CHAT: the Copy-Hybrid Approach to Tabling

Bart Demoen and Konstantinos Sagonas

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

Abstract. The Copying Approach to Tabling, abbrv. CAT, is an alternative to SLG-WAM and based on total copying of the areas that the SLG-WAM freezes to preserve execution states of suspended computations. The disadvantage of CAT as pointed out in a previous paper is that in the worst case, CAT must copy so much that it becomes arbitrarily worse than the SLG-WAM. Remedies to this problem have been studied, but a completely satisfactory solution has not emerged. Here, a hybrid approach is presented: 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 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. Empirical results indicate that CHAT is a better choice for implementing the control of tabling than SLG-WAM or CAT. However, programs with arbitrarily worse behaviour exist.

1 Introduction

In [2], we developed a new approach to the implementation of the suspend/resume mechanism that tabling needs: CAT. The essential characteristic of the approach is that freezing of the stacks (as in SLG-WAM [4]) was replaced by copying the state of suspended computations. 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 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 [2], 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 behaviour of CAT has not shown up as a real problem in our uses of tabling (see [2] and the performance section of this paper), in [3] we have described a partial remedy for this situation. Restricted to the heap, it consists of performing a minor garbage collection while copying; that is, 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. [3] 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-optimised version of CAT suffers from the same worst case behaviour compared to SLG-WAM. Nevertheless, for most applications CAT is still a viable alternative 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 all sorts of hybrid methods are also 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 support for tabled evaluation should be possible without requiring difficult changes: CAT was inspired by this principle and provides such a design. CHAT, the hybrid CAT 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 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. What CHAT does with each of these four areas is described in Section 3 which is the main section of this paper. Section 4 shows best and worst cases for CHAT compared to the SLG-WAM. Section 5 discusses the combinations possible between CHAT, CAT and SLG-WAM. Section 7 shows the results of some empirical tests with our implementation of CHAT and Section 8 concludes.

2 Notation and Terminology

Due to space limitations we assume familiarity with the WAM (see e.g. [1, 6]), SLG-WAM [4] and CAT [2]. We also 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 the paper 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 (or more recent) means younger.

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 paper) 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 $\mathbf{B}$, these fields are denoted as $\mathbf{B}_T[\mathrm{ALT}]$, $\mathbf{B}_T[\mathrm{H}]$ and $\mathbf{B}_T[\mathrm{EB}]$ — T is either Generator, Consumer or Prolog choice point. The SLG-WAM uses four more registers for freezing the WAM stacks; however only two of them are relevant for this paper. We denote them by **HF** for freezing the heap, and **EF** for freezing the local stack.

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*; 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 [4] 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.

3 The Anatomy of CHAT

We describe the actions of CHAT by means of 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_0; then execution continued with some other non-tabled code (P and all choice points shown by dots in Figure 1). Eventually a consumer C was encountered and let us, without loss of generality, assume that G is its generator and G is not completed.[1] Thus, a *consumer choice point* is created for C; see Figure 1. The heap and the trail are shown segmented according to the values saved in the corresponding fields of choice points. The

[1] Otherwise, if G is completed, the whole issue is trivial as a *completed table optimization* can be performed and execution proceeds as in Prolog; see [4].

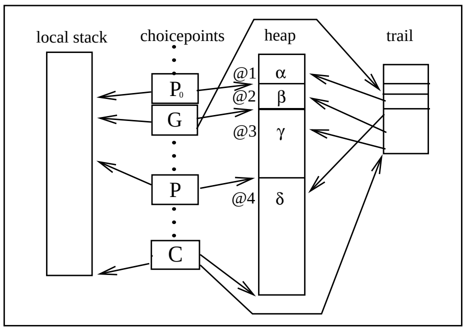


Fig. 1. CHAT stacks immediately upon laying down a consumer choice point.

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.

Without loss of generality, let us assume that C is the only consumer. The whole issue is how to preserve the execution environment of C. CAT does this very simply through (selectively and incrementally) copying all necessary information in a separately allocated memory area — see [2]. The SLG-WAM employs *freeze registers* and freezes the stacks at their current top; allocation of new information occurs below these freeze points — see [4]. We next describe what CHAT does.

3.1 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), $\mathbf{H}$ 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:

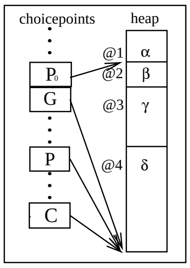
$\mathbf{B}_P[\mathrm{H}] := \mathbf{B}_C[\mathrm{H}]$

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 [4] on why this is correct).

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

for all choice points P between C and its generator (included)

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



The picture on the right shows which H fields of choice points are adapted by CHAT in our running example.
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 $\mathbf{HF} := \mathbf{B}_C[\mathrm{H}]$
on backtracking to a choice point P, the SLG-WAM resets **H** as follows:

if older($\mathbf{B}_P$[H],$\mathbf{HF}$) then $\mathbf{H} := \ \mathbf{HF}$ else $\mathbf{H} := \mathbf{B}_P$[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 between the consumer and the generator up to which it is performed. In principle this is unbounded, so the act of freezing in CHAT can be arbitrarily more costly than in SLG-WAM. However, our experience with CHAT is that this is not a problem in practice; see the experimental results of Section 7.

3.2 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 $\mathbf{EB} := \mathbf{B}$ while in CHAT this must become $\mathbf{EB} := \mathbf{B}[\mathrm{EB}]$.

As far as the complexity of this scheme of preserving environments is concerned, the same argument as in Section 3.1 for the heap applies. In the sequel we will refer to CHAT's technique of freezing a WAM stack without the use of freeze registers as *CHAT freeze*.

3.3 The choice point stack and the trail

CHAT borrows the mechanisms for dealing with the choice point stack and the trail from CAT: 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 a scheduling generator which is usually the leader of a scheduling component (see [2] 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 is similar to 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 consumer and the generator. This amounts to reconstructing the forward trail of the SLG-WAM (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. Figure 2 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.

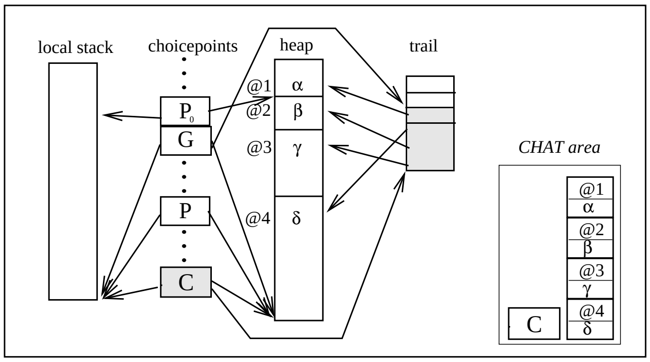


Fig. 2. Stacks and CHAT area after making a CHAT copy and adapting the choice points.

3.4 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 above. Any time a new consumer is encountered, the **B**[EB] and **B**[H] fields of choice points **B** 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. From now on, we will drop the assumption that there is only one consumer.

It is also worth considering explicitly a coup: a change of leaders. 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. For the trail, a similar mechanism as for CAT applies: an incremental part of the trail between the former and the new leader needs to be copied. In [2] 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 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. 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. Indeed, our implementation of CHAT employs both incremental copying of the trail and incremental adaptation of the choice points.

3.5 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 bindings. There remains the installation of the correct top of heap and local stack registers: 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. Figure 3 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

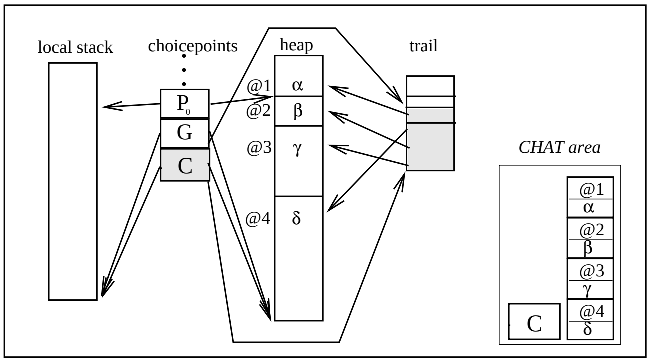


Fig. 3. Memory areas upon reinstalling the CHAT area for a single consumer C.

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 Figure 4; 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 answers through the answer_return instruction (see [4]). 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 $\mathbf{B}_C$;
save in $\mathbf{B}_C$[TR] the new top of trail stack;
alt := $\mathbf{B}_C$[ALT] := answer_return;
goto alt; */* transfer control to alt */*

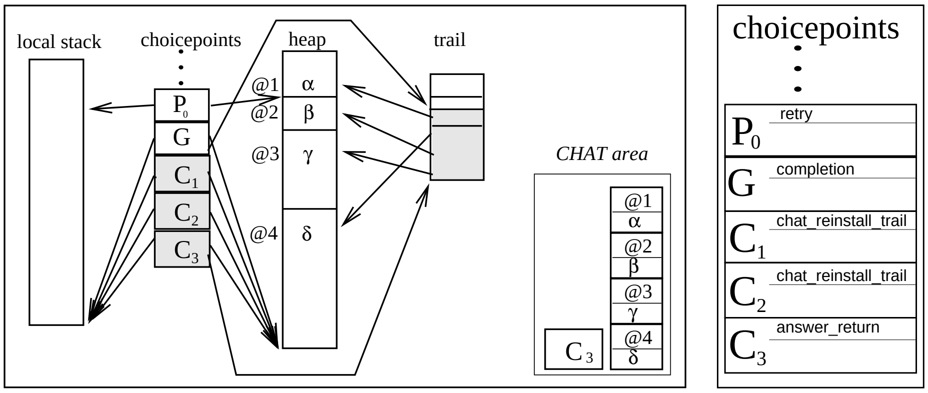


Fig. 4. Scheduling of three consumers C_1, C_2, C_3 and reinstallation of the trail of C_3.

3.6 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.[2] P_0 contains the correct top of local stack and heap in its EB and H fields: these 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 space reclamation at completion of a leader by resetting the freeze registers from the values saved in the leader choice point. Indeed, the SLG-WAM saves **HF, EF**, etc. in all generator choice points; see [4].

3.7 Handling of negation in CHAT

Support for well-founded negation (implemented in the current version of CHAT) is an issue orthogonal to the mechanism used for suspension/resumption. Due to space limitations we do not describe how DELAYING is handled as it is similar to its handling by the SLG-WAM (see [5]), but restrict attention to fixed-order stratified negation as in [4]. The SLG-WAM, upon encountering a negative literal of an incomplete subgoal lays a negation suspension frame in the choice point stack and the current execution path is suspended by freezing the WAM stacks (see [4]). 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 on the CP stack) and incrementally saving its trail entries as in the case of a suspended consumer. Upon derivation of an 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 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 3.5; notable differences are that 1) the negation suspension frames are not reinstalled below the choice point of the generator $\mathbf{B}_G$, but, since for fixed-order stratified negation the subgoal is completed, starting from that point and 2) the possibility to immediately reclaim each CHAT area upon reinstallation of its trail, as this reinstallation only happens once. The second point is valid even in the case of non-stratified negation.

4 Best and Worst Cases

As noted in [2], a worst case for CAT can be constructed by making CAT copy and reinstall arbitrary often, arbitrary large amounts of heap to (and from) the

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

CAT area. Since CHAT does not copy the heap, this same worst case does not apply. Still, CHAT can be made to behave arbitrarily worse than SLG-WAM. We also show an example in which the SLG-WAM uses arbitrary more space than CHAT.

4.1 The worst case for CHAT

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

1. every time a consumer is saved, the choice point stack between the consumer and the leader is traversed; such an action is clearly not present in SLG-WAM neither CAT
2. trail chunks are copied by CHAT for each save of a consumer; 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.

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) :-
    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).
```

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 $(Choices * Consumers)$ times more space and time than CAT.

At first sight, this seems to contradict the statement that CHAT is a better 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.

4.2 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:

```
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.

5 A Plethora of Implementations

After SLG-WAM and CAT, CHAT offers a third alternative for implementing the suspend/resume mechanism that tabled execution needs. It 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, 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; the next section elaborates on this.

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 possible 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 [2] 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

It might have been nice to explore all 36 possibilities, with two or more scheduling strategies and different sets of benchmarks but unlike cats, we do not have nine lives.

6 More Insight

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 [4]. 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.

7 Tests

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).[3] The size of the table space is not shown

[3] It is worth noting here that since conducting the performance tests we have found places in the XSB code where memory reclamation could be improved. Although

as it is independent of the suspension/resumption mechanism that is used. The benchmark set is exactly the same as in [2] 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 3.5, while the SLG-WAM scheduling algorithms scheduled all consumers in one pass and was thus invoked less frequently.

7.1 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).

	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 1. Time performance of SLG-WAM, CAT & CHAT under batched & local scheduling.

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 [2] for additional measurements on why this is so. Space-wise, CHAT wins always from 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 chuncks is compensated for by the lower choice point stack consumption.

this affects all implementation schemes, it probably does not affect them equally, so the space figures should be taken *cum grano salis*.

	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

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

7.2 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 CHAT (and CAT) have faster Prolog execution than SLG-WAM by around 10% according to the measurements of [4] — 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.

	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 3. Time Performance of SLG-WAM, CHAT & CAT.

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 from

	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

Table 4. Space Performance (in KBytes) of SLG-WAM, CHAT & CAT.

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.

8 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. CHAT also offers the same advantages as CAT as far as flexible scheduling strategies goes.

Acknowledgements

The research of the second author is supported by a junior scientist fellowship from the K.U. Leuven Research Council and CCR-9702681.

References

1. H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, Cambridge, Massachusetts, 1991. See also: http://www.isg.sfu.ca/~hak/documents/wam.html.
2. B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming, 10th International Symposium, PLILP'98, Held Jointly with the 6th International Conference, ALP'98*, number 1490 in LNCS, pages 21–35, Pisa, Italy, Sept. 1998. Springer.
3. 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.
4. K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3), May 1998.
5. K. Sagonas, T. Swift, and D. S. Warren. An Abstract Machine for Computing the Well-Founded Semantics. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 274–288, Bonn, Germany, Sept. 1996. The MIT Press.
6. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.