Online appendix for the paper

Tabling as a Library with Delimited Control

published in Theory and Practice of Logic Programming

BENOIT DESOUTER and MARKO VAN DOOREN

Department of Applied Mathematics, Computer Science and Statistics, Ghent University, Belgium
benoit.desouter,marko.vandooren@ugent.be

TOM SCHRIJVERS

Department of Computer Science, KU Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

submitted 29 April 2015; revised 5 June 2015; accepted 5 June 2015

Appendix A Full Benchmark Results

The following tables show the absolute execution times for the benchmarks from Table 2. For clarity, we have also duplicated the relative timing information.

Benchmark	hProlog	XSB		B-Prolog	
		Absolute	Relative	Absolute	Relative
fib					
500	24	O/F	—	0	∞
750	33	O/F	—	2	17
1000	46	O/F	—	1	46
10000	982	O/F	—	370	3
recognize					
20000	205	8	26	59490	0.003
50000	503	17	30	377295	0.001
n-reverse					
500	767	20	38	71	11
1000	2800	90	31	444	6
shuttle					
2000	44	0	∞	459	0.1
5000	138	6	23	1833	0.08
20000	582	24	24	27499	0.02
50000	1586	54	29	172098	0.01
pingpong					
10000	271	6	45	4038	0.07
20000	490	14	35	16222	0.03
path double first loop					
50	653	34	19	52	13
100	4638	266	17	449	10

Benchmark	hProlog	XSB		B-Prolog	
		Absolute	Relative	Absolute	Relative
path double first					
50	162	6	27	11	15
100	989	50	20	82	12
200	6785	371	18	434	16
500	110463	4371	25	5936	19
path right last: pyramid 500					
	1914	55	35	65	29
path right last: binary tree 18					
	108662	1390	78	2169	50
test large joins 2 (size 12)					
	3001	302	10	761	4
joins mondial					
	6444	810	8	939	7

Benchmark	Yap		Ciao	
	Absolute	Relative	Absolute	Relative
fib				
500	0	∞	—	—
750	1	41	—	—
1000	2	19	—	—
10000	22	44	—	—
recognize				
20000	18	11	46	4
50000	37	14	129	4
n-reverse				
500	50	15	17	45
1000	351	8	82	34
shuttle				
2000	0	∞	5	9
5000	0	∞	12	12
20000	0	∞	57	10
50000	0	∞	134	12
pingpong				
10000	0	∞	19	14
20000	0	∞	58	8
path double first loop				
50	0	∞	93	7
100	0	∞	815	6
path double first				
50	0	∞	12	14
100	0	∞	101	10

Benchmark	Yap		Ciao	
	Absolute	Relative	Absolute	Relative
200	0	∞	694	10
500	0	∞	7999	14
path right last: pyramid 500				
	0	∞	70	27
path right last: binary tree 18				
	31	3461	2606	42
test large joins 2 (size 12)				
	0	∞	256	12
joins mondial				
	29	224	1062	6

Appendix B Nonbacktrackable Variables and Term Mutation

In this appendix, we first describe the semantics and implementation effort of the predicates for nonbacktrackable global variables. Such variables are available in many popular Prolog implementations. The overhead of these features is only a small constant. Afterwards, we describe nonbacktrackable mutation. The descriptions are adapted from the SWI-Prolog website as it appeared on June 10, 2015.

Global Nonbacktrackable Variables

nb_setval(+Name, +Value) Associates with the atom `Name` without copying it. The semantics on backtracking to a point before creating the link are poorly defined for compound terms. The principal term is always left untouched, but backtracking behaviour on arguments is undone if the original assignment was trailed and left alone otherwise, which implies that the history that created the term affects the behaviour on backtracking. A `copy_term/2` can be used to avoid this.

nb_getval(+Name, -Value) Get the value associated with the global nonbacktrackable variable `Name` and unify it with `Value`. Note that this unification may further instantiate the value of the global variable.

In terms of implementation, `nb_setval/2` differs from `b_setval/2` by not trailing its argument and freezing the heap in the case of a list or struct:

```
if (has_atom_tag(p2)) /* smallint, char or atom */
    *p1 = (dlong)(p2);
else {
    *p1 = (dlong)lochreg;
    *lochreg = (dlong)p2;
    lochreg++;
    adapt_freeze_hreg(lochreg);
}
```

The heap backtrack pointer `HB` must be set to the top of the heap. This must also happen at every backtrack, for which you may introduce an extra register `FH`. The garbage collection phase also needs to be aware of this `FH` register.

Frozen heap is only reclaimed by garbage collection, but does not make more space reachable: this does not affect the copying phase. The additional cost is in computing the new frozen heap top. In the worst case, this is linear in the number of choicepoints, but the work required is (for each choicepoint) a simple addition. The overhead is really insignificant.

Nonbacktrackable Mutation For nonbacktrackable mutation, many Prologs provide a predicate `nb_setarg/3` that has the semantics defined below. This predicate uses the same technique as `nb_setval/2`.

nb_setarg(+Arg,+Term,+Value) Assigns the `Arg`-th argument of the compound term `Term` with the given `Value`. On backtracking the assignment is not reversed. The term `Value` is not duplicated before assignment.

The implementation can be made thread-safe, reentrant and capable of handling exceptions. Realising these features with a traditional implementation based on assert/retract or `flag/3` is much more complicated.

Appendix C Worklist Completion Code

This appendix gives more implementation details of the completion phase, discussed in Subsection 4.3.

To get answers and dependencies that must be combined from the local worklist, we first extract the worklist from the table, and in preparation, set a flag indicating that we are busy working. The actual work is delegated to `table_get_work_/3`.

```
table_get_work(Table,_Answer,_Dependency) :-
  get_worklist(Table,Worklist),
  set_flag_executing_all_work(Worklist),
  table_get_work_(Worklist,Answer,Dependency).
```

In `table_get_work_/3`, we once more delegate the work nondeterministically, but once an answer-dependency pair is extracted from the local worklist, we copy the dependency to ensure that the original version is not modified. When the first rule eventually fails, the work in this local worklist is done for now, so we unset the flag.

```
table_get_work_(Worklist,Answer,Dependency) :-
  worklist_do_all_work(Worklist,Answer,Dependency0),
  copy_term(Dependency0,Dependency).
table_get_work_(Worklist,_Answer,_Dependency) :-
  unset_flag_executing_all_work(Worklist), fail.
```

The job of nondeterministically extracting answer-dependency pairs is tackled by executing a single step and when this eventually fails, recursively calling yourself unless all the work is done.

```
worklist_do_all_work(Worklist,Answer,Dependency) :-
  ( worklist_work_done(Worklist) ->
    fail
  ;
    worklist_do_step(Worklist,Answer,Dependency)
  ;
    worklist_do_all_work(Worklist,Answer,Dependency)
  ).
```

The job of the local worklist is done for now if the pointer to the cluster of answers that should be combined with dependencies, points to a dummy value. Alternatively, the work is done if there is no dependency cluster, in which case the next entry in the underlying double linked list representation of the worklist points to the dummy value.

```
worklist_work_done(Worklist) :-
  wkl_get_rightmost_inner_answer_cluster_pointer(Worklist,RiacPointer),
  ( wkl_is_dummy_pointer(Worklist,RiacPointer) -> true
  ;
    dll_get_pointer_to_next(RiacPointer,NextPointer),
    wkl_is_dummy_pointer(Worklist,NextPointer)
  ).
```

Taking the Cartesian product of an answer and a dependency cluster happens by first swapping the clusters in local worklist. Next, the pointers to these clusters are dereferenced by the underlying double linked list representation. Finally, one answer and one dependency are nondeterministically yielded for combination.

```
worklist_do_step(Worklist,Answer,Dependency) :-
  wkl_get_rightmost_inner_answer_cluster_pointer(Worklist,ACP),
  wkl_swap_answer_continuation(Worklist,ACP,SCP),
  dll_get_data(ACP, wkl_answer_cluster(AList)),
  dll_get_data(SCP, wkl_suspension_cluster(SList)),
  member(Answer,AList), member(Dependency,SList).
```

Swapping clusters is propagated to the underlying double linked list representation. Afterwards, the pointer to the answer cluster that must be swapped next, must be updated to the new location of the cluster.

```
wkl_swap_answer_continuation(Worklist,ACP,SCP) :-
  dll_get_pointer_to_next(ACP,SCP),
  dll_swap_adjacent_elements_(ACP,SCP),
  wkl_update_righmost_inner_answer_cluster_pointer(Worklist,ACP).
```

Updating the pointer to the answer cluster that must be swapped next, is a no-op if the cluster currently pointed to still has to be combined with dependency clusters, and hence can still propagate to the right. Otherwise, a new cluster is found by walking back in the underlying double linked list representation.

```
wkl_update_righmost_inner_answer_cluster_pointer(Worklist,ACP) :-
  ( wkl_answer_cluster_currently_moved_completely(Worklist,ACP) ->
    wkl_find_new_rightmost_inner_answer_cluster_pointer(Worklist,ACP,ACP2),
    wkl_set_rightmost_inner_answer_cluster_pointer(Worklist,ACP2)
  ;
    true
  ).
```