CHAT is Θ(SLG-WAM)*

Bart Demoen[1] and Konstantinos Sagonas[2]

[1] Department of Computer Science, Katholieke Universiteit Leuven, Belgium
e-mail: `bmd@cs.kuleuven.ac.be`
[2] Computing Science Department, Uppsala Universitet, Sweden
e-mail: `kostis@csd.uu.se`

Abstract. CHAT offers an alternative to SLG-WAM for implementing the suspension and resumption of consumers that tabling needs: unlike SLG-WAM, it does not use freeze registers nor a complicated trail to preserve their execution environments. CHAT also limits the amount of copying of CAT, which was previously put forward as another alternative to SLG-WAM. Although experimental results show that in practice CHAT is competitive with — if not better than — SLG-WAM, there remains the annoying fact that on contrived programs the original CHAT can be made arbitrarily worse than SLG-WAM, i.e. the original CHAT has an intrinsically higher complexity. In this paper we show how to overcome this problem, in particular, we deal with the two sources of higher complexity of CHAT: the repeated traversal of the choice point stack, and the lack of sufficient sharing of the trail. This is achieved without fundamentally changing the underlying principle of CHAT by a technique that manipulates a Prolog choice point so that it assumes temporarily a different functionality and in a way that is transparent to the underlying WAM. There is more potential use of this technique besides lowering the worst case complexity of CHAT: it leads to considering scheduling strategies that were not feasible before either in CHAT or in SLG-WAM. We also discuss extensively issues related to the implementation of the trail in a tabled logic programming system.

1 Introduction

Tabling has by now been recognized as an important feature of logic programming systems. Indeed, a number of applications that were either beyond the reach or very difficult to tackle with conventional Prolog systems are now possible using tabled evaluation. Such application areas include, but are not limited to, verification using model checking [9], program analysis [2], and logic-based databases [12]. Despite this increase in applicability of tabled implementations, for quite a long time, there seemed to be only one possible way of implementing the suspension/resumption mechanism that tabling requires in a logic programming system that was based on WAM. This mechanism is described in [11] as part of the SLG-WAM (the engine of the XSB system [12]) which also defines

* A tight correspondence between alternatives for suspension/resumption in the WAM.

and gives alternative implementations for the other components of a tabled logic programming system, i.e. the tables themselves, the scheduling strategy, the extension of the WAM instruction set and the mechanism for detecting completion. It is clear that the issues involved in the tables themselves are rather loosely coupled to the basic engine; i.e. whether one uses tries or not as data structures for tabling, does not affect the underlying WAM. So are issues related to the choice of the scheduling strategy [6], or whether completion detection is based on exact or approximate dependencies. On the other hand, the implementation of suspension/resumption as in SLG-WAM does affect the WAM, because of its introduction of a set of freeze registers and a forward trail. This compromises to a certain extent the efficiency of the underlying abstract machine, even for plain Prolog execution, but more importantly it does not allow for an easy adoption of the mechanism in an existing system. Finally, it is clear that even though the choice of a scheduling strategy can be orthogonal to the underlying LP engine, some strategies are disadvantaged or even impossible when a particular implementation for suspension/resumption is fixed. So it is important to study suspension/resumption implementation models and their properties. [11] describes one implementation of suspension/resumption but no alternative is hinted at, because it was assumed at that time that "reasonable" (i.e. sufficiently efficient) alternatives did not exist.

A first alternative implementation for suspension/resumption in tabling was offered by CAT [3] which stands for the Copying Approach to Tabling. The guiding principle of the design was that the underlying WAM should not be affected by the introduction of tabling and CAT achieved exactly that: starting from a WAM implementation, CAT implements suspension/resumption of consumers without affecting any part of the WAM. In particular, CAT employs the usual WAM trail and no freeze registers. The price to pay for this orthogonality is copying the state of consumers. Although copying has quite horrible worst cases, in practice CAT works quite well. But the high memory consumption of CAT (under certain scheduling strategies) was worrying and in [4] we have tried to remedy this by copying only data that could be reachable in forward execution; i.e. not saving any data that will be garbage on resumption of a consumer. Although this lowers the space requirements of CAT, the worst case complexity of CAT remains unaffected. Thus, still not satisfied, we proposed in [5] another alternative, CHAT, which combines certain features of CAT with SLG-WAM, hence the H in its name which stands for Hybrid. In particular, heap and local stack are frozen without the need for freeze registers and the trail is partially and incrementally copied so that the WAM trail can be retained. CHAT considerably improves on CAT space wise and offers the same added flexibility of scheduling strategies as CAT. Still, in principle, the original CHAT has two sources of added complexity which can result in arbitrarily worse behavior of CHAT than SLG-WAM (see Section 3.3)[1]. Annoyed by this theoretical problem, in this paper, we give a detailed account on how to guarantee that CHAT will not perform arbitrarily worse than SLG-WAM. This is the main contribution of the paper:

[1] The SLG-WAM can also be arbitrarily worse space-wise than CHAT; see Section 8.

we show how without fundamentally changing the underlying principle of CHAT (i.e. still without changing the underlying WAM for Prolog execution), CHAT can be implemented in such a way that it performs no worse complexity-wise (in both time and space) than SLG-WAM. We also note, however, that in practice CHAT performs better than SLG-WAM.

CHAT as in [5] is our starting point. We will improve on its incremental copying of trail segments, so that the same sharing of trail between consumers is possible as in SLG-WAM, and on the installation of the equivalent of freeze registers in choice points which in CHAT leads to repeated traversal of the choice point stack. The improvement is based on a technique that modifies choice points dynamically and in a transparent way for the underlying abstract machine: on backtracking to the modified Prolog choice point, it performs the incremental copying task that was formerly reserved for generators and then continues with its original alternative. The technique of modifying choice points opens possibilities for new scheduling strategies in the context of tabling. We also believe that it is of interest outside the relatively small area of tabled LP implementations.

In Section 2 we introduce notation used later in the paper. Section 3 briefly describes CHAT, certain aspects of the SLG-WAM, and gives examples that show the two sources of added complexity in plain CHAT. Section 4 shows how repeated traversal of the choice point stack at CHAT save time can be avoided. Section 5 shows how the same sharing of trail as in SLG-WAM is obtained and Section 6 shows how it is exploited by CHAT. Section 7 discusses details of the implementation of the trail in tabled logic programming systems. Section 8 makes an exhaustive comparison of the space complexity of CHAT and SLG-WAM.

2 Notation and Terminology

We assume familiarity with the WAM [14], SLG-WAM [11], and to some extent with CHAT [5] due to space limitations [2]. However, some aspects of the SLG-WAM and CHAT which are crucial for this paper are presented in Sections 3.1 and 3.2. 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; however, this is by no means essential for this paper or for CHAT (see [5]). We will also assume stacks to start from low addresses and to grow downwards; i.e. higher in the stack means older, lower in the stack (or more recent) means younger and a larger address value.

We will use the following notation for WAM registers: **H** for top of heap pointer; **TR** for top of trail pointer; **EB** for top of local stack pointer; **B** for most recent choice point. Three different types of choice points are used: Generator, Consumer or Prolog choice points and are identified by G, C or P respectively. The (relevant for this paper) fields of a choice point are ALT, prevB, H, EB and TR: the next alternative, the previous choice point, the top of heap, local and trail stack respectively upon the creation of the choice point. For a choice point identified by e.g. P, these fields are denoted as P[ALT], P[prevB], P[H], *etc.*

[2] All relevant papers are accessible at `http://www.csd.uu.se/~kostis/Papers/`.

In a tabled system, 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 for non-deterministic languages 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. This 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*; that is, 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 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 paper; see [11] for more details. However, we note that besides determining completion, a leader of a scheduling component is usually[3] responsible for scheduling consumers of all subgoals that it leads to consume their answers.

3 SLG-WAM, CHAT and their Complexity Difference

3.1 Suspension/resumption in SLG-WAM: a brief description

Tabling can be implemented by modifying the WAM to preserve execution environments of consumers that suspend by *freezing* the WAM stacks, i.e. by not allowing backtracking to reclaim space in the stacks as is done in the WAM. The SLG-WAM employs a register-based freezing of the WAM stacks, i.e. 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, setting the freeze registers to point to the current top of the stacks, and upon exhausting all answers from the table fall back to the previous choice point by failing as in the WAM (i.e. undoing the variable bindings and restoring the WAM registers) but without reclaiming any space. Frozen space is reclaimed *only* upon determining completion of a scheduling component.

[3] "usually" because this depends on the scheduling strategy; however, it holds for all scheduling strategies of the XSB system.

Note that this method of freezing the stacks is a constant time operation. It does impose an overhead — even on plain Prolog execution — because allocation of new information on the stacks requires a comparison of the WAM register and the corresponding freeze register of the stack; but this overhead does not change the complexity of the abstract machine.

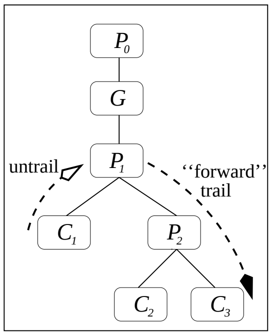
To resume a suspended computation of a consumer, the SLG-WAM needs to have a mechanism to reconstitute its execution environment. Besides resetting the WAM registers (e.g. setting **B** to point to the consumer choice point), the variable bindings at the time of suspension have to be restored. This can be done using what is known as a *forward trail* [11, 15]. An entry in the forward trail consists of a reference cell, a value cell, and a back-pointer to the previous trail entry (see Fig. 1) as opposed to the regular WAM trail which consists of only the reference cell. Note that the trail back-pointer in the SLG-WAM reflects the fact that the trail stack is used to represent the tree of trails belonging to different computations. By following the back-pointer, parts of the trail that

BackPtr	Pointer to previous trail entry
Value	Value to which the variable was bound
VarAddr	Reference to (address of) the trailed variable

Fig. 1. Format of an SLG-WAM (Forward) Trail Entry.

are not part of the same computation are skipped (see below). Given this trail, restoring the execution environment EE of a consumer 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. The exact algorithm of this operation is presented in [11].

Again, it is important to note the following: The forward trail adds a (time and space) overhead to both tabled and plain Prolog execution; however, this overhead is just a constant factor. On the other hand, the SLG-WAM makes good use of the cost of its extended trail: in particular, the back-pointers are used to minimize the cost of switching execution environments. Untrailing does not need to happen up to a generator choice point; instead it is sufficient to untrail to *any* common ancestor of EE_c and EE. In the XSB implementation of the SLG-WAM, this common ancestor is usually related to the nearest choice point; Section 7 elaborates more on this issue. The figure on the right gives a rough idea of the situation for the case of switching execution environments from C_1 to C_3; the common ancestor is P_1.

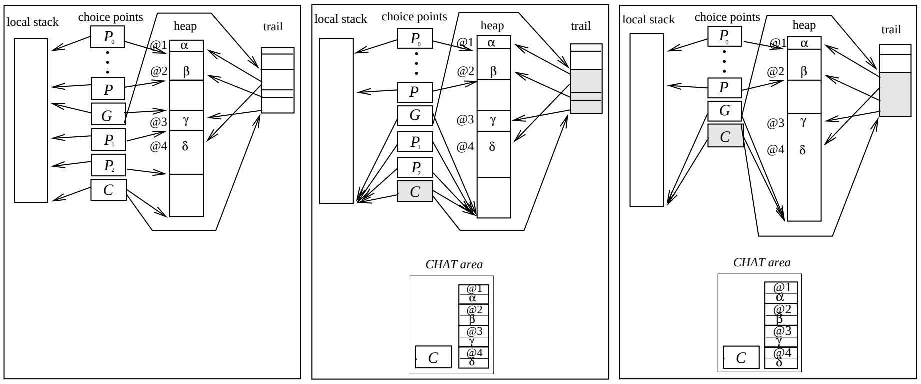


We finish this section by mentioning the design philosophy of the SLG-WAM: The efficiency of tabled execution is the prime goal. As a consequence, the basic operations of a tabled abstract machine were designed to have constant time (suspension through freezing) or lowest possible cost (resumption by exploiting sharing of trail). The small overhead added to some WAM operations is considered a reasonable price for making tabled execution efficient.

3.2 Suspension/resumption in CHAT: a brief description

CHAT's design philosophy is different: Introduction of tabling into a WAM-based system should leave the underlying WAM *unchanged* for (strictly) non-tabled execution. Naturally, CHAT tries to make the suspension/resumption support for tabling as efficient as possible, but *never* by violating the above requirement.

We describe the actions of CHAT through an example. Consider the following state of a WAM-based abstract machine for tabled evaluation: A generator G has already been encountered and a *generator choice point* has been created for it immediately below a (Prolog) choice point P. Then execution continued with some other non-tabled code and let us, without loss of generality, assume that two Prolog choice points P_1 and P_2 were created and then a consumer C was encountered, G is its generator and G is not completed at this point. Thus, a *consumer choice point* is created for C; see Fig. 2(a). The heap and the trail are shown segmented according to the H and TR values saved in choice points; the same segmentation is not shown for the local stack as it is a spaghetti stack; however the EB values of choice points are also shown by pointers.



(a) Upon creation of a consumer choice point.

(b) After freezing: CPs adapted & CHAT copy.

(c) Upon reinstalling the CHAT area of C.

Fig. 2. Stacks & CHAT area while executing under the original CHAT implementation.

CHAT preserves the execution environment of consumers partly by freezing and partly by copying. More specifically, CHAT freezes the heap and the local stack by modifying the H and EB fields of all choice points that lie between a consumer C and the nearest generator G, to C[H] and C[EB] respectively. Note that freezing in CHAT does not happen as in the SLG-WAM. We refer to CHAT's way of freezing stacks as *CHAT freeze*. To preserve information from the remaining stacks, CHAT uses copying: from the choice point stack only the consumer choice point is saved; from the trail the entries that lie between the consumer and the generator together with the values that these entries point to are saved. This copied information is saved in what is termed a *CHAT area*. Fig. 2(b) shows such an area and the resulting state of the stacks after modifying the fields of choice points; shaded parts show the information copied by CHAT.

A rigorous argument why this freezing-copying scheme is correct can be found in [5]. For the purposes of this paper, we do not need to fully explain what happens in case that G is not a leader of a scheduling component and execution backtracks over G. Full details can be found in [5], but essentially the same freezing mechanism is applied (now between G and the immediately older generator) and an incremental copy of the trail is saved in a new CHAT area. It is important however to note that this new CHAT area is copied *once* and this incremental copy is shared by *all* consumers that have their state saved up to G.

On the other hand, if upon failing back to G, G is a scheduling generator (e.g. a leader of a scheduling component), G needs to schedule all its consumers to consume answers from the table after it finishes all its program clause resolution. This implies resuming these consumers by restoring their execution environments. In CHAT resumption is also done through copying: the consumer choice point is installed immediately below the choice point of the scheduling generator, and the saved part of the trail is copied from the CHAT area back to the trail stack. Fig. 2(c) gives a rough idea of a consumer's reinstallation; shaded parts of the stacks show the copied information.

3.3 The complexity issue

The source of increase in complexity of CHAT w.r.t. SLG-WAM is two-fold:

1. for each new consumer, C, CHAT traverses the choice point stack from the consumer up to the nearest generator G; if between C and G there are n Prolog choice points $P_1 \dots P_n$, these can be visited arbitrarily often;
2. assume a generator G and a Prolog choice point P immediately younger, then if the computation starting at P creates consumers $C_1 \dots C_m$, then in SLG-WAM$C_1 \dots C_m$ share the part of the trail between P and G, but in CHAT, each consumer's CHAT area contains a separate copy of that trail part. Again, the space and time difference can be arbitrary.

The example program from [5] (Fig. 3) shows both of these problems (the subscripts g and c denote occurrences of a generator or a consumer tabled subgoal). If the compiler recognizes that predicate `make_choices/2` (which is supposed to create choice points) is indeed deterministic, a more complicated predicate can be used. The reason for giving the second argument to `make_consumers`, is to ensure 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. Against this program, in a query like:

```
?- Choices = 100, Consumers = 200, main(Choices,Consumers).
```

CHAT uses ($Choices{*}Consumers$) times more space and time than SLG-WAM.

We will address these two sources of added complexity separately and we start by dealing with the repeated traversal of the choice point stack as it introduces a technique that is also the basis for the more complicated trail sharing solution.

```
main(Choices,Consumers) :-
    p_g(_), 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, p_c(_), make_consumers(M,[a|Acc]).
:- table p/1.
p(1).
```

Fig. 3. A tabled program with different complexity under SLG-WAM and plain CHAT.

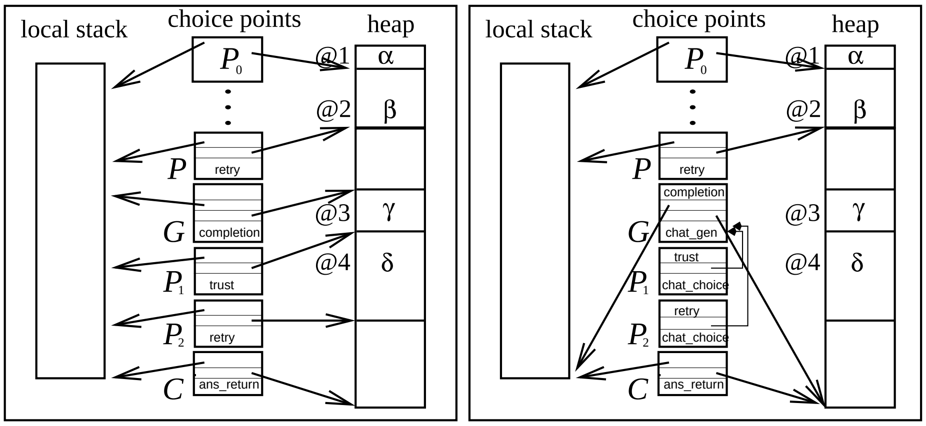
4 Avoiding Repeated Traversal of Choice Points

First note that visiting each choice point *once* does not affect the complexity. Indeed, this adds only a fixed cost to a choice point that was created already. Next, note that we must cater for the Prolog cut (`!/0`) which can cut away non-tabled choice points. As a design principle of CHAT is that no changes to the underlying WAM or its instructions should happen for plain Prolog execution, changing the implementation of cut itself is not an option (in many systems, cut is a constant time operation and does not traverse the choice point stack). Thus, we cannot put information in choice points that can disappear. Note however, that generators are used to produce all answers for consumers, so generator choice points are (and must be) immune to cuts. From this, it follows that a generator choice point is a safe place for storing information. Remember that CHAT adapts the H and EB fields of choice points. We will also make use of the ALT field of choice points, which indicates the next alternative. We will assume that one extra field in generator choice points is available: we call this field SALT and we use it for saving the value of an ALT field. However, Prolog choice points remain unchanged. We propose the code below to be executed upon CHAT-freezing a consumer C whose (nearest) generator is G. We also note that CHAT (as SLG-WAM) has constant time access to the choice point of G upon suspension of a consumer; e.g. via the completion stack. In words: for every choice point P between C and G, save its alternative in its EB-field, change its alternative to a chat_choice instruction, set its H-field to point to the

```
P := C[prevB];
while (P != G && P[ALT] != chat_choice)
  { P[EB] := P[ALT];          /* save ALT in the EB field */
    P[ALT] := chat_choice;    /* set ALT to chat_choice */
    P[H] := G;                /* link P to the generator */
    P := P[prevB];            /* continue with previous choice point */
  }
G[H] := H;                    /* the H register also equals C[H] */
G[EB] := EB;                  /* the EB register also equals C[EB] */
if (G[ALT] != chat_generator)
  { G[SALT] := G[ALT];        /* save generator's ALT field */
    G[ALT] := chat_generator; /* install a new alternative */
  }
```

generator choice point. Only the generator gets the values of **EB** and **H** which point at the current top of the local stack and the heap. The generator gets as alternative the new instruction chat_generator, which is a version of chat_choice for a generator. We will describe both new instructions later.

Fig. 4 parallels Fig. 2(a) & 2(b) and shows the state of the stacks immediately before and after the above code is executed (the trail remains unaffected and so is not shown). Choice points are now shown in more detail; in particular, their EB, H and (S)ALT fields are shown explicitly (some possible values appear in their alternative fields) as it is important to see what takes place. As in Fig. 2,



(a) Right upon consumer CP creation (b) After extended CHAT-freeze

Fig. 4. Stacks while executing under an extended CHAT implementation.

let us assume that there are two Prolog choice points P_1 and P_2 between a generator G and a consumer choice point C. P and all choice points above G are not affected by the execution of the code. The stacks' state shown in Fig. 4(b) is the result of executing the code: the completion instruction is saved in the SALT field of the generator and changed into chat_generator; retry and trust are saved in P_2[EB] and P_1[EB] respectively and the corresponding ALT fields are changed into chat_choice. The values of **EB** and **H** (or alternatively the values of C[EB] and C[H]) are installed in the corresponding fields of G and the Prolog choice points P_1 and P_2 are linked directly to G through their H field.

Note that the above code also handles the case where more that one consumer is present and which lead in the original CHAT to repeated traversal of the whole chain of choice points between the consumer and the generator. Indeed, the test for chat_choice in the condition of the while loop ensures that when a new consumer is frozen, a chain of choice points that was traversed before, is not traversed again; thus, for each backtrack operation to a particular choice point, CHAT visits it at most once more than SLG-WAM. It also shows that setting the H and EB fields of G, updates conceptually the EB and H fields of a set of choice points in a constant time operation.

The above modification of CHAT is enough to get rid of the added complexity of changing the EB and H fields in the choice point stack as performed in the original CHAT. To complete this discussion, we consider the remaining four issues: the implementation of chat_choice and chat_generator, backtracking over a generator, and the actions to be taken in case of a change of leaders:

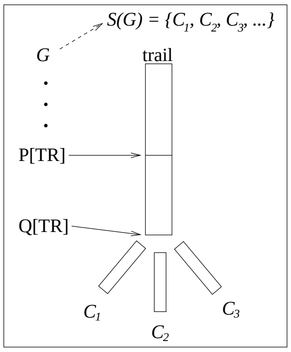
1. Consider a choice point P which has a chat_choice instruction and consider backtracking to this choice point. For convenience, name α the value of the ALT field that was replaced by chat_choice. α points to a retry or trust instruction (or one of their variants or another form of disjunction). chat_choice first installs the EB and H fields from the generator. Thereby P looses the link to the generator, but this link is indeed no longer needed. After that, chat_choice transfers control to α. The code for chat_choice is given below:

```
alt := P[EB];    /* alt points to α now */
G := P[H];       /* G points to the generator */
P[EB] := G[EB];  /* install top of protected local stack */
P[H] := G[H];    /* install top of protected heap */
goto alt;        /* transfer control to α */
```

2. chat_generator is a version of chat_choice for generators: its implementation is similar to that shown above taking into account differences in the fields of choice points that are involved. Since in this case $P = G$, it might seem that there was no need to change the ALT field of the generator and in fact, as far as the management of H and EB fields is concerned, this is true. However, all choice points need to have (a variant of) a chat_choice instruction as we will also use this instruction for the purposes of sharing trail (cf. Section 5).
3. On backtracking over G, one case is that G is a leader: then nothing special needs to be done (except releasing frozen space and the CHAT areas), and another that G is not a leader. In the latter, we consider G shortly as a new consumer (for which no CHAT area is needed) and execute the chat_choice code on the part of the choice point stack between G and its nearest generator.
4. A *coup*, i.e. a change of leaders, happens always by the creation of a consumer C which turns one (or more than one) leader G into a non-leader while some other older generator say G_L becomes the leader of the scheduling component that G belongs to. Since G_L is necessarily older than G, the generator nearest to C is always younger or equal to G, so backtracking over a generator as described above deals with this case already. In other words, a *coup* requires no special action in the context of the extended CHAT model.

5 Sharing More Trail

Let us initially assume that each generator G has an associated flat set $S(G)$ of consumers C which have G as nearest generator; $S(G)$ will later get more structure. All consumers in $S(G)$ have copied (in CHAT areas) part of the trail and for each C one can find up to which trail value C has the trail saved. On backtracking to a choice point P with alternative chat_choice, CHAT has the chance to share the segment of the trail between the current top of trail **TR** and P[TR]: for the sake of explanation, let Q be the choice point that is immediately younger than P as in the picture below. Suppose Q once had a chat_choice instruction. This means that all consumers in $S(G)$ that needed it (in the picture C_1, C_2, and C_3) have copied the trail below Q[TR] (shown as the three regions below Q[TR]). Backtracking to P means that execution is about to forget the trail between Q[TR] and P[TR], so this is the moment to save it. CHAT saves it *once* and lets all the affected consumers share it using the same mechanism that is used for incremental copying of the trail on backtracking over a non-leader generator (see [5]). If Q never had a chat_choice instruction, it might appear that the situation is more difficult, but it is sufficient to note is that if Q never had a chat_choice instruction, there never was a consumer below Q. In that case, P must have lost any chat_choice instruction before backtracking from Q to P happened. This means that up to P, the trail was already saved for all relevant consumers.



Before presenting code that achieves this sharing, we put some more structure in the set $S(G)$. Suppose execution backtracks to a choice point P with a chat_choice instruction. For each consumer C, we denote by $chat\_tr(C)$ the trail pointer up to which C has the trail in its CHAT area. Then, for every $chat\_tr(C)$ there is (or was) a choice point B such that either:

1. B is still on the stack and $chat\_tr(C) = \text{B[TR]}$ or
2. B has been removed (by trust or by cut) and the trail between $chat\_tr(C)$ and P[TR] is still intact

In the first case, B is older than P or $P = B$ and in both cases, no increment of trail must be copied for C. In the second case, if B is older than P, no increment must be copied for C. Otherwise, B is younger than P and the part of the trail between $chat\_tr(C)$ and P[TR] needs to be incrementally added to the CHAT area of C. Note also that the $chat\_tr(C)$ value for a particular C never increases in time, i.e. by copying more of the trail, $chat\_tr(C)$ moves higher in the stack and its value decreases.

The set: $T = \{tr | tr = chat\_tr(C) \wedge tr > \text{P[TR]}\}$ can be used to partition the set of consumers $S(G)$ into sets $CT(tr_i)$ according to the trail value tr_i up to which these consumers have the trail saved in their CHAT area. More formally, $CT(tr_i) = \{C | C \in S(G) \wedge tr_i \in T \wedge chat\_tr(C) = tr_i\}$ for $i = 1, \ldots, n$ for some n and such that $tr_i > tr_{i+1}$.

For a new consumer C with the same nearest generator G, if $n \neq 0$ ($S(G) \neq \emptyset$) then $chat\_tr(C) \geq tr_1$: equality is possible if no trailing occurred in the execution from the nearest choice point (whether generator or not) and C. This property is important: it ensures that new consumers can be added to $S(G)$ in constant time and that the set $S(G)$ can be managed with no added complexity.

Trail sharing implementation With the definitions of $CT(tr_i)$ as above, tr_{n+1} defined as P[TR], and $CT(tr_{n+1}) = \emptyset$, the code to achieve trail sharing is:

```
CT := CT(tr_1);
for (i = 1; i ≤ n; i++)
 { construct (in a new CHAT area) the value trail between tr_i and tr_{i+1};
   link this new CHAT area to the CHAT area of each consumer in T;
   CT := CT ∪ CT(tr_{i+1});
   S(G) := S(G) \ CT(tr_i);
 }
S(G) := S(G) ∪ CT; /* all consumers in CT have same chat_tr(C) = P[TR]*/
```

This code is executed at the end of the chat_choice (and chat_generator) instruction, e.g. just before its `goto alt` statement (see Section 4). n is the number of sets $CT(tr_i)$ in the partition of $S(G)$. In an actual implementation, one would implement and maintain the sets $CT(tr_i)$ as an ordered linked list. There would be no need for having n explicitly, so the use of n does not add extra complexity. The step that links the saved trail to the consumers in T has in CHAT a cost equivalent to the cost of the back-pointers in the forward trail of SLG-WAM.

The tree structure of CHAT areas To further understand how the implementation of trail sharing has the desired complexity, we make explicit the tree structure of CHAT areas and how the trail is shared. When a consumer is frozen, its initial CHAT area is created; it contains just the consumer choice point and a link to a linked list of CHAT trail chunks. One such chunk consists of:

1. the value of the trail pointer up to which this chunk contains a reconstruction of the value trail
2. the reconstructed value trail
3. a link to the next chunk in the chain

So, initially, the chain contains one cell, in which the trail pointer equals C[TR] (if C is the consumer choice point) and an empty value trail: indeed, nothing below C[TR] has been copied (and will never be). This initialization has been chosen for reasons of simplicity, not by necessity. When an incremental part is added to a chain, it is added at the end. Two descendants of one node in the tree necessarily have the same trail pointer value. The set $S(G)$ is actually a forest of such CHAT area trees: there is a tree for each $chat\_tr(C)$ with $C \in S(G)$; see Fig. 5. One can see three consumers with their choice points C_1, C_2, C_3. In the trail chunks, we have indicated the TR value (the numbers 3, 4, 5 and 8) and the reconstructed trail segments: either by `[]` (for indicating an empty segment) or a number of dots equal to the number of trail entries. This number equals the difference between the TR value of the (any) descendant and the TR value of the chunk itself. In Fig. 5, C_1 and C_2 share the part of the trail between 3 and 5. If backtracking now happens to a choice point with a chat_choice and with a TR value smaller than 3, say 1, then all three consumers will share the segment between 1 and 3.

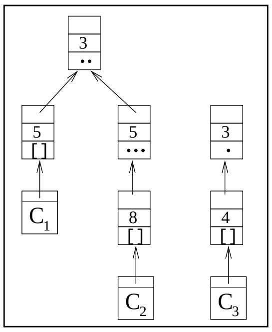


Fig. 5. A CHAT forest.

We can now refine the structure of $S(G)$: it is the list of roots in the forest of CHAT areas. This list is sorted in decreasing order of the values of their TR field. This ordering ensures that adding a new consumer to $S(G)$ is a constant time operation, since the new consumer has a TR root field that is larger than any element of $S(G)$. Also, the set operations in the code above become $O(1)$.

Finally, we deal with the set $S(G)$ on backtracking over G. If G is a leader, then all consumers that G was responsible for, have consumed all their answers: the CHAT area of these consumers can be released and the set $S(G)$ as well. On backtracking over a non-leader G, we merge $S(G)$ with $S(G')$ where G' is the generator immediately younger than G: indeed, $S(G')$ need not be empty !

The case of consumer below consumer Finally, we explain the only situation that is not described yet: suppose a consumer has been reinstalled and consumed an answer; execution continues and suppose a new consumer is encountered like in the execution of following query `?- p`$_g$`(Z).` and program:

```
:- table p/1.
p(Z) :- p_c1(X), p_c2(Y), Z is X + Y, Z < 3.
p(1).
```

Here, the second consumer has the same (nearest) generator as the first, but this is immaterial. As far as protecting the execution environment of `p`$_{c_2}$`(Y)` through extended CHAT freeze is concerned, C_1 can be temporarily treated as a Prolog choice point and the same code as in Section 4 can be used. Furthermore, the second consumer should share the part of the trail that the first consumer has in its CHAT area, i.e. the trail between C_1[TR] and G[TR]. This sharing is most naturally established on backtracking over the first consumer, i.e. on backtracking to the generator (note that there is never a choice point between a reinstalled consumer and its scheduling generator; cf. Fig. 2(c)). This is achieved by adapting the code for the answer_return instruction (the instruction through which consumers resolve against answers) so that after consumption of the last answer, the CHAT trail of the consumer is linked to all CHAT areas needing it.

6 Using the Trail Sharing on Reinstalling Consumers

We have shown how CHAT can share trail chunks between suspended consumers in the same way as in SLG-WAM. This is nice, but not enough: CHAT needs to also be able to exploit this trail sharing when restoring environments. Indeed, SLG-WAM will undo and reinstall the minimal set of bindings needed for moving from one node (the source) in the search tree to another (the target). However, note the following: Moving between two non-consumers is the normal Prolog backtracking and sharing between such execution nodes never exists. Similarly, moving from a consumer to a non-consumer is either usual backtracking after the consumer's creation, or backtracking on completion of a scheduling generator and thus does not involve sharing of trail either. Thus, sharing is possible *only* when the target node is a consumer; still the source node can be a consumer or not. We will show that in both cases, the same use of trail sharing as in SLG-WAM can be achieved by CHAT without added complexity.

6.1 Context switching from one consumer to another

This is the situation in which a generator G schedules one consumer after another. To achieve trail sharing in CHAT, generators should be aware of the fact that they have previously scheduled some consumer. The mechanism for this is easy: on scheduling for the first time a consumer, CHAT can simply replace the completion instruction by a new instruction, say next_completion, which takes into account that scheduling has happened before. I.e. backtracking to this G possibly means that a context switch between consumers is about to take place. An alternative is to let a consumer, upon finishing consumption of the current set of available answers, set a flag that the completion instruction tests. Furthermore, each generator should know which consumer it has previously scheduled: again a global variable (set by the consumer) can be used, or the scheduling generator can keep it in one of its slots, or we can simply rely on the fact that in CHAT the consumer choice point is immediately below the generator and find out its identity from there. Let a_1 denote the first trail area of the consumer which was already scheduled, and a_2 likewise for the next consumer to be scheduled. Then CHAT can use the shared trail — which takes the form of a tree, accessible from its leaves — in a way similar to how SLG-WAM finds the common part of the trail between two consumers. Detailed code to achieve this is given in Fig. 6. In this code, we have assumed that we can go up in the tree of CHAT trail areas with a function `up(`*area*`)` and that a function `TR(`*area*`)` gives the trail pointer in that area. The test `younger()` reflects the order in which CHAT trail chunks are allocated; the implementation has to cater for it either using its own memory allocator, or alternatively through timestamps. The code assumes there is a common ancestor and minor modifications cater for the case in which there isn't. The correctness of this code derives partly from the observation that two consumers never share their lowest trail segment.

```
tr1 := TR(a1); tr2 := TR(a2);
start := a2;
while (tr1 != tr2)
  { if (younger(tr1,tr2))
      { a1 := up(a1);
        untrail from tr1 to TR(a1);
        tr1 := TR(a1);
      }
    else
      { a2 := up(a2);
        tr2 := TR(a2);
      }
  }
while (start != a2)
  { reinstall_bindings_from(start);
    start := up(start);
  }
```

Fig. 6. CHAT code to context-switch.

In this way, the shared part of the trail is not de-installed and re-installed as in the original CHAT. Given the correspondence between the tree structure of the trail in CHAT and in SLG-WAM (cf. also Section 7), and by comparing this code to that given for the restore_bindings procedure in [11], it should be clear that the same use of sharing of trail parts is achieved as in SLG-WAM.

6.2 Context switching from a non-consumer to a consumer

This action happens on backtracking for the first time to a generator that has no more clauses to execute, so execution goes to the completion instruction which will schedule some consumer that waits for answers. The execution of the query `?- p_g(X).` against the following piece of code shows such a situation.

```
:- table p/1.                                    q(1).
p(X) :- comp, q(T), r(T,X).                      q(2).
r(1,X) :- p_c(Y), X is Y + 1, X =< 200.
r(2,100).
```

Suppose that the goal `comp` stands for a computation that left something on the trail but has no more choice points. The goal $\mathtt{p}_c$`(Y)` suspends. Then the goal `q(T)` backtracks to the second and last alternative and the generator gets its first answer (`X = 100`). Then backtracking occurs to the generator's chat_generator instruction, which will cause the addition of the trail part of `comp` to be added to the CHAT area of $\mathtt{p}_c$`(Y)`. Then, on failing back to the generator choice point, a consumer is scheduled: in this case, there is only one consumer and one could argue that it is clear that its installation does not need to undo and then reinstall the bindings represented by the trail of `comp`. In general however, there can be several consumers. If any of these just got an increment of trail (because the generator had a chat_generator) and is now scheduled, sharing is easy to get, on condition that the generator remembers which consumers' CHAT trails were just added to. It is fairly obvious how to do this and one gets again the same use of sharing as in SLG-WAM.

In general, it is of course possible that on backtracking to a generator, it has the completion instruction instead of a chat_generator. This means that no consumer will get the youngest trail increment, simply because the last alternative of the generator did not have any consumers. That means also that sharing is impossible both in CHAT and SLG-WAM.

Finally, note that giving up on using the sharing of trail when switching from a non-consumer to a consumer, does not increase the complexity of overall execution, as this happens for each generator only once and the extra cost is proportional to a trail chunk that was constructed before. This applies equally to both SLG-WAM and CHAT.

7 More about the Trail

In [5] we have already argued how the trail is different from the local stack and the heap: For the local stack and the heap, it is enough (for correctness) to maintain a safe approximation of their tops (any over estimation is safe). On the other hand, a WAM-based implementation must keep track *exactly* of which trail entries are between each two choice points. This is the deeper reason why the SLG-WAM uses back-pointers in its trail: the trail has a tree structure which mimics the tree structure of the choice points, but because choice points tend to disappear (by e.g. cut or trust) the trail has to maintain its tree structure independently. CHAT also uses back-pointers to link incremental CHAT trail areas: when each such trail area contains only one reference-value pair, the CHAT trail looks exactly like the backward-linked SLG-WAM trail.

SLG-WAM trail without back-pointers This similarity between the incremental CHAT trail areas and the SLG-WAM trail raises the question whether it is also possible to implement a trail in SLG-WAM *without* all the back-pointers. This is indeed possible: Let trail chunk mean a part of the trail that was created between the execution of two `try` instructions. Let the first `try` instruction save **TR** on the trail stack [4]. Subsequent trailing will consist in pushing only a reference-value pair. On the execution of the next `try` instruction, push the length of the trail chunk. It is easy to see that this organization still permits finding common ancestors.

Also worth noting is that the SLG-WAM is pessimistic in that the reference-value pair is always constructed, while CHAT postpones construction of a value trail chunk until the trail on the stack is about to be overwritten. It follows that trail chunks in CHAT tend to be larger and less back-pointers will be needed, because choice points can disappear before the trail chunk is constructed.

The concept of "nearest common ancestor" We take this opportunity to dwell on the SLG-WAM mechanism for finding the "nearest common ancestor". Even though intuitively, the nearest common ancestor is a choice point, one has to take into account that choice points (between a consumer and its generator) can die earlier than the consumer because of cut or the `trust` instruction. This means that the nearest common choice point is time-dependent. However, for the context switch from one consumer to another, one does not need the common choice point, but rather the point up to which in the trail one needs to untrail and from which to start reinstalling bindings. Moreover, the trail reflects the tree of choice points as well, and is (barring tidy trail at cut) time-independent. This leads to the SLG-WAM mechanism for finding the nearest common ancestor, based on the trail and on the back-pointers in the trail entries.

By using the choice points, a worse nearest common ancestor could be found in general: sub-optimal untrailing and reinstallation of bindings can result. Moreover, even if choice points do not disappear, finding the common ancestor in SLG-WAM by following the (frozen) choice points can be arbitrarily worse than by following the backward-linked trail entries and can affect the overall complexity. [5]. To show this, it is enough to construct an example in which there are two execution paths leading from a generator to two consumers. Each path has one trail operation and a number N of choicepoints. It will follow that finding the common ancestor choice point of the consumers is $O(N)$ and while finding the common ancestor trail point is $O(1)$.

8 Space Complexity of CHAT compared to SLG-WAM

So far, we have argued that CHAT is Θ(SLG-WAM) time-wise. We now compare the space complexity of CHAT and SLG-WAM. In principle, we should consider

[4] Note that the top of the trail is (in SLG-WAM) in general not equal to **TR**

[5] The reverse is not true because untrailing and reinstalling the bindings is obviously linear in the size of the traversed trail.

each of the four stacks separately because each of them can exhibit different behavior. However, note that space requirements of local stack and heap are identical for CHAT and SLG-WAM. They are both based on making these stacks non-recoverable on backtracking and this happens exactly at the same moment in execution for both CHAT and SLG-WAM. Space is retained in both systems until completion of the scheduling component. The fact that the freezing mechanisms of the two tabling implementation alternatives are different plays no role.

The following analysis assumes that SLG-WAM does not compact the stacks and that CHAT does not take advantage from performing selective completion. The reason is that with selective (i.e. non-stack based) completion it is easy for CHAT to release CHAT trail areas in constant time, while SLG-WAM can achieve the same space reclamation only by means of trail stack compaction which has a higher cost. We also do not consider the effects of tidying the trail on cut for reasons of simplicity.

Space Bounds for the Trail Stack As mentioned in Section 3.1, each trail entry of the SLG-WAM requires three cells. In CHAT a trail entry requires one cell while in the (WAM) stack and two cells when in a CHAT area. In the CHAT area, one additional cell *per chunk* is needed to "connect" a saved trail area to its next increment. The latter implies that the worst case for the CHAT trail area occurs when each trail chunk consists of exactly one trail entry: the CHAT trail area corresponds exactly to a forward trail entry in SLG-WAM. Since in CHAT a trail entry can at the same time also be in the active computation, the following equality holds in this worst case:

$$\mathit{trailsize}(\mathrm{CHAT}) = \frac{4}{3}\ \mathit{trailsize}(\mathrm{SLG\text{-}WAM})$$

The best case for CHAT occurs when there are no tabled subgoals, because then CHAT uses only one third the space, i.e.

$$\mathit{trailsize}(\mathrm{CHAT}) = \frac{1}{3}\ \mathit{trailsize}(\mathrm{SLG\text{-}WAM})$$

In summary, we have that:

$$\frac{1}{3} \leq \frac{\mathit{trailsize}(\mathrm{CHAT})}{\mathit{trailsize}(\mathrm{SLG\text{-}WAM})} \leq \frac{4}{3}$$

Space Bound for the Choice Point Stack Concerning choice point space, CHAT can perform worse than SLG-WAM only because at some points of execution a consumer choice point can be both at the stack and in the CHAT area. The only point at which this happens is when a consumer is resumed and as long as it remains resumed: indeed, as long as it is suspended, the consumer resides only in the CHAT area. Since Prolog choice points between the generator and the consumer tend to favor CHAT (see example below) the worst case cannot involve such Prolog choice points. Since each generator G can have at most one resumed consumer C at a time and since each resumed consumer C can only exist if there is a corresponding generator G, the worst case consists in a succession of (G_i, C_i) pairs where each G_i has scheduled C_i. It follows that the following inequality holds as a worst case for CHAT:

$$\frac{\mathit{choicepointsize}(\text{CHAT})}{\mathit{choicepointsize}(\text{SLG-WAM})} \leq \frac{3}{2}$$

This flatters slightly SLG-WAM, as generator choice points tend to be larger than consumer choice points.

On the other hand, CHAT wins from SLG-WAM arbitrarily when non-tabled choice points between the generator and consumer are popped on reinstalling the consumer. This can be seen in the program of Fig. 7: lots of Prolog choice points get trapped under a consumer; in CHAT, they can be reclaimed, while in SLG-WAM they are frozen and retained till completion. When called with

```
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).
```

Fig. 7. Tabled program where CHAT has better space complexity than SLG-WAM.

e.g. `?- query(29,97).` the maximal choice point usage of SLG-WAM is one generator, $29 * 97$ Prolog choice points plus 97 consumer choice points; while CHAT's maximal choice point usage is the generator, one consumer, 29 Prolog choice points and 97 consumer choice points in the CHAT areas.

9 Related Work

At least in the area of logic programming, "theoretical" implementation papers are not very common. Even at the level of the abstract machine specification, establishing correspondences between different implementation models and relating their properties is quite hard. As a result, the best that theoreticians have thus far achieved is either re-construct a particular abstract machine [8], or verify that the machine [1] or its compiler [10] execute according to the intended semantics of the language. On the other side, implementors are usually content with fully describing their abstract machines, arguing why a particular operation is performed faster in their model (usually at the expense of other operations), and/or comparing their implementation's performance against other systems on a "standard" set of benchmarks (see e.g. [13] and the references therein). To our knowledge, there is no result that a given abstract machine is optimal (i.e. uniformly better that the others): faithful comparisons between abstract machine designs most often show trade-offs. Moreover, worst-case performance of abstract machines is often not mentioned, sometimes not even known by the authors to exist, or all too easily dismissed as unlikely to occur in practice. While this might be true for average uses of an abstract machine, we believe it is important to know in advance how well the design scales and how the abstract machine's performance evolves under extreme circumstances.

There are of course notable exceptions that do prove complexity properties of abstract machines or of execution models. One nice such example is [7] which classifies various Or-parallel execution models based on primitive operations that need to be performed in this context (variable access, task creation and switching) and shows a very strong, negative result: no implementation model (with finite number of processors) can perform all three operations in constant time. While the complexity result in [7] is about a class of execution models, our result is more restricted, as it compares the characteristics of two specific abstract machines. This might account for our result being a positive one: namely that for the control of tabling, the same low cost of suspension/resumption can be obtained with environment sharing (as in SLG-WAM) or with a hybrid approach based on partial copying and partial sharing (as in CHAT). Consequently, the decision on which model to adopt can safely be based on other criteria, like e.g. performance overhead when tabling is not used or simplicity of implementation.

10 Concluding Remarks

We have shown how some adaptations of CHAT lead to an abstract machine for tabled evaluation that does not modify the WAM for non-tabled execution and still guarantees the same time and space complexity of tabled execution as SLG-WAM. Achieving this combination of strengths is by no means straightforward; in fact, for many years, implementors believed this combination was impossible to obtain and the incorporation of tabling into a Prolog system has often been ruled out as too expensive. On the other hand, we have reasons to believe that the complexity result for the modified CHAT described in this paper might be more of theoretical than practical interest because it seems that usually the impact of repeated traversal and sub-optimal trail sharing is very low. Indeed, in practice original CHAT has better performance than SLG-WAM (even on programs dominated by tabled execution), and the worst cases for CHAT have not been observed in real applications. However, as noted, we think that it is essential to know that the original CHAT design is not "just a hacked WAM-based model for implementing tabling that happens to *usually* work well" but that once faced with real programs that suffer from CHAT's worst-case complexity, there exists a "cure" in the form of a relatively simple addition.

Another important idea of independent and more practical interest in this paper is that of manipulating some Prolog choice points — in a way that is transparent to the underlying WAM — to (temporarily) assume a new functionality, which in this case partly overlaps with the functionality that the SLG-WAM reserved for generators. This technique leads naturally to new scheduling strategies of the "premature scheduling" kind, i.e. where scheduling is not only possible by (real) generators, but also by intermediate Prolog choice points that function like scheduling generators for some time. One of the added values of doing so, is that these scheduling strategies have even less context switching overhead (between execution environments of consumers) than strategies currently known. This is because scheduling decisions would then take place lower, or in any case more locally, in the execution tree. Another advantage is that scheduling of some consumers can naturally occur even before the associated generators finish program clause resolution. The performance benefits of doing so are yet to be explored.

Acknowledgements

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

References

1. E. Börger and D. Rosenzweig. The WAM — Definition and compiler correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, pages 20–90. Elsevier Science, North-Holland, 1995.
2. S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems — A case study. In *Proceedings of the ACM SIGPLAN PLDI*, pages 117–126, Philadelphia, May 1996. ACM Press.
3. B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming (PLILP'98)*, number 1490 in LNCS, pages 21–35, Sept. 1998. Springer.
4. B. Demoen and K. Sagonas. Memory Management for Prolog with Tabling. In *Proceedings of ISMM'98: ACM SIGPLAN International Symposium on Memory Management*, pages 97–106, Vancouver, B.C., Canada, Oct. 1998. ACM Press.
5. B. Demoen and K. Sagonas. CHAT: the Copy-Hybrid Approach to Tabling. In G. Gupta, editor, *Practical Aspects of Declarative Languages (PADL)*, number 1551 in LNCS, pages 106–121, San Antonio, Texas, Jan. 1999. Springer.
6. J. Freire, T. Swift, and D. S. Warren. Beyond depth-first strategies: Improving tabled logic programs through alternative scheduling. *JFLP*, 1998(3), Apr. 1998.
7. G. Gupta and B. Jayaraman. Analysis of Or-parallel execution models. *ACM Trans. Prog. Lang. Syst.*, 15(4):659–680, Sept. 1993.
8. P. Kursawe. How to invent a Prolog machine. *New Gen. Comp.*, 5(1):97–114, 1987.
9. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th CAV*, number 1254 in LNCS, pages 143–154, July 1997. Springer.
10. D. M. Russinoff. A verified Prolog compiler for the Warren Abstract Machine. *J. of Logic Program.*, 13(4):367–412, Aug. 1992.
11. K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Trans. Prog. Lang. Syst.*, 20(3):586–634, May 1998.
12. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD'94*, pages 442–453, May 1994. ACM.
13. P. Van Roy. 1983–1993: The wonder years of sequential Prolog implementation. *J. of Logic Program.*, 19/20:385–441, May/July 1994.
14. D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.
15. D. S. Warren. Efficient Prolog memory management for flexible control strategies. In *Proceedings of the 1984 Symp. Logic Program.*, pages 198–202, Feb. 1984. IEEE.