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 ($\land$, $\lor$, $\neg$) and additional ($\otimes$, $\oplus$, $\diamond$, $\ominus$, $|$)

  - $\phi : - \Psi$: define $\Psi$ to be an execution of $\phi$;
  - $\phi \otimes \Psi$: means: execute $\phi$, then execute $\Psi$;

- 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: $\phi$ executes along $\pi$ iff $\phi$ is true on $\pi$

Proof theory: executes $\phi$ along $\pi$ as it proves $\phi$
Transaction Logic Example 1

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

  \[
  \text{reach}(X, Y) \;::=\; \text{reach}(X, Z) \otimes \text{edge}(Z, Y) \otimes \text{delete}(\text{edge}(Z, Y)).
  \]
  \[
  \text{reach}(X, X).
  \]

  - \textit{edge} is a binary fluent
  - \textit{delete(edge}(N,M)) denotes the action of deleting the edge\((N,M)\)

- The first rule defines the action \textit{reach} recursively
Transaction Logic Example 2

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

\[
\text{hCycle}(\text{Start}, \text{Start}) :- \text{not vertex}(X):
\]

\[
\text{hCycle}(\text{Start}, X) :-
\]

\[
\begin{align*}
\text{edge}(X, Y) & \otimes \text{vertex}(Y) \\
\otimes \text{delete(vertex}(Y)) & \otimes \text{insert(mark}(X, Y)) \\
\otimes \text{hCycle}(\text{Start}, Y) & \otimes \text{insert(vertex}(Y)).
\end{align*}
\]

- \text{edge}, \text{vertex}, \text{mark} are fluents;
- \text{delete(vertex}(Y)) denotes the action of deleting the fluent \text{vertex}(Y)
- \text{insert(mark}(X, Y)) denotes the action of inserting \text{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*):

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

\[
\text{pickup}(X) :- \text{clear}(X) \otimes \text{on}(X, Y) \otimes \text{delete} \left( \text{on}(X, Y) \right) \otimes \text{insert} \left( \text{clear}(Y) \right).
\]

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

\[
\text{stack}(0, \text{Block}).
\]

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

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

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

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

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

\[
\text{unstack}(X) :- \text{on}(Y, X) \otimes \text{unstack}(Y) \otimes \text{unstack}(X).
\]
A proof theory for serial-Horn transaction logic

- Serial-Horn rules

\[ \alpha \leftarrow \beta_1 \otimes \beta_2 \otimes ... \otimes \beta_n \]

- Queries are of the form:

\[ \left( \exists \right) \beta_1 \otimes \beta_2 \otimes ... \otimes \beta_n \]

- 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)

\[ P, D_0, D_1, ..., D_n \models (\exists) \beta_1 \otimes \beta_2 \otimes ... \otimes \beta_n \]

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

\[ P, D_0 \leftarrow \beta_1 \otimes \beta_2 \otimes ... \otimes \beta_n \]

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

\[ P, D_0, D_1, ..., D_n \leftarrow \beta_1 \otimes \beta_2 \otimes ... \otimes \beta_n \]
A proof theory for serial-Horn transaction logic (propositional)

Axioms: \( P;D --- \vdash () \)

1. Applying transaction definitions:
   If \( a :- \varphi \) is a rule in \( P \), then
   \( P,D0 --- \vdash (\varphi \otimes \text{rest}) \)

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

2. Querying the database:
   If \( b \) is a fact true in \( D0 \), then
   \( P,D0 --- \vdash \text{rest} \)

   \( P,D0 --- \vdash b \otimes \text{rest} \)

3. Performing elementary updates:
   If \( b \) is an elementary action that changes state \( D1 \) to \( D2 \), then
   \( P,D2 --- \vdash \text{rest} \)

   \( P,D1 --- \vdash (b \otimes \text{rest}) \)
A proof theory for serial-Horn transaction logic (predicative)

Axioms: $P;D\vdash ()$

1. Applying transaction definitions:
   Let $a :\varphi$ is a rule in $P$ (variables have been renamed so that the rule shares no variables with $b \otimes \text{rest}$).
   If $a$ and $b$ unify with a most general unifier $\sigma$, then
   
   $P,D_0\vdash (\exists (\varphi \otimes \text{rest}) \sigma$
   
   $P,D_0\vdash (\exists (b \otimes \text{rest})$

2. Querying the database:
   If $b$ is a fluent literal, $b$ and $\text{rest}$ share no variables, and $b$ is true in the database state $D$ then
   
   $P,D_0\vdash (\exists \text{rest} \sigma$
   
   $P,D_0\vdash (\exists (b \otimes \text{rest})$

3. Performing elementary updates:
   If $b$ and $\text{rest}$ share no variables, and $b$ is an elementary action that changes state $D_1$ to state $D_2$ then
   
   $P,D_2\vdash (\exists \text{rest} \sigma$
   
   $P,D_1\vdash (\exists (b \otimes \text{rest})$
THEOREM  (Soundness and Completeness [Bonner&Kifer 1995]).
If \( \phi \) is a serial-Horn goal, the executional entailment
\[
P, D_0, D_1, ..., D_n \vdash (\exists) \phi
\]
holds if and only if there is an executional deduction of \( (\exists)\phi \) on the path
\( D_0, D_1, ..., D_n \)

- 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

\[
reach(X, Y) :- \ reach(X,Z) \otimes \ edge(Z,Y) \otimes \ delete(edge(Z,Y)).
\]
\[
reach(X,X).
\]

Initial state:

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

1: \(\text{reach}(a,X)\)

2: \(\text{reach}(a,Z),\text{edge(Z,Y)},\text{delete(edge(Z,Y))}\)

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 $D$, $a :\varphi$ is a rule in $P$ (variables have been renamed so that the rule shares no variables with $b \otimes \text{rest}$), $a$ and $b$ unify with a most general unifier $\sigma$, then

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

$$P,D \vdash (\exists (b \otimes \text{rest}))$$

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

When the sequent $P,D \vdash (\exists) \varphi\sigma\gamma$, for some substitution $\gamma$, is derived, the answer $(\varphi\sigma\gamma,D')$ is added to the answer table associated with the table entry $(b,D)$.

1b. Returning answers from answer tables:

If $b \otimes \text{rest}$ 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 $\sigma$, then

$$P,D' \vdash (\exists (\text{rest})\sigma)$$

$$P,D \vdash (\exists (b \otimes \text{rest}))$$

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) \land edge(Z, Y) \land del.edge(Z, Y).
reach(X, X).

Initial state:

Step 1,2:

1: reach(a, X) s0

2: reach(a, Z), edge(Z, Y), del.edge(Z, Y) s0

States: [s0]
Solution table:

<table>
<thead>
<tr>
<th>Initial state &gt; Call</th>
<th>Answer-unification &gt; Final state</th>
</tr>
</thead>
<tbody>
<tr>
<td>reach(a, X), s0</td>
<td>[reach(a, a) &gt; s0]</td>
</tr>
</tbody>
</table>

Lookup table:

<table>
<thead>
<tr>
<th>Lookup node</th>
<th>Call</th>
<th>Index in Answer-unification+Final state table</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>reach(a, X)</td>
<td>0</td>
</tr>
</tbody>
</table>
Consomning 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: \[s_0, s_1=\{\text{edge(a,c),edge(b,a),edge(b,d)}\}, s_2=\{\text{edge(a,b),edge(b,a),edge(b,d)}\}\]

Solution table:

<table>
<thead>
<tr>
<th>Initial state &gt; Call</th>
<th>Answer-unification &gt; Final state</th>
</tr>
</thead>
<tbody>
<tr>
<td>reach(a,X), s0</td>
<td>[reach(a,a)&gt;s0, reach(a,b)&gt;s1, reach(a,c)&gt;s2]</td>
</tr>
</tbody>
</table>

Lookup table:

<table>
<thead>
<tr>
<th>Lookup node</th>
<th>Call</th>
<th>Index in Answer-unification+Final state table</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>reach(a,X)</td>
<td>1</td>
</tr>
</tbody>
</table>
**Consuming Paths Tabling Example**

reach(X, Y) ← reach(X, Z) \& edge(Z, Y) \& del.edge(Z, Y).
reach(X, X).

### Initial state:

```
reach(a, X) \& s0
reach(a, Z) \& edge(Z, Y) \& del.edge(Z, Y) \& s0
reach(a, b) \& s1
reach(a, c) \& s2
reach(a, a) \& s3
reach(a, d) \& s4
```

### Step 6, 7, 8:

```
reach(a, X) \& s0
reach(a, Z) \& edge(Z, Y) \& del.edge(Z, Y) \& s0
reach(a, b) \& s1
reach(a, c) \& s2
reach(a, a) \& s3
reach(a, d) \& s4
```

### States:

\{s0, s1, s2, s3=\{edge(a, c), edge(b, d)\}, s4=\{edge(a, c), edge(b, a)\}\}

### Solution table:

<table>
<thead>
<tr>
<th>Initial state</th>
<th>Call</th>
<th>Answer-unification &gt; Final state</th>
</tr>
</thead>
<tbody>
<tr>
<td>reach(a, X), s0</td>
<td></td>
<td>[reach(a, a)&gt;s0, reach(a, b)&gt;s1, reach(a, c)&gt;s2, reach(a, a)&gt;s3, reach(a, d)&gt;s4 ]</td>
</tr>
</tbody>
</table>

### Lookup table:

<table>
<thead>
<tr>
<th>Lookup node</th>
<th>Call</th>
<th>Index in Answer-unification+Final state table</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>reach(a, X)</td>
<td>2</td>
</tr>
</tbody>
</table>


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.
The number of final states is often finite

THEOREM (Termination)
If $\varphi$ is a serial-Horn goal, all proofs of $P,D\dashv\vdash (\exists)\varphi$ 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)
- Tableing 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: \((\text{InitialState}, (\text{InsertLog}, \text{DeleteLog}))\)
    - 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
    - \textit{main change log}
    - \textit{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**

\[ :- \text{tr_table}(\text{reach}/2). \]

\[ \text{reach}(X, Y) :- \text{reach}(X,Z) \land \text{edge}(Z,Y) \land \text{delete}(\text{edge}(Z,Y)). \]

\[ \text{reach}(X,X). \]

<table>
<thead>
<tr>
<th>Graph size</th>
<th>100</th>
<th>250</th>
<th>350</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.128</td>
<td>1.544</td>
<td>3.940</td>
</tr>
<tr>
<td>2</td>
<td>0.212</td>
<td>2.292</td>
<td>5.996</td>
</tr>
<tr>
<td>3a</td>
<td>0.136</td>
<td>1.540</td>
<td>3.924</td>
</tr>
<tr>
<td>3b</td>
<td>0.152</td>
<td>1.672</td>
<td>4.608</td>
</tr>
<tr>
<td>4a</td>
<td>0.224</td>
<td>2.796</td>
<td>7.880</td>
</tr>
<tr>
<td>4b</td>
<td>0.204</td>
<td>2.128</td>
<td>5.680</td>
</tr>
</tbody>
</table>

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) \land delete(edge1(Z, Y))
...
edge10(Z, Y) \land delete(edge10(Z, Y)).

reach(X, X).

<table>
<thead>
<tr>
<th>Graph size</th>
<th>100</th>
<th>250</th>
<th>350</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
<td>Tabled states</td>
<td>State comp.</td>
</tr>
<tr>
<td>1</td>
<td>6.236</td>
<td>50501</td>
<td>50500</td>
</tr>
<tr>
<td>2</td>
<td>8.568</td>
<td>50501</td>
<td>50500</td>
</tr>
<tr>
<td>3a</td>
<td>4.796</td>
<td>5051</td>
<td>5050</td>
</tr>
<tr>
<td>3b</td>
<td>4.024</td>
<td>5051</td>
<td>5050</td>
</tr>
<tr>
<td>4a</td>
<td>1.780</td>
<td>5051</td>
<td>5050</td>
</tr>
<tr>
<td>4b</td>
<td>1.292</td>
<td>5051</td>
<td>5050</td>
</tr>
</tbody>
</table>

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

\[ hCycle(\text{Start}, \text{Start}) :\text{ not vertex}(X). \quad :- \text{tr_table}(hCycle/2). \]

\[ hCycle(\text{Start}, X) :\text{ edge}(X,Y) \otimes \text{vertex}(Y) \]
\[ \otimes \text{delete}(\text{vertex}(Y)) \otimes \text{insert}(\text{mark}(X,Y)) \]
\[ \otimes hCycle(\text{Start}, Y) \otimes \text{insert}(\text{vertex}(Y)). \]

<table>
<thead>
<tr>
<th>Graph size</th>
<th>50</th>
<th>150</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Solutions</td>
<td>50</td>
<td>150</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>Tabled states</th>
<th>State comp.</th>
<th>CPU</th>
<th>Tabled states</th>
<th>State comp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.252</td>
<td>7403</td>
<td>7500</td>
<td>8.392</td>
<td>67203</td>
<td>67500</td>
</tr>
<tr>
<td>2</td>
<td>0.244</td>
<td>7403</td>
<td>7500</td>
<td>4.144</td>
<td>67203</td>
<td>67500</td>
</tr>
<tr>
<td>3a</td>
<td>0.164</td>
<td>4903</td>
<td>5051</td>
<td>3.956</td>
<td>44703</td>
<td>45151</td>
</tr>
<tr>
<td>3b</td>
<td>0.236</td>
<td>4903</td>
<td>5000</td>
<td>5.644</td>
<td>44703</td>
<td>45000</td>
</tr>
<tr>
<td>4a</td>
<td>0.300</td>
<td>4903</td>
<td>5001</td>
<td>6.852</td>
<td>44703</td>
<td>45151</td>
</tr>
<tr>
<td>4b</td>
<td>0.300</td>
<td>4903</td>
<td>5000</td>
<td>5.696</td>
<td>44703</td>
<td>45000</td>
</tr>
</tbody>
</table>

separate query data structures for efficient queries and updates
double differential (3b, 4b) reduces number of comparisons
Performance evaluation
10 graphs Hamiltonian Cycles

\[ hCycle(\text{Start,Start}) :\text{ not vertex1}(X): \]
\[ hCycle(\text{Start,X}) :\text{ edge1}(X,Y) \ldots \text{edge10}(X,Y) \text{vertex1}(Y) \ldots \text{vertex10}(Y) \]
\[ \text{ delete(vertex1}(Y)) \text{ insert(mrk1}(X,Y)) \ldots \]
\[ \text{ delete(vertex1}(Y)) \text{ insert(vertex1}(Y)) \ldots \text{ insert(vertex1}(Y)) . \]

<table>
<thead>
<tr>
<th></th>
<th>50</th>
<th>150</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.912</td>
<td>1.656</td>
</tr>
<tr>
<td>2</td>
<td>6.052</td>
<td>1.356</td>
</tr>
<tr>
<td>3a</td>
<td>3.076</td>
<td>86.505</td>
</tr>
<tr>
<td>3b</td>
<td>4.340</td>
<td>105.814</td>
</tr>
<tr>
<td>4a</td>
<td>1.656</td>
<td>1.356</td>
</tr>
<tr>
<td>4b</td>
<td>340</td>
<td>27.421</td>
</tr>
</tbody>
</table>

4b wins
### Performance evaluation

#### Blocks World

**move**(X, Y) :- X ≠ Y \(\otimes\) pickup(X) \(\otimes\) putdown(X, Y).

**pickup**(X) :- clear(X) \(\otimes\) on(X, Y) \(\otimes\) delete(on(X, Y)) \(\otimes\) insert(clear(Y)).

**putdown**(X, Y) :- 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) :- N > 0 \(\otimes\) move(Y, X) \(\otimes\) stack(N - 1, Y) \(\otimes\) on(Y, X).

**unstack**(X) :- on(Y, X) \(\otimes\) unstack(Y) \(\otimes\) unstack(X).

**unstack**(X) :- isclear(X) \(\land\) on(X, table).

**unstack**(X) :- (isclear(X) \(\land\) on(X, Y) \(\land\) Y ≠ table) \(\otimes\) move(X, table).

For every final pyramid one way to build it

---

<table>
<thead>
<tr>
<th>Blocks</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Pyramids</td>
<td>120</td>
<td>720</td>
<td>5050</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td><strong>Tabled states</strong></td>
<td><strong>State comp.</strong></td>
<td><strong>CPU</strong></td>
</tr>
<tr>
<td>1</td>
<td>0.212</td>
<td>1546</td>
<td>4210</td>
</tr>
<tr>
<td>2</td>
<td>0.196</td>
<td>1546</td>
<td>4210</td>
</tr>
<tr>
<td>3a</td>
<td>0.196</td>
<td>501</td>
<td>9767</td>
</tr>
<tr>
<td>3b</td>
<td>0.228</td>
<td>501</td>
<td>1300</td>
</tr>
<tr>
<td>4a</td>
<td>0.228</td>
<td>501</td>
<td>9767</td>
</tr>
<tr>
<td>4b</td>
<td>0.204</td>
<td>501</td>
<td>1300</td>
</tr>
</tbody>
</table>
Performance evaluation
Pyramids in 10 Worlds

<table>
<thead>
<tr>
<th>Blocks</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.800</td>
<td>21.457</td>
<td>286.413</td>
</tr>
<tr>
<td>2</td>
<td>1.780</td>
<td>19.441</td>
<td>Err Mem</td>
</tr>
<tr>
<td>3a</td>
<td>1.140</td>
<td>13.208</td>
<td>172.838</td>
</tr>
<tr>
<td>3b</td>
<td>1.808</td>
<td>21.433</td>
<td>287.413</td>
</tr>
<tr>
<td>4a</td>
<td>1.312</td>
<td>15.588</td>
<td>Err Mem</td>
</tr>
<tr>
<td>4b</td>
<td>1.096</td>
<td>11.984</td>
<td>148.109</td>
</tr>
</tbody>
</table>

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
  - 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.
All brands, logos and products are trademarks or registered trademarks of their respective companies.