ELSEVIER

Future Generation Computer Systems 16 (2000) 809–830

FGCS
FUTURE GENERATION COMPUTER SYSTEMS

CHAT: the copy-hybrid approach to tabling☆

Bart Demoen[a], Konstantinos Sagonas[b,*]

[a] *Department of Computer Science, Katholieke Universiteit Leuven, B-3001 Heverlee, Belgium*
[b] *Computing Science Department, Uppsala Universitet, S-751 05 Uppsala, Sweden*

Accepted 24 May 1999

Abstract

The copying approach to tabling (CAT) is an alternative to SLG-WAM and based on incrementally copying the areas that the SLG-WAM freezes to preserve execution states of suspended computations. The main advantage of CAT over the SLG-WAM is that support for tabling does not affect the speed of the underlying abstract machine for strictly non-tabled execution. The disadvantage of CAT as pointed out in a previous paper is that in the worst case, CAT must copy so much that its tabled execution becomes arbitrarily worse than that of the SLG-WAM. Remedies to this problem have been studied, but a completely satisfactory solution has not emerged. Here, a hybrid approach is presented: abbreviated as CHAT. Its design was guided by the requirement that for non-tabled (i.e. Prolog) execution no changes to the underlying WAM engine need to be made. CHAT not only combines certain features of the SLG-WAM with features of CAT, but also introduces a technique for freezing WAM stacks without the use of the SLG-WAM's freeze registers that is of independent interest. This article describes only the basic CHAT mechanism which allows for programs that perform arbitrarily worse than under the SLG-WAM. However, empirical results indicate that even basic CHAT is a better choice for implementing the control of tabling than SLG-WAM or CAT.

Keywords: Abstract machine; Prolog; Tabling; WAM

1. Introduction

In [5], we developed a new approach to the implementation of the suspend/resume mechanism that tabling needs: the 'copying approach to tabling' abbreviated as CAT. The essential characteristic of the approach is that preserving the execution state of suspended computations through freezing of the WAM stacks (as in the SLG-WAM [10]) was replaced by a selective and incremental copying of these states. One advantage is that this approach to implementing tabling does not introduce new registers, complicated trail or other inefficiencies in an existing WAM: CAT does not interfere at all with Prolog execution. Another advantage is that CAT can perform completion and space reclamation in a non-stack based manner without need for memory compaction. Finally, experimentation with new scheduling

☆A short preliminary version of this article was presented at the Workshop on Principles of Abstract Machines, held in conjunction with PLILP/ALP'98, in Pisa, Italy, September 1998.

*Corresponding author.
E-mail addresses: bmd@cs.kuleuven.ac.be (B. Demoen), kostis@csd.uu.se (K. Sagonas)

0167-739X/00/$ – see front matter
PII: S0167-739X(99)00092-8

strategies seems more easy within CAT. On the whole, CAT is also easier to understand than the SLG-WAM. The main drawback of CAT, as pointed out in [5], is that its worst case performance renders it arbitrarily worse than the SLG-WAM: CAT might need to copy arbitrary large parts of the stacks; the SLG-WAM's way of freezing in contrast is an operation with constant cost. Although this bad behavior of CAT has not shown up as a real problem in our uses of tabling (see [5] and the performance section of the current article), in [6] we have described a partial remedy for this situation. Restricted to the heap, it consists of performing a minor garbage collection while copying; i.e. preserve only the useful state of the computation by copying just the data that are used in the forward continuation of each consumer (a goal which resolves against answers from the table). The same idea can be applied to the local (environment) stack as well. [6] contains some experimental data which show that this technique is quite effective at reducing the amount of copying in CAT. This is especially important in applications which consist of a lot of Prolog computation and few consumers. However, even this memory-optimized version of CAT suffers from the same worst case behavior compared to SLG-WAM.

We therefore felt the need to reconsider the underlying ideas of CAT and SLG-WAM once more. In doing so, it became quite clear that CAT and SLG-WAM represent the two extremes of a wide spectrum of possible implementations of tabling: namely CAT being a tabled abstract machine totally based on copying and SLG-WAM one totally based on sharing of execution states. Once this was realised, it also became clear that all sorts of hybrid implementation schemes are possible, e.g. one could copy the local stack while freezing the heap, trail and choice point stack, etc. However, we are convinced that the guiding principle behind any successful design of a (WAM-based) tabling implementation must be that the necessary extensions to support tabling should not impair the efficiency of the underlying abstract machine for (strictly) non-tabled execution, and that support for tabled evaluation should be possible without requiring difficult changes to the WAM: CAT was inspired by this principle and provides such a design. CHAT, the hybrid sharing and copying tabling abstract machine we present here enjoys the same property.

If the introduction of tabling must allow the underlying abstract machine to execute Prolog code at its usual speed, we have to be able to preserve and reconstruct execution environments of suspended computations without using SLG-WAM's machinery; in other words we have to get rid of the freeze registers and the forward trail (with back pointers as in the SLG-WAM). The SLG-WAM has freeze registers for heap, trail, local, and choice point stack. These are also the four areas which CAT selectively copies. Section 6 is the main section of the article and describes what CHAT does for each of these four areas. We start, however, by a brief introduction to the idea of tabling and the issues that have to be addressed by the abstract machine when tabling is incorporated in the execution model of a logic programming language such as Prolog (Section 2). This introduction is followed by some terminology in Section 3 which is used in later sections of this article. The next two sections, present a brief account of the main ideas of the SLG-WAM (Section 4) and of CAT (Section 5), because CHAT is a mixture of ideas from the SLG-WAM and from CAT. Section 7 shows best and worst cases for the basic CHAT design compared to the SLG-WAM. Section 8 discusses the combinations possible between CHAT, CAT, and SLG-WAM. Section 9 shows the results of some empirical tests with our implementation of CHAT and Section 11 concludes.

2. Tabling in logic programming

Tabling in logic programming [14] is similar to memoization in functional languages: the idea there is to remember the first invocation of each function call (henceforth referred to as a *producer* or *generator*) and its computed result, so that subsequent identical function calls (referred to as the *consumers*) can get at the remembered answer without repeating the computation. Tabling can be applied in the context of general programming and is also related to memo-ing in artificial intelligence [9]. See e.g. [3] for a discussion of issues related to tabulation in an essentially functional programming context; more references to the origin of tabling can also be found there. Although tabulation is a powerful instrument in writing efficient programs, it must be realised that it can change the complexity of a program in an arbitrary way, both as a speedup and as a slowdown. Logic programming languages deal with resolving

(sub)goals rather than evaluating function calls to obtain their (single) result. Thus, tabling in logic programming differs from memoization in the functional setting in the following respects:

1. Subgoals can contain variables as opposed to being completely instantiated as in the functional setting. As a result, the notion of "identical call" becomes more flexible, as it can be given sense as "identical up to renaming of the variables" (tabling based on variance) or as "is subsumed by" a previously encountered subgoal (tabling based on subsumption).
2. While in a functional setting, a call can return only one result, in logic, a subgoal can succeed potentially more than once (with a different answer substitution each time) or even fail (i.e. have no answers). To preserve the semantics, a subsequent identical subgoal must consume all the answers produced by the first one (the generator).
3. Since in general it is not known how many answers a subgoal will produce, it is not a priori clear when the generator really has found all its answers. This means that some sort of saturation detection (known as *completion detection* of subgoals in the context of tabling) must be implemented by the abstract machine.
4. Subgoals can also occur in a context involving negation or aggregation (e.g. in a context where all solutions need to be found). In these contexts, evaluation in general cannot proceed until the subgoals meet the completion criterion mentioned above.

Whereas the implementation of memoization in functional programming is — at least conceptually — trivial, it is the combination of the second and third points above that makes the implementation of tabling in a logic programming setting difficult: indeed, the situation can occur that a consumer must necessarily start consuming answers before the generator has finished producing all its answers; i.e. before the generator is *completed*. For a functional program, this would indicate a loop in the program and tabling helps in detecting it. On the other hand, in logic programming, such a situation can be given sense beyond looping with tabling. In the remaining part of this section, we will show the issues involved with a series of examples of increasing complexity. To indicate that a subgoal p is a generator (respectively, a consumer), we will denote it by p_g (or) p_c).

2.1. A completed generator and then a consumer

Consider the program P_1 below (for some appropriate implementation of `find_first_N_primes/2`) where the `table` declaration denotes that `prime/2` is a tabled predicate (a predicate whose subgoals and answers will be stored in a table). Throughout this article, all other predicates are implicitly assumed to be evaluated as in Prolog.

```
:-table prime/2.
prime(N,Prime) :-
  find_first_N_primes(N,Prime).
```

Consider the query $\texttt{?-prime}_g\texttt{(1000,P)}$. A lengthy computation will produce by backtracking the sequence of answers: `P=1, P=2, P=3, P=5, ... , P=7907`. Since the subgoal `prime(1000,P)` never occurred before, this subgoal is a generator and the subgoal together with its answer substitutions is stored in the table for `prime/2`. Once no more answers can be generated for this subgoal, the table for this subgoal gets completed. Asking the same (up to variable renaming) query again now avoids the lengthy computation and simply picks up the answers from the table. Since the second query is asked after the first one was finished, this operation is relatively straightforward; for example, the tabled abstract machine can implement it through backtracking.

2.2. A generator in conjunction with a consumer

Consider now the program P_2:

```
:-table p/1.
p(X) :-(X=1 ; X=2).
```

and the query $\texttt{?- p}_g\texttt{(X)}, \texttt{p}_c\texttt{(Y)}$ The first answer from the generator can be consumed by the consumer before any other answer is generated; this strategy of returning answers called *batched scheduling strategy* tries to mimic

the evaluation strategy of Prolog. The consumer $p_c(Y)$ that is created at the moment there is only the first answer available, consumes this answer and then backtracking will bring the computation back to the generator. Backtracking should, however, not destroy the consumer's state completely as the consumer should remember which answers it has consumed already. Otherwise, the consumer will consume too few answers, or possibly some answers more than once. This means that a consumer subgoal must have its own state and identity so that it can be made to continue with subsequent answers and in an execution context where the goals to the left of the consumer are already resolved. This reasoning also shows that the state of a consumer cannot be destroyed earlier than the moment of completion of its corresponding generator.

Note that the query can also be transformed to

$$?- (p_g(X), \texttt{fail;true}), p_c(X), p_c(Y).$$

where the purpose of the disjunct within the parentheses is to generate all answers of the `p(X)` subgoal and in effect complete its table; only then execution continues with the rest of the computation. This new — and equivalent — query no longer contains an incomplete generator that appears in conjunction with one of its consumers. In fact, such a conceptual transformation is the main point of the *local scheduling strategy* (see [8]) which performs it dynamically. It could then appear that it is enough to consider finished generators and consumers as in the previous section. However, the following shows by example that this is not true.

2.3. A generator which depends on its consumer

Typically Prolog goes in an infinite loop for this kind of programs where a generator depends on some of its consumers. A tabled program, P_3, exhibiting such a phenomenon is shown below.

```
:-table p/1.
p(1).
p(X):- p_c(Y), X is 2 * Y, X < 20.
p(X):- p_c(Y), X is 3 * Y, X < 20.
?- p_g(X).
```

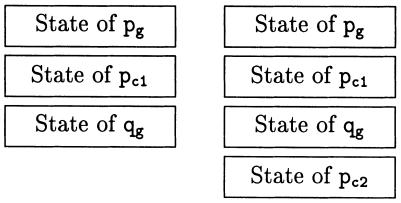
This tabled program produces the set of answer substitutions $X \in \{1,2,4,8,16,3,6,12,9,18\}$, while if executed under a Prolog-style evaluation (without tabling `p/1`), the query will run into an infinite loop after producing the sequence of answer substitutions `X=1, X=2, X=4, X=8`, and `X=16`.

The previously shown transformation does not apply anymore: in order for the generator to produce the answer `2` or `3`, the consumer must have consumed the answer `1` first! In other words, the table for the subgoal `p(X)` cannot be completed before some of its consumers have consumed answers from it.

2.4. Multiple generators which depend on each other

The examples so far presented situations where only one generator is present. In general, however, a tabled evaluation might encounter more than one generator. When generator subgoals are not mutually dependent, evaluation could proceed by finding all answers of the *independent generator* first (at the same time possibly returning them to its consumers), complete this subgoal (meaning that the generator's state and the state of all its consumers can be safely forgotten by the backtracking mechanism of the abstract machine), and only then continue with the remaining subgoals. This process can be repeated until all subgoals are completed and all their consumers have consumed all answers from the corresponding table. For the query $?- p_g(X)$, the tabled program P_4 below provides such an example:

```
:-table p/1.                    :-table q/1.
p(X):-p_c(Y), X is 3 * Y, X<20. q(1).
p(X):-q_g(X).                   q(X) :- q_c(Y), X is 2 * Y, X < 20.
```



(a) Before the coup (b) After the coup

Fig. 1. Stacks during execution of P_5.

Upon encountering the generator for `q(X)`, the tabled abstract machine can completely evaluate this subgoal (as the generator q_g and the consumer q_c do not depend on a `p` subgoal) and only then start returning answers to the consumer $p_c(Y)$. Furthermore, note that in a stack-based abstract machine like the WAM, the states of q_g and q_c are not intermixed on the execution stacks with states of `p`'s subgoals. As a result, states corresponding to `q(X)`'s generator and consumer can upon (or after) `q(X)`'s completion be safely reclaimed by backtracking similar to that of the WAM.

There are cases, however, where subgoals are mutually dependent as is the case in the following modification of program P_4. We denote this program as P_5; the query is again $? - p_g(X)$.

```
:-table p/1.                    :-table q/1.
p(X):-p_c1(Y), X is 3*Y, X<20.  q(1).
p(X):-q_g(X).                   q(X):-p_c2(Y), X is 2*Y, X<20.
```

In this case, `q(X)` cannot be completely evaluated on its own, as it depends on the `p(X)` subgoal because of the goal $p_{c2}(Y)$. So, to determine completion, the subgoal dependencies have to be taken into account. Evaluation thus does not proceed by completely processing one subgoal at a time, but by processing sets of inter-dependent subgoals (where potentially all possible pairs among them are mutually dependent) called *scheduling components*. The tabled abstract machine has thus to be able to maintain information about scheduling components and to possibly switch back and forth between all subgoals in a scheduling component in order to schedule the return of every answer to all their consumers. This context switching between subgoals may need to take place more than once. Regarding completion of scheduling components and space reclamation, note the following: before encountering $p_{c2}(Y)$, the `q(X)` subgoal formed its own scheduling component and could be completed (meaning its state and the state of its possible consumers could be forgotten) on its own. Upon encountering $p_{c2}(Y)$ this situation changed (we will say that a *coup* occurred) and the two scheduling components collapsed into a single one. The placement of the execution states of generators and consumers on the stack in Fig. 1 (stacks grow downwards) shows how the coup collapses the two scheduling components in Fig. 1(a) into one scheduling component in Fig. 1(b). The space corresponding to q_g in Fig. 1(b) cannot be reclaimed before completion of the entire scheduling component.

2.5. Tabling integrated in a Prolog environment

The above discussed issues have relied on a left-to-right, depth-first selection as in Prolog, but were otherwise pure. Even so, the usual Prolog semantics is not respected with tabling: the interesting tabled programs (those in which the generator depends on its consumer) loop in Prolog, while tabling computes their (minimal model) semantics in finite time. Even if the termination characteristics of a Prolog program are not changed by adding tabling, there is the issue of the multiplicity of the answers (tabling removes duplicate answers) and the order in which answers are consumed (tabling does not respect the order). These points might deviate from Prolog practice, but the effect is that Prolog with tabling behaves more logically and closer to the theory of the logic programming paradigm; e.g. computed answer substitutions are sets rather than sequences.

Apart from the issues above, the incorporation of tabling in full Prolog raises some new points which we briefly indicate below. Tabling in programs with side effects (I/O or `assert`/`retract`) can produce unexpected results, because consumers do not use program clause resolution and because tabling does not respect the order or multiplicity of answers in the same way as in Prolog. Next, the Prolog cut (`!/0`) operator poses either semantics or performance problems when it cuts over a generator subgoal. Therefore, currently XSB disallows cuts over tabled predicates. Also all-solution predicates like `findall/3` pose a problem. For one thing, it is difficult to give meaning to a program that uses aggregation that is not stratified like in

```
:-table p/1.
p(1).
p(X) :-findall(Y,p(Y),L), member(Z,L), X is Z+1, X < 10.
```

but in the cases where the aggregation is stratified, a special version of Prolog's `findall/3` (named `tfindall/3`) can be used in XSB. As for negation, the situation is better for tabled predicates than for ordinary Prolog predicates, thanks to the better termination properties of tabling and the use of delaying to resolve loops through negation (see [4]). Indeed, XSB supports even non-stratified negation (through a predicate called `tnot/1`) and evaluates programs according to the well-founded semantics (see [11]).

3. Notation and terminology

We assume familiarity with the usual implementation model for Prolog: the Warren abstract machine (WAM) [12]; see also [1] for a gentle introduction. Brief descriptions of the SLG-WAM and of CAT are included in the next two sections. More information on SLG-WAM and CAT can be found in [10] and [5], respectively. As a starting point, we assume a four stack WAM, i.e. an implementation with separate stacks for the choice points and the environments as in SICStus Prolog or in XSB. This is by no means essential to this article and whenever appropriate we mention the necessary modifications of CHAT for the original WAM design. We will also assume stacks to grow downwards; i.e. higher in the stack means older, lower in the stack means younger or more recent.

Notation. We will use the following notation: **H** for top of heap pointer; **TR** for top of trail pointer; **E** for current environment pointer; **EB** for top of local stack pointer; **B** for most recent choice point; the (relevant for this article) fields of a choice point are ALT, H and EB, the next alternative, the top of the heap and local stack, respectively, at the moment of the creation of the choice point; for a choice point of type T pointed by **B**, these fields are denoted as $\mathbf{B}_T$[ALT], $\mathbf{B}_T$[H] and $\mathbf{B}_T$[EB] — T can be either Generator, Consumer or Prolog. The SLG-WAM uses four more registers for freezing the four WAM stacks; **HF** for freezing the heap, **EF** for the local stack, **TRF** for the trail and **BF** for the choice point stack.

Although some of the following terminology of tabling was used in Section 2, we also include it here so as to provide a complete account of relationships between the concepts or implementation issues that are involved in the construction of a tabled abstract machine.

Tabling terminology. In a tabling implementation, some predicates are designated as *tabled* by means of a declaration; all other predicates are *non-tabled* and are evaluated as in Prolog. The first occurrence of a tabled subgoal is termed a *generator* and uses resolution against the program clauses to derive answers for the subgoal. These answers are recorded in the table (for this subgoal). All other occurrences of identical (e.g. up to variance) subgoals are called *consumers* as they do not use the program clauses for deriving answers but they consume answers from this table. Implementation of tabling is complicated by the fact that execution environments of consumers need to be retained until they have consumed all answers that the table associated with the generator will ever contain.

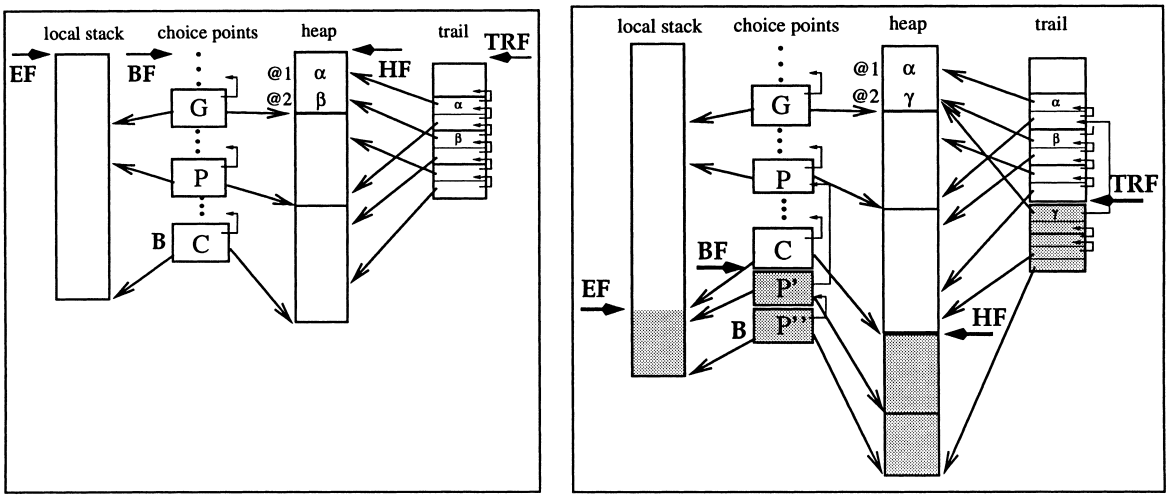
To partly simplify and optimize tabled execution, implementations of tabling try to determine *completion* of (generator) subgoals: i.e. when the evaluation has produced all their answers. Doing so involves examining dependencies between subgoals and usually interacts with consumption of answers by consumers. The SLG-WAM has a particular stack-based way of determining completion which is based on maintaining *scheduling components*; i.e.

sets of subgoals which are possibly inter-dependent. A scheduling component is uniquely determined by its *leader*: a (generator) subgoal G_L with the property that subgoals younger than G_L may depend on G_L, but G_L depends on no subgoal older than itself. Obviously, leaders are generally not known beforehand and they might change in the course of a tabled evaluation. How leaders are maintained is an orthogonal issue beyond the scope of this article; see [10] for more details. However, we note that besides determining completion, leaders of a scheduling component are usually responsible for scheduling consumers of all subgoals that they lead to consume their answers.

4. SLG-WAM

Tabling can be implemented by modifying the WAM to preserve execution environments of suspended consumers by *freezing* the WAM stacks, i.e. by not allowing backtracking to reclaim space in the stacks as is done in the WAM. In implementation terms, this means that the SLG-WAM adds an extra set of *freeze registers* to the WAM, one for each stack and allocation of new information occurs below the frozen part of the stack. Suspension of a consumer is performed in the SLG-WAM by creating a consumer choice point to backtrack through the answers in the table, setting the freeze registers to point to the current top of the stacks, and upon exhausting all answers fail back to the previous choice point without reclaiming any space. Frozen space is reclaimed *only* upon determining completion. Note that a side-effect of having frozen segments in the stacks is that the stacks actually represent trees: for example, contrary to the WAM, choice points on the same branch of the computation may not be contiguous and the previous choice point may be arbitrarily higher in the stack.

Memory areas of the SLG-WAM and their relationships are depicted in Fig. 2. Initially all freeze registers point to the beginning of the stacks; they are shown by arrows next to each stack. After executing some Prolog code the execution encounters a generator G and a generator choice point is created for it. The execution continues, some more choice points are created and eventually a consumer C is encountered. The SLG-WAM stacks at this point are shown in Fig. 2(a). The heap and the trail are shown segmented by choice points; the same segmentation is not shown for the local stack as it is a spaghetti stack. From the trail, some pointers point to cells older than the generator G: these cells have addresses @1 and @2 in the picture, and the values of the cells are α and β. One can see that



(a) Stacks on encountering a consumer (b) Continuing forward execution after freezing

Fig. 2. Memory areas while executing under an SLG-WAM-based implementation.

a trail entry in this picture consists of two pointers and a value, while in WAM, a trail entry is just one pointer. On encountering C the stacks are frozen by setting the freeze registers to point to the current top of the stack (cf. Fig. 2(b)). After possibly returning answers to C, the execution fails out of the consumer choice point of C, and suppose that the youngest choice point with unexplored alternatives is the choice point denoted as P. As shown in Fig. 2(b), allocation of new information (shown very lightly shaded) takes place below the freeze registers and no memory above the freeze registers is reclaimed. Notice the conceptual tree form of e.g. the choice point stack as shown by previous pointers from choice points: e.g. the parent choice point of P' is P. Finally, note that by continuing with some other part of the computation, some cells in the heap or local stack may change value: e.g. cell @2 from β to γ.

As expected, to resume a suspended computation of a consumer, the SLG-WAM needs to have a mechanism to reconstitute its execution environment. Besides resetting the WAM registers (e.g. setting **B** to point to the consumer choice point), the variable bindings at the time of suspension have to be restored. This can be done using what is known as a *forward trail* [10]. An entry in the forward trail consists of a reference cell, a value cell, and a pointer to the previous trail entry. These entries are shown in Fig. 2: entries for @1 and @2 record the values α, β, and γ. Because of the previous pointers the trail is also tree-structured. Given this trail, restoring the execution environment EE from a current execution environment EE_c, is a matter of untrailing from EE_c to a common ancestor of EE_c and EE, and then using values in the forward trail to reconstitute the environment of EE.

5. CAT

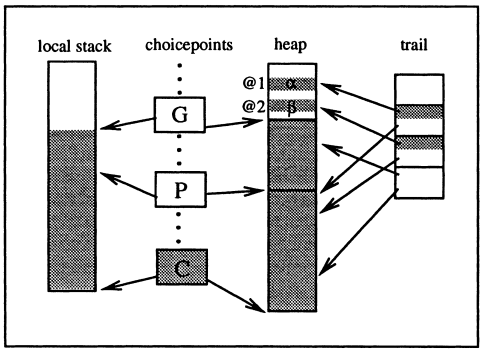
Instead of maintaining execution environments of suspended consumers through freezing the stacks and using an extended trail to reconstruct them, one can also preserve environments of consumers by copying all the relevant information about them in a separate memory area, let execution proceed as in the WAM, and reinstall these copies whenever the corresponding consumers need to be resumed. This is the main idea behind CAT: the 'copying approach to tabling'. An advantage of this approach is that, contrary to the SLG-WAM, Prolog execution happens as in the WAM: there is no need for a forward trail nor freeze registers and the stacks do not have a tree form. CAT selectively copies information needed to reinstall suspended environments in *CAT areas* as briefly explained below. [1]

The CAT area has four memory areas (containing information from each of the four WAM stacks). Fig. 3 shows these memory areas in a CAT-based implementation. When execution encounters a consumer, a choice point C is created for it. Let the youngest generator choice point in the stack be G (the dots show possible Prolog choice points that appear in between). A CAT copy is about to be made; the situation is depicted in Fig. 3(a). The shaded parts in Fig. 3(a) show exactly what CAT copies. From the heap, the CAT copies the part between the current top **H** and $\mathbf{B}_G[H]$. The part of the local stack that needs copying is between **EB** and $\mathbf{B}_G[E]$. One could think that from the choice point stack, CAT needs to copy from **B** till $\mathbf{B}_G$, but [5] argues this is wrong because this would lead to re-execution: instead, it is correct to copy only the consumer choice point.

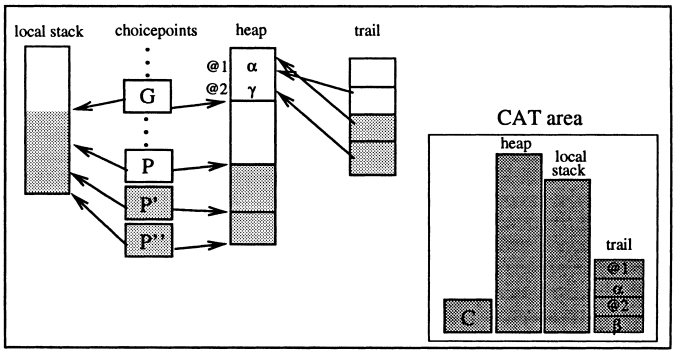
Copying the trail is more complicated: as we do not save the part of the heap that is older than $\mathbf{B}_G$ and since this part can contain values that were put there during execution more recent than $\mathbf{B}_G$, we need to save together with the trailed addresses also the values these trailed addresses now contain; we do not need a similar value trail for the part of the heap that is more recent than $\mathbf{B}_G$ because we copy that part completely.

The copied information is saved in a CAT area which is separate from the stacks (cf. Fig. 3(b)) and execution continues as in the WAM by failing out of the consumer choice point. Contrary to what happens in the SLG-WAM, backtracking in CAT now reclaims space. Fig. 3(b) shows a possible situation where backtracking has taken place up to a Prolog choice point P which lied between G and C in Fig. 3(a), and then an alternative path of the computation was tried (shown in a darker shade). Note that this new computation has resulted in the stacks having different contents than what is saved in the CAT areas (although as shown some parts are still intact). Also note that if P'' is a

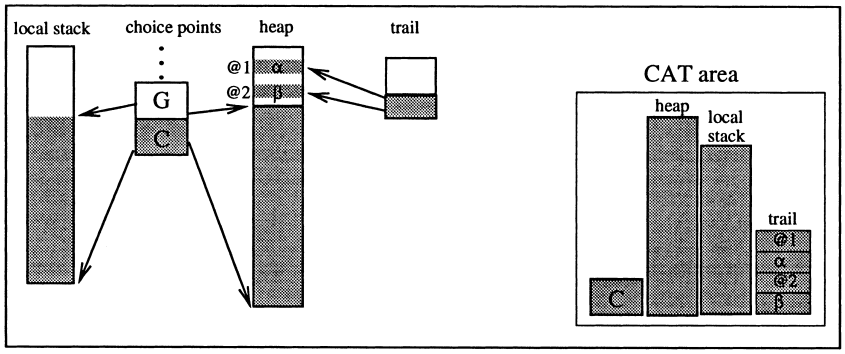
[1] Actually, it does so in a more incremental way, but as this is not relevant for this article we refer to [5] for more details.



(a) Stacks just before making the CAT copy



(b) Stacks and CAT area after making the CAT copy and continuing computation



(c) Memory areas upon reinstalling the CAT area for consumer C

Fig. 3. Memory areas while executing under a CAT-based implementation.

consumer choice point, another CAT area will be created at this point. Eventually, through backtracking execution will fall back to G and after G exhausts all resolution with program clauses, the evaluation reinstalls consumers with unresolved answers that have copied up to the generator G.

The resulting stacks are shown in Fig. 3(c): through copying, the consumer has just been reinstalled below $\mathbf{B}_G$ and can start consuming its answers from the table. Note that after reinstalling the consumer, the choice point and trail stack are in general smaller than at the time of saving the CAT area. The CAT area itself remains in existence until it can be determined that the associated generator is complete.

6. The anatomy of CHAT

We describe the actions of CHAT by means of example situations. First consider a similar example as that used for SLG-WAM and for CAT in the previous sections: a generator G has already been encountered and a *generator choice point* has been created for it immediately below a (Prolog) choice point P_0; then execution continued with some other non-tabled code (P and all choice points shown by dots in Fig. 4). Eventually a consumer C was encountered

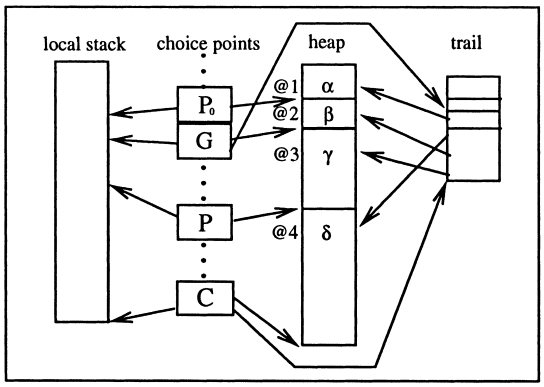


Fig. 4. CHAT stacks immediately upon laying down a consumer choice point.

and let us, without loss of generality, assume that G is its generator and G is not completed.[2] Thus, a *consumer choice point* is created for C; see Fig. 4. As previously, the heap and the trail are shown segmented according to the values saved in the corresponding fields of choice points. The local stack is not segmented in the same way, the EB values of choice points are shown by pointers nevertheless.

Without loss of generality, let us initially assume that C is the only consumer. The whole issue is how to preserve the execution environment of C. As seen, CAT does this very simply through (selectively and incrementally) copying all necessary information in a separately allocated memory area — see [5]. The SLG- WAM employs *freeze registers* and freezes the stacks at their current top; allocation of new information occurs below these freeze points — see [10] or Section 4. We now describe what CHAT does.

6.1. Freezing the heap without a heap freeze register

As mentioned, we want to prevent that on backtracking to a choice point P that lies between the consumer C and the nearest generator G (included), **H** is reset to the $\mathbf{B}_P$[H] as it was on creating P. However, the WAM sets

$$\mathbf{H} := \mathbf{B}_P[\mathrm{H}]$$

upon backtracking to a choice point pointed to by $\mathbf{B}_P$. We can achieve that no heap lower than $\mathbf{B}_C$[H] is reclaimed on backtracking to P, by manipulating its $\mathbf{B}_P$[H] field, i.e. by setting

$$\mathbf{B}_P[\mathrm{H}] := \mathbf{B}_C[\mathrm{H}]$$

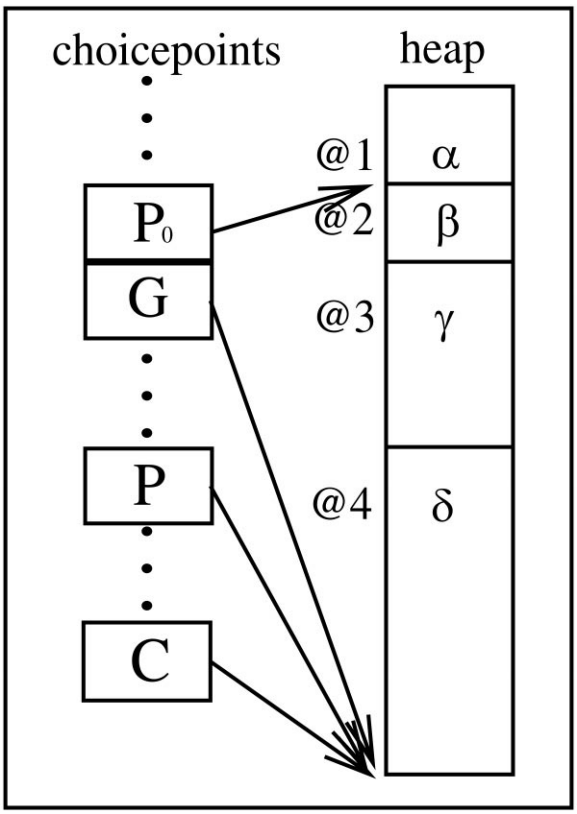
at the moment of backtracking out of the consumer. Note that rather than waiting for execution to backtrack out of the consumer choice point, this can happen immediately upon encountering the consumer (see also [10] on why this is correct).

More precisely, upon creating a consumer choice point for a consumer C the action of CHAT is

```
for all choice points P between C and its generator (included)
  B_P[H] := B_C[H]
```

The picture (illustrating the protection of the heap using the (HAT-freeze technique) shows which H fields of choice points are adapted by CHAT in our running example.

[2] Otherwise, if G is completed, the whole issue is trivial as a *completed table optimization* can be performed and execution proceeds by backtracking through answers in the table as if these were stored as Prolog facts; see [10].



To see why this action of CHAT is correct, compare it with how the SLG-WAM freezes the heap using the freeze register **HF**:

when a consumer is encountered, the SLG-WAM sets **HF** := $\mathbf{B}_C$[H]
on backtracking to a choice point P, the SLG-WAM resets **H** as follows:
`if older`($\mathbf{B}_P$[H], **HF**) then **H** := **HF** else **H** := $\mathbf{B}_P$[H]

In this way, CHAT neither needs the freeze register **HF** of the SLG-WAM nor uses copying for part of the heap as CAT.

The cost of setting the **B**[H] fields by CHAT is linear in the number of choice points that exist between the consumer and the generator up to which the setting is performed. In principle this is unbounded, so the act of freezing in CHAT can be arbitrarily more costly than in SLG-WAM. However, this problem only exists for the basic CHAT abstract machine as described in this article and is completely dealt with in the full CHAT machine described in a forthcoming paper [7]. Secondly, our experience with CHAT is that this problem seems not to occur in practice, at least not often; see the experimental results of Section 9. This is fortunate, because full CHAT is more difficult to implement than basic CHAT.

6.2. Freezing the local stack without EF

The above mechanism can also be used for the top of the local stack. Similar to what happens for the H fields, CHAT sets the EB fields in affected choice points to $\mathbf{B}_C$[EB]. In other words, the action of CHAT is

`for all choice points` P `between the consumer` C `and its generator (included)`
$\mathbf{B}_P[\mathrm{EB}] := \mathbf{B}_C[\mathrm{EB}]$

The top of the local stack can now be computed as in the WAM

`if older (`**B**[EB],**E**`) then` **E**+`length(environment) else` **B**[EB]

and no change to the underlying WAM is needed.

Again, we look at how the SLG-WAM employs a freeze register **EF** to achieve freezing of the local stack: **EF** is set to **EB** on freezing a consumer. Whenever the first free entry on the local stack is needed, e.g. on backtracking to a choice point **B**, this entry is determined as follows:

`if older (`**B**[EB],**EF**`) then` **EF** `else` **B** [EB]

The code for the `allocate` instruction is slightly more complicated as a three-way comparison between **B**[EB], **EF** and **E** is needed.

It is worth noting at this point that this schema requires a small change to the `retry` instruction in the original three stack WAM, i.e. when choice points and environments are allocated on the same stack. The usual code (on backtracking to a choice point **B**) can set **EB** := **B** while in CHAT this must become **EB** := **B**[EB].

As far as the complexity of this scheme of preserving environments is concerned, the same argument as in Section 6.1 for the heap applies. Below we will refer to CHAT's technique of freezing a WAM stack without the use of freeze registers as *CHAT freeze*.

6.3. The choice point stack and the trail

CHAT borrows the mechanisms for dealing with the choice point stack and the trail from the CAT model. From the choice point stack, CAT copies only the consumer choice point. The reason is that at the moment that the consumer C is scheduled to consume its answers, all the Prolog choice points (as well as possibly some generator choice points) will have exhausted their alternatives, and will have become redundant. This means that when a consumer choice point is reinstalled, this can happen immediately below the scheduling generator (see [5]) for a more detailed justification why this is so. CHAT does exactly the same thing: it copies in what we call a *CHAT area* the consumer choice point. This copy is reinstalled whenever the consumer needs to consume more answers.

Also for the trail, CHAT's actions are similar to those of CAT: the part of the trail between the consumer and the generator is copied together with the values the trail entries point to. However, as also the heap and local stack are copied by CAT, CAT can make a selective copy of the trail, while CHAT must copy all of the trail between the

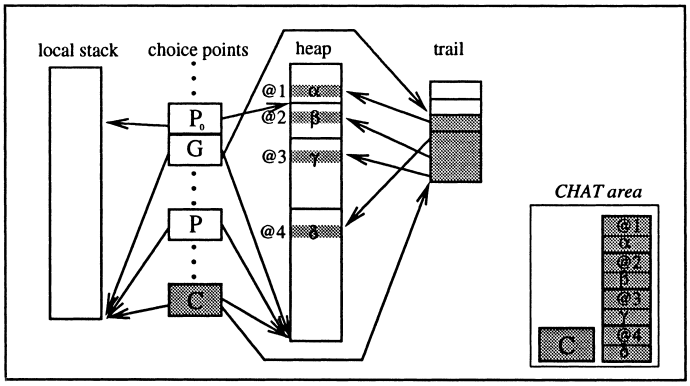


Fig. 5. Stacks and CHAT area after making a CHAT copy and adapting the choice points.

consumer and the generator. Note that this amounts to reconstructing the forward trail of the SLG-WAM (albeit without back-pointers) for part of the computation.

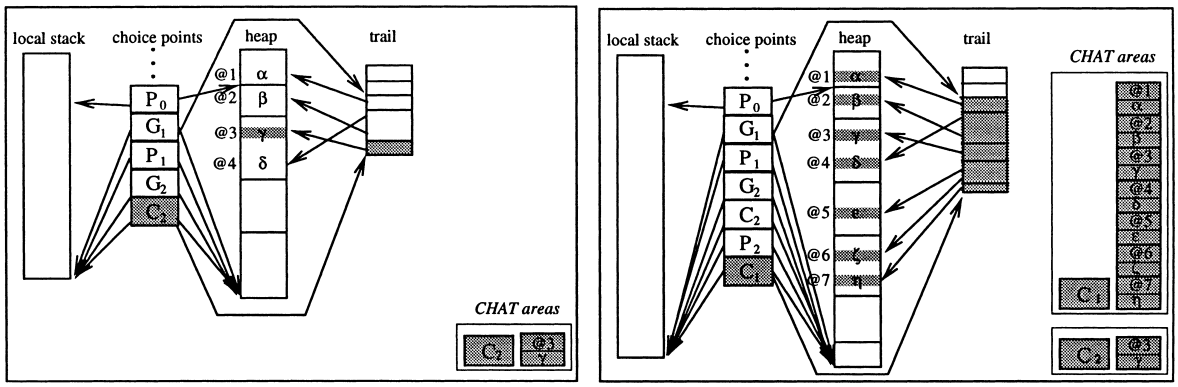
For a single consumer, the cost of (partly) reconstructing the forward trail is not greater (in complexity) than what the SLG-WAM has incurred while maintaining this part of the forward trail. Fig. 5 shows the state of CHAT immediately after encountering the consumer and doing all the actions described above; the shaded parts of the stacks show exactly the information that is copied by CHAT.

6.4. More consumers and change of leader: a more incremental CHAT

The situation with more consumers, as far as freezing the heap and local stack goes, is no different from that described so far. Any time a new consumer is encountered, the **B**[EB] and **B**[H] fields of choice points that exist between the new consumer and its generator are adapted. Note that the same choice point can be adapted several times and that the adapted fields can only point lower in the corresponding stacks; an example is shown in Fig. 6. Assume that there exist two generators G_1 and G_2 and that a consumer C_2 of the same tabled subgoal as generator G_2 is encountered. The action of CHAT is to copy the trail and perform CHAT freeze till G_2; see Fig. 6(a). By continuing with forward execution, e.g. by returning an answer to C_2 another consumer C_1 is encountered (this time of the same subgoal as G_1) and the action is now to perform CHAT freeze till G_1; Fig. 6(b). Note how the adapted fields now point lower in the stacks.

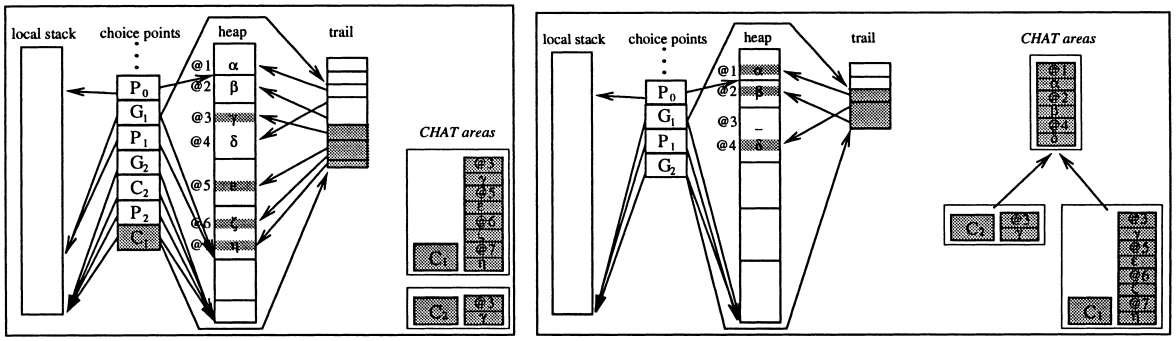
It is also worth considering explicitly a coup: a change of leaders. The example shown in Fig. 6 already presents such a situation: at the moment when C_2 is encountered, the answers of G_2 do not depend on the answers of G_1. This ceases to be the case when C_1 is encountered: the two scheduling components — containing G_2 and G_1, respectively — collapse into one and now the single leader G_1 is responsible for determining fixpoint and completion of both subgoals. Note that as far as the heap and local stack is concerned, nothing special needs to be done if each consumer performs CHAT freeze till its current leader at the time of its creation; Fig. 6(b) already shows this.

For the trail, a similar incremental coup handling mechanism as for CAT applies to CHAT as well: an incremental part of the trail between the former and the new leader needs to be copied and this copy needs to be attached to all consumers whose leader changed. In our running example this would amount to making an incremental trail copy of the following address-value pairs: (@4,δ), (@2,β) and (@1,α) and attaching it to the CHAT area of consumer C_2. In [5] it is shown that this need not be done immediately at the moment of the coup, but can be postponed until backtracking happens over a former leader so that the incremental copy can be easily shared between many



(a) Stacks and CHAT areas on suspending C_2 (b) Stacks and CHAT areas on suspending C_1

Fig. 6. Actions of CHAT when CHAT freeze and trail copy is non-incremental.



(a) Stacks and CHAT area on suspending C_1

(b) Incremental actions immediately before failing out of G_2

Fig. 7. Actions of CHAT with incremental CHAT freeze and incremental trail copying.

consumers. It also leads directly to the same incremental copying principle as in CAT: each consumer needs only to copy trail up to the nearest generator and update this copy when backtracking over a non-leader generator occurs.

The incrementality of copying parts of the trail also applies to the change of the EB and H fields in choice points: instead of adapting choice points up to the leader, one can do it up to the nearest generator; see Fig. 7(a) . In this scheme, if backtracking happens over a non-leader generator, then its EB and H fields have to be propagated to all the choice points up to the next generator; see Fig. 7(b). Indeed, our implementation of CHAT employs both incremental copying of the trail and incremental adaptation of the choice points. Fig. 7 shows a more accurate picture of tabled execution in our CHAT implementation; the actions upon the suspension of C_1 are those shown in Fig. 6(a). Note that the incremental copy of the trail is shared between all consumers that need it; Fig. 7(b).

6.5. Reinstalling consumers

As in CAT, CHAT can reinstall a single consumer C by copying the saved consumer choice point just below the choice point of a scheduling generator G. Let this copy happen at a point identified as $\mathbf{B}_C$ in the choice point stack. The CHAT trail is reinstalled also exactly as in CAT by copying it from the CHAT area to the trail stack and reinstalling the saved bindings. There remains the installation of the correct top of heap and local stack registers:

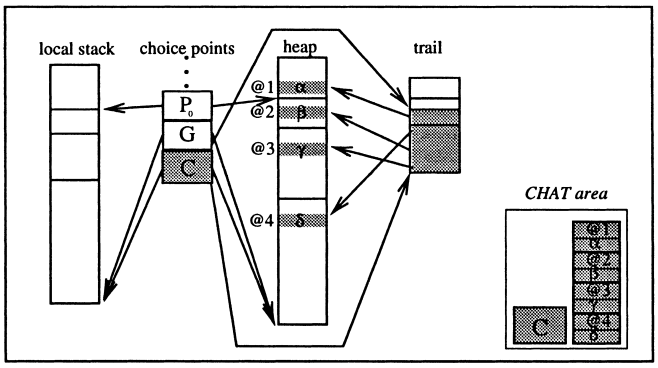


Fig. 8. Memory areas upon reinstalling the CHAT area for a single consumer C.

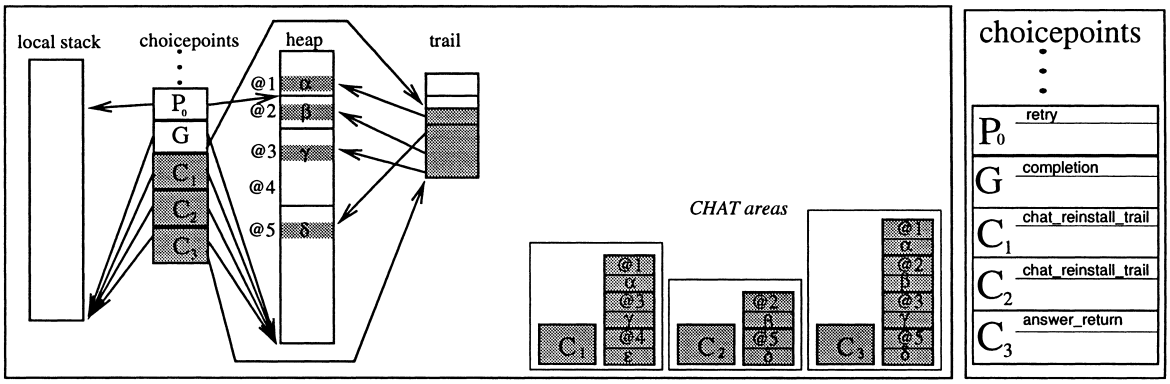


Fig. 9. Scheduling of three consumers C_1, C_2, C_3 and reinstallation of the trail of C_3.

since the moment C was first saved in the CHAT area, it is possible that more consumers were frozen, and that these consumers are still suspended (i.e. their generators are not complete) when C is reinstalled. It means that C must protect also the heap of the other consumers. This is achieved by installing in $\mathbf{B}_C$ the EB and H fields of G at the moment of reinstallation. This will lead to correctly protecting the heap as G cannot be older than the leader of the still suspended consumers and G was in the active computation when the other consumers were frozen. Fig. 8 depicts the resumption of the consumer that was suspended as in Fig. 5 and gives a rough idea of the reinstallation of the execution environment of a single consumer; shaded parts of the stacks show the copied information.

The above describes a scheduling algorithm that schedules one consumer at a time. This contrasts with the current SLG-WAM scheduler which schedules in one pass all the consumers for which unconsumed answers are available. The CHAT scheduler can do the same as follows: the saved consumer choice points of all scheduled consumers $C_1, \ldots, C_k$ are copied one after the other in the choice point stack (again starting from immediately below the choice point of the generator G). The EB and H fields of these choice points reflect the corresponding fields of G. However, only one of these consumers can have its execution environment (as represented by bindings in the corresponding CHAT trail area) reinstalled at any given point. Thus, only the last scheduled consumer C_k can immediately reinstall its trail and start consumption of its answers. The situation is depicted for $k = 3$ in Fig. 9; the right part of the figure shows the ALT fields of choice points in detail. After all answers have been returned to C_k, this choice point is exhausted and execution fails back to the choice point of another scheduled consumer C_{k-1}; at this point C_{k-1} through the use of a `chat_reinstall_trail` instruction reinstalls its trail *once* and starts consumption of the current set of answers through the `answer_return` instruction (see [10]). In short, upon failing to a consumer choice point whose $\mathbf{B}_C$[ALT] is a `chat_reinstall_trail` instruction the action taken is as follows:

```
reinstall the trail from the CHAT trail area of the consumer in B_C;
save in B_C[TR] the new top of trail stack;
alt:= B_C [ALT]:=answer_return;
goto alt; /* transfer control to alt */
```

6.6. Releasing frozen space and the CHAT areas upon completion

The generator choice point of a leader is popped only at completion of its component. At that moment, the CHAT areas of the consumers that were led by this leader can be freed: this mechanism is again exactly the same as in CAT. Also, there are no more program clauses to execute for the completed leader and backtracking occurs to the previous choice point, say P_0.[3] P_0 contains the correct top of local stack and heap in its EB and H fields: these

[3] If there is no previous choice point, the computation is finished.

fields could have been updated in the past by CHAT or not. In either case they indicate the correct top of heap and local stack.

The SLG-WAM achieves this reclamation of stacks at completion of a leader by resetting the freeze registers from the values saved in the generator choice point of the leader. Indeed, the SLG-WAM saves **HF, EF, TRF** and **BF** in all generator choice points; see [10].

6.7. Handling of negation in CHAT

Although efficient support for well-founded negation implies the ability to suspend and resume negative literals of incomplete tabled subgoals, the ability to implement the full set of operations of SLG resolution [4] and thus compute the well-founded semantics is an issue orthogonal to the actual mechanism used for suspension/resumption. Our current implementation of CHAT provides support for well-founded negation and indeed, the handling of DELAYING and SIMPLIFICATION SLG operations is similar to its handling by the SLG-WAM (see [11]). We therefore mainly restrict our attention here to fixed-order stratified negation as in [10]. Upon encountering a negative literal of an incomplete subgoal the SLG-WAM lays down a negation suspension frame in the choice point stack and the current execution path is suspended by freezing the WAM stacks (see [10]). CHAT mimicks this behaviour by creating a CHAT area for the suspended computation, saving immediately the negation suspension frame there (i.e. without creating it first on the CP stack) and incrementally saving its trail entries as in the case of a suspended consumer. Upon derivation of an (unconditional) answer for a subgoal with negation suspensions, the CHAT areas of these suspensions can be immediately reclaimed effectively failing the corresponding execution paths. The more interesting case is when a subgoal fails (i.e. gets completed with no answers): then all its negation suspension frames can be reinstalled through copying them one after the other in the choice point stack. The topmost one can immediately reinstall its trail and continue with its saved forward continuation. The situation is completely analogous to the case of scheduling multiple consumers described in Section 6.5; notable differences are that:

1. Since in the case of fixed-order stratified negation the subgoal is completed, the negation suspension frames are not reinstalled below the choice point of the generator $\mathbf{B}_G$, but starting from that point. In the case where DELAYING is needed, the subgoal is not completed, so the resumption happens immediately below the choice point of the leader.
2. CHAT has the possibility to immediately reclaim the CHAT area of negation suspension frames upon reinstallation of its trail as this reinstallation only happens once. This action, which in general is not easy to do in the SLG-WAM because the suspension frames may be trapped above the freeze registers in the choice point stack, is valid even in the case of non-stratified negation.

7. Comparing CHAT with SLG-WAM

As noted in [5], a worst case for CAT can be constructed by making CAT copy and reinstall arbitrarily often, arbitrarily large amounts of heap to (and from) the CAT area. Since CHAT does not copy the heap, this same worst case does not apply. Still, CHAT —in its basic form described in this article —can be made to behave arbitrarily worse than the SLG-WAM. We also show an example in which the SLG-WAM uses arbitrarily more space than CHAT.

7.1. The worst case for CHAT

There are two ways in which CHAT can be worse than the SLG-WAM:

1. Every time a consumer is saved, the part of the choice point stack between the consumer and the leader is traversed; such an action is clearly not present neither in the SLG-WAM nor in CAT.
2. Some trail chunks are copied by CHAT for each initial suspension of a consumer and these chunks are not shared between consumers. The inefficiency lies in the fact that consumers in the SLG-WAM can share a part of the

trail even strictly between the consumer and the nearest generator; this is a direct consequence of the forward trail with back pointers. Both space and time complexity are affected. Note that the same source of inefficiency is present in CAT as well.

The following example program shows both effects. The subscripts g and c denote the occurrence of a subgoal that is a generator or consumer for p(_).

```
query(Choices,Consumers) :-
  make_choices(Choices,_),
  make_consumers(Consumers,[]).
make_choices(N,trail) :-
  N>0, M is N-1, make_choices(M,_).
make_choices(0,_).
make_consumers(N,Acc) :-
  N>0, M is N-1,
  pc(_), make_consumers(M,[a|Acc]).
:-table p/1.
p(1).
```

Predicate `make_choices/2` is supposed to create choice points; if the compiler is sophisticated enough to recognize that it is indeed deterministic, a more complicated predicate with the same functionality can be used. The reason for giving the extra argument to `make_consumers` is to make sure that on every creation of a consumer, **H** has a different value and an update of the H field of choice points between the new consumer and the generator is needed —otherwise, an obvious optimization of CHAT would be applicable. The query is e.g. `?-query(100,200)`. CHAT uses (*Choices* ∗ *Consumers*) times more space and time than SLG-WAM for this program. If the binding with the atom trail were not present in the above program, CHAT would also use a factor of (*Choices* ∗ *Consumers*) more time than CAT.

At first sight, this seems to contradict the statement that CHAT is a better implementation model than CAT. However, since for CHAT the added complexity is only related to the trail and choice points, the chances for running into this in reality are lower than for CAT whose worst behaviour is also related to the heap and the local stack.

7.2. A best case for CHAT

The best case space-wise for CHAT compared to SLG-WAM happens when lots of non-tabled choice points get trapped under a consumer: in CHAT, they can be reclaimed, while in SLG-WAM they are frozen and retained till completion. The following program shows this phenomenon:

```
query(Choices,Consumers) :-
  pg(_), create(Choices,Consumers), fail.
create(Choices,Consumers) :-
  Consumers>0,
  (make_choicepoints(Choices), pc(Y), Y=2
  ;C is Consumers-1, create(Choices,C)
  ).
make_choicepoints(C):-
  C > 0, C1 is C-1, make_choicepoints(C1).
make_choicepoints(0).
:-table p/1.
p(1).
```

When called with e.g. `?-query(25,77).` the maximal choice point usage of SLG-WAM contains at least $25 * 77$ Prolog choice points plus 77 consumer choice points; while CHAT's maximal choice point usage is 25 Prolog choice points (and 77 consumer choice points reside in the CHAT areas). Time-wise, the complexity of this program is the same for CHAT and SLG-WAM.

One should not exaggerate the impact of the best and worst cases of CHAT: in practice, such contrived programs rarely occur and probably can be rewritten so that the bad behaviour is avoided. One can note that a small change to the SLG-WAM gets rid of its worst case behaviour: just saving and restoring the consumer choice point in the CHAT way is enough. On the other hand, removing the worst case behaviour of CHAT (while still not changing the underlying WAM for strictly non-tabled execution) is doable but considerably more involved. This subject is explored in detail in [7].

8. Alternatives for implementing tabling

After SLG-WAM and CAT, CHAT offers a third alternative for implementing the suspend/resume mechanism that tabled execution needs. The influence of CAT and SLG-WAM on CHAT is apparent: CHAT shares with CAT the characteristic that Prolog execution is not affected and with SLG-WAM the high sharing of execution environments of suspended computations. On the other hand, viewed from a lower-level implementation perspective, CHAT is not really a mixture of CAT and SLG-WAM: CHAT copies the trail in a different way from CAT and CHAT freezes the stacks differently from SLG-WAM, namely with the CHAT freeze technique. CHAT freeze can be achieved for the heap and local stack only. Getting rid of the freeze registers for the trail and choice point stacks can only be achieved by means of copying; Section 8.2 elaborates on this.

8.1. A plethora of abstract machines for tabling

Thus, it seems there are three alternatives for the heap (SLG-WAM freeze, CHAT freeze and CAT copy) and likewise for the local stack, while there are two alternatives for both choice point and trail stack (SLG-WAM freeze and CAT copy). The decisions on which mechanism to use for each of the four WAM stacks are independent. It means there are at least 36 different implementations of the suspend/resume mechanism which is required for tabling!

It also means that one can achieve a CHAT implementation starting from the SLG-WAM as implemented in XSB, get rid of the freeze registers for the heap and the local stack, and then introduce copying of the consumer choice point and the trail. This was our first attempt: the crucial issue was that before making a complete implementation of CHAT, we wanted to have some empirical evidence that CHAT freeze for heap and local stack was correct. As soon as we were convinced of that, we implemented CHAT by partly recycling the CAT implementation of [5] which is also based on XSB as follows:

- Replacing the selective trail copy of CAT with a full trail copy of the part between consumer and the closest generator.
- Not copying the heap and local stack to the CAT area while introducing the CHAT freeze for these stacks; this required a small piece of code that changes the H and EB entries in the affected choice points at CHAT area creation time and consumer reinstallation.
- Modifying XSB's handling of negation, previously based on the SLG-WAM, to be in accordance with Section 6.7.

As a final comment, it might have been nice to explore all 36 possibilities of implementing tabling, with two or more scheduling strategies and different sets of benchmarks but unlike cats, we do not have nine lives!

8.2. More insight on CHAT's design

One might wonder why CHAT can achieve easily (i.e. without changing the WAM) the freezing of the heap and the local stack (just by changing two fields in some choice points) but the trail has to be copied and reconstructed.

There are several ways to see why this is so. In WAM, the environments are already linked by back-pointers, while trail entries (or better trail entry chunks) are not. Note that SLG-WAM does link its trail entries by back-pointers; see [10]. Another aspect of this issue is also typical to an implementation which uses untrailing (instead of copying) for backtracking (or more precisely for restoring the state of the abstract machine): it is essential that trail entry chunks are delimited by choice points; this is not at all necessary for heap segments. Finally, one can also say that CHAT avoids the freeze registers by installing their value in the affected choice points: the WAM will continue to work correctly, if the H fields in some choice points are made to point lower in the heap. The effect is just less reclamation of heap upon backtracking. Similarly for the local stack, on the other hand, the TR fields in choice points cannot be changed without corrupting backtracking.

9. Performance measurements

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. C(H)AT area (for C(H)AT).[4] The size of the table space is not shown as it is independent of the suspension/resumption mechanism that is used. The benchmark set is exactly the same as in [5] where more information about the characteristics of the benchmarks and the impact of the scheduling can be found. We also note that the versions of CAT and CHAT we used did not implement the multiple scheduling of consumers described in Section 6.5, while the SLG-WAM scheduling algorithms scheduled all consumers in one pass and was thus invoked less frequently.

9.1. A benchmark set dominated by tabled execution

Programs in this first benchmark set perform monomorphic type analysis by tabled execution of type-abstracted input programs. Minimal Prolog execution is going on as tabling is also used in the domain-dependent abstract operations to avoid repeated subcomputations. Tables 1 and 2 show the time and space performance of SLG-WAM, CHAT and CAT for the batched (indicated by B in the tables) and local scheduling strategy (indicated by L).

For the local scheduling strategy, CAT and CHAT perform the same time-wise and systematically better than SLG-WAM. Under the batched scheduling strategy, the situation is less clear, but CHAT is never worse than the other two. Taking into account the uncertainty of the timings, it is fair to say that except for `read_o` all three implementation schemes perform more or less the same time-wise in this benchmark set for batched scheduling.

As can be seen in Table 2, the local scheduling strategy has a clear advantage in space consumption in this benchmark set: the reason is that its scheduling components are tighter than those of batched scheduling — we refer to [5] for additional measurements on why this is so. Space-wise, CHAT wins always over CAT and 6 out of 8 times from SLG-WAM (using local scheduling). Indeed, most often the extra space that CHAT uses to copy trail entry chunks is compensated for by the lower choice point stack consumption.

9.2. A more realistic mix of tabled and Prolog execution

Programs in this second benchmark set are also from an application of tabling to program analysis: a different abstract domain is now used and although tabling is necessary for the termination of the fixpoint computation, the domain-dependent abstract operations do not benefit from tabling as they do not contain repeated (i.e. identical) subcomputations; they are thus implemented in plain Prolog. As a result, in this set of benchmarks 75–80% of the execution concerns Prolog code. We consider this mix a more "typical" use of tabling. We note at this point that

[4] It is known that the memory reclamation of XSB could be improved. Although this affects all implementation schemes, it probably does not affect them equally, so the space figures should be taken *cum grano salis.*

Table 1
Time performance of SLG-WAM, CAT and CHAT under batched and local scheduling

	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
CHAT(B)	0.21	0.42	0.13	0.15	0.15	0.46	0.14	0.73
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
CHAT(L)	0.22	0.42	0.12	0.15	0.14	0.40	0.11	0.53
CAT(L)	0.22	0.42	0.12	0.15	0.14	0.40	0.11	0.55

Table 2
Space performance of SLG-WAM, CAT and CHAT under batched and 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
CHAT(B)	9.6	11.6	8.4	24.7	35.1	770	276	1080
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
CHAT(L)	5.8	7.2	5.6	19	8.2	16	13.2	101
CAT(L)	7.9	10.7	7.1	29.5	12.5	17	23.5	246

CHAT (and CAT) have faster Prolog execution than SLG-WAM by around 10% according to the measurements of Sagonas and Swift [10] — this is the overhead that the SLG-WAM incurs on the WAM. In the following tables all figures are for the local scheduling strategy; batched scheduling does not make sense for this set of benchmarks as the analyses are based on an abstract least upper bound (lub) operation. For lub-based analyses, local scheduling bounds the propagation of intermediate (i.e. not equal to the lub) results to considerably smaller components than those of batched: in this way, a lot of unnecessary computation is avoided.

Table 3 shows that CAT wins on average over the other two. CHAT comes a close second: in fact CHAT's performance in time is usually closer to those of CAT than those of SLG-WAM. Space-wise — see Table 4 — CHAT wins over both SLG-WAM and CAT in all benchmarks. It has lower trail and choice point stack consumption than SLG-WAM and as it avoids copying information from the local stack and the heap, it saves considerably less information than CAT in its copy area.

Table 3
Time performance of SLG-WAM, CHAT and CAT

	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
CHAT	1.25	0.62	1.03	2.54	9.73	32	0.84	2.76	10.17	6.14	8.65
CAT	1.24	0.62	0.97	2.50	9.56	32.2	0.83	2.75	9.96	6.38	8.54

Table 4
Space performance (in kBytes) of SLG-WAM, CHAT and CAT

	akl	color	bid	deriv	read	browse	serial	rdtok	boyer	plan	peep
SLG-WAM	998	516	530	472	5186	9517	279	1131	2050	1456	1784
CHAT	433	204	198	311	4119	7806	213	746	819	963	1187
CAT	552	223	206	486	8302	7847	227	821	1409	1168	1373

10. Related work

As we have noted before in [5], there is a strong analogy between the SRI-model [13] for implementing OR-parallelism and MUSE [2] 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 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. Since CHAT is a hybrid design that combines in a particular way SLG-WAM and CAT design principles, one might expect that there exists also a hybrid design that combines principles from the SRI-model and MUSE. Apparently this is not the case as private conversations with people involved in OR-parallelism revealed. It remains to be seen whether a hybrid design of an abstract machine for OR-parallelism makes sense and in particular whether the CHAT freeze technique can be of use in this domain.

11. Conclusion

CHAT offers one more alternative to the implementation of the suspend/resume mechanism that tabling requires. Its main advantage over SLG-WAM's approach is that no freeze registers are needed and in fact no complicated changes to the WAM. As with CAT, the adoption of CHAT as a way to introduce tabling to an existing logic programming system does not affect the underlying abstract machine and the programmer can still rely on the full speed of the system for non-tabled parts of the computation. Its main advantage over CAT is that CHAT's memory consumption is lower and much more controlled. The empirical results show that CHAT behaves quite well and also that CHAT is a better candidate for replacing SLG-WAM (as far as the suspension/resumption mechanism goes) than CAT. Indeed, CHAT has been fully integrated in the distribution of XSB as of version 2.0. Finally, CHAT also offers the same advantages as CAT as far as flexible scheduling strategies goes.

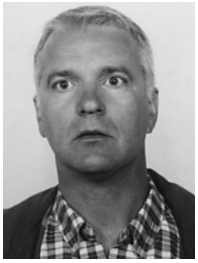
Acknowledgements

This research was performed while the second author was at the Computer Science Department of K.U. Leuven and his work was supported by a junior scientist fellowship from the K.U. Leuven Research Council.

References

[1] H. Aït-Kaci, Warren's Abstract Machine: A Tutorial Reconstruction, MIT Press, Cambridge, MA, 1991. See also: http://www.isg.sfu.ca/~hak/documents/wam.html.
[2] K.A.M. Ali, R. Karlsson, The muse approach to OR-parallel Prolog, Int. J. Parallel Programming 19 (2) (1990) 129–162.
[3] R.S. Bird, Tabulation techniques for recursive programs, ACM Computing Surveys 12 (4) (1980) 403–417.
[4] W. Chen, D.S. Warren, Tabled evaluation with delaying for general logic programs, J. ACM 43 (1) (1996) 20–74.
[5] B. Demoen, K. Sagonas, CAT: the copying approach to tabling, in: C. Palamidessi, H. Glaser, K. Meinke (Eds.), Principles of Declarative Programming, 10th International Symposium, PLILP'98, Held Jointly with the Sixth International Conference, ALP'98, LNCS, vol. 1490, Pisa, Italy, Springer, Berlin, September 1998, pp. 21–35
[6] B. Demoen, K. Sagonas, Memory management for Prolog with tabling, in: Proceedings of ISMM'98: ACM SIGPLAN International Symposium on Memory Management, Vancouver, BC, Canada, ACM, New York, October 1998, pp. 97–106.
[7] B. Demoen, K. Sagonas, CHAT is Θ(SLG-WAM), in: H. Ganzinger, D. McAllester, A. Voronkov (Eds.), Proceedings of the Sixth International Conference on Logic for Programming and Automated Reasoning, LNAI, vol. 1705, Tbilisi, Republic of Georgia, Springer, Berlin 1999, pp. 337–357.
[8] J. Freire, T. Swift, D.S. Warren, Beyond depth-first strategies: improving tabled logic programs through alternative scheduling. J. Functional and Logic Programming 1998 (3) (1998).

[9] D. Michie, Memo functions and machine learning, Nature 218 (1968) 19–22.
[10] K. Sagonas, T. Swift, An abstract machine for tabled execution of fixed-order stratified logic programs, ACM Trans. Programming Languages Syst. 20 (3) (1998) 586–634.
[11] K. Sagonas, T. Swift, D.S. Warren, An abstract machine for computing the well-founded semantics, in: M. Maher (Ed.), Proceedings of the Joint International Conference and Symposium on Logic Programming, Bonn, Germany, MIT Press, Cambridge, MA, September 1996, pp. 274–288.
[12] D.H.D. Warren, An abstract Prolog instruction set, Technical Report 309, SRI International, Menlo Park, USA, October 1983.
[13] 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, San Francisco, California, IEEE Computer Science Press, September 1987, pp. 92–102.
[14] D.S. Warren, Memoing for logic programs, Commun. ACM 35 (3) (1992) 93–111.



Bart Demoen has degrees in Mathematics (1975) and Computer Science (1983) and a Ph.D. in Theoretical Physics (1979) from the K.U. Leuven. He worked several years in a software company and was the main developer of a commercial Prolog system. His main interest is in the implementation of declarative programming languages. His publications cover topics like optimization techniques for WAM, garbage collection, abstract interpretation, partial evaluation, implementation of tabling and formal specification. He is also involved in link time optimization, optimized compilation for DSPs, the implementation of the constraint programming language HAL and the efficient execution of inductive learning systems. He currently teaches as a professor at the K.U. Leuven.

Kostis Sagonas is a travelling scientist: He has a degree in Informatics from the University of Athens, Greece, and M.Sc. and Ph.D. degrees in Computer Science from the State University of New York at Stony Brook, USA. He is currently a Senior Lecturer at the Computing Science Department of Uppsala University in Sweden after having spent two and a half productive years as a guest researcher of the DTAI group of the K.U. Leuven, Belgium. His main research interests are in the area of programming language implementation with emphasis on declarative languages. He is a firm believer that theory and practice should be combined even more and tries to do so by applying interesting ideas from the area of logic to produce more advanced programming systems. He likes seeing his research being used and so far he has been involved in the design and implementation of the XSB, Syntactica, and Semantica systems which are freely available somewhere on the WEB.