On Improving the Efficiency of Deterministic Calls and Answers in Tabled Logic Programs

Miguel Areias and Ricardo Rocha
DCC-FC & CRACS
University of Porto, Portugal
miguel-areias@dcc.fc.up.pt *ricroc@dcc.fc.up.pt*

EPIA 2009, Aveiro, Portugal, October 2009

Tabling in Logic Programming

- **Tabling** is an implementation technique where answers for subcomputations are stored and then reused when a repeated computation appears.
  - Tabled subgoals are evaluated by storing their answers in an appropriate data space, called the **table space**.
  - Variant calls to tabled subgoals are resolved by **consuming** the answers already stored in the table instead of being re-evaluated against the program clauses.

- Tabling has proven to be particularly effective in logic (**Prolog**) programs:
  - **Avoids recomputation**, thus reducing the search space.
  - **Avoids infinite loops**, thus ensuring termination for a wider class of programs.

Motivation

- The execution model in which most tabling engines are based allocate a **choice point** whenever a new tabled subgoal is called. This happens even when the **call is deterministic** (i.e., defined by a single matching clause).

Motivation

- The execution model in which most tabling engines are based allocate a **choice point** whenever a new tabled subgoal is called. This happens even when the **call is deterministic** (i.e., defined by a single matching clause).
- Moreover, when a **deterministic answer** is found for a tabled call, the call can be **completed early** and the corresponding choice point can be removed. However, most tabling engines do not do that and only complete later.

Motivation

- The execution model in which most tabling engines are based allocate a **choice point** whenever a new tabled subgoal is called. This happens even when the **call is deterministic** (i.e., defined by a single matching clause).
- Moreover, when a **deterministic answer** is found for a tabled call, the call can be **completed early** and the corresponding choice point can be removed. However, most tabling engines do not do that and only complete later.
- In this work, we propose two different solutions to reduce this memory and execution overhead to a minimum.
  - Reduce choice point's size for deterministic tabled calls since some information is never used.
  - Complete and remove choice point when finding a deterministic answer for a tabled call.

Motivation

- The execution model in which most tabling engines are based allocate a **choice point** whenever a new tabled subgoal is called. This happens even when the **call is deterministic** (i.e., defined by a single matching clause).
- Moreover, when a **deterministic answer** is found for a tabled call, the call can be **completed early** and the corresponding choice point can be removed. However, most tabling engines do not do that and only complete later.
- In this work, we propose two different solutions to reduce this memory and execution overhead to a minimum.
  - Reduce choice point's size for deterministic tabled calls since some information is never used.
  - Complete and remove choice point when finding a deterministic answer for a tabled call.
- We will focus our discussion on a concrete implementation, the **YapTab system**, but our proposal can be generalized and applied to other tabling engines.

Basics
Compilation of Tabled Predicates in YapTab

- Tabled predicates defined by several clauses are compiled using the **table_try_me**, **table_retry_me** and **table_trust_me** WAM-like instructions in a similar manner to the generic `try_me`/`retry_me`/`trust_me` WAM sequence.
- Tabled predicates defined by a single clause are compiled using the **table_try_single** WAM-like instruction.

Basics
Compilation of Tabled Predicates in YapTab

```
:- table t/1.
t(X) :- ...

% compiled code generated by YapTab for predicate t/1
t1_1:  table_try_single t1_1a
t1_1a: 'WAM code for clause t(X) :- ...'
```

- As **t/1** is a deterministic tabled predicate, the **table_try_single** instruction will be executed for every call to this predicate.

Basics
Compilation of Tabled Predicates in YapTab

```
:- table t/3.
t(a1,b1,c1) :- ...
t(a2,b2,c2) :- ...
t(a2,b1,c3) :- ...
t(a3,b1,c2) :- ...

% compiled code generated by YapTab for predicate t/3
t3_1:  table_try_me t3_2
t3_1a: 'WAM code for clause t(a1,b1,c1) :- ...'
t3_2:  table_retry_me t3_3
t3_2a: 'WAM code for clause t(a2,b2,c2) :- ...'
t3_3:  table_retry_me t3_4
t3_3a: 'WAM code for clause t(a2,b1,c3) :- ...'
t3_4:  table_trust_me
t3_4a: 'WAM code for clause t(a3,b1,c2) :- ...'
```

Basics
Compilation of Tabled Predicates in YapTab

- **t/3** is a non-deterministic tabled predicate, but some calls to this predicate can be deterministic, i.e., defined by a single matching clause.
- For example, the calls **t(X,Y,c3)** and **t(a3,X,Y)** are deterministic as they only match with a single t/3 clause.

```
:- table t/3.
t(a1,b1,c1) :- ...
t(a2,b2,c2) :- ...
t(a2,b1,c3) :- ...
t(a3,b1,c2) :- ...
```

Basics
Compilation of Tabled Predicates in YapTab

- **t/3** is a non-deterministic tabled predicate, but some calls to this predicate can be deterministic, i.e., defined by a single matching clause.
- For example, the calls **t(X,Y,c3)** and **t(a3,X,Y)** are deterministic as they only match with a single t/3 clause.

```
:- table t/3.
t(a1,b1,c1) :- ...
t(a2,b2,c2) :- ...
t(a2,b1,c3) :- ...
t(a3,b1,c2) :- ...
```

- For this kind of deterministic calls, YapTab uses the just-in-time indexing mechanism of Yap to dynamically generate **table_try_single** instructions.

Basics
Just-In-Time Indexing in YapTab

- Tabled calls matching more than a single clause are dynamically indexed using the **table_try**, **table_retry** and **table_trust** WAM-like instructions in a similar manner to the generic `try/retry/trust` WAM sequence.

```
% indexed code generated by YapTab for call t(X,b1,Y)
table_try   t3_1a
table_retry t3_3a
table_trust t3_4a

t3_1a: 'WAM code for clause t(a1,b1,c1) :- ...'
t3_2a: 'WAM code for clause t(a2,b2,c2) :- ...'
t3_3a: 'WAM code for clause t(a2,b1,c3) :- ...'
t3_4a: 'WAM code for clause t(a3,b1,c2) :- ...'
```

Basics
Just-In-Time Indexing in YapTab

- Tabled calls matching a single clause are dynamically indexed using the **table_try_single** instruction.

```
% indexed code generated by YapTab for call t(X,Y,c3)
table_try_single t3_3a
% indexed code generated by YapTab for call t(a3,X,Y)
table_try_single t3_4a

t3_1a: 'WAM code for clause t(a1,b1,c1) :- ...'
t3_2a: 'WAM code for clause t(a2,b2,c2) :- ...'
t3_3a: 'WAM code for clause t(a2,b1,c3) :- ...'
t3_4a: 'WAM code for clause t(a3,b1,c2) :- ...'
```

Basics
Last Matching Clause

- When evaluating a tabled predicate, the **last matching clause** of a call is implemented by one of these instructions:
  - **table_try_single**: when we have a deterministic predicate or a deterministic call optimized by indexing code.
  - **table_trust_me**: when we have a generic call to the predicate (all the arguments of the call are unbound variables).
  - **table_trust**: when we have a more specific call optimized by indexing code (some of the arguments are at least partially instantiated).

Basics
Last Matching Clause

- When evaluating a tabled predicate, the **last matching clause** of a call is implemented by one of these instructions:
  - **table_try_single**: when we have a deterministic predicate or a deterministic call optimized by indexing code.
  - **table_trust_me**: when we have a generic call to the predicate (all the arguments of the call are unbound variables).
  - **table_trust**: when we have a more specific call optimized by indexing code (some of the arguments are at least partially instantiated).
- The computation state that we have when executing a **table_trust_me** or **table_trust** instruction is similar to that one of a **table_try_single** instruction, that is, in both cases the current clause can be seen as deterministic as it is the **last (or single) matching clause** for the call at hand.
- Thus, we can view the **table_trust_me** and **table_trust** instructions as a special case of the **table_try_single** instruction and use the same approach to efficiently deal with deterministic tabled calls.

Deterministic Tabled Calls
Tabled Nodes

- A tabled node is a WAM choice point extended with some extra fields:
  - The top section contains the usual WAM fields needed to restore the computation on backtracking plus two extra fields.
  - The middle section contains the **argument registers** of the call.
  - The bottom section contains the **substitution variables**, i.e., the set of free variables which exist in the terms in the argument registers of the call.

cp_b	Failure continuation CP
cp_ap	Next unexploit alternative
cp_tr	Top of trail
cp_cp	Success continuation PC
cp_h	Top of global stack
cp_env	Current Environment
cp_dep_fr	Dependency frame
cp_sg_fr	Subgoal frame

An	Argument Register n
⋮	⋮
A1	Argument Register 1

m	Number of Substitution Vars
Vm	Substitution Variable m
⋮	⋮
V1	Substitution Variable 1

Deterministic Tabled Calls
Tabled Nodes

- As the computation is never resumed in a deterministic tabled node, we can remove some fields.
  - The **cp_cp**, **cp_h**, **cp_env** and **cp_dep_fr** fields.
  - The **argument registers**.

cp_b	Failure continuation CP
cp_ap	Next unexploit alternative
cp_tr	Top of trail
cp_cp	Success continuation PC
cp_h	Top of global stack
cp_env	Current Environment
cp_dep_fr	Dependency frame
cp_sg_fr	Subgoal frame

An	Argument Register n
⋮	⋮
A1	Argument Register 1

m	Number of Substitution Vars
Vm	Substitution Variable m
⋮	⋮
V1	Substitution Variable 1

Deterministic Tabled Calls
Tabled Nodes

- The remaining fields are still required because:
  - **cp_b** is needed for failure continuation.
  - **cp_ap** and **cp_tr** are needed when backtracking to the node.
  - **cp_sg_fr** is needed by the new answer and completion operations.
  - The **substitution variables** are needed by the new answer operation.

cp_b	Failure continuation CP
cp_ap	Next unexploit alternative
cp_tr	Top of trail
cp_sg_fr	Subgoal frame

m	Number of Substitution Vars
Vm	Substitution Variable m
⋮	⋮
V1	Substitution Variable 1

Deterministic Tabled Calls
Tabled Nodes

- Considering that **A** is the number of arguments registers and that **S** is the number of substitution variables, the percentage of memory saved with the new representation can be expressed as:

$$1 - \frac{4 + 1 + S}{8 + A + 1 + S}$$

- This memory reduction increases when the number of argument registers is bigger and the number of substitution variables is smaller.

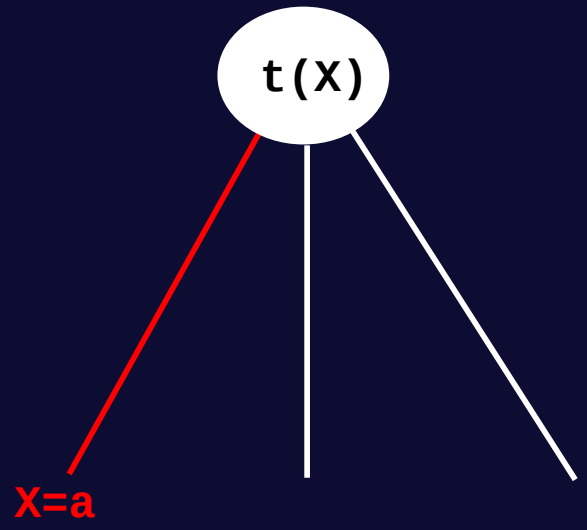
Early Completion
Last Matching Answer

- During execution it is not always possible to conclude beforehand that no more answers will be found for a particular tabled call.
- This is a safe conclusion only when the tabled call is deterministic (i.e., the clause being executed for the tabled call at hand is the last matching clause), and the choice point for the tabled call is the topmost choice point (i.e., no alternative paths exist for evaluating the tabled call at hand).
- Again, we will generalize the idea of being deterministic and we will consider that a tabled answer is deterministic when the answer is the **last matching answer** for the corresponding tabled call.

Early Completion
Last Matching Answer

```
:-table t/1.

t(X):- ...
t(X):- ...
t(X):- ...,a(X),...
```

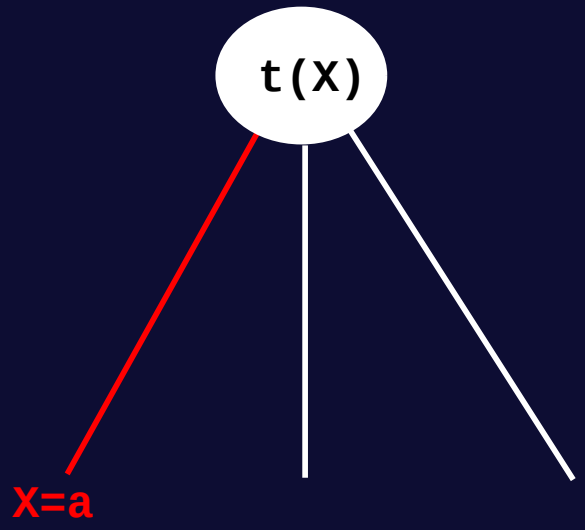


- Is **X=a** a deterministic answer (last matching answer) ?

Early Completion
Last Matching Answer

```
:-table t/1.

t(X):- ...
t(X):- ...
t(X):- ...,a(X),...
```

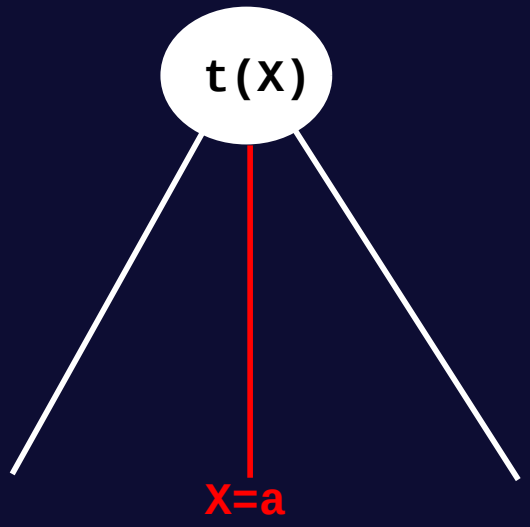


- Is **X=a** a deterministic answer (last matching answer) ?
  - **Not possible to know at this stage**. As `t(X)` (table_try clause) is not in the last matching clause, new answers can still be found in the two remaining clauses.

Early Completion
Last Matching Answer

```
:-table t/1.

t(X):- ...
t(X):- ...
t(X):- ...,a(X),...
```

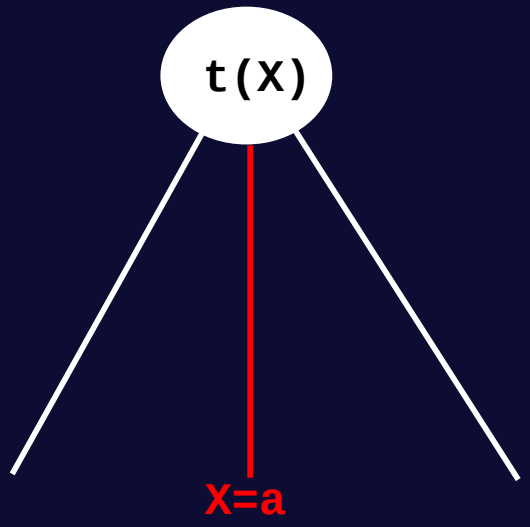


- Is **X=a** a deterministic answer (last matching answer) ?

Early Completion
Last Matching Answer

```
:-table t/1.

t(X):- ...
t(X):- ...
t(X):- ...,a(X),...
```

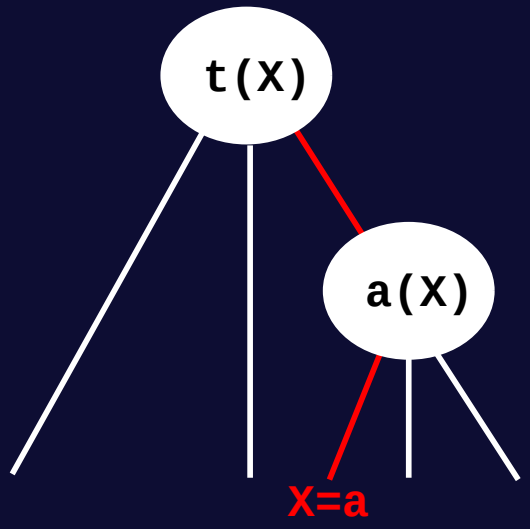


- Is **X=a** a deterministic answer (last matching answer) ?
  - **Not possible to know at this stage**. As `t(X)` (table_retry clause) is not in the last matching clause, new answers can still be found in the remaining clause.

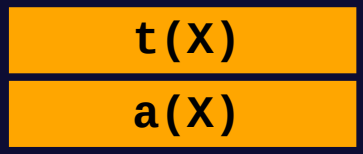
Early Completion
Last Matching Answer

```
:-table t/1.

t(X):- ...
t(X):- ...
t(X):- ...,a(X),...
```



Choice Point Stack

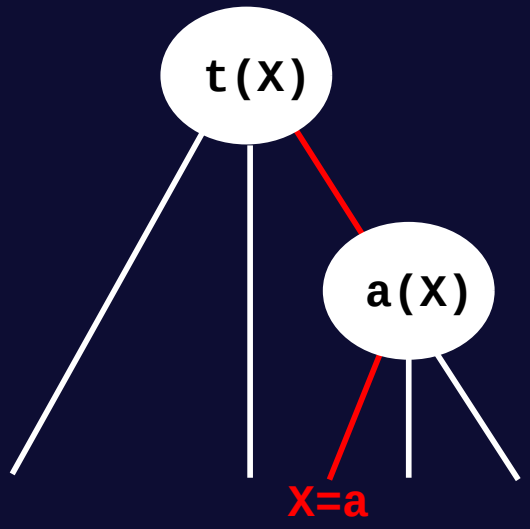


- Is **X=a** a deterministic answer (last matching answer) ?

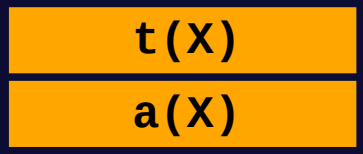
Early Completion
Last Matching Answer

```
:-table t/1.

t(X):- ...
t(X):- ...
t(X):- ...,a(X),...
```



Choice Point Stack

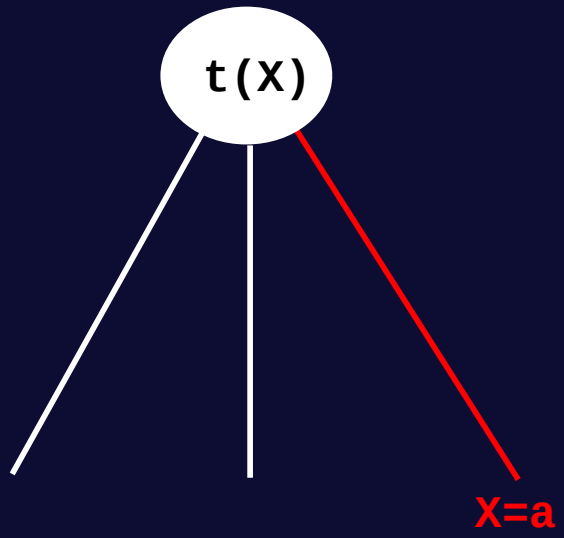


- Is **X=a** a deterministic answer (last matching answer) ?
  - **Not possible to know at this stage**. `t(X)` is not the top most choice point on stack, new answers can still be found in the remaining clauses for `a(X)`.

Early Completion
Last Matching Answer

```
:-table t/1.

t(X):- ...
t(X):- ...
t(X):- ...,a(X),...
```



Choice Point Stack

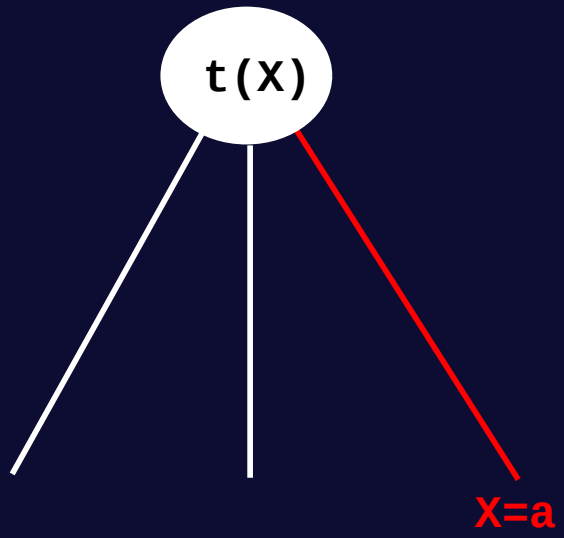
t(X)

- Is **X=a** a deterministic answer (last matching answer) ?

Early Completion
Last Matching Answer

```
:-table t/1.

t(X):- ...
t(X):- ...
t(X):- ...,a(X),...
```



Choice Point Stack

t(X)

- Is **X=a** a deterministic answer (last matching answer) ?
  - **Yes**. `t(X)` is the **top most choice point** and it is executing the last matching clause (table_trust clause). Table for `t(X)` can be completed.

Experimental Results

Args	Subs	YapTab(a) Memory	YapTab(a) Time	YapTab+Det(b) Memory	YapTab+Det(b) Time	(b)/(a) Memory	(b)/(a) Time
5	4	18,751	224	11,719	160	0.62	0.71
5	2	17,188	216	10,157	148	0.59	0.69
5	0	15,626	216	8,594	152	0.55	0.70
11	10	28,126	332	16,407	240	0.58	0.72
11	5	24,219	256	12,501	268	0.52	1.05
11	0	20,313	232	8,594	184	0.42	0.79
17	16	37,501	444	21,094	340	0.56	0.77
17	8	31,251	436	14,844	300	0.47	0.69
17	0	25,001	312	8,594	236	0.34	0.76
Average						**0.52**	**0.76**

Memory usage in KBytes and running times in milliseconds for three deterministic tabled predicates (with arities 5, 11 and 17) that call themselves recursively 100,000 times with three different sets of free variables in the arguments.

Experimental Results

Program	Size	YapTab(a) Memory	YapTab(a) Time	YapTab+Det(b) Memory	YapTab+Det(b) Time	(b)/(a) Memory	(b)/(a) Time
fib/2	1000	250	984	203	884	0.8120	0.8984
	2000	375	2,880	305	2,804	0.8133	0.9736
	4000	500	6,492	407	6,420	0.8140	0.9889
seq/3	500	45,914	792	39,079	448	0.8511	0.5657
	1000	183,625	8,108	156,282	3,272	0.8511	0.4036
	2000	734,438	135,580	718,813	117,483	0.9787	0.8665
fib_t/2	1000	250	988	125	368	0.5000	0.3725
	2000	375	3,040	188	1,268	0.5013	0.4171
	4000	500	6,516	250	2,828	0.5000	0.4340
seq_t/3	500	45,914	804	78	252	**0.0017**	0.3134
	1000	183,625	8,844	157	952	**0.0009**	0.1076
	2000	734,438	131,904	313	7,048	**0.0004**	0.0534

Memory usage in KBytes and running times in milliseconds for original and transformed versions of Sequence Comparisons and Fibonacci problems.

Conclusions

- We have presented a proposal for the efficient evaluation of deterministic tabled calls and deterministic tabled answers.
- Our preliminary results are quite promising as they suggest that, for certain applications, it is possible not only to reduce the memory usage overhead but also the running time of the evaluation.
- Further work will include exploring the impact of applying our proposal to more complex problems, seeking real-world experimental results allowing us to improve and expand our current implementation.