Script generated by TTT

Title: Seidl: Virtual_Machines (23.06.2015)

Date: Tue Jun 23 10:15:52 CEST 2015

Duration: 89:23 min

Pages: 43

37 Clause Indexing

Observation

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

$\Longrightarrow$ Inspecting the first argument, many alternatives can be excluded :-)

$\Longrightarrow$ Failure is earlier detected :-)

$\Longrightarrow$ Backtrack points are earlier removed. :-))

$\Longrightarrow$ Stack frames are earlier popped :-)))

Example

The app-predicate:

$$\text{app}(\boxed{X}, Y, Z) \leftarrow X = [\,],\ Y = Z$$
$$\text{app}(\boxed{X}, Y, Z) \leftarrow X = [H|X'],\ Z = [H|Z'],\ \text{app}(X', Y, Z')$$

- 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 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		CONS:	jump A_2	// constructor [|]
	jump A_2				
			ELSE:	fail	// default

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		CONS:	jump A_2	// constructor [|]
	jump A_2				
			ELSE:	fail	// default

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

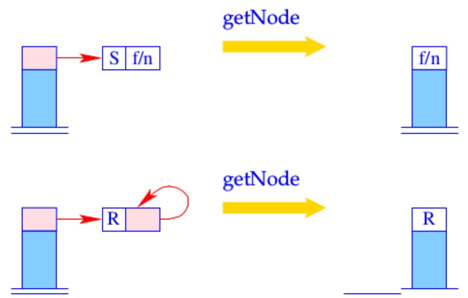
fail = `backtrack()`

It directly triggers backtracking :-)

Then we generate for the predicate p/k:

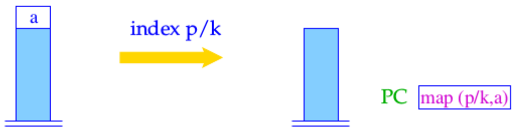
$\text{code}_P\ rr$	=	p/k:	putref 1	
			getNode	// extracts the root label
			index p/k	// jumps to the try block
			tchains rr	
		A_1 :	$\text{code}_C\ r_1$	
			...	
		A_m :	$\text{code}_C\ r_m$	

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:



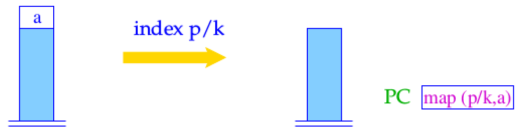

```
switch (H[S[SP]]) {
case (S, f/n):    S[SP] = f/n; break;
case (A,a):       S[SP] = a; break;
case (R,_) :      S[SP] = R;
}
```

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 :-))

38 Extension: The Cut Operator

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

Example

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

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

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

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:

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

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

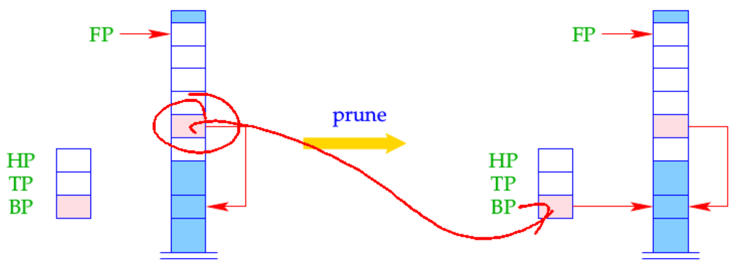
Consider our example:

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

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;

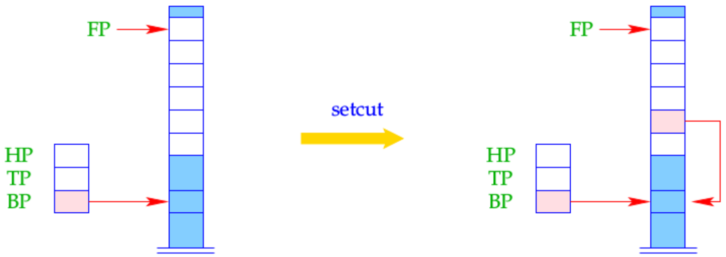
Problem

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

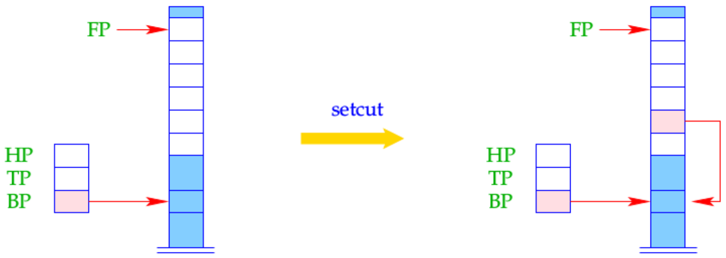
The predicate notP should succeed whenever p fails (and vice versa :-)

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

where the goal fail never succeeds. Then we obtain for notP :

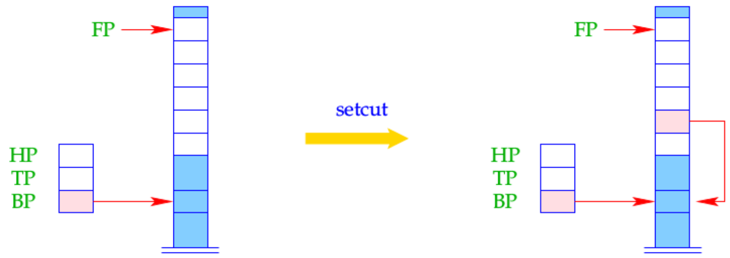
```
setbtp      A:  pushenv 1    C:  prune        B:  pushenv 1
try A           mark C           pushenv 1        popenv
delbtp          putref 1         fail
jump B          call p/1         popenv
```

The instruction setcut just stores the current value of BP:



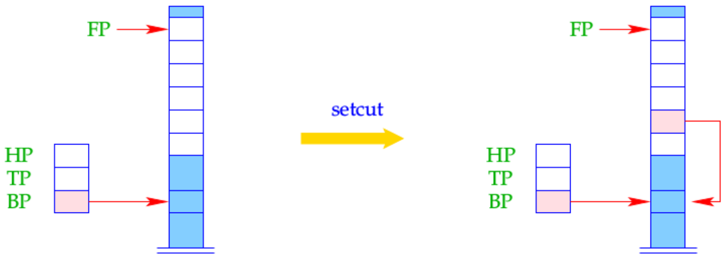
BPold = BP;

The instruction setcut just stores the current value of BP:



BPold = BP;

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 :-)

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

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
```

The Final Example   Negation by Failure

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

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

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:

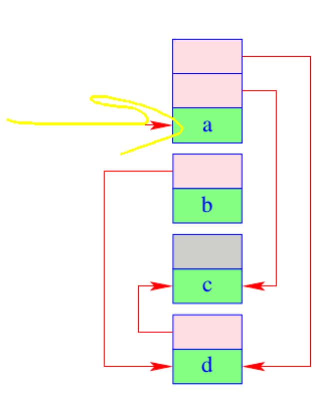
1. Marking of all reachable objects in the from-space.
2. Copying of all marked objects into the to-space.
3. Correction of references.
4. Exchange of from-space and to-space.

(1) Mark: Detection of live objects:

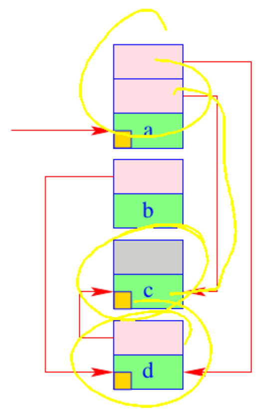
- all references in the stack point to live objects;
- every reference of a live object points to a live object.

$\Longrightarrow$

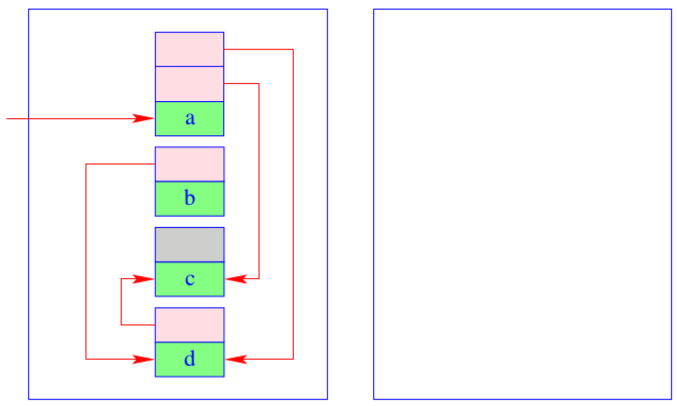
Graph Reachability



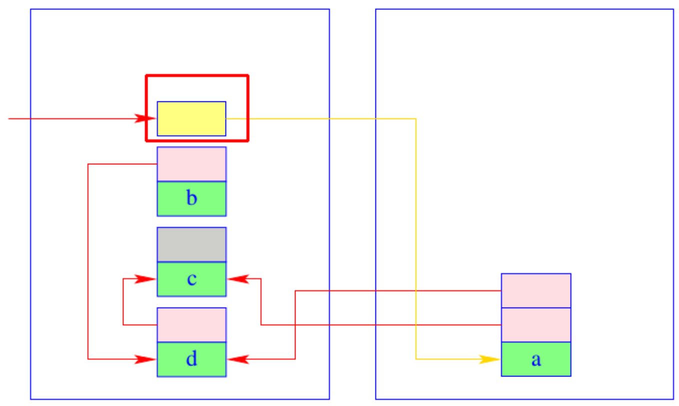
350



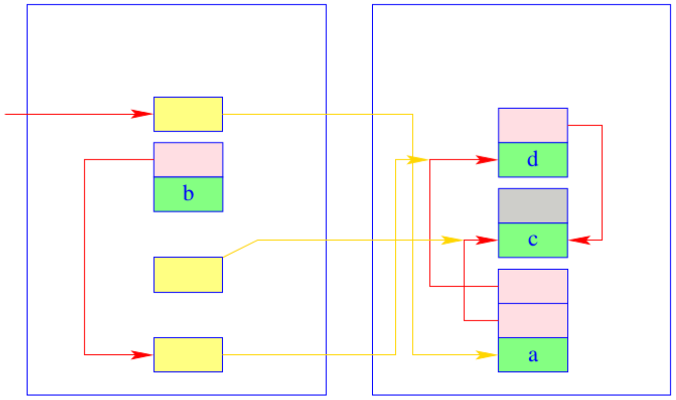
351



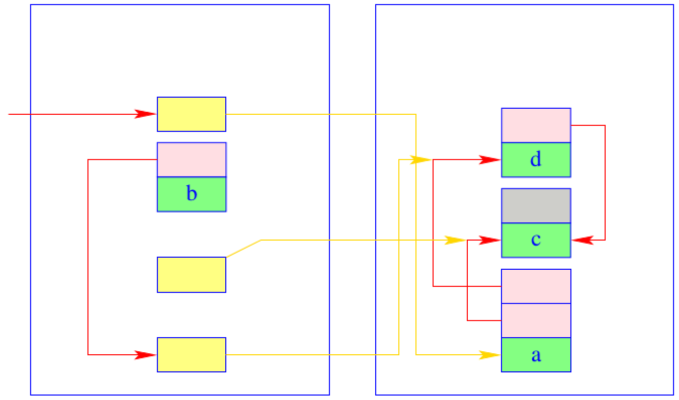
353



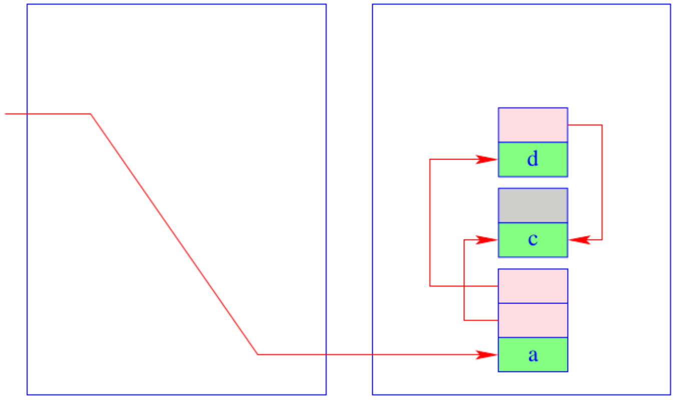
354



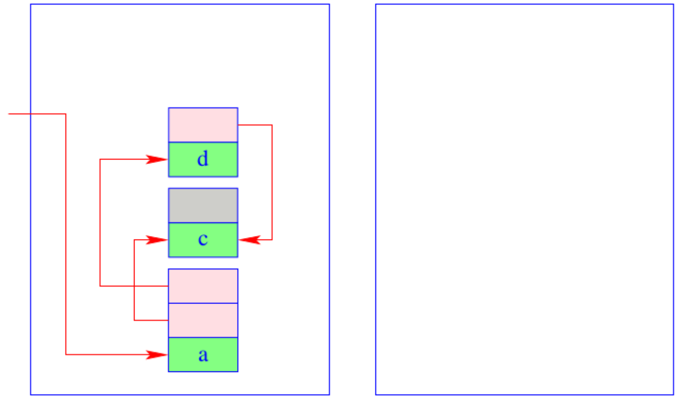
b
d
c
a



b
d
c
a



d
c
a



d
c
a

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

- 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.

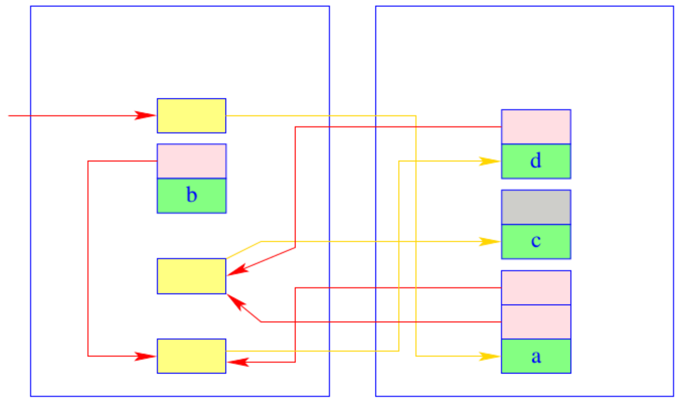
363

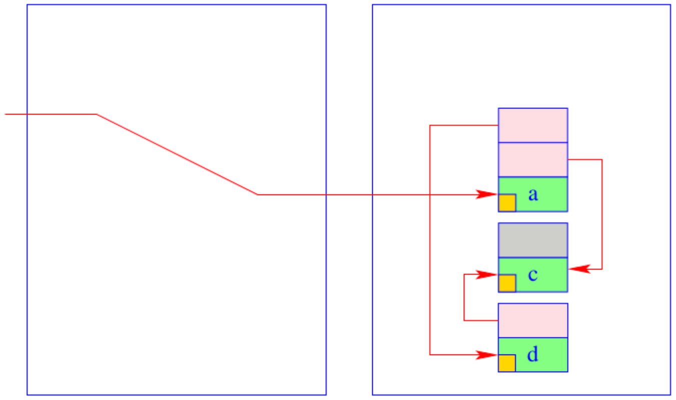
Caveat

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

- 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 :-(

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