Concurrent and Local Evaluation of Normal Programs

Rui Marques[1] and Terrance Swift[2]

[1] CITI, Dep. Informática — FCT, Universidade Nova de Lisboa
[2] CENTRIA — Universidade Nova de Lisboa

Abstract. Tabled evaluations can incorporate a number of features, including tabled negation, reduction with respect to the well-founded model, tabled constraints and answer subsumption. Many of these features are most efficiently evaluated using the Local evaluation strategy, which fully evaluates each mutually dependent set of tabled subgoals before returning answers to other subgoals outside of that set. In this paper, we introduce a formalism, Concurrent Local SLG by which multiple threads of computation concurrently perform Local evaluation of the well-founded semantics, and which is a framework for multi-threaded tabling in the XSB system. We prove several properties of Local evaluation within single-threaded tabled computation. We then extend SLG to a model of concurrency and show that the completeness and complexity of SLG are retained when computed by multiple threads. Finally, we extend Local evaluation to concurrent SLG, and show that the properties of Local evaluation continue to hold under concurrency.

This paper provides an operational semantics for a type of concurrent TLP that relies on a scheduling strategy called Local evaluation [4]. The model of concurrency adopted is one in which threads of computation execute separate subgoals while sharing completed tables. The main idea behind Local evaluation is that it fully evaluates a single mutually dependent set of tabled subgoals before performing operations (such as returning answers) to subgoals outside of that set. Experiments in several implementations have shown that Local evaluation utilizes space efficiently (see e.g. [4, 10]) and as a result it has been implemented for several Prologs.

Another feature of Local evaluation is shown in an example in [4] in which tabling was used to compute the shortest path between two nodes. When Local evaluation was used the shortest path could be computed in a time proportional to the number of nodes in the graph, while if a non-Local scheduling strategy was used the time was proportional to the number of *paths* in the graph – i.e. the time was exponential in the number of nodes. Comparing path lengths to compute a shortest path can be considered as an instance of *answer subsumption* in which answers are retained and propagated only if they are maximal over a partial order or are a monotonic function of answers so far produced.

Using SLG resolution [1] as a basis, this paper presents the following results about concurrent and Local evaluations.

– As analysis of Local evaluation in the literature has been mostly empirical, Local SLG evaluation is formally defined in Section 2 and shown complete

for queries to normal programs. Properties are derived about dependencies between subgoals in a Local evaluation, about the return of answers, and about the extent of non-completed subgoals in an evaluation.
- Section 3 presents SLG$_C$, an extension of SLG to concurrent evaluations in which completed tables are shared among threads. SLG$_C$ is complete for queries to normal programs, and its abstract complexity is the same as SLG.
- Concurrent *Local* SLG (Local SLG$_C$) is then defined in Section 3.1. It is shown that properties of Local SLG evaluations extend to the sub-evaluations performed by each concurrently executing thread, and a property is derived about the structure of dependencies between threads.
- Section 4 sketches the implementation of Local SLG$_C$ in XSB, where the engine design is directly motivated by the preceding results for subgoal and thread dependencies. In addition to having the properties of finite evaluations presented in this paper, XSB's implementation of Local SLG$_C$ has been extended to support tabled constraints, answer subsumption, tabled dynamic code, and space reclamation.

We begin with a review of SLG evaluation.

1 SLG Evaluation

This presentation of SLG reformulates the operations of [1] using the model of a forest of trees. However, for reasons of space we make the following restrictions throughout this paper. First, the formal definitions in this paper consider only finite evaluations, although the statements of theorems that are true for transfinite evaluations are not restricted. Second, our definition of Completely Evaluated (Definition 4) does not permit Early Completion. And third, we do not formally define the concept of a supported answer. All of this formalism can be found in the full version of this paper, available at `http://www.cs.sunysb.edu/~tswift/papers.html`.

Terminology and assumptions We assume the standard terminology of logic programming and an understanding of the well-founded semantics (see [12]). All programs discussed are normal, and defined over a countable language of predicates and function symbols. If L is a literal, the *underlying subgoal* of L is L if L is positive and S if $L = not\ S$. A *3-valued interpretation* I of a program P is a set of literals defined over the Herbrand base of P, H_P. For $A \in H_P$, if $A \in I$, A is true in I, and if $not\ A \in I$, A is false in I; otherwise A and $not\ A$ are undefined in I. When I is an interpretation and A is an atom, $I|_A$ refers to

$$\{L \mid L \in I \text{ and } (L = G \text{ or } L = not\ G) \text{ and } G \text{ is in the ground instantiation of } A\}$$

The well-founded model of a program P is denoted as $WFM(P)$. In the following sections, we use the terms *goal*, *subgoal*, and *atom* interchangeably. Variant terms are considered to be identical.

The nodes in SLG trees are built from atoms and default literals along with a special type of literal called a *delay* literal.

Definition 1 (Delay Literals). *A* negative delay literal *has the form not A, where A is a ground atom. A* positive delay literal *has the form* A^{Subg}_{Ans}*, where A is an atom whose truth value is based on that of some answer Ans for the subgoal Sub. If* θ *is a substitution, then* $(A^{Subg}_{Ans})\theta = (A\theta)^{Subg}_{Ans}$.

The annotations in positive delay literals are used to propagate truth values when a given answer to a given subgoal becomes unconditionally true or false.

Definition 2 (*SLG* Trees and Forest). *An SLG forest consists of a set of SLG trees. Nodes of SLG trees have the form:*

$$Answer\_Template \text{ :- } DelaySet|GoalList$$

or simply fail. *In the first form, the Answer_Template is an atom, DelaySet is a set of delay literals and GoalList is a sequence of literals. The second form is called a* failure node.

An SLG tree T *is associated with a (possibly empty)* marking sequence*, which is a sequence of terms possibly preceded by the distinguished term* complete*. The first element of the marking sequence for T is denoted as* marking(T)*. For a term t,* $setMark(T, t)$ *prepends t to the marking sequence of T.*

A node N is an answer *when it is a leaf node for which* GoalList *is empty. If the DelaySet of an answer is empty it is termed an* unconditional answer*, otherwise, it is a* conditional answer.

The root node of a given SLG tree has the form S :- $|S$ where S is a subgoal — a property ensured by Definition 6. Thus, within a forest each tree and subgoal are uniquely associated, so when T is an SLG tree in a forest $\mathcal{F}$ whose root node is S :- $|S$ it is sometimes convenient to use the terminology S *is the root node for* T; T *is the tree for* S; and S is in $\mathcal{F}$. If $marking(T) = complete$, we refer to both S and T as *completed*. Until Section 3, marking sequences will either be empty or will contain only the term *complete*. Literals in a *GoalList* are resolved by an arbitrary but fixed literal selection strategy. For simplicity, throughout this paper literals are always selected in a left-to-right order.

SLG operations transform one forest of trees into another. One of the operations, Answer Return is based on answer resolution, which is extended to take account of delay literals.

Definition 3 (Answer Resolution). *Let N be a node* A :- $D|L_1, ..., L_n$*, where* $n > 0$*, and* $Ans = A'$:- $D'|$ *an answer whose variables have been standardized apart from N. N is* SLG resolvable *with Ans if* $\exists i$*,* $1 \leq i \leq n$*, such that* L_i *and* A' *are unifiable with an mgu* θ*. The* SLG resolvent *of N and Ans on* L_i *is:*

$$(A \text{ :- } D|L_1, ..., L_{i-1}, L_{i+1}, ..., L_n)\theta$$

if D' *is empty; otherwise the resolvent has the form:*

$$(A \text{ :- } D, L_i{}^{L_i}_{A'}|L_1, ..., L_{i-1}, L_{i+1}, ..., L_n)\theta$$

The SLG Completion operation marks a set of trees as *complete* when they can produce no more useful answers – a condition captured as follows.

Definition 4 (Completely Evaluated). *A set $\mathcal{S}$ of subgoals in a forest $\mathcal{F}$ is* completely evaluated *if no $S \in \mathcal{S}$ is completed; and if for each $S \in \mathcal{S}$, for each node N in the tree for S:*

1. *The underlying subgoal of the selected literal of N is completed; or*
2. *There are no applicable* New Subgoal, Program Clause Resolution, Answer Return, Negation Return *or* Delaying *operations (Definition 6) for N.*

In order to prevent $\mathcal{S}$ from being repeatedly completed, the preceding definition explicitly prohibits $\mathcal{S}$ from containing any completed subgoals.

SLG forests are related to interpretations in the following manner.

Definition 5. *Let $\mathcal{F}$ be a forest. The* interpretation induced by $\mathcal{F}$, *$I_{\mathcal{F}}$, is the smallest set such that:*

- *A (ground) atom $A \in I_{\mathcal{F}}$ iff A is in the ground instantiation of some unconditional answer Ans :- | in $\mathcal{F}$.*
- *A (ground) literal not $A \in I_{\mathcal{F}}$ iff A is in the ground instantiation of a completely evaluated subgoal in $\mathcal{F}$, and A is not in the ground instantiation of any answer in $\mathcal{F}$.*

An atom S is successful in $\mathcal{F}$ *if some tree in $\mathcal{F}$ has an unconditional answer S. S is* failed in $\mathcal{F}$ *if S is completed and the tree for S contains no answers. An atom S is* successful *(*failed*) in $I_{\mathcal{F}}$ if S' (not S') is in $I_{\mathcal{F}}$ for every S' in the ground instantiation of S. A negative delay literal not D is successful (failed) in a forest $\mathcal{F}$ if D is failed (successful) in $\mathcal{F}$. Similarly, a positive delay literal D_{Ans}^{Subg} is successful in $\mathcal{F}$ if $Subg$ has an unconditional answer Ans :- | and failed if $Subg$ has no answer with head Ans.*

Given these concepts, the SLG operations themselves can be stated.

Definition 6 (SLG Operations). *Given a forest $\mathcal{F}_n$ of a SLG evaluation of program P, $\mathcal{F}_{n+1}$ may be produced by one of the following operations.*

1. New Subgoal*: Let $\mathcal{F}_n$ contain a non-root node*
$$N = Ans \text{ :- } DelaySet | G, GoalList$$
where G is the selected literal S or not S. Assume $\mathcal{F}_n$ contains no tree with root subgoal S. Then add the tree S :- $|S$ to $\mathcal{F}_n$.
2. Program Clause Resolution*: Let $\mathcal{F}_n$ contain a root node $N = S$:- $|S$ and C be a program clause $Head$:- $Body$ such that $Head$ unifies with S with mgu θ. Assume that in $\mathcal{F}_n$, N does not have a child $N_{child} = (S$:- $|Body)\theta$. Then add N_{child} as a child of N.*
3. Answer Return*: Let $\mathcal{F}_n$ contain a non-root node*
$$N = Ans \text{ :- } DelaySet | S, GoalList$$
whose selected literal S is positive. Let Ans be an answer node for S in $\mathcal{F}_n$ and N_{child} be the SLG resolvent of N and Ans on S. Assume that in $\mathcal{F}_n$, N does not have a child N_{child}. Then add N_{child} as a child of N.

4. NEGATION RETURN: *Let $\mathcal{F}_n$ contain a leaf node*
$$N = Ans \text{ :- } DelaySet|not\ S, GoalList.$$
whose selected literal not S is ground.
   (a) NEGATION SUCCESS: *If S is failed in $\mathcal{F}_n$, then create a child for N of the form: Ans :- $DelaySet|GoalList$.*
   (b) NEGATION FAILURE: *If S succeeds in $\mathcal{F}_n$, then create a child for N of the form* fail.
5. DELAYING: *Let $\mathcal{F}_n$ contain a leaf node $N = Ans$:- $DelaySet|not\ S, GoalList$, such that the selected literal npt S is ground, S is in $\mathcal{F}_n$, but S is neither successful nor failed in $\mathcal{F}_n$. Then create a child for N of the form Ans :- $DelaySet, not\ S|GoalList$.*
6. SIMPLIFICATION: *Let $\mathcal{F}_n$ contain a leaf node $N = Ans$:- $DelaySet|$, and let $L \in DelaySet$*
   (a) *If L is failed in $\mathcal{F}$ then create a child* fail *for N.*
   (b) *If L is successful in $\mathcal{F}$, then create a child Ans :- $DelaySet'|$ for N, where $DelaySet' = DelaySet - L$.*
7. COMPLETION: *Given a completely evaluated set $\mathcal{S}$ of subgoals (Definition 4), for each $S \in \mathcal{S}, setMark(T, complete)$, where T is the tree for S.*
8. ANSWER COMPLETION: *Given a set of unsupported answers $\mathcal{UA}$, create a failure node as a child for each answer $Ans \in \mathcal{UA}$.*

In the above definition, the ANSWER COMPLETION operation relies on the concept of *unsupported answers*. Unsupported answers are conditional answers that are false in the well-founded model, and reflect certain unfounded sets in that model. While necessary for completeness of SLG, the ANSWER COMPLETION operation is not affected by local or concurrent evaluations, so for reasons of space, we omit its formal definition (see the full version of this paper).

SLG Evaluations An SLG evaluation consists of a (possibly transfinite) sequence of forests. However as noted, we restrict definitions of evaluations to be finite for reasons of space (see the full version of this paper for the general case).

Definition 7 (SLG Evaluation). *Given a program P and goal G, an SLG evaluation $\mathcal{E}$ is a sequence of SLG forests $\mathcal{F}_0$,$\mathcal{F}_1$,...,$\mathcal{F}_\beta$, such that:*

- *$\mathcal{F}_0$ is the forest containing a single tree* G :- | G
- *For each successor ordinal, $n + 1 \leq \beta$, $\mathcal{F}_{n+1}$ is obtained from $\mathcal{F}_n$ by an application of an SLG operation from Definition 6.*

If no operation is applicable to $\mathcal{F}_\beta$, $\mathcal{F}_\beta$ is called a final forest *of $\mathcal{E}$. If $\mathcal{F}_\beta$ contains a leaf node with a non-ground selected negative literal, it is* floundered.

The correctness is formulated as follows for transfinite evaluations

Theorem 1 ([1]). *Let $\mathcal{E}$ be an SLG evaluation of a goal to a program P. Then $\mathcal{E}$ has a final forest $\mathcal{F}$. Let A be an atom such that A :- $|A$ is the root of some tree in $\mathcal{F}$. Then if $\mathcal{F}$ is non-floundered, $WFM(P)|_A = I_{\mathcal{F}}|_A$.*

2 Local SLG Evaluations

As noted above, a Local SLG evaluation fully evaluates each mutually dependent set of tabled subgoals before performing operations to subgoals outside of that set. We begin to formalize that notion by defining what it means for one subgoal to depend on another.

Definition 8 (Subgoal Dependency Graph). *Let $\mathcal{F}$ be a forest, and let S_1 :- $|S_1$ be the root of a non-completed tree in $\mathcal{F}$. The subgoal S_1* directly depends on *a subgoal S_2 iff S_2 is not completed in $\mathcal{F}$, and there is some node N in the tree for S_1 such that S_2 is the underlying subgoal of the selected literal of N.*

The Subgoal Dependency Graph of $\mathcal{F}$, SDG($\mathcal{F}$) = (V,E)*, is a directed graph in which $(S_i, S_j) \in E$ iff subgoal S_i directly depends on subgoal S_j, and V is the underlying set of E. S_1* "depends on" *S_2 in $\mathcal{F}$ is there is a path from S_1 to S_2 in $SDG(\mathcal{F})$.*

Since the SDG of a forest is a directed graph, it can be partitioned into disjoint sets of strongly connected components, or SCCs, where a node with no outgoing edges is considered to be in a *trivial* SCC. We refer to a given SCC by the set of its vertices (subgoals), and distinguish *independent* SCCs.

Definition 9 (Independent SCC). *A strongly connected component $\mathcal{S}$ is* independent *if $\forall S \in \mathcal{S}$: if S depends on some S', then $S' \in \mathcal{S}$.*

By Definition 9 it is straightforward that a trivial SCC is independent, and that each independent component is *maximal* — i.e. contained in no larger SCC. Local evaluation, then, performs operations on independent SCCs. Formally:

Definition 10 (Local SLG Evaluation). *Given a program P and goal G, a Local SLG evaluation $\mathcal{E}$ is a sequence of SLG forests $\mathcal{F}_0$,$\mathcal{F}_1$,...,$\mathcal{F}_\beta$, such that:*

1. *$\mathcal{F}_0$ is the forest containing a single tree* G :- | G
2. *For each successor ordinal, $n + 1 \leq \beta$, $\mathcal{F}_{n+1}$ is obtained from $\mathcal{F}_n$ by an application of an SLG operation from Definition 6 such that:*
   (a) *if a* New Subgoal *operation is applied to create a tree S :- $|S$ then S is the underlying subgoal of a selected literal in a tree whose root subgoal is in an independent SCC of $SDG(\mathcal{F}_n)$;*
   (b) *a* Program Clause Resolution, Answer Return, Negation Return *or* Delaying *operation is only applied to a node on a tree whose root subgoal is in an independent SCC of $SDG(\mathcal{F}_n)$;*

$\mathcal{E}$ is delay avoiding *if no* Delaying *operation is performed in a forest if any other operation is applicable.*

In the transfinite extension of Definition 10, a Local (SLG) evaluation works as an unrestricted SLG evaluation whenever an independent SCC does not exist in a forest, leading to the following theorem.

Theorem 2 (Completeness of Local Evaluation). *Let P be a program and G a goal. Then there exists an SLG evaluation $\mathcal{E}$ of G against P with final forest $\mathcal{F}$ if and only if there exists a local SLG evaluation $\mathcal{E}^L$ of G against P with final forest $\mathcal{F}^L$ such that $I_{\mathcal{F}}|_G = I_{\mathcal{F}^L}|_G$.*

While Local evaluation is ideally complete for the well-founded semantics, its importance arises from its efficiency for certain classes of programs, along with properties that can be used to ensure the correctness of implementations. The first such property is:

Theorem 3. *Let $\mathcal{E}^L$ be a finite Local SLG evaluation. For each $\mathcal{F}$ in $\mathcal{E}^L$ $SDG(\mathcal{F})$ has one and only one independent SCC.*

Theorem 3 implies the following corollary which will be used by the implementation described in Section 4.

Corollary 1. *Let $\mathcal{E}^L$ be a finite Local SLG evaluation. For each $\mathcal{F}$ in $\mathcal{E}^L$ there is at most one incoming edge for each maximal SCC in $SDG(\mathcal{F})$.*

The following corollary captures the notion that in a Local evaluation, a subgoal may only return answers out of its SCC once its SCC has been completed.

Corollary 2. *In any forest $\mathcal{F}$ of a Local SLG evaluation, if an answer A is used in an* Answer Return *operation to a node in a tree with root subgoal S, then the tree for A has been completed, or is in the same SCC as S in $SDG(\mathcal{F})$.*

Corollary 2 has practical importance for answer subsumption since it implies that no answer A need be returned out of an SCC if the model entails an answer that is preferred to A – only the preferred answer need be returned. In addition, it is easy to see that if Local evaluation were extended to ensure that all appropriate Simplification and Answer Completion operations are performed for an independent SCC just after it has been completed, the following statement also holds. If a forest in Local evaluation contains a conditional answer $A = S$:- $D|$ and S is successful or failed in $\mathcal{F}$, A will never be propagated outside of the SCC. This strategy reduces the overall number of Simplification and Answer Completion operations and has been adopted by the XSB engine when computing non-stratified programs. The space efficiency of Local evaluation is stated as follows:

Theorem 4. *Let $\mathcal{E}^L$ be a finite delay-avoiding Local evaluation of a goal G to a program P, and let $\mathcal{E}$ be an SLG evaluation of G to a P. Then for any forest $\mathcal{F}^L$ in $\mathcal{E}^L$, there exists a forest $\mathcal{F}$ in $\mathcal{E}$ such that $SDG(\mathcal{F}^L)$ is a subgraph of $SDG(\mathcal{F})$.*

3 Sharing Completed Tables in a Concurrent Evaluation

Rather than starting with a single top-level atomic query, a concurrent SLG, SLG_C, evaluation is initialized with a set of atomic queries, such that each atomic query is evaluated by a different thread of computation. In this model of concurrency, threads share only *completed* tables so that a thread is prevented from consuming answers from a (non-completed) table owned by another thread. This disallows consume-producer models of concurrency and implies that different threads may not collaborate to evaluate subgoals within a single SCC.

However as discussed below, within a Local evaluation the restriction may not be binding since Local evaluations prevent consumer-producer models by their nature, and since the scope of an SCC in a Local evaluation is relatively small.

Formally, this model of concurrency marks every non-completed tree in a given forest with a *thread identifier* (cf. Definition 2). As terminology, if N is a node in a tree T, $marking(N)$ denotes $marking(T)$, and if S is a subgoal $marking(S)$ denotes $marking(T)$, where T is the tree for S.

Definition 11 (Thread). *A* thread identifier *is an element of a set of terms that does not include the term* complete*. Given an SLG forest $\mathcal{F}$ in an evaluation $\mathcal{E}$, a* thread state *is the maximal set $\mathcal{T}$ of trees in $\mathcal{F}$ such that for all $T \in \mathcal{T}$ $marking(T) = t$ where t is a thread identifier. A* thread *in $\mathcal{E}$ is the sequence of thread states for a given marking. A thread is* active *in $\mathcal{F}$ if its thread state in $\mathcal{F}$ is non-empty.*

Let S be a subgoal, T the tree for S, and N a node in a forest. N is thread compatible *with S if $marking(T) =$* complete *or $marking(T) = marking(N)$.*

SLG_C uses SLG forests and other notions from Definitions 1-5, but differs in that certain SLG_C operations may create or change thread markings, and markings may restrict the applicability of operations based on whether a node and subgoal are thread compatible according to the previous definition. Definition 12 presents a new operation called Usurpation, along with those operations that differ from Definition 6 where the difference in each altered operation is underlined.

Definition 12 (SLG_C Operations). *Given an SLG forest $\mathcal{F}_n$, $\mathcal{F}_{n+1}$ may be produced by one of the following operations.*

1. New Subgoal*: Let $\mathcal{F}_n$ contain a non-root node*
$$N = Ans \text{ :- } DelaySet|G, GoalList$$
where G is the selected literal S or not S. Assume $\mathcal{F}_n$ contains no tree with root subgoal S. Then add the tree $T = S$:- $|S$ to $\mathcal{F}_n$, and $setMark(T, marking(N))$.
2. Answer Return*: Let $\mathcal{F}_n$ contain a non-root node*
$$N = Ans \text{ :- } DelaySet|S, GoalList$$
whose selected literal S is positive. Let Ans be an answer node for S in $\mathcal{F}_n$ such that N is thread compatible with S and let N_{child} be the SLG resolvent of N and Ans on S. Assume that in $\mathcal{F}_n$, N does not have a child N_{child}. Then add N_{child} as a child of N.
3. Negation Return*: Let $\mathcal{F}_n$ contain a leaf node*
$$N = Ans \text{ :- } DelaySet|not\ S, GoalList.$$
whose selected literal not S is ground where N is thread compatible with S.
   (a) Negation Success*: If S is failed in $\mathcal{F}_n$, then create a child for N of the form: Ans :- $DelaySet|GoalList$.*
   (b) Negation Failure*: If S succeeds in $\mathcal{F}_n$, then create a child for N of the form* fail.

4. COMPLETION*: Given a completely evaluated set $\mathcal{S}$ of subgoals such that for all $S \in \mathcal{S}, marking(S) = t$, then for each $S \in \mathcal{S}, setMark(T, complete)$, where T is the tree for S.*
5. USURPATION*: Let $\mathcal{S}$ be a set of subgoals in deadlock (Definition 13), $S_U \in \mathcal{S}$, and T_U the tree for S_U. For each $S \in \mathcal{S}, setMark(T, marking(T_U))$.*

The thread compatibility restrictions can mean that an SLG operation is applicable in a given forest, but that the corresponding SLG$_C$ operation is not. The USURPATION operation is designed to address cases where SLG$_C$ operations might get stuck – which are formalized as situations of deadlock.

Definition 13 (Deadlock). *A set $\mathcal{S}$ of subgoals in a forest $\mathcal{F}$ is in* deadlock *if:*

1. *For each $S \in \mathcal{S}$ there are no applicable* NEW SUBGOAL*,* PROGRAM CLAUSE RESOLUTION*,* ANSWER RETURN*,* NEGATION RETURN *or* DELAYING *operations of Definition 12; and*
2. *There exists no $\mathcal{S}'$ such that $\mathcal{S} \subseteq \mathcal{S}'$ and $\mathcal{S}'$ is completely evaluated in $\mathcal{F}$.*

Example 1. As defined, SLG$_C$ evaluations may use any scheduling strategy, and are not restricted to Local evaluations. They also begin with a *set* of goals rather than with a single goal. Figure 1 illustrates a simple, non-Local, SLG$_C$ evaluation of the goal {`a(X),b(X)`} to the program P_2, where `a(X)` is initially marked with thread identifier 1 and `b(X)` with thread identifier 2. Through NEW SUBGOAL operations, trees for `c(X)` and `e(X)` are created and associated with thread identifier 1, while `d(X)` is created and associated with thread identifier 2. Evaluation continues until there is a deadlock, as shown in Figure 1b. Note in Figure 1b, that while there is an answer that could be returned to the node `e(1):- |d(X)` in a non-Local evaluation, the node is associated with thread identifier 1, while the answer is associated with thread identifier 2 so that the return is prohibited by the thread compatibility restrictions. USURPATION is the only operation applicable to this forest; assume that thread identifier 1 performs the USURPATION, marking trees for `c(X)`, `d(X)`, and `e(X)` with identifier 1. Afterward, an answer for `e(1)` is derived, leading to Figure 1c. Further ANSWER RETURN operations lead to Figure 1d.'v All of the subgals in thread identifier 1 have been completely evaluated, but the subgoal `b(X)` in thread identifier 2 cannot be completely evaluated until the answer for `d(X)` is resolved with the node `b(X):- |d(X)`. Since a completed subgoal is thread compatible with any thread, once `d(X)` is completed, the answer for `d(X)` can be resolved.

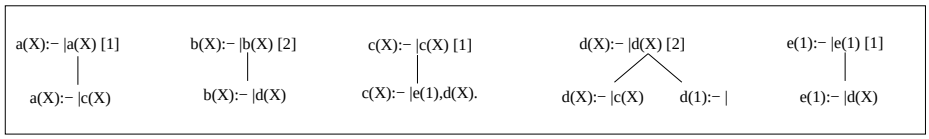
The definition of a SLG$_C$ evaluation is nearly the same as for SLG (Definition 7), but is initialized so that each atomic query in the set of goals it is presented with is marked with a different thread identifier (Its formal, transfinite, definition can be found in the full version of this paper). In addition, SLG$_C$ forests are based on Definition 2, so the definition of an interpretation induced by a forest is identical in both frameworks, leading to the following theorem.

Theorem 5 (Correctness of SLG$_C$). *Let P be a program and $\mathcal{G}$ a finite non-empty set of goals. Then a SLG$_C$ evaluation of $\mathcal{G}$ against P exists with final state*

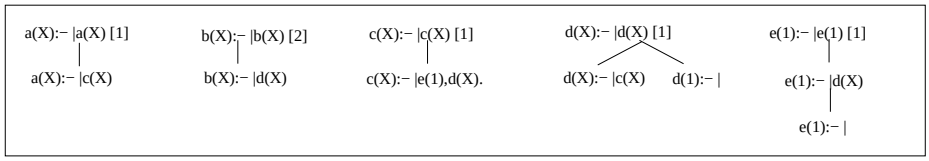
```
a(X):- c(X).  b(X):- d(X).  c(X):- e(1),d(X).  d(X):- c(X).
                                                d(1).

e(1):- d(X)
```

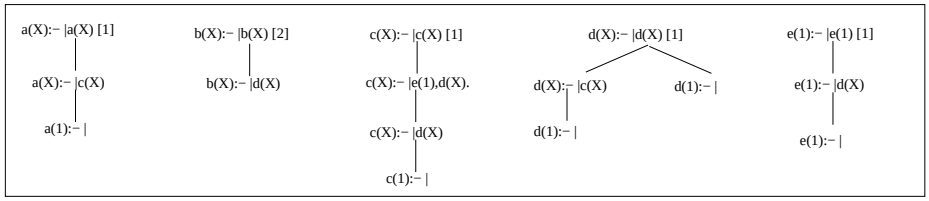
(a) The Program P_2



(b) State α: Deadlock



(c) State β: Answer for `e(1)`



(d) State γ: Complete Evaluation for Thread Identifier 1

Fig. 1. A non-Local SLG_C Evaluation of P_2

$\widehat{\mathcal{F}}$, *iff for every* $G_i \in \mathcal{G}$ *there exists an SLG evaluation of* G_i *against* P *with final state* $\mathcal{F}^i$ *and* $I_{\widehat{\mathcal{F}}} = (\bigcup I_{\mathcal{F}^i})$.

The completeness portion of the theorem follows from a demonstration that for any SLG operation on a forest, an equivalent SLG_C operation is applicable after zero or more USURPATION operations. The following theorem bounds the number of USURPATION operations in a finite evaluation, which implies that the abstract complexity of SLG_C is the same as that of SLG.

Theorem 6 (Complexity of USURPATION**).** *Let* $\mathcal{E}$ *be a finite* SLG_C *evaluation with final forest* $\mathcal{F}$*, and* $S_{\mathcal{F}}$ *the set of all subgoals in* $\mathcal{F}$*. Then there are at most* $|\mathcal{S}_{\mathcal{F}}|$ USURPATION *operations performed.*

3.1 Concurrent Local Evaluations

In SLG_C the Subgoal Dependency Graph (Definition 8) can be partitioned into disjoint sub-graphs for each thread state of a forest.

Definition 14 (Thread Subgoal Dependency Graph). *For each thread state t in a forest $\mathcal{F}$, the* Thread Subgoal Dependency Graph *of t ($Thread\_SDG(\mathcal{F}, t)$) consists of the sub-graph of $SDG(\mathcal{F})$ determined by subgoals in $\mathcal{F}$ whose marking is t.*

Local SLG$_C$ evaluation is based on independent SCCs within Thread SDGs, rather than within a global SDG.

Definition 15 (Local SLG$_C$). *Given a program P, a set $\mathcal{T}$ of thread identifiers, and a finite non-empty set G of goals, a Local SLG$_C$ evaluation $\mathcal{E}$ is a sequence of forests $\mathcal{F}_0$,$\mathcal{F}_1$,...,$\mathcal{F}_\beta$, such that:*

1. *$\mathcal{F}_0$ is a set-minimal forest containing the trees $T_i = G_i$:- $|G_i$, for each $G_i \in \mathcal{G}$, where for each T_i there is a $t_i \in \mathcal{T}$ such that $marking(T_i) = t_i$, and $t_i \neq t_j$ if $i \neq j$.*
2. *For each successor ordinal, $n + 1 \leq \beta$, $\mathcal{F}_{n+1}$ is obtained from $\mathcal{F}_n$ by an application of an operation from Definition 12 such that:*
   (a) *if a* New Subgoal *is applied to create a tree $T = S$:- $|S$ then S is the underlying subgoal of a selected literal in a tree whose root subgoal is in an independent SCC of $Thread\_SDG(\mathcal{F}_n, marking(T))$;*
   (b) *a* Program Clause Resolution, Answer Return, Negation Return *or* Delaying *operation is only applied to a node on a tree whose root subgoal is in an independent SCC of $Thread\_SDG(\mathcal{F}_n, marking(T))$.*

This finitary definition can be extended to the transfinite evaluations, leading to the following theorem.

Theorem 7 (Correctness of Local SLG$_C$). *Let P be a program and $\mathcal{G}$ a finite non-empty set of goals. Then a Local SLG$_C$ evaluation of $\mathcal{G}$ against P exists with final state $\widehat{\mathcal{F}}$, iff every $G_i \in \mathcal{G}$ there exists an SLG evaluation of G_i against P with final state $\mathcal{F}^i$ and $I_{\widehat{\mathcal{F}}} = (\bigcup I_{\mathcal{F}^i})$.*

The following theorem is an analogue of Theorem 3, and implies that each thread of an Local SLG$_C$ evaluation has the dependency properties of Section 2.

Theorem 8. *Let $\mathcal{F}$ be a forest in a finite Local SLG$_C$ evaluation. Then for each active thread t in $\mathcal{F}$, $Thread\_SDG(\mathcal{F}, t)$ has one and only one independent SCC.*

The Thread Dependency Graph can be seen as a homomorphism of the SDG of a given SLG$_C$ forest.

Definition 16 (Thread Dependency Graph). *Let t_1 and t_2 be two active threads in a SLG forest $\mathcal{F}$. t_1* directly depends *on t_2 if there exist a subgoal in t_1 that directly depends on a subgoal in t_2 (according to Definition 8). The* Thread Dependency Graph TDG($\mathcal{F}$) = (V,E) *of $\mathcal{F}$ is a directed graph where V is the set of active threads in $\mathcal{F}$ and $(t_i, t_j) \in E$ iff t_i directly depends on t_j.*

Based on the thread dependency graph, the following theorem shows that any thread depends on at most one other thread.

Theorem 9. *Let $\mathcal{F}$ be a forest in a finite Local SLG$_C$ evaluation. Then for each node in $TDG(\mathcal{F})$ there is at most one outgoing edge.*

As a practical matter, this theorem indicates that each thread of computation will wait on the results from at most one other thread. so that the thread communication and dependency detection required to implement the USURPATION operation will be relatively simple.

4 Implementing SLG$_C$ in the SLG-WAM

We summarize the changes made to XSB's SLG-WAM in order to implement Local SLG$_C$. Our discussion omits numerous optimizations required for efficiency. In particular, due to space restrictions we do not discuss the propagation of subgoal dependencies between threads, or the handling of subgoals that have been usurped multiple times (see [8] for details). We first describe Local SLG$_C$ for definite programs before considering negation.

Since XSB's SLG-WAM implements Local evaluation, it is evident from Section 3 that the main addition is the USURPATION operation, which mainly affects the SLG-WAM tabletry instruction. This instruction occurs at the entry point of a tabled predicate when a tabled subgoal $Subg$ is called. In the sequential

```
Instruction tabletry
    /* Subg is in argument registers; T_current is current thread */
    Perform the subgoal_check_insert(Subg) operation in the table for this predicate
    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.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

deadlock(T_current,depends_thread )
    while( depends_thread ≠ NULL )
       if( depends_thread = T_current ) return true;
       else depends_thread ← depends_thread.suspended_on_thread);
    return false;

usurp(T_current, first_usurped)
    Traverse SCC_dl to reset suspended_on_thread dependency of each usurped thread
    Unlock global TDG mutex
    Traverse SCC_dl to
       Propagate the proper subgoal dependency to each usupred thread
       Reset stacks of each (suspended) usurped thread
```

Fig. 2. Summary of Concurrent Local SLG implementation in the SLG-WAM.

SLG-WAM `tabletry` is essentially responsible for determining whether a NEW SUBGOAL operation is required. The instruction first determines whether *Subg* is in the table using its representation in the WAM argument registers. If *Subg* is not in the table, a NEW SUBGOAL operation is effectively performed. *Subg* will have been copied to the table during the check; and a *generator* choice point is created to backtrack through program clauses, to check whether the subgoal's SCC has been fully evaluated, and to schedule ANSWER RETURN operations if the SCC is not fully evaluated. On the other hand, if *Subg* is in the table, `tabletry` creates a *consumer* choice point to backtrack through any answers to *Subg* in the table and thereby perform ANSWER RETURN operations.

Extensions to tabletry for Local SLG$_C$ are summarized in Figure 2. If *Subg* is new, it is copied into the table as in the sequential case, but in order to represent the TDG a thread identifier is associated with *Subg*. For this association the *subgoal frame*, a structure containing information about each tabled subgoal, is extended with a *ThreadMark* cell. The essential difference from the sequential case of `tabletry` occurs when *Subg* is not new and is currently marked by another thread (and therefore not marked as completed). In this case deadlock detection is performed: if a deadlock is not found, the calling thread $T_{current}$ suspends as it does not have any applicable Local SLG$_C$ operations; otherwise $T_{current}$ performs a USURPATION operation. In addition to changes in tabletry, a change is made to the SLG-WAM completion instruction so that any thread suspended on *Subg* is awakened when *Subg* is completed (a condition variable based on the predicate symbol of *Subg* is used for this awakening).

The design of deadlock detection in the SLG-WAM relies on Theorem 9, which states that each thread may be suspended on at most one other thread. The SLG-WAM adds a *suspended_on_thread* field to the context of each thread to denote any thread dependency. As shown in Figure 2, when a thread $T_{current}$ performs deadlock detection, it starts by checking whether the thread marking *Subg* is suspended using this *suspended_on_thread* field: if the thread is not suspended, $T_{current}$ may suspend without fear of deadlock and it will be awakened when *Subg* is completed. If the marker of *Subg* is suspended, the deadlock detection code follows the *suspended_on_thread* field. By Theorem 9, any loop in the TDG must be a simple cycle so that deadlock detection is a simple while loop that exits in one of two cases. If $T_{current}$ is found in the *suspended_on_thread* field for one of the traversed threads, then $T_{current}$ depends (transitively) on itself and deadlock occurs; otherwise if the *suspended_on_thread* field of a traversed thread is null, $T_{current}$ transitively depends on a subgoal that is actively being computed.

The fact that the thread dependencies for deadlocked threads form a simple cycle also underlies the control flow of the usurp() function which consists of two traversals of the deadlocked TDG cycle, denoted SCC_{dl}. Each traversal begins with the thread that marks *Subg*. In the first traversal, the usurping thread $T_{current}$ updates the TDG, setting the *suspended_on_thread* field of each usurped thread to its own id. Adjusting the TDG must be performed under global mutual exclusion: otherwise two concurrently usurping threads might produce an incoherent TDG. In the second traversal, which is not under mutual exclusion,

the execution stacks in each usurped thread $T_{usurped}$ are examined and manipulated – an operation that is safe since $T_{usurped}$ has suspended on a subgoal due to thread compatibility restrictions. The manipulation ensures that $T_{usurped}$ will no longer generate answers for usurped subgoals that it has marked, but rather will be set to consume answers. This stack manipulation is considerably simplified by the property that the SDG for $T_{usurped}$ will depend on a *single* usurped subgoal $S_{T_{usurped}}$ – the first subgoal in SCC_{dl} that $T_{usurped}$ encountered. (the property is implied by Corollary 1 together with Theorem 8). However to determine $S_{T_{usurped}}$, both subgoal dependencies contained in $T_{usurped}$ and subgoal dependencies across usurped threads must be considered. Accordingly, usurp() also propagates cross-thread dependencies (the actual mechanism is not shown in this summary) and uses these dependencies when resetting the stacks of $T_{usurped}$. As a result, when $T_{usurped}$ is awakened it will call $S_{T_{usurped}}$ again from scratch and backtrack through the answers of the completed subgoal.

This approach has the virtue of conceptual simplicity, but any partial computations for the usurped subgoals are lost, and will be recomputed by the usurping thread. Theorem 6 states that the maximal number of USURPATION operations in a SLG$_C$ evaluation is linear in $atoms(P)$, the number of atoms in a program P. In [8] it is shown that USURPATION operations affect only constant-time operations so that even if answers for usurped subgoals are recomputed, the complexity of the well-founded semantics is unaffected.

Extensions for Negation and Answer Subsumption As suggested by the changed operations in Definition 12, the SLG-WAM requires few modifications beyond those presented to extend Local SLG$_C$ to the well-founded semantics. Consider first stratified programs. In the SLG-WAM, if the underlying (tabled) subgoal $Subg$ of a selected negative literal is not new and not complete, the computation path "suspends" and resumes only when $Subg$ has been completed. These operations are essentially the same as the interactions between threads so far described. In the case of non-stratified negation the first new operation to consider is the DELAYING operation. If $Subg$ is involved in a loop through negation, the resumption mechanism is the same except that a bit in the subgoal frame of $Subg$ is set to indicate that $Subg$ was delayed rather than completed. Several cycles of delaying may be needed before $Subg$ is finally completed, but each cycle may be handled by the thread suspension and usurpation mechanisms described. When $Subg$ is completed, any SIMPLIFICATION operations for its SCC are performed before awakening any threads suspended on $Subg$, so that SIMPLIFICATION is not affected by the concurrency mechanisms. Beyond negation, answer subsumption is implemented as an extension to the SLG-WAM new_answer operation which is unaffected by Local SLG$_C$.

Performance Several performance studies have been made on tabling with Local SLG$_C$ [6, 8, 7]. We focus on tests of scalability in which a list of M queries is distributed to N threads and the elapsed time measured. [7] measured the use of Local SLG$_C$ on programs which analyzed configuration reachability for various extensions of Petri nets. Depending on particular formalism for the net, the programs were definite, or used well-founded negation, tabled constraints or answer subsumption. For nearly all of these benchmarks, left-recursive reacha-

bility of the form `reachable(bound,free)`) scaled perfectly to 4 processors (the number available for this experiment).

In constrast, [8] measured scalability on a worst case: where multiple threads concurrently evaluated right recursion on random graphs of varying densities, using queries of the form `rightRec(bound,free)`. Observe that for right-recursion over a graph, the connectivity of the SDG directly reflects the connectivity of the graph. Consider properties of a random graph of V vertices (cf. [11]). If each vertex has at most 1 edge there can be no cycles; if each vertex has between 1 and $\ln(V)$ edges the graph (and SDG) is likely to be split up into several SCCs; while if each vertex has $\ln(V)$ or more edges the graph is likely to be connected, with the SDG consisting of a single SCC. While somewhat preliminary, [8] indicates that the number of deadlocks are relatively small. For graphs with between 1 and $\ln(V)$ edges per vertex, this is either because the graphs do not contain large SCCs or because the subgoals in these SCCs are quickly completed. For fully connected graphs, each thread is usurped at most once. As expected, scalability is poor for the connected graphs as usurped threads must wait for the SCC to be evaluated. However, the elapsed time for the Local SLG$_C$ is never worse than that for a Local single-threaded evaluation on any graph. In other words, for these benchmarks the implementation of Local SLG$_C$ is not affected by the cost of recomputing answers for usurped subgoals and degenerates into a mostly sequential evaluation where threads wait for the completion of SCCs.

5 Discussion

Local SLG$_C$ is well suited for multi-threaded evaluations that benefit from Local evaluation and can provide speedups on problems that can be subdivided relatively easily. At the same time, Local SLG$_C$ is not intended to support general table parallelism. Local evaluation itself prevents one thread from consuming answers concurrently produced by another thread if the consuming and producing subgoals are in different SCCs. Beyond this, a Local SLG$_C$ evaluation may have a number of threads suspended on incomplete or usurped subgoals, although Theorem 6 puts a limit on the number of USURPATION operations.

We believe that a salient strength of Local SLG$_C$ is its formal basis. By Theorem 7 several threads can cooperate to correctly compute the well-founded semantics, and by Theorem 6 the abstract complexity is the same as a sequential SLG evaluation. By Theorem 8 each thread in a Local SLG$_C$ evaluation will have a single independent SCC, and so each thread will have properties of a Local evaluation, including the space efficiency property of Theorem 4. By Corollary 2 (and Theorem 8) each thread will only return answers from completed tables, a useful property for computing the well-founded semantics and answer subsumption. As noted in Section 4, the implementation of Local SLG$_C$ directly relies on Theorem 9 and Corollary 1. As a result of the theory-oriented design, the implementation of Local SLG$_C$, although delicate, mainly requires about 300 lines of code to be added to the `tabletry` instruction: thus Local SLG$_C$ should be relatively easy to port to other tabling engines that implement Local evaluation.

Related Work These strengths and limitations distinguish (Local) SLG$_C$ from previous work, which we briefly discuss (see [6] for more details). [5] presents

an approach to distributed tabling in which the SDG (Definition 8) is distributed among threads, the dependencies partially represented by numerical encodings associated with subgoals, and a message-counting algorithm used for termination detection. Maintaining the distributed SDG leads to an approach that is cubic in the number of messages. SLG$_C$ differs from [5] in being a more minimal extension of SLG requiring only the addition of markings and USURPATION, and in retaining the complexity of SLG. Another distributed tabling method, [2] avoids the cubic message complexity by using a centralized table manager to maintain dependency and other information and a credit-recovery algorithm to detect completion of the SCCs. SLG$_C$ differs from [2] in not requiring an explicit table manager and in using the "optimistic" USURPATION operation to control concurrency, as well as in being a formalism sufficient for proving completeness and other properties. [9] presents algorithms for adding tabling to an or-parallel engine and implements these algorithms in YAP, with impressive results for definite programs. As mentioned above, unlike [9] Local SLG$_C$ does not address general table parallelism, although it addresses normal programs and is based on a formalization which permits a concise implementation. Perhaps the closest work is [3] which allows threads to share answers when tables are not completed: Concurrent SLG differs from this work in using a simpler method of concurrency control, as well as in modeling normal rather than definite programs.

Acknowledgements: The authors thank Manuel Carro, Pablo Chico de Guzmán, and anonymous reviewers for their careful comments.

References

1. W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.
2. C. Damásio. A distributed tabling system. In *Proceedings of the 2nd Workshop on Tabulation in Parsing and Deduction, TAPD'2000*, pages 65–75, 2000.
3. J. Freire, R. Hu, T. Swift, and D. S. Warren. Parallelizing tabled evaluation. In *7th International PLILP Symposium*, pages 115–132. Springer-Verlag, 1995.
4. J. Freire, T. Swift, and D. S. Warren. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. *JFLP*, 1998(3), 1998.
5. R. Hu. *Distributed Tabled Evaluation*. PhD thesis, SUNY at Stony Brook, 1997.
6. R. Marques. *Concurrent Tabling: Algorithms and Implementation*. PhD thesis, Universidade Nova de Lisboa, 2007.
7. R. Marques, T. Swift, and J. Cunha. Extending tabled logic programming with multi-threading: A systems perspective. 2008.
8. R. Marques, T. Swift, and J. Cunha. A simple and efficient implementation of concurrent local tabling. Available at `http://www.cs.sunysb.edu/~tswift`, 2008.
9. R. Rocha, F. Silva, and V. S. Costa. On applying or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming*, 4(6), 2004.
10. R. Rocha, F. Silva, and V. Santos Costa. Dynamic mixed-strategy evaluation of tabled logic programs. In *ICLP*, page 250264, 2005.
11. J. Spencer. *The Strange Logic of Random Graphs*. Springer, 2000.
12. A. van Gelder, K.A. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.