CAT: the Copying Approach to Tabling

Bart Demoen and Konstantinos Sagonas

Department of Computer Science
Katholieke Universiteit Leuven
B-3001 Heverlee, Belgium
e-mail: {bmd,kostis}@cs.kuleuven.ac.be

Abstract. The SLG-WAM implements tabling by freezing the WAM stacks: this technique has a reasonably small execution overhead, but is not easy to implement on top of an existing Prolog system. We propose a new technique for the implementation of tabling: the Copying Approach to Tabling. CAT does not interfere with normal Prolog execution and can be introduced in an existing Prolog system orthogonally. We have implemented CAT starting from XSB by taking out SLG-WAM and adding CAT. We describe the additions needed for adopting CAT in a WAM implementation. We show a case in which CAT performs arbitrarily worse than SLG-WAM, but on the other hand we present empirical evidence that CAT is competitive and often faster than SLG-WAM. We discuss issues related to memory management and the impact of the scheduling.

1 Introduction

Tabling in logic programming has been proven useful in a wide range of application areas such as parsing, deductive databases, program analysis based on abstract interpretation, and recently verification through model checking. The most practical implementation of tabling is found in XSB [11]: it also seems the only general Prolog system with tabling. The *Table Space* of XSB is organized using *tries* and table access mechanisms are optimised even further through *substitution factoring* [8]. Also, XSB currently implements two different scheduling strategies [5]. In this paper, we will be concerned mainly with the third aspect of tabling, which is the control and which is orthogonal to the other two. Control, i.e. the need to *suspend* and *resume* computations, is a main issue in a tabling implementation, because some subgoals, called *generators*, generate answers that go into the tables, while other subgoals, called *consumers*, consume answers from the tables; as soon as a generator depends on a consumer, the generator and the consumer must be able to work in a coroutining fashion, something that is not readily possible in a WAM implementation of Prolog. The execution of the query `?-` $\texttt{p}_g$`(X).` against the following small program exemplifies this situation.

```
:- table p/1.
p(1) :- p_c(Y).
p(2).
```

The subscripts g and c denote the occurrence of a subgoal that is a generator or consumer for this particular query. The answer `p(1)` cannot be generated before $\mathtt{p}_c$ has consumed the other answer, `p(2)`, from the answer table that $\mathtt{p}_g$ fills. At the moment that $\mathtt{p}_c$ consumes the answer `p(2)`, it must be in an execution state which is the same as when it was selected first. But, in a WAM implementation, backtracking has removed part of that state — because without backtracking, the answer `p(2)` could not have been generated — so the state of p_c must be preserved. The SLG-WAM [10], the abstract machine of XSB, preserves consumer states by *freezing* them, i.e. by not allowing backtracking to reclaim space on the stacks as is done in WAM. In implementation terms, this means that the SLG-WAM has a extra set of *freeze registers*, one freeze register for each of heap, trail, local and choice point stack.[1] Moreover, the trail needs to record also values because they have to be reinstalled together with the consumer.

This management of control through freezing slows down execution which is not related to tabling: this overhead is actually smaller than people usually assume (order of 10% for an emulated implementation [10]), but we will show in this paper that it can be completely avoided through the adoption of CAT. Moreover, freezing is complicated and not easy to put into an existing Prolog system: this fact might be the main reason why logic programming systems do not yet generally offer tabling. Also here, CAT saves the day because by using CAT, tabling can be added to a WAM implementation in an orthogonal way.

Instead of freezing the consumer state, one could imagine that the whole state of the abstract machine (i.e. all the stacks) is saved in a separate memory area, and then execution just fails over the consumer. When we need to reinstall the consumer, we can just revert to the saved copy and feed the consumer with its answers. This is not a good solution for two reasons: copying the whole WAM state is unnecessary (as we will show later) and it also leads to unnecessary recomputation. The execution of the query `?-` $\mathtt{p}_g$`(X).` against the following program shows this.

```
 :- table p/1.
p(1) :- b.    b :- pc(Y).
p(2).         b :- ... .
```

The state of the abstract machine at the moment $\mathtt{p}_c$ is called, contains the choice point for `b`. Still, the second alternative of `b` will have been exhausted by the time $\mathtt{p}_g$ generates its first answer (in this case `p(1)`). When we reinstall the consumer $\mathtt{p}_c$, we do not want to reinstall the alternative for `b` as the ... represent an arbitrary amount of computation. Thus, a more selective copying of the WAM state can and should be done. CAT does exactly this: it copies selectively (and incrementally) execution states of consumers and reinstalls these copies when needed.

CAT does not require any changes to the WAM, only additions in the form of a few new instructions which can be alternatively seen as new built-in predicates. The choice points for tabled subgoals in CAT differ slightly from usual WAM

[1] Throughout this paper, we assume a WAM model with environment and choice point stacks separated (as XSB or SICStus Prolog implement) rather than combined as in the original WAM. We also assume that stacks grow downwards, i.e. higher in the stack means older, lower means younger.

choice points but this is not visible for non-tabled execution. Moreover, unlike SLG-WAM, no freeze-registers are needed for CAT, nor a more complicated trail. In short, CAT allows introduction of tabling into a WAM like implementation without any performance overhead. Therefore, we believe that CAT is an attractive approach to incorporate tabling in any logic programming system.

In the next section, a brief introduction to tabling and the SLG-WAM is given and some terminology is set. We then explain CAT step by step in situations of increasing complexity in Section 3. All along the additions to WAM are introduced and memory management is discussed. Section 4 discusses a worst case behaviour of CAT and a possible remedy. Section 5 discusses in more detail the CAT implementation and the relation between SLG-WAM and CAT. Section 6 compares the performance of CAT and SLG-WAM in the context of XSB. We end with an overview of related and future work.

2 Tabling and SLG-WAM: Concepts and Terminology

Due to space limitations, we only present concepts and terminology of tabled evaluation and of the SLG-WAM which are necessary to make the paper reasonably self-contained. We assume the usual terminology of logic programming and refer the reader to [2] for issues related to SLG resolution and to [10] for a detailed description of the SLG-WAM. Also due to space limitations we restrict ourselves to mainly definite programs and we keep the presentation informal.

2.1 Basic overview of tabling

A *tabled program* is a program augmented (automatically or by the programmer) with tabling declarations of the form: `:- table` $p_1/n_1, \ldots, p_k/n_k$. where p_i is a predicate symbol and n_i is an integer. These declarations ensure that all queries to the predicate p_i of arity n_i will be executed using tabled evaluation (e.g. SLG resolution). Other predicates are implicitly assumed as *non-tabled* in which case SLD resolution is used for queries to these predicates. Slightly abusing terminology, we will speak of tabled subgoals as well as tabled predicates. Following SLG resolution we will consider two tabled subgoals to be the same if they are *variants* of each other; i.e. identical up to variable renaming; however note that this is an orthogonal issue to the design of SLG-WAM or CAT. Tabled subgoals which are encountered in the evaluation of a query against a program are persistently stored in a global data structure called a *subgoal table*. When a tabled subgoal, s, is called, a check must be made to see whether s exists in the subgoal table or not. This is the purpose of the SLG NEW SUBGOAL operation. If s is new, it is termed a *generator*, it is entered in the table and will use PROGRAM CLAUSE RESOLUTION to derive answers. Through the NEW ANSWER operation, the set of derived answers of s will also be recorded in a global data structure called the *answer table* of s. Note that there is a one-to-one correspondence between generators and answer tables. If, on the other hand, (a variant of) s already exists in the table, it will resolve against answers from its answer table. In this case, we call the subgoal a *consumer* of s. Answers are fed to the consumer one at a time through the ANSWER RETURN operation.

A basic concept in tabled evaluations is *completion* of (generator) subgoals and their associated answer tables. Informally, a subgoal s (and its answer table) is called *complete* if all its answers have been derived. On a slightly more operational level, through the SLG COMPLETION operation a subgoal can be determined as complete if all program and answer clause resolution has been performed for clauses of this subgoal. As there might exist dependencies between subgoals, it is often the case that subgoals cannot be determined complete on an individual basis, but their (mutual) dependencies also have to be taken into account. This suggests that the subgoal dependency graph DG has to be examined and *sets* of mutually dependent subgoals can be completed when they are involved in a *strongly connected component* Λ of DG that is *independent*: i.e. none of Λ's subgoals depends on a subgoal outside Λ. When all subgoals are completed, the evaluation has reached a fixpoint and stops.

2.2 Implementation of tabling in the SLG-WAM

As one of the examples in the introduction shows, tabling cannot be implemented using the pure depth-first search of the WAM: this is mainly due to the fact that the generation and consumption of answers are asynchronous events. This means that an abstract machine for tabling has to maintain or reconstruct execution environments of consumers until these have consumed all answers that are generated for the subgoals; i.e. consumers have to be retained until fixpoint or completion of the associated generators. Likewise, newly derived answers must be queued to resolve against subgoals which do not necessarily correspond to the current execution environment. SLG-WAM offers a particular way to implement these features.

Suspending and resuming consumers in the SLG-WAM The SLG-WAM implements tabling by *suspending* consumers when these have exhausted all answers currently in the table and *resuming* them when new answers have been derived for them. Suspension is performed in SLG-WAM by creating a *consumer choice point* to represent the suspended environment, freezing all stacks by setting the freeze registers to point to the current top, and then failing to a previous choice point without reclaiming any stack space; in particular, the freeze registers are not reset. Space is not reclaimed above these freeze registers until completion of the appropriate generator. Resuming, besides restoring the WAM registers to the values saved in the consumer choice point, uses the addresses and the values saved in a *forward trail* [13, 12] to restore variable bindings along the path to the suspended consumer; see [10] for exactly how this is done. An unconsumed answer is then returned to the restored consumer and execution continues by taking the forward continuation of the restored computation.

The purpose of freezing is that execution states of consumers are retained until the consumption of all their answers. So suspension interacts with completion: if upon consuming the last currently available answer of a subgoal, the subgoal cannot be determined complete, i.e. that it has all its answers, the consumer needs to be suspended so that its execution environment is available if new answers are generated for it. If, on the other hand, a consumer of a complete subgoal is encountered, a *completed table optimization* can be performed: suspension through freezing is not necessary and the consumer can backtrack through the answers in the table as if they were program clauses stored as facts.

Scheduling and incremental completion in the SLG-WAM A *scheduling strategy* determines when answers are returned to consumers: SLG allows for many different scheduling strategies each of which can have different performance characteristics; see [5]. XSB currently implements two different scheduling strategies called *batched* and *local evaluation*. Both of them are based on partitioning the subgoals encountered during the course of an evaluation into *scheduling components*. As this partitioning also interacts with and is influenced by completion we examine scheduling and completion together.

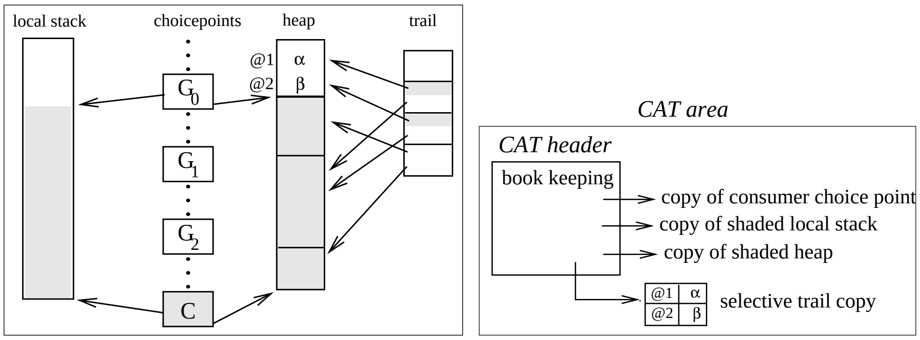
In definite programs, completion (determining fixpoint) can be postponed till the end of the evaluation. However, for the SLG-WAM to reclaim space and thus be effective on large programs a more fine grained, incremental completion is needed. To efficiently perform incremental completion, the SLG-WAM (and CAT) contains an area of memory new to the WAM, the *Completion Stack*. The completion stack can be seen as a restriction of the choice point stack to just the choice points for generators and is used to efficiently keep track of dependencies between subgoals and of scheduling components. Specifically, the completion stack maintains, for each subgoal s, a representation of the older (highest) generator subgoal s_L upon which s or any subgoal below s *may* depend. This subgoal is called the *leader* of its scheduling component. When s and all subgoals younger than s have exhausted all program clause resolution, s can be checked for completion. If s is the leader of its component, A, and thus does not depend on subgoals higher in the stack than itself, if all consumers of subgoals in A have consumed all answers, then s and all other subgoals in A can complete and the (possibly frozen) space that corresponds to subgoals of the component can be reclaimed. Otherwise, if the leader s_L is higher in the completion stack than s, then s may depend upon subgoals that appear higher than itself on the completion stack, and execution backtracks to the previous alternative without reclaiming any space. This is the main idea behind the implementation of incremental completion in the SLG-WAM (see [10] for more details).

In both batched and local evaluation, scheduling of ANSWER RETURN operations is based on (and limited to the subgoals of) the component that is on the top of the completion stack. This explains why these are called scheduling components. More specifically, the leader of the topmost component is responsible for checking whether all answers have been returned to all consumers of subgoals that it leads, and schedule ANSWER RETURN operations if unresolved answers exist for some consumer. This check, called fixpoint_check in [10], is needed independently of whether each generator schedules its consumers or not after performing all program clause resolution and checking whether it can complete. Scheduling of consumers on failing back to the leader is always possible as some scheduling strategies cannot determine fixpoint in a purely stack-based manner; see [10, 5] for why this is so. To appreciate the design of CAT, it is thus important to keep in mind that consumers of subgoals of a scheduling component may need to be resumed when execution has failed back to the leader.

3 A step by step introduction to CAT

3.1 A first approximation: no incremental completion

Let us first make the approximation that incremental completion is not performed: there is only one scheduling component (for all subgoals), and the *single leader* does *all* the scheduling (for all consumers of subgoals that it leads) on failing back to it. In this setting consider how execution goes: At the moment a consumer is found, its consumer choice point is installed on the choice point stack. Fig. 1(a) shows the stacks: generator choice points G_0 (which is the leader) up to G_2, followed by the consumer choice point C. The vertical dots in between these choice points and above G_0 denote possible Prolog choice points, not related to execution of tabled predicates. The heap is shown segmented according to the tabling choice points and so is the trail. The same segmentation is not shown in the local stack, as it is a spaghetti stack. From the trail, some pointers point to cells older than the segment between G_0 and G_1: these cells have addresses @1 and @2 in the picture and the values in these cells are α and β.



(a) Stacks after creation of a consumer choice point: shaded parts are copied.

(b) The CAT area with selective trail.

Fig. 1. Conservative saving of a consumer state (without incremental completion).

Fig. 1(b) shows the information as saved by CAT: there is a (dynamically allocated) frame of fixed size which we name the *CAT header*. Apart from some bookkeeping fields, it contains a pointer to areas which contain copies of the shaded parts of the stacks: for heap and local stack, these shaded areas of Fig. 1(a) are copied as is; they are the part of heap and local stack created between the creation of G_0 and the consumer C. From the choice point stack, CAT only needs to save the consumer choice point: the justification is that at the moment C is scheduled to consume its answers, all the Prolog choice points as well as the non-leader generator choice points will have exhausted their alternatives, and will have become redundant. This also means that when a consumer choice point is reinstalled, this can happen immediately below the leader G_0.

CAT copies the trail selectively as well: since CAT copies all of the heap between G_0 and C, there is no need to save the trail entries that point into this region and similarly for the trail entries that point to the saved part of the local stack. On the other hand, just saving the trail entries pointing to the older region of the heap (and local stack), is not enough to reinstall the consumer, because backtracking (up to G_0) will have undone the binding of say cell @1 to α. It means that CAT must also save the value α, or in general the current value of heap (and local stack) cells older than G_0 and pointed to by trail entries younger than G_0. Fig. 1(b) shows in more detail a saved trail. We call the CAT header together with the saved stacks, a *CAT area.*

After CAT has copied the consumer state, it removes the consumer choice point from the stack and activates a general failure in the WAM. Forward execution might then create other consumers (and CAT areas). Execution will eventually fail back to the leader G_0.

In this setting, after exhausting all alternatives of G_0, the leader must also make sure that the consumers consume their answers. The scheduling and restoration of consumers happens as follows: For each consumer, C, the saved portion of the heap and the local stack is copied back to its original place. Also, the saved values on the trail are reinstalled and the saved consumer choice point is copied just below G_0. This reinstalled consumer choice point can now start consuming answers from the tables as in SLG-WAM. After all currently available answers have been consumed, CAT must also update the *LastAnswer* field of the consumer choice point in the corresponding CAT area. Note that after reinstalling the consumer, the trail and choice point stack are in general smaller than at the moment of saving the consumer state. See Fig. 2(a).

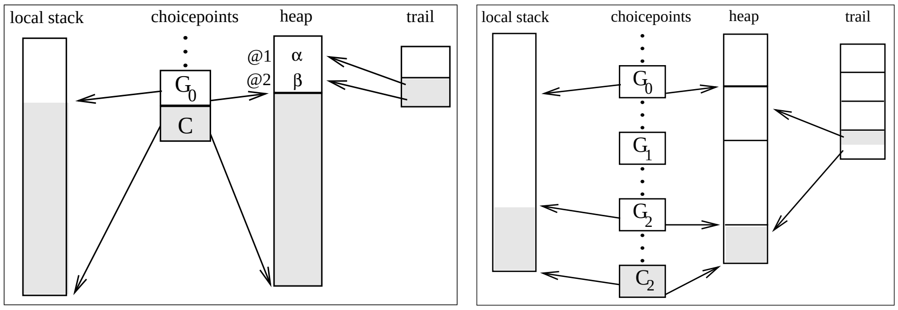
In the following sections we will refine CAT, but here already, we have laid out the basics for understanding CAT: the state of a consumer is saved by copying it; this copy consists of the parts of the heap and local stack between the consumer and a generator, a more selective trail copy and only the consumer choice point.

3.2 Adding incremental completion based on fixed leaders

In the previous section, we assumed that completion was non-incremental and all subgoals belonged to one scheduling component led by a single leader. As mentioned in Section 2.2, for definite programs, this is a valid scheme for CAT as scheduling and completion can always be postponed till the end of the evaluation. However, as in the SLG-WAM, it is more efficient to perform incremental completion and have smaller scheduling components because then (1) CAT would copy and reinstall a smaller part of the stacks, and (2) subgoals can complete and free the CAT areas of their consumers earlier. Indeed, look at the execution of `?- p`$_g$`(X).` against the following program:

```
 :- table p/1, q/1.
p(1) :- q_g(Y).      q(3) :- q_c(Z).
p(2).                q(4).
```

The answers of q/1 do not depend on the answers of p/1, so a generator of q/1 can be a leader and form a scheduling component in itself. It means that at the moment $\mathtt{q}_c$(Z) is called, its leader is not $\mathtt{p}_g$(X), but the subgoal $\mathtt{q}_g$(Y), and the consumer $\mathtt{q}_c$ can always be scheduled on failing back to the generator $\mathtt{q}_g$. So, in order to reinstall the state of $\mathtt{q}_c$, it is enough to copy stacks between $\mathtt{q}_c$ and the leader $\mathtt{q}_g$. Fig. 2(b) shows a variant of Fig. 1(a): the consumer is C_2 and its leader is G_2; the shaded areas to be copied are smaller than before. Note, however, that this schema is not practical because it assumes that the fixed leaders of scheduling components are known beforehand.



(a) The stacks just after the consumer has been restored.

(b) The leader is closer to the consumer: the copy is smaller.

Fig. 2. Reinstallation and incremental saving of consumer states.

3.3 A coup: the leader changes

The principle of the previous two sections was: save consumer state up to the leader generator that might or will schedule the consumer. Even in cases where the fixed leaders are not known in advance, this works well until there is a change of leader: in practice a change of leader happens often. The query ?- $\mathtt{p}_g$(X). executed against the following program shows such a coup:

```
:- table p/1, q/1.
p(X) :- q_g(Y), fail. q(3) :- q_c(X).
p(1).                 q(4) :- p_c(X).
```

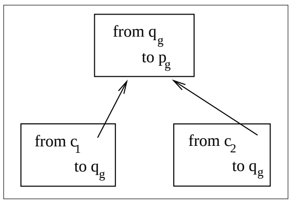
When the consumer $\mathtt{q}_c$ is saved, its leader (as maintained by dependencies kept in the completion stack) is the generator $\mathtt{q}_g$. Indeed, at that point, it is not yet known that the answers of q will depend on answers of p. Later, at the moment the consumer $\mathtt{p}_c$ is created, a coup takes place: the generator $\mathtt{q}_g$ is no longer the leader of a scheduling component and $\mathtt{p}_g$ has become the leader of $\mathtt{q}_c$. It means that in the future $\mathtt{q}_c$ might need to be restored by $\mathtt{p}_g$, so the saved state of $\mathtt{q}_c$ should at restoration time contain also the part of the stacks between $\mathtt{p}_g$ and $\mathtt{q}_g$. We could eagerly — at the moment of the coup — save this missing part and add it to the CAT area of $\mathtt{q}_c$. The alternative is to wait until execution is about to backtrack over generator $\mathtt{q}_g$: then, CAT saves the increment needed for $\mathtt{q}_c$ (i.e. the part of the stacks between $\mathtt{p}_g$ and $\mathtt{q}_g$) and links it up to the CAT area of q_c. The advantage of waiting until this moment to save the increment, is that there might be other consumers in the same need for an extension of their CAT area, and thus the part of the stacks between p_g and q_g can potentially be shared between all these consumers, instead of copying the same part for every one of them. This will become more clear below.

3.4 Incremental copying of consumer states

The previous section showed that in the absence of precise information about leaders and scheduling components, there is possibly a need for extending the saved state whenever the leader changes. Now consider the following program, which differs from the one before only by an extra (last) clause for `q/1`:

```
:- table p/1, q/1.    q(3) :- q_c1(X).
p(X) :- q_g(Y), fail. q(4) :- p_c(X).
p(1).                 q(5) :- q_c2(X).
```

The two consumers for `q/1` have been given an index for ease of reference. At the moment consumer q_{c_1} is saved, copying happens up to the leader which is then q_g; when consumer q_{c_2} is saved, the coup has happened already, meaning that for q_{c_2}, we copy up to p_g, the new leader. Later when backtracking happens over the generator q_g, we copy the part between q_g and p_g so that this can be linked to consumer q_{c_1}. Note though that q_{c_2} already contains that part of the stacks ! So, we have copied twice the same information from the stacks (the part between p_g and q_g) and it is very difficult to avoid this in the schema which copies a consumer state up to its current leader. Since we have already the mechanism to link parts of saved states, we can use it in a more systematic way as follows: instead of saving a consumer state up to its leader, CAT always saves up to the closest generator G. When execution fails back to this generator, all consumers younger than G have copied all information needed for their restoration: they can be scheduled to consume their answers. If G is a leader of a scheduling component, on reaching fixpoint, completion can occur and the space for the CAT areas can be freed. Otherwise, since backtracking over this generator will occur, a new increment up to the previous generator is linked to all the consumers that need it. Applied to the above example, it means that the double copying of the old schema, does not happen anymore. This sharing of CAT areas between consumers is also shown schematically in the following rough picture. This incremental saving of consumer states, is the one finally implemented in CAT: it performs less copying by improving sharing of consumer states. CAT also allows for more flexible scheduling strategies since now even non-leaders can schedule consumers; moreover in the context of CAT, it is natural that a generator can schedule all consumers with a saved state that reaches up to this generator, not just its own.



4 The Main Problem of CAT and a Remedy

Consider the program:

```
:- table p/1.
p(X) :- produce(LargeTerm), p_c(Z), X = 2.
p(1).
```

The query `?- p_g(X).` produces the answers `p(1)` and `p(2)` and the consumer p_c is saved and restored once. `produce/1` is a computation which produces a large term as output argument. In the setting above, the consumer state contains this large term, so it is copied on saving and copied on restoring the consumer. So the CAT area for a consumer can be arbitrarily large, meaning that both saving and restoring a consumer can take arbitrarily long. The SLG-WAM does not suffer from this problem: freezing the stacks is a constant time operation and the cost of restoring a consumer in SLG-WAM is related to the trail. Since it is easy to construct examples in which consumers have to be restored arbitrarily often, it follows that CAT can be made to perform arbitrarily worse than SLG-WAM. However, this bad behaviour of CAT was not observed in any of the programs we have so far used tabling for. [2]

As it happens in the above example, this large term is not used in the continuation of the consumer, which means that copying it on saving the consumer was unnecessary. In general one can say that none of the heap entries which are not reachable for the continuation of a consumer, need to be saved. So, it seems worthwhile to perform a garbage collection that is restricted to the part of the heap to be copied, just before saving a consumer state. This is explored in more detail in [4] which also deals with a similar compaction of the local stack.

5 Implementation of CAT and Relation to SLG-WAM

We have implemented CAT within XSB; XSB itself implements SLG-WAM which freezes stacks as a means to save a consumer state. This means that XSB has a more complicated trail, trail test, setting of top-of-stack, etc. As a CAT prefers warmth, we have first *heated up* XSB, by removing all the freeze related code and replacing it with the plain WAM equivalent. In this warm version of XSB, we have then implemented CAT, reusing from XSB everything related to the implementation of incremental completion and the access and storing mechanisms for tabling of [8]. Besides adding the incremental CAT copying and restoration described in Section 3.4 and slightly modifying the scheduling (as will be explained below), only instructions related to tabling (tabletry, answer_return, ...) which do not belong to the WAM, were changed. So, now there exist two versions of XSB: one that implements SLG-WAM and one that implements CAT. These will form the basis of a comparison between SLG-WAM and CAT. Since to our knowledge, there exists no other implementation of SLG-WAM (nor of CAT) we will henceforth refer to the two mechanisms as if they were implementations.

---

[2] The bad figures for read_o in Table 1 and 2 have another reason: see Section 6.

Current similarities and differences with the SLG-WAM In our first version of CAT we have retained the incremental completion algorithm of the SLG-WAM. As mentioned, in definite programs, completion can always be postponed and it is mainly used for space reclamation. As the SLG-WAM freezes the stacks at the top, non stack-based selective completion (and space reclamation) of subgoals is not possible without fragmenting the stacks and thus requires stack compaction. As this is probably costly, the SLG-WAM always completes the component in which the youngest generator belongs: therefore, the SLG-WAM uses a completion algorithm based on *approximate* subgoal dependencies. This algorithm is known to trap subgoals in scheduling components (see [10]) and may arbitrarily postpone their completion despite the fact that they are independent of other subgoals. CAT, as a true feline, is much more flexible in completion because reclamation of the CAT areas is not based on compaction. We plan to add an exact completion algorithm to our CAT, but for fairness, in this paper we compare CAT and SLG-WAM under the same completion algorithm.

Unlike the SLG-WAM, CAT reclaims the trail and choice point stack even before completion of a component.[3] In particular, it reclaims generator choice points for non-leaders when these have exhausted program clause resolution. This means that the substitution factor variables (an optimization of [8]) cannot be stored in the choice point stack as in the SLG-WAM, but in a place that survives backtracking until completion: we have opted for the heap. As another small technical point, since non-leader generators are reclaimed by CAT, local scheduling is implemented by creating a CAT area for generators that are not leaders. This is in accordance with the definition of local evaluation that specifies that these generators behave as consumers (cf. [5]).

Finally, as noted, CAT allows for more flexible scheduling algorithms, in particular more fine-grained ones. However, the current version only performs scheduling on failing to the leader in a manner similar to the fixpoint_check of the SLG-WAM. A difference with the SLG-WAM is that scheduling decisions are taken more often by CAT since CAT can reinstall only one consumer at a time, while SLG-WAM can schedule several consumers in one go.

Extending CAT to normal logic programs The XSB implementation of the SLG-WAM evaluates programs according to the well-founded semantics. There is really nothing that prevents CAT to also work for this class of programs as the handling of negation is an orthogonal issue. The same low-overhead (around 1%) mechanisms that the SLG-WAM uses to maintain exact subgoal dependencies [10], to detect and break cycles through negation can be combined with CAT as well. Space limitations prohibit a full discussion but we simply note that the handling of negative (tabled) literals in CAT is analogous to the handling of consumer subgoals: the execution state of negative literals is also preserved in a CAT area by copying. A small difference is that a *negation suspension frame* [10] rather than a consumer choice point is copied and that this frame is reinstalled *once* and *at the place of the generator choice point* (upon its completion) rather than immediately below it.

[3] Actually, it reclaims all stacks, but parts of heap and local stack stay in CAT areas.

6 Performance Evaluation

As expected, CAT performs better than SLG-WAM in Prolog code; around 10% according to our measurements (see also [10]). Also, CAT and SLG-WAM have indistinguishable performance in artificial tabling benchmark programs from the database community like transitive closures over chains, cycles and trees of various lengths and same generation over cylinders (cf. [11, 10, 5]). So we compare CAT and the SLG-WAM on more realistic sets of programs from an application area where tabling has been proven worth having in a general purpose logic programming system: abstract interpretation. All measurements were conducted on an Ultra Sparc 2 (168 MHz) under Solaris 2.5.1. Times are reported in seconds, space in KBytes. Space numbers measure the maximum use of the stacks (for SLG-WAM) and the total of max. stack + max. CAT area (for CAT).

6.1 A benchmark set dominated by tabled execution

The first benchmark set is taken from [3]: the programs perform type analysis by program abstraction and execution of the abstracted program under tabled evaluation. Tabling is used both for termination, efficient storage of the analysis results and to avoid redundant subcomputations in the domain-dependent abstract operations (see [3]). With the exception of a few utility predicates like `append/3`, all other predicates are tabled in this benchmark set and the size of the table space (not shown) is quite large: this set of benchmarks programs is heavily dominated by tabling operations.

Tables 1 and 2 show time and space performance of the analysis under the two scheduling strategies of XSB. On this set of programs, SLG-WAM performs more or less the same time-wise with batched (B) and local (L) strategy with a very noticeable advantage for the local strategy in space consumption as its scheduling components are tighter. CAT under local scheduling performs slightly better than SLG-WAM in time and slightly worse in space. CAT under batched scheduling is slower than the SLG-WAM for the last three benchmarks by 15–60%: these are also the benchmarks for which CAT uses more than 10 times more space than SLG-WAM. This behaviour is mostly due to the approximate completion algorithm that is used. In this benchmark set, a high percentage of the tabled subgoals, and more specifically the abstract operations, is semi-det: i.e. produces at most one answer. Because completion is based on an approximation of subgoal dependencies, these semi-det subgoals often get trapped in an approximate scheduling component and cannot be completed on

	cs_o	cs_r	disj_o	gabriel	kalah_o	peep	pg	read_o
SLG-WAM(B)	0.23	0.45	0.13	0.17	0.15	0.44	0.12	0.58
CAT(B)	0.22	0.41	0.13	0.15	0.14	0.50	0.15	0.92
SLG-WAM(L)	0.23	0.43	0.13	0.16	0.16	0.42	0.12	0.61
CAT(L)	0.22	0.42	0.12	0.15	0.14	0.40	0.11	0.55

Table 1. Time performance of CAT vs. SLG-WAM under batched & local scheduling.

	cs_o	cs_r	disj_o	gabriel	kalah_o	peep	pg	read_o
SLG-WAM(B)	9.7	11.4	8.8	20.6	40	317	119	512
CAT(B)	13.6	19.4	11.7	45.3	84	3836	1531	5225
SLG-WAM(L)	6.7	7.6	5.8	17.2	13.3	19	15.8	93
CAT(L)	7.9	10.7	7.1	29.5	12.5	17	23.5	246

Table 2. Space performance of CAT vs. SLG-WAM under batched & local scheduling.

their own. As it is not known whether new answers will be derived for these subgoals, their consumers have to suspend (and create a CAT copy) rather than use the completed table optimisation (see Section 2.2). Some extra measurements for the read_o program shed more light: under batched scheduling 3112 consumers are saved (max. CAT area of 5208 KBytes) and only 192 times a consumer is restored (totaling 371 KBytes) to consume new answers; in contrast, the corresponding numbers for local scheduling are 264 consumers saved and 224 restorations (240 and 371 Kbytes respectively — the bigger restoration is due to sharing of the saved space by incremental copying, but not by restoration). With the current approximate completion algorithm, under batched scheduling, CAT performs worse than SLG-WAM. We strongly believe that for this benchmark set CAT will be more competitive to SLG-WAM (under batched scheduling) if completion is based on exact subgoal dependencies (cf. also Section 5).

6.2 A more realistic mix of tabled and Prolog execution

The second set of benchmarks is taken from [6]. These programs perform abstract interpretation. The abstract operations are implemented in Prolog and occupy a high percentage of the total execution time (around 75–80%). Regardless of the issue SLG-WAM vs. CAT, only the local scheduling makes sense in this program set because the analyses are based on an abstract least upper bound operation, resulting in a very poor performance for the batched strategy (see also [5]). Tables 3 and 4 compare SLG-WAM and CAT in time and space. Overall, CAT performs time-wise slightly better (5–20%) in this set. It performs considerably better (25–140%) than SLG-WAM in space with only two exceptions.

	akl	color	bid	deriv	read	browse	serial	rdtok	boyer	plan	peep
SLG-WAM	1.48	0.67	1.11	2.56	9.64	32.6	1.17	3.07	10.02	7.61	9.01
CAT	1.24	0.62	0.97	2.50	9.56	32.2	0.83	2.75	9.96	6.38	8.54

Table 3. Time performance of CAT vs. SLG-WAM.

7 Related Work

There is an analogy between the SRI-model [12] for implementing OR-parallelism and MUSE [1] on one hand, and the SLG-WAM for implementing tabling and CAT on the other: like the SLG-WAM, the SRI-model has a complicated management of the stacks and switching from one worker to another — the analogue of suspending one consumer to resume another one — uses a trail structure that is more complicated than the WAM trail, because bindings have to be undone as

	akl	color	bid	deriv	read	browse	serial	rdtok	boyer	plan	peep
SLG-WAM	998	516	530	472	5186	9517	279	1131	2050	1456	1784
CAT	552	223	206	486	8302	7847	227	821	1409	1168	1373

Table 4. Space performance (in KBytes) of CAT vs. SLG-WAM.

well as reinstalled. Like MUSE, CAT avoids complicated stacks by copying the portion of the stacks that is particular to a consumer or in the case of MUSE a worker. We believe this analogy is so strong, that although not conscious at the time, we must have been influenced by our knowledge of MUSE when getting the idea for CAT. [1] notes that the overhead of copying is small compared to all other work to be performed and our experience with CAT is similar.

In [9] a design for combining tabling and or-parallelism was presented. The implementation of tabling was based on the SLG-WAM, the only available WAM-based model of implementing tabling at that time, while or-parallelism was supported through environment copying as in the MUSE model. In view of the above similarity between CAT and MUSE, we believe that CAT also offers a more natural way of combining tabling and or-parallelism as the same basic machinery can be used for satisfying the implementation requirements of both forms of suspend/resume mechanisms.

Another related technique based on copying is described in [7]. It provides a set of library functions for introducing backtracking in C programs: the parts of the C-stack that must become active again after failure are copied incrementally. This is achieved by changing at run time the return address in activation records. In contrast, in CAT the zones to be copied are delimited by invocations of tabled predicates, so that by compiling these with special instructions, such a change is performed effectively at compile time.

8 Conclusion and Future Work

Since any logic programming system can benefit from having tabling, it has been our long term goal to make tabling more accessible and more attractive to add to existing systems. It always seemed to us that the main obstacle to this were some aspects of the SLG-WAM: in particular the machinery that the SLG-WAM imposes on the WAM to implement the suspend/resume mechanism needed for tabling. This machinery is quite complicated and unnecessarily slows down the underlying basic system. The other issues in tabling implementations — the tabling data structures and the scheduler — were always orthogonal and do not affect the underlying implementation. We have shown that CAT is a true alternative to SLG-WAM for implementing the control that tabling requires. CAT is simpler to add and it has no influence on the efficiency of the underlying implementation. In addition, CAT helps in reasoning about reachability of objects in the execution tree and it seems that it allows more easily for more flexible scheduling strategies. Empirical tests show that most of the benchmark programs are not slowed down by the CAT technique compared to the implementation of SLG-WAM in XSB and that under the local evaluation strategy, the memory consumption is most often better than under SLG-WAM. This might come as a surprise, but the simplicity of CAT is the key to its performance.

However, to fully achieve our goal, at least the following are needed: a clear definition of the interface between the tabling components and a general purpose LP system; a complete integration of CAT in the memory management of a LP system (a forthcoming paper deals with this [4]); and a better understanding of the interaction of CAT with scheduling strategies.

Acknowledgements

The second author is supported by a K.U. Leuven junior scientist fellowship.

References

1. K. A. M. Ali and R. Karlsson. The Muse approach to OR-parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, Apr. 1990.
2. W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, Jan. 1996.
3. M. Codish, B. Demoen, and K. Sagonas. Semantics-Based Program Analysis for Logic-Based Languages using XSB. *Springer International Journal of Software Tools for Technology Transfer*, Aug./Sept. 1998. To appear.
4. B. Demoen and K. Sagonas. Memory Management for Prolog with Tabling. Technical Report CW 261, K.U. Leuven, Apr. 1998. Submitted for publication.
5. J. Freire, T. Swift, and D. S. Warren. Beyond Depth-First Strategies: Improving Tabled Logic Programs through Alternative Scheduling. *Journal of Functional and Logic Programming*, 1998(3), Apr. 1998.
6. G. Janssens and K. Sagonas. On the Use of Tabling for Abstract Interpretation: An Experiment with Abstract Equation Systems. In *Proceedings of TAPD-98: Tabulation in Parsing and Deduction*, pages 118–126, Paris, France, Apr. 1998.
7. P.-E. Moreau. A choice-point library for backtrack programming. In K. Sagonas, editor, *Proceedings of the JICSLP-98 Workshop on Implementation Technologies for Programming Languages based on Logic*, Manchester, U.K., June 1998.
8. I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Tabling Mechanisms for Logic Programs. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 687–711, Tokyo, Japan, June 1995. The MIT Press. Extended version to appear in the JLP.
9. R. Rocha, F. Silva, and V. S. Costa. On Applying Or-Parellelism to Tabled Evaluations. In *Proceedings of the First International Workshop on Tabling in Logic Programming*, pages 33–45, Leuven, Belgium, June 1997.
10. K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM TOPLAS*, 20, 1998. To appear.
11. K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994. ACM.
12. D. H. D. Warren. The SRI Model for OR-Parallel Execution of Prolog — Abstract Design and Implementation Issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, San Francisco, California, Sept. 1987. IEEE CS Press.
13. D. S. Warren. Efficient Prolog memory management for flexible control strategies. In *Proceedings of the 1984 Symposium on Logic Programming*, pages 198–202, Atlantic City, New Jersey, Feb. 1984. IEEE Computer Science Press.