Supporting Pruning in Tabled LP⋆

Pablo Chico de Guzmán[1], Manuel Carro[2,3], and Manuel V. Hermenegildo[2,3]

[1] ElasticBox, Inc.
[2] IMDEA Software Institute, Spain
[3] School of Computer Science, Univ. Politécnica de Madrid, Spain
pchico@clip.dia.fi.upm.es, {mcarro,herme}@fi.upm.es

Abstract. This paper analyzes issues which appear when supporting pruning operators in tabled LP. A version of the once/1 control predicate tailored for tabled predicates is presented, and an implementation analyzed and evaluated. Using once/1 with answer-on-demand strategies makes it possible to avoid computing unneeded solutions for problems which can benefit from tabled LP but in which only a single solution is needed, such as model checking and planning. The proposed version of once/1 is also directly applicable to the efficient implementation of other optimizations, such as early completion, cut-fail loops (to, e.g., prune at the toplevel), if-then-else, and constraint-based branch-and-bound optimization. Although once/1 still presents open issues such as dependencies of tabled solutions on program history, our experimental evaluation confirms that it provides an arbitrarily large efficiency improvement in several application areas.

Keywords: Logic Programming, Tabling, Pruning, Performance.

1 Introduction

Tabled LP [1] overcomes several limitations of the SLD resolution strategy. In particular, it guarantees termination for programs with the *bounded term-size* property and can improve efficiency in programs which repeatedly perform some computation. These characteristics help make logic programs less dependent on clause and goal order, thereby bringing operational and declarative semantics closer together. Tabled LP has been successfully applied in many areas including deductive databases, program analysis, or semantic Web reasoning.

The operational semantics of tabled LP differentiates the first call to a tabled predicate, the *generator*, from subsequent variant calls (calls which are identical modulo variable renaming), the *consumers*. Generators resolve against program clauses and insert the answers they compute in the *table space*. Consumers read answers from the table space and suspend when no more answers are available (therefore breaking infinite loops) and wait for the generation of more answers by their generators. A generator is said to be *complete* when it is known not to be able to generate more (unseen) answers. In order to check this property, a fixpoint procedure is executed where all the

---

consumers inside the generator execution subtree are reading their pending answers until no more answers are generated. After completion, memory used by consumer suspensions can be reclaimed. The completion operation is complex because a number of generators may be mutually dependent, thus forming a Strongly Connected Component (SCC [2]) in the graph of generator dependencies. As new answers for any generator can result in the production of new answers for any other generator of the SCC, we can only complete all generators in an SCC at once, when the completion fixpoint has been reached. The SCC is represented by the *leader* generator: the youngest generator which does not depend on older generators. A leader generator defines the next completion point.

One of the key decisions in the implementation of tabled LP is when a generator returns its computed answers, i.e., the *scheduling* or *evaluation strategy*. *Local evaluation* is the most widely spread evaluation strategy: it executes the full completion fixpoint procedure before returning any answer outside the generator execution subtree. It is efficient in terms of time and stack usage when all answers are needed, but performs speculative work when only a subset of the answers is required. The speculative work performed by local evaluation makes pruning quite ineffective in practice, since it cannot take place until all answers have already been computed.

A work-around for the speculative work of local evaluation is *answer-on-demand* tabled evaluation, where generators return answers as soon as they are computed. The first attempt proposed is *batched evaluation*, but it can be very inefficient memory-wise because it delays completing fixpoint computations without reclaiming the memory used by consumer suspensions. *Swapping evaluation* [3] works around this issue with a memory behavior which is closer to that of local evaluation. Swapping evaluation avoids speculative work before returning demanded answers, but it performs the same amount of work as local evaluation when backtracking. This brings the necessity for pruning operators in tabled LP in order to be able to discard unnecessary alternative execution paths. The contribution of this paper is a *discussion of the issues related to pruning in tabled LP which motivate the implementation of an efficient pruning operator —a version of once/1— with a more natural semantics for the realm of tabling than that of the standard cut operator. once/1 is implemented under swapping tabled evaluation, and we identify a series of optimizations, programming patterns, and general types of applications where it can be used advantageously*. The final goal is to enlarge the domains in which tabled LP can be put to work in a natural way.

This paper concentrates on *proper tabling* (or *suspension-based tabling*), which does not recompute execution paths in order to recover the execution state of a suspended consumer. Also, *variant tabling* is assumed, i.e., a call is considered a consumer iff it is identical, modulo variable renaming, to a previous generator. Adapting the proposed solutions to work under *subsumptive tabling*, which considers a goal A to be a consumer of a goal B if $B\theta \equiv A$ for some substitution θ (or, in general, for tabling under constraints where consumers are defined under the notion of entailment [4]) is left for future work. We assume some familiarity with the WAM [5] and proper tabling implementations.

The rest of the paper is organized as follows: Section 2 introduces a number of issues that appear when performing pruning in tabled LP and proposes solutions for them. Section 3 motivates the use of pruning operators in tabled LP by showing different

:- **table** t/2.   :- **table** t/1.   :- **table** t/1, r/1.

t(A,B) :- p(A,C), !, ...   t(X) :- t(Y), !, fail.   t(X) :- r(X).
t(A,B) :- ...   t(1).   t(1).

p(A,B) :- t(B,C), ...   ?−t(X) vs ?− t(1).   r(X) :- t(X).
                                                     r(2).
?−t(X,Y).
                                                     ?−**once**(t(X)) vs ?−**once**(r(Y)),**once**(t(X)).

Fig. 1. !/0 example   **Fig. 2.** !/0 inconsistency   **Fig. 3.** Solution order dependency

applications where this combination is useful. Section 4 shows some implementation
details of the proposed once/1 pruning operator. Section 5 evaluates our solution ex-
perimentally. Finally, Section 6 gives an overview of the related work and Section 7
summarizes some conclusions.

2   Issues to Support Pruning in Tabled LP

A desirable feature of any program language is "to compute only what is needed and to
compute it only once". Tabled LP is useful to solve the second problem, but it needs the
combination of answer-on-demand tabled evaluation with pruning operators to solve
the first one. This section analyzes the issues of combining tabled LP with pruning
operators, showing the drawbacks of the standard cut operator (!/0) and proposing a
different declarative semantics for our version of the once/1 pruning operator.

2.1   !/0 Operator in Tabled LP

The operational semantics of !/0 is strongly based on the depth-first search strategy of
Prolog. The fact that tabled LP does not follow this strategy — the execution order
of tabled clauses is dynamic — makes the operational semantics of !/0 under Prolog
to be not applicable under tabled evaluation. This is specially relevant in the presence
of mutually recursive calls (Figure 1). It is for example quite possible that !/0 cuts the
second clause for one call to t/2 (when, e.g., t(B,C) is a consumer of a complete tabled
call) but not for other calls (when, e.g., t(B,C) is a consumer which suspends). This
behavior is caused by the effects of !/0 spanning across clauses. This is inadequate in
the context of tabled LP, since the execution order of the clauses of a generator is not
always easy to predict.

   Another example of the ill behavior of !/0 in tabled LP is shown in Figure 2. The
tabled evaluation of the query t(X) executes the first clause of t/1 and suspends the con-
sumer call t(Y), executing the second clause of t/1 on backtracking. The second clause
of t/1 generates the answer X=1, which is returned to the toplevel. On backtracking,
X=1 is consumed by the consumer t(Y) before the final failure of the execution. On the
other hand, the tabled evaluation of the query t(1) executes the first clause of t/1, but now

t(Y) does not suspend. t(Y) can be either a generator — which would return answers after its evaluation — or a consumer which reads the available t(1) answer. Thereby, t(Y) succeeds and !/0 prunes the second alternative before executing a failure. Therefore, !/0 produces an inconsistency since t(1) fails and t(X) succeeds with the answer $X = 1$.

2.2   Behavior of Once/1

As we have seen, !/0 adapts badly to tabled LP, but pruning is a necessity for general program techniques such as generate-and-test programs if the generation of further potential solutions is to be pruned when a test condition succeeds — i.e., if only one solution (a *witness*) is necessary. In the context of tabled LP, once/1 provides a much more appropriate semantics: once(G) executes G but it also cancels any external backtracking over G. once(G) is guaranteed to produce at most one solution without any guarantee as to which particular solution it is — it could even be a random pick — and can be expressed in terms of !/0 for a non-tabled goal as once(G) :– call(G), !. Thereby, once/1 is less dependent on the execution order of the generator clauses than !/0 because once/1 does not span across clauses.

once/1 is also useful in order to extend the functionality of !/0 for updating tabling data structures. For example, consider a once(G) call which succeeds. The data and control structures which would be necessary to re-enter the execution of G are not needed any more. To this end, once/1 must remove not only the choicepoints belonging to the current execution path — as !/0 does — but also the consumers which appeared inside the execution subtree of G. The resumption of these consumers might lead to subsequent solutions of the once/1 call, which would contradict the previous rationale. Note that these consumers are, in implementations that use proper tabling, either protected from backtracking or copied away to a separate memory area and would not be pruned by !/0. Note also that this consumer removal, which is necessary for correctness, is not done by other tabled LP approaches to pruning and is not trivial. For example, one of these consumers, say C, might belong to the consumer list of a generator not being pruned, and the completion fixpoint operation of this other generator would resume C.

One may expect that once/1 will return the solution which is less expensive to compute (e.g., the first one to be computed), but the solution order in tabled LP depends on the shape of generator dependencies, which in turn depends on the program history in classical tabled LP implementations. Consider the program in Figure 3 and the query once(t(X)). Execution enters the first clause of t/1 and calls the generator r(X). The execution of the first clause of r/1 suspends since t(X) is a consumer. r(X) computes the solution $X = 2$ and this is the only solution propagated to the toplevel because of the once/1 success. On the other hand, the execution of the query once(r(Y)), once(t(X)) calls the generator r(Y) and its first clause calls to the generator t(Y), whose first clause suspends because r(Y) is a consumer. t(Y) computes the solution $Y = 1$. After that, once(t(X)) is called, which can consume the previous solution $X = 1$. This is the only solution propagated to the toplevel because of the success of once/1. Therefore, the solution to once(t(X)) depends on the execution history. Moreover, if we impose $X = 1$ after calling once(t(X)), the execution succeeds or fails depending on the program history. We can resume this behavior in the following dependency chain:

program history $\Rightarrow$ SCCs $\Rightarrow$ solution order $\Rightarrow$ once/1 solution $\Rightarrow$ program results

The shape of generator dependencies could be made to depend on statically predictable characteristics which would remove dependencies from the history, but we did not find any completely satisfactory order. For example, the lexicographical order could be used, but let us consider the execution of the previous query once(t(X)). When t(X) is called from the first alternative of r(X), t(X) could be recomputed as if it was a generator because r(X) cannot depend on t(X) (r(X) comes first under the lexicographical order). Given a program, its solutions would not depend on the program history since the lexicographical generator priority fixes the shape of generator dependencies, but a change to the names of the tabled predicates could change the order of the solutions, which is arguably not the best situation. In general, orders which rely on syntactic characteristics of the source code are sensitive to changes on the program text which are very common and which programmers do not expect to result in alterations to the behavior of the program.

This dependency on the program history is an open issue in existing tabled LP supporting pruning operators, although it has not been documented before. However, we strongly believe that having a once/1 operator available is worthwhile because the combination of pruning and answer on-demand tabled evaluation is very efficient in a variety of applications, as we will show in the next section. Also, the behavior of the program execution in tabled LP with pruning operators is consistent as long as the programmer is aware that (s)he cannot rely on which particular solution will be returned by a call to once/1. Keeping this property in mind, the query once(t(X)), $X = 1$ makes little sense and it is a questionable programming pattern.

3   Applications of Once/1

We motivated the once/1 operator as a general instrument for programs which benefit from tabled LP but which only need a subset of all the possible solutions. In addition, there are a number of programming patterns where once/1 is quite useful and which are worth mentioning.

3.1   Generate and Test Applications

Consider a model checker based on tabled LP, such as XMC [6], which performs reachibility analysis. A typical case is the verification of a mutual exclusion protocol where each configuration state is a tuple with a state q_i for each process P_i. For example, for three processes the state $\langle q_1, q_2, q_{me} \rangle$ represents a configuration where process P_1 is in state q_1, process P_2 is in state q_2, and process P_3 is in a *mutual exclusion* state.

A model checker application would provide the predicate reach/2, which returns in its second argument all the configuration states reachable from the configuration state given by its first argument. Therefore, all the configuration states reachable from the state I_0 are returned by the query ?– reach(I_0, X). Note that, for verification purposes, the search can be stopped when two processes are in the mutual exclusion state at the same time. This condition can be expressed with the following facts and query (where initial/1 returns the initial state):

check($\langle q_{me}, q_{me}, \_ \rangle$).
check($\langle q_{me}, \_, q_{me} \rangle$).
check($\langle \_, q_{me}, q_{me} \rangle$).

?– initial(I_0), **once**(reach(I_0, X), check(X)).

3.2 Early Completion Optimization

Early completion [7] is an optimization for tabled LP which completes a generator call when a new answer does not further instantiate the call and is therefore the most general answer. In that case, further backtracking over the early-completed generator is unnecessary. This is the same objective that a once/1 which succeeds pursues. Early completion optimization can then be easily implemented by associating a once/1 call which does not appear in the program and whose final activation is to be dynamically decided (which we term a *virtual* once/1) with all the generator calls. When all the free variables of a generator call remain unbound when one of the generator answers is found, a (virtual) success of the generator virtual once/1 call can be simulated. As we will see in Section 5.2, early completion optimization based on once/1 clearly outperforms other early completion optimization implementations. Also, as early completion optimization is performed when free variables remain uninstantiated, early completion optimization based on once/1 does not present the issues commented in Section 2.2.

3.3 Pruning at the Top Level

A (virtual) once/1 call can be also associated to the toplevel query in order to perform pruning at the top level. Similarly to the implementation of early completion optimization, pruning at the top level when no more answers are demanded by the user can be achieved by simulating a (virtual) success of the toplevel virtual once/1 call.

3.4 If-Then-Else Prolog Transformation

The Prolog program transformation for the classical *Cond* -> *A*; *B* statement is as follows:

> **if**–then–**else**(Cond,A,B)) :- Cond, !, A.
> **if**–then–**else**(_,_,B)) :- B.

which does not work if *Cond* needs tabled evaluation. This is due to two main reasons: a) *Cond* might suspend and then, *B* would be executed; later on, a resumption of *Cond* might lead to the execution of *A*; b) as remarked in Section 2, !/0 does not ensure at most one solution of pruned tabled calls. The first issue can be solved by supporting negation in tabled LP [8], which is usually implemented by the tnot/1 operator. The second one can be solved by using once/1 instead of !/0. The new transformation for *if-then-else* statements in tabled LP would be:[1]

> **if**–then–**else**(Cond,A,B)) :- **once**(Cond), A.
> **if**–then–**else**(Cond,_,B)) :- tnot(Cond), B.

---

[1] Note that the call to tnot/1 succeeds at most once.

```
:- table path/3.
path(X,Y,Cost) :-              min_path(X,Y,Best,Min) :-
     edge(X,Y,Cost).                once(path(X,Y,Best)),
path(X,Y,C) :-                      NewBest #< Best,
    C #= C1 + C2,                   once((min_path(X,Y,NewBest,Min) ; Min = Best)).
    C #>= 0,
    edge(X,Z,C1),              ?- min_path(X,Y,_,Min).
    path(Z,Y,C2).
```

Fig. 4. Constraint-based optimization

3.5 Application to Minimization Problems

Although we currently support only variant tabling under swapping evaluation, Ciao can combine tabled LP and CLP [4] and work to combine them with once/1 under swapping evaluation is underway. The resulting system can be applied, for example, to a declarative and efficient formulation of optimization. Consider the program in Figure 4.[2] min_path/4 iteratively calls path($X,Y,Cost$) and successively constrains the path cost. It is called inside once/1 because we are interested in a single solution. Note that the recursive calls can perform the reactivation operation (which will be explained in Section 4.6) in order to continue the generator execution at the point where it was pruned after imposing some more tighter constraints. When the constraints are too tight and the path/3 call fails, the immediately previous cost is returned. The procedure implements a *branch and bound* algorithm where tabled LP avoids loops and redundant work, constraints are used to implement bounds which cut the search, and once/1 restricts the search to return only one witness.

4 Implementation Details of the Once/1 Operator

This section recalls the general ideas of swapping evaluation [3] and explains the implementation of the once/1 operator, which is based on the management of once scopes and the pruning procedure associated with them. We will also see some optimizations as the reactivation operation or memory reclaiming after a pruning operation.

4.1 Swapping Evaluation

Pruning needs answer-on-demand tabled evaluation to be more effective. Swapping evaluation [3] is an answer-on-demand strategy for tabled LP which solves the memory consumption issues of batched evaluation. It implements a different behavior for *internal* and *external* consumers. An internal consumer appears inside the execution subtree of the leader of the generator of the consumer. E.g., in a program with clauses {(:-table a/0), (a :- a), (a)} with the query ?- a, a, the leftmost a/0 in the query is a generator, the a/0 in the body of the first clause is an internal consumer, and the rightmost

---

[2] The symbol # differentiates constraints from arithmetical operators.

a/0 in the query is an external consumer. Using swapping evaluation, internal consumers behave as usual, but external consumers read answers from the table space and, when no more answers are available, they move the choice points and their corresponding trail cells of their generators to the top of the stacks in order to modify the backtracking execution order. The original generator is then transformed into a consumer and the external consumer becomes a generator which can produce more answers, avoiding the use of memory for external consumer suspensions — which is the most important source of memory consumption in batched execution.

4.2 Once Scope Data Structure

A *once scope* is a data structure associated with a once/1 call which keeps track of relevant information in order to perform the pruning operation. Once scopes are hierarchically organized, because a once/1 call can be called from the execution subtree of another once/1 call (the latter being the *parent* of the former). Note that this hierarchical structure includes the (virtual) once scopes associated to generator calls. Therefore, the consumer list of a generator is directly accessible via its virtual once scope.[3]

A once scope S is composed of the following fields: choicepoint, parent, children set, consumer set and generator set. choicepoint indicates the choicepoint at time of the once/1 call corresponding to S. parent indicates the parent once scope of S. children set stores the set of once/1 calls which are immediately called from S (those once scopes whose parent field points to S). consumer set is the set of consumer calls which are called when S is the *active once scope*. The active once scope is the youngest once scope of those whose execution subtree is being executed. Similarly, generator set is the set of generator calls which are called when S is the active once scope.

4.3 The Management of Once Scopes

Figure 5 shows Prolog code for the once/1 operator, which is responsible for managing the once scopes. Once scopes are stored on the *once scope stack*, whose topmost element is the *active once scope*. new_once/1 initializes $Scope$, a new scope for the current once/1 call. It initializes the choicepoint and parent fields of $Scope$ and updates its children set field.[4] push_once/1 pushes $Scope$ onto the once scope stack to indicate that it is now the active once scope. If G succeeds, once_proceed/0 performs the pruning operation related to the active once scope. After that, pop_once/0 pops off $Scope$.

```
once(G) :-
    new_once(Scope)
    push_once(Scope),
    undo(forward_trail(
        push_once(Scope),
        pop_once)),
    call(G),
    once_proceed,
    pop_once,
    undo(forward_trail(
        pop_once,
        push_once(Scope))).
```

Fig. 5. once/1 predicate

One additional difficulty is that consumer calls which suspend within a once/1 call have discontiguous executions. Let us consider the call once(C), where C is a consumer

---

[3] We consider the consumer list of a generator as the consumers appearing inside the generator execution subtree instead of the repeated calls up to variable renaming. This does not affect the completion operation fixpoint.

[4] children set is also updated when a generator completes or after a swapping operation.

call which suspends. The execution might exit the once/1 subtree on suspension and reenters it when resuming, which requires the once scope structure to be popped off on suspension and pushed on on resumption. The mechanism we have used is to leave actions on the trail to be executed on untrailing (e.g., when suspending to enter another clause) and on resumption (when reinstalling the trail to continue a suspended call after a new answer is available). We insert in the trail, via the undo/1 operation,[5] the forward_trail/2 goal, which is defined to execute its second argument when called. This second argument is then invoked on backtracking when C is suspended. The resumption mechanism in turn recognizes forward_trail/2 when reinstalling the trail and calls its first argument. Therefore, the first undo/1 always discards the scope when the call finally fails. In the case of a consumer inside the execution subtree of once/1, it also uninstalls the scope when performing untrailing to suspend and pushes the scope back onto the stack on resumption. The second undo/1 performs the reverse operation, which is needed to neutralize the actions of the first undo/1 in order to resume consumers outside the execution subtree of the current once/1 call: it reinstalls the once scope on backtracking which will be popped off by the first undo/1 and pops off the once scope which has been previously reinstalled by the first undo/1 on consumer resumption.

The virtual once scopes associated with generators and the toplevel execution are managed by a similar code, but once_proceed/0 is not executed by default. For these cases, once_proceed/0 is executed if the early completion optimization can be performed or no more answers are demanded by the user, respectively.

4.4   Terminology

Note that the consumers and generators of a once scope S also belong to the once scope of the parent of S (although they do not directly appear in their consumer/generator set fields). We recursively define the *once-recursive consumer set* of a once scope S as the members of the consumer set field of S plus the *once-recursive consumer set* of the members of the children set field of S. We define similarly the *once-recursive generator set* of a once scope.

Remember that we have associated a *virtual* once scope to all generators. The consumer list of a generator — those appearing inside its execution subtree but not in the execution subtree of internal generator calls — are the members of the once-recursive consumer set of the once scope associated with the generator. We also define the *recursive consumer set* of a once scope S as the once-recursive consumer set of S plus the *recursive consumer set* of the once scopes associated with the members of the once-recursive generator set of S, i.e., it also includes the consumers inside the execution subtree of internal generators, and therefore it is made up of the set of consumers inside the execution subtree of the once/1 call. We also define the *recursive generator set* of a once scope S accordingly.

For example, following Figure 6, there is a once/1 call (associated to the once scope $ONCE_B$) which is internal to the execution of another once/1 call (associated to the once scope $ONCE_A$). $ONCE_{G_1}$ and $ONCE_{G_2}$ are the virtual once scopes associated to, respectively, the generators G_1 and G_2. These generators are called from the execution

---

[5] undo/1 is a common facility which leaves a goal call in the trail to be invoked on untrailing.

$ONCE_A$:
  Parent: *NULL*
  Consumer Set: $\{C1, C2\}$
  Generator Set: $\{G1\}$
  Once Set: $\{ONCE_B\}$

$ONCE_{G_1}$:
  Parent: $ONCE_A$
  Consumer Set: $\{C4\}$
  Generator Set: $\{G2\}$
  Once Set: $\{\}$

$ONCE_B$:
  Parent: $ONCE_A$
  Consumer Set: $\{C3\}$
  Generator Set: $\{\}$
  Once Set: $\{\}$

$ONCE_{G_2}$:
  Parent: $ONCE_{G_1}$
  Consumer Set: $\{C5\}$
  Generator Set: $\{\}$
  Once Set: $\{\}$

Fig. 6. Once structures

of the once/1 call associated to $ONCE_A$. The internal consumers of a virtual once scope are included into the recursive consumer set of any of its parent once scopes, but they are not included into the once-recursive consumer set of any of its parent calls. Consequently, the once-recursive consumers of the once scope $ONCE_A$ are $C1$, $C2$ and $C3$, and the recursive consumers of the once scope $ONCE_A$ are $C1$, $C2$, $C3$, $C4$ and $C5$.

4.5   The Pruning of a Once Scope

once_proceed/0 is responsible for pruning the active once scope. Its pseudo-code is:

> DELETE act_once_scope from ParentOf(act_once_scope);
> $current\_choicepoint$ = InitChoicepoint(act_once_scope);
> **for each** $G \in$ recur_gen_set(act_once_scope) **do** state(G) = PRUNED;

The first line deletes the active once scope from the once scope set of its parent once scope. This operation causes the removal of all the consumers inside the execution subtree of the active once scope, because the active once scope (and its consumers) is not reachable from the once scope of any generator any more and then, these consumers will not be traversed by the execution of any completion fixpoint procedure. After that, the current choicepoint is updated to the one of the active once scope in order to discard pending search of the execution subtree of the once/1 call being pruned. Finally, all the non-complete generators inside the execution subtree of the active once scope are marked as PRUNED in order to avoid inconsistencies if one of their consumers appear. We follow a similar approach to the one of incomplete tables [9], but it is improved with the use of the *reactivation operation* (see Section 4.6). The main goal of the incomplete table proposal is to avoid the generator recomputation when the answers of a PRUNED generator are enough to evaluate a (future) consumer. (Future) consumers consume the available answers from its PRUNED table, and only if all such answers are exhausted, the generator is computed from scratch. Later, if the computation is pruned again, the same process is repeated until eventually the subgoal is completely evaluated. Note that each recomputation from scratch computes at least one more solution, keeping the tabled LP termination property for programs with the *bounded term-size*.

```
:- table t/1, r/1.

t(X) :- r(X).
t(1).
...
t(X) :- ...

r(X) :- large_comp, once(t(X)).
```
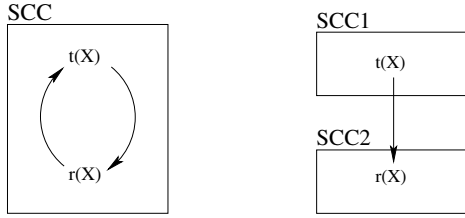

Fig. 7. Consumer Optimizations

4.6   Pruning Optimizations

We here propose some optimization which can be applied to the implementation of once/1. They do not affect to the operational semantics of once/1, but can improve the time/memory execution of tabled LP applications.

The Reactivation Operation. The subtree under a pruned generator might not be fully explored at the moment of pruning, possibly discarding the computation of pending answers. Thereby, (future) consumers of pruned generators might require answers which were not computed due to the pruning. Two main approaches have been proposed so far: either keep the solutions in the answer table and protect the execution subtree from backtracking [10], or keep the solutions in the answer tables but discarding the execution subtree [9]. The former, based on the *reactivation operation*, might be interesting for applications where the pruned generators are often reactivated, arbitrarily improving the execution speed. However, we decided to implement the latter because the memory consumption of the former could be unacceptable. For example, using stack freezing, the trail section to be saved at time of pruning is unbounded because backtracking might be performed until the initial choicepoint.

We improve over incomplete tables for cases where the reactivation operation comes for free. once_proceed/0 marks as PRUNED the generators inside the execution subtree of the active once scope and updates the current choice point, but these operations can be performed lazily on backtracking (just before entering the pruned alternatives). Therefore, the execution subtree of pruned generators is kept on the stacks and can be reused by a swapping operation executed in the continuation code. This special case of the swapping operation implements the reactivation operation for free. On the other hand, after backtracking over the prune generators, the execution of the pruned generators is reclaimed and it must be computed from scratch if future consumers demand more answers than the ones available.

Memory Optimizations after Pruning Success. Another optimization is related to the removal of the consumers inside the execution subtree of active once scope — which trivially speeds up the completion operation because fewer consumers have to be traversed in the completion operation fixpoint. This removal can be also used to reduce

dependencies between generators being evaluated — fewer dependencies lead to the sooner completion of generators and better memory use — and to reclaim unneeded consumer memory of pruned consumers which is protected from backtracking.

In the example in Figure 7 (left-side), the query t(X) is a generator whose first clause calls r(X), another generator. r(X) starts a large computation and, afterward, once(t(X)) is called. The first clause of t(X) suspends when calling r(X). The generator dependency graph at this moment is shown in Figure 7 (in the middle), where there is only a completion point represented by the leader t(X). After the consumer call to r(X) suspends, execution backtracks and the second clause of t(X) is executed, computing the answer X=1. This makes the once/1 call succeed and the previously suspended consumer is removed (and therefore ignored by the completion fixpoint). This can be used for reclaiming the memory associated to these consumers —which is probably frozen on the stacks —and for updating the graph of generator dependencies as shown in Figure 7 (right-side), removing the dependency of r(X) on t(X). The new graph of generator dependencies defines two different completion points, corresponding to two different leaders, t(X) and r(X). Therefore, r(X) can complete on backtracking. The completion of r(X) improves the program memory behavior because the memory used by large_comp/0 is reclaimed before exploring alternative clauses of the generator t(X). The pseudo-code to perform this optimization is as follows:[6]

$Oldest_{leader}$ = oldest(Gen(C) s. t. ($C \in$ recur_cons_set(act_once_scope)));
for ($G = G_a$; $Oldest_{leader}$!= G; G = ParentOf(G))
  Leader(G) = oldest($G \cup$ {Gen(C) s. t. ($C \in$ recur_cons_set(OnceScope(G)))});

The first line computes the oldest dependency of the active once scope. The second line traverses generators starting from the youngest one being executed, G_a, until $Oldest_{leader}$, in order to update their generator dependencies.[7] Since once_proceed/0 has just been executed, the third line computes the new oldest dependency of G without taking into account the pruned consumers. At the end of the execution of this code, the leader fields have been updated according to the new graph of generator dependencies.

There is another optimization for reclaiming the memory frozen by consumers which is independent from the updating of generator dependencies. This optimization refers to the case where the topmost frozen memory corresponds to consumers being pruned. In this case, we can update the value of the frozen memory in order not to protect from backtracking the memory of consumers which have been pruned. The pseudo-code for this optimization is as follows:[8]

MAX_{froz_mem} = max(FrozMem(C) s. t. ($C \in$ recur_cons_set(act_once_scope)));
if ($FrozMem$ == MAX_{froz_mem})
  $FrozMem$ = max(FrozMem(C) s. t. ($C \in$ recur_cons_set(OnceScope(Leader(G_a)))));

The first line computes MAX_{froz_mem}, the maximum frozen memory by the consumers inside the execution tree of the once scope being pruned. Since once_proceed/0

---

[6] This code follows the call to once_proceed/0 in Figure 5.
[7] Generator dependencies of generators older than $Oldest_{leader}$ are unaffected.
[8] This code follows the call to once_proceed/0 in Figure 5.

has just been executed, the third line computes the new maximum frozen memory without taking into account the pruned consumers. $FrozMem$ is only updated if its previous value is different than the one of MAX_{froz_mem} because if these values are the same, the current frozen memory corresponds to a consumer outside of the execution subtree of the once scope being pruned.

5   Performance Evaluation

We have implemented the once/1 pruning operator under swapping evaluation in Ciao and compared its performance w.r.t. XSB version 3.3.6. Both systems were compiled with gcc 4.5.2 and executed on a machine with Ubuntu 11.04 and a 2.7GHz Intel Core i7 processor.

5.1   Applications Searching an Answer Subset

Table 1 shows execution times in ms. for a set of applications which can take advantage of once/1 in order to compute a subset of answers. numbers searches for an arithmetic expression which evaluates to a given natural number (N), given a list of natural numbers (S) (100+ lines). Tabled LP is used to avoid the recomputation of recursive calls with a subset of S. The suffix none indicates a query where N cannot be obtained using S, the suffix easy indicates a query where the first solution implies the computation of a small fraction of the search space and suffix stand indicates a query with no special characteristics (i.e., no specific search tree shape was sought). iproto, leader, and sieve are model checking applications where reachability analysis is performed (600+ lines each). numbers uses once/1 in the definition of its tabled predicates in order to return only one answer. iproto, leader and sieve queries are embedded in a once/1 operator in order to prune the search when the first answer is returned, e. g. once(iproto(init,$FinalState$)). We measure the time to return the first answer for each query in the *first* column and also until final failure the *all* column (i.e., when all the solutions are computed, when that is the case). We show execution times of Ciao under local and swapping evaluation, using once/1 or not.

numbers_none cannot take advantage of either swapping evaluation or once/1 because it must explore the full search space, since no solutions are found. Its different execution times provide an intuition regarding the overhead of swapping evaluation and once/1, which in this case are both almost negligible. In numbers_easy, local evaluation has to compute all the possible expressions while swapping evaluation can return the first one and stop, which takes much shorter. The use of once/1 allows swapping evaluation to discard alternative execution paths before performing backtracking — note that swapping evaluation would compute all the answers on backtracking unless once/1 is used, which gives us a strong reason for the necessity of combining answer-on-demand tabled evaluation and pruning operators. With respect to local evaluation, it takes some more time to return all the solutions than to return the first one, because they have to be reconstructed from the table space where they were stored after having been computed before returning the first one. once/1 under local evaluation makes it possible to discard solutions when the first one is found, but recursive tabled calls were still completely evaluated in a speculative way. We conclude that a pruning operator can be used

Table 1. Execution time (ms.) of local vs. swapping evaluation with/without pruning operators

	Local				Swapping			
	No once/1		once/1		No once/1		once/1	
Query	first	all	first	all	first	all	first	all
num_none	21 912	21 912	22 365	22 365	22 574	22 574	22 982	22 982
num_easy	20 613	21 108	17 023	17 027	1 624	22 421	1 708	1 792
num_stand	24 296	25 312	22 750	22 753	8 403	26 058	8 729	8 906
iproto	2 992	3 184	3 014	3 016	1 112	3 024	1 126	1 141
leader	9 940	10 324	10 182	10 186	3 296	10 963	3 412	3 433
sieve	35 554	36 272	35 986	35 991	7 081	37 139	7 107	7 123

under local evaluation, but it is obviously less interesting than under swapping evaluation. numbers_stand has a behavior similar to numbers_easy, but the first solution takes some more time to be computed. Note that the effects of pruning on execution time depend heavily on when the first answer is found, because pruning only affects the remaining search space. iproto, leader, and sieve show an overall behavior similar to that of numbers.

5.2 Early Completion Based on Once/1

Existing proposals for early completion optimization are highly dependent on the syntactic form of the generator clauses and often allow unnecessary computations. For example, the XSB early completion optimization updates the next instruction of the generator choicepoint to be the completion fixpoint procedure, avoiding the computation of the alternative generator clauses. It does not perform either reactivation of pruned generator calls or updating of the graph of generator dependencies based on the consumer removal. These drawbacks are overcome by our early completion optimization based on once/1. As an example, let us analyze the behavior of the handcrafted code in Figure 8 in XSB. t1/0 is a generator whose first clause calls t2/0, another generator. t2/0 calls t1/0, performing a consumer suspension. On backtracking, t2/0 cannot complete because of this dependency. Now, the second clause of t1/0 is executed and it succeeds. At this moment, t1/0 can be completed early (discarding pending execution alternatives), but its fixpoint procedure is still executed. The consumer of t1/0 is resumed, (speculatively) executing a large computation (sleep(2)). Obviously, the resumption of this consumer is unnecessary for forward execution and it would not have been performed under early completion optimization based on once/1. In contrast, the generator t2/0 would have been marked as pruned to be later reactivated if needed.

Table 2 shows execution times in ms. for a set of benchmarks which can take advantage of early completion optimization. genome computes relations following a genome structure represented as a graph. The suffixes give some rough indications of the shape of the graph. We measure Ciao and XSB, using local evaluation in both cases for fairness in the comparison.[9] The no_early column shows execution times taken after modifying the XSB sources to deactivate early completion optimization and the early column

---

[9] Note that early completion is effective even under local evaluation since it prunes the generator execution after computing its first solution.

Table 2. Execution time (ms.) of early completion optimization

	XSB			Ciao		
	no_early	early	speedup	no_early	early	speedup
bad_xsb	2000	2000	1.00	2000	0.3	6666
genome_chain	28.8	24.8	1.16	33.1	23.5	1.41
genome_grid	102.4	60.4	1.69	116.2	12.7	9.15
genome_cycle	290.0	212.5	1.36	324.7	1.8	180.38
genome_dense	3.2	0.4	8.00	5.1	0.2	25.50

shows execution times with the early completion optimization activated. As explained previously, the early completion optimization in Ciao is based on once/1.

bad_xsb takes 2000 ms. in XSB and less than 1 ms. in Ciao, confirming the previous analysis. The rest of the benchmarks show more realistic scenarios, where XSB (no_early) executes sometimes faster than Ciao (no_early). One reason is that the Ciao tabled LP implementation is based on a program transformation which imposes some overheads. But the main focus of interest here is the search space which is pruned by the early completion optimization. XSB takes advantage of early completion optimization, speeding up the execution between 1.16× and 8×, while Ciao obtains speedups between 1.41× and 180×, showing that early completion optimization based on once/1 can clearly be more effective in many cases. The execution times of genome_cycle and genome_grid, which generate situations similar to the one of bad_xsb where the Ciao early completion optimization discards expensive fixpoint computations which XSB executes, are the ones where Ciao gets the most advantage w.r.t. XSB.

```
:- table t1/0, t2/0.

t1 :- t2.
t1.

t2 :- t1, sleep(2).

?– t1.
```

Fig. 8. bad_xsb example

6   Related Work

To the best of our knowledge, there are five previous attempts to incorporate pruning operators in tabled LP [11,12,9,13,10]. [11] works under the dynamic reordering of alternatives (DRA) technique [14]. Because the abstract machine for DRA is much more WAM-like than the implementations of proper tabling, the authors claim that the DRA implementation of cut is closer to that of !/0 in the WAM. They argue that, since the DRA scheduling strategy is deterministic, this allows for a well-defined !/0, with a more intuitive operational semantics. DRA tries non-looping alternatives first and looping alternatives later on, and this is the order in which !/0 prunes. In fact, proper tabling implementations could be made to follow the same order for consumer resumptions as DRA. However, we tend to agree with [10] that the behavior resulting from the implementation of !/0 in DRA can still be confusing, as argued in Section 2.2. Also, DRA is based on recomputation of looping alternatives, while proper tabling does not

re-execute except for the cost of reinstalling trailed bindings, offering a quite different trade-off. Our proposal is tailored to proper tabling.

Tabling modes [12] is also based on !/0. It is used at the level of program definition, which restricts the flexibility for the case of applications which sometimes need all the solutions and sometimes need a subset of them. It uses a lazy strategy, which computes all the solutions as local evaluation does. Consequently, tabling modes do not prune the tabled evaluation. A minimization problem as that in Section 3.5 would not use previous solutions to prune the search space.

Incomplete tables [9] is also based on !/0. They do not provide a robust implementation (Yap Prolog documentation alerts that the behavior of tabled LP with !/0 is undefined). Also, its implementation does not support the reactivation operation.

Demanded tables [13] implements a version of once/1. In this work, calls which are being consumed by external consumers (demanded table) are not pruned, which makes it necessary to perform runtime analysis to detect if a generator call is being demanded. We avoid this analysis by supporting reactivation of tables. We do not care if a generator to be pruned is being demanded, since the demanding consumers would reactivate the generator if needed.

JET [10] is closer to the spirit of this work, although no implementation is provided. The ideas presented are also based on reactivation of tables, but this work does not provide any pruning operator for the user. Instead, pruning takes place on *JET points*, which are detected by static analysis. This is a deliberate design decision to facilitate the job of the programmer, but it implies a loss of pruning power. For example, our numbers benchmark would not benefit from JET pruning. We strongly believe that the semantics of once/1 is clear enough for the programmer, although we could of course adapt our pruning operator to be based on analysis. Other minor advantages of our pruning operator are that once/1 is linear in the number of generator choicepoints while JET pruning is linear in the number of choicepoints, that once/1 does not impose any overhead if pruning is not used, and that once/1 does not store any choicepoint more than once to allow future reactivations, among others.

Finally, the most important contribution of our pruning mechanism is the pruning of consumers inside the execution subtree of a pruning operator. They must be removed in order not to execute the continuation of a pruning operator more than once — the resumption of these consumers might lead to a new execution of this continuation code. Moreover, we propose some memory optimizations to take advantage of the consumer removal after a pruning operation.

7 Conclusions

We argue that none of the previous approaches for pruning in tabled LP is fully satisfactory although a pruning operator under answer-on-demand tabled evaluation is a necessity in order to enlarge the application domain of tabled LP. To this end, we have presented and evaluated a pruning operator under swapping evaluation, and reported on benchmarking of its implementation in Ciao, comparing it to previous proposals, and showing that it offers advantages in terms of efficiency and programmability. We have also shown how our pruning operator can be used as a basis for implementing a number of optimizations.

References

1. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM 43(1), 20–74 (1996)
2. Tarjan, R.: Depth-First Search and Linear Graph Algorithms. SIAM J. Comput. 1, 140–160 (1972)
3. Chico de Guzmán, P., Carro, M., Warren, D.S.: Swapping Evaluation: A Memory-Scalable Solution for Answer-On-Demand Tabling. TPLP 10 (4-6), 401–416 (2010)
4. Chico de Guzmán, P., Carro, M., Hermenegildo, M.V., Stuckey, P.: A general implementation framework for tabled CLP. In: Schrijvers, T., Thiemann, P. (eds.) FLOPS 2012. LNCS, vol. 7294, pp. 104–119. Springer, Heidelberg (2012)
5. Ait-Kaci, H.: Warren's Abstract Machine, A Tutorial Reconstruction. MIT Press (1991)
6. Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S., Dong, Y., Du, X., Roychoudhury, A., Venkatakrishnan, V.: XMC: A Logic-Programming-Based Verification Toolset. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 576–580. Springer, Heidelberg (2000)
7. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems 20(3), 586–634 (1998)
8. Swift, T., Warren, D.S.: XSB: Extending Prolog with Tabled Logic Programming. TPLP 12(1-2), 157–187 (2012)
9. Rocha, R.: Handling Incomplete and Complete Tables in Tabled Logic Programs. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 427–428. Springer, Heidelberg (2006)
10. Sagonas, K.F., Stuckey, P.J.: Just Enough Tabling. In: PPDP 2004, pp. 78–89. ACM (August 2004)
11. Guo, H.F., Gupta, G.: Cuts in Tabled Logic Programming. In: Demoen, B. (ed.) CICLOPS 2002, pp. 62–73 (July 2002)
12. Guo, H.F., Gupta, G.: Simplifying Dynamic Programming via Mode-directed Tabling. Softw. Pract. Exper. 38(1), 75–94 (2008)
13. Castro, L.F., Warren, D.S.: Approximate Pruning in Tabled Logic Programming. In: Degano, P. (ed.) ESOP 2003. LNCS, vol. 2618, pp. 69–83. Springer, Heidelberg (2003)
14. Guo, H.-F., Gupta, G.: A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In: Codognet, P. (ed.) ICLP 2001. LNCS, vol. 2237, pp. 181–196. Springer, Heidelberg (2001)