Tabling for Transaction Logic

Paul Fodor Michael Kifer
State University of New York at Stony Brook

Paper presentation at PPDP-2010
12th International Symposium on Principles and Practice of Declarative Programming
July 26-28, 2010, Hagenberg, Austria

Tabling for Transaction Logic

- **Transaction Logic** is a logic for representing declarative and procedural knowledge
  - Applications in logic programming, databases, AI planning, workflows, Web services, security policies, reasoning about actions, and more
- Implementations (Flora2, Toronto)
  - **Problem with existing implementations:**
    - Not logically complete due to the inherent difficulty and time/space complexity (analogous to the difference between plain Prolog and Datalog)
- **Solution:**
  - Tabling for a logically complete evaluation strategy for Transaction Logic
  - Several optimizations
  - Performance evaluation (for six different implementations)

Outline

- Overview of Transaction Logic
- Tabling Transaction Logic
  - Proof theory with tabling
  - Soundness and completeness results
- Difficulties with implementing tabling
  - State copying, comparison (time, space)
  - Solutions: logs vs. full state materialization, table skipping, various data structures (tries, B-trees)
- Experimental results
- Conclusions and future work

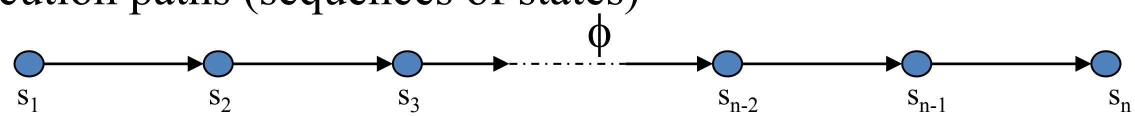
Overview of Transaction Logic

- Logic for programming state-changing actions and reasoning about their effects
- Elementary state transitions
  - **insert/1** and **delete/1** specify basic updates of the current state of the database
  - have both a truth value and a side effect on the database
- Connectives: classical logical connectives ($\wedge$, $\vee$, $\neg$) and

  additional ($\otimes$, $\oplus$, $\Diamond$, $\odot$, $|$)

  φ :- Ψ : define Ψ to be an execution of φ;

  $\varphi \otimes \Psi$ means: execute φ, then execute Ψ;
- A formula in Transaction Logic is a transaction and has truth values over execution paths (sequences of states)



States: $s_1, s_2, s_3, \ldots, s_{n-2}, s_{n-1}, s_n$

Initial state: s_1

Final state: s_n

Semantics: φ executes along π iff φ is true on π

Proof theory: executes φ along π as it proves φ

Transaction Logic Example 1

- Consuming paths (reachability in the graph by traversing edges and then swallowing them):

$$reach(X, Y) \text{ :- } reach(X,Z) \otimes edge(Z,Y) \otimes delete(edge(Z,Y)).$$

$$reach(X,X).$$

  - *edge* is a binary fluent
  - *delete(edge(N,M))* denotes the action of deleting the *edge(N,M)*

- The first rule defines the action *reach* recursively

Transaction Logic Example 2

- Hamiltonian cycle (visits each vertex exactly once, Hamiltonian cycles are detected here by swallowing the already traversed vertexes):

hCycle(Start,Start) :- *not vertex(X):*

hCycle(Start,X) :-

edge(X,Y) $\otimes$ *vertex(Y)*

$\otimes$ *delete(vertex(Y))* $\otimes$ *insert(mark(X,Y))*

$\otimes$ *hCycle(Start,Y)* $\otimes$ *insert(vertex(Y)).*

  - *edge, vertex, mark* are fluents;
  - *delete(vertex(Y))* denotes the action of deleting the fluent *vertex(Y)*
  - *insert(mark(X,Y))* denotes the action of inserting *mark(X,Y)* in the state
- The second rule does the search
  - Many possible ways to fail
  - Only few succeed
- The changes are backtrackable

Transaction Logic Example 3

- STRIPS-like planning for building pyramids of blocks (actions: *pickup*, *putdown*, **recursive** *stack*, *unstack*):

$move(X, Y) \text{ :- } X \neq Y \otimes pickup(X) \otimes putdown(X, Y).$

$pickup(X) \text{ :- } clear(X) \otimes on(X, Y) \otimes delete(on(X, Y)) \otimes insert(clear(Y)).$

$putdown(X, Y) \text{ :- } clear(Y) \otimes not\ on(X,Z1) \otimes not\ on(Z2,X) \otimes delete(clear(Y)) \otimes insert(on(X, Y)).$

$stack(0,Block).$

$stack(N,X) \text{ :- } N > 0 \otimes move(Y,X) \otimes stack(N - 1, Y) \otimes on(Y,X).$

$stack(N,X) \text{ :- } N > 0 \otimes on(Y,X) \otimes unstack(Y) \otimes stack(N,X).$

$unstack(X) \text{ :- } on(Y,X) \otimes unstack(Y) \otimes unstack(X).$

$unstack(X) \text{ :- } isclear(X) \wedge on(X, table).$

$unstack(X) \text{ :- } (isclear(X) \wedge on(X, Y) \wedge Y \neq table) \otimes move(X, table).$

$unstack(X) \text{ :- } on(Y,X) \otimes unstack(Y) \otimes unstack(X).$

A proof theory for serial-Horn transaction logic

- Serial-Horn rules

$$\alpha \text{ :- } \beta_1 \otimes \beta_2 \otimes \dots \otimes \beta_n$$

- Queries are of the form:

$$(\exists)\ \beta_1 \otimes \beta_2 \otimes \dots \otimes \beta_n$$

- Models as executions (mappings from paths to classical models)
  - Executional entailment: the truth assignments of TR transactions are evaluated over execution paths (sequences of states)

$$P, D_0, D_1, \dots, D_n \models (\exists)\ \beta_1 \otimes \beta_2 \otimes \dots \otimes \beta_n$$

- SLD-like resolution proof strategy
  - aims to prove statements of the form

$$P, D_0 \text{---}\vdash \beta_1 \otimes \beta_2 \otimes \dots \otimes \beta_n$$

- An *inference* succeeds if and only if it finds an execution for the transaction — a sequence of database states $D_1, \dots, D_n$— such that

$$P, D_0, D_1, \dots, D_n \vdash \beta_1 \otimes \beta_2 \otimes \dots \otimes \beta_n$$

A proof theory for serial-Horn transaction logic (propositional)

Axioms: $P;D\text{---} \vdash (\,)$

1. *Applying transaction definitions:*

   If a :- φ is a rule in P, then

$$\frac{P,D0\text{---}\vdash (\varphi \otimes rest)}{P,D0\text{---}\vdash (a \otimes rest)}$$

2. *Querying the database:*

   If b is a fact true in D0, then

$$\frac{P,D0\text{---}\vdash \quad rest}{P,D0\text{---}\vdash b \otimes rest}$$

3. *Performing elementary updates:*

   If b is an **elementary action** that changes state D1 to D2, then

$$\frac{P,D2\text{---}\vdash \quad rest}{P,D1\text{---}\vdash (b \otimes rest)}$$

A proof theory for serial-Horn transaction logic (predicative)

Axioms: P;D--- ⊢ ( )

1. *Applying transaction definitions:*

Let *a* :- φ is a rule in P (variables have been renamed so that the rule shares no variables with *b* ⊗ *rest*).

If *a* and *b* unify with a most general unifier σ, then

$$\frac{P,D_0 \text{ ---}\vdash (\exists)\,(\varphi \otimes \mathit{rest})\,\sigma}{P,D_0 \text{ ---}\vdash (\exists)\,(b \otimes \mathit{rest})}$$

2. *Querying the database:*

If *b* is a fluent literal, *b* and *rest* share no variables, and *b* is true in the database state D then

$$\frac{P,D_0 \text{ ---}\vdash (\exists)\,\mathit{rest}\,\sigma}{P,D_0 \text{ ---}\vdash (\exists)\,(b \otimes \mathit{rest})}$$

3. *Performing elementary updates:*

If *b* and *rest* share no variables, and *b* is an **elementary action** that changes state D_1 to state D_2 then

$$\frac{P,D_2 \text{ ---}\vdash (\exists)\,\mathit{rest}\,\sigma}{P,D_1 \text{ ---}\vdash (\exists)\,(b \otimes \mathit{rest})}$$

A proof theory for serial-Horn transaction logic

THEOREM (Soundness and Completeness [Bonner&Kifer 1995]).

If φ is a serial-Horn goal, the executional entailment

$P, D_0, D_1, \ldots, D_n \models (\exists)\ \varphi$

holds if and only if there is an executional deduction of $(\exists)\varphi$ on the path $D_0, D_1, \ldots, D_n$

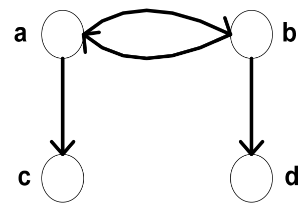
- No particular way of applying the inference rules.
- If these rules are applied in the **forward direction**, then all execution paths will be enumerated, but is undirected, exhaustive and implementational impractical
- **Backward direction (**the usual SLD resolution with left-to-right literal selection): goal-directed search, efficient, BUT **incomplete** (similarly to Prolog)

Consuming Paths Tabling Example

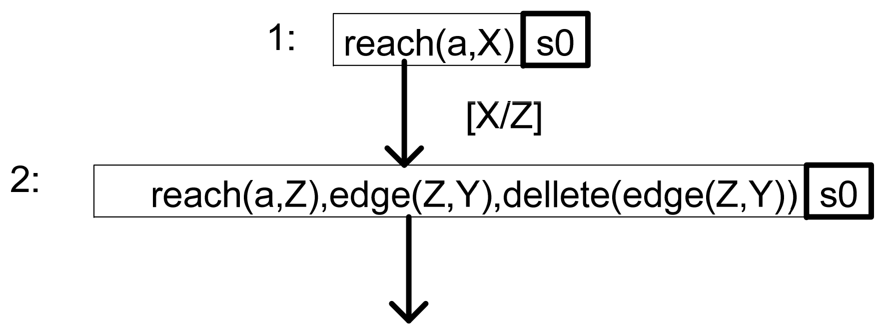
reach(X, Y) :- reach(X,Z) ⊗ *edge(Z,Y)* ⊗ *delete(edge(Z,Y)).*

reach(X,X).

Initial state:



Query: *reach(a,X)* - all *X* reachable from *a* (and return state)



infinite derivation path

Transaction Logic Tabling

- Tabling in Datalog
  - Memoize calls
  - Remember answers
- Major differences from tabling Datalog:
  - Also memoize the database states in which the calls were made
  - Remember result states created by execution of calls

Transaction Logic Tabling

- Algorithm:
  - On calling a subgoal to a tabled predicate in a state, check if this is the first occurrence of this subgoal in that state:
    - If the call is new, save (goal; state) in a global table, and continue using normal clause resolution to compute answers and the result database states for the subgoal.
      - The computed (answer; result-state) pairs are recorded in the answer table created for the (goal; state) pair
    - If the call is not new and a pair (goal; state) exists in the table, the answers to the call are returned directly from the answer table for (goal; state)
  - Side note – real algorithm: a tabled goal dominates another in tabled resolution if the two goals are variants of each other (variant tabling), or if the first goal subsumes the second (subsumptive tabling)

Modified proof theory for tabling for serial-Horn transaction logic

Modified Inference Rule 1

1a. Applying transaction definitions for tabled predicates:

If *b* is a call to a tabled predicate encountered for the first time at state D, *a* :- φ is a rule in P (variables have been renamed so that the rule shares no variables with *b* ⊗ *rest*), *a* and *b* unify with a most general unifier σ, then

$$\frac{P,D\text{---}\vdash (\exists)\,(\varphi \otimes \mathit{rest})\sigma}{P,D\text{---}\vdash (\exists)\,(b \otimes \mathit{rest})}$$

The pair (*b*,D) is added to the table space.

When the sequent P,D---D'⊢ (∃) φσ γ , for some substitution γ, is derived, the answer (φσγ,*D'*) is added to the answer table associated with the table entry (*b*,D).

1b. Returning answers from answer tables:

If *b* ⊗ *rest* is a goal call to program P at state D, *b*'s predicate symbol is declared as tabled, and a dominating pair (*c*,D) in the table space. The answer table for (*c*,D) has an entry (*a*,D') and *a* and *b* unify with mgu σ, then

$$\frac{P,D'\text{---}\vdash (\exists)\,(\mathit{rest})\,\sigma}{P,D\text{---}\vdash (\exists)\,(b \otimes \mathit{rest})}$$

1c. Applying transaction definitions for non-tabled predicates:

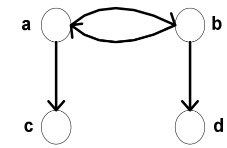
Same as rule 1 in the old theory.

Consuming Paths Tabling Example

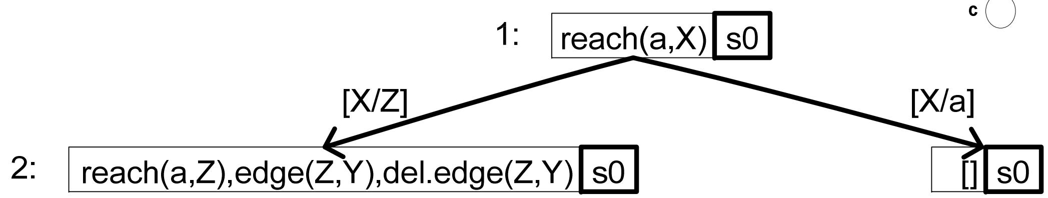
reach(X,Y) ← reach(X,Z) ⊗ edge(Z,Y) ⊗ del.edge(Z,Y).

reach(X,X).

Initial state:



Step 1,2:



States: [s0]

Solution table:

Initial state > Call	Answer-unification > Final state
reach(a,X) , s0	[reach(a,a)>s0]

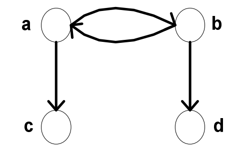
Lookup table:

Lookup node	Call	Index in Answer-unification+Final state table
2	reach(a,X)	0

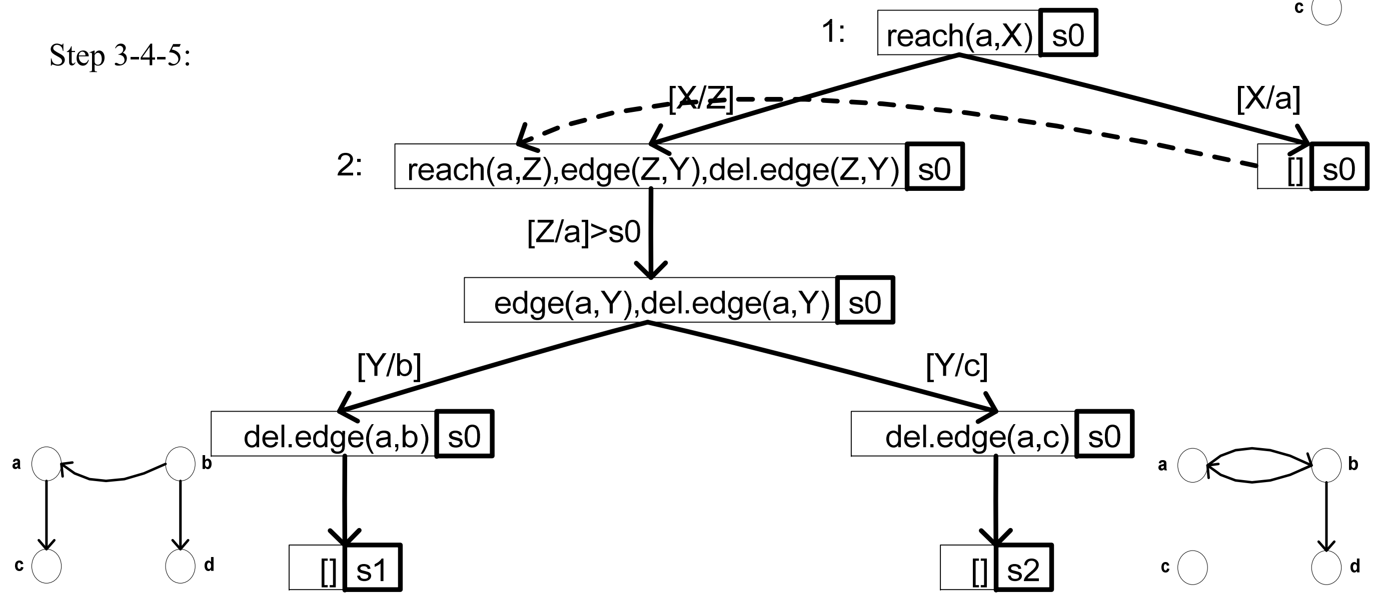
Consuming Paths Tabling Example

```
reach(X,Y) ← reach(X,Z) ⊗ edge(Z,Y) ⊗ del.edge(Z,Y).
reach(X,X).
```

Initial state:



Step 3-4-5:



States: [s0, s1={edge(a,c),edge(b,a),edge(b,d)}, s2={edge(a,b),edge(b,a),edge(b,d)}]

Solution table:

Initial state > Call	Answer-unification > Final state
reach(a,X) , s0	[reach(a,a)>s0, reach(a,b)>s1, reach(a,c)>s2]

Lookup table:

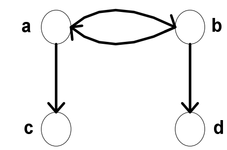
Lookup node	Call	Index in Answer-unification+Final state table
2	reach(a,X)	1

Consuming Paths Tabling Example

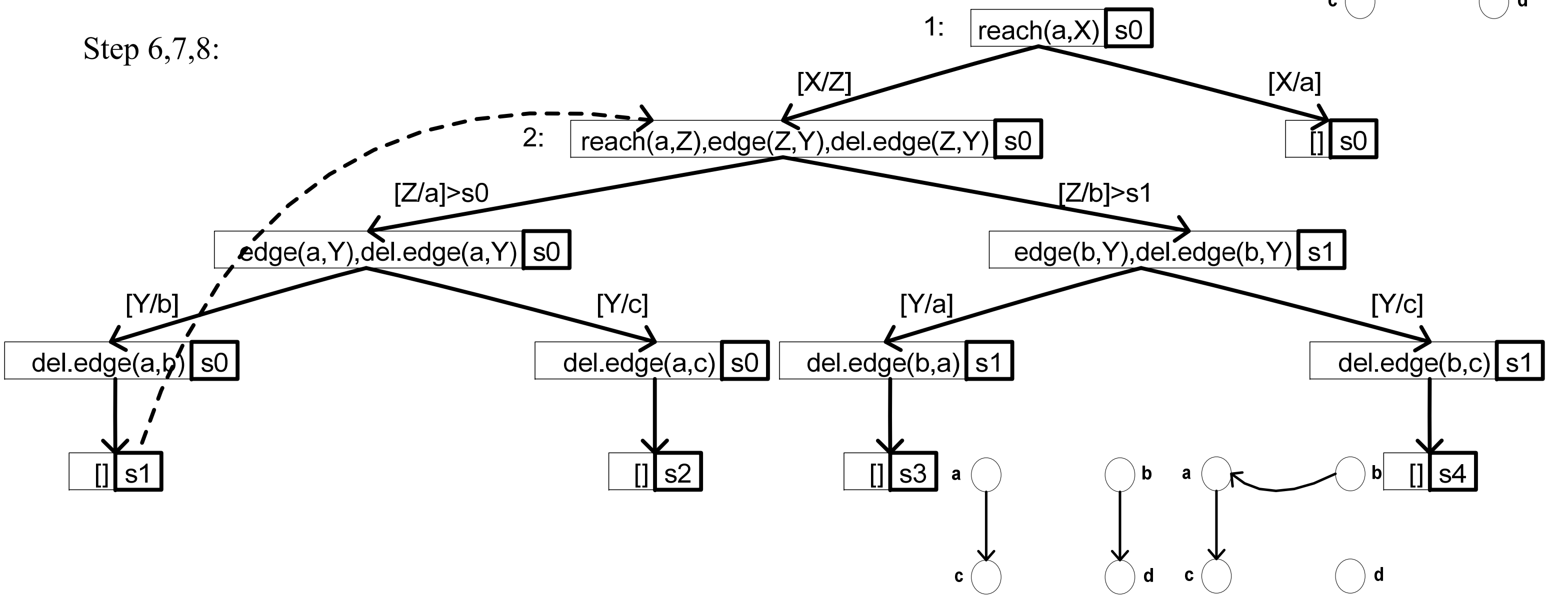
reach(X,Y) ← reach(X,Z) ⊗ edge(Z,Y) ⊗ del.edge(Z,Y).

reach(X,X).

Initial state:



Step 6,7,8:



States: [s0, s1, s2, s3={edge(a,c),edge(b,d)}, s4={edge(a,c),edge(b,a)}]

Solution table:

Initial state > Call	Answer-unification > Final state
reach(a,X) , s0	**[reach(a,a)>s0, reach(a,b)>s1, reach(a,c)>s2, reach(a,a)>s3, reach(a,d)>s4]**

Lookup table:

Lookup node	Call	Index in Answer-unification+Final state table
2	**reach(a,X)**	**2**

Tabling serial-Horn transaction logic

THEOREM (Soundness and Completeness)

The tabled proof theory is sound and complete.

Completeness in the sense that:

- it guarantees that all final states will be found
- it does not guarantee that all execution paths will be found
  - Finding all execution paths is what we wanted to avoid: there can be an infinite number of ways to reach a final state.

Tabling serial-Horn transaction logic

- The number of final states is often finite

THEOREM (Termination)

If φ is a serial-Horn goal, all proofs of P,D---⊨ (∃)φ address only a finite number of database states and a finite number of goals, the proof theory terminates.

Implementation, Problems and Solutions

- The transactional semantics of actions (easy)
- Tabling of database states:
  - **Space** - duplication of information
  - **Time** - operations:
    - *Copying of states*: once tabled the contents of that state must stay immutable
    - *Comparison of states:*
      - for tabled transactions, check whether a goal/state pair is already tabled
      - newly created states need to be compared with other tabled states to determine if it is a genuinely new state or not
    - *Querying of states*: new states created during the execution of transactions must be efficiently queryable

Space issues

- Changes (logs) << States
  - differential logs: (*InitialState*, (*InsertLog*,*DeleteLog*))
    - saves space
    - reduces the amount of time for copying states
    - trade-off between the decreasing cost of storing and copying
- Various forms of compression
  - Sharing of logs using tries: high degree of sharing
  - Factoring: facts stored on the heap and shared using pointers
  - Table skipping: only the states associated with certain tabled subgoals are stored and indexed for querying
  - Double-differential logs: when table-skipping is used, only changes relative to the previous saved state are kept, not relative to the initial state
    - *main change log*
    - *residual change log*

Time issues

- State comparison:
  - Most state comparisons fail. Determine that quickly via an incremental hash function
  - Compare the rest in linear time using tries
  - Separate state repository for the calls and states
- Data structures for querying
  - special query data structures from logs
  - trade-off of update vs. query time
- Copying of states - table skipping and factoring

Evaluation

- Common features:
  - Data compression via factoring
  - Differential logs.
  - State comparison via incremental hash functions and tries for the main differential logs (linear comparison)

= speed up by 3 orders of magnitude and use 2 orders of magnitude less memory

- **Implementation 1:** no table skipping, logs are ordered lists, updates insertion and delete sort
- **Implementation 2:** logs are ordered lists (optimal copying and sharing in tries) and query tries (to speed up querying)
- **Implementations 3a and 3b:** use *table skipping* (reduce the number of tabled states, no state copying or comparison for execution of non-tabled actions), 3a uses single differential log, 3b uses double differential logs, use *sorted lists to represent logs*
- **Implementations 4a and 4b:** use table skipping, 4a uses single differential logs, 4b uses double logs, *use tries to represent logs* (4a single differential and 4b main differential log)

Performance evaluation Consuming paths

:- *tr_table(reach/2).*

reach(X, Y) :- reach(X,Z) ⊗ edge(Z,Y) ⊗ delete(edge(Z,Y)).

reach(X,X).

Graph size	100			250			350		
# of Solutions	5050			31375			61425		
	CPU	Tabled states	State comp.	CPU	Tabled states	State comp.	CPU	Tabled states	State comp.
1	0.128	5051	5050	1.544	31376	31375	3.940	61426	61425
2	0.212	5051	5050	2.292	31376	31375	5.996	61426	61425
3a	0.136	5051	5050	1.540	31376	31375	3.924	61426	61425
3b	0.152	5051	5050	1.672	31376	31375	4.608	61426	61425
4a	0.224	5051	5050	2.796	31376	31375	7.880	61426	61425
4b	0.204	5051	5050	2.128	31376	31375	5.680	61426	61425

4b (table skipping, double differential, tries) slower for small problems when few updates between tabled calls

Performance evaluation
10 Consuming paths in 10 graphs

:- tr_table(reach/2).

reach(X, Y) :- reach(X,Z)

⊗ *edge1(Z,Y)* ⊗ *delete(edge1(Z,Y))*

...

⊗ *edge10(Z,Y)* ⊗ *delete(edge10(Z,Y)).*

reach(X,X).

Graph size	100			250			350		
	CPU	Tabled states	State comp.	CPU	Tabled states	State comp.	CPU	Tabled states	State comp.
1	6.236	50501	50500	47.642	201001	201000	92.425	313751	313750
2	8.568	50501	50500	ErrMem	ErrMem	ErrMem	ErrMem	ErrMem	ErrMem
3a	4.796	5051	5050	37.182	20101	20100	71.840	31376	31375
3b	4.024	5051	5050	30.073	20101	20100	58.083	31376	31375
4a	1.780	5051	5050	13.536	20101	20100	25.929	31376	31375
4b	1.292	5051	5050	8.564	20101	20100	16.325	31376	31375

table skipping, tries for storing logs, double differential help
4b wins for multiple updates between tabled calls

Performance evaluation Hamiltonian Cycles

hCycle(Start,Start) :- not vertex(X). *:- tr_table(hCycle/2).*

hCycle(Start,X) :- edge(X,Y) ⊗ vertex(Y)

⊗ delete(vertex(Y)) ⊗ insert(mark(X,Y))

⊗ hCycle(Start,Y) ⊗ insert(vertex(Y)).

Graph size	50			150		
# of Solutions	50			150		
	CPU	Tabled states	State comp.	CPU	Tabled states	State comp.
1	0.252	7403	7500	8.392	67203	67500
2	0.244	7403	7500	4.144	67203	67500
3a	0.164	4903	5051	3.956	44703	45151
3b	0.236	4903	5000	5.644	44703	45000
4a	0.300	4903	5001	6.852	44703	45151
4b	0.300	4903	5000	5.696	44703	45000

separate query data structures for efficient queries and updates

double differential (3b, 4b) reduces number of comparisons

Performance evaluation
10 graphs Hamiltonian Cycles

hCycle(Start,Start) :- not vertex1(X):

hCycle(Start,X) :- edge1(X,Y)⊗...⊗edge10(X,Y)⊗vertex1(Y)⊗...⊗ vertex10(Y)

⊗ delete(vertex1(Y)) ⊗ insert(mark1(X,Y)) ⊗ ...

⊗ hCycle(Start,Y) ⊗ insert(vertex1(Y)) ⊗ ... ⊗ insert(vertex1(Y)) .

Graph size	50	150
1	4.912	ErrMem
2	6.052	ErrMem
3a	3.076	86.505
3b	4.340	105.814
4a	1.656	ErrMem
4b	1.356	27.421

4b wins

Performance evaluation
Blocks World

move(X, Y) :- X ≠Y ⊗pickup(X) ⊗putdown(X, Y).
pickup(X) :- clear(X) ⊗on(X, Y) ⊗delete(on(X, Y)) ⊗insert(clear(Y)).
putdown(X, Y) :- clear(Y) ⊗not on(X,Z1) ⊗not on(Z2,X) ⊗delete(clear(Y)) ⊗insert(on(X, Y)).
stack(0,Block). *:- table(stack/2).*
stack(N,X) :- N > 0 ⊗move(Y,X) ⊗stack(N - 1, Y) ⊗on(Y,X).
stack(N,X) :- N > 0 ⊗on(Y,X) ⊗unstack(Y) ⊗stack(N,X).
unstack(X) :- on(Y,X) ⊗unstack(Y) ⊗unstack(X). *:- table(unstack/1).*
unstack(X) :- isclear(X) ∧ on(X, table).
unstack(X) :- (isclear(X) ∧ on(X, Y) ∧ Y ≠ table) ⊗move(X, table).
unstack(X) :- on(Y,X) ⊗unstack(Y) ⊗unstack(X).

For every final pyramid one way to build it

Blocks	5			6			7		
# of Pyramids	120			720			5050		
	CPU	Tabled states	State comp.	CPU	Tabled states	State comp.	CPU	Tabled states	State comp.
1	0.212	1546	4210	2.392	13327	42792	29.265	130922	480326
2	0.196	1546	4210	2.100	13327	42792	26.265	130922	480326
3a	0.196	501	9767	2.192	4051	107882	27.905	37633	1364911
3b	0.228	501	1300	2.528	4051	13020	31.661	37633	144354
4a	0.228	501	9767	3.268	4051	107882	41.958	37633	1364911
4b	0.204	501	1300	2.268	4051	13020	28.117	37633	144354

Performance evaluation Pyramids in 10 Worlds

Blocks	5	6	7
1	1.800	21.457	286.413
2	1.780	19.441	Err Mem
3a	1.140	13.208	172.838
3b	1.808	21.433	287.413
4a	1.312	15.588	Err Mem
4b	1.096	11.984	148.109

4b wins
better data structures (B+-trees) help more

Summary and Future Work

- Adapted the tabling from ordinary logic programs to Transaction Logic
  - The *tabled* proof theory of Transaction Logic is sound and complete
  - Difficult to implement tabling for Transaction Logic
  - Proposed optimizations for both time and space
- Interpreter in XSB Prolog

  http://flora.sourceforge.net/tr-interpreter-suite.tar.gz
  - different optimizations, comparison
- Future Work:
  - Extend tabling to Concurrent Transaction Logic (interleaved actions)
  - Better data structure for storing states using B+ trees (efficient copying and sharing)

Thank you!

Questions?

Disclaimer: The preceding slides represent the views of the authors only.