Extension: The Cut Operator

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

Example:

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

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.

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 2
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:

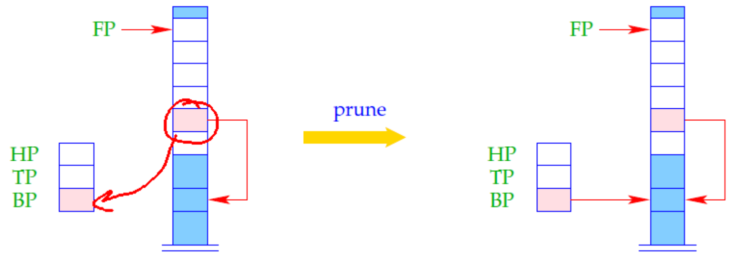
```
     setbtp
     try A
     delbtp
     jump B

A:   pushenv 2
     mark C
     putref 1
     call p/1

C:   prune
     pushenv 2
     putref 1
     putref 2
     move 2 2
     jump q1/2

B:   pushenv 2
     putref 1
     putref 2
     move 2 2
     jump q2/2
```


The new instruction prune simply restores the backtrack pointer:



BP = BPold;

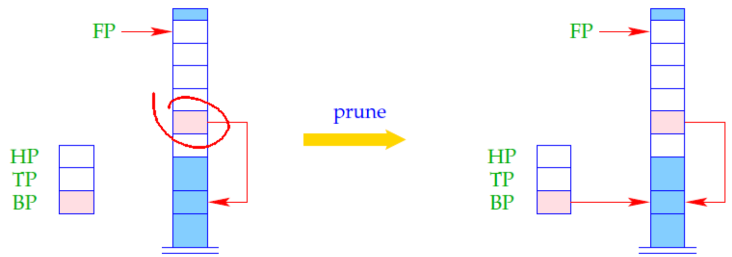
Problem:

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 new instruction prune simply restores the backtrack pointer:



BP = BPold;

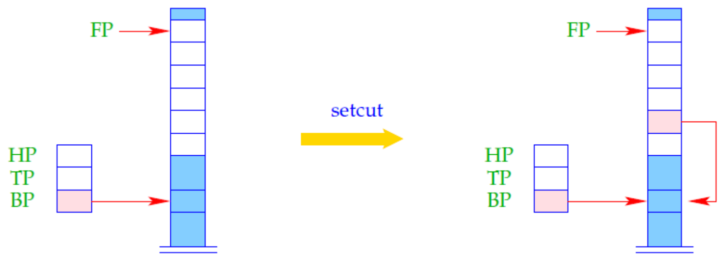
Problem:

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 :-)

$$\begin{array}{lll} \mathsf{notP}(X) & \leftarrow & \boxed{\mathsf{p}(X)}, !, \mathsf{fail} \\ \boxed{\mathsf{notP}(X) \quad \leftarrow} & & \end{array}$$

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 $\mathrm{p}/1$		popenv		

The Final Example: Negation by Failure

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

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

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 $\mathrm{p}/1$		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.

$\Longrightarrow$

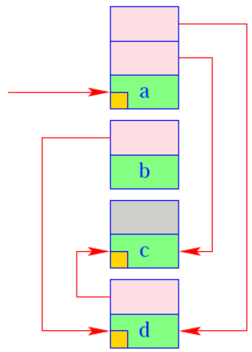
Graph Reachability

(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

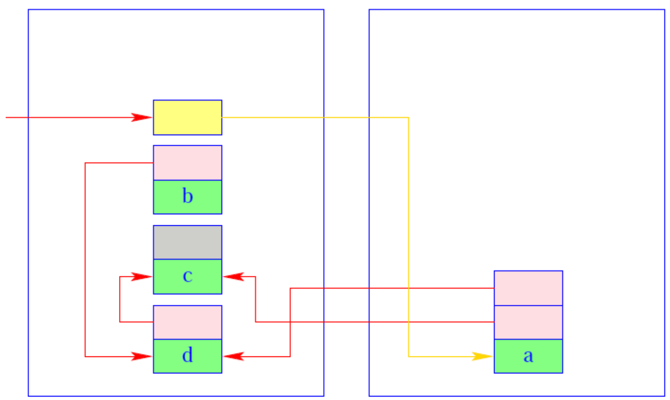


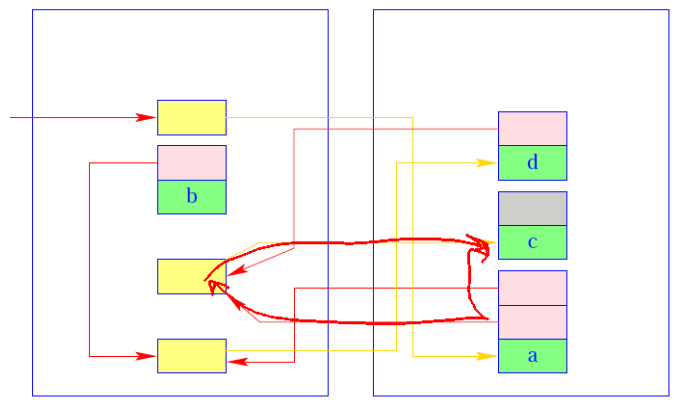
(2) Copy: Copying of all live objects from the current from-space into the current to-space. This means for every detected object:

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

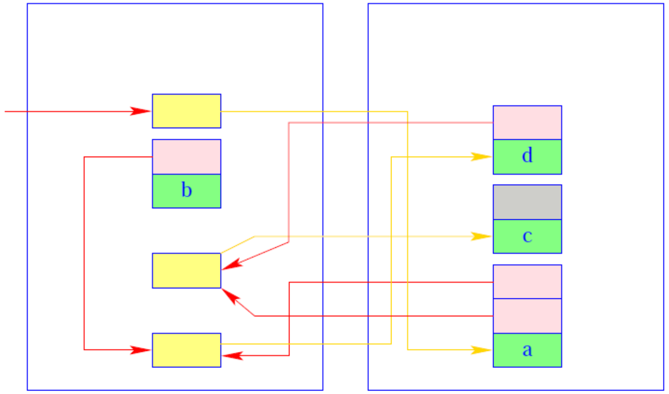
$\Longrightarrow$

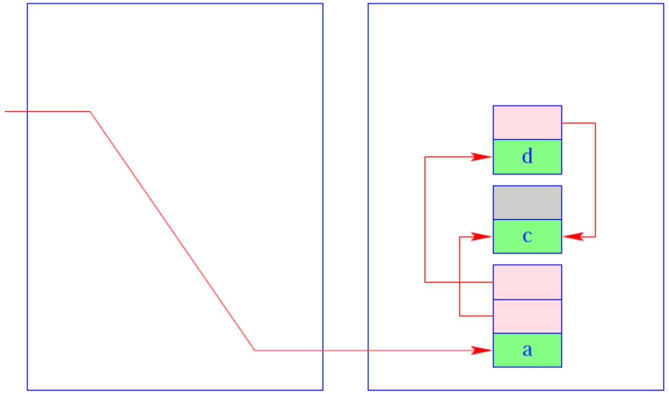
all references of the copied objects point to the forward references in the from-space.





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



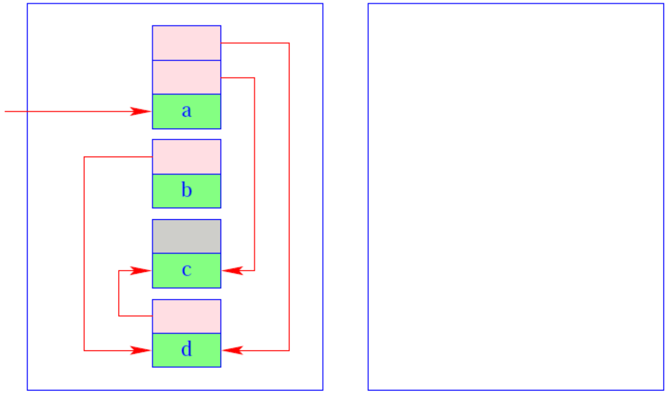


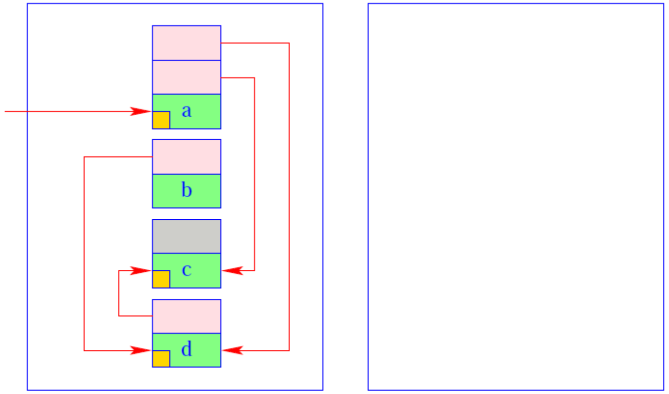
Warning:

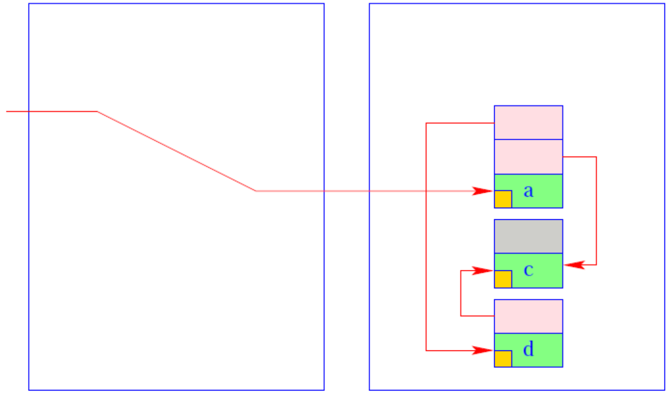
The garbage collection of the WiM must harmonize with backtracking.

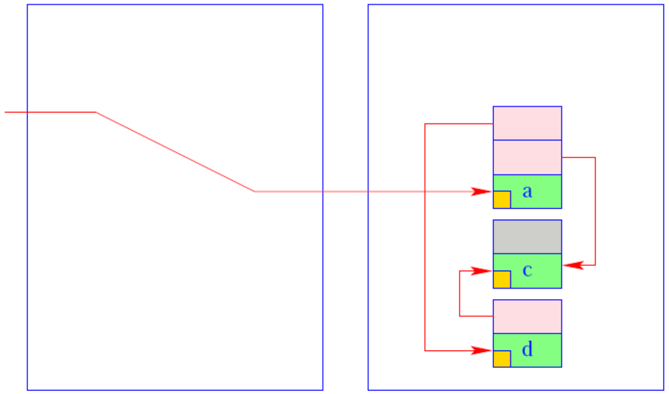
This means:

- 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:

```
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:

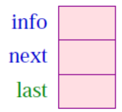
- We adopt the C++ perspective on classes and objects.
- We extend our implementation of C. In particular ...
- Classes are considered as extensions of structs. They may comprise:
  - ⇒ attributes, i.e., data fields;
  - ⇒ constructors;
  - ⇒ member functions which either are virtual, i.e., are called 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 :-)

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:

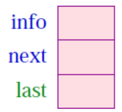


39 Object Layout

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:

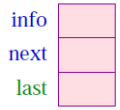


39 Object Layout

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:



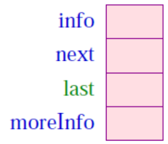
Idea (cont.):

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

Example:

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

... results in:



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: