Concurrent and Local Evaluation of Normal Programs

Rui Marques
CITI, Dep. Informatica FCT, Universidade Nova de Lisboa

Terrance Swift
CENTRIA Universidade Nova de Lisboa

Motivation

- Develop a MT-TLP system
  - That supports a variety of tabling functions
  - That is maintainable
  - That uses algorithms that are provably correct
    - Otherwise, the complexity of
      tabling + abstract engine + concurrency
      is too much
- Here we examine a critical feature of such a system using a scheduling strategy called Local Evaluation

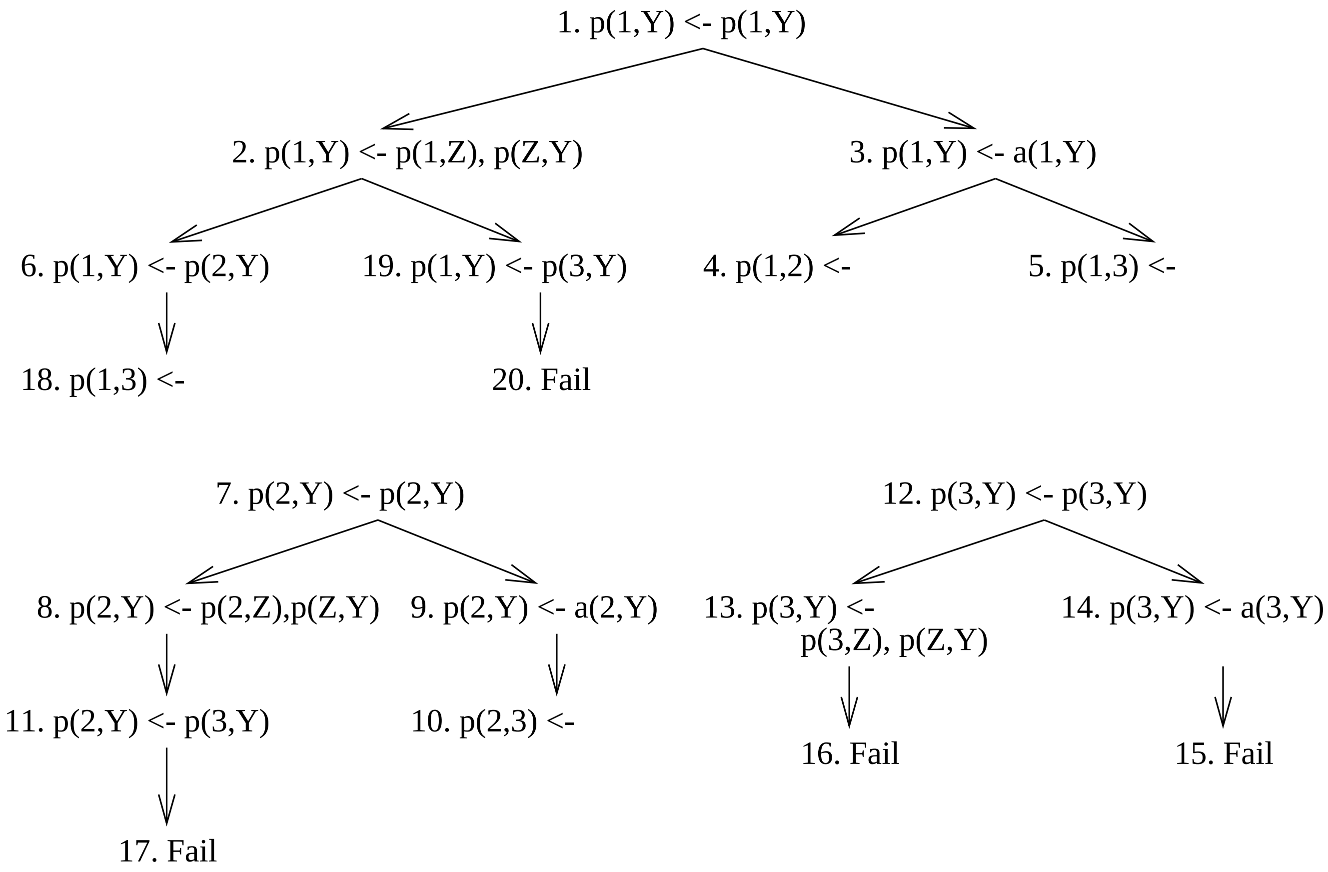
This presentation assumes a minimal amount about tabling and so simplifies various definitions: see the paper for all the formal details

Local Evaluation: Model of Tabling

- A *tree* models the steps taken to derive answers for a tabled subgoal
  - In this presentation we may refer to subgoals and their trees interchangeably
  - A tree that has been completely evaluated may be marked as *completed*
- A *forest of trees* models the state of an evaluation
- A (transfinite) *sequence* of forests models an evaluation

Local Evaluation: Main Idea

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

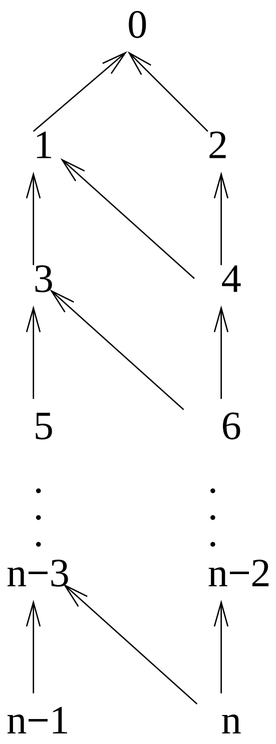


1. p(1,Y) <- p(1,Y)
2. p(1,Y) <- p(1,Z), p(Z,Y)
3. p(1,Y) <- a(1,Y)
6. p(1,Y) <- p(2,Y)
19. p(1,Y) <- p(3,Y)
4. p(1,2) <-
5. p(1,3) <-
18. p(1,3) <-
20. Fail
7. p(2,Y) <- p(2,Y)
12. p(3,Y) <- p(3,Y)
8. p(2,Y) <- p(2,Z),p(Z,Y)
9. p(2,Y) <- a(2,Y)
13. p(3,Y) <- p(3,Z), p(Z,Y)
14. p(3,Y) <- a(3,Y)
11. p(2,Y) <- p(3,Y)
10. p(2,3) <-
16. Fail
15. Fail
17. Fail

```
:- 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).
```

Local Evaluation: Example

```
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.
```

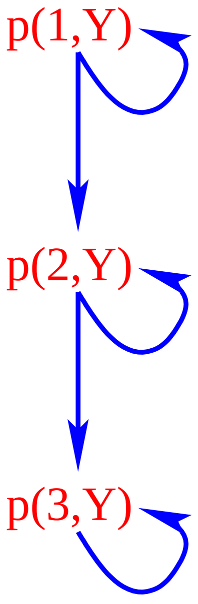


- Time for `?-p(bound,free)` is linear in edges for Local Evaluation, Linear in size of paths for Batched Evaluation

Local Evaluation: Details

Let's be more precise about Local Evaluation (see the paper for the "fine print"). Let $\mathcal{F}$ be a forest in a tabled evaluation

- A subgoal dependency graph *Subgoal Dependency Graph* for $\mathcal{F}$
  - Has a vertex for each non-completed subgoal in $\mathcal{F}$
  - Has an edge (S_1, S_2) if S_2 is the underlying subgoal of a selected literal (or a delay literal) in the tree for S_1



Local Evaluation: Details

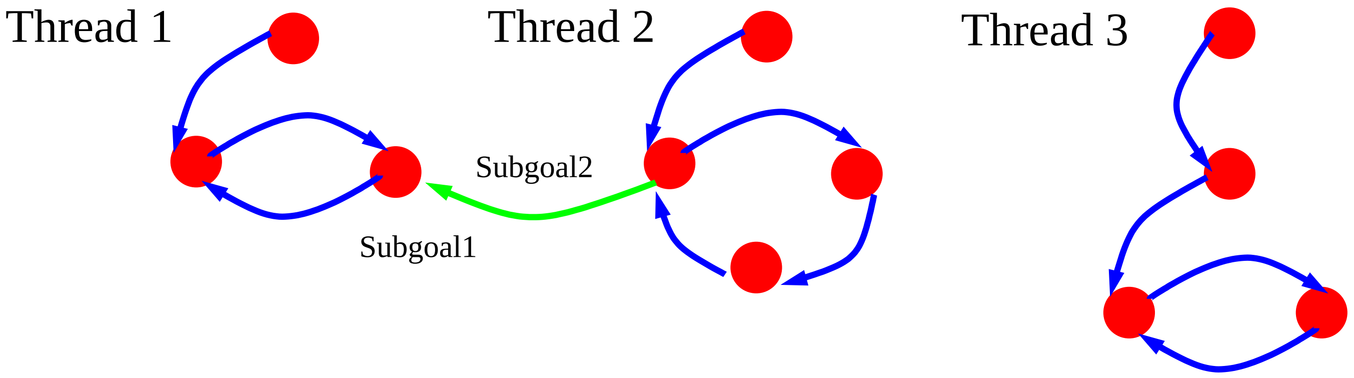
- There is a function from SLG forests to SDGs – i.e. a forest $\mathcal{F}$ defines $SDG(\mathcal{F})$
- Since SDGs are directed graphs, Strongly Connected Components (SCCs) can be defined for them.
  - A *maximal* SCC is contained in no other SCC
  - An *independent* SCC $\mathcal{S}$ is one where no subgoal in $\mathcal{S}$ depends on a subgoal not in $\mathcal{S}$
- A *Local Evaluation* is one where an operation is applied only to trees whose subgoals are in a maximal independent SCC (modulo a few small provisos)

Concurrent SLG

Concurrent SLG adds a few new definitions to SLG

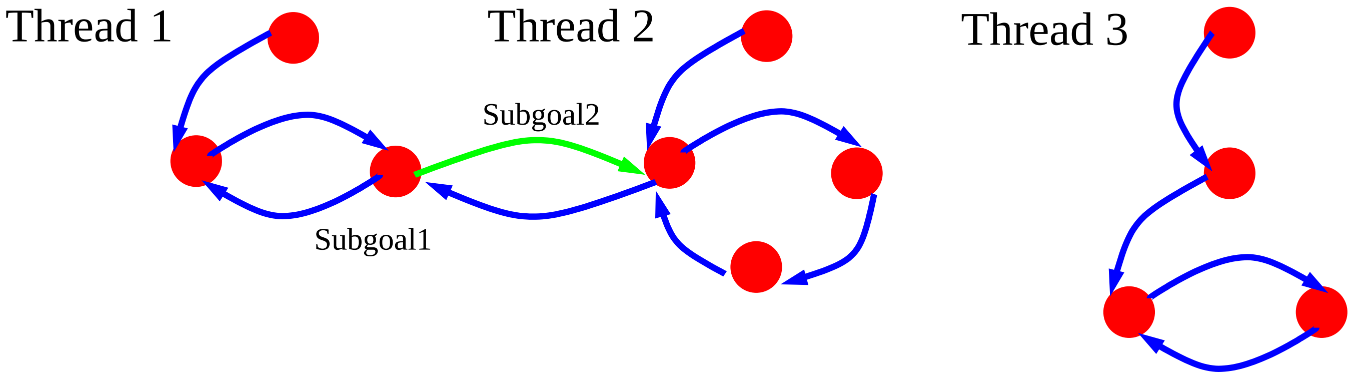
- Each tree T in a forest ($\mathcal{F}$) is marked with a unique thread id (informally $Thread$ owns T or the subgoal of T)
- A node N is *thread compatible* with a subgoal S if S is complete or S and N are owned by the same thread
- A tabled literal cannot be resolved (or delayed or simplified) unless its node is thread compatible with its selected subgoal
- A set of subgoals cannot be completed unless they are owned by the same thread
- This can lead to *deadlock* – mutually dependent subgoals owned by different threads where no operations are possible on trees for those subgoals
- To resolve this, a new USURPATION operation is defined
  - If $Thread_1$ owns a subgoal that is in a deadlock, it can usurp (remark) all the subgoals in that deadlock cycle

Concurrent Local SLG: Example



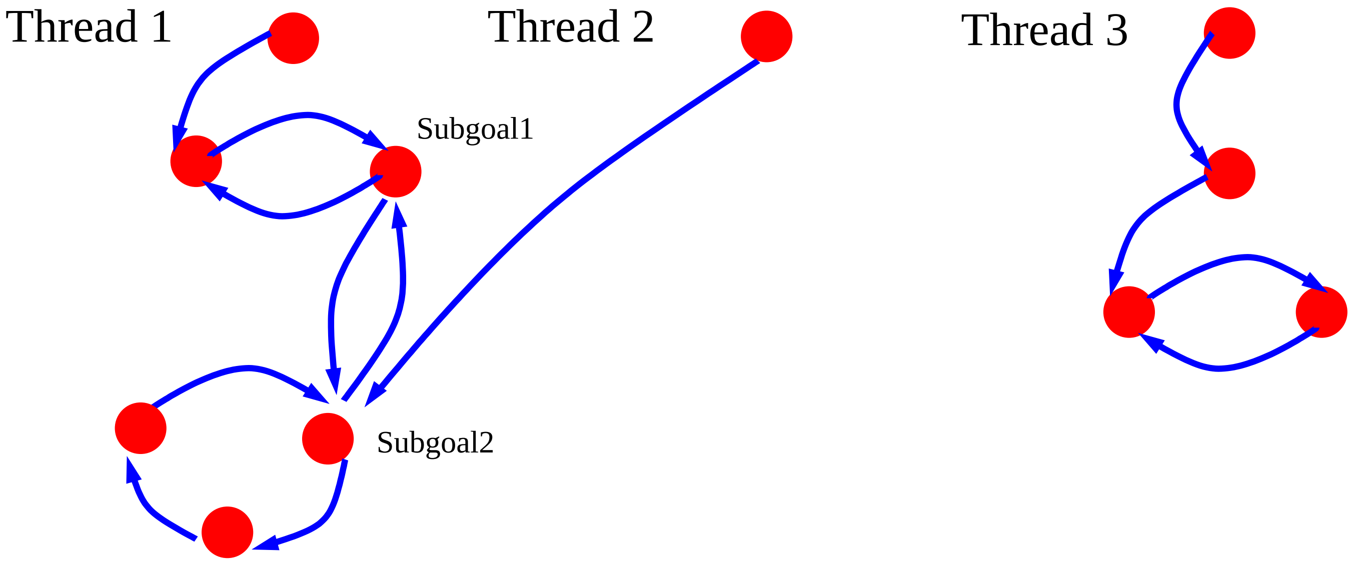
- Abstract SDG for a computation
- $Subgoal_2$ owned by thread 2 calls $Subgoal_1$ owned by thread 1
- For thread 2 to perform a Local Evaluation, it must apply operations in the tree for $Subgoal_1$
- However, since $Subgoal_1$ is owned by thread 1, it is not thread compatible with thread 2
- No operations apply to subgoals owned by thread 2 in this forest; (thread 2 suspends)

Concurrent Local SLG



- Now $Subgoal_1$ calls $Subgoal_2$
- No operations are applicable to the SCC owned by thread 1 and thread 2: it is in *deadlock*

Concurrent Local SLG



- Thread 1 *usurps* the subgoals of thread 2 and remarks the trees (and subgoals)
- Now thread compatability restrictions don't apply

Concurrent Local SLG: Dependencies

- $SDG(\mathcal{F})$ is as before.
- $SDG(\mathcal{F}, Thread)$ is $SDG(\mathcal{F}_{Thread})$ where $(\mathcal{F}_{Thread})$ is the subforest of $\mathcal{F}$ that T owns.
- The *Thread Dependency Graph* of $\mathcal{F}$ $(TDG(\mathcal{F}))$ is defined as follows
  - The vertices of $TDG(\mathcal{F})$ are the thread ids marking trees in $\mathcal{F}$
  - $(Thread_1, Thread_2)$ is an edge in $TDG(\mathcal{F})$ if (S_1, S_2) is an edge in $SDG(\mathcal{F})$, $Thread_1$ owns S_1, and $Thread_2$ owns S_2

Properties of Concurrent Local SLG

- Concurrent SLG has same correctness and termination properties as SLG
- Addition of USURPATION does not change complexity of SLG
- Concurrent Local SLG is based on locality in the $SDG(\mathcal{F}, Thread)$ for each thread
  - Has the same correctness and termination properties as Local SLG
  - N threads, each performing a Local Evaluation on $SDG(\mathcal{F}, Thread)$, together perform a Local Evaluation on $SDG(\mathcal{F})$ (N finite)

Concurrent Local SLG: Operational Properties

- Any thread $Thread$ contains a single maximal independent SCC in $SDG(\mathcal{F}, Thread)$
- Each node in $TDG(\mathcal{F})$ has at most one outgoing edge
  - i.e, any deadlock is a simple cycle in $TDG(\mathcal{F})$

These properties considerably simplify the algorithm. A thread suspends when it selects a literal that is not thread compatible with itself. Thus

- 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 can then resumes execution by backtracking through answers for the completed table)

Concurrent Local SLG: Implementation

- 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
```

Concurrent Local SLG: Implementation

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
```

Concurrent Local SLG: Implementation

- 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).
  - So far, experiments show that usurpation occurs surprisingly rarely
- 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 tabling 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

Summary

- Local Evaluation is critical for Answer Subsumption (quantitative and paraconsistent logics, maximal abduction) and useful for WFS. Also reduces stack space for many computations.
- Our approach shares completed tables *only*. This leads to an implementation
  - Whose correctness and critical properties are provable
  - That can be modularly added to a tabling engine
  - Is currenly working in XSB (please use anonymous CVS until v. 3.2 comes out)
  - Supports WFS (including residual programs), Answer Subsumption, and Tabled Constraints
- Concurrent Batched Evaluation allows a tabled consumer and producer to communicate in a manner analogous to message queues. XSB contains an experimental version of this
- In the longer term, we are working on have dynamic scheduling of Concurrent Local and Batched Evaluation