Multi-Threaded Tabled Logic Programing in XSB

Terrance Swift
CENTRIA

Joint Work with Rui Marques and José Cunha
CITI

Summary of CICLOPS 08 paper, plus *Concurrent and Local Evaluation of Normal Program* (ICLP 08).

Overview

- A review of common features in multi-threaded Prologs
- A review of tabling features in XSB
  - Scheduling, variance vs subsumption, negation, constraints, space reclamation
- Multi-threaded tabling features in XSB
- Algorithms for shared multi-threaded tables

Multi-threaded Prolog

- Several Prologs support multi-threading including Ciao, SWI, YAP, XSB, and others.
- In XSB, the Prolog-level support is fairly typical and consists of:
  - Thread-shared static code
  - Private execution stacks, *including* Prolog's heap
    - Note difference from Java
  - Shared atom tables, streams, sockets, etc.
    - Note difference from Erlang
  - Dynamic code that can be thread-private or thread-shared
  - One "main" thread and zero or more created threads
    - SWI allows console thread to be switched
  - An API for thread creation, synchronization and communication

ISO Multi-threading API
Thread Operations

YAP, SWI and XSB support a core API that is now in an ISO draft. Its semantics is based on pthreads, although Prolog threads are *much* more heavy-weight than pthreads.

- `thread_create(#Goal,-ThreadId,+Options)` creates a thread to execute `Goal`. Aliases, exit handlers, etc can be associated with the thread.
  - SWI allows a time-out value for threads (in XSB... soon)
- `thread_join(+ThreadIds,-ExitTerms)` waits for a set of threads to exit, and unifies `ExitTerms` with a list of terms indicating whether the goal succeeded, failed, explicitly exited or threw an error term
- Cancelling threads is easier for the user in Prolog than in C. Cancellation is implemented as a virtual machine interrupt if the thread is running, or as a signal handler on blocking operations.
  - XSB currently allows cancellation during message queue operations and sleep
- One Prolog thread can signal another by giving it a signal goal to execute. The signalled thread suspends its current goal, executes the signal goal, and then returns to the goal it has been executing (unless the signal goal exits).
- The state of each running or joinable thread can be obtained via `thread_property/2`. (No guarantee of consistency.)

ISO Multi-threading API Communication and Synchronization

- Mutexes can be dynamically created or destroyed
  - Mutexes can be explicitly locked, unlocked, and tried.
  - The safest use of mutexes is `with_mutex(+Mutex,?Goal)`. This predicate locks `Mutex`, executes `Goal` deterministically; `Mutex` is unlocked when `Goal` succeeds, exits, or fails.
  - A thread is guaranteed to release its mutexes when it exits (even if cancelled or if it throws an error)
  - The state of each mutex can be obtained via `mutex_property/2`
- Message queues can also be dynamically created or destroyed
  - Message queues contain non-ground Prolog terms; readers can query a message queue for unification
  - Public message qeueus can have multiple writers and readers. Each thread also has its own private message queue that only it can read from (and its own private signal queue).
  - The state of each message queue can be obtained via `message_queue_property/2`

Multi-threaded Prologs: Summary

- ISO draft is just a start
  - Needs to incorporate certain language constructs of Ciao, Erlang and other systems
  - Needs to consider more fully the ramifications of multi-threading on existing Prolog constructs
    - E.g. what should `abolish/1` mean in an MT context? What sort of consistency should be supported for a given threads computation?
    - Should MT-Prolog behave like C, where anything goes? Or like Oracle, where data (and other) consistency is ensured, regardless of the number of users?
    - Logtalk was an early adapter of MT-Prolog, and has uncovered numerous bugs in all MT-Prologs; Logtalk also pioneered programming of simple and- and or- parallelism using multi-threading
- MT-TLP is even newer. It has received less use and less feedback than MT-Prolog (which is also new).

Tabling: Definite Programs

- Tabling can affect the termination and complexity of a program, as well as making programs more "declarative".
- Tabling is supported at some level by XSB, YAP, B-Prolog, ALS, Mercury, Ciao, and other Prologs. All of these support at least definite programs.
- Algorithms differ in how environments are suspended and resumed (e.g. shared environments (SLG-WAM: XSB, YAP), copied environment (Call Contiuation: Ciao; CAT: Mercury), recomputed environments (ALS, B-Prolog).
- Table storage mechanisms may differ between systems. Many are based on tries (XSB, YAP, Ciao and others)
- Systems implement batched scheduling, local scheduling or other scheduling mechanisms.
- In XSB, dynamic clauses for a predicate can be tabled just as static code

Tabling

For most types of tabling there will be an example for analyzing Petri-like nets. Reachability is illustrated, but path derivation could also be used.

- Reachability is a central problem for Petri Net analysis, to which problems such as liveness, deadlock-freedom, and the existence of home states can be reduced.
- The following slide illustrates the *entire* program for 1-safe (elementary) Petri Nets as a tabled definite program
  - Configuration of a Net is maintained as a set of terms (terms differ for elementary nets, place-transition nets, color nets, preference nets, etc.)
- A definite program for analyzing workflow nets is approximately twice the size. Te program supports nearly all standard workflow control patterns in (van der Aalst, ter Hofstede, 2003). This includes splits, synchronized merges, discriminators, cancellation, etc.
- Nets thus provide a good example of how TLP can be declarative. Code can be found in CVS `mttests` module of XSB on `sourceforge.net`.

Tabling: Definite Programs

```
% Prolog representation of the Producer-Consumer Net
:- index(trans/2,trie).
trans([p1],[p2],t1).          trans([b2,p2],[p1,b1],t2).
trans([b1,c1],[b2,c2],t3).    trans([c2],[c1],t4).

% Program to determine reachability of an elementary net
:- table reachable/2.
reachable(InConf,NewConf):-
   reachable(InConf,Conf),
   hasTransition(Conf,NewConf).
reachable(InConf,NewConf):-
   hasTransition(InConf,NewConf).

hasTransition(Conf,NewConf):-
   get_trans_for_conf(Conf,AllTrans),
   member(Trans,AllTrans),
   apply_trans_to_conf(Trans,Conf,NewConf).

get_trans_for_conf(Conf,Flattrans):-
   get_trans_for_conf_1(Conf,Conf,Trans),
   flatten(Trans,Flattrans).

get_trans_for_conf_1([],_Conf,[]).
get_trans_for_conf_1([H|T],Conf,[Trans1|RT]):-
   findall(trans([H|In],Out,Tran),trans([H|In],Out,Tran),Trans),
   check_concession(Trans,Conf,Trans1),
   get_trans_for_conf_1(T,Conf,RT).

check_concession([],_,[]).
check_concession([trans(In,Out,Name)|T],Input,[trans(In,Out,Name)|T1]):-
   ord_subset(In,Input),
   ord_disjoint(Out,Input),!,
   check_concession(T,Input,T1).
check_concession([_Trans|T],Input,T1):-
   check_concession(T,Input,T1).

apply_trans_to_conf(trans(In,Out_Name),Conf,NewConf):-
   ord_subtract(Conf,In,Diff),
   flatsort([Out|Diff],Temp),
```

Tabled Negation

- Tabling can be used to compute the 3-valued well-founded semantics
- Used to compute inheritance in object-logics (e.g. Flora, CDF)
- Used to code preferences (e.g. Preference Logic Grammars)
  - Use of prefernce Logic Grammars reduced the size of an industry data "standardizer" by 500%
- Used to compute a residual program to be sent to an ASP solver.
  - An Smodels interface is supported by the XSB package XASP
- For the Workflow Net example, well-founded negation can be used to ensure that a transition T is taken from a configuration C only if no other transition is preferred to T for C.
  - Lookaheads from a given state can be performed
  - This allows preferences to be added to a workflow in a modular way, to support local policies or requirements
  - 1-line change in previous program

```
hasTransition(Conf,NewConf):-
   get_trans_for_conf(Conf,AllTrans),
   member(Trans,AllTrans),
   sk_not(unpreferred(Trans,AllTrans,Conf)),
   apply_trans_to_conf(Trans,Conf,NewConf).
```

- CENTRIA researchers are the leaders in using TLP negation for AI applications

Call Variance vs. Call Subsumption

- When should a tabled subgoal reuse a table and when should it create a new one?
- Two terms are variants if they are identical up to variable names – if their mgu consists only of substitutions of variables for variables. E.g. p(X,Y,Y) and p(A,B,B)
- A term T_1 subsumes a term T_2 if T_1 is more "general" than T_2 – if the an mgu of T_1 and T_2 consists only of bindings to variables in T_1. E.g. p(X,Y,Z) and p(a,B,B)
- Most tabling systems use call variance, which uses a table if a table exists whose subgoal is a variant of a selected subgoal
- Call subsumption is useful for computing fixed-points of programs, e.g. program analysis, RDF inferences, certain types of deductions in OWL (e.g. OWL wine example).
- Call subsumption could be performed over any partial order. For instance, on the ordering of the real numbers, p(X):$\{X < 3\}$ subsumes p(X):$\{X < 2\}$.
  - At the engine level, XSB supports call subsumption only for the partial order (lattice) of terms.
  - In 3.1 call subsumption works only for lrd-stratified programs; 3.2 will have call subsumption for full WFS

Answer Variance vs. Answer Subsumption

- For calls, subsumption is useful on the partial order (lattice) of terms.
- For answers, this is not usually very useful, but other partial orders or lattices are:
  - Interval [0,1] for fuzzy sets
  - Lattice of uncertainty intervals in Dempster-Schaeffer inference
  - Lattices for paraconsitent logics
  - Monotonic recursive aggregates (min, max)
  - Comparing abductive solutions
- Example. Naive reachability for Place/Transition Nets will not terminate. Instead, a technique called ω-abstraction must be used. A place marked with ω tokens indicates that it may contain any number of tokens.
- In XSB, the left-recursive `reachable/2` predicate is rewritten as:

```
reachable(InConf,NewConf):-
   filterPOA(reachable(InConf),Conf,gte_omega,omega_abstr,call_abstr),
   hasTransition(Conf,NewConf).
```

  `filterPOA/5` calls `reachable(InConf,-ConfTemp)` and succeeds if there is no answer for the table (returning `Conf`) or if `Conftemp` is greater than any element in the table for `reachable(InConf,-ConfTemp)` that has key `reachable(Conf,_)`. `ConfTemp` may be abstracted by `omega_abstr/2` and before returning it as `Conf`.

Tabled Constraints

- Many Constraint Logic Programs do not benefit from tabling, as the logic program is used primarily to set up the constraints.
- Tabling *can* help search through a state space where the states can be labelled with constraints.
  - Examples in natural language analysis, program analysis, verification, ILP
- In XSB, attributed variables are copied into and out of tables as any other term
- Need to handle constraint interrupts before accessing the table for a tabled subgoal or answer
- Example. Constraints can be used to encode a type of Colored Petri net. Rather than collecting tokens, a transition collects constraints until the constraint set entails a formula (and does not fire otherwise). Once it fires, new constraints may be applied to the resulting configuration.

```
apply_trans_to_conf(trans(In,Entailment,Out),Conf,NewConf):-
   unify_for_entailment(In,Conf,MidConf),
   entailed(Entailment),
   call_new_constraints(Out,OutPlaces),
   flatsort([OutPlaces|MidConf],NewConf).
```

`unify_for_entailment/3` simply unifies variables in the transition with those of the configuration to produce a constraint store for entailment checking.

Incremental Tabling

- Answers for a tabled subgoal may depend on dynamic facts used in the derivation of the answers. When these facts change, the answers may become invalid.
  - Incremental tabling provides automatic recomputation of answers as dynamic facts are asserted or retracted.
  - Useful for maintaining graphical views of an underlying logical model. For instance, user interfaces driven by Interprolog/XJ or deductive spreadsheets.

Table management and space reclamation

- Tables can be explicitly abolished (truncated), and are implicitly abolished when a declaring the tabled predicates is reconsulted.
  - Tables can be abolished at the subgoal or predicate level. In addition, all tables for predicates in a given module can be abolished, as well as all tables present in the system.
  - Space for abolished tables may not be immediately reclaimable if there are choice points that will backtrack through these tables. In XSB, garbage collection is performed at the subgoal and predicate level.
  - When well-founded evaluation creates a residual program (with clauses) deleting a single table could lead to dangling pointers. E.g. if a residual clause were `p(a):- tnot(q(b))`, deleting the subgoal `q(b)` or the predicate `q/2` may lead to dangling pointers in the table for `p(a)`. In such a case, XSB will perform a *cascading abolish* that also abolishes `q(a)`, and so on.
  - YAP allows automatic reclamation via a least-recently-used algorithm.

Goals for a Multi-Threaded Tabling System

1. Any tabling function should be available to any active thread using tables that are private to a thread.
2. Any tabling function should be available to any active thread using tables that are shared among threads.
3. Private tables should be highly scalable up to the number of cores available.
4. For problems that support large amounts of parallelism, shared tables should be able to provide speedup proportional to the number of cores availabl

These goals are not easy to fulfill!

MT-TLP: Private Tables

- Private tables are suitable to ensure query completeness or to support a particular semantics (WFS, GAPs, Preference Logics).
- Private tables use sequential tabling algorithms, but with thread-safe code. They generally require no synchronization above the level of memory management.
- Each thread can access the residual program for its private tables
  - Each thread must be able to reclaim its own table space upon exit or explicit abolish (including delay listes, etc. to support WFS)
  - Must ensure that memory allocation for table space data structures is efficient

MT-TLP: Summary of Private Table Functionality

Feature	Private Tables
Tabled constraints	Supported
Answer subsumption	Supported
Tabled Dynamic Code	Supported
Tabled negation	Supported
Space reclamation	Supported
Call subsumption	Supported
Incremental recomputation	“Almost” supported
XASP (Residual Program $\rightarrow$ Smodels)	“Almost” supported

Goal 1: *Any tabling function should be available to any active thread using tables that are private to a thread.* is “almost” supported.

Private Table Performance

N. threads	1	2	Overhead	4	Overhead
`Elementary Net`	5.94	6.23	4.8%	6.25	5.2%
`Dynamic Elementary Net`	6.03	6.03	0%	6.03	0%
`Workflow Net`	19.21	19.68	2.4%	19.95	3.8%
`Omega Net`	7.18	8.33	16.0%	10.3	46.0%
`Omega Net Specialized`	6.37	6.37	0%	6.37	0.0%
`Constraint Net`	2.75	2.84	3.2%	2.85	3.6%
`Preference Net`	3.74	3.77	0.8%	3.82	2.1%
`Call Subsumption`	.86	1.04	20.0%	1	43%

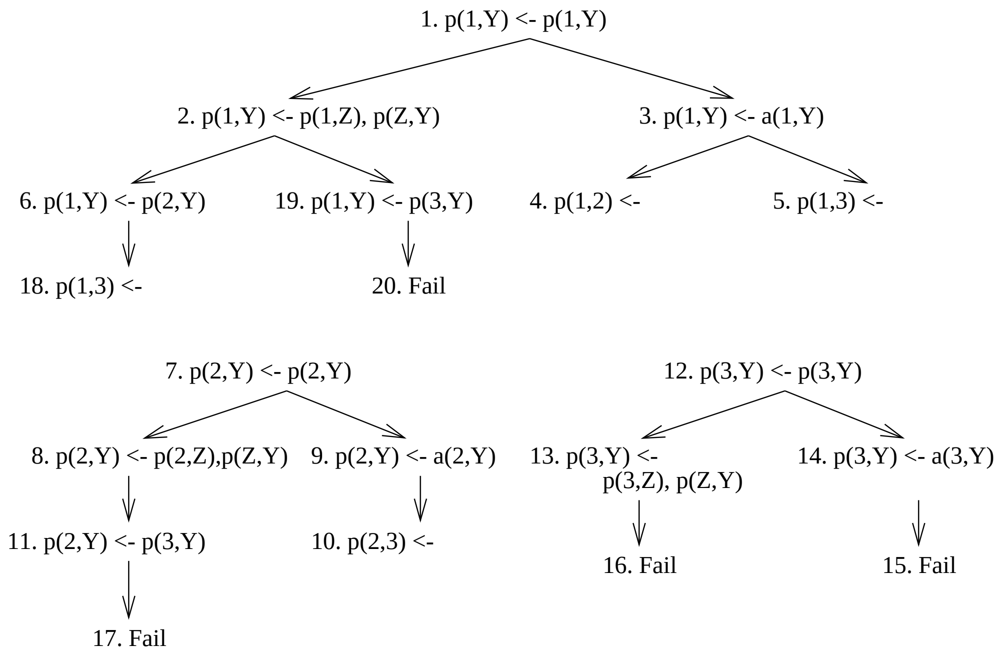
- `Elementary Net` uses tabled definite programs for Petri Net reachablilty; `Dynamic Elementary Net` is the same, but uses dynamic clauses. `Workflow Net` uses definite programs to test workflows.
- `Constraints Net` tests the constraint-based colored Petri nets, and `Preferences Net` tests extension for well-founded preferences.
- `Omega Net` tests ω-abstraction using answer subsumption. `filterPOA/5` relies on `call/N`, which accesses XSB's shared predicate table, causing poor performance. A specialized version, `Omega Net Specialized` avoids this problem
- `Call Subsumption` tests an unbound call to right recursion over a graph. Its poor speedup is apparently due to high demands for memory management
- Goal 3: *Private tables should be highly scalable up to the number of cores available* is mostly satisfied (for 4 cores), although more work needs to be done on call subsumption.
- All performance numbers in this paper were obtained on a 4-core linux machine whose use was kindly provided by Luis Caires

Shared Tables!la

- Computing shared tables in a MT framework depends critically on the scheduling strategy for tabling.
- Let's digress for a few slides.

Scheduling: Local Evaluation

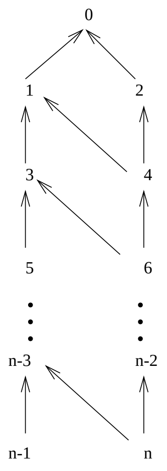
- Local Evaluation: Completely evaluate each mutually dependent set of subgoals before returning an answer to a subgoal not in that set.



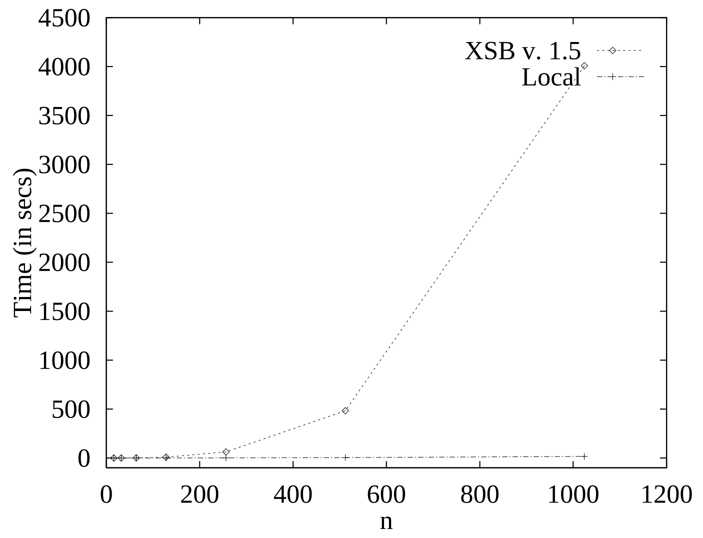
```
:- table p/2.
p(X,Y) :- p(X,Z), p(Z,Y).
p(X,Y) :- a(X,Y).
a(1,2).  a(1,3).  a(2,3).
```

Scheduling: Local Evaluation

```
sgi(X,Y)(D) :- arc(X,Y).
sgi(X,Y)(D) :-
     arc(X,Z), subsumes(min)(sgi(Z,Z1),D1),
     arc(Y,Z1), D is D1+1.
```



(a)

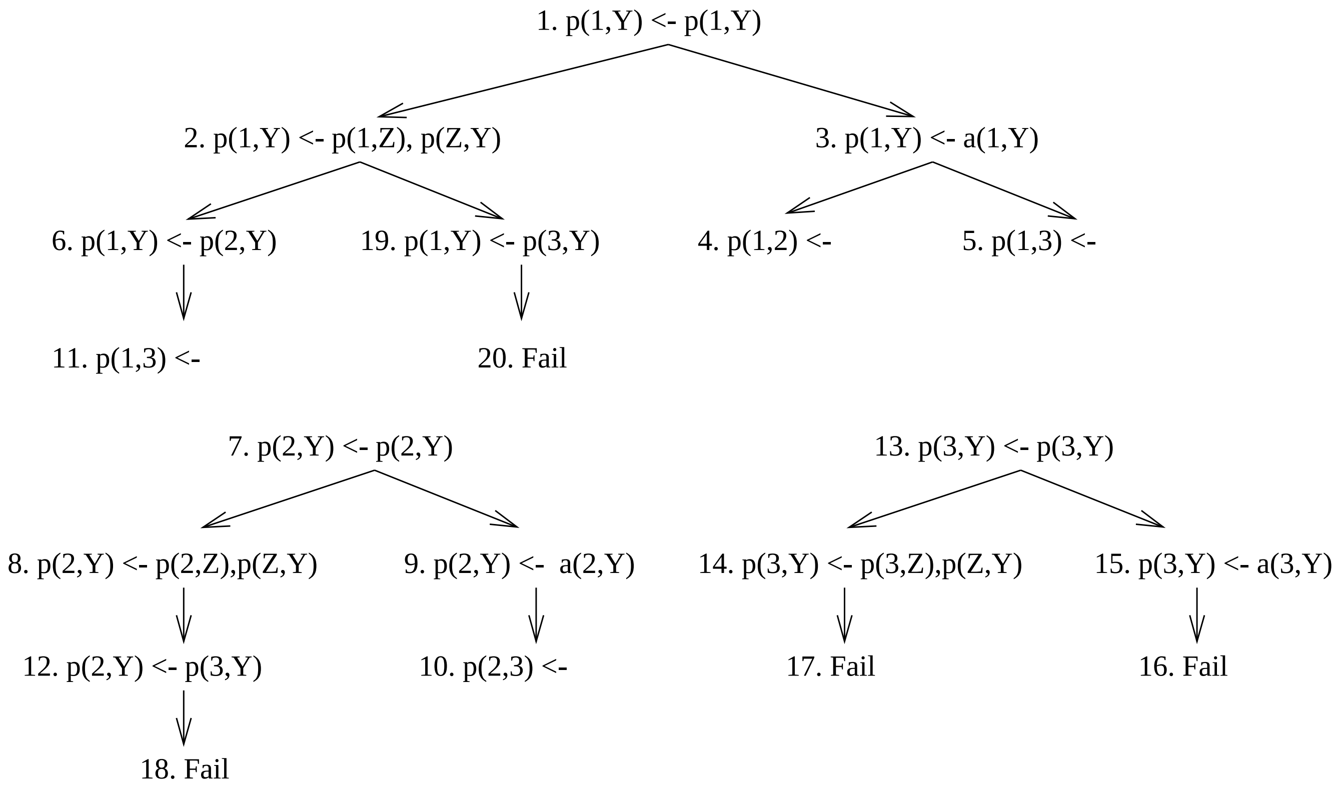


(b)

Figure 1: *(b) shows the execution time for the query* `subsumes(min)(sgi(n-1,n),I)` *on graphs of the form depicted in (a) for varying* `n`

Scheduling: Batched Evaluation

- Batched Evaluation: Return an answer to parent environments as soon as it is derived.



```
:- table p/2.
p(X,Y) :- p(X,Z), p(Z,Y).
p(X,Y) :- a(X,Y).
a(1,2).  a(1,3).  a(2,3).
```

Scheduling Strategies

- *Negation and tabled aggregation* Local Evaluation is superior for tabled aggregation as only optimal answers are returned out of a maximal SCC. Local Evaluation also can be more efficient for non-stratified negation as it may allow delayed answers that are later simplified away to avoid being propagated.
- *Time to first answer* Because Batched Evaluation returns answers out of an SCC eagerly, it is faster to derive the first answer to a tabled predicate.
- *Time for left recursion* Batched Evaluation is somewhat faster than Local Evaluation for left recursion as Local Evaluation imposes overhead to prevent answers from being returned outside of an independent SCC.
- *Stack space* Local Evaluation ofte requires less space than Batched Evaluation as it fully explores a maximal independent SCC, completes the SCC's subgoals, reclaims space, and then moves on to a new SCC.
- *Efficiency for call subsumption* Because Local Evaluation completes tables earlier than Batched Evaluation it can be faster for call subsumption, as subsumed calls can make use of completed subsuming tables.
- *Integration with cuts* Local Evaluation integrates better with cuts, as tabled subgoals may be fully evaluated before the cut takes effect.

Shared Tables: Scheduling

- In a multi-threaded environment, Local Evaluation and batched evaluation have radically different properties
- Consider a simple "producer-consumer" fragment

```
:- table producer/1, consumer/1.
consumer(Term):- producer(Term).
```

  - In Local Evaluation the producer is prohibited from returning terms to the consumer since they are not mutually dependent (in the same SCC)
  - No prohibition is made in batched
  - Yet the properties of Local Evaluation are still needed, particularly for returning "best" the answers out of an SCC.

Shared Tables: Local Evaluation

- In Local Evaluation, tables are shared among threads only if they are completed (completely evaluated).
- Nonetheless, this is useful in a number of situations
  - E.g. a web server for Flora or some other object logic. Queries to the T-box (inheritance, attributes of clases) may be shared for amortization, while queries to the A-box may be thread-private
  - While it is mainly oriented towards allowing multiple threads to evaluate shared tables concurrently using Local Evaluation, coarse parallelism can be exploited when a search space can be partitioned into (mostly) disjoint subspaces.

Shared Tables: Local Evaluation

- Implementation is based on *Concurrent Local SLG*
  - If $Thread_1$ calls a table T that is being computed by a different thread, $Thread_2$, $Thread_1$ checks to see if a some tables owned by $Thread_1$ and $Thread_2$ are mutually dependent (perhaps through other threads).
  - If not, $Thread_1$ suspends until T is completed (by $Thread_2$ or some other thread)
  - If so, $Thread_1$ *usurps* the all tables in the SCC $\mathcal{S}$ of T. All other threads that own tables in cS are suspended until $\mathcal{S}$ is completed.
- *Concurrent Local SLG* has been formalized.
  - It has the same completeness, termination, and complexity as Local SLG
  - N threads, each performing a local evaluation, together perform a local evaluation

Shared Tables: Local Evaluation

Concurrent Local SLG depends on the definition of a *Subgoal Dependency Graph*

Definition 1 (Subgoal Dependency Graph) *Let $\mathcal{F}$ be a forest in a SLG evaluation. A tabled subgoal S_1* directly depends on *a tabled subgoal S_2 in $\mathcal{F}$ iff neither the tree for S_1 nor that for S_2 is marked as complete and S_2 is the selected literal of some node in the tree for S_1. The* Subgoal Dependency Graph of $\mathcal{F}$, SDG($\mathcal{F}$)*, is a directed graph* V,E *in which V is the set of root goals for non-completed trees in $\mathcal{F}$ and $(S_i, S_j) \in E$ iff S_i directly depends on S_j.*

$$\begin{tikzcd}
p(a,Z) \arrow[d] \arrow[loop right] \\
p(b,Z) \arrow[d] \arrow[loop right] \\
p(c,Z) \arrow[loop right]
\end{tikzcd}$$

Shared Tables: Local Evaluation

- There is a function from SLG forests to SDGs – i.e. a forest $\mathcal{F}$ defines $SDG(\mathcal{F})$
- Since SDGs are directed graphs, Strongly Connected Components (SCCs) can be defined for them. A maximal SCC is one that is contained in no other SCC, and an independent SCC $\mathcal{S}$ is one such that no subgoal in $\mathcal{S}$ depends on a subgoal not in $\mathcal{S}$
- Incremental Completion can be performed an SCC at a time, or a set of SCCs at a time.
- In a concurrent SLG evaluation, each tree for an incomplete subgoal is associated with a single thread.
  - The SDG is taken to be the SDG of the combined forest of each thread's evaluation and a thread T's SDG (SDG(T)) is the SDG formed by the subforest of trees that T owns.
  - The *Thread Dependency Graph* (TDG) represents the dependencies of a subgoal owned by any given thread on a subgoal owned by another thread

Shared Tables: Local Evaluation

It can be proved that in a Concurrent Local Evaluation

- Each suspended thread is suspended on a single subgoal and on a single thread
- Any thread T contains a single independent SCC in $SDG(T)$

These properties considerably simplify the algorithm.

- Any deadlock is a simple cycle in the TDG,
- If a thread T detects a deadlock, all threads in the deadlock cycle will be suspended except for T
- Each suspended thread T can be awakened when the subgoal on which it was suspended completes. T then resumes execution by backtracking through answers for the completed table

Shared Tables: Local Evaluation

- Implementation changes mostly concern the tabletry instruction that is called when a tabled subgoal is encountered.
- completion isntruction also wakes up threads suspended on a completed subgoal

Instruction tabletry (sequential version)

```
    /* $Subg$ is in argument registers; $T_{current}$ is current thread */
    Perform the subgoal_check_insert($Subg$) operation in the table
    If $Subg$ is new
        Create a generator choice point to resolve program clauses
    Else if $Subg$ is incomplete
        Create a consumer choice point to resolve answer clauses
    Else if $subg$ is complete
        Branch to root of trie to execute instructions for completed table
```

Shared Tables: Local Evaluation

Instruction tabletry (Concurrent Local Version)

```
/* $Subg$ is in argument registers; $T_{current}$ is current thread */
Perform the subgoal_check_insert($Subg$) operation in the table
If $Subg$ is not new and is marked by another thread
    Lock global TDG mutex
    If deadlock($T_{current}$,$Subg.ThreadMark$)
        /* all other threads in the independent SCC are suspended at deadlock *
        usurp($T_{current}$,$Subg$,$Subg.ThreadMark$)
    Else unlock TDG mutex; suspend the calling thread until $Subg$ completes
/* Proceed as in the sequential case */
/* if $Subg$ was usurped, treat it as a new subgoal */
If $Subg$ is new
    Create a generator choice point to resolve program clauses
    Unlock global TDG mutex
Else if $Subg$ is incomplete
    Create a consumer choice point to resolve answer clauses
Else if $subg$ is complete
    Branch to root of trie to execute instructions for completed table
```

Shared Tables: Local Evaluation

Because of the properties of a SDG for Local Evaluation

- deadlock() needs to traverse only a simple cycle
- usurp() can rely on the usurped threads being suspended

```
deadlock($T_{current}$,$depends\_thread$ )
    while( $depends\_thread \neq$ NULL )
        if( $depends\_thread = T_{current}$ ) return true;
        else $depends\_thread \leftarrow subgoal\_thread.suspended\_on\_thread$);
    return false;
```

```
usurp($T_{current}$, $dep\_SF$, $first\_usurped$)
    Traverse SCC to reset $suspended\_on\_thread$ dependencies
    Unlock global mutex that protected TDG
    Traverse SCC to
        Reset stacks of each (suspended) usurped thread
        Propagate the proper subgoal dependency to each usupred thread
```

Shared Tables: Local Evaluation

- As currently implemented in XSB, a usurping thread rederives usurped computations from scratch (although it does not need to re-insert previously derived answers into the table).
- Details of the implementation are subtle; however they amount to about 300 lines of code added to `tabletry` with minimal refactoring of existing code.
  - This means that the approach is quite general for various tablig functions (as shown below)
  - It also means that there is little overhead for this approach beyond overheads for shared table space (i.e. one or two new conditions in `tabletry`)
  - It also means that the approach is portable: all tabling systems execute special code when encountering a tabled subgoal

Shared Table Functionality for Local Evaluation

Feature	Shared Tables (Local)
Tabled Dynamic Code	Supported
Tabled constraints	Supported
Answer subsumption	Supported
Tabled negation	Supported
XASP (Residual Program $\rightarrow$ Smodels	"Almost" supported
Space reclamation	Partially Supported
Call subsumption	Not supported
Incremental recomputation	Not supported

- The generality of Concurrent Local SLG means that it can support most tabling functionality
- Full space reclamation for abolished tables is not yet fully supported because of lack of resources to implement a table garbage collector in a multi-threaded environment.
- Further work needs to be done to extend Concurrent Local SLG to call subsumption and to assess its efficiency.
- Incremental recomputation is difficult for shared tables, since each thread that is using a table must maintain its view of the table throughout a query

Shared Tables: Efficiency of Concurrent Local SLG

N. threads	1	2	Speedup	4	Speedup
`Shared Elementary`	25.12	13.00	1.93	6.55	3.83
`Shared Dynamic Elemtary`	24.8	13.02	1.90	6.59	3.76
`Shared Workflow`	41.25	20.78	1.98	10.58	3.89
`Shared Omega`	19.58	10.38	1.88	5.57	3.51
`Shared Constraint`	11.13	5.56	2.00	2.83	3.93
`Shared Preferences`	3.73	1.86	1.99	0.95	3.92

- Benchmarks show that usurpation occurs surprisingly infrequently in pratice.
- Overhead of shared tables over private tables is about 10-40 depending on benchmark and platform
- Overheads are mostly due to handling mutual exclusion for tables and to memory allocation. Further work is needed in these areas.

Shared Tables: Batched Evaluation

- For Concurrent Batched Evaluation, if one thread calls an (incomplete) table owned by another thread, it backtracks through the existing answers, rather than suspending
- This fixes Producer/Consumer example, and is all you need unless two threads own subgoals in the same SCC
- When there are intra-SCC dependencies among threads, Concurrent Batched Evaluation is a generalization of Sequential Batched Evaluation.
- In a sequential (or thread-private) computation, when the engine backtracks to the oldest subgoal in an SCC, it schedules the return of unconsumed answers for each consuming node in the SCC by creating a chain of choice points, and then backtracks into the newly created chain.

Shared Tables: Batched Evaluation

In a concurrent computation, let $\mathcal{S}$ be an SCC with subgoals owned by different threads

- Suppose a thread T_1 computing subgoals in $\mathcal{S}$ backtracks to the oldest subgoal that it "owns" in $\mathcal{S}$.
- If another thread computing $\mathcal{S}$ is active, T_1 will suspend and will be wakened when a thread performs batched scheduling for $\mathcal{S}$;
- Tf T_1 is the last unsuspended thread computing subgoals in $\mathcal{S}$, T_1 itself will perform a fixed point check and batched scheduling and awaken the other threads computing $\mathcal{S}$ — either to return further answers or to complete their tables.
- As implemented in XSB, Concurrent Batched Evaluation thus allows parallel computation of subgoals, but has a sequential fixpoint check that synchronizes multiple threads when they compute the same SCC.

Shared Table Functionality for Batched Evaluation

Feature	Shared Tables (Batched-β)
Tabled Dynamic Code	Supported
Tabled constraints	Supported
Answer subsumption	Supported
XASP (Residual Program $\rightarrow$ Smodels	“Almost” supported
Tabled negation	Partially Supported
Space reclamation	Partially Supported
Call subsumption	Not Supported
Incremental recomputation	Not Supported

- Unlike Concurrent Batched Evaluation, Concurrent Local Evaluation does not yet support full WFS.
- Currently adjusting heuristics to coordinate behavior between threads